Knowledge Acquisition using an English-Based Method Editor

Jim Blythe and Surya Ramachandran
USC, Information Sciences Institute
4676 Admiralty Way,
Marina del Rey. CA 90292.
*{blythe,*rama}@isi.edu

Abstract

We describe an editor for problem-solving knowledge that communicates with the user through English paraphrases of the knowledge. Although it does not support the full range of modifications one might want to make, the value of the tool lies in the fact that the user need not understand the syntax of the expert system to make modifications. By analyzing the problem-solving knowledge, the tool can allow the user to select semantically coherent chunks of the knowledge. It then presents English paraphrases of possible substitutions which would result in new problem-solving knowledge that is syntactically correct. In this way the tool expands the range of modifications that a naïve user can make to problem-solving knowledge in an expert system.

Introduction

The ability to change the contents of the knowledge base without knowing the representation language and with just a basic understanding of the domain is one of the ultimate goals of any knowledge acquisition tool. (Simon 86) states the need for natural language interfaces in a very compelling way:

> "[...] continue to move forward with the natural language understanding capabilities of the computer part of our system. [...] problems are going to be stated initially by human beings in natural language. Unless we face up to that fact, we are going to foreclose forever the computer doing a very large part of our job [...]"

Research in natural language processing has made significant progress in the area of information extraction from text (Cardie 97). In the area of knowledge acquisition, developing knowledge bases from textual input has been investigated in (Goel 96) and (Hahn 96). From unstructured English text, automated tools are able to extract facts, conceptual relations, and even complex events. Other research in natural language has looked at the structure of natural language dialogues between a person and a KA tool with some interesting findings regarding its discourse structure (LuperFoy 95). Other related knowledge acquisition research is in the development of tools are those that use easy-to-use paradigms that are intuitive for naive users (Gaines 93). Making all of our KA tools and approaches communicate with users in a language like English that they already are familiar with may result in a much more widespread use of AI technology and in particular knowledge based systems. Often times, unconstrained English will be a challenge as an interface to our systems, given that at least today they do not deal well with

properties inherent to natural language such as ambiguity, ellipse, and anaphora. But it is feasible with today's technology to develop KA interfaces that use a very structured and restricted subset of English without users noticing, giving them the illusion that they are communicating in natural language. This is one of the aims of our research in trying to make KA tools more accessible to end-users.

This paper describes an english-based method editor that can be used to make modifications to knowledge bases developed using the EXPECT architecture (Swartout and Gil 95; Gil 94; Gil and Paris 94). The tool makes use of the strict typing of the representation language and the grammar that EXPECT follows in order to create an English-based front end that the expert and novice domain expert can use to modify knowledge bases. EXPECT's grammar is based on previous work on the Explainable Expert System (EES) project (Swartout 81), which was a framework for building knowledge-based systems that could provide good explanations of their behavior.

Simply, the tool first converts problem solving knowledge into English-like structures. Then these structures are displayed in a window where the user can select atomic or multiple chunks of text. Then the interface provides the user with a set of alternatives (also parsed in English) that can be substituted in the method. The user may select one of the provided alternatives and commit the change. The important points are that the user can modify such structures without ever having to refer to the underlying semantics or language that the knowledge is represented in (source code) and that the result is grammatically correct since the alternatives are provided so as to preserve the syntax.

```
((name calculate-time-to-transport-in-ship)
 (capability (calculate-time-to-transport
                 (obj  (?cargo is (inst-of weight-value)))
                 (in (?ship is (inst-of ship)))
                 (from (?origin is (inst-of location)))
                 (to   (?dest is (inst-of location)))))
 (result-type (inst-of time-value))
 (method-body
   (value-divide
         (obj (find (obj (spec-of distance))
                    (from ?origin)
                    (to ?dest)))
         (by (r-speed ?ship)))))
```

Figure 1: An example of an EXPECT Problem Solving Method.

A simple example motivates our approach. Figure 1 shows an EXPECT method, a chunk of problem-solving knowledge that computes the time to transport an item in a ship, by dividing the distance to travel by the speed of the ship. A domain expert may wish to modify this piece of knowledge in a number of ways, for instance to make sure the distance is computed along waterways or to modify the speed computation to take the weight of the payload into account. These are relatively simple modifications that we would like the expert to be able to make directly, but the syntax of the method is daunting. Figure 2 shows the same method being edited with the English-based front end. In the second window is a sentence representing the body of

the method: “find the distance from the first location to the second location and divide the distance by the speed of the ship”. This is understandable with no knowledge of the underlying syntax or the domain terms chosen by the knowledge engineer. The user can alter the method by selecting a part of the sentence and choosing from a set of provided alternatives for that part (shown in the second window from the bottom). The selected part can be a single atomic unit like “speed” or a meaningful fragment like “the speed of the ship”. In Figure 2, the noun phrase “the speed of the ship” was selected and it is being replaced with the sub-task “find the speed of the ship with the weight value”. The English-based front end uses a parse tree of the method to ensure that all selectable fragments of text correspond to coherent fragments of code in EXPECT. It provides a textual description of each alternative for the selected sentence fragment and ensures that each one corresponds to a code fragment that has the same role as the selected fragment, so that after the change the new method description is still grammatical EXPECT code.

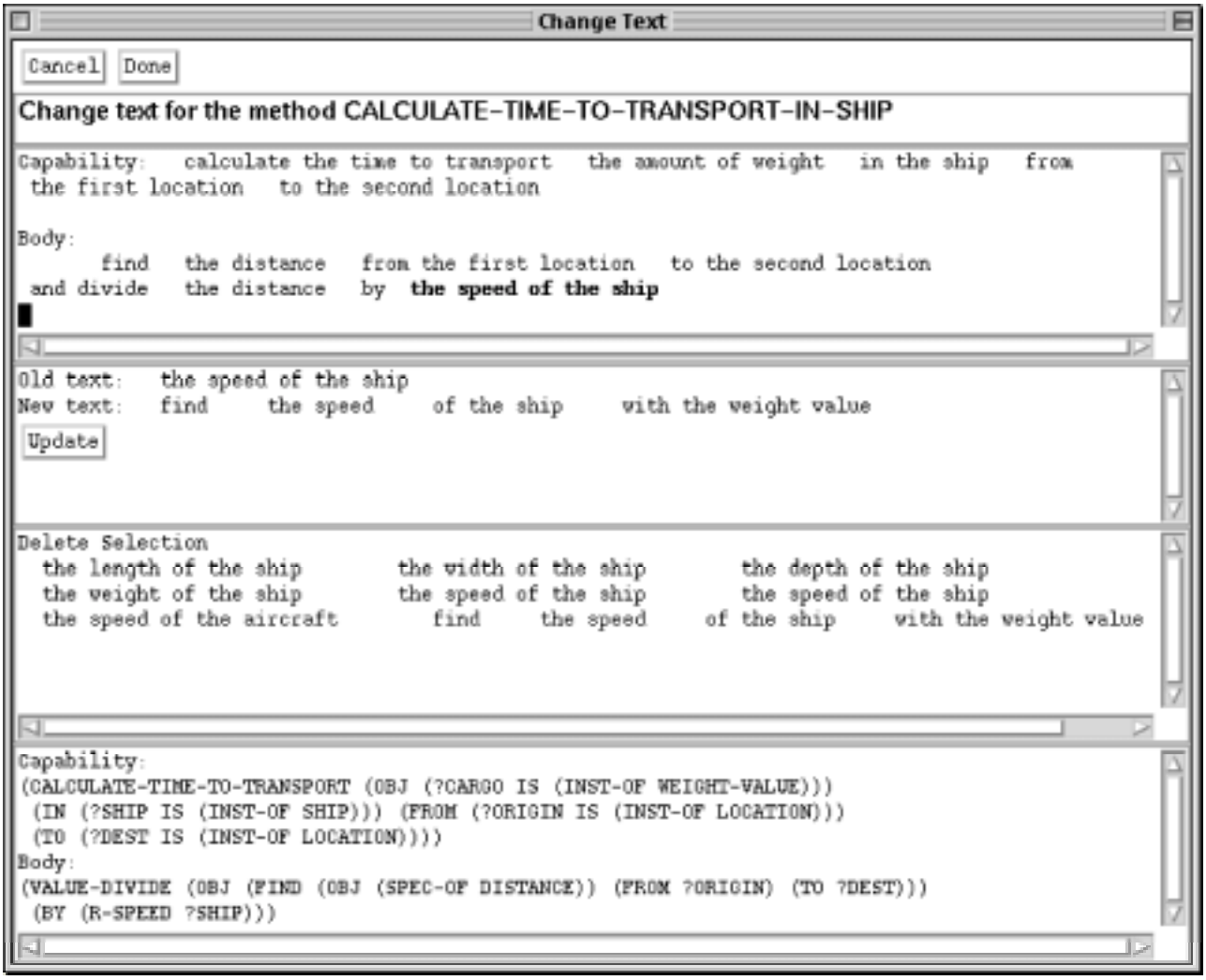

Figure 2: The English-based method editor.

In this paper we describe how the English-based editor is implemented and discuss some of the advantages and limitations of the approach. In the next section we give a brief overview of the EXPECT system and our approach to knowledge acquisition. Then we talk about the motivation behind the development of such a tool. Next we give a detailed description of our approach to generating fragments of English text and using them to interact with a user to modify problem-solving knowledge. Finally we discuss some of the issues raised by this work and future directions.

The EXPECT Framework for Knowledge Acquisition

EXPECT (Swartout and Gil 95; Gil 94; Gil and Paris 94) is a flexible KA tool that has been used for a variety of tasks and domains including configuration (Gil and Melz 96) and planning (Valente et al. 96). EXPECT provides an environment for building and maintaining knowledge-based systems that are accessible to end-users. This requires capabilities such as explanation of the system's behavior and support for knowledge-base construction and maintenance. EXPECT builds on previous research on the Explainable Expert System (EES) project (Swartout 81), which was a framework for building knowledge-based systems that could provide good explanations of their behavior. In EXPECT, any information necessary to perform a task is represented distinctly according to its nature either as domain facts or as problem-solving knowledge. Domain facts are represented in LOOM (MacGregor 88 and MacGregor 94), a knowledge representation system of the KL-ONE family (Woods and Schmolze 92). LOOM provides a descriptive logic representation language and includes a classifier for inference.

Problem-solving knowledge is expressed as EXPECT methods (figure 1). A problem solving method in EXPECT is an abstract and generic description of how a goal can be achieved and includes the goal, a method body that describes the procedure to achieve that goal, and the result that the method is expected to return. Methods decompose higher-level abstract goals into more detailed ones. EXPECT's problem-solving representation language is tightly coupled with the LOOM representation. This provides a good basis for relating goals and domain knowledge and a better understanding of what the different types of knowledge contribute to the solution of the task.

In order to ensure coherence among the various types of knowledge, the problem solver uses the factual and problem-solving knowledge sources to perform a static analysis of a given problem, recording how each piece of knowledge contributes to the problem-solving process. EXPECT employs the reasoning capabilities of LOOM augmented with goal refinement and reformulation. EXPECT’s analysis is effectively a partial evaluation of the given problem. Goals are represented as LOOM concepts, and plan-goal matching is based on the LOOM classifier. If no method is found to achieve a posted goal, the goal is reformulated using the factual domain knowledge into a set of sub-goals that can be achieved. This analysis provides the knowledge acquisition tool with an understanding of both the functionality and the nature of all the information used for the task.

Whenever the problem solver cannot achieve a goal, the system takes this as an indication that the knowledge that the system currently possesses may be insufficient. Instead of terminating problem solving and reporting a failure, it notifies the knowledge acquisition module that will analyze the problem and determine whether there is a need to request the user's intervention. The

problem solver provides detailed information to the knowledge acquisition tool that is crucial to support the user in correcting the problem.

The facility to relate all the different types of knowledge in the system and capture their influence in the system's behavior enables EXPECT's knowledge acquisition tool to support the user in changing the system's knowledge. The problem-solving strategy is represented explicitly so that the knowledge acquisition module reasons about it and dynamically derives the knowledge roles that must be filled out, as well as any other information needed for problem solving.

To allow the user to make modifications by changing some of the steps or perhaps adding a new method using analogy, the method editor should be able to translate what the methods mean and what are the alternatives that it can provide and suggest to the user.

An English-based Editor for EXPECT

Although EXPECT's KA tools and approach are powerful, it is still hard for naïve users not familiar with logic. The philosophy behind the English-based editor is that, although the kind of knowledge that a good programmer possesses may be essential to perform the full range of modifications that need to be made during the lifetime of a KB, a significant and useful set of modifications are possible without this knowledge. We are exploring ways to bring these capabilities to end users.

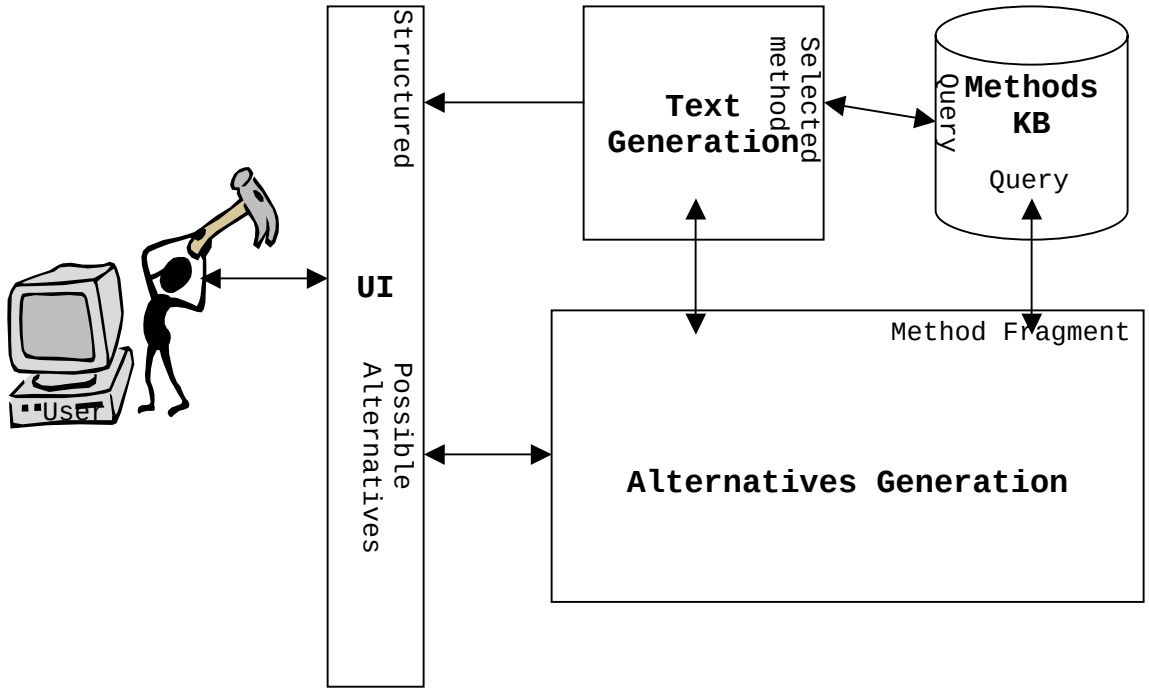

Figure 3: System Architecture

Two main steps are involved in our approach. First, the problem solving knowledge is converted into an English-like structured text fragment. Second, selectable parts of the text are modified by

choosing among alternatives that are also presented via an English paraphrase. This allows the user to make useful changes without directly coming into contact with the underlying syntax.

The English text-based method editor for EXPECT consists of three modules as shown in Figure 3. The Parser module produces readable text fragments to describe methods and alternative method fragments to users. The Alternatives Generation module generates a list of likely alternatives for any fragment of a method that might be selected by the user. The User Interface module makes use of the lower-level capabilities of the other modules to present the user with an English description of a method in which different pieces can be selected for alteration, with similar descriptions of alternative strings for any part of the paraphrase that is selected.

Before we describe each module, we describe how problem-solving knowledge is represented in EXPECT. The problem-solving knowledge represents the actions that can be performed over all the objects in the domain. A problem solver tries to find a way to combine the available methods that together achieve some goal given by the user. In EXPECT, the methods in a domain are defined in a hierarchical fashion. They are defined independently from the higher-level tasks that use them. They can have different levels of detail, ranging from very abstract generic methods to very specific ones.

In EXPECT, the methods in a domain are defined in a hierarchical fashion. They are defined independently from the higher-level tasks that use them. They can have different levels of detail, ranging from very abstract generic methods to very specific ones. At the lowest level are primitive methods, which cannot be decomposed any further and can be executed directly or translate into user defined actions. Our language for actions tries to represent the intent of each action, i.e., the goal that it can achieve.

As seen in Figure 1, EXPECT's problem solving methods have three main parts: the *capability* section of the method that specifies what the method can achieve, the *method body* that describes how the action is decomposed into lower-level actions and the *result type* that specifies what EDT the action returns. In order to build a method editor it is important that this hierarchical decomposition of methods is incorporated not only at the method level but also in the specific parts of the method itself. If we use a flat text generation scheme, the resulting text fragment would lose the hierarchical nature of its representation and with it vital information on the structure of the method itself. A flat representation would be adequate to select a single word that represents some atomic element of the code. But in order to select different sections of the method through their corresponding English strings, there has to be some association of the method structure and English that is generated from it.

Text Generation

The method body and capability are each decomposed into a parse tree, where each node represents a piece of the method (see figure 4). Associated with each node is the english text that it generates, and all its children and the english text that they generate.

Each node has four fields: the English form, the EXPECT code that generates this english form, a unique index number and a list of the node's children.

All structures in EXPECT have (or return) a value that has an EXPECT Data Type (EDT), an object type as shown in Table 1. This greatly facilitates the ability of the interface to derive the type for a given piece of code. This in turn allows for efficient alternatives generation (as described in later sections). The method structure has clearly delineated sections and code changes to one can be easily mapped to changes in other sections of the method (e.g. when we

change the variable reference in the capability section of a method, say from "ship" to "aircraft", then the associated variable in the body, say "?ship1" now represents an instance of an airport).

Type	**Example**	**Notation**
a specific instance	the city of New York	NY
a concept	the concept location	(spec-of location)
an instance of a concept	any seaport	(inst-of seaport)
a set of instances	a set of numbers	(set-of (inst-of number))
a set of specific instances	1, 2, 3, 4	(1 2 3 4)
a set of specific concepts	integer, real, fraction	(integer real fraction)
a set of concepts	a set of types of numbers	(set-of (spec-of number))

Table 1: EXPECT Data Types (EDTs)

Consider a part of a problem solving method that determines the time taken for a ship to travel a given distance:

```
(VALUE-DIVIDE (OBJ ?DISTANCE1) (BY (R-SPEED ?SHIP1)))
```

The resulting tree structure is shown in Figure 4.

The English paraphrase is generated from the first element of every node in depth-first order as follows:

```
" divide  the distance by the speed of the ship"
"-      --   -        -  -           -   -   "
"1      3            6  8                   "
```

where we have added the second and third lines in this paper to highlight the extra spaces in the string and their meaning. The second string has dashes to mark blank spaces and the third string shows the corresponding nodes in the parse tree that allow the user to select a chunk of text instead of an atomic unit.

The user may want to change only a piece of this code, which may be a single construct/atomic-unit or some contiguous sequence of them. The above decomposition helps capture user's intent more precisely. Blank spaces between the words represent one of three distinct features. First are

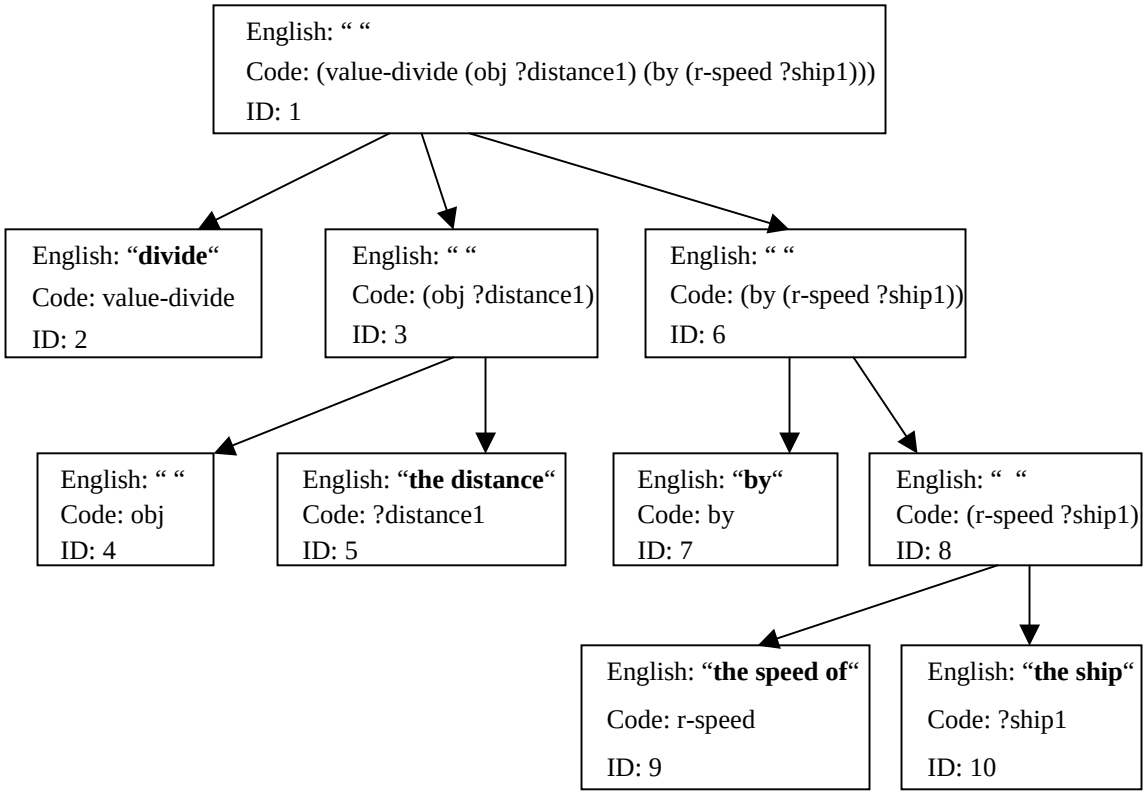

Figure 4: Tree structure for an EXPECT method body fragment.

the natural blank spaces between words. Second, some of the code does not have English translation (eg: OBJ does not translate into anything meaningful that can be put in its place. Third, some blank spaces represent a nesting of sequential elements. So if it’s the user's intent to change (by (r-speed ?ship1)) or (r-speed ?ship1) as a whole unit, because of the hierarchical form of the parsed code, there will always be a blank that denotes every contiguous chunking of nested code. For example the annotated string (with the blank spaces selectively numbered) blank 1 will (when the mouse pointer is brought over it) cause the entire code segment to be selected for replacement. Similarly 3, 6, and 8 represent different chunks of code that can be replaces, identified by the index on the appropriate node.

Another observation is that there is a conversion to “the ship" from "?ship1”. In creating the tree structure an association list is built with the variables and their associated types. The associated EXPECT Data Type (EDT) for the variable is obtained from the capability section which will have “(?ship1 is (inst-of ship))”. So this association list (or a part of it) would look like : (... (?ship1 (inst-of ship)) (?origin (inst-of location)) ....). This is then used whenever a variable is encountered to generate a more descriptive translation of the variable.

When two variables share the same type, as in the code fragment from Figure 1:

```
"(find (obj (spec-of distance)) (from ?origin) (to ?dest))"
```

the order in which the variables are introduced is used to distinguish them in the code. Thus, this fragment is translated into:

“find the distance from the **first** location to the **second** location”.

Finally, simple transformations such as changing “value divide” to “divide” are made through post-editing rules supplied with the KB.

User Interface

The User Interface module is currently written in CLIM. The interface shows the user three sub-windows (figure 2). The first shows the English version of the method currently being edited. The second shows two fields, "old text" and "new text" and the third displays a list of alternatives for the selection in "old text", whenever there is one.

All entries in the method window and the alternatives window can be selected by the user to fill either the "old text" or "new text" fields. To choose a piece of the method to modify, either from the capability or the body, the user can select it to fill "old text". The method display makes use of the tree structure described in the previous section so the user can click on any coherent chunk of English of the parsed method.

For example if a sub-goal in the body is:

```
"divide the distance by the speed of the ship"
```

the user could click on the whole sub-goal, or "the speed of the ship", or just "the ship" or "divide".
In this case, the user has selected the chunk of text corresponding to the “speed of the ship” and has chosen to replace it with an alternate chunk of text that is a method in itself to “find the speed of the ship with respect to the load it’s carrying”.

When a selection is made for the "old text" field, the tool fills the alternatives window with possible alternatives to be selected for the "new text" field. The alternatives all belong to the same classification in the grammar, e.g. relation name, parameter name, or something that can be expanded to be an instance or concept. This helps the user make changes that are grammatical. The next section discusses how the alternatives are chosen, and how they are ordered in the window.

Once the user selects a value for the "new text" field, they can select "update" in the middle window to make the change in a copy of the method. When this is done, the method window changes to show an English language version of the modified method. However, the method is not changed in EXPECT knowledge base until the user selects "done".

Generation of Alternatives

This section describes what happens when the user selects a piece of text, replaces it with another one and then commits one or more of these changes.

Alternatives generation deals with the user's intent over selected text to be replaced. The user selects some text from the English paraphrase of the method capability or the method body. The system displays (in English) a list of ordered, relevant possible replacements in the alternate window. The basis for alternatives generation is the fact that every chunk of text that can be selected through the interface is linked to a node in the parse tree that was generated. This node in turn links to a fragment of EXPECT code, which has an EXPECT data type associated with it. The system identifies relevant replacements by exploring a neighborhood of the selected code fragment in the space defined by EXPECT's method grammar. The ordering of the list of alternatives reflects the system's guess at what the user is most likely to select. Once the selection of the replacement text has been made, the user can refresh, or see the changes in the original code by the use of the "update" button. This process of selection and updating continues until the user is satisfied with the new method. At this point the user can either commit these changes to the knowledge base or cancel out of the operation.

The system generates alternatives when the user selects a string to be replaced. This is governed by what piece of code is chosen (ie: variable, relation, sub-goal, etc.) and the type (EDT) of the code chosen (the return type) which is established by the type hierarchy (ie: The method body has a return type based on the result field). At the end of the replacements, the code is committed to the EXPECT knowledge base.

For any selection made by the user, the system will generate a set of alternatives that constitute a valid replacement, i.e. that fill the same role in the EXPECT grammar. The list for the selection of alternatives is:

i) **Concept and instance descriptions**. (e.g. (inst-of number)) We retrieve all sibling concepts, which will be displayed in the alternatives window. We also retrieve all relations and possible variable assignments whose results have the same data type. Lastly we retrieve all methods that have a return data type similar to the concept or instance descriptions.

ii) **Relation names**. (e.g. r-speed) Currently, the alternatives are made up of all the relations that are defined on the same EDT as the one selected. The list of variables of the same data type and all methods that have a similar return data type.

iii) **Goal/Sub-goal names**. (e.g. `FIND, FILTER, COMPUTE`) The system generates alternatives based on similar goal names (found by looking at all the sub-concepts of expect actions) are displayed, all concepts and relations and possible variable assignments with the same data type.

iv) **Goal parameters**. (e.g. `WITH, FROM, ON`) All known parameters are displayed, since there is just a small number of them.

v) **Sub-goal expressions**. (e.g. `(FIND (OBJ (SPEC-OF DISTANCE)) (FROM ?ORIGIN) (TO ?DEST)))` . The list of alternatives are ordered as the following; variables that match the return-EDT of the goal, relations that have the same EDT, and other goals that have the same return/result type.

vi) **Variables**. (e.g. `?PORT1, ?PORT2, ?S)` The list of alternatives are ordered as follows: similar variables in the plan (with the same EDT), other variables in the plan, relations based on the EDT of the code selected, and goals that have the same return type (EDT) as the selected code.

vii) **Instances**. e.g.: constants, etc. The list of alternatives are ordered as the following; similar instances that may be part of the method, other instances from the domain (matching EDT), variables from the domain (matching EDT), relations with the same EDT, and goals that have the same return type.

Conclusions

The contributions of this work can be summarized as follows. The user interface provides an easy way for novice as well as expert domain experts to modify and add knowledge to existing knowledge bases. The English paraphrasing enables the user to concentrate more on the knowledge that is to be added than on its representation and syntax in the knowledge base. By restricting the user to choose from alternatives that are predetermined to be syntactically correct, the interface ensures that the result of modification does not lead to syntactic inconsistencies. The interface is domain independent and can be used over any EXPECT knowledge base without any change to it.

As knowledge bases grow large, the set of all relevant alternatives can become hard to manage in a simple list, so we are looking at ways the user can provide information to narrow the range of alternatives. The simplest is to allow search, as tools like Ontosaurus (Swartout et. al, 96) do, so the user can for example type "ports" and see every alternative that mentions the string "ports", rather than have to scroll through the unfiltered list.

We are also working to improve the tool's capabilities and functionality beyond simple editing to allow the creation of new problem solving methods from scratch. Our approach combines the English parsing ability with adaptive forms (Frank and Szekely 98), which have been shown to be a powerful general method for entering data with restricted grammars. We are also planning to integrate this editor with other KA tools that have been developed for EXPECT.

Acknowledgements

We gratefully acknowledge the support of DARPA with contract DABT63-95-C-0059 as part of the DARPA/Rome Laboratory Planning Initiative, and with grant F30602-97-1-0195 as part of the DARPA High Performance Knowledge Bases Program.

References

Cardie, C., Empirical Methods in Information Extraction, AI Magazine, pp 65-79, Winter 1997

Frank M. R. and Szekely, P., Adaptive Forms: An interaction paradigm for entering structured data. In Proc. of the ACM International Conference on Intelligent User Interfaces, pp 153-160, 1998.

Gil, Y. Knowledge refinement in a reflective architecture. In Proceedings of the 12th National Conference on Artificial Intelligence, Seattle, WA, 1994.

Gil, Y., and Paris, C. Towards method-independent knowledge acquisition. Knowledge Acquisition, Vol. 6 (2) pp :163--178, 1994.

Gil, Y. and Melz, E. Explicit Representations of Problem-Solving Methods for Knowledge Acquisition. In Proc. of the National Conference on Artificial Intelligence (AAAI-96), 1996.

Goel, A., Mahesh, K., Peterson, J., Eiselt, K., Unification of Language Understanding, Device Comprehension and Knowledge Acquisition. In Proceedings of the 10th Knowledge Acquisition for Knowledge-Based Systems Workshop, 1996

Hahn, U., Klenner, M., Schnattinger, K. Automated Knowledge Acquisition Meets Metareasoning: Incremental Quality Assessment of Concept Hypotheses During Text Understanding, In Proceedings of the 10th Knowledge Acquisition for Knowledge-Based Systems Workshop, 1996

LuperFoy, S. Implementing File Change Semantics for Spoken-Language Dialogue Managers, in Proc of the Spoken Dialogue System Workshop of the European Spoken Communication Association, Vingsoe, Denmark, 1995

MacGregor, R. A Deductive Pattern Matcher. In Proceedings of the Seventh National Conference on Artificial Intelligence (AAAI-88). St. Paul, MN, August 1988.

MacGregor, R. A Description Classifier for the Predicate Calculus. In Proceedings of the Twelfth National Conference on Artificial Intelligence, (AAAI-94), pp. 213-220, 1994.

Schmidt, G. and Wetter T., Towards knowledge acquisition in natural language dialogue. In proceedings of the European Knowledge Acquisition Workshop. pp 239-252. 1989.

Swartout, W. R., Explaining and Justifying Expert Consulting Programs. In proceedings of the 7th International Joint Conference on Artificial Intelligence. 1981.

Swartout, W. R., and Gil, Y. EXPECT: Explicit Representations for Flexible Acquisition. In Proc. of the 9th Knowledge Acquisition for Knowledge-Based Systems Workshop, Banff, Alberta, 1995.

Swartout, W. R., Patil, R., Knight, K. and Russ, T. Toward Distributed Use of Large-Scale Ontologies. Proc. of 10th Knowledge Acquisition for Knowledge-Based Systems Workshop, 1996.

Valente, A., Gil, Y. and Swartout, W. R. INSPECT: an Intelligent System for Air Campaign Plan Evaluation based on EXPECT. ISI Technical memo, June 1996. Available on the Web at: the URL: http://www.isi.edu/~valente/inspect/inspect.html

Woods, W. and Schmolze J., The KL-ONE family. Computers and Mathematics with Applications. 23(2-5). pp 133-177. 1992.