The Space Congress® Proceedings

1988 (25th) Heritage - Dedication - Vision

Apr 1st, 8:00 AM

Empirical Results in Automatic Software Documentation/ Explanation and a Plan to Increase Its Tractability

Kent W. Machovec

Lockheed Software Technology Center, 96-01,30E 2100 East St. Elmo Road, Austin, Texas 78744-1016

512/448-9733

Empirical Results in Automatic Software Documentation/Explanation and a Plan to Increase Its Tractability

Kent W. Machovec

Lockheed Software Technology Center, 96-01, 30E
2100 East St. Elmo Road, Austin, Texas 78744-1016
512/448-9733

A good notation is more than mere "convenience," for it also allows us to structure our ideas hierarchically: we can focus our attention at the appropriate level.
—J.E. Stoy, *Denotational Semantics*, 1977.

ABSTRACT

Documentation is an important aspect of software, and issues of project personnel turnover, contractual obligations, and so forth all represent significant concerns. Documentation, in essence, provides *understanding*. We report on the result of an experiment in automatic software documentation/explanation in which we input a program (written in REFINE™) and output English describing the functionality of the code. Perceptions on the nature of the documentation/explanation task are presented. Significant issues in the design of such systems (e.g., intended audience, source language, abstraction, concept recognition, etc.) are identified. The characterization of this area as "nontrivial" leads to a research plan to broaden the source of solutions to related areas: machine translation, decompilation, question answering systems, pattern (image) recognition, knowledge representations developed for automatic programming, empirical analysis of software modification strategies, and other topics. Potential applications of this area of research are projected, among them: software reuse, intelligent debugging assistant, software porting aids, maintenance assistant, software restructuring.

Keywords: automatic software documentation, automatic software explanation, program understanding, pattern recognition, VHLL, software translation.

1.0 INTRODUCTION

The development of current interactive software environments (Barstow 1984a) parallels the impetus to the research and subsequent development of data base management systems (DBMSs). The following excerpt from *The Development of Data-Base Technology* (Sibley 1976, p.3) illustrates the problem of shared access to common data:

> In time, astute data processing managers recognized a problem: while stored within the data vaults of the organization . . . data was essentially unavailable. The programs that input, stored, and used the data were essentially the owners of the data. Any other user found it difficult to obtain, integrate, or transform the "available" data for use in another program. Thus, every new need for data involved writing a new program to obtain the data before it could be processed by yet another program, and even this was difficult—the data formats were 'locked' in the original programs, and sometimes the original [source] code had been lost!

Current software knowledge-base management systems (such as Topping 1987 and Kotik 1986) share the following objectives with DBMS technology:

1. Common data accessible to multiple "tools" (compilers, *automatic documentation tools*, on-line manual browsing (Walker 1988), intelligent software project management tools (Bimson 1988)
2. Persistent object systems (do away with the file system) (PersObjWrksp 1987)
3. Integration (rather than ad hoc combination of tools (Huseth 1987)

4. Reduction/elimination of redundancy (of both data and programs written to access the data)
5. Concurrent access (Larson 1987).

The Express project (McInroy 1988) represents Lockheed's commitment to a revolutionary software development approach. The work reported here is one facet of the solution being pursued by the Express software development system.

Documentation is an important aspect of software, and issues of project personnel turnover, contractual obligations, and so forth, all represent significant concerns. As such, the scientific community is directly addressing this area with research in tools and techniques which facilitate automatic software documentation/explanation.

Section 2 gives the relevant results from the translation experiment—including both general observations as well as an identification of significant issues in the overall problem of automatic software documentation/explanation (AUTODOC). Section 3 reports on the actual experiment performed and shows the relevance of the criteria (established in section 2) in the context of the specific experiment. Section 4 attempts to expand the set of useful constraints on the AUTODOC problem by identifying other areas of research whose objectives and methods are synergistic. Section 5 describes potential applications of research in automatic documentation/explanation. Section 6 presents the conclusions of the article.

2.0 EXPERIMENTAL RESULTS

General Observations

Automatic documentation is the *inverse* of the process of users communicating their desired computation to the machine. Note that this has had various names: coding, programming, specifying, specification acquisition, direct manipulation, graphical programming, and so on. And not all techniques use the same notational methods.

The observation that some of the checking done by the translator was identical to checks performed by the source compiler led to the idea of attempting to leverage off the source language compiler as much as possible. In REFINE, the following constructs have identical syntactic form: a function call with one argument, an object class membership test ("object" in the sense of object-oriented programming), and an attribute reference. To determine what was actually the case it was necessary to check more of the way it was used in the program context. While this isn't surprising (by any means), there *is* a better way to do this, which would do two things: (1) allow the translator to work at a higher level of abstraction and (2) make the translator impervious to (superficial) changes in the syntax of the client's source language. A solution would be to define some sort of "cooperative protocol" between the source compiler and the translator whereby the "compiler" (in this usage, the term *compiler* is misleading) would be receptive to queries from the translator. For example, the "compiler" would take a subexpression from the translator and determine if it recognized the abstract "concept" of creating a new KB object and initializing 3 of its 6 attributes.

Also, we came to the realization that *parsing* is somewhat relative to the level that one is attempting to understand. REFINE helped avoid having to write a traditional parser for the source language, but we still had the problem (now at a higher level) of attempting to determine which partitioning of the, in this case, abstract syntax tree would yield some significant meaning.

Significant Issues Which Were Identified

Here we detail some of the issues which will characterize the general problem of automatic documentation/explanation. These *six issues* overlap to varying degrees, but do contain distinctions. Also, we are not intending to provide complete answers to these issues. They are merely being identified. Section 4 will describe several potentially applicable approaches to the problem.

The first, and perhaps most elementary issue, is that of the *intended audience* for the description. A

description for the programmer who just coded the function must be different from a description for a maintenance programmer, which must be different from one for a system architect. Ourston 1987 identifies an automated approach to formal government documentation requirements (e.g., DOD-STD-2167). This aspect will affect points three, four, and five.

The second significant matter is that of the *source language* of the software to be documented. And within this category there are many different subproblems—depending upon the actual language (see section 3 for a description of some of the issues inherent in analyzing machine code programs for example). Different (computer) languages have certain constructs which make the task of determining the functional contribution of a given subexpression difficult (if not impossible). This aspect has, of course, lead to the design of languages and computational paradigms which avoid these problems from the start (e.g., Bailey 1985 and Backus 1978).

Thirdly, the issue of the *level of abstraction.* As is evident from the title of this article, the phrase "automatic *documentation*/explanation" may be somewhat of a misnomer. It's helpful to ask: What, essentially, is documentation?. Fundamentally, documentation provides *understanding.* Programmers document their programs to record design decisions made; this helps later when the author (or, more significantly, a maintainer) needs to change the source. Of course, *understanding* has the same hierarchical granularity as the thing we try to understand (i.e., a source program). Do you understand why this local variable is being used? Do you understand what role this function plays in the behavior of the abstract data type? Do you understand how some large architectural module fits into the whole software/hardware system? There are different levels of understanding. This is an issue that one need decide—the level of description the system will support.

The *interaction method* makes up the fourth characteristic. This ranges from generating some static representation to some hypertextlike interactive browsing of structure to a more active, cooperative dialog (similar to Hirschberg 1986).

The fifth aspect of automatic documentation/explanation is the *notation used for output description.* This could be English, predicate calculus, finite-state automata diagrams, Petri nets, or Booch diagrams (Booch 1983).

The sixth, and probably the most difficult, aspect of the whole problem is *concept recognition.* If the task is to understand some assembly code, then a higher-level concept could be a *loop;* if the objective is to comprehend what a particular (high-level language) loop statement accomplishes, then the abstracted concept could be *"enumerate and determine if any member of data structure X has some property—delete those that do,"* and so on. The assumption is that particular concepts would be of interest and, additionally, that one can characterize the method of recognizing them.

To deal with this concept recognition issue, we must identify constraints which make the recognition task accomplishable. Many design decisions get "compiled out" of the resulting source code. We are still attempting to develop better modeling formalisms for initial system description (Reiss 1987, Greenspan 1985). In addition, alternate approaches such as domain specific automatic programming (Barstow 1984b, Sayrs 1986) apply application domain knowledge for program generation. The higher-level domain concepts would help to disambiguate certain patterns of program structure and constrain the search for low-level patterns. The domain knowledge of goals and intentions would be very valuable.

3.0 THE SPECIFIC TRANSLATION EXPERIMENT

For purposes of discussion, the following terms are defined: *client*—the software source code of the program to be "documented"; *translator*—the system which will admit a client program, analyze its structure, and output documentation. For the experiment, the implementation language for both the translator as well as the client was REFINE. REFINE is characterized as a multiparadigm very high level language (VHLL) integrating logic, functional, imperative, object-oriented, and transformational programming.

The objective of the experiment was to take an already existing, working module and generate English which described the functionality of the source code. The module submitted for documentation was a track database manager. This module was a component of a larger C^3I simulation consisting of a sensor simulator and an end-user situation display. The functionality of the track data base manager was implemented with six transformation rules.

The initial task was to develop a hand-written specification of the English description which would characterize the track manager. For generality, a "paper" grammar was constructed which identified the REFINE language constructs to be processed. For each language construct, a REFINE function was defined which would take a REFINE KB object (in this case, an abstract syntax tree node) and determine if the node was an instance of, for example, an existential quantification operator and then print the appropriate English for that construct.

Certain contexts were identified where the juxtaposition of two language operators should map to a specific description. An example is the $\neg \exists$ should be "there is no" or "there is not any." The REFINE pattern language was helpful in testing the immediate context of the syntax tree of the program. To test for the abovementioned context the following REFINE expression could be evaluated: A='¬ ∃ .. [@@]' (This expression asks the question: Is the application program expression which is currently bound to A, an instance of a logical negation followed by an existential quantifier?). The documentation function for the ¬ operator would detect one of the special contexts and (in this case) not emit a description but would instead set a *negative voice* context attribute on the existential operator syntax node. This context annotation would be checked by the existential documentation function which would emit appropriate English. Languages allow the recursive occurrence of constructs (e.g., the argument to the ¬ operator could itself be any well-formed boolean expression—including the ¬ operator). The paper grammar described above was used to define the (recursive) documentation function application control flow. A before and after view of the translation of one of the transformation rules appears in figure 1.

These ad hoc techniques made it difficult to generate alternative English phrases which carried the same meaning (to avoid monotonous verbiage). Also, any change in the surface syntax of the REFINE would affect the pattern-matching expressions (i.e., the affected parts of the *translator* would have to be changed and recompiled).

In addition, the order of "concepts" in the program source does not necessarily correspond to the order that humans would find most convenient. The issue was raised while determining how best to explain the right-hand side of the *possibly-create-new-track-entry* rule (fig. 1). Logically, the transformation rule's pre- and postconditions are conjunctions of expressions and therefore have no specific order of evaluation. In the right-hand side of the transform, the first five conjunctive expressions all relate to creating a sighting-history-entry object and subsequently initializing the message-number, location, speed, and observation-time attributes, respectively. While they happened to have been originally coded next to each other in the conjunction, they need not have appeared in that order (the REFINE compiler would have detected the fact that the object, `s`, must be created before it can be initialized).

Bulletization of text (e.g., "the following two items are all red: * car; * boat; ...") is a valuable linguistic/cognitive aid in understanding the relationships of groups of tokens. As such, we made use of it in the English paraphrase of the transformation rule. The REFINE expression (fig. 1a) that indicates object creation, `(sighting-history-entry s)`, serves as a kind of *marker*. When the translator sees this particular type of expression it will survey all the conjuncts (on the right-hand side of the rule) for equality expressions mentioning the variable `s` (e.g., `(message-number s) = (r ∧ incoming-report-message-number)`). After it emits the header string (e.g, `Create an object, called locally s, which is an instance of the object class sighting-history-entry and having the following attributes:`) it will, for every such equality, call the recursive documentation function which emits the text `* message-number, initialized to the incoming-report-message-number field of r` (fig. 1b).

```
(defobject possibly-create-new-track-entry #t rule
          (r:incoming-report) transform
 ¬∃x [x ε *track-database*
       ^ (track-id x) = r↑incoming-report-track-ID]
->
    (sighting-history-entry s)
^   (message-number s) = (r↑incoming-report-message-number)
^   (location s) = (r↑incoming-report-location)
^   (speed s) = (r↑incoming-report-speed)
^   (observation-time s) = (r↑incoming-report-observation-time)
^   (track-entry x-prime)
^   (track-id x-prime) = (r↑incoming-report-track-ID )
^   (last-observation-time x-prime) = (r↑incoming-report-object-type)
^   (affiliation x-prime) = (r↑incoming-report-affiliation)
^   (sighting-history x-prime) = [s]
^   x-prime ε *track-database*)
```

A

```
The rule, POSSIBLY-CREATE-NEW-TRACK-ENTRY, is defined as follows:

The input to POSSIBLY-CREATE-NEW-TRACK-ENTRY is R which  is of type
   INCOMING-REPORT. .

These conditions must be true in order for the rule to fire:

There is no element of *TRACK-DATABASE* such that its TRACK-ID
   attribute is equal to the INCOMING-REPORT-TRACK-ID field of R.

These operations will be performed when the rule fires:

Create an object, called locally S, which is an instance of object class
   SIGHTING-HISTORY-ENTRY and having the following attributes:
       * MESSAGE-NUMBER, initialized to the INCOMING-REPORT-MESSAGE-NUMBER field of R
       * LOCATION, initialized to the INCOMING-REPORT-LOCATION field of R
       * SPEED, initialized to the INCOMING-REPORT-SPEED field of R
       * OBSERVATION-TIME, initialized to the INCOMING-REPORT-OBSERVATION-TIME field of R

Create an object, called locally X-PRIME, which is an instance of object
   class TRACK-ENTRY and having the following attributes:
       * TRACK-ID, initialized to the INCOMING-REPORT-TRACK-ID field of R
       * LAST-OBSERVATION-TIME, initialized to the INCOMING-REPORT-OBJECT-TYPE field of R
       * AFFILIATION, initialized to the INCOMING-REPORT-AFFILIATION field of R
       * SIGHTING-HISTORY, initialized to a literal sequence consisting of S

X-PRIME is an element of *TRACK-DATABASE*.
```

B

Fig. 1. The input and output of the prototype translator. *A*, REFINE transformation rule (source code to be documented) as input; *B*, English output describing the functionality of the rule.

An Application of the Criteria to the Experiment

The intended audience for this experiment was someone who was coding the track manager or someone who might have been in a maintenance role. Johnson (1987, p.17) confirms this sort of tool helps in discovering "differences of opinion" in terms of what a given expression achieves:

> We originally developed this tool to make Gist more accessable to people who were not familiar with Gist. . . . We were surprised to discover that it served another (and perhaps more important) role as a debugging aid. We found that a natural language paraphrase of a specification often made bugs in the specification very apparent that were hidden in the formal specification. Partially, this was due to the fact that natural language is inherently more understandable, but it is also due to the fact that the paraphrase provided the user with another view of his specification that stressed certain aspects of the specification that were not emphasized in the formal notation.

As noted above, the English description could have also been useful to a person trying to learn the specific language.

The source language of the client program in the experiment was REFINE. The top-level language construct was the transformation rule. This made the analysis task easier for several reasons. REFINE is a VHLL. A VHLL* is a language in which a program can be described at a higher level of abstraction than is possible with conventional higher level languages. Generally speaking, one can describe the same functionality with less "code" using a VHLL than with a higher-level language. This means that at least the low-level concept recognition task is simplified (in actuality, what "low-level" *means* has merely shifted to the more abstract VHLL language level).

The transformation rule was a preferred language constituent to document, since it is an example of a declarative programming construct. The transformation rule is of the form A -> B where A and B are predicates (or conjunctions of predicates) defining states of the system. The transformation rule semantics are essentially that if the conditions specified by A can be found to be true (for some binding of the variables) then transform the system so that the state specified by the postconditions (i.e., B) will become the new state. Declarative languages have a property known as *referential transparency* (Darlington 1985). A language system is referentially transparent if "the meaning of a whole can be derived solely from the meaning of its parts" (i.e., the meaning is not time-dependent upon past computations). This aspect of the transformation rule construct aided the translation, since we were not forced to look at how a variable was used in another program/function.

The level of abstraction was fairly low. But the abstractions were representative of the types of things the human would like to know quickly without having to look at so many tokens in the program. The example in figure 1 tells the reader that they will be *creating* a new track-entry, *initializing* its attributes, and it will then be added to the track database. The system *recognized a higher level programming cliche* (for a description of cliches given in Waters 1985, see section 4 below) of *creating an object* and *initializing all of its attributes to values from fields of record r*.

The interaction method used in this experiment was a simple static generation of the description. The notation used for the output description was English. The analysis of the six rules identified a recurring pattern: object *x* has some property (e.g. a new instance has just been created, and x is bound to it) and that object will have several of its attributes "manipulated" (again, in the case of creating a new instance, several of its attributes will be initialized to certain values). This recurring pattern is essentially "converted" to the bulletization form for the output description.

---

*VHLLs have the following properties: (1) implicit specification of control flow (e.g. functional/logic programming, data flow); (2) high-level data structures (e.g., sets, bags, sequences, lists, relations, patterns); (3) aggregate operators (operators which can be applied directly to an entire data structure in a single user-level operation, e.g., reduction operator in APL, MAP feature of LISP); (4) associative referencing (the ability to identify and extract from a larger structure, the particular objects one desires by providing a partial description of them, e.g., relational DBMS, Prolog).

4.0 A PLAN TO BRING MORE SOLUTIONS TO BEAR

Automatic documentation/explanation seems to be a hard problem. There are a number of areas of research that we feel could be profitably leveraged. Some of these provide paradigms or analogies (e.g., image-pattern recognition) for approaching the task while others are more directly related.

Machine translation (MT) is essentially concerned with automated techniques for translation from one natural language to another (e.g., English to German). The problem of automatic documentation/explanation is similar in that we start with a text in some language (in this case a formal computer language such as REFINE) and would like to translate it to another language (English). I believe MT to be a very good candidate for leveraging techniques for our needs. Slocum (1987) describes objectives that intersect with needs in software explanation: "We outline . . . a means of dealing with two daunting problems confronting MT system developers: (1) that of developing a *comprehensive, consistent* description of a language, and (2) that of formally describing the interaction among concept, word, and linguistic environment." Additionally, he (1987, p.3) identifies a spectrum for the depth of analysis:

> At the shallowest level, one can perform **direct** translation, which is characterized by the fact that analysis of the source language (SL) is restricted to the minimum work necessary to produce a translation in a single, specific target language (TL). Assuming that such a shallow analysis can be sufficient for high-quality translation (and there are arguments concerning whether or not this is true), it remains the case that another translation of the same input, into a second TL, requires complete re-analysis of that input [given that you're doing a direct one-to-one mapping in a single operation]. . . .
>
> At the other extreme, one can opt to analyze an input into a deep, language-independent representation of meaning called an **interlingua**. From this meaning structure, one can in principle translate into multiple languages by merely synthesizing multiple outputs—without ever re-analyzing the input. . . . One major problem with this is that linguists are not yet able to specify an interlingua: the necessary theories have not yet been devised (indeed, that one can possibly exist is mere conjecture).
>
> A compromise position involves an intermediate stage called **transfer**. An input is analyzed into a structure deeper than that required for direct translation, but not (quite) language-independent like an interlingua. Then a transfer step transforms that structure (peculiar to the SL) into an equivalent structure peculiar to the TL, and the synthesizer uses the result to produce the output translation.

Computer language translators (both source-to-source, as well as decompilers) also represent an area to survey for potentially applicable techniques (although we have had mixed experience with respect to the quality of the resultant code in the case of source-to-source translation). *Decompilation* is the process of translating from a lower-level language (typically assembly/machine code) to a higher-level programming language. The common motivation for performing the translation is that the original source code has either been lost or is unavailable. It is advantageous to maintain software in the higher-level language, rather than at the machine code or even assembly level. This research is of great interest, since automatic documentation/explanation is attempting to analyze a source program (be it in a HLL or a VHLL) and determine higher-level properties of the source code. Some initial problems in this area concern merely identifying the separation of program from data (Horspool 1980). Next-level recognition tasks relate to detecting higher-level control structures (Lichtblau 1985).

Work on question answering and/or intelligent data base query systems reflects the confluence between AI and DBMS technology. This yields several applications of natural language processing (NLP) techniques: natural (English) language interfaces as well as user/dialog models which provide intelligent help to the user. Hirschberg (1986, p. 631) illustrates some of the objectives: "Emulation of natural discourse in computer-human interaction can permit the generation of more cooperative responses to user input, which may correct user misconceptions, avoid the licensing of misconceptions, convey information concisely, provide appropriate explanations of system reasoning, and so on; . . . such emulation can also permit systems to infer information that may be implicitly conveyed by a user and so make explicit questioning unnecessary."

In a fundamental sense, automatic documentation/explanation is a case of pattern recognition. Pattern recognition is the process of observing some low-level input stream and inductively perceiving an abstraction which aggregates some part of the input stream. Automatic documentation/explanation has commonality with, specifically, *image*-pattern recognition (Waltz 1975). The aspect of attempting to recognize low-level patterns and *building* upon these abstractions to induce perception of even higher-level patterns occurs in both: in the case of the former: (assuming the language is a HLL) variable assignments, iteration, object creation; in the case of the latter: straight lines, connected lines, real-world objects (such as a house). Rich (1983, p. 349) provides a taxonomy of recognition tasks and articulates the interrelation between levels:

> *digitization*—Divide the continuous input into discrete chunks.
>
> *smoothing*—Eliminate sporadic large variations in the input . . . these spikes are usually the result of random noise.
>
> *segmentation*—Group the small chunks produced by digitization into larger chunks corresponding to logical components of the signal. . . . For visual understanding, the segments usually correspond to some kind of significant feature of the objects in the picture [e.g., obvious lines].
>
> *labeling*—Attach to each of the segments a label that indicates which, of a set of building blocks, that segment represents. . . . For vision, this means assigning labels such as: "This line represents an exterior edge of a figure." . . . It is not always possible to decide, just by looking at a segment, what label should be assigned to it. [The labeling procedure] can assign multiple labels to a segment and leave it up to the later analysis procedure to choose the one that makes sense in the context of the entire input, or it can apply its own analysis procedure in which many segments are examined to constrain the choice of label for each segment.
>
> *analysis*—Put all the labeled segments together to form a coherent object. This is the stage where the fewest generalizations can be made. It is almost always necessary to exploit a great deal of domain-specific knowledge here. . . . The one thing that almost all the analysis procedures have in common is that they are variations on the basic process of constraint satisfaction. This is because during the higher-level analysis process, just as with low-level labeling, there are often many possible interpretations of a given piece of input. But when surrounding pieces are considered, the number of interpretations that lead to a consistent overall interpretation is considerably reduced.

Another source of constraints on the recognition task is based on the observation that identifiers used in programs (e.g., names for functions, variables, data structures, etc.) represent compiled knowledge or semantics of the purpose of the specific identifier as well as provide potential clues to higher level program design decisions. Of course, one must be careful here, since we have the problem of "noise"—an identifier may, due to program maintenance, become "misleading" with respect to its new functionality. (An interesting idea to explore with respect to software restructuring would be to check a program's identifiers and determine if the semantics of the identifier have remained consistent with apparent use/definition of the identifier—the idea is automatic identifier semantic consistency maintenance—this could be of use since a class of maintenance errors seem to be related to the maintainer thinking an identifier does such and such as a result of the identifier's semantics). Computer science has developed the social convention of using long, descriptive identifier names. This has the purpose of implicitly documenting the function of the identifier (and, again, providing an allusion to one or more design decisions).

Three ideas are identified as the basis of the Programmer's Apprentice Project (Waters 1985): the *assistant approach*, the concept of *cliches*, and the *plan calculus*. Cliches represent the following:

> A standard method for dealing with a task—a lemma or partial solution. . . . A cliche consists of a set of *roles* embedded in an underlying *matrix*. The roles represent parts of the cliche which vary from one use of the cliche to the next but which have well defined purposes. The matrix specifies how the roles interact in order to achieve the goal of the cliche as a whole. . . . Given a particular domain, cliches provide a vocabulary of relevant intermediate and high-level concepts.

The *plan* is a higher level knowledge representation formalism than the cliche:

> . . . Plan—a representation which is abstract in that it deliberately ignores some aspects of a problem in order to make it easier to reason about the remaining aspects of the problem. . . . The plan formalism . . . is

designed to represent two basic kinds of information: *the structure of particular programs* and *knowledge about cliches* [italics mine].

The plan formalism was developed primarily for computer-aided construction of software (a longer-term goal is an understanding of human problem-solving behavior). However, as the following quote indicates, the inverse objective of deriving abstract concepts from program source (i.e., AUTODOC) shares some objectives with the plan calculus: "An important aspect of the plan formalism is that it *abstracts away* from the *syntactic features of programming languages* and *represents* the *semantic features of a program* directly. Besides facilitating the manipulation of programs, this has the collateral advantage of making the internal operations of KBEmacs *substantially programming language independent* " [italics mine].

Software inevitably endures a process of evolution. The addition of functionality, the correction of bugs, the redesign of major subcomponents, and of course porting software to a new environment all are sources of change. Understanding both *motivations* for software changes as well as *specific strategies* for modification (or the lack thereof!) would be valuable to the program understander. These are additional types of patterns to be identified in program source code (and in the case of dysfunctional patterns, to be corrected). Research on software restructuring (Arnold 1986) is relevant for both techniques for recognizing patterns as well as empirical characterizations of how software gets changed. More basic research on understanding how software evolves (Dershowitz 1983, Zelinka 1983, Gray 1987) is important in formalizing the identification of patterns for a program understander.

The area of automatic program understanding is certainly not new. The following work also contributes to the solution of the problem of software understanding: Waters 1976, 1978; Ruth 1976; Soloway 1981; Wile 1983; Rich 1985; Johnson 1987. As the title of this paper implies, this is a report of a prototype implemented to explore the issues in automatic program documentation/explanation. It is at this juncture, after possessing an originally acquired viewpoint on the software understanding problem, that we are venturing out to other specifically related work (above). I believe this is justified based upon my (as yet unarticulated) theory of original knowledge discovery. I provide but an aphorism here: "Turing had a strong predeliction for working things out from first principles, usually in the first instance without consulting any previous work on the subject, and no doubt it was this habit which gave his work that characteristically original flavor" (Wilkinson 1987, p. 246).

5.0 APPLICATIONS OF THIS RESEARCH AREA

We believe that this research area has many potential applications (in addition to the obvious automatic explanation objective). Software reuse could be a potential benefactor of a system that could read in *existing, field tested* software, and "determine" what the software does functionally (since the result of such analysis would be formal products, these formalisms could be communicated to some software reuse system for indexing). Of course, people needing to perform any sort of maintenance programming would find a use for an intelligent system which could accept queries on certain specific aspects of an already operating software system or could view previously written software via Booch diagrams (1983) (assuming of course that the software was written in Ada). Automatic software restructuring (Arnold 1986) is another example of potential relevance of this work.

A person needing to code (for the first time) a "delete" routine for some particular data structure may like to ask: How has data structure X been used in the program? (particularly when person A isn't the one who designed X). Intelligent debugging assistance has been investigated (both from a "traditional" debugging standpoint as well as a computer-aided-instruction/tutorial approach), and some of the techniques of abstracting "computational concepts" from the source code may help such debugging systems identify violated assumptions, and so forth.

The problem of porting software will probably always be with us. It seems conceivable that a tool could at least *identify* code (in an existing system) that is dependent upon the current operating