Tackling Uncertainty in A Cognitive Framework for Source Code Understanding

Yang Li and Hongji Yang
De Montfort University
England
yangli, hyang@strl.dmu.ac.uk

William Chu
TungHai University
Taiwan
chu@cis.thu.edu.tw

K. S. Liu
Department of Information Science
Feng Chia University
Taiwan

Abstract

Source code understanding plays an important role in software maintenance. Although existing knowledge-based program understanding techniques distinguish themselves from structural/syntax-based program analysis methods for acquiring high-abstraction-level oriented result, they all fail to address uncertainty issues inherited in program comprehension process. The lost benefit from tackling huge amount of uncertainty information renders those knowledge-base methods both ineffective and inefficient; in most cases, they simply provide a framework where the linking among different information sources is manually carried out by programmers. In this paper, we devote ourselves to addressing the uncertainty issues in a cognitive framework proposed in previous work for code understanding [12]. An example is given to illustrates this approach.

Keywords : Program Comprehension, Uncertainty Reasoning, Possibility Theory, Natural Language Understanding, Artificial Intelligence

1 Introduction

Program understanding plays an important role in nearly all software related tasks, especially software maintenance. Studies of the software maintenance process indicate that software maintainers, on the average, spend approximately half of their time developing an understanding to software [16]. One of the primary reasons for this is that the documentation and other formal descriptions of large systems are often inadequate and unreliable. As a result, software maintainers typically rely on source code as the only completely reliable source of information on the software. The process of trying to develop an understanding of the software by manually navigating through the code is extremely time consuming and error prone. This situation has created a need for technologies and tools supporting in this activity.

Although a number of tools [6, 20, 3, 19] were developed for the purpose of program comprehension, their focusing on syntactic rather than semantic analysis seperates them from the users' satisfaction. Inspired by the cognitive studies [13, 4, 10, 14] which suggest that the understanding process is one in which programmers make use of stereotyped solutions to problems in making sophisticated high-level decisions about a program, several studies [15, 18, 9, 8] were undertaken to develop knowledge-based approaches for program understanding.

On one hand, most of the existing knowledge-based approaches rely heavily on real-time user-supplied information that might not be available at all times. For instance, goals that a program are supposed to achieve or transformation rules that are appropriate for analysing a specific code fragment are not always clear to the user. On the other hand, program transformation which attempts to screen out the noise in source code representation and provide maintenance programmers with a more abstract view of the system seems to step into the world of NP-hard problems and got "lost" in low level details.

To address the above-mentioned drawbacks, we must rethink the strategies by which we are dealing with program understanding, more specifically, the following questions should be pondered on:

- What is the "understanding" of a program in general sense?
- What is lost during *specification* to *code* stages? Need/can we recover every detail?
- Is compromise allowed between high accuracy (demanding large resource) and low accuracy (need only low cost) for program understanding, i.e., can we tolerate uncertainty in the understanding process?

In [11], the retrospect of the history of Artificial Intelligence (AI) was given and the impact of AI on software reverse engineering was analysed. During

50 years of AI history, the early attempt of designing a general-purpose theorem prover which can address all the issues in the world was finally found impracticable and gave way to the later-prevalent techniques known as knowledge engineering and expert system. The reason behind this shift is due to the high complexity of real world problems which usually lead to *combinatorial explosion* in computing and thus a significant reduction of the complexity of problems is needed. Expert system is aimed to capture only necessary fraction of human knowledge which is small in quantity and effective to use. In this case, uncertainty will inevitably arise on the way to approximate the real world.

Program understanding is also an intelligence behaviour. It should to some extent comply with AI principles. In [12], an expert system which aims to elevate source code to domain level was introduced. The heart of this system is a cognitive framework which can integrate information from different information sources. The acquisition of knowledge from single information source can itself be achieved by an expert system (Variable names, mathematical model, etc.). Due to complexity of uncertainty issues inherited in the understanding process (ambiguity of names, imprecise evidence, incomplete knowledge, unreliable information sources, etc.), in this paper, we will introduce how we tackle the uncertainty issues in our source code understanding system. Both the cognitive model we devised and the uncertainty processing addressed here were not found in existing work. We hope through this paper, uncertainty knowledge in program understanding can be given enough attention it deserves.

The rest of paper is organised as follows. In Section 2, we will refresh our memory of the cognitive framework introduced in [12]. In Section 3, a rough description of uncertainty processing in different components of the framework and how they are integrated is given. An example is used to illustrate this approach in Section 4 and finally we reach our conclusion and propose future work.

2 Our Cognitive Model for Source Code Understanding

When analysing source code, programmers are willing to look for mathematical models or algorithms embedded in the code. These kinds of information gathered can act as a strong evidence supporting what the function of program section analysed is. We term this group of information as *formal knowledge sources* which, once recovered, can render programmers high confidence. Beside this, *informal information sources*, such as names of variables and procedures, program comments can, to a large degree, inspire the imagination of programmers about the program being analysed. After all, the software implement the function of domain and setting link between source code and domain model can make the consequent analysis of source code much easier. Informal information sources can help elevate mathematical/structure-oriented model to the level of domain/semantic-oriented model. The latter is what is highly appreciated by different kinds of staff, from program re-engineers, through software designers, to management executives.

Based on this psychological necessity discussed above, we present a cognitive model for source code understanding in Figure 1 [12]. This approach is devised under the principle of AI engineering. In common with most AI applications, it also employs a knowledge base and knowledge engineering techniques. Moreover, it integrates formal and informal information processing techniques into a single framework. In Figure 1, source code is first dispatched into a formal information analysis module (1) and an informal information analysis module (2). In (1), formal knowledge bases are used to extract formal knowledge models hidden in source code; while in (2), informal information extraction rules are employed to acquire key concepts from the names of variables, procedures and program comments precede to construct the domain level context of the program being analysed. Next, we combine the output of (1) and (2) in (3) to provide better cognitive result. This semantic result can be further abstracted and augmented under the aid of the knowledge base and provide heuristics for the analysis of other program sections.

3 Dealing with Uncertainty

Uncertainty is common phenomenon in identification, recognition, comprehension, and other understanding activities [5]. More often than not, we have to reason in a world of uncertainty, because observation is incomplete, unbelievable, or even the rule itself is not 100% certain or reliable. If we confine ourselves to certainty reasoning, we will miss the potential of tremendous information treasure which can be acquired from either valuable expertise or the weakly but still connected linkages among entities which certainty reasoning can't tackle. The same principle applies to program understanding. Many methods for dealing with uncertainty exist in the literature. Among them the most classical methods are Bayesian method, Fuzzy Set, Possibility theory, and Demster-Shafter Evidence theory [7, 1, 2]. We choose Possibility Theory [2] as uncertainty reasoning model for the purpose of preliminary evaluation. A collection of rules for different component of our system and the uncertainty characteristics of these rules will be roughly introduced in the rest of this section. More vivid illustration of the application of these rules can be found in Section 4.

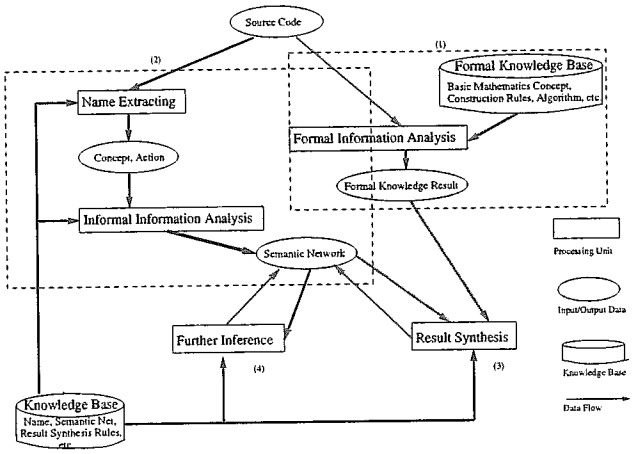

Figure 1: Framework of Source Code Understanding Using Combined Formal/Informal Information Processing

3.1 Formal Information

Formal information is embedded in variable type, program construct, control flow, data flow, etc. They have the characteristics of unambiguous, easy-to-processing, etc. Many program analysis techniques [17] exist around this topic. For clarity, we have given a taxonomy of rules for discovering formal knowledge: mathmatics-sensitive rules, protocol-sensitive rules and algorithm-sensitive rules. In our preliminary evaluation, we give the highest authority weight to formal knowledge.

3.2 Informal Information

Informal information exists in: naming of variables and procedures, program comments, etc. We have developed a set of rules for discovering them.

3.2.1 Variable and Procedure Names

Rules of the naming of variables and procedures are gathered. The applicability of naming rules to the analysis of the names in source code lies in:

- Only a small number of names are generally used in one program. They are confined by the context of domain.
- Only a small number of naming rules are used by the programmers.
- The finite combination of the numbers of names and rules make it possible for computers to reason out the possibly correct name within acceptable time boundary.

In our system, the component of name extraction is among the key components prior to constructing a semantic network which reflects the domain context of the program. The quality of the result in this stage will greatly affect the efficiency and quality of that in later stages. Because for the moment we needn't integrate names with other information source, we don't assign the authority weight to this component. However, we do set a high threshold to the output of this component and only the result with high precision can be filtered through and forwarded to subsequent stages. In this way, the fruitless and time-consuming processing in the system can be significantly avoided. As an important control parameter, the adjustment of threshold can also significantly affect the performance of the system.

3.2.2 Program Comments

During forward engineering most programmers write down program comments in natural languages along side source code for later reference. These kinds of information source are scattered around source code and play a vital role for program understanding.

It was established that a complete syntactic and semantic analysis of text was not feasible, given the current state of the art of in computational linguistics and the quantities of documentation that have to be processed in source code. Short-cuts which retain a degree of *understanding* have to be used in the analysis process to locate the key concepts and events.

Program comments analysis share the same role with extracting names of variables and procedures mentioned in Section 3.2.1 and the same considerations of uncertainty constraint are given to it. The high threshold is also set on the output of this component.

```
******************
Atom Naming Rules
******************

Rule          Operation                                          Rule Strength

---------------------------------------
...

Complete      full word is written for a particular name         (1.0)
Head3         Write the first 3 characters                       (0.2)
Head2+Tail1   Write the first two characters and the last characters  (0.3)

...
```

Script 1

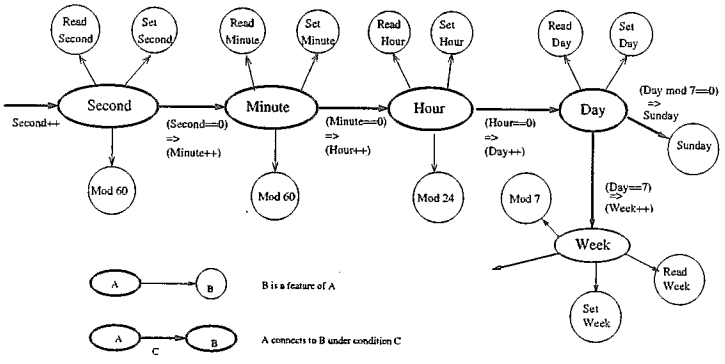

Figure 2: Semantic Network of a Clock

3.3 Generating Semantic Network

The extracted key concepts in Section 3.2.1 and Section 3.2.2 are used to construct the context of program section in the level of application domain, which can help elevating source code to a higher level. In our system, there are predefined name-context mapping table called *Name Dictionary* and semantic-network base. The former is used to decide the possible context in which the given concept resides; while the latter holds a list of typical context where detailed description of each context is given in the representation of *semantic network*. Script 2 shows the structure of Name Dictionary. An exemple of semantic network can be found in Figure 2.

```
***************
Name Dictionary
***************

Name    Morphological  Semantic Nets1  Semantic Nets2  ...
        Feature
-------------------------------------------------------------
...

Second  Noun           Clock-Net (0.8)  Ordinal-net (0.8) ...
Minute  Noun           Clock-Net (1.0)  ...

...
```

Script 2

We can see that one single key concept can appear in more than one context. In each context, the bracketed real number following the name of the semantic net gives the strength of that rule. To construct a united semantic network which reflects the context of a procedure, we simply connect all the candidate semantic sub-networks obtained in previous stages. Each sub-net retains its own possibility degree.

3.4 Combining Formal and Informal Knowledge and Further Inference

Although rigorous, formal knowledge especially mathematical knowledge alone can hardly leap to the level of domain without the help of informal information described in the natural language. In the mean time, although easier to understand by programmers, informal knowledge can not be elicited directly as a domain-level explanation of program section without the strong support and proof from the formal information. This situation forced us to come up with a suitable way of combining formal and informal information which can take both advantages from them. In our system, we use model matching techniques to combine formal and informal information. Once acceptably matched, formal knowledge and informal knowledge are strongly linked.

After successfully elevating mathematical/ algorithmic/ structural-oriented model into domain model, we can further apply expert rules which are closely connected with the domain context. This can in turn contribute a lot of heuristic hints to other uninterpretable program sections.

4 An Example

4.1 A Program Section

A "clock program" is shown in Script 3 as an example.

```
*************************************************
        A  Clock Program
*************************************************
...

int Second, Minute, Hour, Day;

...

void Increase(){

        Second++; /* Click a second */
        if(Second==60){
                Second=0;
                Minute++;
                if(Minute==60){
                        Minute=0;
                        Hour++;
                        if(Hour==24){
                                Hour=0;
                                Day++;
                        }
                }
        }
}

void Routine(){
        if(Day % 7==0)
                gotoChurch(Jason);
}

...

*************************************************

                Script 3
```

4.2 Extracting Groups from Source Code

The source program in Script 3 is first transformed into the Control Flow Graph. Figure 3 shows the Control Flow Graph of Procedure *Increase()* in Script 3.

We concentrate on the data flow analysis of individual variables. For example, in Figure 3, the dotted lines represent the data flow of variable "Second" and a mathematical analysis is given to "Second" in order to extract its mathematical model:

"int Second": We can easily infer from this declaration that $(Second, +)$ is a finite group, where "Second" is an integer set and "+" denotes arithmetic addition.

Dotted lines which captures the data flow of variable "Second": Further mathematical rules are applied in this sub-graph and such result is reasoned out that $(Second, +)$ forms a *finite group* where $Second = \{0, ..., 59\}$, "+" is *arithmetic addition under modulo 60.*

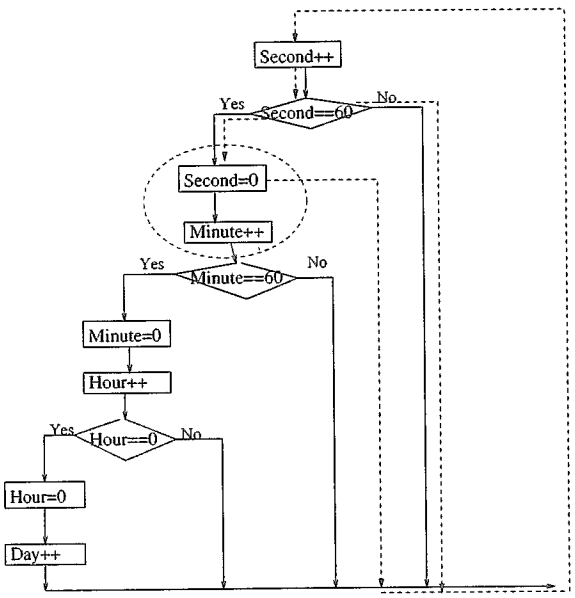

Figure 3: Control Flow Graph and Analysis of "Clock" Source Code

Variable Name	Mathematics Result
"Second"	a finite group (Second, +), where $Second = \{0, ..., 59\}$; "+" is *arithmetic addition under modulo 60*
"Minute"	a finite group (Minute, +), where $Minute = \{0, ..., 59\}$; "+" is *arithmetic addition under modulo 60*
"Hour"	a finite group (Hour, +), where $Hour = \{0, ..., 23\}$; "+" is *arithmetic addition under modulo 24*
"Day"	a finite group (Day, +), where Day={integer set}; "+" is *arithmetic addition*

Table 1: The Result of Applying Mathematical Analysis to the Example Program

	"Second"	"Minute"	"Hour"	"Day"
"Second"		$(Second == 0) \rightarrow (Minute++)$		
"Minute"			$(Minute == 0) \rightarrow (Hour++)$	
"Hour"				$(Hour == 0) \rightarrow (Day++)$
"Day"				

Table 2: Mathematics Relationships among the Variables in Table 1

Similarly, we can also get finite groups $(Minute, +)$, $(Hour, +)$, $(Day, +)$, where $Minute = \{0, ..., 59\}$, $Hour = \{0, ..., 23\}$ respectively, $Day = \{IntegerSet\}$ and "+" denotes arithmetic addition under modulo 60, 24 and large infinite integer numbers respectively.

Relationship among groups: Analysis is also give to the relationship between two groups and the interface between the two groups is captured. For example, the dotted circle in Figure 3 captures the interface between group "Second" and group "Minute" and such relationship that $(Second == 0) \rightarrow (Minute++)$ is identified. The similar analysis is given to group "Minute"and "Hour", group "Hour" and "Day".

The result above is summarised in Table 1 and Table 2. In this example, we give full score, 1.0, to the authority weight of formal knowledge and all these results obtained are filtered through to next stage.

Although, the names of variables such as "Second" presented in the example are meaningful to programmers, from the viewpoint of mathematics or a compiler, "Second" is no more meaningful than "V1" or "011010" and although we can make certain that there do exist four groups and three relationships among these groups, we can not make sure that these four groups actually form a clock because there may be other explanations for this behaviour. We need to take the advantage of informal information such as natural language description contained in the source code to aid our decision-making.

4.3 Extracting Names from Source Code

By looking up in the *name dictionary* (see Section 3.3) and applying *naming rules* (see Section 3.2.1), we extract variable names from the program and show them in Script 4. The extracted names are listed in the second column followed by their possibility value in the third column. Because all the names here present themselves in their full name in the program, therefore, TRUE (1.0) is assigned to their possibilities. The source in the first column records where a *name* comes from. The bracked *V/P/PS* follows indicates the source is a variable name/procedure name/segment of procedure name. The field of *source* is therefore playing a *tracing* function. The reason we set the field of *morphological feature* is due to its important role in syntax analysis and semantic reasoning. The Semantic Net Set in the last column is used to reflect the context of corresponding name. It is populated by *Name Dictionary* shown in Script 2. There can be more than one context for a given name. The bracketed number follows each sub-net indicates the possibility by which the context of *a procedure* is associated with that sub-net. It is determined by (1) the possibility by which the name can be infered from the source; (2) the possibility by which the sub-net can be supported by the name. If there is more than one source in the procedure supporting the sub-net, we will synthesise these results prior to constructing a semantic net, which will be discussed in Section 4.5. We set 0.3 to the *output threshold* of this component.

4.4 Dealing with Program Comments

By collecting program comments in Script 3 we got:

```
Click a second
```

From the semantics of "click a second", we can reduce the candidate semantic nets in Procedure *Increase()* to two, say, clock-net and financial-net by removing ordinal-net which has nothing to do with "click a second". This simplifies the context of procedure *Increase()*.

	Clock-Net (1.0)	Financial-Net (0.8)	...
Matching Result1	1.0	0.2	...

Table 3: Matching Degree for Mathematics and Informal Information

	Clock-Net	Financial-Net	...
Matching Result2	0.85	0.17	...

Table 4: The Result after Combining Mathematics and Informal Information

```
++++++++++++++++++++++
Name Extraction Result
++++++++++++++++++++++

Source          Name        Possibility     Morphological       Semantic Net Set
                                              Feature
-------------------------------------------------------------------
Increase (P)    Increase      (1.0)           Verb          Clock-Net (0.8), Financial-Net (0.8), ...
Second   (V)    Second        (1.0)           Noun          Clock-Net (0.8), Ordinal-Net (0.8), ...
Minute   (V)    Minute        (1.0)           Noun          Clock-Net (1.0), ...
Hour     (V)    Hour          (1.0)           Noun          Clock-Net (1.0), ...
Day      (V)    Day           (1.0)           Noun          Clock-Net (1.0), ...

Routine  (P)    Routine       (1.0)           Noun          Christian-Net (0.3), Sanitation-Net (0.3), ...
Goto     (PS)    Goto         (1.0)           Verb          Christian-Net (0.3), Sanitation-Net (0.3), Shopping-Net (0.3), ...
Church   (PS)    Church       (1.0)           Noun          Christian-Net (1.0), Sanitation-Net (0.4), Architecture-Net (0.6), ...
-------------------------------------------------------------------
```

Script 4

4.5 Constructing Semantic Network

Let's take procedure *Increase()* for example. There are more than five sources, i.e., increase, second, minute, hour, day, give support to clock-net; on the contrary, relatively few source, e.g., increase, contributes credit to financial-net. Here, we give full authority weight to each source. After computing, we got the possibility of clock-net and financial-net to be 1.0, 0.8 respectively.

4.6 Guess the Clock

So far, we have obtained the mathematical result in Section 4.2 and informal hypothesis in Section 4.5. By comparing candidate sub-nets like Clock-Net, Financial-Net with mathematical results in Table 1 and Table 2 we found that Clock-Net can best match with the mathematical results. The matching result is shown in Table 3. The calculation for "matching degree" is determined by two factors: similarity of network topology and network connective condition.

Taking also the possibility of each semantic subnet as the context of the procedure , the authority weights of formal information weight and informal information (We give 1.0 and 0.85 to the weights of the two information sources. All of them is adjustable according to the quality of corresponding information source) into consideration, we finally got the result of synthesis of formal and informal information for *procedure Increase()* as shown in Table 4.

To provide later reasoning phases a more meaningful input, here, we set *acceptable threshold* to be 0.60 and therefore Financial-Net (0.14) is blocked while Clock-Net (0.8) can be filtered through. Based on this result, we can guess that the program section is actually implementing the function of a "Clock".

Here we can see how program code is elevated to domain level and the link between concept and code is set up. Most important, a linkage strength, say 0.85 is attached to this link and quantifies such a coupling.

4.7 Inference: Jason Goes to Church every Sunday

We now analyse *procedure Routine()*. By using the context of Clock shown in Figure 2, we can infer "Day==Sunday" (0.85) from "if (Day % 7 =0)". Here again, 0.85 which follows "Day==Sunday" is the possibility of such conclusion. It is incurred by the uncertainty of "day" as "day" is a component of clock and clock has the possibility value of 0.85. Based on this, *procedure Routine()* can be abstracted to "Jason goes to Church every Sunday" (0.85).

4.8 Further Inference: Jason Is a Christian Or a Cleaner

If our knowledge base contains sufficient knowledge, further inference can be evoked to give more abstract result. For instance, Christian-Net and Sanitation-Net in this case will be resorted to and provide the results like "Jason is a Christian" or "Jason is a cleaner". We will not discuss this situation furthermore in this paper.

4.9 Summary

This example shows how a high level semantic knowledge comes out from our expert system when source code is input. Particularly, it illustrates how uncertainty is defined, processed, propagated, controlled and attached in this rule-based reasoning process. We give relatively a high authority weight to formal information than informal information. Moreover, we impose more constraints on informal information especially early stages in order to reduce the bad effect of ambiguity of informal information to the least. Based on the uncertainty values of formal and informal information, their similarity and authority weights, these two kinds of heterogeneous information were finally linked and the linkage strength is also endorsed.

5 Conclusion and Future Work

Code understanding is the first step in reverse engineering and it plays important roles in various software engineering activities. One of the reasons why automated program understanding failed to be used in practice is due to the programmers' attitudes to uncertainty and ambiguity which hamper them from benefiting from the huge amount of information where uncertainty is a nature. In this paper, we address the uncertainty issues arising from our expert system for source code understanding and show how the uncertainty in code understanding process is defined, processed, propagated and controlled. In our preliminary evaluation, we employ Possibility Theory as uncertainty reasoning model. The evaluation result of this uncertainty-oriented approach show a fairly promising perspective.

The future work is to deepen the existing methods, e.g., to dig out more information sources hidden in the source code which can contribute to the understanding of program and to carry out more experiment and evaluation of the existing uncertainty processing models in a larger scale program.

References

1. *A Mathematical Theory of Evidence.* Princeton University, 1976.
2. *Possibility Theory.* Plenum, New York, 1988.
3. K. H. Bennett and M. P. Ward. Using formal transformations for the reverse engineering of real-time safety critical software. In *Proc. Second Safety-Critical Systems Symposium*, pages 204–223, Birmingham, 1994. Springer Verlag.
4. R. Brooks. Towards a theory of the comprehension of computer program. *International Journal of Man-Machine Studies*, 18(6):543–554, 1983.
5. B. J. C. Fuzzy models – what are they and why? *IEEE Transactions on Fuzzy Systems*, 1(1):1–6, 1993.
6. P. Chan and M. Munro. Pui : A tool to support program understanding. In *Proceedings of the International Workshop on Program Comprehension; WPC'97, Dearborn Michigan.* IEEE Press, 1997.
7. K. George. Probabilistic vs. possibilistic conceptualization of uncertainty. In B. A. et. al., editor, *Analysis and manangement of Uncertainty*, pages 13–25. Elsevier, 1992.
8. M. T. Harandi and J. Q. Ning. Knowledge-based program analysis. *IEEE Software*, pages 74–81, Jan 1990.
9. W. L. Johnson and E. Soloway. Proust: Knowledge-based program understanding. *IEEE Trans. on Software Engineering*, 11(3):267–275, 1985.
10. S. Letovsky. Cognitive processes in program comprehension. *Empirical Studies of Programmers*, pages 58–79, 1986.
11. Y. Li and H. Yang. Will ai help software reverse engineering? In *CACUK*, England, Sep 1998.
12. Y. Li, H. Yang, and Z. Cui. An automated, intelligence-based approach to elevating source code to domain model. In *CACUK*, England, Sep 1999.
13. D. C. Littman, J. Pinto, S. Letovsky, and E. Soloway. Mental models and software maintenance. *Empirical Studies of Programmers*, pages 80–98, 1986.
14. N. PenNington. Stimulus structures and mental representations in expert comprehension of computer programs. *Cognitive Psychology*, 19:295–241, 1987.
15. C. Rich and L. Wills. Recognizing a program's design: A graph-parsing approach. *IEEE Software*, 7(1):82–89, 1990.
16. R. Rock-Evans and K. Hales. Reverse engineering: Markets, methods and tools. vol. 1, Ovum Ltd., 1990.
17. A. Temin and E. Rich. Automating the desk analysis of programs. In *Proc. 20th Hawaii Int. Conf. on System Sciences*, pages 54–63, 1987.
18. L. Wills. Flexible control for program recognition. In *Proc. the Working Conf. Reverse Engineering*, Baltimore, 1993.
19. H. Yang, P. Luker, and W. Chu. Code understanding through program trnsformation for reusable component. In *5th International Workshop on Program Comprehension*, pages 148–157. IEEE Computer Society, May 1997.
20. P. Young and M. Munro. Visualising software in virtual reality. In *Proceedings of the International Workshop on Program Comprehension.* IEEE Press, 1998.