Program Understanding and the Concept Assignment Problem

Ted J. Biggerstaff
Microsoft Research
and
Bharat G. Mitbander and Dallas Webster
Microelectronics and Computer Technology Corporation (MCC)

ABSTRACT

A person understands a program because he is able to relate the structures of the program and its environment to his conceptual knowledge about the world. The problem of discovering individual human oriented concepts and assigning them to their implementation oriented counterparts for a given program is the *concept assignment problem.* We argue that the solution to this problem requires methods that have a strong plausible reasoning component based on *a priori* knowledge. We illustrate these ideas through example scenarios using an existing design recovery system called DESIRE.

1. Human understanding and the concept assignment problem

A person understands a program when he is able to explain the program, its structure, its behavior, its effects on its operational context, and its relationships to its application domain in terms that are qualitatively different from the tokens used to construct the source code of the program. That is, it is qualitatively different for me to claim that a program "reserves an airline seat" than for me to assert that

> "if (seat = request(flight)) && available(seat) then reserve(seat,customer)."

Apart from the obvious differences of level of detail and formality, the first case expresses computational intent in human oriented terms, terms that live in a rich context of knowledge about the world. In the second case, the vocabulary and grammar are narrowly restricted, formally controlled and do not inherently reference the human oriented context of knowledge about the world. The first expression of computational intent is designed for succinct, intentionally ambiguous (i.e., informal), human level communication whereas the second is designed for automated treatment, e.g., program verification or compilation. Both forms of the information must be present for a human to manipulate programs (create, maintain, explain, re-engineer, reuse or document) in any but the most trivial way. Moreover, one must understand the association between the formal and the informal expressions of computational intent.

If a person tries to build an understanding of a unfamiliar program or portion of a program, he or she must create or reconstruct the informal, human oriented expression of computational intent through a process of analysis, experimentation, guessing and crossword puzzle-like assembly. Importantly, as the informal concepts are discovered and interrelated concept by concept, they are simultaneously associated with or assigned to the specific implementation structures within the program (and its operational context) that are the concrete instances of those concepts. The problem of discovering these human oriented concepts and assigning them to their implementation instances within a program is the *concept assignment problem* [4] and we address this problem in this paper.

2. The concept assignment problem

2.1. Programming Oriented Concepts vs. Human Oriented Concepts

A central hypothesis of this paper is that a parsing-oriented recognition model based on formal, predominately structural patterns of programming language features is necessary but insufficient for the general concept assignment problem. While parsing-oriented recognition schemes certainly play a role in program understanding, the signatures of most human oriented concepts are not constrained in ways that are convenient for parsing technologies. (See Sidebar on Automatic Concept Recognition) So there is more to program understanding than parsing. In particular, there is the general concept assignment problem, which requires a different approach.

More specifically, parsing technologies lend themselves nicely to the recognition of programming

oriented concepts (e.g., numerical integration, searches, sorts, structure transformations, etc.), because they are easily understood almost completely in terms of the patterns of their algorithms (i.e., numerical computation and data manipulation steps).

On the other hand, human oriented concepts such as *acquire target* or *reserve airplane seat* are decoupled from the formal patterns of their algorithms because they involve an arbitrary semantic mapping from operations expressed on numbers and data structures to computational intentions expressed in terms of domain concepts (e.g, a target or a seat). There is no algorithm (or, equivalently, no set of inference rules) that allow us recognize these concepts with complete confidence.

Is this difference just a manifestation of a *layers of abstraction* model, in which the higher level abstractions are defined in terms of the lower level abstractions? Can we just write deterministic rules relating the layers? Observations of humans trying to understand programs suggest that this is not the case. It appears that there is truly a paradigm shift between programming oriented and human oriented concepts. There is a change both in the kind of features that must be used to recognize the two kinds of concepts and the nature of processing required. Programming oriented concepts are signaled by the formal features of the programming language or other features that can be deductively or algorithmically derived from those features (e.g., variable liveness or data flow properties) while human concept recognition appears to additionally use informal tokens, require plausible reasoning and rely heavily on *a priori* knowledge from the specific domains. Thus, concept assignment is more like a decryption problem than a parsing problem.

In the remainder of the paper, we will give an example of this paradigm shift, in which we use *a priori* knowledge to drive the assignment of human oriented concepts and focus upon how tools, both naive and intelligent, can aid in that process.

3. An Example

In trying to assign concepts to code, one has two general tasks:

1) identify which entities and relations are really important, and

2) assign them to known (or newly discovered) domain concepts and relations.

The first task relies heavily on generic formal information (e.g., data structures, functions, calling relations, etc.) plus some informal information such as grouping and association clues. The second task relies more heavily on domain knowledge, e.g., knowledge of the problem domain entities and typical program architectures.

We consider the example C definitions in Figure 1 to see how we can identify concepts in code. The example is taken from a multi-tasking window system [1] written in C. These definitions constitute the set of data items necessary to handle breakpoint processing within a debugger. We will examine what can be plausibly inferred about this set of statements without any knowledge of the application domain context (i.e., task 1) and then what additional knowledge can plausibly be inferred given knowledge of the application domain context (i.e., task 2).

```
                                        <BLANK LINE>
unsigned char brkpts [MAXPROCS] [MAXBRKS];  /*Bytes to be restored at bkpts*/
unsigned char *brkat [MAXPROCS] [MAXBRKS];  /*Locations of set break points*/
unsigned int nbrkpts [MAXPROCS];            /*Number of breakpoints set for a process*/
int breakpoint;                             /* No of task hitting breakpoint*/
unsigned int breakcs, breakip;              /*Address of breakpoint*/
unsigned int breakflags;                    /*Flags register value at breakpoint*/
unsigned int breakss, breaksp;              /*Top of stack within breaker routine.
                                              Points to saved registers.*/
unsigned int current_ip, current_cs;        /*Current instruction address*/
                                        <BLANK LINE>
```

Figure 1 : A Code Example That Illustrates Data Grouping

For task 1, we use generic knowledge to infer that these statements are related to each other in some non-casual way, because

1) they are grouped together (*proximity*),

2) *bracketed* with blank lines,

3) exhibit a strong surface similarity among many of the formal and informal tokens (e.g., breakpoint, brkpts, breakcs, etc.), and

4) exhibit coupling via common tokens among several definitions (e.g., coupling via MAXPROCS and MAXBRKS).

Based on these features, we can tentatively assign the generic concept **data-group** to them, indicating that taken as a set, they are likely to be an instance of some (currently unknown) application domain data concept. Further, we expect that this data-group concept is a composite of some set of strongly related, detailed data subcomponents that are signaled by individual programming language tokens defined in the example. Presumably, at some time during the recognition process, the specifics of which particular application data concept we assign will be (plausibly) inferred from accumulated evidence.

For task 2, we assign the data-group and its subcomponents to domain specific concepts, utilizing *a priori* domain specific knowledge such as illustrated informally in Figure 2. The file drawers represent data stores, the ellipses functions, the arrows data/control flows and the text blocks other concepts such as debugging events. This is a fuzzy model, in that all concepts and relationships are weakly constrained, thereby allowing the model to cover a wide variety of

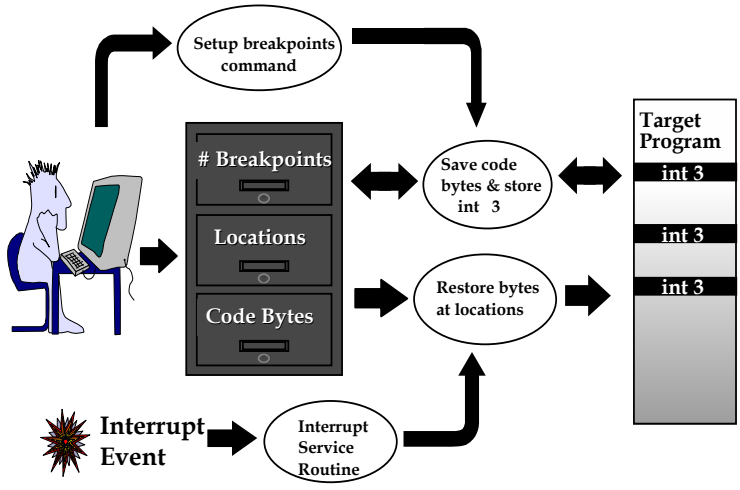

Figure 2 : A Model of Breakpoint Processing in Debuggers

concrete designs. We believe that a person with expertise in breakpoint processing must possess a model similar to this.

This model expresses one way in which debuggers typically handle breakpoints. That is, when the user asks for a breakpoint to be setup at a specific address, the original code at that address is saved in the debugger's data area and then it is replaced by code that will generate an interrupt when executed. That interrupt is how the debugger gets control back from the program being debugged (i.e., the target program). Immediately after regaining control, the debugger replaces the interrupt command byte with the original target program code, thereby returning the target program to its original form. At this point, the user would see exactly the same code as he originally wrote, which is what he expects.

How might a knowledgeable user relate this model to specific instances of the concepts in a program under analysis? What features might he use to make the concept assignments? Let us start with the recognition of the data store concepts (e.g., the *Locations* of breakpoints concept.)

Features that suggest concept assignments are:

1) natural language token meanings,

2) occurrences of closely associated concepts,

3) individual relations paralleling those in the model, and

4) the overall pattern of relationships in the model.

We illustrate each such feature in our example.

Certain **natural language tokens** -- words, phrases and abbreviations -- are features of (i.e., signal a likely reference to) the breakpoint-data concept (e.g., "breakpoint," "brkpts," and "brkat"), while others signal possible references to **concepts** that are closely **associated** with the breakpoint-data concept (e.g, the concepts address, registers, instruction, process and task). Finding evidence of these associated concepts adds evidence to the possibility that "breakpoint," "brkpts," "brkat" and so forth are indeed signaling a reference to the concept breakpoint-data.

Further evidence might be provided by the used_by **relations** between these data items and some previously assigned breakpoint processing function(s) (e.g., some known breakpoint processing function that uses brkpts, breakpoint, brkat or nbrkpts). For example, the user might already know about:

- bpint3, which handles the actual breakpoint interrupt;
- set_breaks & set_brkpt, which together replace bytes of target program code with hardware interrupt code bytes (i.e., breakpoint interrupt bytes) and save the original code bytes in the table brkpts and their addresses in brkat; or
- restore_breaks and restore_brkpt, which together replace the hardware interrupt code bytes with the code bytes that were originally in the target program before the breakpoints were set.

If the user has already proposed concept assignments to any of these functions (e.g., bpint3), then these concept assignments add weight to the evolving assignments associated with the data-group. On the other hand, the concept assignment could occur in the reverse order with breakpoint-data concept assigned first. In this case, association of the breakpoint-data concept with this data-group would serve as evidence for the subsequent concept assignments of bpint3, set_breaks, restore_breaks and so forth.

4. Concept Assignment Tools and Scenarios

4.1. Automated Assistance

Based upon our hypothesis about the underlying nature of the concept assignment problem, we have built a Design Recovery system called DESIRE [2,3] that is designed to be a program understanding assistant. DESIRE contains both naive and intelligent facilities to assist the user in attacking the concept assignment problem. The naive assistant facilities assume that the user is the intelligent agent and provide simple but computationally intensive services to support that intelligence.

The intelligent assistant facilities include a Prolog-based inference engine and a knowledge-based pattern recognizer called DM-TAO (Domain Model - The Adaptive Observer). These are more experimental and attempt to provide a limited amount of intelligent assistance in assigning concepts.

In this section, we will use scenarios to examine how such assistant tools can be (and have been) used to foster, simplify and accelerate the concept assignments in the previous example.

4.2. Scenario 1: Suggestive Data Names as First Clue

In this scenario, we suppose that a user is browsing the global data of some unfamiliar program and discovers the breakpoint data group of Figure 1. Let us further assume that this user has the domain knowledge that is illustrated in Figure 2. Under this scenario, the names "brkpts", "brkat" and "nbrkpts" along with their associated comments should suggest candidate concept assignments. In particular, brkpts is a potential instance for the *Code bytes* data store, brkat for the *Locations* data store and nbrkpts for the *# Breakpoints* data store.

The next logical step is to explore the functions that use these globals to try to identify the functional units *Save code bytes ...* and *Restore bytes ...* . Our user forms a query that asks for a Germ[1] browser view of all of the functions that use these global variables along with all of the call chains to these functions, resulting in the view shown in Figure 3.

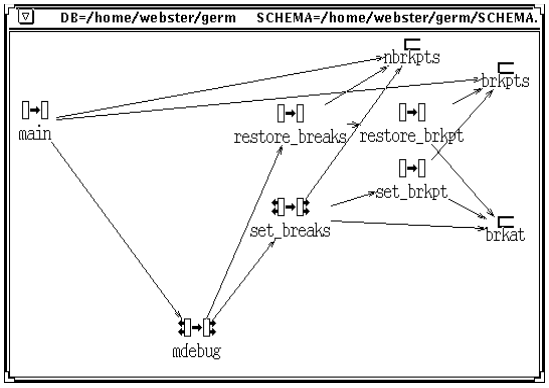

Figure 3 : Germ View of Use/Call Graph

These results reveal several strong candidates (set_brkpt, set_breaks, restore_brkpt and restore_breaks) for assignment to the save/set and restore concepts. He would now examine the source code to verify these tentative assignments and discovery that the evidence is strong enough to assign the two "set" routines to the *Save code bytes ...* concept and the two "restore" routines to the *Restore bytes ...* concept.

[1] Germ (Graphical Entity-Relation Modeller) is a generalized schema driven viewer with a great deal of hypertext functionality.

However, he is still in the dark about the breakpoint *Interrupt Service Routine* and the S*etup...* concept, i.e., the user-driven interface function that triggers the saving of the breakpoints. Since interrupt service routines are invoked by the hardware, it would not have turned up in the call chains. But interrupt routines do communicate with the rest of their application via global data. Further, our target routine will be related (indirectly, perhaps) to the interesting functions and global data that we have discovered so far. Thus, our user needs a way to search for global variables and functions loosely related to the current set of interesting functions and data. In DESIRE, this is accomplished by requesting a program slice[2] [6] that is based on the set of currently interesting program entities.

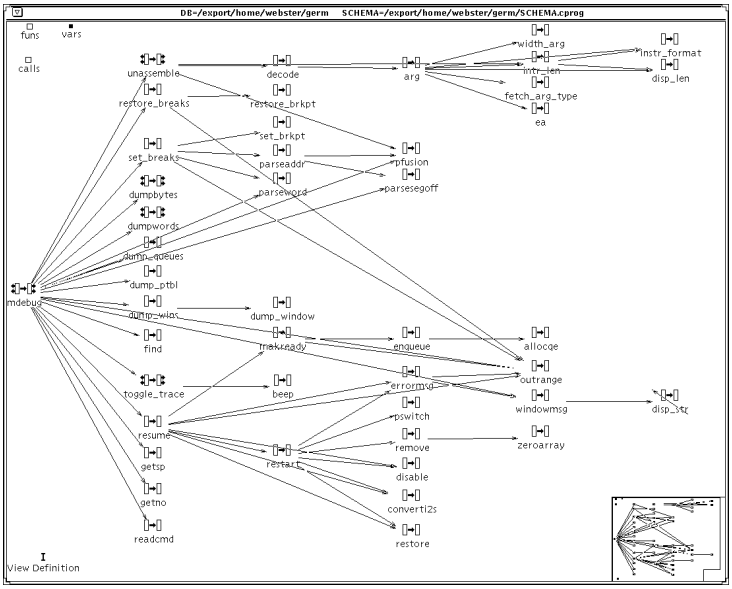

Figure 5 : Germ Browser Call Graph

DESIRE's Slicer does more than generate static views. It is a highly interactive tool that allows slices to be rapidly generated, extended, contracted and shifted based on a (typically shifting) set of currently interesting program entities called the interest set. It also includes a powerful operations for finding and combining interest sets.

In our example, the user might start with an interest set that includes the functions and global data so far assigned (i.e., restore_brkpt, restore_breaks, set_brkpt, set_breaks, nbrkpts, bkpts and brkat) and generate a slice based on these interests. Figure 4 shows part of the slice generated.

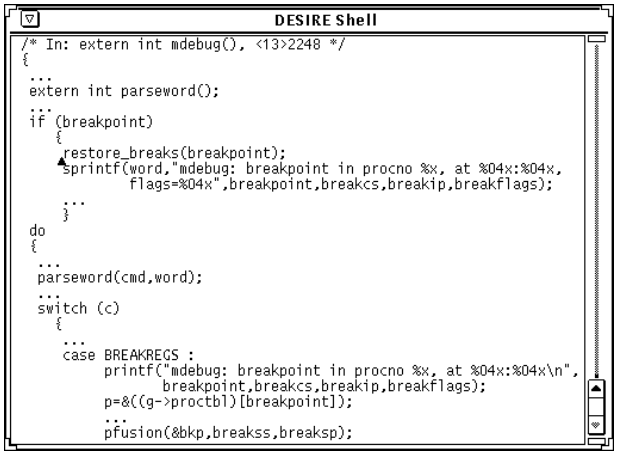

Figure 4 : Slicer's View of Part of mdebug Code

The slice introduces several new global variables because of the conditional branch that leads to the call to restore_breaks in mdebug. And all of these new global variables play a part in breakpoint processing. The flag *breakpoint* triggers the operation that restores the code bytes (i.e., *Restore...* concept) and the others (e.g., *breakcs, breakip* and *breakflags*) are part of the breakpoint's state. Inclusion of these variables in the slice, will also bring in bpint3 -- the breakpoint interrupt service routine -- because it uses these global variables to communicate with the main part of the debugger.

Elsewhere in mdebug (not shown in diagram), the user finds the code that calls set_breaks, and it is embedded within logic that interprets the user's debug commands. That is, mdebug is the assignment for the *Setup breakpoint command* concept. With this discovery, all of the key concepts have been assigned to specific program concepts thereby, providing a framework for further detailed analysis of the code, involving human interpretation.

[2]Roughly speaking, a slice for a variable is all of the statements that affect the value of the variable. There are several variations on this available.

4.3. Scenario 2: Patterns of Relationships as First Clue

Another approach to program analysis is to try to identify the clusters of related functions and data that form an abstract overview of the program. We call these clusters *modules*, to distinguish them from files, classes, objects, or other formal programming language structures.

How might one go about trying to discover such a framework in a language such as C? Much of DESIRE's tool set and methodology is designed to support the identification of such modules. Sometimes module clusters depend upon domain specific knowledge but often the module structures are revealed by more generic program features, such as

- Functions that are coupled by shared global variables, or
- Functions that are coupled by shared control paths.

Suppose that our user is searching for functional clusters based on shared control paths, that is a set of functions that are tightly bound because all call paths to them contain a single function, called the *dominator*. Our debugging example contains just such a cluster where the dominator is "mdebug." Why might our user suspect that this is a cluster?

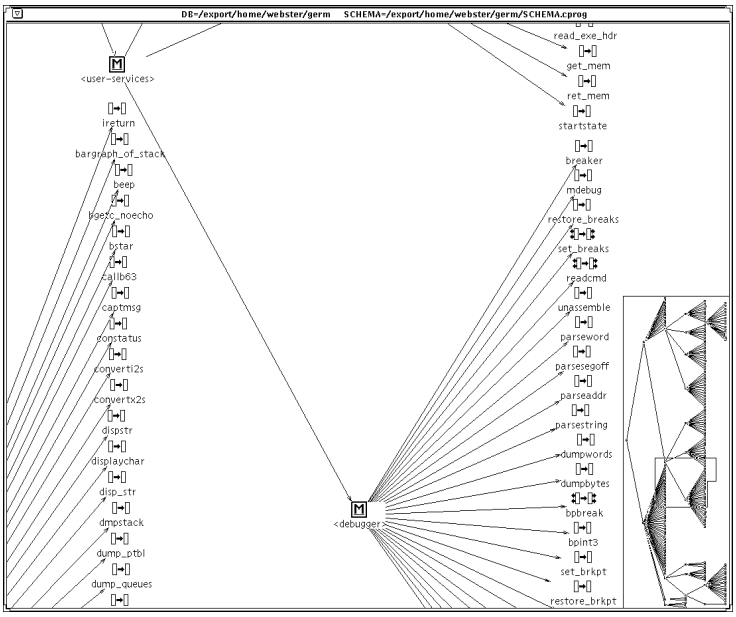

Figure 7: Module View of the System

Perhaps he notices a suggestive call graph pattern of functions that appear connectively isolated except for a rich set of connections to mdebug. (See Figure 5.) So, our user runs a cluster analysis with mdebug as the dominator. The results are shown in Figure 6.

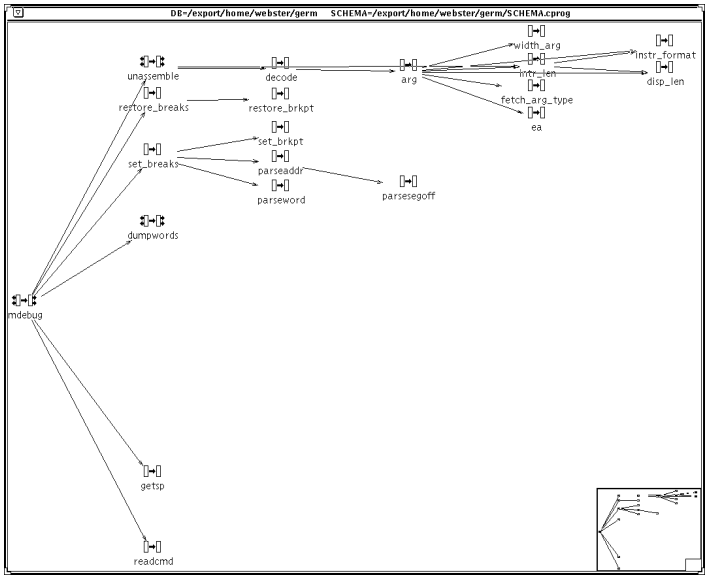

Figure 6 : Results of Cluster Analysis

The functions found include the set and restore functions from scenario 1, but also a number of functions involved in unassembling machine instructions (e.g., unassemble and decode), another set for reading and parsing user commands (e.g., readcmd and parseaddr) and others for dumping information (e.g., dumpwords). Further exploration will suggest additional candidates for inclusion based on functions that are conceptually related to the debugger.

At this stage, the user asks that this clustering relationship be recorded as a (new) module and an aggregate node is created in the DB. This new module node groups these functions so that they can be dealt with as a unit. Typically, the user will want to simplify some other

graphical view, so he would collapse (i.e., hide) all of these functions temporarily inside this new module node.

The user could proceed with other cluster analyses and eventually assign each function to some module. This allows him to get an module-based overview of the system. See Figure 7. These cluster results can be used in other tools -- the browser, query engine or the Slicer.

It should be clear from these scenarios that concept assignment benefits from a wide variety of naive tools for viewing, analysis and query. The detailed nature and usage of these tools are heavily influenced by the style of the investigators. However, the central invariant requirement is that the tools provide the mechanism for creating opportunistic associations and juxtapositions of information. Now, let us show how it is possible for the machine to play a more intelligent support role.

4.4. Scenario 3: Intelligent Agent Provides First Clue

Another approach would be for our user to ask DM-TAO -- DESIRE's experimental intelligent assistant for concept assignment -- to scan the code and present a list of candidate concepts based on the knowledge represented in its domain model (DM) knowledge. The results are used to glean a rough sense of the conceptual highlights of the code being studied or to serve as focal points for further investigation using the naive tools described in earlier sections.

The current version of DM-TAO can provide several kinds of insights into the source code:

- **Conceptual Highlights:** Look for all instances that correspond to any concept in the DM;
- **Conceptual grep:** Look for instances of a user-specified concept; and
- **Identification:** Propose a concept assignment for the currently selected code.

In our example, the user might start with a search of type 1 to perform a broad sweep of the code looking for important concepts. This will find **breakpoint-data**, DM's name for the model shown in Figure 2. The user could then ask to see the specific code associated with that concept and TAO would present the code from Figure 1 in a window. At this point, the user may need to understand the **breakpoint-data** concept in greater detail and so he selects the line in which brkat is declared and asks TAO to suggest a concept assignment for the selection (a type 3 query). As shown in Figure 8, TAO infers that the selection is an instance of the *breakpoint-location* concept, which is the DM's internal name for the *Locations* of breakpoints concept. This provides the user a place to start further analysis.

How does DM-TAO accomplish its assignments? The distinctiveness of DM-TAO and the problems that it attacks merit some elaboration. It uses the DM to drive a connectionist-based inference engine (TAO), similar to [5]. The DM is built as a semantic/connectionist hybrid network in which each domain concept (e.g., *Locations* of breakpoints) is represented as a node and the relationships between nodes are represented as explicit links (e.g., *Save code bytes* and *Locations* of breakpoints are related via a *uses* link). There are a variety of network node types: concept node, feature node, term node, syntax node etc., depending on the information being represented. The nodes are grouped together into layers. The feature, term and syntax nodes form the input layer of the network, while the concept nodes are loosely organized at different levels of abstraction, generally reflecting the conceptual infrastructure of the domain model. The different inter-concept relationships are represented by corresponding inter-node link types. Every link in the system has a real-valued weight associated with it, quantifying the strength of the relationship between the two nodes connected by it.

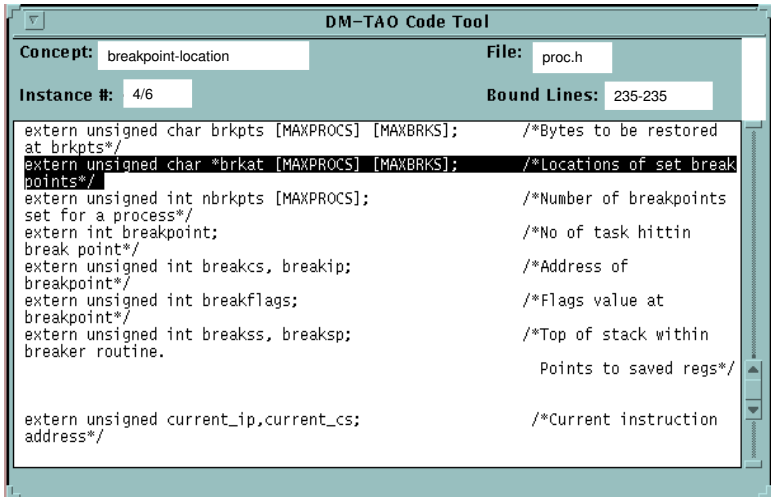

Figure 8: DM-TAO Suggests Assignment

The nodes serve as the processing units of the network and generate appropriate signal strengths or activation levels as a nonlinear function of the input. For most nodes (except the input layer), the input signal is a function of the activations generated by the connected nodes in the previous layer modulated by the weight on the connecting link. Nodes in the input layer are directly driven by the actions of a feature-extractor which extracts features such as syntax, lexical clues, clustering clues etc. Their activation level is a function of the number of corresponding clues found in the current target code segment, the degree of the match, and the activation history of related feature nodes. The signals generated in the input layer are propagated throughout the network via a controlled spreading activation process, which continues until the concept nodes compute their activation levels. If the computed output of a concept node is higher than a certain value - called the recognition threshold, then the domain concept represented by that concept node is predicted to be present in the corresponding section of code from which the relevant clues were extracted.

The accuracy of prediction of the network is a function of the weights distributed on it's links. The system adapts it's response via a 'training' process, which modulates these weights according to certain rules to obtain an optimal distribution. In DM-TAO, the training is effected in two stages: 1) The network is initially primed with *a priori* knowledge from the domain model regarding the degree of the association between two connected concepts (a qualitative assessment of low, medium or high provided by the domain builder). 2) The network weights are adjusted in a performance driven manner using qualitative relevance feedback from the user regarding the validity of the tentative concept assignments made by the system.

While DM-TAO has shown promise, it is still evolving and very much a research prototype.

5. Evaluation of DESIRE

In order to be credible, the evaluation of any system meant to assist a user in understanding real programs should be performed in a real-world context. Consequently, the testing and evaluation of DESIRE has always been done with real users. Even though all of the tools discussed here are experimental prototypes, they have been in use on real, large-scale programs (of up to 220 KLOC) since 1989 by a number of different users in several companies. DM-TAO is the one exception. It is still a research prototype that we have not yet released for use outside the lab. We feel that the result is better because of this approach.

DESIRE was first released to selected users in several companies in the spring of 1989. By 1992, it had been installed at more than a dozen sites in seven companies. The users are what we would characterize as early adopters and for the most part are quite self sufficient. However, there was still a fairly heavy interaction with the users. A dozen or so sites is about the limit that a small research group can handle without impeding research progress.

To date, the use of DESIRE has fallen primarily into two classes: 1) exploration for debugging or porting and 2) documentation for understanding and reporting. The most popular tools for exploration are the Slicer, the generic query system and the Prolog based analysis system. For documentation, Germ is the hands down winner. It is often used for reporting passive, artfully tailored views of program structures for publication or understanding.

DM-TAO is nearly complete but is still missing several key facilities necessary for doing large-scale validation experiments. Consequently, we have been limited to small experiments that required a good deal of manual labor. These experiments show promise but not yet definitive.

Even though it has some of the weaknesses of a research prototype, DESIRE has been used to do real work.

6. Conclusions

Since the concept assignment problem is an obviously hard problem, automation of even a small portion of it requires architectures that process a range of information types varying from formal to informal such that the information inferred from the informal can improve the ability to infer information from the formal and visa versa. Further, it seems clear from our analysis of example code that much understanding relies strongly, though not exclusively, on plausible inference. Finally, we conclude that deep understanding relies on an *a priori* knowledge base that is rich with expectations about the problem domain and the typical architectures.

We are encouraged by the preliminary results of DM-TAO. While we believe that the concept assignment problem will probably never be completely automated, some useful automation is possible. We believe that by incorporating those parts that we can automate into mixed-initiative systems in which the software engineer provides those elements that are beyond automation, it is possible to significantly accelerate and simplify the understanding of programs.

REFERENCES

1. Ted J. Biggerstaff, Systems Software Tools, Prentice-Hall (1986).

2. Ted J. Biggerstaff, "Design Recovery for Reuse and Maintenance" IEEE Computer, Vol. 22, No. 7, (July, 1989), pp. 36-49.

3. Ted J. Biggerstaff, Josiah Hoskins and Dallas Webster, "DESIRE: A System for Design Recovery," MCC Technical Memo STP-081-89, (April, 1989).

4. Ted J. Biggerstaff, Bharat Mitbander and Dallas Webster, "The Concept Assignment Problem in Program Understanding, ICSE, Baltimore, MD (May, 1993).

5. Jerome A. Feldman, Mark A. Fanty, Nigel H. Goddard and Kenton J. Lynne, "Computing with Structured Connectionist Networks," CACM Vol 31, No. 2, (February, 1988).

6. M. Weiser, "Program Slicing," IEEE TSE, Vol 10, (1984), pp 352-357.

Glossary

Domain Model - A knowledge base that defines concepts in a specific application domain (e.g., debuggers) as a set of entities and all of their interrelationships (e.g., the *Uses* relationship between the entities *Locations of breakpoints* and *Save code bytes* entities).

Dominator - A procedure or function f is the dominator of another procedure or function g if all call paths to g go through f.

Parsing Oriented Recognition Model - A recognition strategy that uses of a finite set of pattern templates each of which specifies a concept occurrance as a set of features. This is a recursive process in which the simplest, most elemental concepts are recognized first and then these concepts become features of larger-grained, conposite concepts.

Recognition Model - The method or architecture chosen to perform recognition.

Signature - The set of features (e.g., syntax, semantic, graphical, etc.) that signal the occurance of a specific concept pattern.

Program Slice - A program slice with respect to a specific program variable reference is all statements of the program that affect the value of that variable at that location.

Sidebar 1:

Automatic Concept Recognition

Concept assignment is a process of recognizing concepts within a computer program -- which includes all artifactual information associated with the code -- and building up an "understanding" or model of the program by relating the recognized concepts to portions of the program, to its operational context and to one another. One of the simplest operational models for the concept recognition and understanding process is to view it as a parsing process[1,2]. In this view, any given concept can be recognized from a specific signature (i.e., some pattern of features) within the target program. Indeed, many basic Computer Science algorithms such as *quicksort* are amenable to this process. The recognizer program uses a finite set of pattern templates that recognize the concept signatures by a parsing process, where the simplest, most elemental concepts are recognized first and then these concepts become features of larger-grained, composite concepts. A degenerate case of this recognition process is the familiar process of parsing programming languages for compilation.

These patterns typically rely almost completely on the formal, structure-oriented patterns of features, which is largely a result of the nature of the technology (namely, parsing technology) that is conveniently available to attack this problem. For parsing technologies to be effective, they rely heavily upon the premise that the concepts to be recognized are completely and (mostly) unambiguously determined by the formal, structural features of the entity being parsed and that these features are contextually quite local (e.g., as in context free languages).

References:

1. Mehdi T. Harandi and Jim Q. Ning, "Knowledge-Based Program Analysis," IEEE Software, Vol. 7, No. 1, (January, 1990), pp. 74-81.
2. Charles E. Rich and Linda M. Wills, "Recognizing a Program's Design: A Graph-Parsing Approach," IEEE Software, Vol. 7, No. 1, (January, 1990), pp. 82-89

Sidebar 2:

Related Research and Technology

There are a variety of technologies that address facets of the program understanding problem. The approaches taken and facilities included vary widely based on the research or technology purpose. A few broad (overlapping) categories that are relevant to program understanding are:

Maintenance and Re-engineering: The forces of change (e.g., computer "downsizing") are resulting in increased automation supporting program maintenance and re-engineering. These tools are variously focused on program reorganizing [7,10], program porting, or database re-engineering [4].

Reusable Component Recovery: Closely related to maintenance tools are those aimed at extracting reusable information from existing code, either in the form of executable components or non-executable business rules. [6]

Program Analysis and Development Aids: The development of large-scale systems requires increasingly greater levels of tools support for the programmer:

- Search, extraction and condensation of *explicit, static, and often distributed* program information, such as provided by query systems [1], program slicers, language-aware editors, etc.,
- Computation of *implicit* program information such as provided by module groupings [9] or data flow[2, 8], and
- Generator-based tools with strongly domain-oriented visual metaphors, clip-art assembly methods and hypermedia-like navigational aids [5].

Documentation and Understanding Aids: Documentation tools produce publication-oriented projections of *concrete* program information (e.g., browser views and other diagrammatic descriptions) as well as more *abstracted* views such as CASE-oriented design views. In addition, expert systems that can answer a limited class of questions about a target program [3] are beginning to emerge.

Also see sidebar titled *Automatic Concept Recognition*.

References:

1. Yih Farn Chen, Michael Y. Nishimoto, and C.V. Ramamoorthy, "The C Information Abstraction System", IEEE TSE, Vol. 16, No. 3 (March 1990), pp. 325-34.
2. Gerardo Canfora, Aniello Cimitile and Ugo de Carlini, "A Logic-Based Approach to Reverse Engineering Tools Production," IEEE TSE, Vol. 18, No. 12 (December, 1992).
3. Premkumar Devanbu, Ronald J. Brachman, Peter G. Selfridge, and Bruce W. Ballard, "LaSSIE: a Knowledge-based Software Information System," Proceedings of the 12th International Conference on SoftwareEngineering, Nice, France (March, 1990).
4. J-L. Hainaut, M. Chandelon, C. Tonneau and M. Joris, "Contribution to a Theory of Database Reverse Engineering," Proceedings of the Working Conference on Reverse Engineering, Baltimore, MD (May, 1993).
5. Microsoft Visual Basic™ Programmer's Guide, Version 3.0, Microsoft Corporation (1993).
6. Jim Ning, Andre Engberts and Wojtek Kozaczynski, "Recovering Reusable Components from Legacy Systems by Program Segmentation," Proceedings of the Working Conference on Reverse Engineering, Baltimore, MD (May, 1993).
7. Philip Newcomb and Lawrence Markosian, "Automating the Modularization of Large COBOL Programs: Application of an Enabling Technology for Reengineering," Proceedings of the Working Conference on Reverse Engineering, Baltimore, MD (May, 1993).
8. Charles Rich and Richard C. Waters, "The Programmers's Apprentice: A Research Overview," IEEE Computer, Vol 21, No. 11 (November, 1988).
9. Robert W. Schwanke, "An Intelligent Tool for Re-engineering Software Modularity", Proc. 13th ICSE, May 13-15, 1991, Austin, TX, pp. 83-92.
10. M. P. Ward and K. H. Bennett, "A Practical Program Transformation System for Reverse Engineering," Proceedings of the Working Conference on Reverse Engineering, Baltimore, MD (May, 1993).

Ted Biggerstaff is Research Program Manager at Microsoft Research. His interests include software reuse, design recovery, reverse engineering, re-engineering, transformational programming, and neural networks. He can be reached at Microsoft Research, Mail Stop 9S/1032, One Microsoft Way, Redmond, Washington, 98052-6399 or on the Internet at tedb@microsoft.com.

Bharat Mitbander is Member of Technical Staff at Microelectronics and Computer Technology Corporation (MCC). His research interests include design information recovery, application dredging, knowledge discovery, neural networks and design automation. He can be reached at MCC, 3500 W. Balcones center Drive, Austin, TX 78759-5398 or via the Internet at mitbander@mcc.com.

Dallas Webster is Senior Member of Technical Staff at Microelectronics and Computer Technology Corporation (MCC). His research interests include design information recovery, application dredging, and knowledge acquisition, representation, and presentation. He can be reached at MCC, 3500 W. Balcones center Drive, Austin, TX 78759-5398 or via the Internet at webster@mcc.com.