The Concept Assignment Problem in Program Understanding

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 they are able to relate the structures of the program and its environment to their human oriented conceptual knowledge about the world. The problem of discovering individual human oriented concepts and assigning them to their implementation oriented counterparts for a given a program is the *concept assignment problem.* We will argue that the solution to this problem requires methods that have a strong plausible reasoning component. We will illustrate these ideas through example scenarios using an existing design recovery system called DESIRE. Finally, we will evaluate DESIRE based on its usage on real-world problems over the years.

Keywords -- reverse engineering, slicing, knowledge base, domain, connectionist, concept recognition, plausible reasoning.

1. Human understanding and the concept assignment problem

A person understands a program when they are 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 in any but the most trivial way. That is, if one wants to create, maintain, explain, re-engineer, reuse or document a program, one must possess both forms of computational intent. What is more, one must understand the association between the formal and the informal expressions of computational intent.

If a person starts to build an understanding of a heretofore unknown 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* and this is the problem that we will address in this paper.

2. Approaches to the concept assignment problem

2.1. Recognizing Implied Concepts

Concept assignment is a process of recognizing concepts within a computer program and building up an "understanding" of the program by relating the recognized concepts to portions of the program, 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. In this view, any given concept instance, such as the *acquire-target* concept in a fire control program for a tank, can be recognized from a specific signature (i.e., some pattern of features) within the target program. 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 (i.e., 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 these features are contextually quite local (e.g., as in context free languages).

In contrast, our research makes a different assumption about the concepts to be recognized. It assumes that the formal, structural features play a lesser role in the recognition of concepts that are important for human understanding and further, that the patterns defining these important concepts are far more open to variation and ambiguity than can be naturally accommodated by parsing technology. This alters the model of concept recognition from one that is characterized as a recursive, algorithmic, deterministic and orderly building up of more abstract components out of less abstract components, to one that is characterized as a more opportunistic, non-deterministic and chaotic piecing together of evidence for a concept until some threshold of confidence is reached about its identity.

2.1.1. Programming Oriented Concepts vs. Human Oriented Concepts

The 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 mainly because the signatures of most human oriented concepts are not constrained in ways that are convenient for parsing technologies. This is not to say that parsing-oriented recognition schemes play no role in program understanding because they certainly do. We are simply saying that there is more to it than that. There is a part of the program understanding process (i.e., general concept assignment) that requires a different approach.

To make the case for this assumption, we need make the argument that a parsing model is not the right approach for this part of the recognition problem and to propose a substitute. We can make the case by comparing the class of concepts that can be easily recognized by a simple parsing model with the class of concepts that we have referred to as human oriented concepts. Let us call the first class programming oriented concepts and the second class human oriented concepts. The question that we shall examine in this section is whether these two classes are the same class and if not, how they differ. Table 1 compares these two classes of concepts with respect to a number of facets or dimensions and suggests that these are indeed different classes. Let us consider the nature of these two classes along these dimensions.

Domain Characterization: Programming oriented concepts (e.g., numerical integration, searches, sorts, structure transformations, etc.) are understood almost completely in terms of the patterns of their algorithms (i.e., numerical computation and data manipulation steps). Clearly, this characteristic makes them relatively easy to recognize by analyzing (i.e., parsing) these patterns.

It is much more of a leap to get from mathematical and data manipulation primitives to concepts such as *acquire target* or *reserve airplane seat*, since any such recognition process must 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 *a priori* connection between the domains of "mathematics and data manipulation", and any application domain such as "airline seat management." In short, once we get past a small domain of numerical computation and non-domain-specific data manipulation, there is no algorithm (or, equivalently, no set of inference rules) that allow us to compute this mapping with complete confidence. In moving from recognizing programming oriented concepts to recognizing human oriented concepts, we have moved from a problem that is amenable to algorithmic reasoning to a problem that requires plausible reasoning. In short, there is a large gap between the set of programming oriented primitives in a signature pattern and the concept expressing the computational intent in human oriented terms.

Table 1

Properties of Concept Types

Property	Programming Concepts	Human Concepts
Domain characterization	Numerical computation and data manipulation	Arbitrary domain concepts
Feature types	Formal elements -Language syntax and semantics -Data flow -Control flow -Deducible properties	Formal and informal -Natural language tokens -Proximity and grouping -Design conventions -Domain conventions -Previous solution states -Weight of evidence
Reasoning method	Deductive or algorithmic	Plausible or fuzzy reasoning
Uniqueness of solution	Unique or canonical	Multiple equivalent solutions
Precision	Precise	Approximate

Kinds of Features: The difference is once again clear if we look at the kinds of features that signal concepts in these two distinct domain classes. Programming oriented concepts by their definition 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).

On the other hand, given only formal features, it is often incredibly difficult to make the leap to domain concepts such as *reserve airplane seat* without some additional clues that relate the numbers and data structures to the application domain concepts. These additional features include natural language tokens and structure (e.g., in comments and identifier names); the proximity and grouping of statements and definitions (e.g., conceptually related definitions are often grouped together); design conventions (e.g., layering or object-oriented designs, which imply structural relationships that may or may not be expressible in the programming language but, like proximity, signal deeper domain relationships); application domain design conventions (e.g., process schedulers typically use data items from process tables); previous concept assignments (e.g., recognition of a process table enhances the ability to later recognize the scheduler that uses it); and the additive weight of differing kinds of evidence (e.g., proximity reinforces domain conventions, natural language token clues, and so forth). When observing a person puzzling out the understanding of a program, it is clear that features from these various classes are used in concert to make the (usually plausible) assignment of domain meaning to the formal structures within the program.

Reasoning Method: Different reasoning methods underlie these two concept types. Programming oriented concepts can be easily recognized by deductive or algorithmic processes whereas human oriented concepts seem to force plausible reasoning -- guessing based on natural language clues, the weight of accumulated evidence, heuristic rules of thumb, *a priori* knowledge of inter-concept constraints, and so forth. If one observes a person who is starting to understand a large program he has never seen, the first thing he does is read data names, function names and comments. Oral protocols reveal obvious streams of plausible reasoning using mostly the informal information from these clues.

Uniqueness: Parsing technology is biased toward unique or canonical solutions. If there are several equivalent conceptual forms possible for a signature, then the technology must provide a method for determining equality among them and this adds a significant level of computational complexity to the recognition system making it impractical for all but the smallest of problems. By contrast, human oriented concepts comfortably allow multiple, roughly equivalent representations for concepts (e.g., "seat reservation" is conceptually equivalent to "reserve airplane seat"). Further, the notion of two concepts being roughly equivalent is a fuzzy notion and appears to behave much like the human oriented concept recognition process itself. That is, we conclude concepts are equivalent based on the weight of evidence rather than an axiom-driven proof of equivalence.

In conclusion, we are led to the belief that human oriented concepts (which typically tend toward domain specificity) are different from the class of programming oriented concepts and are not easily recognized by simple, pattern driven parsing models. Further, we believe that the recognition process depends heavily upon *a priori* knowledge about the application domain, the domain entities that are typically important and the typical relationships among them. Thus, we are driven toward a processing model based on a fuzzy or plausible reasoning method, which is in turn supported by *a priori* knowledge of the application domain.

In the remainder of the paper, we will use an example of such *a priori* knowledge to examine how it might be used by a programmer who is trying to understand a program. We will 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) to identify what few entities and relations are really important out of the many in the code, and 2) to assign them to the known domain concepts and relations. The first task relies most heavily on generic information such as formal programming language structures (e.g., data structures, functions, calling relations, and so forth) as well as 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. The two tasks are complementary with each providing knowledge that simplifies the other.

Now let us look at an example, as depicted in Figure 1, and examine how we can identify important entities in code with the help of fuzzy recognition and plausible inference. The following set of statements is taken from a multi-tasking window system [1] written in C. Taken as a set, they constitute the set of data structures necessary to handle breakpoint processing within a built-in debugger subsystem. 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

In task 1, we can 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 application domain 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.

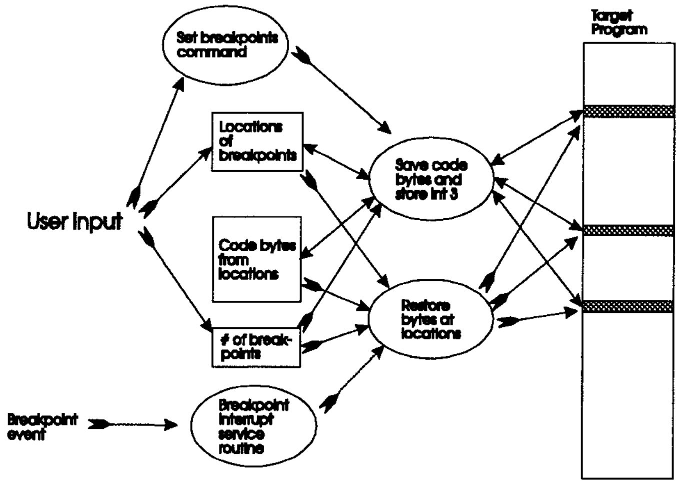

Set breakpoints command
Locations of breakpoints
Code bytes from locations
# of break-points
Save code bytes and store Int 3
Restore bytes at locations
Breakpoint interrupt service routine
User Input
Breakpoint event
Target Program

Figure 2 : A Model of Breakpoint Processing in Debuggers

In task 2 (assigning the data-group and its subcomponents to domain specific concepts), we utilize *a priori* domain specific knowledge such as illustrated informally in Figure 2. This is an informal model in which the boxes represent data stores, the ellipses functions, the arrows data flows and the text blocks other concepts such as debugging events. This is a fuzzy model in that all concepts are weakly constrained, thereby allowing the model to cover a wide variety of specific 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 set 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 code 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? For the sake of brevity, we will defer recognition of the functions until later in the paper and at this point, focus on the data store concepts (e.g., the *Location of breakpoints* concept.)

The features that suggest the concept assignments are: 1) natural language token meanings, 2) occurrence of closely associated concepts, 3) individual relations paralleling those in the model and 4) the overall pattern of relationships in the model. Examples of each such feature are seen in this example.

There are certain **natural language tokens** -- words, phrases and abbreviations -- that are features of (i.e., that signal a reference to) the breakpoint-data concept. For example, "breakpoint," "brkpts," "brkat" and so forth signal the likelihood (although not absolute certainty) of a reference to the concept breakpoint-data. Further, there are other natural language tokens that signal possible references to **associated concepts** that are likely to be found in a breakpoint-data instance (e.g, the concepts address, registers, instruction, process, task and so forth). 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 uses brkpts, breakpoint, brkat or nbrkpts). In this case, there are some such functions that the user might already know about, notably "bpint3," which handles the actual breakpoint interrupt; "set_breaks," which replaces bytes of target program code with hardware interrupt code bytes (i.e., breakpoint interrupt bytes) and saves the original code bytes in the table "brkpts" and their addresses in "brkat"; "restore_breaks," which replaces the hardware interrupt code bytes with the code bytes that were originally in the target program before the breakpoints were set; and several other functions.

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 concept assignments of "bpint3", "set_breaks" and "restore_breaks".

4. Concept Assignment Tools and Scenarios

4.1. Automated Assistance

Based upon the 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 two distinct kinds of facilities that aid the user in attacking the Concept Assignment problem: 1) naive assistant facilities, and 2) intelligent assistant facilities. The naive assistant facilities assume that the user is the intelligent agent and the naive facilities provide simple but computationally intensive services to support that intelligence.

The intelligent assistant facility, which is called DM--TAO (Domain Model -- The Adaptive Observer), is more experimental and attempts to provide a limited amount of intelligent assistance in performing concept assignment.

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

4.2. Scenario 1: Suggestive Data Names as First Clue

In this scenario, we will assume that a hypothetical user is browsing the global data of some program he has never seen and discovers the breakpoint data group shown earlier. 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 candidate instance for the *Code bytes from locations* data store, brkat is a candidate instance for the *Location of breakpoints* data store and nbrkpts is a candidate instance for the *# of breakpoints* data store.

The next logical step for our hypothetical user is to explore the functions that use these globals to try and identify the functional units that save/set and restore these code bytes. Let us say our user forms a query using DESIRE's Prolog-based facility to look for the functions that use these global variables. This is one of about half a dozen ways that he could discover this information using the various DESIRE facilities. The results reveal two strong candidates (set_brkpt and restore_brkpt) for assignment to the save/set and restore concepts. It also introduces a new function, mdebug, which the user will want to explore a little later. So, we assume that he puts mdebug on his agenda of code to be browsed and analyzed, and goes back to the save/set and restore concepts. While set_brkpt and restore_brkpt are strong candidates for the save/set and restore concepts, this might not be all of the story, so the user queries the system to determine what functions call set_brkpt and restore_brkpt. This results in the discovery of the two other functions, restore_breaks and set_breaks, which sound like they operate on a set of breakpoints whereas set_brkpt and restore_brkpt sound more like they operate on individual breakpoints. He probably would now look at the source code of restore_breaks, restore_brkpt, set_breaks and set_brkpt to verify that they do indeed perform the save/set and restore functions described in the breakpoint model, as indeed they do. At this point, the evidence is sufficient to make a pretty strong assignment of the pair restore_breaks and restore_brkpt to the restore concept and the pair set_breaks and set_brkpt to the save/set concept.

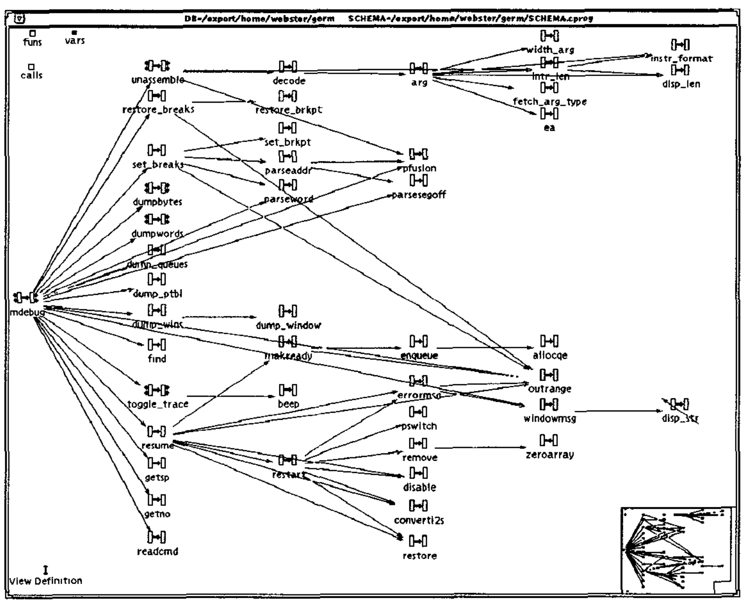


Figure 3 : Germ View of Call Graph

However, he is still in the dark about the breakpoint interrupt service routine and the function that requests the save/set function. He is likely to use the current assignments as the starting point and try to extend the context, expecting that he might find these concepts called from within some slightly expanded context. Starting with the set_breaks and restore_breaks functions and recursively looking up the call paths, he discovers that both are called by the function mdebug. Now it would seem like a good time to get an overview of the functions in the current context. Using a graphical browser called Germ, asks for a call graph that includes all of these functions starting at mdebug. This is shown in Figure 3.

This view introduces a number of new avenues (functions) to explore. In particular, there are a number functions that are doing some kind of parsing (e.g., parseword) and others that appear to perform dumping of various kinds of data (e.g. dumpbytes). These names trigger associations with other models of debugging functionality and our hypothetical user would probably add these to his agenda to be explored and analyzed later. But for the moment, he continues to add evidence to and fill out the concept assignments for the breakpoint model.

Our user notices mdebug and recalls that it calls set_breaks. A simple query reveals that mdebug is the only such routine that does. Therefore, mdebug is a candidate for the *Set breakpoints command* concept. A quick look at the code and in particular, at the point in mdebug's code where set_breaks is called verifies this concept assignment. Of course, mdebug does a great deal more than just call set_breaks, thereby playing the role of multiple concepts, but at the moment our user is focused on understanding the breakpoint model.

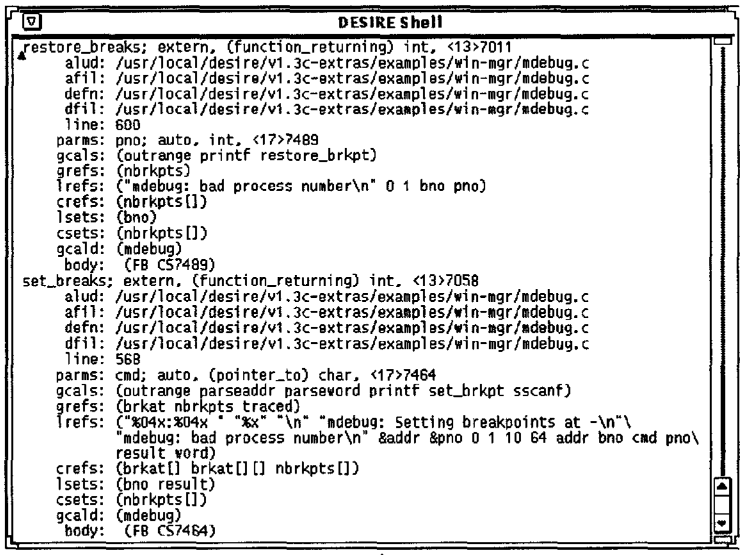
Now, the user might turn his attention to exploring the broader calling relationships in order to understand the wider context in which the breakpoint model occurs and hopefully discover the last unassigned concept, viz. the *Breakpoint interrupt service routine*. Where is it? Why has it not shown up? This is not surprising because interrupt service routines are not invoked explicitly by the user code so, we would not expect it to show up in a call graph view. The address of an interrupt service routine is usually stored in a table at a location known by the interrupt hardware and it is invoked by the hardware when an interrupt event with its number occurs. Nevertheless, such interrupt service routines do communicate with the rest of their program through global data. Therefore, our hypothetical user could use the set of global data identified so far to form a query that looks for functions that are not included in mdebug's call tree but do use some of the breakpoint global data. And this strategy will payoff for the user. He discovers a C function called bpint3 and by examining the code will verify that bpint3 is indeed the *Breakpoint interrupt service routine* concept sought.

The strategy that we have just used and the DESIRE facilities that we have illustrated are typical of those actually used by DESIRE users. Of course, there are many alternative routes and sub-routes we could have taken but we cannot explore all of them here. Nevertheless, we will use a several alternative strategies to illustrate other DESIRE facilities that can be brought to bear on the concept assignment problem.

4.3. Scenario 2: Suggestive Function Names as First Clue

Now, let us choose a different scenario in which different choices were made because different opportunities were noticed first. Let us suppose that the user chose to examine the functions in the target program first using any one of the several tools/views available. He could be examining the graphical call view in Germ or the text view in his favorite editor when he notices two functions whose names (restore_breaks and set_breaks) trigger an association with the breakpoint model. Examining the various text strings associated with the functions (e.g., the printf format string "mdebug: Setting breakpoints at -\n") supply additional evidence for this association. So, our user chooses to explore the context of these functions and chooses DESIRE's Slicer tool (which extends the slicing method of [18]) to do this exploration.

This Slicer is a tool that allows views to be rapidly generated, extended, contracted and shifted based on a set of program entities that are considered interesting



DESIRE Shell

```
restore_breaks; extern, (function_returning) int, <13>7011
     alud: /usr/local/desire/v1.3c-extras/examples/win-mgr/mdebug.c
     afil: /usr/local/desire/v1.3c-extras/examples/win-mgr/mdebug.c
     defn: /usr/local/desire/v1.3c-extras/examples/win-mgr/mdebug.c
     dfil: /usr/local/desire/v1.3c-extras/examples/win-mgr/mdebug.c
     line: 600
    parms: pno; auto, int, <17>7489
    gcals: (outrange printf restore_brkpt)
    grefs: (nbrkpts)
    lrefs: ("mdebug: bad process number\n" 0 1 bno pno)
    crefs: (nbrkpts[])
    lsets: (bno)
    csets: (nbrkpts[])
    gcald: (mdebug)
     body:  (FB CS7489)
set_breaks; extern, (function_returning) int, <13>7058
     alud: /usr/local/desire/v1.3c-extras/examples/win-mgr/mdebug.c
     afil: /usr/local/desire/v1.3c-extras/examples/win-mgr/mdebug.c
     defn: /usr/local/desire/v1.3c-extras/examples/win-mgr/mdebug.c
     dfil: /usr/local/desire/v1.3c-extras/examples/win-mgr/mdebug.c
     line: 568
    parms: cmd; auto, (pointer_to) char, <17>7464
    gcals: (outrange parseaddr parseword printf set_brkpt sscanf)
    grefs: (brkat nbrkpts traced)
    lrefs: ("%04x:%04x " "%x" "\n" "mdebug: Setting breakpoints at -\n"\
           "mdebug: bad process number\n" &addr &pno 0 1 10 64 addr bno cmd pno\
           result word)
    crefs: (brkat[] brkat[][] nbrkpts[])
    lsets: (bno result)
    csets: (nbrkpts[])
    gcald: (mdebug)
     body:  (FB CS7464)
```

Figure 4 : Slicer's DB View

at the current moment. This set is called the *interest* set[1]. The views that the interest set engenders are highly dynamic, quick and easy to change based on the user's evolving understanding of the context. The Slicer provides three ways to specify program entities for the interest set:

- Explicitly reference symbols by identifier name, which may produce a number of different instances of a symbol,
- Explicitly reference symbols by in internal name or oid (i.e., object id) which is unique for each symbol, or
- Implicitly reference a set of symbols (the Slicer's so called *collection*) by a formal specification of the set.

The third method of reference is the most powerful and often used. For example, it provides the ability to specify the set of all symbols that are **defined** (or **not**) in the **program** in a given **scope** that are of a particular **kind** (e.g., function or variable) and are in (or **not** in) a particular **relation** (e.g., calls, called_by, ref_by, set_by and so forth) with some other set of **objects** (e.g., interests, collection or other).

In our example, the user probably starts with an interest set that includes restore_breaks and set_breaks and asks for a DB (data base) view of the symbols, which reveals all of the relationships known by DESIRE (Figure 4). It shows all of the raw data information that is cached in DESIRE's internal data base for these functions, i.e., what functions call them, what functions they call, where they are defined, their internal oids, what globals they reference, what ones they set and so forth.

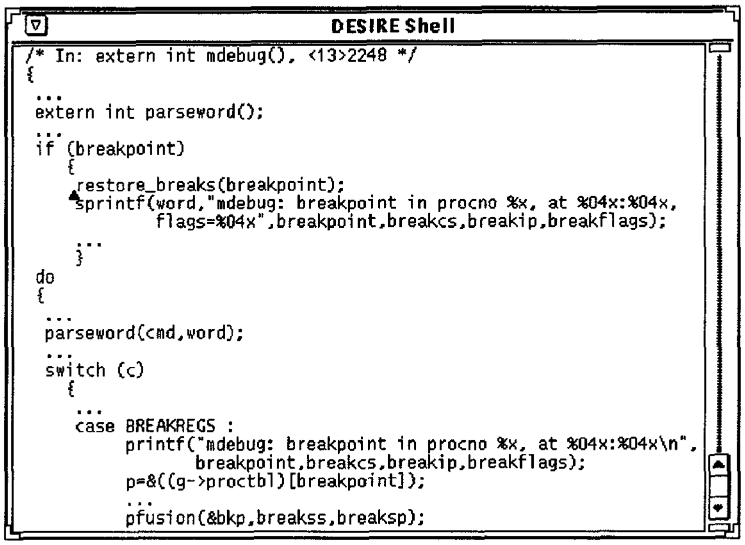
This opens up a number of new avenues for exploration. The user could explore the text file further with the editor or he could chose to explore the relations in the program. Deriving the relational structure of a program with a text editor, even one with good search and hypertext capabilities, is tiresome at best. So, we will assume that the user will specify the desired explorations symbolically to the Slicer and let it do the work and keep track of the growing context (via the interest and collection sets).

So, the user extends the interest set with the global data referenced (i.e., accessed or set by) any function in the interest set. This will add nbrkpts (number of breakpoints) and brkat (locations of breakpoints) to the interest set. At this point, we assume that he has made the correct concept assignments for these two global data variables and begins to wonder where the target program code bytes are kept. Maybe in a global variable used by one of the routines called by restore_breaks or set_breaks, so he adds this set of functions to the interest set resulting in restore_brkpt and set_brkpt (plus a couple of others) being added to

[1]There are really two sets, the *interests* and the *collection* that are calculated, recalculated, merged in various ways and manipulated by the slicer during an interactive exploration session.

the interest set. A DB view of these two routines reveals a new global variable (i.e., brkpts) and a quick look at its definition, which includes a revealing comment ("Bytes to be restored at bkpts") pretty much nails the concept assignment for brkpts.

Just as in the previous scenario, some concept assignments are still open and the strategy will be the same, expand the context and examine what new program entities get added to the interests. Our user does this by adding the functions that call the functions in the interest set, which adds mdebug and then he adds all global variables referenced by entities in the interest set. This adds breakpoint, breakcs and the rest of the data variables from Figure 1. The user will now ask for a usage slice of the program to get an integrated, contextual view of the all of the entities from the interest set. This view reconstructs the code by including all uses of the interests plus all of the control paths necessary to get to those usages. All other parts of the program are elided and are shown as "..." in the slice view[2]. Figure 5 illustrates a portion of a usage slice.



```
/* In: extern int mdebug(), <13>2248 */
{
 ...
 extern int parseword();
 ...
 if (breakpoint)
    {
     restore_breaks(breakpoint);
     sprintf(word,"mdebug: breakpoint in procno %x, at %04x:%04x,
             flags=%04x",breakpoint,breakcs,breakip,breakflags);
     ...
     }
 do
 {
  ...
  parseword(cmd,word);
  ...
  switch (c)
    {
     ...
     case BREAKREGS :
          printf("mdebug: breakpoint in procno %x, at %04x:%04x\n",
                 breakpoint,breakcs,breakip,breakflags);
          p=&((g->proctbl)[breakpoint]);
          ...
          pfusion(&bkp,breakss,breaksp);
```

Figure 5 : Slicer's View of Part of mdebug Code

Probably, the user would continue to explore this area of the program driven by other related models but these explorations would proceed much like this scenario. So, we will turn our attention to other approaches and strategies available to the DESIRE user.

[2]Actually, users typically ask for such slice views with each change of the interest set but for the sake of brevity, we have chosen to show only one example of this kind of view.

4.4. Scenario 3: Patterns of Relationships as First Clue

Another approach to program analysis is to try to identify the clusters of functions and data that appear to be closely related in order to form a structural framework on which to hang the details of the program. We call these clusters *modules*, not to be confused with files, objects, or other formal programming language structures. Object oriented languages include formalisms (i.e., the ability to specify classes) that allow such frameworks to be formally described but non-object oriented languages do not. So, how might one go about trying to discover such a framework in a language such as C?

Sometimes module groupings depend upon domain specific knowledge such as in the breakpoint model, but often the module structures are revealed by more generic program features. Sometimes the generic features of the program structure provide clues that a given set of functions and data are logically related. What are such generic features? Relations (e.g., calls or references) and the kind of program entity (e.g., function or data) are the two most obvious and useful kinds of features, and we will use these plus others to search for clues to our program's organizational structure.

So let us go through a scenario that starts by the user looking for program clusters that are suggested by generic clues and then using those clusters as the jumping off point for a more domain specific exploration to fill in the details. What kind of cluster criteria might our user bring to bear on this problem?

Some clusters reveal themselves because they consist of a set of functions that are cohesively bound to some set of global data that they management. For example, process tables typically have a set of functions that perform process management functions such as changing the state of a process, e.g., kill a process, suspend a process, start a process and so forth.

Alternatively, some clusters are characterized by a set of functions that are tightly bound because their call paths are dominated by a single function. That is, all functions in the set can be reached by call paths that contain the dominant function and by no paths

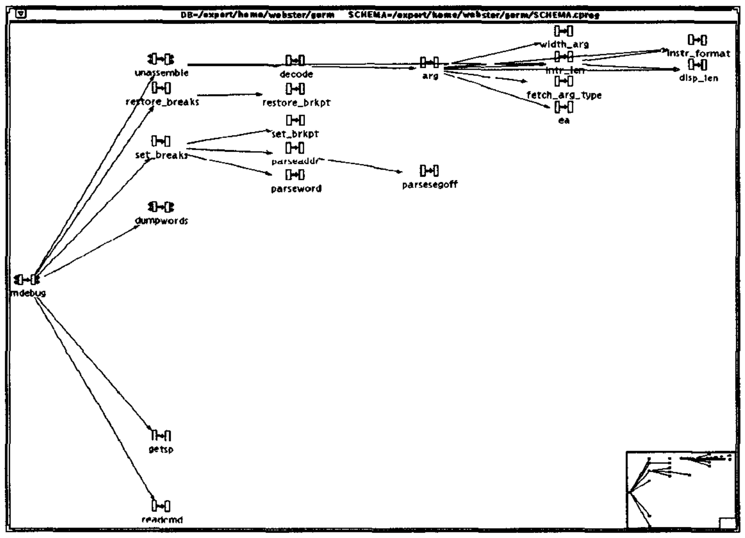


Figure 6: Cluster Analysis

that do not. Our debugging example considered earlier contains just such a cluster where the dominant function is "mdebug." All other functions in the group are called only on paths that go through mdebug. How might our user find this cluster?

There are a variety of ways but let us say the our user notices a suggestive pattern in the graphical browser where there are a little group of functions that appear connectively isolated except for a rich set of connections to mdebug. This is a candidate for a cohesive cluster but verification is needed because the eye cannot be sure. Therefore, our user brings up the cluster analysis tool and chooses to run a cluster analysis on mdebug. A Prolog program from the library runs and attempts to form a cohesive cluster with mdebug the dominant function. The results are shown in Figure 6. The analyzer verifies that the cluster noticed by the user is indeed a cohesive cluster. This graph contains all functions that can be reached on call paths through mdebug and only on such paths. Compare this diagram to Figure 3 and notice that various functions called by mdebug have been eliminate by the cluster analyzer (e.g., resume and dump_win). Obviously, these functions are called from other functions as well. The user will now have to decide whether some of the functions that were eliminated because of formal features should be added back in because semantically they really belong to this module or whether other functions not including in the call graph of mdebug and therefore, not found by the cluster analyzer should be included in this module. In this case, he decides no on the first issue, but later in his analysis he will discover several functions (e.g., bpint3, the breakpoint interrupt service routine) and edit them into the module. This is a typical scenario that includes a mix of automation and human intelligence.

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, which will then appear in the Germ graphical view. This new module node groups these functions so that they can be dealt with as an individual. If user then wants to simplify the graphical view, which is typical, he would collapse (i.e., hide) all of these functions temporarily inside this new module node.

Our user might use these results in Germ, Prolog or the Slicer to complete the exploration using the same strategies and tools shown in the other two scenarios. But eventually, the user would proceed with other cluster analyses and eventually assign all functions to some module thereby, allowing him to get an overview of the system, a so called *module* view. See Figure 7. The relational links in Figure 7 are subcomponent links rather than calls links. That is, each module contains a set of functions or other modules. This forms a tree structured overview of the target system.

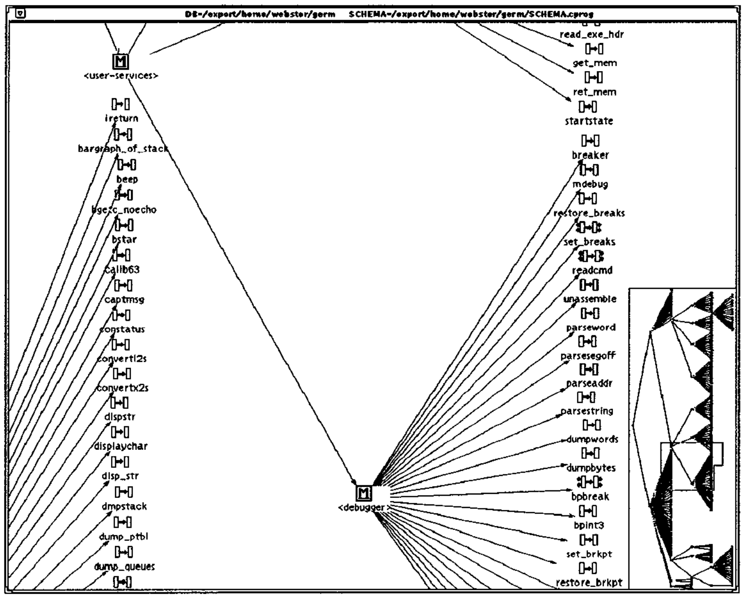


Figure 7: Module View of the System

Not shown in these scenarios are the results of the user's other explorations, which typically follow scenarios similar to scenario 1 and 2. In the course of such explorations, the user will build up many notes, crosslinks and summaries that record the results of his explorations and these will be all available as the starting point for future analyses.

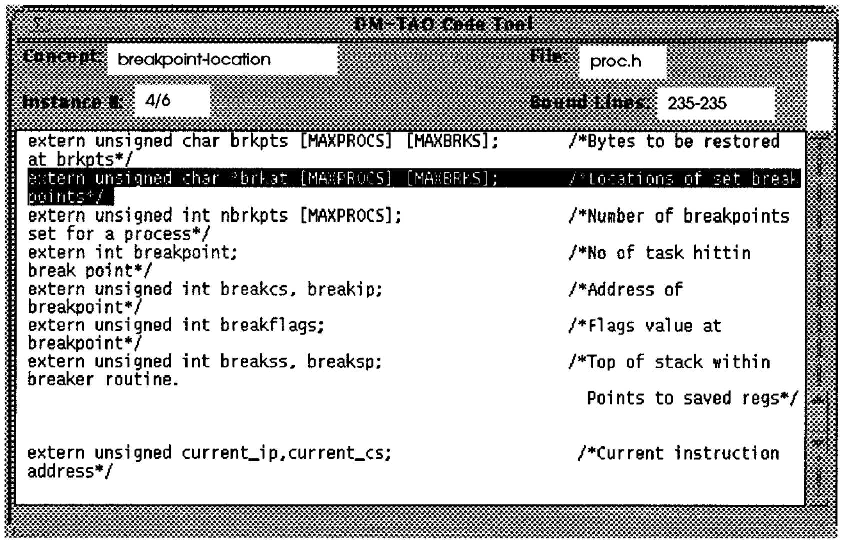
It should be clear from these scenarios that concept assignment with the human playing the role of the intelligent agent requires a wide variety of viewing, analysis and query tools. The detailed nature of these tools are heavily determined by the style of the investigators. However, the central invariant requirement is that the tools provide the mechanism for creating opportunistic associations and juxtapositions of information. Now, let us consider whether it is possible for the machine to play a support role that involves more intelligence.

4.5. Scenario 4: Intelligent Agent Provides First Clue

Another approach would be for our user to ask DM-TAO -- an experimental intelligent assistant for concept assignment -- to scan the code and present a list of candidate concepts based on 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 starting points for further investigation using the naive tools described in earlier sections.

The current version of DM-TAO can answer several kinds of questions about source code: 1) **Conceptual Highlights:** Look for any concepts that correspond to some concept in your DM; 2) **Conceptual grep:** Look for instances of a user-specified concept; and 3) **What's this?:** Propose a concept assignment for the currently selected code. In our scenario, we will assume that the user starts with a question of type 1 to perform a broad sweep of the code looking for important concepts. In the resulting concept list, the user notices **breakpoint-data**, DM's name for the model shown in Figure 2. The user asks to see the specific code associated with that concept and TAO presents the code from Figure 1 in a window. (TAO records the location of each concept found during the highlights search, so access is quite fast.) 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 (question type 3). The result is 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 what we have informally referred to as the *Location of breakpoints* concept. This provides the user a place to start. From here on, the user would use the same strategies and tools that we have already seen in the previous scenarios to verify the concept assignments and work out the detailed pattern of relationships.

How does DM-TAO accomplish its assignments? It uses a domain model (DM) to drive a connectionist-based inference engine (TAO), similar to [7]. The DM is built as a network in which each concept (e.g., *Location of breakpoints*) is represented as a node and the relationships between nodes are represented as explicit links (e.g., *Save code bytes* and *Location of breakpoints* are related via a *uses* link). The information associated with each concept includes: the typical features that characterize the concept, its relationships to other concepts in the domain, relevant informal knowledge - such as the terminology likely to be used by a programmer when referring to this concept in code, the syntactic and/or conceptual context this concept is likely to occur in, etc. The domain model also captures the underlying semantics in the target domain through a rich set of inter-concept relations embodying the nature and degree of the semantic associations between the domain concepts.



```
extern unsigned char brkpts [MAXPROCS] [MAXBRKS];       /*Bytes to be restored
at brkpts*/
extern unsigned char *brkat [MAXPROCS] [MAXBRKS];       /*Locations of set break
points*/
extern unsigned int nbrkpts [MAXPROCS];                 /*Number of breakpoints
set for a process*/
extern int breakpoint;                                  /*No of task hittin
break point*/
extern unsigned int breakcs, breakip;                   /*Address of
breakpoint*/
extern unsigned int breakflags;                         /*Flags value at
breakpoint*/
extern unsigned int breakss, breaksp;                   /*Top of stack within
breaker routine.
                                                          Points to saved regs*/


extern unsigned current_ip,current_cs;                   /*Current instruction
address*/
```

Figure 8: DM-TAO Suggests Assignment

To facilitate inferencing, this domain information is represented as a semantic/connectionist hybrid network. The concepts and their features are represented by nodes, which are of different 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 present in the domain model 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.

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 those in the input layer), the input is a function of the activations generated by the nodes in the previous layer that they are connected to, modulated by the weight on the connecting link. Nodes in the input layer are directly driven by the actions of a feature-extractor which scans the target code for relevant features - such as syntactic clues, lexical terms which might embody a concept-reference, 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

The evaluation of any system meant to assist a user in understanding real programs, that is not performed in a real-world context is necessarily suspect. Consequently, the testing and evaluation of DESIRE has always been done with "live ammo" and real users, which has often led to some discomfort for the research team. Nevertheless, 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 implemented 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 load of interaction with the users. A dozen or so sites is about the limit that a small research group can handle without impeding research progress.

The users of DESIRE have not always plumbed the full depths of the recovery facilities available. For example, cluster analysis is not often used because the users have not wanted to do a complete recovery. Often, they are under intense time pressure to bring in foreign code and get it running. Therefore, they tend to do a minimal recovery. The canonical pattern for a minimal recovery seems to be 1) generate and print out various Germ overviews, 2) use the query system (sometimes Germ and sometimes Prolog) to focus on areas with compiling or execution problems, 3) use the Slicer in a highly interactive, fast changing analysis of processing threads (driven by the problems) and 4) interspersed with all of the rest, use the hypertext navigation facilities to check definitions, etc. The only users who plumb the depths of DESIRE are those that are doing a full recovery (e.g., for re-engineering) and have a well defined, documented process for executing that recovery.

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

We have found that in the domain of multi-tasking windows systems, which is where we have our richest set of training data and experience, DM-TAO can recognize an interesting set of concepts. In one experiment, we chose three files (about 600 lines of code) containing data definitions within this domain. We performed a manual analysis of these files to identify the most important concepts to understanding the data. There were 27 human-oriented concepts in this set, of which only 20 were defined in the domain model. Next we ran the files through DM-TAO asking two kinds of questions: 1) what are the important domain specific concepts in these files, and 2) for specific code segments, what is the concept most closely associated with this code segment? DM-TAO recognized 20 of the 27 most important concepts in the files and produced three false positives, which we attributed to the fact that the net was only weakly trained.

When focusing on specific code segments, DM-TAO had a tendency to over generalize, recognizing the most specific appropriate concept as well as its superconcept (e.g., a queue concept as well as a data-holder concept). We currently believe that some of this is due to the fact that some of the feature extractors we planned (e.g., syntax categories) are not yet fully implemented. We are currently working on this and other mechanisms to control over generalization.

Even in it incomplete state, DM-TAO is interesting and promising but not yet ready for wide scale use.

DESIRE has always had at least two personalities because our client base is so varied. On the one hand, some of our clients wanted us to focus all of our energy on the risky research ideas (DM-TAO) and ignore the product-like features needed for practical use. On the other hand, many of our most active users wanted to do real work with DESIRE and requested new languages such as FORTRAN, particular report types and new features such as incremental module load and unload to support huge programs. Since the largest number of active clients were using the naive assistant tools, the requests for naive tools and features probably got the lion's share of our efforts. For example, there are tens of report types implemented, incremental module load/unload was implemented and FORTRAN will be delivered in a few months.

Because of this dual personality, the strengths and weaknesses are quite different depending upon which client group you ask. On the whole, however, I think we can legitimately record some strengths and weaknesses from the user feedback that would achieve fairly broad consensus.

Strengths

- The slicing mechanism is a popular feature with the users and this may result from the fact the Slicer was heavily used (in bootstrap mode) to develop itself, to analyze some legacy code used elsewhere in DESIRE and to develop other parts of DESIRE. The slice representation appears to be only part of its value. Two other features stand out: 1) the interactive operating regimen engendered by the interest and collection set facilities, and 2) the query facility that allows a description of interest sets in natural, abstract terms rather than requiring the explicit listing of individual interests.

- The graphical browser is heavily used as a facility for reporting passive, artfully tailored views of program structures for the purpose of publication or for reporting on passive mediums such as paper. This is a popular way to use the graphical views. To some degree this may be because conventionally, other graphical tools (e.g., CASE tools) have been used mostly to passively document designs after the fact (their interactive design capabilities notwithstanding.)
- The logic programming engine (Prolog) and the embedded Scheme extension language have been valuable in two ways: 1) as system extension facilities, and 2) as general engines for program analysis. Many of our own extensions were built using one or the other of these facilities. For example, the cluster analyzers were written in Prolog.
- The various query facilities provide an armory of easy ways to analyze a system and are heavily used especially when porting code.
- The hypertext navigation facility is extremely important because even though an abstract understanding of a program is possible without looking at code, any depth of understanding requires analyzing the details of the code. Hypertext provides among other things a conceptual zoom capability.

Weaknesses

- It is clear that we need a fuzzy search mechanism that goes beyond slicing, regular expressions or logic programming specifications (all of which were implemented). DM-TAO seems like the right idea but it is too early in its development to make a final judgment about its effectiveness. Currently, it is a loosely coupled component, so the advantages of complete integration into the rest of the system are not yet well understood.
- The graphical browser (Germ) lacks certain practical features (e.g., multiple windows per browser process) that were not considered important enough in a research environment to gain our attention but would certainly enhance its facility for interactively analyzing programs and creating intermediate, tailored views. The current process for this is slower and more clumsy than we would like especially for large programs. However, the large program aspect of this problem appears to be common to all graphical browsers that we have seen (commercial and research) and therefore, we believe that it may be inherent to the browser approach to the viewing of program structures.
- A second problem is also inherent to all graphical browsers used as interactive investigation tools for large programs. It is the impedance mismatch between automated graphical organizations of large amounts of information and the natural symbolic organization (i.e., chunking) of that information. Germ's aggregates, which allow substructure to be instantly hidden or revealed, begin to address this problem but they are not automatically formed, except during cluster analysis. There is much intelligence required in the chunking process.
- Internally, information exchange between the various tools (e.g., Germ, Prolog, and the Slicer) and their associated views is sometimes more clumsy than we would like. The problem is corrected in a partially completed re-design of DESIRE using an object-oriented canonical representation across all tools.

These weaknesses fall into two basic categories 1) inherent limitations in the technological approach (e.g., automatically organizing large amounts of graphical information), and 2) missing facilities and features. Category one suggests an agenda for future research and category two suggests an agenda for productization.

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

6. Relation to Commercial Products and Other Research

There are a variety of commercial products and research prototypes that address some part of the program understanding process. They can be conveniently differentiated by two key properties: 1) the degree of formality in the representations that they deal with and 2) the degree of domain specificity. Not unexpectedly, there is a rough correlation of reasoning methods with the degree of formality in the presentation. The more formal the representation, the more likely that they use deductive reasoning or algorithmic methods to derive information from the program. The more informal the representation, the more likely they use some fuzzy reasoning method such as plausible reasoning, pattern recognition or heuristic methods.

Similarly, the degree of dependence on domain knowledge correlates roughly with application specificity. The most general tools tend to build all knowledge into the tool, whereas the most domain specific tools are likely to depend on models of their domain specialties. Figure 9 summarizes current results in the area of program understanding and design recovery. It characterizes a number of product classes and research approaches by placing them according to where their architectural properties fit in this scheme.

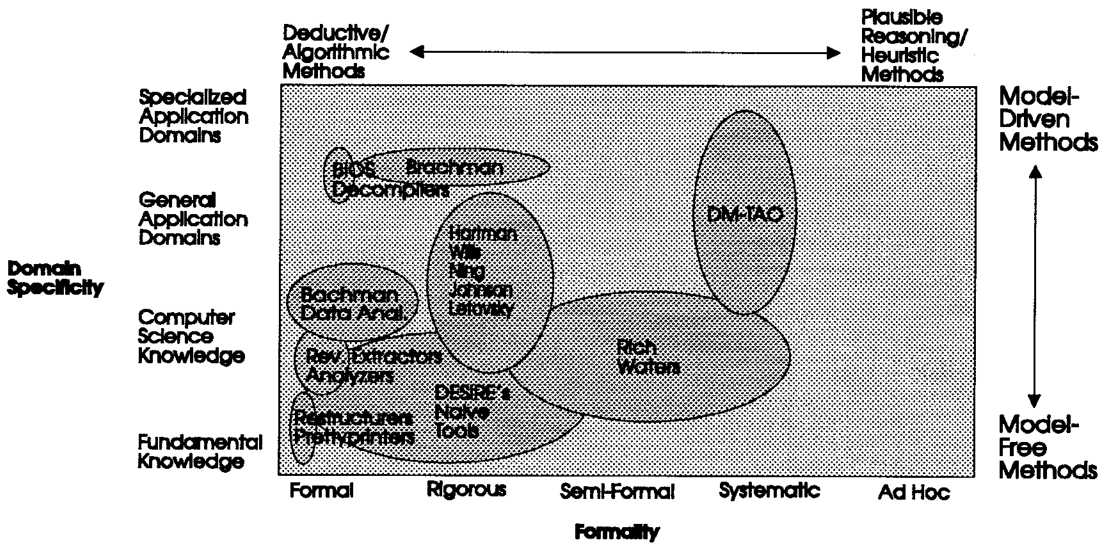


Figure 9: Program Understanding Landscape

Commercial Systems: There are no commercial tools that strongly address the general problem of program understanding and those that address some piece of the problem are clustered mostly in the lower left part of the diagram. These tools tend to work only with knowledge having the lowest level of conceptual specificity and use approaches that are mostly model-free. That is, these tools depend strongly upon general (non-domain-specific) knowledge and like many of the facilities shown in the scenarios are largely naive support tools. These tools provide facilities such as cross referencing, language restructuring, graphic reformulation (e.g., charting services), integration with CASE tools and so forth. Notable entries in this class are ProCase's SMARTsystem tools and Reasoning Systems' toolsets based on REFINE. A few tools like Bachman's MIS oriented tools or BIOS decompilers provide a bit more domain-specific knowledge, but they do not really address the concept assignment problem in any strong sense. Bachman's tools provide some intelligent help in redesigning data base schemas. Likewise, BIOS decompilers offer some help in relating absolute data addresses to meaningful symbolic names. Both of these toolsets offer useful services but do not address the program understanding or concept assignment problem in a general sense.

Research on Naive Agents: There are a number of research projects that focus variously on integrated tool sets [5], algorithmic analysis of program properties, program slicing [18] or analysis of program structure implied by generic program features [10, 15]. Such facilities are the building blocks for advanced program understanding systems because the implied properties and structures that they compute are a convenient starting point for intelligent agent processing whether by human or machine.

Research on Intelligent Agents: Research in this area has attacked more challenging understanding problems but because it is research, it has tended to address more constrained problems -- problems that are easily accomplished by a small number of researchers. There are three easily distinguishable research approaches:

1) Highly domain specific, model driven, rule based question answering systems that depend on a manually populated data base describing the software system, typified by the Lassie system [6],

2) Plan driven, algorithmic program understanders or recognizers [8, 9, 11, 12, 13, 14, 16, 17], typified by the Recognizer system of [14, 17], and

3) Model driven, plausible reasoning understanders, typified by DESIRE's DM-TAO subsystem [2, 3].

Approaches 1 and 2 have the desirable characteristics that they are good at faithfully and completely deriving concepts within small-scale programs but suffer from the problem of not being able to deal readily with large-scale programs because the inference chains or parsing procedures tend to become computationally infeasible in the face of overwhelming numbers of details. An exception to this kind of computational growth appears to be Hartman's work but the technique is something of an approximation technique and therefore, might legitimately be classified somewhere between approaches 2 and 3.

Conversely, approach 3 systems can easily handle large-scale programs and their computational growth appears to be linear in the length of the program under analysis. They, of course, suffer the converse problem in that their results are approximate and are therefore, imprecise and not completely trustworthy.

Systems of the class 3 type appear to be good at winnowing large numbers of details into a few interrelated problem domain entities that abstract and approximate a program's conceptual framework. They are less successful at deriving detailed implications such as detailed computational behavior descriptions. Conversely, class 2 is good at the latter and less successful at the former. Because they appear to complement each other so well, we expect these streams of research to evolve into hybrid systems that use different techniques to address the different aspects of the problem. For example, we performed some experiments that use logic programming to find clusters of tokens that are related based on surface features (e.g., similarities in spelling) and used the resulting clusters to focus DM-TAO's attention. Other similar hybrid strategies include using the algorithmic clustering facilities to extract implicit program features (e.g., patterns of data coupling and control cohesion) for input to DM-TAO and similarly, using DM-TAO to suggest frameworks for deeper analysis by DESIRE's naive analyzers and the human software engineer (e.g, stereotypical module architectures for specific types of problems and domains).

7. Conclusions

We conclude that 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 (i.e., as represented via concept assignments) is derived via a process that relies strongly, though not exclusively, on plausible inference. Finally, we conclude that rich understanding relies on an *a priori* knowledge base that is rich with expectations about the problem domain and the program architectures typical of that problem domain.

We are encouraged by the preliminary results of DM-TAO and 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] Daniel Brotsky, "Program Understanding through Cliche Recognition" MIT AI Lab Working Paper 224, (December, 1981).

[5] 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.

[6] 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).

[7] 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).

[9] Mehdi T. Harandi and Jim Q. Ning, "Knowledge-Based Program Analysis," IEEE Software, Vol. 7, No. 1, (January, 1990), pp. 74-81.

[9] John Hartman, "Automatic Control Understanding for Natural Programs," Ph.D. Dissertation, University of Texas, (1990).

[10] D. Hutchens and V. Basili, "System Structure Analysis: Clustering with Data Bindings," IEEE TSE, 11(8), (1985).

[11] Stanley Letovsky and Elliot Soloway, "Delocalized Plans and Program Comprehension," IEEE Software, (May, 1986).

[12] Stanley Letovsky, "Cognitive processes in Program Comprehension," Journal of Systems and Software, Vol. 7, pp. 325-339, (1987).

[13] Jim Q. Ning, "Knowledge-Based Approach to Automatic Program Analysis," Ph.D. Dissertation, University of Illinois, Urbana-Champaign, (1989).

[14] 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.

[15] Robert W. Schwanke, "An Intelligent Tool for Re-engineering Software Modularity", Proc. 13th ICSE, May 13-15, 1991, Austin, TX, pp. 83-92.

[16] E. Soloway and W. L. Johnson, "PROUST:Knowledge-Based Program Understanding," IEEE Transactions on Software Engineering, Vol. SE-11, No. 3, pp. 267-275, (March 1985).

[17] Linda M. Wills, "Automated Program Recognition by Graph Parsing," Ph.D. Dissertation, MIT, also published as MIT AI Laboratory Technical Report 1358, (1992).

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