Ted J. Biggerstaff, Bharat G. Mitbander, and Dallas E. Webster

Program Understanding and the **Concept Assignment Problem**

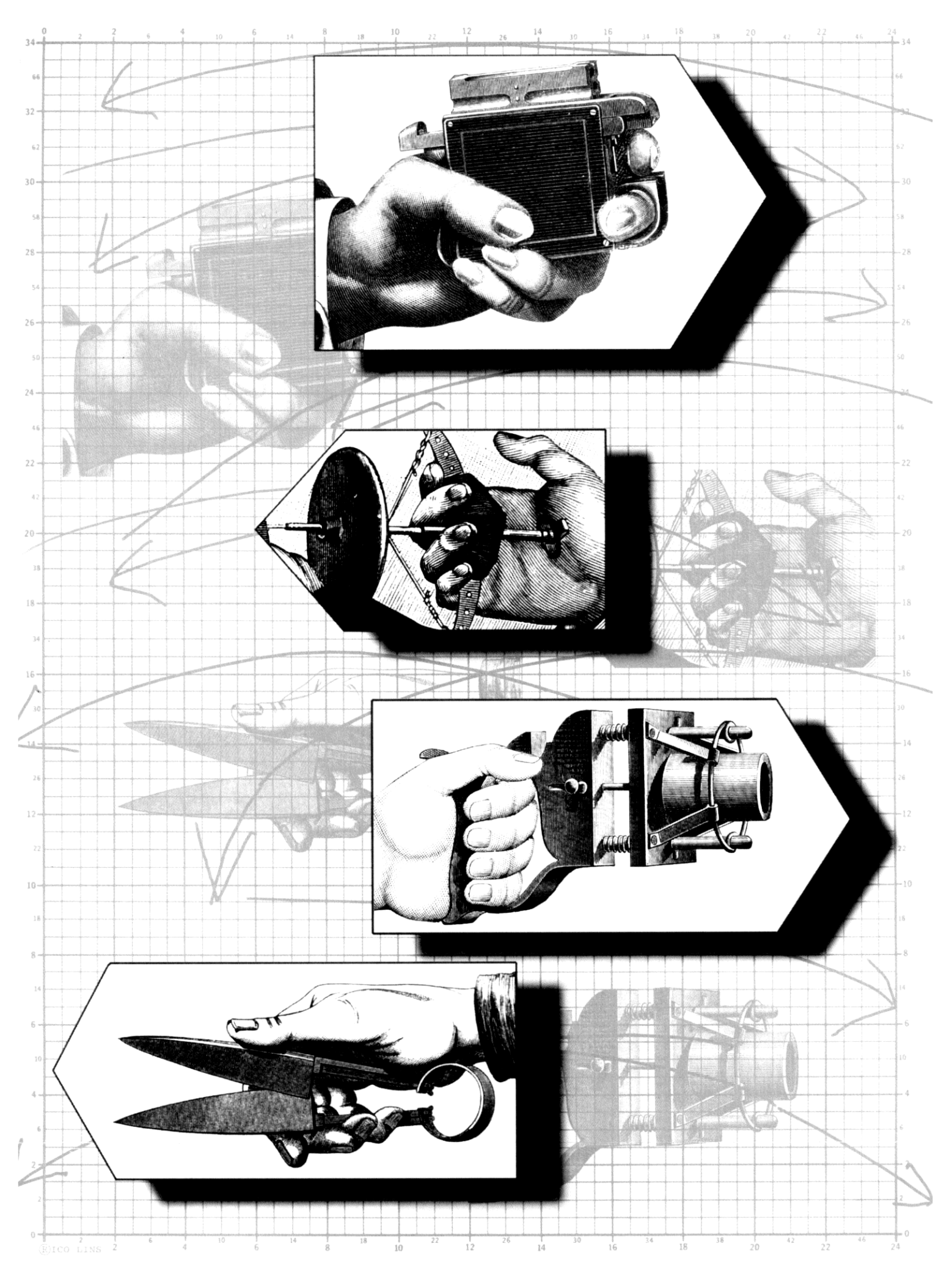
A person understands a program when able to explain the program, its structure, its behavior, its effects on its operational context, and its relationships to its application domain in terms that are qualitatively different from the tokens used to construct the source code of the program. For example, it is qualitatively different for me to claim that the 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 in level of detail and formality, the first case expresses computational intent in human-oriented terms—terms that involve 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, reengineer, 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.

When a person tries to develop an understanding of an unfamiliar program or portion of a program, the informal, human-oriented expression of computational intent must be created or reconstructed through a process of analysis, experimentation, guessing, and crossword puzzle-like assembly. 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 realizations within a specific program or its context is the *concept assignment problem* [4].

In practice, there are several general strategies and classes of tools that can successfully address this problem. We will illustrate some of these strategies through example scenarios and some classes of tools that support them through examples of the DESIRE (DESign Information Recovery Environment) suite of tools. The problem, strategies, and tools are relevant to anyone who creates, maintains, changes, reengineers, reuses, or otherwise manages the design of a program or system.

A central hypothesis of this article is that a parsing-oriented recognition approach based on formal, predominantly structural patterns of programming language features is necessary but not sufficient for solving 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. So there is more to program understanding than parsing. (See sidebar titled "Automatic Concept Recognition").

More specifically, parsing technologies lend themselves nicely to the recognition of *programming-oriented concepts* (e.g., numerical integration, searches, sorts, structure transformations) 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 allows us to 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 this is not the case. It appears there is truly a paradigm shift between programming-oriented and human-oriented concepts. There is a change both in the kinds 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, inherently ambiguous tokens [6], 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.

We give an example of this paradigm shift, in which *a priori* knowledge is used to drive the assignment of human-oriented concepts and focus on how tools, both naive and intelligent, can aid in that process.

An Example

Two general tasks are required when attempting to assign concepts to code:

1. Identify which entities and relations out of the often overwhelming numbers in a large program are really important.
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) 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 application architectures and relationships.

Consider the example in Figure 1 to see how concepts in code can be identified. This example is taken from a multitasking 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).

For task 1, we use generic knowledge to infer that these statements are related to one another in some noncasual way, because they are

1. 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), 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

Glossary

Domain model. A knowledge base that defines application-specific concepts (e.g., debugger concepts) as a set of entities and relationships (e.g., the entities *Locations* of breakpoints and *Save code bytes* are related by the *Uses* relationship).

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 a finite set of pattern templates, each of which specifies a concept occurrence as a set of syntactic features and patterns. Recognition is a recursive process in which the simplest, most elemental concepts are recognized first and then become features of larger-grained, composite concepts.

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 the location of the reference.

Recognition model. The method or strategy chosen to perform program recognition.

Signature. The set of features (e.g., syntax, semantic, graphical) that *together* signal the occurrence of a specific concept.

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 will assign the data-group and its subcomponents to domain-specific concepts, utilizing *a priori* domain-specific knowledge such as illustrated informally in Figure 2. In this diagram, 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 concrete designs. We assume 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 up 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 originally written, which is what is expected.

How might a knowledgeable user relate this model to specific instances of the concepts in a program under analysis? What features might be used 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
4. the overall pattern of relationships in the model

We illustrate these features in our example.

Certain natural language tokens—words, phrases and abbreviations—are features of (i.e., signal a likely reference to) the **breakpoint-data**[1] concept (e.g., "breakpoint," "brkpts," and "brkat"), while others signal possible references to concepts that are

[1] **"breakpoint-data"** is the canonical name used in DESIRE's knowledge base for the set of data items expected by the breakpoint model of Figure 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 illustrating data grouping

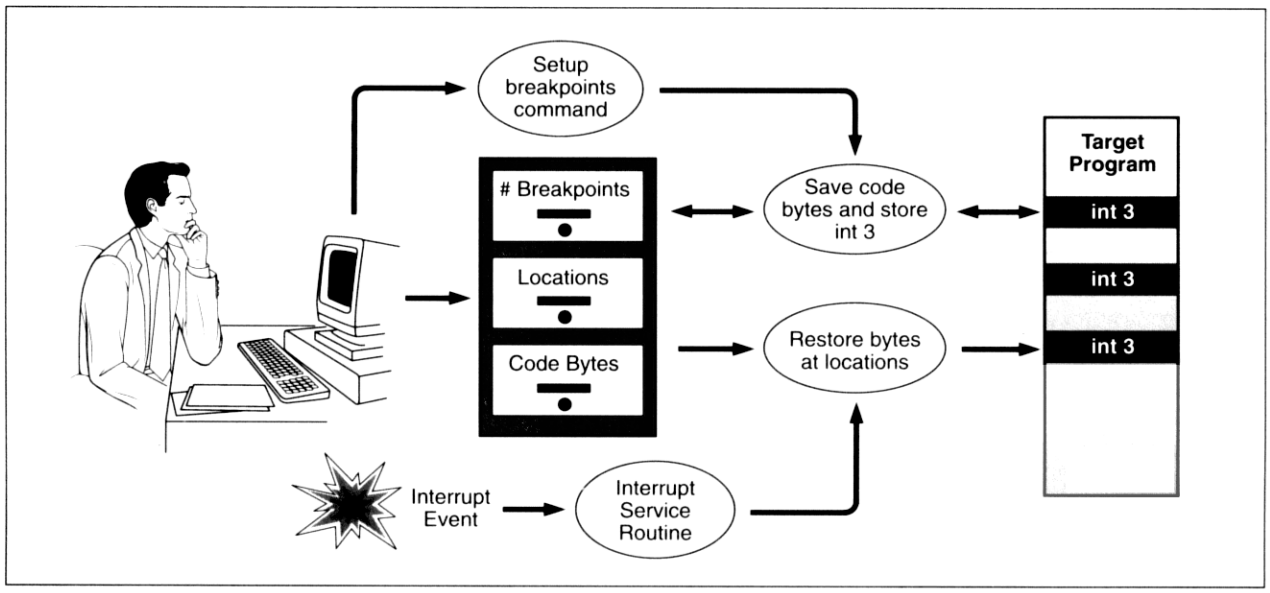


Figure 2. A model of breakpoint processing in debuggers

closely associated with the breakpoint data concept (e.g., the concepts address, registers, instruction, process and task). Finding evidence of these associated concepts adds to the possibility that "breakpoint," "brkpts," "brkat" and so forth are indeed signaling a reference to the concept breakpoint data.

Further evidence could be provided if the user has already discovered that these data items are used by some previously assigned (i.e., known) breakpoint-processing function(s). For example, the user might already know about one or more of the following functions:

- bpint3, which handles the actual breakpoint interrupt;
- set_breaks and 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 the table brkat; or
- restore_breaks and restore_rbrkpt, 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 the **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.

Concept Assignment Tools and Scenarios

One can approach the tasks outlined previously in a variety of ways. Of course, the human mind is one of the best pattern recognizers and is a significant part of any solution to the concept assignment problem. But even primitive tools such as 'grep' and commercial reengineering tools can be quite useful. Based on 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 the user is the intelligent agent and they provide simple but computationally intensive services to support that intelligence. On the other hand, the intelligent assistant facilities, which include a Prolog-based inference engine and a knowledge-based pattern recognizer called DM-TAO (Domain Model-The Adaptive Observer), are more experimental and attempt to provide a limited amount of intelligent assistance in assigning concepts.

We will use several 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.

Scenario 1: Suggestive Data Names as First Clue

In this scenario, we suppose a user is browsing the global data of some unfamiliar program and discovers the breakpoint data group of Figure 1. Let us further assume this user has the domain knowledge 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 of the *Code bytes* data store, brkat of the *Locations* data store and nbrkpts of 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 asks for a Germ[2]

[2] Germ (graphical entity-relation modeler) is a schema-driven graph browser with hypermedia functionality that relates browser items to elements of target programs.

Automatic Concept Recognition

Concept assignment is a process of recognizing concepts within a computer program—including all artifactual information associated with the code—and building 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 in which the simplest, most elemental concepts are recognized first and then they 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—largely a result of the nature of the technology (specifically, parsing technology) conveniently available to attack this problem. For parsing technologies to be effective, they rely heavily on 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. Harandi, M.T. and Ning, J.Q. Knowledge-based program analysis. *IEEE Softw. 7*, 1 (Jan. 1990), 74–81.
2. Rich, C.E. and Wills, L.M. Recognizing a program's design: A graph-parsing approach. *IEEE Softw. 7*, 1 (Jan. 1990), 82–89.

Figure 3. Germ view of use/call graph

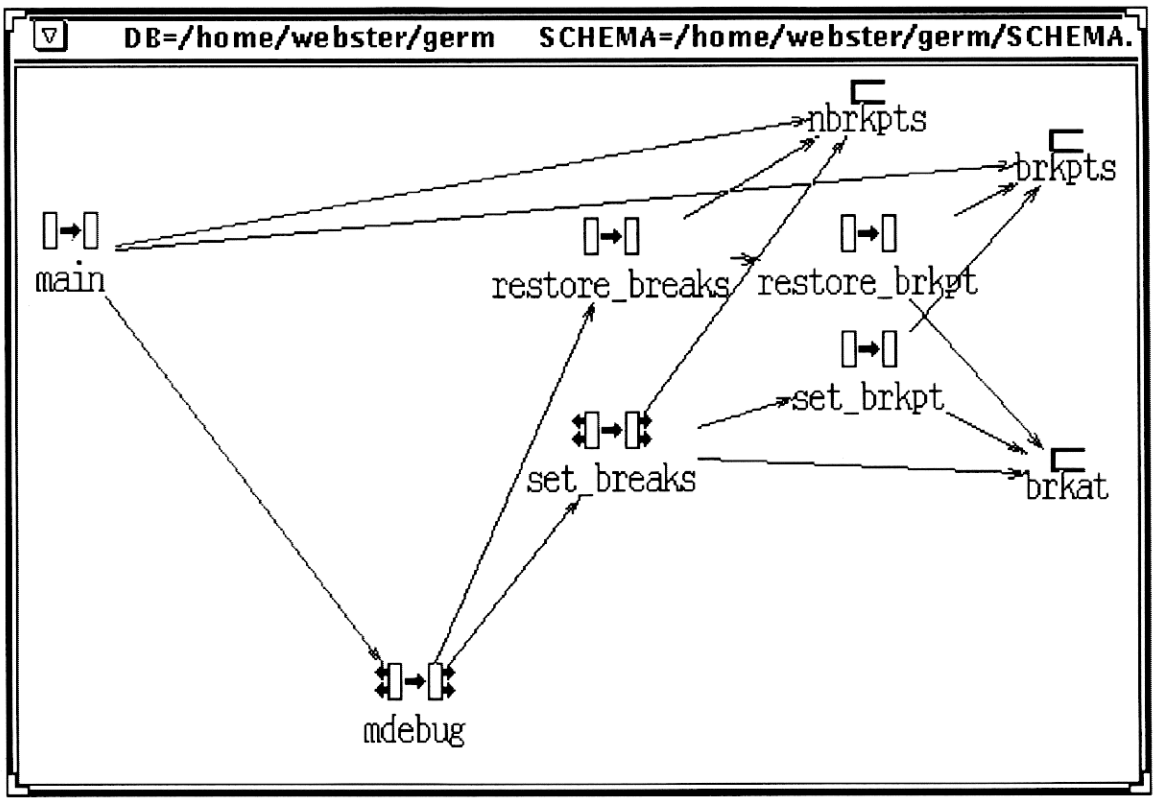


Figure 4. Slicer's view of part of mdebug code

DESIRE Shell

```
/* 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);
```

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.

These results reveal several strong candidates (set_brkpt, set_breaks, restore_brkpt, and restore_breaks) for assignment to the *Save code bytes* and *Restore bytes* concepts. The user examines the source code to verify these tentative assignments and discovers 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.

However, the user does not yet have assignments for the breakpoint *Interrupt Service Routine* and the *Setup . . .* concept (i.e., the user-driven interface function that triggers the saving of the breakpoint). Since interrupt service routines are invoked by the hardware, the *Interrupt Service Routine* concept would not have turned up in the call chains. But interrupt routines do communicate with the rest of their application via global data. Further, the two routines corresponding to *Interrupt Service Routine* and *Setup . . .* will be somehow 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 that are loosely related to the current set of interesting functions and data. In DESIRE, this is accomplished by requesting a program slice[3] [8] based on the set of currently interesting program entities.

While a simple tool like 'grep' can isolate the lines of code containing a given string, it does not identify control paths or other computational dependencies related to a variable of interest as DESIRE's slicer does. Moreover, DESIRE's slicer 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 powerful operations for generating 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.

The slice introduces several new global variables because of the conditional branch leading to the restore_breaks call 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 the figure), the user finds the code that calls set_breaks, and it is embedded within logic that interprets the user's debug commands. Hence, mdebug is the assignment for the *Setup breakpoints 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 by an intelligent human agent.

Scenario 2: Patterns of Relationships as First Clue

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

How might one go about trying to discover such an architecture in a language such as C? Sometimes module clusters depend on 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 a user is searching for functional clusters based on shared control paths, looking for a cluster of functions that are tightly related because all call paths to them contain a single function, called their *dominator.* Our debugging example contains just such a cluster where the dominator is mdebug. Why might our user suspect this is a cluster? Perhaps a suggestive pattern of calls within the call graph is noticed. For example, a set of functions that appear connectively isolated except for a rich set of connections to mdebug might be detected (see Figure 5). So, our user runs a cluster analysis with mdebug as the dominator—the results are shown in Figure 6.

The functions found not only include the set and restore functions discussed in 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., parseaddr) and others for dumping information (e.g., dumpwords). Further exploration will suggest additional candidates for inclusion in the proposed debugger module based on functions that are perhaps less directly related to the debugger concept.

At this stage the user asks that this clustering relationship be recorded as a (new) module, and an aggregation node is created in the database. This new module node groups these functions so they can be dealt with as a unit. Typically, the user will want to use this aggregation relationship to simplify some graphical views by collapsing (i.e., hiding) the complexity of all of these functions temporarily inside this single module node.

The user would likely proceed with other cluster analyses until all functions are assigned to some module, which provides a module-based overview of the system (see Figure 7). These cluster results can then be used

[3] Loosely defined, a slice for a variable is all of the statements that affect the value of the variable. There are several variations on this idea in DESIRE.

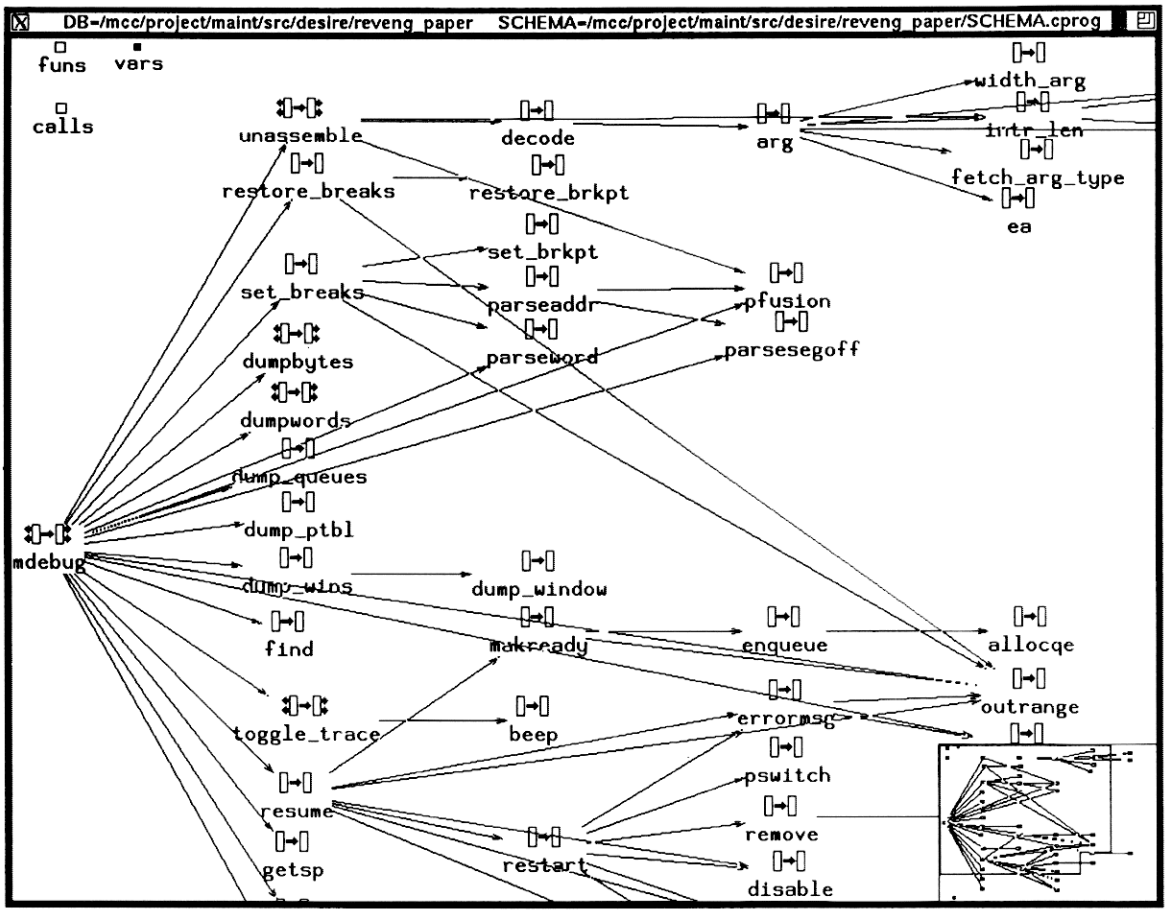
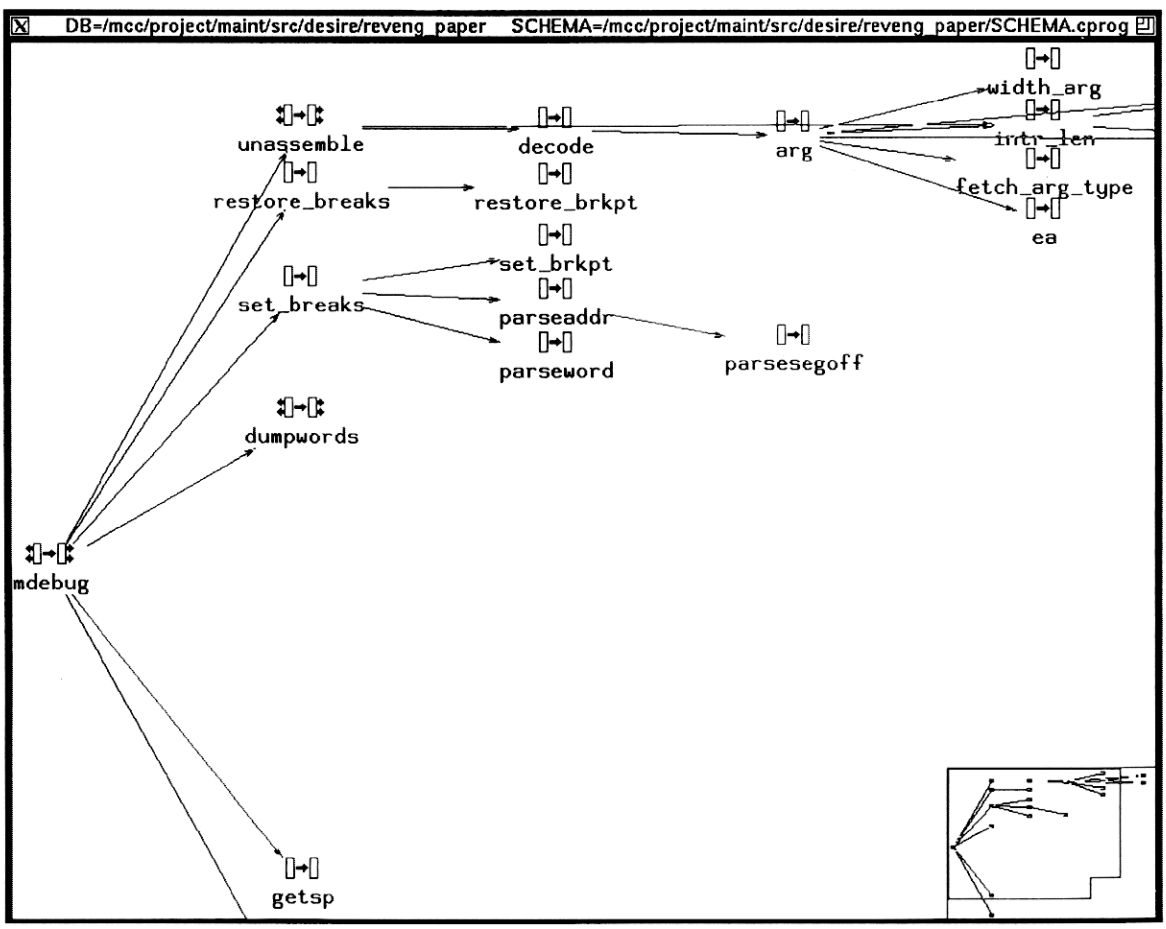


Figure 5. Germ browser call graph

Figure 6. Results of cluster analysis



in other tools—the browser, query engine or the slicer.

It should be clear from the prior 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.

Scenario 3: Intelligent Agent Provides First Clue

Another approach would be to browse the code looking for evidence of key concepts based on the user's experience. But with DESIRE, the user may ask DM-TAO—DESIRE's experimental intelligent assistant for concept assignment—to scan the code for him and present a list of candidate concepts based on the knowledge represented in its domain model (DM) knowledge base. 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 earlier.

The current version of DM-TAO can provide several kinds of analyses of 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.
- **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.** 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 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 it attacks merit some elaboration. It uses the DM to drive a connectionist-based inference engine (TAO), similar to [5] and [7]. 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, for example, representing the different kinds of information present. 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 interconcept relationships are represented by corresponding internode 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 their 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 syntactic, lexical, and clustering clues. 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 its links. The system adapts its 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.

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 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 since it is still a re-

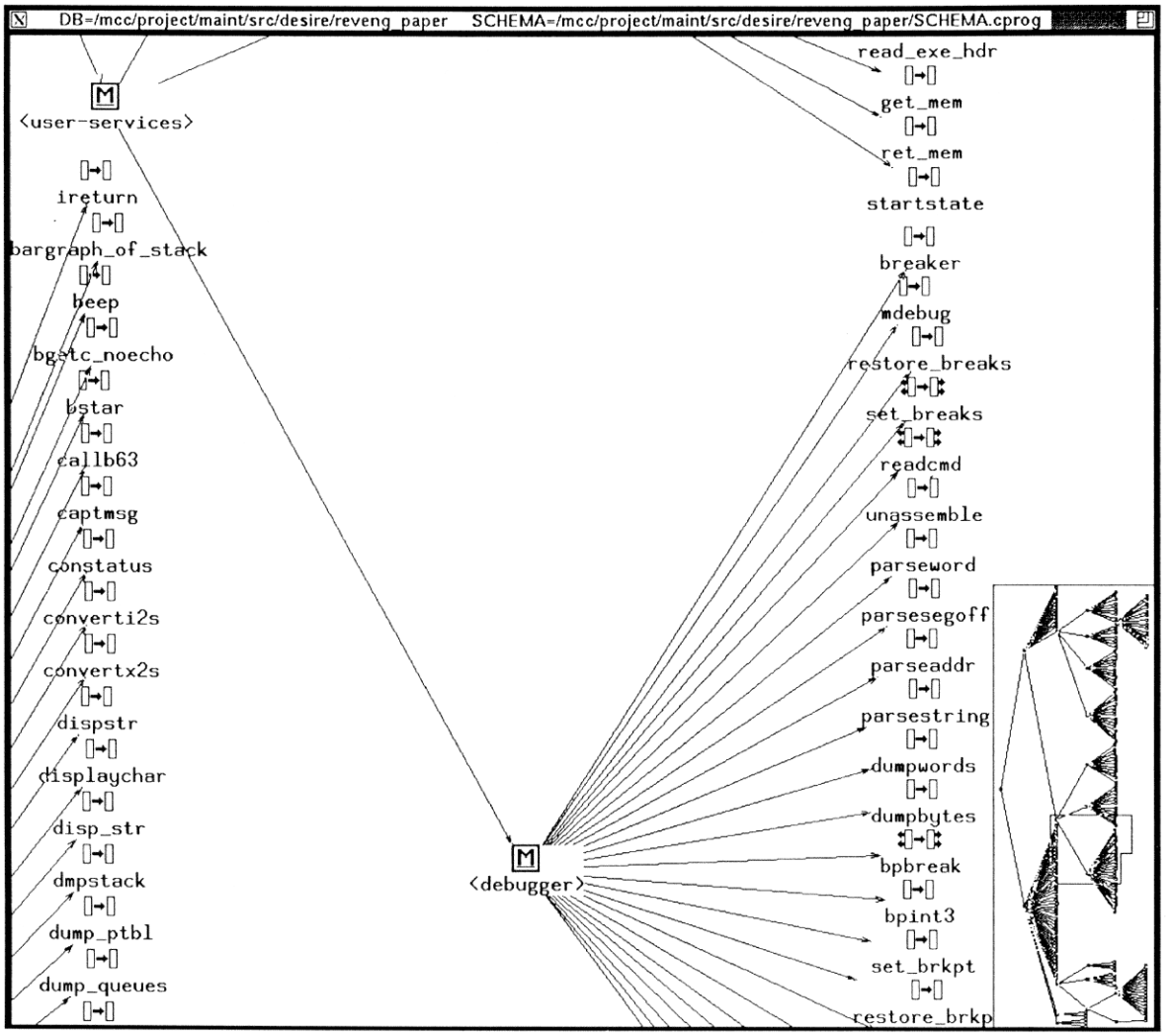
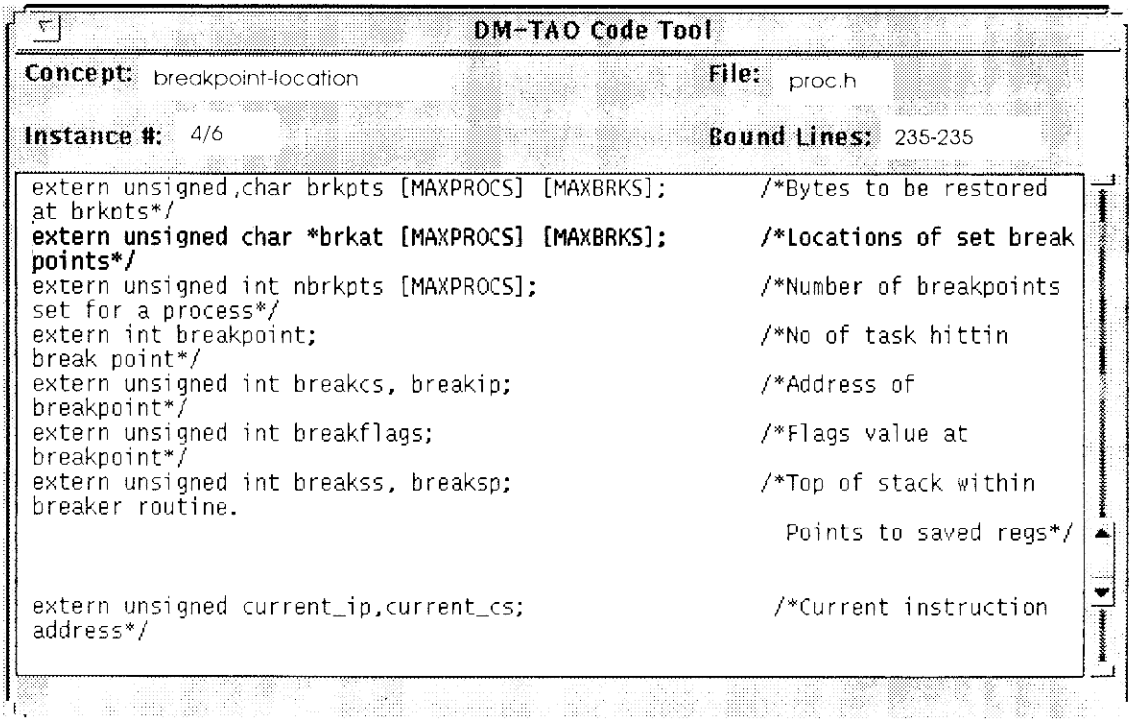


Figure 7. Module view of the system

Figure 8. DM-TAO suggests assignment



search prototype and has not yet been released for use outside the laboratory.

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 the limit 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 favored—it is often used for reporting passive, artfully tailored views of program structures for publication or analysis.

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 requiring a good deal of manual labor. These experiments show promise but are not yet definitive. Although it has some of the weakness of a research prototype, DESIRE has been used to do real work and validate the strategies described.

Conclusions

Since the concept assignment problem is obviously a difficult one, automation of even a small portion of this problem requires architectures that process a range of information types varying from formal to informal, so the information inferred from the informal can improve the ability to infer information from the formal and vice versa. Further, it seems clear from our analysis of sample 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 typical program architectures.

We are encouraged by the preliminary results of DM-TAO and the strategies that motivated it. While we believe 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. Biggerstaff, T.J. *Systems Software Tools.* Prentice-Hall, Englewood Cliffs, N.J., 1986.
2. Biggerstaff, T.J. Design recovery for reuse and maintenance. *IEEE Comput. 22,* 7 (July 1989), 36–49.
3. Biggerstaff, T.J., Hoskins, J., and Webster, D. DESIRE: A system for design recovery. MCC Tech. Mem. STP-081-89, Apr. 1989.
4. Biggerstaff, T.J., Mitbander, B., and Webster, D. The concept assignment problem in program understanding. In *Proceedings of the International Conference on Software Engineering* (Baltimore, Md. May 1993).
5. Feldman, J.A., Fanty, M.A., Goddard, N.H., and Lynne, K.J. Computing with structured connectionist networks. *Commun. ACM 31,* 2 (Feb. 1988).
6. Furnas, G.W., Landauer, T.K., Gomez, L.M. and Dumais, S.T. The vocabulary problem in human-system communication. *Commun. ACM 30,* 11 (Nov. 1987).
7. Henninger, S. Supporting the process of finding reusable software. Tech. Rep. UNL-CSE-94-003, Department of Computer Science and Engineering, University of Nebraska-Lincoln, 1994.
8. Weiser, M. Program slicing. *IEEE TSE, 10* (1984), pp 352–357.

About the Authors:

TED BIGGERSTAFF is research program manager at Microsoft Research. Current research interests include software reuse, design recovery, reverse engineering, re-engineering, transformational programming, and neural networks. **Author's Present Address:** Microsoft Research, Mail Stop 9S/1032, One Microsoft Way, Redmond, Washington, 98052-6399; email: tedb@microsoft.com

BHARAT MITBANDER is a member of the technical staff at Microelectronics and Computer Technology Corporation (MCC). Current research interests include design information recovery, application dredging, knowledge discovery, neural networks, and design automation.

DALLAS E. WEBSTER is a senior member of technical staff at Microelectronics and Computer Technology Corporation (MCC). Current research interests include design information recovery, application dredging, and knowledge acquisition, representation, and presentation. **Authors' Present Address:** MCC, 3500 W. Balcones Center Drive, Austin, TX 78759-5398; email: {mitbander, webster}@mcc.com

Related Research and Technology

A variety of technologies 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 relevant to program understanding are:

Maintenance and reengineering: The forces of change (e.g., computer "downsizing") are creating increased automation, supporting program maintenance and reengineering. These tools are variously focused on program reorganizing [6, 10], program porting, or database reengineering [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 nonexecutable business rules [7].

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 [2], program slicers, and language-aware editors;
- Computation of *implicit* program information such as provided by module groupings [9] or data flow [1, 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 *abstract* 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. Canfora, G., Cimitile, A., and de Carlini, U. A logic-based approach to reverse engineering tools production. *IEEE TSE, 18,* 12 (Dec. 1992).
2. Chen, Y.F., Nishimoto, M.Y., and Ramamoorthy, C.V. The C information abstraction system. *IEEE TSE, 16,* 3 (Mar. 1990), 325–34.
3. Devanbu, P., Brachman, R.J., Selfridge, P.G., and Ballard, B.W. LaSSIE: A knowledge-based software information system. In *Proceedings of the International Conference on Software Engineering* (Nice, France, March, 1990).
4. Hainaut, J-L., Chandelon, M., Tonneau, C., and Joris, M. Contribution to a theory of database reverse engineering. In *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. Newcomb, P. and Markosian, L. Automating the modularization of large COBOL programs: Application of an enabling technology for reengineering. In *Proceedings of the Working Conference on Reverse Engineering* (Baltimore, Md., May 1993).
7. Ning, J., Engberts, A., and Kozaczynski, W. Recovering reusable components from legacy systems by program segmentation. In *Proceedings of the Working Conference on Reverse Engineering* (Baltimore, Md., May 1993).
8. Rich, C., and Waters, R.C. The programmers apprentice: A research overview. *IEEE Comput. 21,* 11 (Nov. 1988).
9. Schwanke, R.W. An intelligent tool for reengineering software modularity. In *Proceedings of the International Conference on Software Engineering* (Austin, Tex., May 13–15, 1991), 83–92.
10. Ward, M.P. and Bennett, K.H. A practical program transformation system for reverse engineering. In *Proceedings of the Working Conference on Reverse Engineering* (Baltimore, Md., May 1993).