Supporting the Restructuring of Data Abstractions through Manipulation of a Program Visualization

Robert W. Bowdidge
IBM T. J. Watson Research Center

and

William G. Griswold
University of California, San Diego

With a meaning-preserving restructuring tool, a software engineer can change a program's structure to ease future modifications. However, deciding how to restructure the program requires a global understanding of the program's structure, which cannot be derived easily by directly inspecting the source code.

We describe a manipulable program visualization—the *star diagram*—that supports the restructuring task of encapsulating a global data structure. The star diagram graphically displays information pertinent to encapsulation, and direct manipulation of the diagram causes the underlying program to be restructured. The visualization compactly presents all statements in the program that use the given global data structure, helping the programmer to choose the functions that completely encapsulate it. Additionally, the visualization elides code unrelated to the data structure and to the task, and collapses similar expressions to help the programmer identify frequently occurring code fragments and manipulate them together. The visualization is mapped directly to the program text, so manipulation of the visualization also restructures the program.

We present the star diagram concept and describe an implementation of the star diagram built upon a meaning-preserving restructuring tool for Scheme. We also describe our creation of star diagram generators for C programs, and test the scalability of the star diagram using large C and MUMPS programs.

Categories and Subject Descriptors: D.2.2 [**Software Engineering**]: Tools and techniques—*modules and interfaces*; D.2.2 [**Software Engineering**]: Tools and techniques—*user interfaces*; D.2.7 [**Software Engineering**]: Distribution and Maintenance—*restructuring*; D.3.3 [**Program-**

This work was supported in part by NSF Grants CCR-9211002 and NSF-CCR-9508745, and a SAIC Summer Fellowship. Parts of this paper are revised and extended from a paper presented at the ACM SIGSOFT '94 Symposium on the Foundations of Software Engineering, December 1994. Name: Robert W. Bowdidge
Affiliation: IBM T. J. Watson Research Center
Address: P.O. Box 704, Yorktown Heights NY 10598
Name: William G. Griswold
Affiliation: Computer Science and Engineering Department, University of California, San Diego
Address: La Jolla CA 92093

ming Languages]: Language constructs and features—*abstract data types*; I.2.2 [**Artificial Intelligence**]: Automatic programming—*program transformation*

General Terms: semi-automated restructuring, software visualization,tool-supported restructuring

Additional Key Words and Phrases: star diagram, meaning-preserving restructuring

1. INTRODUCTION

Any useful computer program is likely to require changes during its life in order to correct errors, support new peripherals, and adapt to changes in the domain served by the program. In fact, maintenance of existing source code is responsible for a substantial fraction of a computer program's lifetime cost. Lientz and Swanson observed that large companies spent 50% of their programming effort on maintenance [Lientz and Swanson 1980], and Boehm documented an Air Force project where the cost to develop the program was $30 per line, but cost to maintain the program was $4,000 per line [Boehm 1975]. Moreover, maintenance becomes more expensive over time as programmers layer modifications upon the existing program to meet the demands of users. Each new modification requires the programmers to spend more time understanding the code, applying the change, and fixing bugs resulting from the change [Lehman and Belady 1985].

Some structural problems are caused by modifications that do not fit well into the program's original design. Although good software engineering practice encourages programmers to plan for future modifications, not every future design change can be predicted. User requests for changes are often a consequence of using the system after delivery. Also, it may be deemed too costly to accommodate all anticipated changes in a single coherent system design.

An example of an unanticipated change resulting in structural problems is the current move to network-centric computing. Most network computing paradigms allow data artifacts to be accessed only by (remote) procedure call or object reference. In migrating an existing application to a network computing platform, accesses to simple variables, for example, must to be wrapped in procedural or object abstractions.

1.1 Program Restructuring

A programmer sometimes restructures existing programs to simplify a planned enhancement. The programmer localizes design decisions scattered throughout the code into a single module, then applies the modification to the new module. This "restructure, then modify" process simplifies reasoning about, implementing, and testing the enhancement because the programmer only considers the behavior of the module in relative isolation. However, restructuring by hand is both error prone and difficult because a programmer's changes can introduce subtle flaws and bugs into a previously working program [Griswold and Notkin 1992]. As a result, a programmer must work methodically and carefully to produce a correct restructured program.

Consider the example of migrating an application to a network computing platform. All accesses to the variable in question must be identified and appropriately abstracted. Such changes are complicated by the fact that the variable accesses are

```
...
(let
  ((len (length
         (list-ref *line-storage*
                   lno))))
...
  )
...
(set! linelen
  (length
   (list-ref *line-storage* lnum)))
...
```

(a) Before extract-function

```
(define words-on-line
  (lambda (line-number)
    (length
     (list-ref *line-storage*
               line-number))))
...
(let ((len
        (words-on-line lno)))))
...
  )
...
(set! linelen
      (words-on-line lnum))
...
```

(b) After extract-function

Fig. 1. The extract-function transformation is given a common expression occurring throughout the Scheme program. It then creates a new function whose body is the expression, and replaces occurrences of the expression with calls to the new function. The tool user also specifies the name of the new function, its location in the code, and what parts of the expression will be parameterized.

likely to be widely distributed, and similar accesses may look different due the way the code is organized. The programmer must identify all the accesses, classify them, decide which parts to put inside the abstractions and which are input parameters. The appropriate abstractions must be defined, and then the variable accesses are replaced with references to these abstractions. Once the variable is appropriately wrapped, it is relatively straightforward to make it available as a net object.

Some restructuring changes can be performed automatically, such as removing gotos from programs [Bush 1985; de Balbine 1975], but cannot take into account any unique structural requirements of the planned change. Tool-assisted restructuring, on the other hand, allows the maintainer to decide what sort of restructuring is needed for the specific change, and the tool performs the changes in a meaning-preserving fashion. One way to support this approach is with meaning-preserving restructuring transformations that are chosen and applied by the programmer to selected portions of the program [Griswold and Notkin 1993]. Transformations include extracting a new function from inlined statements (see Figure 1), inlining functions, and extracting embedded constants or expressions as parameters to functions. These transformations often mimic common operations performed by programmers restructuring programs by hand [Griswold and Notkin 1992]. Similar transformations have been defined for object-oriented languages [Johnson and Opdyke 1993; Opdyke 1991].

A prototype tool for restructuring imperative Scheme programs has been implemented. (Scheme, a Lisp-like language [Dybvig 1987], contains many of the features of programming languages that make program analysis difficult, such as pointers, side effects, and assignment; the restructuring tool does not handle closures or `eval`.) The tool currently supports twenty basic transformations [Griswold and Notkin 1993], although composite transformations can be created from the existing transformations. To restructure, the programmer uses a graphical user interface to select a transformation to apply and a component to apply it to, along with information about the names of new identifiers, locations for new constructs, and the like. As a precondition to transformation, the tool performs syntactic and semantic checks using an abstract syntax tree (AST) and program dependence graph (PDG) to ensure the transformation will be meaning-preserving. If the check fails, the problem is reported to the programmer so that an alternative approach can be undertaken.

1.2 Difficulties of Planning and Performing Restructuring

A transformation-based restructuring tool allows the programmer to concentrate on what the improved structure of the program should be and what restructuring transformations should be performed in order to get to that goal, rather than focusing attention on local modifications to ensure that bugs are not introduced into the code.

Planning a restructuring, however, is not trivial. Consider, for example, the scenario in Figure 1 of identifying a frequently repeated computation in the code and converting it into a new function to be called in place of the repeated computations. Such an operation tends to occur when the programmer is encapsulating an existing data structure into an abstract data type (ADT). If designed correctly, the resulting ADT will localize—and hence simplify—modifications of the data structure [Parnas 1972; Liskov et al. 1977]. To plan an encapsulation, a programmer might search the program using printed source code listings or a text editor. Unlike views derived from the source code, the source itself allows the programmer to examine the context of uses, perform odd searches, and "explore the code" without a specific focus of interest. However, the technique does not scale to large programs because it is costly to find and compare expressions scattered across multiple pages or files in order to group them into a candidate ADT interface function. The source code view of a restructuring tool will suffer these same drawbacks.

To overcome these shortcomings, a programmer will typically use a text searching tool such as `grep` [Aho 1980] to provide assistance in locating variable references across many files. When planning the encapsulation of a variable, the programmer searches for the variable name, and `grep` returns a global view of how that variable is used throughout that program, formatted as a list of relevant lines in the program. However, `grep` does not understand the underlying programming language. Searches with `grep` may match variables with similar names in other scopes, or miss indirect references to a data structure (such as when the variable is passed as an argument to a function, and the argument is manipulated within that function). Also, `grep` proves less useful for encapsulating types because no single search can identify all uses of all variables of a given type. `grep` also cannot easily highlight nearby, cohesive statements to provide relevant context important to the programmer; instead, `grep` can show only a fixed number of nearby lines. Even if these weaknesses are resolved, simply listing all the computations that reference the data structure potentially involves a substantial amount of text. Such a solution does not show the similarities among calculations, which the programmer must still identify by memorization and recall.

Consequently, we can see that restructuring to encapsulate an existing data structure or type requires the programmer to gain a rich, *global* understanding of the system in order to identify redundant code, and also understand *in detail* how the redundant computations appear in all their contexts in order to propose an appropriate function that correctly replaces all the occurrences.

1.3 Solution: The Star Diagram

Our question, then, is how do we help programmers perform encapsulation with a restructuring tool?

We initially explored a direct-manipulation restructuring interface [Griswold and Bowdidge 1993] based on structure charts [Stevens et al. 1974]. However, structure charts, like most standard design notations, display only existing abstractions, not code fragments that could be turned into new abstractions.

The problems of restructuring with structure charts and the tools programmers currently use suggest a set of requirements for an "encapsulation tool". The approach needs to provide a global, condensed view of the program. The programmer should be able to gain an understanding of the code from a single view, without having to search through the code. The view should elide details of the program irrelevant to encapsulation. The view should also help the programmer match and group similar expressions when identifying likely functions. The tool should have knowledge of the source code's language in order to avoid incorrectly identifying variable uses and provide some indication when cohesive calculations are split across multiple statements. Finally, we assume that a programmer performing modifications best understands what the final form of the program should be. Therefore, we want the system to provide the information that the programmer needs to plan the interface, rather than automatically formulating the plan and performing the changes without user interaction.

To meet these needs, we created a new visualization called the *star diagram* (see Figure 2). The star diagram graphically presents the direct and indirect uses of a data structure, and also highlights the redundancies amongst those computations. Additionally, direct manipulations of the view can invoke restructuring transformations on the underlying source code.[1]

We first present the star diagram concept, describing the visualization and the transformations it supports, and then provide an example of the star diagram's use for restructuring a Scheme version of Parnas's KWIC program [Parnas 1972]. We then describe a generic algorithm for generating star diagrams that can be customized to accommodate a variety of programming languages and encapsulation requirements. Using this algorithm, we show that the star diagram concept applies to real-world languages by creating a star diagram generator for C programs, and then examine the scalability of star diagrams by measuring the size of star diagrams for medium-sized public-domain and commercial programs (`emacs`, the SunOS networking code, and a MUMPS medical database system) ranging from seventy thousand to a half million lines long. A separate study was conducted to explore how the star diagram impacts programmer behavior during restructuring [Bowdidge and Griswold 1997].

[1]The visualization's name is meant to suggest computations radiating in all directions from the data structure references in the center. Perhaps fortunately for the users of our tool, off-the-shelf graph layout tools did not support this kind of display. However, we have considered displaying read accesses to the data structure in one direction and displaying write accesses in the opposite direction, which would give the diagram a more star-like appearance.

Fig. 2. Star diagram for the array `*line-storage*` in the KWIC program. In the left column, the scrolling list displays a list of current functions in the interface, and the Remove button at the top allows the tool user to remove a function from the interface so it can be manipulated using the star diagram. The top of the window contains menus for window operations. The main star diagram subwindow consists of the seven transformation buttons at the top, and the star diagram in the rest of the window space.

2. THE STAR DIAGRAM CONCEPT

The star diagram is designed to help a programmer identify how to encapsulate all instances of a type into an ADT, as well as to support transforming the program to achieve that encapsulation. However, because the Scheme language does not support static typing, our Scheme star diagram is designed for encapsulating a single variable representing a global data structure. As we will see in later sections, variants of the star diagram support encapsulating multiple variables and types.

2.1 Description of the Star Diagram

Figure 2 shows a star diagram window being used to encapsulate the `*line-storage*` variable in the KWIC program. (See Appendix A for the source code for our implementation of KWIC.) The star diagram window is made up of four parts. The star diagram visualization itself fills most of the window, and appears as the directed graph in the bottom half of the window. A "button bar" above the star diagram visualization provides a button for each restructuring transformation that can be performed from the star diagram. To the left of the window, a scrolling list specifies the functions currently in the interface of the ADT being created. Finally, along the top of the window, a set of menus gives access to less frequently used commands.

2.1.1 *Star Diagram Visualization.* The star diagram visualization itself appears in the lower portion of the star diagram's window.

Basically, a star diagram is a graphical representation of the "consumes" and "controls" relations in the program source for all references to a variable. In addition, the graphical representations of references that are involved in similar consuming or controlling relationships are "stacked" on each other, resulting in a tree structure.

More precisely, a *star diagram* for a program is a tree of program components such that:

—Its *root node* is a set of program components selected by the programmer (e.g., all syntactic references to a chosen variable or all references to a chosen data structure).

—A node's *successors* (i.e., tree children) are a similarity partition of all the program components that (a) syntactically contain an element in the node or (b) reference the value produced by an element in the node.

—The similarity partition is determined by a *similarity criterion* (often called the *stacking criterion*). For example, two program components may be deemed similar if they are defined by the same construct, such as `+`.

—Nodes denoting functions definitions have no successors.

When a star diagram is graphically displayed, the root node is placed on the left, and successors are progressively laid out top-to-bottom, left-to-right. Nodes whose sets contain more than one element are drawn as boxes with a stacked appearance. All the function nodes denoting the same function definition are overlaid, giving the diagram the appearance of a graph.

The resulting tree of nodes contains all of the computations that refer, directly or indirectly, to the data structure. Because of a node's relationship to its predecessor and the similarity criterion applying to the computations it contains, each node represents a unique computation on the data structure. For instance, in a star diagram for `*line-storage*` (Figure 2), the path

`*line-storage*` → `list-ref` → `length`

denotes all Scheme expressions of the form

`(length (list-ref *line-storage*` *line-number* `))`

which compute the number of words on the specified line of text.[2]

Consequently, each node in the tree represents the implementation of a potential abstract operation on the data structure. Nodes near the left of the tree are expressions or statements that could implement "low level" operations encapsulating little more than the data structure representation itself. Nodes farther to the right could implement "higher level" operations that also encapsulate how some semantic properties are realized from the data structure. Stacking also reduces the size of the star diagram and eases collectively manipulating similar computations. As an example, in Figure 2 the

`*line-storage*` → `list-ref` → `length`

[2] We will usually use the arrow shorthand given above when referring to individual nodes in the star diagram. Because many nodes can have the same label, this approach allows us to unambiguously specify a node.

path is a path of stacked nodes whose corresponding expressions come from the functions `cssetup`, `csword`, and `csline`. Despite their use in different contexts, the three corresponding expressions all represent computing the number of words in a line of text.

The function definition nodes reveal where the uses of the variable occur without requiring perusal of the source code. Other definitions of star diagrams have included file nodes to provide additional context [Griswold et al. 1996]. Context beyond a function definition or file (such as what calls a function) is not shown because, in our experience with Scheme and C, the benefits of the extra context are outweighed by the disadvantage of making the view much larger (see Section 5). Object-oriented languages, with their much smaller methods, may require a different treatment.

The star diagram provides additional hints to the tool user about the program and the types of operations that can be performed on a given node. Function call nodes have thick borders to highlight the possibility that interesting computations may be hidden in the called function. Such hidden computations can be manipulated by one of the inlining transformations. Edges in the star diagram that correspond to a flow through a temporary are highlighted in purple to indicate that the computations are not actually nested. Because function definition nodes represent declarations rather than calculations, we represent them as light blue parallelograms to make them distinct yet less obvious. Because the location of a parameter within an operation often affects how the operation affects the parameter, node labels for built-in operations such as `set!` and `if` indicate the parameter position of the contained expression with square brackets.

2.1.2 *Direct Manipulation of the Star Diagram.* To allow manipulating the program through the star diagram, the tool user uses a direct-manipulation interface [Hutchins et al. 1985; Schneiderman 1983] to specify a restructuring change. The interface supports three common tasks: navigating to the source code view to explore details that are not visible in the star diagram, performing restructuring transformations, and measuring progress in the encapsulation task.

To navigate to other views of the program such as the source code view, the tool user double-clicks on a node, which triggers the appropriate highlighting and scrolling. When multiple statements are associated with a node, the first is shown immediately in the text view; the tool user can step through the rest of the occurrences.

To help the tool user quickly identify and perform transformations, the star diagram lists the available transformations in a button bar just above the star diagram graph. When the tool user selects a node in the star diagram, the tool highlights the transformation buttons that can be applied to it. Because each transformation operates on only certain types of programming constructs, this feedback helps the tool user identify valid manipulations. Less frequently used actions appear in the menus along the top of the window.

The star diagram's interface also helps the tool user measure progress in the task by separately displaying what parts of the encapsulation have been completed and what has yet to be done. In particular, completed interface functions for the ADT being created appear in the list on the left hand side of the star diagram window.

Also, the bodies of these functions are no longer depicted in the star diagram view. Depicting progress in the task in this way facilitates focusing on the remaining computations of interest and also indicates completion of the task when the star diagram is empty.

2.1.3 *Restructuring Operations in the Star Diagram.* To perform a restructuring action on the code associated with a star diagram node (box or parallelogram), the tool user selects the node, and then presses the button corresponding to the desired restructuring transformation.[3] Dialog boxes prompt the tool user for additional parameters to the transformations, such as the name and parameters for a function to be created. Once the tool user has specified any additional parameters, the transformation immediately checks to ensure that the changes applied to the program will preserve meaning, and if so, then the transformation manipulates the source code of the program. The restructuring tool then updates the star diagram window to show the current state of the program. As the tool user exposes functions that are desired as members of the ADT's interface, the programmer uses the Move Into Interface transformation to move the function's definition into the module interface and place the name of the function in the list of interface functions. Because all uses of the variable being encapsulated that occur inside that new function are deemed to be encapsulated, the variable uses are removed from the star diagram. Consequently, the star diagram explicitly maintains a list of what uses of the variable have been encapsulated, and which have not. The Remove button at the top of the scrolling list removes previously-added functions from the interface if the tool user finds that a function should not be in the interface.

The star diagram supports six transformations that are tailored to the needs of data encapsulation. Other transformations can be applied from the source code view.

Extract Function. Extracting a computation (or a set of similar computations) makes the associated code the body of a new function, and replaces the occurrences of the computations with calls to the function. (Figure 1 shows the effect of the transformation.) This transformation resembles the text-based restructuring tool's extract function transformation, but infers parameters to the transformation such as the destination location of the function. The tool user provides a name for the function; the tool can infer required parameters, but lets the tool user customize them to get the exact interface desired.

In the star diagram, a path from a node to the root denotes a computation on the data structure, and so selecting a node for extraction creates a new function encapsulating the computation on the data structure. If a stack of nodes is selected, then all of the computations denoted are extractable as a single function definition. The tool user can then opt to exclude any of the computations from the extraction, if necessary. Unlike selections in the text view, a selected node can denote a non-contiguous section of code, so the tool may be required to "gather" statements together with a move transformation before extracting them.

[3]The names chosen for the transformations were chosen in order to be suggestive to typical programmers. Many of these transformations are known by other names in the program derivation [Partsch and Steinbruggen 1983] and program optimization [Aho et al. 1986] literatures.

Inline Function. Inlining a function call replaces the call with the body of the function. Inlining is the opposite of extracting a function; when a function definition inappropriately encapsulates a computation, a possible solution is to inline the function at one or more call sites.

When the transformation is applied to a function call node in the star diagram, only the selected calls are inlined. When applied to a function definition node, all calls of the function are inlined and the definition is deleted. The inlining of a call node has the visual effect of replacing the node with the portions of the called function body that use the data structure.

Extract Parameter. Extracting an expression from a function body (1) adds a parameter to the function definition, (2) replaces the expression with a reference to the new parameter, and (3) changes all calls on that function to pass the expression's result as an argument.

In the star diagram, such an extracted expression always contains a reference to the data structure being encapsulated. Consequently, this operation moves the context of the data structure computation from its current function body to the function's callers. In effect, the operation moves uses of the variable "up" the function call hierarchy of the program.

Inline Parameter. Removing a parameter from a function definition results in replacing the references to the parameter with the expression that was passed as an argument. If there are multiple calls on the function, all the arguments for the parameter must be the same expression.

In the star diagram, the argument expression being inlined always contains a reference to the data structure being encapsulated. Hence removing the parameter has the effect of "pushing" the data structure reference inside the function body and down the call tree so that the function body now encapsulates it.

Rename Function. Renaming a function definition or call node changes the name of the function. The transformation allows the tool user to change function names to better reflect the abstraction represented by the new ADT.

Make Call. This transformation allows the programmer to convert an expression that matches an existing function into a call to the similar function. The transformation replaces the selected expression with a call to the function, and identifies what subexpressions become parameters to the function call.

2.2 Example: Encapsulation of the Line Storage in KWIC

In order to show how the tool user performs encapsulation with the star diagram, we demonstrate the encapsulation of the `*line-storage*` data structure in Parnas's KWIC Index program.

2.2.1 *Parnas's KWIC Program.* Parnas's KWIC indexing program creates a permuted index of a text file [Parnas 1972]. Figure 3 shows a sample input and output for the KWIC indexing program. KWIC's source code is provided in Appendix A.

The initial implementation of KWIC is written using a functional decomposition design. In a functional decomposition, the programmer creates functions based on a stepwise refinement of the task to be performed, breaking the large problem

Input: text file	Output: permuted index
	`brown the quick`
	`dog lazy`
	`fox jumped over the`
`the quick brown`	`jumped over the fox`
`fox jumped over the`	`lazy dog`
`lazy dog`	`over the fox jumped`
	`quick brown the`
	`the fox jumped over`
	`the  quick brown`

Fig. 3. Sample execution of the KWIC Index program.

into component tasks or phases [Bergland 1981; Stevens et al. 1974]. The program operates by invoking four major routines when creating the index. First, the program reads the input text file into memory, then "rotates" the lines to create the circularly-shifted lines. The program then sorts the lines and finally outputs the index.

One side effect of this functional decomposition is that the routines communicate with each other via shared data structures. Parnas points out that a weakness of a functional decomposition is that any change to the data structure format requires global changes to the program. In this case, three global arrays—`*line-storage*`, `*circ-index*`, and `*alph-index*`—allow the routines to share information. The text file is stored in the `*line-storage*` array, with one line per array entry. Because the entire file has to be in memory, the size of text files to be indexed must be smaller than the computer's memory. We would prefer KWIC to handle arbitrarily large files, and so consider redesigning `*line-storage*` so that lines in the text file are always accessed from disk, or by providing a caching mechanism to keep some lines in memory.

Rather than immediately making the change by finding every occurrence of `*line-storage*` throughout the code and making the appropriate change, it is preferable to exploit the benefits of information hiding by first restructuring KWIC to encapsulate the `*line-storage*` data structure inside an abstract data type. A set of interface functions will then hide the internal representation of the data structure from the rest of the program.

2.2.2 *Encapsulating* `*line-storage*` *with the Star Diagram*

Extracting two line-oriented operations. To begin encapsulating the list `*line-storage*`, the tool user creates a star diagram for the list. Figure 2 shows the star diagram in the midst of restructuring KWIC. (Three transformations—all Inline Parameter—were performed before this point to shorten the example.) Following the bottom path in the star diagram starting from the far left, the tool user can see that `*line-storage*` occurs frequently inside a list reference (i.e., `list-ref`), and these instances can be further subdivided into identifying the length of an element via `length` and accessing a member of the list element by another `list-ref`. The stacking of the nodes attracts the tool user's attention because these frequently-occurring expressions may represent likely interface operations. In the former case, the name of the operation `list-ref` as applied to `*line-storage*`, followed by a call to length, suggests that perhaps a line is being accessed and then the number of its components is being computed. Likewise, in the latter case, the second list-ref connected to the right of (i.e., enclosing) the first list-ref suggests that one of the components of the line is being accessed. In addition, the tool user sees that these accesses are contained within functions called cswords, csword, and csline. From this information, the tool user is tempted to extract two operations called words-in-line and get-word-in-line, respectively. However, before attempting the change, the tool user examines the source code. By double-clicking on each of the node stacks, the text view scrolls to the program text associated with the star diagram nodes. For each potential function, the tool user's hypotheses are confirmed.

To extract the first node stack as a new function words-in-line, the tool user clicks on the length stack and then presses the Extract Function button at the top of the star diagram. The tool textually displays all the instances denoted by the node stack, allowing the user to select which instances are truly representative of the new function. In this case, the tool user decides to replace all of the instances with calls to the new function. The tool then displays a dialog box prompting the tool user for a name for the new function (i.e., words-in-line) and for parts of the calculations to be made parameters to the new function (see Figure 4). In this case, there is one parameter; the variable line identifies the line of interest. The tool user fills in these values and chooses a name for the parameter, then clicks on the Extract button in the dialog box. At this point the underlying restructuring tool checks that the transformation will preserve the behavior of the original program, which the tool finds is true. Then the new function is created and the selected computations are replaced with calls to the new function. Finally, the star diagram is redrawn. The newly extracted nodes are shown on a path to the new function node, and the connections to the right of the length node disappear because they now reference *line-storage* through function calls. Because the tool user wants this new function to be part of the module interface, the words-in-line definition node is selected and the Move into Interface operation is invoked, removing the operation and its path from the star diagram and placing its name on the interface list.

A similar sequence is performed for extracting get-word-in-line. Figure 5 shows the resulting star diagram. The star diagram for *line-storage* is now roughly half the size, permitting the tool user to focus on the four remaining unencapsulated instances.

Modifying the interface of an existing operation. The tool user notes that three of these remaining accesses are built-in Scheme operations, and one is a call of a user-defined function (allwords). The user-defined call stands out, so the tool user decides to focus on it. It is seen that the result of allwords is bound to a variable called numcslines, and so the tool user suspects that allwords computes a count of all the words in the entire line storage. To confirm the hypothesis, the tool user navigates to the code for allwords, and sees that it indeed calculates the number of words in the argument list. At this point, the tool user decides that such an operation is a natural member of the interface of the Line Storage module, and hence should encapsulate *line-storage* rather than receive it as an argument.

Fig. 4. After selecting the length node in the path `*line-storage*` → `list-ref` → `length` and specifying that all four occurrences should be extracted as a new function, the tool prompts for the new function name and the parameters for the new function (as seen above). Once the parameters and function name are specified, the transformation is performed.

Fig. 5. Star diagram after the functions `get-word-in-line` and `words-in-line` have been extracted.

Fig. 6. Star diagram after Inline Parameter has been applied to `allwords`, but before function is placed in the interface.

The tool user performs this change by applying the Inline Parameter transformation on the `allwords` call node. The tool then checks that all calls on `allwords` pass the same expression, which in this case is trivially true because there is only one call on it. The tool then removes the argument in the call to `allwords`, removes the parameter from the definition, and replaces the use of the parameter with the argument, `*line-storage*`. Finally, the star diagram is redrawn with a path from `*line-storage*` to the `allwords` definition node, replacing the path to the call node. Figure 6 shows the star diagram after this transformation. Now `allwords` hides a use of `*line-storage*`, so the tool user moves `allwords` into the module interface, which removes its path from the star diagram and puts its name in the interface list.

Completing the encapsulation. The remaining instances are handled similarly to the first two extractions. The `length` reference to `*line-storage*` is verified to compute the number of lines in the list, and it is extracted as `lines-in-file`. The remaining two references appear inside the `insline` definition, which is determined to add words to the list during initialization. The tool user decides that the `insline` function belongs in the interface of the Line Storage module, and moves it into the module interface. This last transformation leaves the star diagram empty, indicating that the `*line-storage*` data structure is fully encapsulated in the Line Storage module by the functions in the interface list: `lines-in-file`, `words-in-line`, `get-word-in-line`, `allwords`, and `insline`.

Because of the resulting encapsulation, the desired enhancement—support for putting lines in secondary storage—can be performed by changing only functions in the Line Storage module. For instance, `insline` can be changed to precompute and save the values formerly computed by `allwords` and `lines-in-file`. Function `get-word-on-line` can be changed to read the given line from disk, then return the requested word.

2.3 Discussion

The KWIC example shows that the star diagram visualization supports planning an encapsulation, maintaining knowledge of the current state of the transformation, and identifying what actions are possible at any given time. Because of their complexity, these issues were explored more thoroughly in a separate programmer study [Bowdidge and Griswold 1997].

The example also shows how the star diagram satisfies our design criteria. In particular:

Provides global, detailed view. The tool user sees all of the uses of the data structure in a single, condensed view, rather than having to read through the entire source code. Because the tool focuses on the task of encapsulation, the condensed view contains the information essential for the tool user, grouping relevant expressions normally scattered throughout the code.

Elides unimportant details. The source code represented by a star diagram node is normally invisible to the tool user, but can be viewed by clicking on the node. When restructuring decisions require understanding a particular statement in detail, the tool displays the appropriate text. For example, when selecting parameters for a new function, the dialog box for selecting parameters displays the relevant line.

Groups similar statements. Contextual information in the diagram helps identify properties that will drive the design of the abstract interface. Stacks of nodes highlight frequently occurring computations that are candidates for becoming part of the ADT's interface. Also, references to the data structure at different abstraction level than other operations in the diagram (`allwords`) indicates an inconsistency in the existing interface. The names of the operations in the star diagram provide additional informal knowledge about the program, and can allow the programmer to guess at the behavior of a given statement while looking at the "big picture" provided by the star diagram.

Supports restructuring. All the needed transformations could be applied through the star diagram.

The example also demonstrates that the star diagram accrues several benefits by being implemented on top of the existing meaning-preserving restructuring tool. First, the meaning-preserving nature of the tool guarantees that errors are not introduced by the restructuring transformations, freeing both the view and the tool user to focus on the encapsulation. Second, all views in the system—source code, star diagram, and any other visualizations—are updated after every transformation, giving the tool user immediate feedback on the progress and state of the encapsulation. Finally, the availability of other views through navigation from the star diagram allows the star diagram to be specialized for encapsulation without compromising other restructuring tasks.

3. DESIGN AND CONSTRUCTION OF STAR DIAGRAMS

For the star diagram to be practical, it must be applicable to languages in wide use. The star diagram must also be simple to create and draw, especially if the concepts are to scale to large programs. Because some encapsulation activities will manipulate large abstractions or have task-specific requirements, the star diagram concept must be adaptable to the task at hand. For example, the tool user might wish to customize the star diagram to highlight pertinent details, indicate likely restructuring operations, label concepts in the code rather than just operations, or hide expressions irrelevant to the encapsulation.

These needs are addressed by defining the star diagram in terms of a generic, customizable algorithm that operates on universal program data structures such as the abstract syntax tree (AST) and data flow def-use chains. We describe the role these representations play in deriving the star diagram, the generic algorithm, and the instantiation of the algorithm for Scheme programs.

3.1 Representations Needed to Build the Star Diagram

The representations chosen to create the star diagram must generate the program relationships necessary for the star diagram, and should be efficient enough to allow creation of star diagrams for large programs.

The star diagram represents relationships between expressions in the program text by two visual cues: the linking of nodes and the stacking of nodes. Linking of nodes implies one expression is consuming or controlling the other. Stacking of nodes implies that the underlying expressions are computationally similar. The star diagram highlights these relationships because they act as informal measures of the cohesion between two statements, and can guide the tool user to place related expressions into the same function or module as part of the encapsulation task.

Both can be partially generated from the abstract syntax tree (AST), a parsed representation of the program that represents compound expressions or statements as a tree. The AST is easily constructed during a parse of the program. An AST can be ten times larger than the source text it is derived from, but since each star diagram arm only represents the computations within a single function, only a small portion of the AST is required at any time. Relatively inexpensive dataflow information can be used to identify additional cohesive expressions not identified via the AST.

The linking relationship is captured directly in the AST. An expression is represented by a subtree. The consumer of an expression's result (or controller of its evaluation) is its parent. Thus the linking relationship is essentially the parent relationship in the tree. However, the result of an expression is sometimes stored into an intermediate variable in order to break up large expressions into manageable pieces. For instance, the following uses `y` as a temporary, thus hiding the fact that `x` is indirectly consumed by `printf`:

```
int x,y;
x = 0;
y = x + 3;
printf("%d\n",y);
```

Consequently, the parent relationship in the AST does not always directly identify the consumer of the expression's result. In such cases, def-use chains derived from data flow analysis [Aho et al. 1986] can be used to traverse to the users of the variable. A variable is classified as a temporary only if it is local to the current scope; other variables are very likely part of some abstraction. This restriction also limits the cost of the def-use analysis.

Two nodes are stacked in the star diagram if (1) their predecessors in the diagram are stacked, (2) their nodes represent the same operation, and (3) the predecessors'

AST nodes are treated similarly by the operations (e.g., both predecessors are in the conditional portion of an if statement). All three attributes are easily calculated. The first is determined directly from the star diagram. The second can be checked by testing operators stored in the AST nodes for equality. The third is verified by checking the locations of the predecessors' AST nodes within their parents. These stacking criteria require only that the operations and their consumption relation match; no check is made on the remaining arguments. However, stronger definitions of similarity could be useful. For example, the star diagram could distinguish between setting a variable to zero and setting a variable to any other value.

3.2 Generic Algorithm for Generating a Star Diagram

At a high level, the star diagram algorithm creates the star diagram primarily by examining the AST. To demonstrate the algorithm, we present a small example. Figure 7 shows some fragments of code using the `*line-storage*` variable, the corresponding AST, and the resulting star diagram. To generate a star diagram, the algorithm first parses the source code and generates the abstract syntax tree. Next, the algorithm finds each use of `*line-storage*` in the program, and moves from the AST node representing the use of the variable towards the root of the AST. When it reaches the root of the AST, one arm, or path, through the star diagram, has been completed. The algorithm combines similar prefixes from each new arm of the star diagram with the existing star diagram to generate the stacks of similar expressions.

Our detailed algorithm, presented in Figure 8, extends this description of the star diagram by including details such as how the algorithm generates labels, what kinds of nodes in the AST get displayed in the star diagram, and so on. In this algorithm, six operations are parameterized by calls to customizable functions: **Roots**, **Successors**, **EndNodeTest**, **SkipNodeTest**, **SimilarNodeTest**, and **Label**. These functions represent the components of the star diagram algorithm. By changing the implementation of the functions, the appearance of the star diagram and the details that it displays can be customized to support the star diagram user's wishes. We describe each function in turn.

Roots. The **Roots** function returns the list of AST nodes that the root of the star diagram will represent.

In the Scheme star diagrams, the root of the star diagram is defined as representing all uses of a variable throughout the program, and thus the star diagram shows the computations consuming the values in the root. To generate the star diagram, the tool user selects any use of the variable. The star diagram generator finds the definition of the variable, then finds all uses of that definition.

Successor. The **Successor** function is a heuristic based on our expectations of what counts as the "consumer" or "controller" of a given operation's result, or what statements should be considered cohesive with respect to the ADT being created. Normally, the successor node in the star diagram corresponds to the parent node in the AST, except that scope constructs are normally ignored. If no consumer or controller can be found, then the **Successor** function returns the AST node representing the syntactic scope in which the computation exists.

The major exception to the parent heuristic is for assignments. If the expression

b) abstract syntax tree

a) source code

```
void cssetup() {
 ...
 numwords=length(lineStorage[line]);
 ...
}

void csword () {
 ...
 wordNumber = wordNumber %
        length(lineStorage[line]);
 ...
}
```

c) resulting star diagram

Fig. 7. Generating star diagrams from the abstract syntax tree. (a) Sample fragment of code. (b) Parts of an AST representing the sample code. (c) Part of the resulting star diagram. To generate the star diagram, we start by finding every use of a given variable (in this case, `*line-storage*`), then move up the tree, combining similar nodes into stacks of nodes in the star diagram.

containing the variable of interest occurs on the right-hand side of an assignment statement, then the algorithm assumes the computation is continued where the assigned variable is used. We use the def-use chains encoded as dependencies in the PDG to map from an assignment of a variable to the locations where the value may be consumed. If the node represents an assignment to a global variable, then the consuming calculations are probably non-cohesive, scattered throughout the code, and do not represent a single local calculation to be turned into a function. **Successor** returns the enclosing scope and ignores the dataflow relations.

SkipNodeTest. **Successor** ignores certain nodes that are irrelevant to encapsulation and should not be displayed. These are identified by **Successor** with **SkipNodeTest**. AST nodes that are ignored fall into two categories. Some nodes in the AST represent syntactic constructs such as declarations that are redundant or are irrelevant to the "references" or "controls" relation. For example, `let` scopes have a binding list that contains all the variables declared by the *let*. Such a construct does not consume or control the expressions in the variable bindings, nor is it independently manipulable. Moreover, a binding list always occurs with the `let` itself, and so is somewhat redundant.

EndNodeTest. The algorithm stops generating an arm of the star diagram when **EndNodeTest** evaluates to true on the current AST node. In Scheme, the

```
DrawStar() {
    DrawGraph(GenerateStar(Roots()));
}

/* Given a star diagram node, GenerateStar() calculates the children of the given star
diagram node, then calls itself recursively for each child. Returns the completed graph. */
GenerateStar(starDiagramNode) {
    astNodes = starDiagramNode→astNodes; /* expressions represented by node */
    /* Remove AST nodes that represent the end of a star arm. */
    remove-if (astNodes, EndNodeTest);
    successors = (∀ astNode: astNodes) {
        /* Ignore irrelevant AST nodes when moving up the AST tree. */
        do {
            element = Successor(element);
        } while (SkipNodeTest(element));
    }
    /* Divide successor AST nodes into sets of successor star diagram nodes nodes using using SimilarTest */
    groups = Partition(successors, SimilarTest);

    /* Each group of successors represents a node in the star diagram */
    (∀ group: groups) {
        /* The first AST node in the group is representative of the type of all */
        if ((ASTNodeType(first(group)) == FunctionDefinition) &&
            (already created the star diagram node for the function def node)) {
                /* Add the corresponding function node to the list of children for the node. */
                insert(FunctionStarDiagramNode(first(group)),starDiagramNode.children);
            } else {
                allocate newStarDiagramNode;
                newStarDiagramNode.astNodes = group;
                /* Generate the label based on either the group, or in this case, on the first
                AST node in the group, assuming all AST nodes in the group have the same label. */
                newStarDiagramNode.label = Label(first(group));
                insert(newStarDiagramNode,starDiagramNode.children);
                newStarDiagramNode.parent = starDiagramNode;
                GenerateStar(newStarDiagramNode); /* Recurse until done. */
            }
        }
        return starDiagramNode;
}
```

Fig. 8. Pseudocode algorithm for generating star diagrams. The parameterized functions (**Roots**, **Successors**, **SkipNodeTest**, **EndNodeTest**, **SimilarTest**, and **Label**) are described in the text.

EndNodeTest returns true when the node is a function definition node or represents the top-level scope.

SimilarTest. The **SimilarTest** indicates whether two successors of the same node stack are similar and hence should be stacked together.

In Scheme, two nodes are usually stacked if the nodes' AST node types match. Variable uses and function call nodes require a stricter test; we check the name associated with the node to determine whether the accesses refer to the same variable or function.

Label. The label for a node should help the tool user quickly guess at the behavior of the expression. In Scheme, a label usually contains the operation, such as "plus", or "call: *function-name*". In function calls and operations where the parameter position in which the expression occurs might convey the nature of the use, the label indicates placement of the expression using the variable. For example, we chose to identify the location of parameters in the `set!` operation, and in `if` and `do` loops in the Scheme restructuring tool.

3.3 Implementation Details

The code for generating and manipulating the star diagram consists of about 2,000 lines of Common Lisp code. The direct manipulation subsystem is implemented with the PICASSO user interface toolkit [Rowe et al. 1991]. The star diagram graph is laid out using the ISI Grapher [Robins 1985]. The ISI Grapher draws attractive directed graphs, but trades off speed for output quality. Because the star diagram is primarily a tree representation, with arms combining only at the far right side of the diagram, the star diagram could be laid out with a tree layout algorithm, then have the function definition nodes added afterward [Chen 1996].

Although the description of the star diagram generation algorithm suggests that the tasks of extracting information from the AST and the construction of the star diagram are combined, our implementation actually performs the tasks separately, first finding every use of the variable, then creating a star "arm", the list of nodes representing an arm in the star diagram, and finally creating the tree or graph representing the star diagram.

4. USING THE STAR DIAGRAM ON REAL-WORLD PROGRAMS

Unless star diagrams can be created for large programs written in widely-used languages, the concept will not be useful in practice. To evaluate the applicability of the star diagram concept to other languages, we implemented star diagram generators for C and MUMPS, and constructed several star diagrams for complex, widely used data structures in programs written in these languages. We in fact implemented two types of C star diagram generators: star diagrams that assist encapsulation of a variable, and type-oriented star diagrams that assist encapsulation of a type. This section discusses the issues involved in creating the generators, and the next section assesses the scalability of the star diagram concept.

4.1 Star Diagrams for C Programs

C's structured syntax allows using the AST-based algorithm from Section 3 without modification, although the algorithm's six generic functions must be customized to accommodate language-specific details. The **Roots** function and **SimilarTest** functions both operate as in the Scheme algorithm. Because our implementations of C star diagrams have no access to dataflow information, the **Successor** function always finds the successor by moving up the AST. The **EndNodeTest** merely checks for the root of the AST. The **SkipNodeTest** excludes AST nodes considered irrelevant to encapsulation such as goto-label statements and compound statements.

We implemented a C star diagram generator using these functions. (Part of the implementation was later reused in Chen's star diagram planning tool [Chen 1996; Griswold et al. 1996].) We used a prototype C restructuring tool's AST infrastructure [Morgenthaler and Griswold 1995], and Moen's tree drawing algorithm [Moen 1990]. The AST served our needs because it supports finding all uses of a variable or type, and it is easy to traverse from the leaves of the tree to the root. We chose Moen's tree drawing algorithm because of its speed and compact layouts. The generator spends most of its time parsing and creating the AST. On on a Sun Sparc 10/61, parsing the program and generating a star diagram for a variable with 36 uses in a 1,700 line program requires about four seconds. For a variable with 2000 uses in a 75,000 line program, parsing the program takes 300 seconds, and generating the star diagram takes than 10 seconds.

4.2 Example of a C Star Diagram

We can demonstrate the C star diagram generator by planning the restructuring of `adventure`, a dungeon exploration game [Adams 1980]. `adventure`, a port of a BASIC program into C, consists of 1,700 lines of code, excluding comments and blank lines. The game allows the player to move around a virtual world, divided into *rooms*, in order to solve a puzzle.

`adventure` maintains a global array of `Room` structures, called `rooms`, with each entry containing a room description and an array of destination room numbers indicating where the player moves when exiting in a specific direction. This data structure is not encapsulated, but it is widely used throughout the program since it is the basis for all activities in the program. A change such as modifying the geometry of rooms—all rooms are modeled as cubes with six possible exits—would require changing every file in the system. Prior to such a change, then, we would like to encapsulate the data structure.

To plan this encapsulation, we can examine a star diagram for the `rooms` variable (see Figure 9). This star diagram represents all 25 uses of the variable throughout the program (each node displays the number of similar computations it represents). We can see `rooms` represents parts of an array abstraction because 20 of 25 accesses occur inside an array reference (represented by the `arrayRef` node), which occurs during a field access (one of the `getField:` nodes). So, even without the source code, we can identify that the `rooms` variable is an array, and it is an array of structures. Of the five remaining uses, one is an assignment to the variable. Examining the associated source code identifies it as the initial allocation of the array using `malloc`. The final four references,

`PtrDec` → `FullDec` → `top-level`

represent one actual and three external declarations (i.e., imports) of the variable. In hiding `rooms`, the actual declaration would be hidden in a new file and the external declarations would be removed, since all accesses would be through module

Fig. 9. Star diagram for the `rooms` array, with expressions labeled with their assigned abstractions.

interface functions.

Most of our attention is focused on the array references because of the number of occurrences (indicated by the stacked nodes). We can see that two nodes are connected to the array reference node, and both represent accesses to fields of a structure. We can surmise the first, the `desc` field, represents the room description because many of the accesses occur within the `DescribeRoom` function. The second, `dir`, is harder to diagnose. We could predict its behavior from a knowledge of the game's behavior, from the scope of some accesses (such as `DoGoAction` which probably handles the player's `MOVE` command), or from comments in the source code. We can see from the star diagram that the `dir` field must be an array, because it always occurs as part of an array reference. If we examine the code, we will find that each entry of the array represents exits from the room. Thus, we could decide at this point that a likely function for the interface of a module encapsulating the `rooms` array could be a "RoomInDirection" function that, given a current room and direction, returns the room number located through that exit. However, we can choose to examine deeper levels of the star diagram to see if other candidate functions exist.

Although the array references occur in what seem like a large number of contexts, we can quickly identify the contexts of interest. The call to `printf` and the `dereference` node within a call to `fscanf` occur in the function `ReadDataFile`, and are actually initialization and debugging code in the routines for loading a puzzle. These can either be replaced by accessor functions, or the initialization function can simply continue to directly access the internals of the `Rooms` module.

The other three contexts are more interesting; by examining the source code, we see that the "`!=`" node represents code testing if the direction out of a room is zero. The code represented by this star diagram node probably instantiates a test for the existence of an exit from the room. The test for equality (represented by the star diagram node directly below the inequality) tests for equality to zero. That test probably checks for an the non-existence of an exit. These represent complementary functions, and so can be created as two functions with a restructuring tool, and replaced after restructuring by a single call using a negate operation on one of the calls. The final case, assignment, appears in the source code to set the `currentRoom` variable to the new room when the player moves.

Uses of the `desc` field can similarly be examined through the star diagram. Three of the five uses occur in the `DescribeRoom` function, which suggests that there are not three simple computations occurring, but one complex operation using the variable in different contexts. We can examine the source code to confirm this hypothesis:

```
if ((rooms[currentRoom].desc)[0] == '*') {
    PrintDescription(rooms[currentRoom].desc+1);
} else {
    PrintDescription("I'm in a ");
    PrintDescription (rooms[currentRoom].desc);
}
```

From reading the source code and comments, we can identify the behavior. This code prints out the description of the room; a star at the beginning of the name suppresses the usual "`I'm in a`" prefix before printing the room description. The `if` statement thus represents the "PrintRoomDescription" abstraction, while the existing `PrintDescription` function actually represents describing the room as a whole—the items in the room, the room's description, and other details of the setting. This `if` statement occurs in three of the five branches off the `desc` node in the star diagram. The other two uses, the assignment to the description field and the `printf`, again perform initialization and debugging in the initialization function.

We can thus use the knowledge we have gleaned from the star diagram to propose an interface for a module encapsulating the `rooms` variable. The uses in the initialization function can be replaced with a reader and writer function, or the code for reading the portion of the input file describing the puzzle's rooms can be converted into a "LoadRooms" function. A "PrintRoomDescription" function can be extracted from the `DescribeRoom` function. We can then create "CanMoveDirTest", "CannotMoveDirTest", and "GoDirection" functions from the uses of the `dir` field. Figure 9 highlights the nodes in the star diagram that represent the functions in an interface for a `rooms` abstract data type.

The `adventure` example shows us that understandable star diagrams can be created for the C programming language. The star diagram provides the information needed to plan encapsulations, and displays an overview of how the represented code behaves, as evidenced by our ability to identify the rough structure of the `rooms` variable from this view. The star diagram for `rooms` also compares favorably to other approaches to plan encapsulations in C programs. Although the diagram takes more space than the eighteen lines of `grep` output needed to display all uses of the variable (Figure 10), the star diagram also groups similar expressions and allows us to quickly perceive a complete interface.

```
filen.c:83:extern Room *rooms;
filen.c:224:    rooms = (Room *) malloc((numRooms+1)*sizeof(Room));
filen.c:226:        fscanf(file,"%d %d %d %d %d %d",&(rooms[i].dir[0]),
filen.c:227:            &(rooms[i].dir[1]),&(rooms[i].dir[2]), &(rooms[i].dir
[3]),
filen.c:228:            &(rooms[i].dir[4]),&(rooms[i].dir[5]));
filen.c:229:        rooms[i].desc = GetAString(file,240);
filen.c:231: printf ("Room: %s  %d %d %d %d %d %d\n",rooms[i].desc,
filen.c:232:    rooms[i].dir[0],rooms[i].dir[1], rooms[i].dir[2],
filen.c:233:    rooms[i].dir[3],rooms[i].dir[4], rooms[i].dir[5]);
saadvn.c:79:extern Room *rooms;
saadvn.c:102:Room *rooms;
saadvn.c:194:     if ((rooms[currentRoom].desc)[0] == '*') {
saadvn.c:195:           PrintDescription(rooms[currentRoom].desc+1);
saadvn.c:198:             PrintDescription (rooms[currentRoom].desc);
saadvn.c:215:           if (rooms[currentRoom].dir[i] != 0) {
saadvn.c:732:     if (rooms[currentRoom].dir[noun-1] == 0){
saadvn.c:745:       currentRoom = rooms[currentRoom].dir[noun-1];
saveFilen.c:69:extern Room *rooms;
```

Fig. 10. Output of grep when used to find occurrences of the rooms variable.

4.3 Type-Oriented Star Diagrams

The star diagram generators described up to now only supported encapsulating a single data structure instantiated as a variable. Although some programs only require the encapsulation of a single global variable, other scenarios include encapsulating a type or encapsulating a data structure made of two variables. The star diagram concept can be extended to handle both cases.

Typed star diagrams can be constructed for C by changing the **Roots** function to find all uses of all variables of a given type, and changing the **SimilarNodeTest** function to stack the root nodes even when the roots represent variables with different names. In a language such as C, type star diagrams could miss operations upon a data structure if the structure is cast to a different type before beginning the operation. By treating typecasts as basic operations on the data structure, however, the casts appear in the star diagram when the object of the examined type is cast to the new type, and can be manipulated like other expressions. Fast points-to analysis [Steensgaard 1995] can also be used to infer such relationships so that type cast variables can be directly included in the star diagram. Figure 11 shows an example of a type-oriented star diagram for C.

Not every variable of a type is necessarily an instance of a common abstraction. For example, an abstraction such as "error messages" might be represented by the type pointer to character (or String), but not all strings are error messages. The star diagram could avoid false matches by permitting the tool user to selectively exclude instances of the type.

Type-oriented encapsulation results in potentially more informative diagrams. Consider the `allwords` call from the example of encapsulating `*line-storage*` in Section 2.2. In that case, only the function call node for `allwords` was shown in the star diagram, but not its body, requiring the tool user to examine the contents of `allwords` with the text viewer to see how `*line-storage*` is manipulated within

```
int numberOfChildren (TreeNode *node) {
  if (node->firstChild == NULL) {
     return (0);
  } else {
    int count = 0;
    TreeNode *child = node->firstChild;
    while (child != NULL) {
      count++;
      child = child->nextSibling;
    }
    return (count);
  }
}
```

Fig. 11. Type-oriented star diagram for a fragment of C code. The star diagram shows all the references to all variables of type TreeNode in the above source code.

the function. However, it might have been more helpful to actually see the body as part of the star diagram, because this would reveal overlaps with other computations. With type-oriented encapsulation, both the call and the body are in the diagram because the parameter is of the type of `*line-storage*`.

4.4 Language Features that Complicate Creating a Star Diagram

Some features in C and in other languages complicate creation of a star diagram. First, text-based macro languages that must be preprocessed before parsing hide macro abstractions from analysis tools such as the star diagram that rely on parsing. Techniques for parsing certain classes of macros can reduce this problem, but never eliminate it [Griswold et al. 1996].

Second, the structure of a language's grammar may not allow the star diagram algorithm to infer data and control dependences from nesting relationships. For example, MUMPS is a a line-based, interpreted language similar to BASIC. Because statements are parsed in a line-by-line fashion, the AST does not directly connect statements normally nested in the C or Scheme programs. For example, when a `for` loop extends across separate lines, portions of the `for` on the next line occur in a syntactically independent unit. As a result, the successor of a given node in the star diagram cannot be found simply by moving up the AST, but sometimes requires the star diagram algorithm to examine the code—in effect identifying control dependencies—to identify the matching statement. Creating star diagrams for an assembly language program requires both control and data flow analysis to order calculations and identify where a register or value is used next.

5. SCALABILITY OF THE STAR DIAGRAM CONCEPT

While small examples can show the strength of the star diagram concept, programmers must be able to use the star diagram to restructure large systems. Specifically, the star diagram must not become too large or complex to assimilate. Thus, we must identify how the overall size of star diagrams increase for a varying number of variable uses.

To determine the scalability of the star diagram concept, we compared the sizes of the star diagrams for frequently and infrequently used variables. (The size of the source code is irrelevant to the size of the star diagram since its height is governed by the number of uses, and its length by the complexity of nested expressions in

program	LOC	variable/type	variable references	size of grep output (lines)
KWIC	200	`*line-storage*`	10	10
Adventure Interpreter	1,400	`currentRoom`	36	34
		`items`	83	78
		`rooms`	25	18
GNU Emacs	76,000	`minibuf_window`	48	65
		`current_buffer`	1,909	1,354
		`struct buffer`	2,409	N/A
CHCS (MUMPS)	590,000	`DIC`	46,613	43,257
SunOS (UNIX)	88,000	`struct mbuf`	1,035	N/A

Table 1. Summary of the programs and variables examined using star diagrams. All are C programs, except KWIC (Scheme) and CHCS (MUMPS). All star diagrams display uses of a variable, except for `struct mbuf` and `struct buffer`, which are type-oriented star diagrams. Because grep cannot identify all uses of variables of a particular type no grep figure is given for type-oriented star diagrams.

the program.)

We found the vertical size of large star diagrams may be too large for a star diagram user to easily assimilate without additional changes to the star diagram visualization. Although the size of the star diagrams is partially a result of the inherent complexity of the target variable or type, we can suggest a number of enhancements to the visualization that hides irrelevant detail and thus reduces the size of the star diagram.

5.1 Measuring the Size of Star Diagrams

We investigated a variety of star diagrams from Scheme, C, and MUMPS programs, including both variable- and type-oriented star diagrams (see Table 1). For Scheme, we examined the star diagram in Figure 2, the largest star diagram in the `KWIC` example. For the `adventure` program (described in Section 4.2), we examined the integer variable `currentRoom`, the infrequently used array `rooms` described in the example, and the frequently used array `items`. We also created three star diagrams for the `emacs` editor [Stallman 1993]. Two star diagrams displayed variables in `emacs` (`minibuf_window` and the widespread `current_buffer`). We also created a type-oriented star diagram for all the pointers to `buffer` structures. To compare the star diagrams for C and MUMPS programs, we examined the `DIC` variable from portions of the Comprehensive Health Care System (CHCS) source code, a half-million line hospital management system [Griswold and Atkinson 1995]). Finally, we created a star diagram for all variables of type `struct mbuf` in the SunOS networking code.

Table 1 lists the variables examined in each of these programs, the number of references to the variable of interest throughout the program (or, in the type-oriented star diagram examples, the number of uses of variables of the given type), and the analogous `grep` for variable star diagrams. Table 2 describes the overall size of the star diagram, both in the pages required to draw the star diagram and number of nodes in the star diagram. To judge the amount of compression accomplished by

program	variable	size (pages) rows	size (pages) cols	nodes	nodes unmerged	distinct arms	arm length max.	arm length avg.
KWIC	*line-storage*	1	1	37	55	13	9	5.5
Adventure Interpreter	currentRoom	1	1	77	176	24	7	5.1
	items	2	1	126	493	39	9	6.6
	rooms	1	1	38	151	12	7	5.7
Emacs	minibuf_window	2	2	139	279	46	11	5.1
	current_buffer	26	3	3,633	16,410	932	27	10.6
	struct buffer	29	3	3,741	17,893	1,029	25	9.6
CHCS	DIC	20	2	2,440	281,632	698	18	7.5
SunOS	struct mbuf	11	2	1,218	6,505	393	13	4.2

Table 2. Measurements of the size of star diagrams in pages, number of nodes, number of arms, and length of arms.

stacking nodes, we measure both the total number of nodes in the diagram, and the number of "unmerged nodes" if we performed no stacking or matching of common expressions. The "number of pages" figures describe the size of a printed version of the graph. For large graphs that cannot be displayed on a single sheet, we print multi-page graphs that can be taped together and displayed on a wall. Pages are printed in portrait mode. Our choice of font and spacing allows us to fit about 36 star arms vertically on a page, and 10 nodes of a star diagram arm horizontally.

5.2 Observations on Measurements of the Star Diagram

First, it appears that star diagram arms are usually short enough that the programmer can see much of an arm without scrolling horizontally. The length of star diagram arms in the sample stars (Table 2) are typically small. (This was expected. The length of arms is not dependent on program size, but rather on the degree of nesting of expressions and statements in a function, which is small for most programs.)

The average length of arms in the star diagram is always under ten nodes, and usually under seven. Because our star diagram printouts usually fit ten nodes on an 8 1/2" wide paper, and the Scheme star diagram generator can easily show twelve to fifteen nodes on a 19" computer screen, we can assume that a significant fraction of a star diagram graph can fit horizontally on paper or screen.

Although the average measurements suggest that the star diagram's horizontal component will usually fit on a page or screen, the maximum lengths of star diagram arms (shown in Table 2) always appear to require two or three paper pages to display. When the length of the star diagram arms becomes larger than a screen width, the star diagram tool user can no longer see the right hand side of the star diagram, and will be forced to either scroll to see the necessary information, or plan the interface with imperfect information.

These long arms are usually irrelevant. Some long star diagram arms are caused by language structures such as long if-then-else sequences, which create uninteresting star diagram arms. Most other long star arms are caused by the complexity of expressions and functions in the code, and for these cases, the outside star arms usually represent a single, large expression irrelevant to the encapsulation. Thus,

Program	variable	Level of graph: 1	2	3	4	5	6	7	8	9
KWIC	`*line-storage*`	5	6	7	5	6	4	1	1	1
Adventure Interpreter	`currentRoom`	1	9	16	17	15	11	7		
	`items`	1	3	6	19	29	28	20	14	5
	`rooms`	1	3	3	7	10	8	5		
Emacs	`minibuf_window`	8	20	19	23	27	16	11	7	33
	`current_buffer`	4	63	182	243	304	331	300	263	224
	`struct buffer`	1	71	218	288	366	393	337	281	221
CHCS	`DIC`	2	25	112	274	423	479	413	290	182
SunOS	`struct mbuf`	166	265	290	228	132	67	35	17	9

Table 3. Measurements of height of star diagrams, in number of nodes at each level of the graph.

simple truncation can remove any scaling problems caused by the length of star diagram arms.

By contrast, the height of the star diagram is mainly dependent on the number of arms in the star diagram. Because the vertical component of a star diagram is bound in the worst case by the number of references to the variables of interest, the star diagram's height increases quickly. Because each variable use occurs in a slightly different context, the star diagram arms eventually diverge. Table 2 reveals that the number of distinct arms in the star diagram can be quite large—1,029 for `DIC`—typically amounting to about half the number of variable uses being examined. The height of the graph closely matches the number of arms; the maximum height of the graph is usually a large fraction of the number of arms. Thus, in the existing star diagrams, we do find that the vertical component follows the number of references being displayed (see Table 3).

In Table 2, we see how the size of each star diagram maps to the number of pages required to display the star diagram. For a variable occurring less than a hundred times, its star diagram fits on a single page. (Although Table 2 shows that `items` and `minibuf_window` require multiple pages, they only partially exceed one page, and can fit on a single page via slight reduction.) The larger star diagrams vary. For the `emacs` examples and for the `struct mbuf` example, the star diagrams require roughly the same space as `grep`—for example, displaying the uses of `current_buffer` requires 24 pages for the `grep` output and 26 pages for the star diagram. The `DIC` star diagram, with several hundred pages of `grep` output, fits on 20 pages, partially due to significant stacking of nodes.

5.3 Approaches for Improving Scalability

The above analysis suggests that the number of distinct arms is the major barrier to constraining a star diagram's size. Four approaches to managing the number of paths appear promising.

First, for larger star diagrams, the paper approach is promising because the programmer can more quickly scan across large expanses of the diagram. The largest star diagram example, `struct buffer`, fits on ten sheets of drafting paper. The star diagram user can still manage drawings of such a size because paper designs simplify browsing by overlaying, folding, annotating, and highlighting interesting portions of drawings. To make using paper easier, a method of mapping from a star diagram node to the relevant code (such as indexing nodes) in a star diagram tool could allow smoothly navigating from paper to the tool and back again.

Second, the layout algorithm could cluster portions of the star diagram to bring closely related nodes close together in the diagram, allowing the restructuring of large star diagrams in subsections. Although working on subsections could allow a programmer to make inappropriate choices by overlooking key information outside the subsection, the approach may be useful when the programmer believes that the subsections interact minimally. For example, the tool user could partition a star diagram according to the category of operation. In the star diagram for the `struct mbuf` example, we partitioned the star diagram into subsections for (a) field accesses, (b) manipulations of the `mbuf` structure itself, and (c) function calls passing `mbuf` structures. If field access operations are cohesive with other field access operations, the tool user can examine and restructure each subsection independently with minimal backtracking.

A third approach is to increase the stacking of nodes in the star diagram. Two ways are: (1) loosen the stacking criteria to cause as much overlap as possible, or (2) identify concepts larger than expressions, such as groups of statements conceptually performing a single action, and use these conceptual groups to form meta nodes.

Finally, we can avoid such scalability problems by displaying only the portions of the star diagram of immediate interest to the star diagram user. For example, the tool user could direct the star diagram view to omit unstacked (singleton) nodes. Table 4 suggests that giving the user control over omitting singletons could remove two thirds of the nodes in certain star diagrams. Also, we have observed that users of the star diagram tend to focus on the left hand side of the diagram, but occasionally examine the right hand side of the star diagram [Bowdidge and Griswold 1997]. The left hand side nodes represent the common, small expressions that are likely candidates for new functions, and the function definition nodes at the far right of the diagram provide the context of uses. Consequently, the middle of the diagram can be elided under user control to bring all the needed information onto the screen without loss of important details. Because a star diagram approaches its maximum height in just a few levels, major reductions in the size of a star diagram will only occur when pruned to the second or third level (Table 3). However, if the diagram can be interactively customized [Chen 1996; Griswold et al. 1996], then temporarily eliding the diagram to such a severe level need not compromise understanding the diagram as a whole.

5.4 Summary

The ability to effectively view and manipulate a star diagram depends upon a number factors, in particular the number of references in the root and the rate of branching in the root's successors. The length of the arms is a secondary factor. Without some sort of adaptation, the effectiveness on a star diagram is likely to be limited on the programs for which it is most needed. This is not too surprising; the problems encountered in these programs is inherent in their size, age, and consequent complexity.

Fortunately, the star diagram's definition can be extended and customized to allow changing the layout, stacking criterion, and the temporary truncation of the

Program	variable	number of nodes		size of average stack	
		stacked	singleton	all nodes	stacked nodes
KWIC	`*line-storage*`	4	33	1.5	5.5
Adventure Interpreter	`currentRoom`	30	47	2.3	4.3
	`items`	83	87	3.3	4.7
	`rooms`	16	32	3.6	7.2
Emacs	`minibuf_window`	21	144	2.1	7.3
	`current_buffer`	1,606	3,668	3.6	9.4
	`struct buffer`	1,889	3,774	3.2	7.4
CHCS	`DIC`	1300	1,140	96.3	179.9
SunOS	`struct mbuf`	1,570	559	3.8	4.8

Table 4. Amount of stacking occurring on star diagram nodes.

star diagram to a few levels to allow focusing on the details of current interest. Such changes add complexity to a star diagram's implementation and use, but this is probably unavoidable.

6. EXTENSIONS TO THE STAR DIAGRAM CONCEPT

The examples of planning and performing restructurings presented in earlier sections are simpler than what might be found in actual practice. First, although the star diagram concept allows us to identify large blocks of similar code that represent a single abstraction, abstractions in real programs can consist of multiple nearby statements in the same scope. Second, data structures are not necessarily instantiated by a single variable, but from multiple related data structures (such as a top pointer and counter for a stack). Finally, some design decisions, such as identifying that a proposed operation is too general, cannot be identified from the star diagram.

6.1 Selecting Parameters During Function Extraction

When extracting a group of similar expressions as a new function, a programmer must examine the source code to fully understand what the code does and what likely parameters for the new function should be. All dissimilarities in the selections must be parameterized away, since the selections must share the new function body as common code. (In fact, this is how the tool computes the default parameters during function extraction.) However, the programmer might wish to have some of the similar computations be parameters as well. When there are a large number of expressions being extracted, textually comparing all the expressions to identify the similarities and differences may be difficult because it is not possible to view all of the selections simultaneously. The textual parameter dialog used in extracting functions (see Figure 4) uses a prototype instance of the actual code for the purpose of identifying parameters.

Rather than using the node stacks to view the source code for parameter selection, an alternative is to extend the star diagram concept. The current star diagram compares variable uses by examining the enclosing expressions. We could reverse this process to choose parameters, having the arms of the star diagram comparing arguments to the function call. Once the tool user selects a node in the basic star diagram, a separate "backward" star diagram for the selected path can be used by the tool user to extend the code selection and assist in selecting function parameters (see Figure 12). A backward star diagram shows all paths corresponding to subexpressions and incoming data flows for the selected code, not just the subexpression associated with the variable being encapsulated. As in the forward star diagram, an edge either denotes nesting of expressions within an argument to the function call, or data flow through a variable assignment to a use in an argument. A backward star diagram for a selected path also overlaps similar nodes. Consequently, the tool user can quickly discern where the similarities of the code selections end. A node with a vertical bar indicates that the given parameter varies between occurrences, with the variants listed next to the bar.

By selecting different nodes in the backwards star diagram view, the tool user can specify which computations will remain with the new function body and which will become arguments to the new call by selecting the "dividing points" between computations inside and outside the function. The tool at first chooses the most specific or smallest expressions as parameters that still allow all the similar expressions representing the ADT operation to be converted to calls. For example, in Figure 12, the default choice would be to parameterize the second parameter to the `list-ref` operation, so the rightmost bar in the backwards star diagram would be highlighted to indicate that the bar represented the dividing line between inside and outside. If `*line-storage*` were to be passed in to the function, the tool user could select that node as well to indicate the variable reference should be "outside" the function.

Similarly, if the tool user wished to create a new function with the body `(length (list-ref (car x)))` (where `x` is a parameter), the tool user could select the leftmost bar and place all calculations to the left of the selection (and thus inside the outermost `car`) outside the function. In addition, making such a selection would also remove the selection from the rightmost bar; along a given path, there can be only one dividing line. In this case, only the two expressions with the `(car x)` calculation would be converted into calls to the new function. The other expressions in the regular star diagram's node stack could not be converted into calls to the new function because the new function body contains code not found in those expressions.

6.2 Data Structures Using Multiple Variables

Related expressions may not be included in the star diagram because the data structure being abstracted is actually represented by multiple variables. For example, a linked list might be represented by a head pointer and a counter for the length of the list. The preferred encapsulation would include functions manipulating both of the variables. To support such an encapsulation, the tool could allow creating overlapped star diagrams for both variables. When the two uses of the two variables are sufficiently intertwined in their usage, their paths will overlap significantly, and a single selection by the tool user would select the appropriate computation. However, in the linked-list example, the update of the counter is syntactically independent of the head pointer—the two paths might merge only at the function definition node that contains them both. This hint could lead the tool user to make two simultaneous path selections—the one for the insert and the one for the counter increment—and inspect the textual relationships of the resulting selections in the text view before choosing to extract them together. This approach could also facilitate combining two different references to the same variable into a single extracted function.

Since the layout of a star diagram containing two or more roots typically would not be planar, modifications to this approach might be necessary. One approach, which is used by our tool for planning restructurings of C programs, is to put all the related variables in the same root [Griswold et al. 1996]. This provides planarity and still displays all code of interest in a single star diagram. One potential downside to this approach is that it is not possible to determine a priori which paths are related to which variables—a node has to be selected to display the associated program text. At the other extreme, the tool could support the creation of a physically separate star diagram for each variable. In this case the relationships between paths in different star diagrams would be more difficult to see. To counter this problem, when the programmer selects a particular node, the star diagram tool could highlight related paths or nodes in other star diagrams, guiding the programmer to explore the relationship in more detail if desired.

6.3 Identifying Related Statements

Although the backward star diagram increases the encapsulation choices provided by the basic star diagram, some choices are still prohibited or disguised.

For instance, there might be output statements that pertain to the data structure, but do not actually access it. In KWIC, the following code prints out the final index:

```
(let ((numcslines (length *alph-index*)))
  (write "Alphabetic listing:")   ; excluded
  (do ((i 0 (1+ i)))
      ((= i numcslines) nil)
    (write (csline (vector-ref *alph-index* i)
                   *line-storage*)))))
```

No forward or backward star diagram would display the `write` before the `do` loop, because it is not nested in any operation or linked through assignments to the surrounding code. However, if the tool user could get access to this code through the star diagram, it might reasonably be chosen as part of a new `print-index` function. Comments are likewise a problem. In the case of an input–output operation like `write`, there is in fact an implicit global variable—the file system—being read and modified, thus "connecting" the two `write`s. If the data flows to and from this variable were included in the backward star diagram, the first `write` would be included as an implicit input to the second `write`. However, this solution only handles relationships corresponding to data flow.

We can consider two methods for including computations not related by data flow or nesting in the star diagram:

—Simple scope constructs such as compound statements could be included in the diagram, so that the tool user can select all the code in them. This method assumes that the code inside a scope is cohesive. However, including simple scope nodes results in cluttered diagrams with some programming styles, and scoping is an imperfect indicator of relatedness.

—The tool user could customize a selection through the text view, growing it or shrinking it as deemed necessary. To provide some of the benefits of the star diagram in the text, an enhanced text view that highlights forward and backward data flow from a selected piece of code could be provided. Such an extension could simplify the selection of data flow extensions of the selected code without sacrificing the ability to include other nearby statements. As noted earlier, however, if there are multiple selections, it is sometimes not practical to view and manipulate them all together as text.

As with selecting parameters, we see that finding related statements requires the tool user to understand details of the source code not shown in the star diagram. Choosing the parameters for a new function may require the tool user to compare differences between a large number of ostensibly similar expressions. In other cases, planning a new function may require the tool user to read comments or identify statements related in a manner other than the nesting and dataflow presented in the star diagram. Although the star diagram reduces the amount of code the tool user must examine in order to identify similarities between many expressions, the essential problem of scale still remains in a mitigated form.

7. RELATED WORK

Previous work that inspired the star diagram or complements the star diagram concept falls into three major categories: determining the "relatedness" of expressions in the code, assistance for planning a restructuring action, and tool-assisted encapsulation.

7.1 Coupling and Cohesion

Coupling and cohesion are subjective criteria that can be applied to a system when evaluating its structure [Stevens et al. 1974]. The cohesion of a program component is a measure of how related its subcomponents are. The coupling between two components is a measure of how related they are. (It is not always clear whether a relationship is coupling or cohesion, because it is sometimes subjective whether two components are subcomponents of a module or separate components.) In general, high cohesion within a component is desirable, because when one of its subcomponents must change, the other subcomponents are likely to require changes for the same reason. Strong coupling between components, on the other hand, is undesirable. Consequently, automatic computation of cohesion and coupling measures has been investigated as a way of assessing modularity, perhaps for driving manual restructuring. Most approaches are based on clustering or slicing.

Schwanke assumes that procedures referencing the same name must share design information on the named item, and are thus "design coupled" [Schwanke 1991]. He uses this concept as a clustering metric to identify procedures that should be placed in the same module. This approach places no special emphasis on types or names of data structures.

In contrast, Embley and Woodfield's clustering method focuses on the types and operations exported from a module and the types used within a module [Embley and Woodfield 1988]. The method identifies modules that act on more than one abstract data type or act on only subparts of a type, thus suggesting poor cohesion or inappropriate coupling.

The approach of Ott and Thuss [Ott and Thuss 1989] computes the cohesion of statements within a procedure via slicing [Weiser 1984]. In this approach, computations that help produce a particular value—as recognized by following the data and control dependences back from the assignment that produces the value—are assumed to be cohesive. Conversely, computations that are not working together to produce a value are non-cohesive. In this sense, slicing seems to capture communicational and functional cohesion. For example, when the slices of two output (reference) parameters in a procedure do not intersect, then the computations that produced the parameter results are unrelated with respect to slicing, and so are less likely to be communicationally or functionally cohesive tasks.

The star diagram displays relationships that suggest several of the commonly recognized kinds of coupling and cohesion:

—The fact that all the nodes in the diagram pertain to a single data structure indicates design coupling or functional cohesion, depending on the context. The repetition captured by node stacks also suggests design coupling.

—Node links in the star diagram capture data flow and control dependence relationships within a function definition that directly pertain to the data structure of interest. As with slicing, these suggest communicational or functional cohesion. However, for a star diagram node in which the data structure is being updated, backward data dependences are represented, as captured by backward slicing. When the data structure is merely referenced, forward data dependences are represented, as captured by forward slicing. The intuitive rationale for this distinction is that when a variable is being updated, the computations that may require encapsulation are the ones producing the update value; whereas when a variable is being referenced, the computations that may require encapsulation are the ones using the referenced value.

Intuitively, those computations shown closer to the root of the star diagram are more strongly related to the maintenance of the data structure, and hence are more likely to be functionally cohesive. Consequently, an edge identified by the tool user as representing (the weaker) communicational cohesion might imply that the more strongly cohesive expression represented by the path to the left should be extracted as a function. In such a case, the result of a call on the new function would represent the communicational relationship.

—The scoping information provided by the function definition nodes and the link to the text view suggest temporal and sequential cohesion.

7.2 Visualizations for Planning Restructuring Actions

The star diagram visualization helps a programmer understand the current implementation of a system, then redesign the system by manipulating the existing implementation. Many other visualization tools, such as Rigi, reflexion models, and Seesoft, help identify an existing design and plan evolutionary modifications, although these tools usually do not manipulate the source code for the programmer or provide explicit tools for planning modification of the source code. In each tool, a programmer can opportunistically use features of the tool to identify and record a possible structural change. Each tool differs from the star diagram according to its intended task, the level of abstraction of its visualization, and the support provided for planning and tracking possible modifications.

The Rigi reverse engineering environment provides semi-automatic construction of graphical views of a system to assist broad architectural understanding [Muller et al. 1992]. A view based on entities such as functions, files, classes, or modules is initially constructed by the environment and then augmented by the programmer. The additions allow the programmer to describe the system in terms of abstractions not visible in the implementation. Such additions can be used to redevelop documentation or to plan changes that are related to managing these invisible abstractions. By hierarchically grouping functions and files into components, the programmer records a possible remodularization which can then be used to guide changes to the source code.

Murphy and Notkin's reflexion model tool [Murphy and Notkin 1995] compares a high level description of the system provided by a software engineer with the actual structure of the source code. The software engineer describes a portion of the system in the manner of a rough block diagram. The software engineer identifies the primary abstractions, the files or functions that encode the abstraction, and the expected interactions between these abstractions. The reflexion model tool identifies where the model and actual implementation diverge. These divergences identify components that interact in a manner unexpected by the programmer. By examining each, the programmer can iteratively refine the model to match the implementation. By examining the code associated with given arcs, the programmer can also navigate to the code causing dependencies between components, and plan modifications to improve the structure of the system. Because the models generally represent relations between existing abstractions in the code, reflexion models provide support for structural tasks, such as planning restructuring or understanding a large system.

The Seesoft system [Eick et al. 1992] supports visualizing a property of each source line of a program (such as age) as a colored graphical line, thus allowing a programmer to identify patterns between lines in multiple files by comparing colors. Each file of source code in a program is represented as a long box, and each line of code is represented by a horizontal line of pixels in the box. The color of the pixels represents a value of a metric measured for that line, such as how recently the line was modified, whether a line is associated with a specific modification, or whether a given line is a member of a program slice [Weiser 1984; Ball and Eick 1994]. Seesoft's visualizations can facilitate planning modifications by highlighting historical or intrinsic structure that otherwise might be invisible. For example, a programmer can observe the frequency and scope of references to a global variable to assess the need to encapsulate it. Like the star diagram, Seesoft presents similarities between widely separated lines of code. Seesoft's line-based focus provides finer granularity than reflexion models or the models generated by Rigi. However, Seesoft's use of graphics to represent file size, position to represent source-line location, and color to highlight source-line relationships obligates it to providing a (low-resolution) representation of every line in the system.

Each of these tools provides a high level, comprehensive view of a program, making them suitable for the overall planning of maintenance activities such as the choice of data structures to encapsulate with the star diagram.

7.3 Creating Abstract Data Types

Creating abstract data types from existing code can be useful for converting an existing legacy application into a restructured and improved object-oriented program. Approaches found in the literature tend to be automated, attempting to improve the overall structure and maintainability of the system by introducing information hiding. An automated object-extraction tool must perform two tasks: find likely data structures to encapsulate, then modularize the code to encapsulate a given data structure. For both tasks, the program can be analyzed and manipulated at various levels of granularity – files, functions, or expressions.

The first task – finding likely objects – is primarily an information-gathering task. In general, restructuring algorithms build a graph describing relationships between program components, then partition the graph to identify a likely abstract data type and components acting upon it. For example, the RE^2 project groups sets of procedures that operate on a given data structure into a single ADT. If a procedure uses several data structures, a set of rules identifies the most likely owner [Canfora et al. 1993; Canfora et al. 1993]. Achee and Carter similarly group procedures into ADTs, but use a weighting scheme to identify the most likely containing ADT [Achee and Carver 1994]. Cutillo, Fiore, and Visaggio use program slices to identify related statements within a procedure that act upon the data structure and could be extracted as a new procedure [Cutillo et al. 1993].

The second task for creating an ADT involves identifying and creating the set of functions that will serve as operations on the ADT. When the operations on the ADT will be chosen from existing functions, this process is relatively simple: the tool chooses a subset of the functions using the ADT as operations on the ADT. For example, Newcomb and Markosian's COBOL restructuring tool creates new modules by separating selected blocks of COBOL code to be moved into another program, then adding appropriate syntax to perform subroutine calls to the new module [Newcomb and Markosian 1993]. The object-finding algorithms described above use similar methods.

When operations are created from expressions within existing functions, the process is more complicated. Not only must the references to the abstract data type be identified, but computations representing an abstract operation on the ADT must also be noted. This requires both identifying cohesive expressions around each use as well as identifying similar expressions applied to multiple uses. With Srinivasan's restructuring tool, the tool user first identifies existing functions that should be abstract operations on the ADT. Any unencapsulated references remaining after partitioning the functions are encapsulated automatically using the simple approach of creating reader and writer routines [Srinivasan 1993]. Newcomb's system for creating objects in existing COBOL programs identifies code representing a single operation on a data structure by slicing the enclosing basic block for each use. Each slice becomes a operation on the new ADT. Each new function is generalized by turning constants, integers, and string literals into parameters of the function. Finally, duplicate functions are combined [Newcomb 1995]. The Architectural Recovery Tool (ART) uses a database of cliches (common code structures) to identify likely abstractions [Tonella et al. 1996].

The star diagram is similar to the second class of tools listed above, helping the programmer to create operations on a given data structure once a target for encapsulation is chosen. Like Cutillo and Newcomb, the star diagram permits creating functions from expressions within an existing function, not merely entire functions. It uses dataflow information to identify candidate expressions to convert into functions. Like Newcomb and Srinivasan's approaches, the star diagram converts uses of the abstract data type at multiple places in the code into calls to a single function.

The star diagram differs from the tools listed above in three ways: strictness of the criteria for determining if two expressions correspond to the same candidate function, flexibility for choosing the function to create, and availability of information on the frequency of similar functions. First, the star diagram's matching criteria are less strict than Newcomb or ART's approach, primarily because the interactive nature of the star diagram allows the programmer to select which of a large number of candidate expressions should become a function. Although the number of choices can be huge, the star diagram presentation highlights relevant issues. Candidates are presented hierarchically from smallest to largest, grouped by similarity of access, and annotated with the frequency of each candidate use. Secondly, because the star diagram is interactive, the programmer decides exactly which code corresponds to the current ADT and which code should remain in the calling function. In this way, the programmer can ensure that the new ADT has a consistent and small set of operations that operate on it. In contrast, Srinivasan's restructuring tool only creates accessor functions, and Newcomb creates functions from the slice of the enclosing basic block with respect to the use of the data structure. Such functions may be too generic or too specific for reuse or modification of the ADT. Finally, the star diagram identifies how frequently each expression occurs. The programmer can interpret a stack of one node (i.e. a possible function body that only occurs once in the program) as corresponding to an abstraction too specific to be a likely operation on the ADT. Conversely, a candidate function occurring much more commonly than other uses of the ADT could indicate a case where the programmer should consider if the function is too general to be a useful operation on the ADT.

Although the star diagram is intended as a representation to help a programmer choose functions, it could be used as the basis for an automatic object generator. For example, an automatic generator could create functions from nodes near where the star diagram begins to significantly diverge. Nodes to the right of such a node represent more specific and less common expressions. Nodes to the left of this point probably represent overly general expressions. However, with such an algorithm, the tool may not choose an operation representing a meaningful and easily expressed abstraction. A better approach might be to use a large database of common ADT operations (such as Rich and Waters's cliches [Rich and Waters 1988]) to identify common calculations, choose an appropriate abstraction, and specify likely abstraction names and parameters [Tonella et al. 1996]. For example, such a mechanism might discover that the operations being chosen match a list or hash table model. With such knowledge, the tool can identify which operations correspond to hash table operations, ignore expressions less relevant to the abstraction, and suggest meaningful names for operations and parameters.

8. CONCLUSION

The hiding of data structures within modules can simplify the changes to those data structures. However, appropriate encapsulation choices are difficult to articulate without knowing what changes are going to be required by users. In part for these reasons, changing software is often expensive. However, encapsulating a data structure in a program after the fact to correct the problem is a difficult and error-prone process requiring the programmer to have a detailed understanding of how the data structure is manipulated throughout the program.

To address this problem, we introduced a manipulable program visualization called the *star diagram*. The star diagram both highlights information pertinent to encapsulation and supports the application of meaning-preserving restructuring transformations on the program through a direct-manipulation user interface. The visualization graphically and compactly presents all statements in the program that use the given global data structure directly or indirectly, helping the programmer to choose the functions that completely encapsulate it. The star diagram's tree structure allows the programmer to quickly identify similar expressions and note how they differ. Additionally, the visualization elides code unrelated to the data structure and to the task, and collapses similar expressions to allow the programmer to identify frequently occurring code fragments and manipulate them together. The visualization is mapped directly to the program text, so manipulation of the visualization also automatically restructures the program.

We studied how star diagrams can simplify restructuring from four different perspectives. First, we presented the concept of the star diagram, and showed its practicality by implementing a star diagram view on top of an existing Scheme restructuring tool. By implementing star diagram generators for C and MUMPS, we showed that the concept can be applied to programs in other languages. We investigated the scalability of the star diagram concept by creating and examining star diagrams for variables in large C programs. Also, in a separate study we found that programmers could actually use the star diagram to perform encapsulation [Bowdidge and Griswold 1997]. The star diagram encouraged programmers to perform encapsulations in an exploratory manner, instead of planning the entire restructuring from a quick scan of the code. Such an exploratory style can be beneficial in a restructuring tasks that primarily involve changing a system's design. All these perspectives show the star diagram's promise for supporting software maintenance.

Our experiences implementing the star diagram and other visualizations for restructuring provide lessons for future tools. First, manipulable visualizations for restructuring must display the objects we intend to manipulate. As we found with the structure charts, many common design notations only present high-level abstractions, and thus do not display the expressions that need to be manipulated when performing an encapsulation. Second, because restructuring quickly causes significant change to the source code, the tool invalidates the tool user's mental model of the source code. Thus, all views must be kept up-to-date at all times. Third, the visualization must display the information most essential to programmers during the specific restructuring task. For the star diagram, we chose the atypical approach of grouping expressions by similarity, rather than location, to support identifying common code fragments. Finally, new visualizations should also focus on a well-specified manipulation task to avoid overextending the interface.

A number of issues have been raised in this research but not definitively resolved.

Use of the star diagram without a restructuring tool. The star diagram has shown promise even in a non-restructuring environment. We have seen in examples of star diagram use (Sections 2.2 and 4.2) that the star diagram helped identify abstractions latent in the code. We have also found program anomalies while perusing star diagrams. In a star diagram for `adventure` (Section 4.2), we found an assignment in the place of an equality test in a boolean expression. In `emacs` we found cases where a static computation was performed inside a loop, instead of outside it. There are also early results suggesting that a tool user can manipulate a star diagram to plan a restructuring [Chen 1996; Griswold et al. 1996].

Extracting multi-statement functions. The star diagram currently does not support either identifying or creating functions composed of multiple statements that are only indirectly related. Allowing the programmer to select multiple nodes in the star diagram is part of the solution, but it may also be necessary for the star diagram to infer relationships between syntactically independent computations and present them as a single node in the star diagram (Section 6).

Visualizing, comprehending, and manipulating large star diagrams. Although Section 5 shows that usable star diagrams can be created for large programs, we also found that reducing star diagrams to a comprehensible size is strongly dependent on the complexity of the data structure being visualized, so universal solutions are unlikely. As a result, techniques are required for customizing the star diagram to the special requirements for the task at hand.

Support for other restructuring tasks. Because the star diagram visualization is designed to support a specific task, it can only be useful for a class of restructuring problems. Other views in the restructuring tool, such as the structure diagram view, assist tasks the star diagram does not, and vice versa [Griswold and Bowdidge 1993]. Because the effort required to create a specialized view is currently quite high, some views may be deemed not worth the effort. In some cases, then, the tool user may be required to fall back on restructuring via the text view. The question then is whether a small number of views is sufficient, or whether there is an inexpensive way to create new views.

ACKNOWLEDGMENTS

We would like to thank J. David Morgenthaler and Darren C. Atkinson for use of their parsing and analysis tools for producing the C and MUMPS star diagrams. We are also grateful to the reviewers for their detailed comments and suggestions for improving this paper.

APPENDIX

A. KWIC INDEX PROGRAM

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;
;;; INPUT MODULE
```

```
;;;
;;; necessary if we're not processing the revised scheme
;;;(define 1+ (lambda (x) (+ 1 x)))
;;;(define 1- (lambda (x) (- 1 x)))
 ;;;
;;; list of list of words.
;;;
(define *line-storage* nil)

(define putfile (lambda (linelist)
  (letrec
    ;;
    ;; Adds a line.  By convention, lineno=0 implies the first line.
    ;;  We assume the input is a list of symbols.  We convert to strings
    ;;  for comparisons and such.
    ;;
    ((insline (lambda (line)
       (if line (set! *line-storage* (cons line *line-storage*))))))

  (do ((restlist linelist (cdr restlist)))
      ((null? restlist) nil)
    (insline (car restlist))))))


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;
;;; CIRCULAR SHIFTER
;;;
;;;  This module creates the illusion (or reality) that the
;;;  lineholder has had all the circular shifts of lines inserted for
;;;  all lines.  For line i < j, all of i's shifts come before the j's.
;;;  The shifts are inserted in order starting from the original line.
;;;
;;;  Amazing fact:  for a line with N words, there are N circular
;;;  shifts.  This means that shift M is the line containing the Mth
;;;  word in the file, with the first word of the shift being the Mth
;;;  word.
;;;


(define *circ-index* nil)

;;;
;;; Build an index of circulars.  These are represented as pairs of
;;;  (lineno, wordno).
;;;
(define cssetup
    (lambda ()
      (letrec
       ((allwords (lambda (ls)
                    (do ((restls ls (cdr restls))
                         (sum 0 sum))
                        ((null? restls) sum)
                      (set! sum (+ sum (length (car restls))))))))
       (let ((numcslines (allwords *line-storage*))
             (cslineno 0)
```

```
            (numlines (length *line-storage*))
            (numwords nil))
        (set! *circ-index* (make-vector numcslines))
        (do ((lineno 0 (1+ lineno)))
            ((= lineno numlines) nil)
          (set! numwords (length (list-ref *line-storage* lineno)))
          (do ((wordno 0 (1+ wordno)))
              ((= wordno numwords) nil)
            (vector-set! *circ-index* cslineno (list lineno wordno))
            (set! cslineno (1+ cslineno))))))))


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;
;;; ALPHABETIZING MODULE
;;;
;;;  This contains function alph.  It creates an array just like CS's
;;;  but is sorted.
;;;

(define *alph-index* nil)

(define alph (lambda ()
  (letrec
    ;; Says if shiftno1 is less than or equal to shiftno2
   ((csline<= (lambda (shift1 shift2 ls)
     (letrec
       ;; Return the word on line number shiftno, at word number
       ;; wordno in the line.  Result is a string.
      ((csword (lambda (shift wordno ls)
         (let* ((lno  (car shift))
                (fwno  (cadr shift))) ; number of the first word in the shift
           (list-ref (list-ref ls lno)
                     (modulo (+ fwno wordno) (length (list-ref ls lno)))))))

       ;; Returns the number of words in line number shiftno
       (cswords (lambda (shift ls)
         (length (list-ref ls (car shift))))))

      (let ((lasti (min (cswords shift1 ls)
                        (cswords shift2 ls)))
            (result nil)
            (done? nil))
        (do ((i 0 (1+ i)))
            (done? result)
          (let ((maxed? (= i lasti))
                (cword1 (symbol->string (csword shift1 i ls)))
                (cword2 (symbol->string (csword shift2 i ls))))
              (if (or maxed? (not (string=? cword1 cword2)))
                (begin
                 (set! done? t)
                 (set! result
                       (if maxed?
                           (<= lasti (cswords shift2 ls))
                           (string<=? cword1 cword2))))))))))
    (swap-indices (lambda (vec i j)
```

```
                    (let ((temp (vector-ref vec i)))
                      (vector-set! vec i (vector-ref vec j))
                      (vector-set! vec j temp))))
   ;; Look at each cs-line from start to end and put its index in
   ;; the upper or lower half of *alph-index*.  An equal comparison
   ;; defaults to the left side.
   (qsplit (lambda (start end split)
     ;; start one below bot,and use bot as <= split
     (let ((low (1+ start))
           (high end))

       ;; swap the split and start so split doesn't get mixed in swaps.
       (swap-indices *alph-index* start split)
       (set! split start)

       ;; do split
       (do ()
           ((> low high) nil)
         (if (csline<= (vector-ref *alph-index* low)
                       (vector-ref *alph-index* split)
                       *line-storage*)
             (set! low (1+ low))
             (begin
              (swap-indices *alph-index* low high)
              (set! high (1- high)))))

       ;; On exit of loop, we are guaranteed that (1- low) is in the low end.
       ;;  In the worst case, it is start (i.e., split).  So we swap this
       ;;  with the split, and then qalph will sort everything above and
       ;;  below (1- low).
       (swap-indices *alph-index* split (1- low))

       (1- low))))

   ;; Quicksort the shifted lines from start to end.
   (qalph (lambda (start end)
     (if (< start end)
         (let* ((split start)
                (middle (qsplit start end split)))
           (begin
            (qalph start (1- middle))
            (qalph (1+ middle) end)))))))

  ;; THE REAL CODE
  (let ((numitems (length *circ-index*)))
   (set! *alph-index* (make-vector numitems))
   (do ((i 0 (1+ i)))
       ((= i numitems) nil)
     (vector-set! *alph-index* i (vector-ref *circ-index* i)))
   (qalph 0 (1- numitems))))))


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;
;;; OUTPUT MODULE
;;;
```

```
(define allalphcslines (lambda ()
 (letrec
   ;; builds a circularly shifted line storage and a shiftspec
  ((csline (lambda (shift ls)
     (let* ((lno    (car shift))
          (fwno   (cadr shift))
          (wrdcnt (length (list-ref ls lno)))
          (revcs nil))
     (do ((i 0 (1+ i)))
         ((= i wrdcnt))
       (set! revcs
             (cons (list-ref (list-ref ls lno) (modulo (+ i fwno) wrdcnt))
                   revcs)))
     (reverse revcs)))))

  (let ((numcslines (length *alph-index*)))
    (write "Alphabetic listing:")
    (do ((i 0 (1+ i)))
        ((= i numcslines) nil)
      (write (csline (vector-ref *alph-index* i) *line-storage*)))))))


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;
;;; MASTER CONTROL
;;;

(putfile (list '(a b c d) '(one) '(hey this is different) '(a b c d)))
(cssetup)
(alph)
(allalphcslines)
```

REFERENCES

ACHEE, B. L. AND CARVER, D. L. 1994. A greedy approach to object identification in imperative code. In *Workshop on Program Comprehension* (1994), pp. 4–11.

ADAMS, S. 1980. Pirate's adventure. *BYTE 5*, 12 (December), 192–212.

AHO, A. V. 1980. Pattern matching in strings. In R. V. BOOK Ed., *Formal Language Theory: Perspectives and Open Problems*, pp. 325–347. New York: Academic Press.

AHO, A. V., SETHI, R., AND ULLMAN, J. D. 1986. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, Reading, MA.

BALL, T. AND EICK, S. G. 1994. Visualizing program slices. In *IEEE Symposium on Visual Languages* (St. Louis, Missouri, 1994), pp. 288–295. IEEE Computer Society.

BERGLAND, G. D. 1981. A guided tour of program design methodologies. *IEEE Computer 14*, 10, 18–37.

BOEHM, B. W. 1975. The high cost of software. In E. HOROWITZ Ed., *Practical Strategies for Developing Large Software Systems*, pp. 3–15. Reading, MA: Addison-Wesley.

BOWDIDGE, R. W. AND GRISWOLD, W. G. 1997. How software tools organize programer behavior during the task of data encapsulation. *Empirical Software Engineering 2*, 3, 221–268.

BUSH, E. 1985. The automatic restructuring of COBOL. In *Conference on Software Maintenance* (Washington D.C., 1985), pp. 35–41. IEEE Computer Society Press.

Canfora, G., Cimitile, A., and Munro, M. 1993. A reverse engineering method for identifying reusable abstract data types. In *1st Working Conference on Reverse Engineering* (Baltimore, MD, 1993), pp. 73–82. IEEE Press.

Canfora, G., Cimitile, A., Munro, M., and Tortorella, M. 1993. Experiments in identifying reusable abstract data types in program code. In *IEEE Conference on Program Comprehension* (Capri, Italy, 1993), pp. 36–45. IEEE Computer Society Press.

Chen, M. 1996. A tool for planning restructuring of data abstractions in large systems. Master's thesis, Computer Science Department, University of California, San Diego.

Cutillo, F., Fiore, P., and Visaggio, G. 1993. Identification and extraction of "domain independent" components in large programs. In *1st Working Conference on Reverse Engineering* (Baltimore, MD, 1993). IEEE Press.

de Balbine, G. 1975. Better manpower utilization using automatic restructuring. In *National Computer Conference* (1975).

Dybvig, R. K. 1987. *The Scheme Programming Language*. Prentice-Hall, Englewood Cliffs, NJ.

Eick, S. C., Steffen, J. L., and Sumner, J., E. E. 1992. Seesoft – a tool for visualizing line-oriented software statistics. *IEEE Transactions on Software Engineering 18*, 11, 957–968.

Embley, D. W. and Woodfield, S. N. 1988. Assessing the quality of abstract data types written in Ada. In *10th International Conference on Software Engineering* (1988), pp. 144–153.

Griswold, W. G. and Atkinson, D. C. 1995. Managing design trade-offs for a program understanding and transformation tool. *Journal of Systems and Software 30*, 1–2, 99–116.

Griswold, W. G., Atkinson, D. C., and McCurdy, C. 1996. Fast, flexible syntactic pattern matching and processing. In *WPC '96: 4th Workshop on Program Comprehension* (Berlin, Germany, 1996).

Griswold, W. G. and Bowdidge, R. W. 1993. Program restructuring via design-level manipulation. In *Workshop on Studies of Software Design*, Volume 1078 of *Springer-Verlag Lecture Notes in Computer Science* (Baltimore MD, 1993).

Griswold, W. G., Chen, M. I., Bowdidge, R. W., and Morgenthaler, J. D. 1996. Tool support for planning the restructuring of data abstractions in large systems. In *ACM SIGSOFT '96 Symposium on the Foundations of Software Engineering* (San Francisco, CA, 1996). ACM Press.

Griswold, W. G. and Notkin, D. 1992. Computer-aided vs. manual program restructuring. *ACM SIGSOFT Software Engineering Notes 17*, 1 (January).

Griswold, W. G. and Notkin, D. 1993. Automated assistance for program restructuring. *ACM Transactions on Software Engineering and Methodology 2*, 3, 228–269.

Hutchins, E. L., Hollan, J. D., and Norman, D. A. 1985. Direct manipulation interfaces. *Human-Computer Interaction 1*, 311–338.

Johnson, R. E. and Opdyke, W. F. 1993. Refactoring and aggregation. In *Object Technologies for Advanced Software: First JSSST International Symposium*, Lecture Notes in Computer Science, pp. 264–278. Springer Verlag.

Lehman, M. M. and Belady, L. A. 1985. *Program Evolution: processes of software change.* APIC studies in data processing. Academic Press, London.

Lientz, B. and Swanson, E. 1980. *Software Maintenance Management: A Study of the Maintenance of Computer Application Software in 487 Data Processing Organizations.* Addison-Wesley, Reading MA.

Liskov, B., Snyder, A., Atkinson, R., and Schaffert, C. 1977. Abstraction mechanisms in CLU. *Communications of the ACM 20*, 8, 564–576.

Moen, S. 1990. Drawing dynamic trees. *IEEE Software*, 21–28.

Morgenthaler, J. D. and Griswold, W. G. 1995. Program analysis for practical program restructuring. In W. G. Griswold Ed., *Workshop on Program Transformations for Software Evolution* (Seattle, WA, 1995).

Muller, H. A., Tilley, S. R., Orgun, M. A., and Corrie, B. D. 1992. A reverse engineering environment based on spacial and visual software interconnection models. In *SIGSOFT '92: Fifth Symposium on Software Development Environments* (Washington D.C., 1992), pp. 88–98. ACM Press.

Murphy, G. and Notkin, D. 1995. Software reflexion models: Bridging the gap between source and high-level models. In *Symposium on the Foundations of Software Engineering* (Washington D.C., 1995), pp. 18–28. ACM Press.

Newcomb, P. 1995. Reengineering procedural into object-oriented systems. In *2nd Working Conference on Reverse Engineering* (Toronto, ON, 1995), pp. 237–249. IEEE Press.

Newcomb, P. and Markosian, L. 1993. Automating the modularization of large COBOL programs: Application of an enabling technology for reengineering. In *1st Working Conference on Reverse Engineering* (Baltimore MD, 1993), pp. 222–230. IEEE Press.

Opdyke, W. F. 1991. *Refactoring: A Program Restructuring Aid in Designing Object-Oriented Application Frameworks.* Ph. D. thesis, University of Illinois, Urbana-Champaign.

Ott, L. M. and Thuss, J. J. 1989. The relationship between slices and module cohesion. In *11th International Conference on Software Engineering* (Pittsburgh, Pennsylvania, 1989), pp. 198–204.

Parnas, D. L. 1972. On the criteria to be used in decomposing systems into modules. *Communications of the ACM 15*, 12, 1053–1058.

Partsch, H. and Steinbruggen, R. 1983. Program transformation systems. *ACM Computing Surveys 15*, 3, 199–236.

Rich, C. and Waters, R. C. 1988. The Programmer's Apprentice: A research overview. *IEEE Computer 21*, 11, 10–25.

Robins, G. 1985. Applications of the ISI Grapher. In *Fourth Annual Artificial Intelligence and Advanced Computer Technology Conference* (1985), pp. 105–129.

Rowe, L. A., Konstan, J. A., Smith, B. C., Seitz, S., and Liu, C. 1991. The PICASSO application framework. In *14th ACM Symposium on User Interface Software and Technology* (Hilton Head, SC, 1991), pp. 95–106. ACM Press.

Schneiderman, B. 1983. Direct manipulation: A step beyond programming languages. *IEEE Computer 16*, 8 (August), 57–69.

Schwanke, R. W. 1991. An intelligent tool for re-engineering software modularity. In *13th International Conference on Software Engineering* (Austin, TX, 1991), pp. 83–92. IEEE Computer Society Press.

Srinivasan, R. 1993. Automatic software design recovery and re-modularization using source transformation. Master's thesis, Queen's University, Kingston, Ontario, Canada.

Stallman, R. 1993. GNU Emacs (the extensible self documenting text editor).

Steensgaard, B. 1995. Points-to analysis in almost linear time. In *23rd ACM Symposium on Principles of Programming Languages* (St. Petersburg Beach, FL, 1995).

Stevens, W. P., Myers, G. J., and Constantine, L. L. 1974. Structured design. *IBM Systems Journal 13*, 2, 115–139.

Tonella, P., Fiutem, R., Antoniol, G., and Merlo, E. 1996. Augmenting pattern-based architectural recovery with flow analysis: Mosaic – a case study. In *Proceedings of Working Conference on Reverse Engineering (WCRE '96)* (Monterey, CA, 1996), pp. 198–207. IEEE Press.

Weiser, M. 1984. Program slicing. *IEEE Transactions on Software Engineering SE-10*, 4, 35–357.

```
(length (list-ref *line-storage* lno))
(length (list-ref *line-storage (car shift)))
(length (list-ref *line-storage* lno))
(length (list-ref *line-storage*
                  (car (car line-number-pair))))
```

Fig. 12. A possible backwards star diagram for the path `*line-storage*` → `list-ref` → `length`. The four statements at the bottom are the lines of code represented in the diagram.