Automated Assistance for Program Restructuring

WILLIAM G. GRISWOLD
University of California—San Diego
and
DAVID NOTKIN
University of Washington

Maintenance tends to degrade the structure of software, ultimately making maintenance more costly. At times, then, it is worthwhile to manipulate the structure of a system to make changes easier. However, manual restructuring is an error-prone and expensive activity. By separating structural manipulations from other maintenance activities, the semantics of a system can be held constant by a tool, assuring that no errors are introduced by restructuring. To allow the maintenance team to focus on the aspects of restructuring and maintenance requiring human judgment, a transformation-based tool can be provided—based on a model that exploits preserving data flow dependence and control flow dependence—to automate the repetitive, error-prone, and computationally demanding aspects of restructuring. A set of automatable transformations is introduced; their impact on structure is described, and their usefulness is demonstrated in examples. A model to aid building meaning-preserving restructuring transformations is described, and its realization in a functioning prototype tool for restructuring Scheme programs is discussed.

Categories and Subject Descriptors: D.2.2 [**Software Engineering**]: Tools and Techniques—*computer-aided software engineering;* D.2.7 [**Software Engineering**]: Distribution and Maintenance—*corrections*; *enhancement*; *extensibility*; *restructuring*; D.2.10 [**Software Engineering**]: Design; D.3.m [**Programming Languages**]: Miscellaneous; K.6.3 [**Management of Computing and Information Systems**]: Software Management—*software maintenance*

General Terms: Management

Additional Key Words and Phrases: CASE, flow analysis, meaning-preserving transformations, software engineering, software evolution, software maintenance, software restructuring, source-level restructuring

This work was supported in part by an IBM Graduate Fellowship, NSF grant CCR-8858804, NSF grant CCR-9113367, NSF grant CCR-9211002, a DEC External Research Program Grant, and The Xerox Corporation.
Authors' addresses: W. G. Griswold, Department of Computer Science and Engineering, 0114, University of California, San Diego, CA 92093-0114; email: wgg@cs.ucsd.edu; D. Notkin, Department of Computer Science and Engineering, FR-35, University of Washington, Seattle, WA 98195; email: notkin@cs.washington.edu.
Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.
© 1993 ACM 1049 331X/93/0700-228 $01.50

1. INTRODUCTION

Computer software has been touted as a panacea for the engineer because it is so malleable compared to physical construction media such as concrete, steel, and silicon. However, its apparent flexibility has not been successfully exploited—software maintenance (enhancement and repair) remains disproportionately expensive relative to the expected cost of the required changes and the quality of the resulting software [35].

In studies of OS/360 and other large systems, Belady and Lehman [7] observed that the cost of a change grew exponentially with respect to a system's age. They associated these rising costs with decaying structure caused by the accumulation of unanticipated changes:

> The addition of any function not visualized in the original design will inevitably degenerate structure. Repairs, also, will tend to cause deviation from structural regularity since, except under conditions of the strictest control, any repair or patch will be made in the simplest and quickest way. No search will be made for a fix that maintains structural integrity [8, p. 113].

They conclude that this cannot go on indefinitely without having to rebuild the system from scratch, and the need to minimize software cost

> suggests that large-program structure must not only be created but must also be maintained if decay is to be avoided or postponed. Planning and control of the maintenance and change process should seek to ensure the most cost-effective balance between functional and structural maintenance over the lifetime of the program. Models, methods and tools are required to facilitate achieving such balance [33, p. 383].

The structural principle of information hiding is used to isolate within a module each design decision that is likely to change. Effective isolation reduces costs by localizing—to the implementation of a module—the parts of a system that must be modified when one of these design decisions later changes [41].

Unfortunately, the initial design of a system cannot isolate all design decisions. First, Parnas emphasizes that modules are used to isolate design decisions that the designer *anticipates* are likely to change [40]. But many changes, especially user-driven enhancements, cannot be anticipated by the original designers. Second, given the range of possible future changes, it is not economically feasible to accommodate every one, even if they are properly anticipated [9, pp. 20–21, 40]. Third, the module concept allows only one decomposition of a program. Given a set of design decisions to hide, there may be conflicting feasible modularizations of a program. One must be chosen in favor of the others.

Because you cannot isolate every design decision, inevitably a change will be requested that cannot be implemented by modifying a single module. Thus some system changes must cross module boundaries, requiring consistent modification of module interfaces and implementations [40]. This inherent need for consistent nonlocal change is the key characteristic of structural degradation.

One way structural degradation can be reduced is to restructure the program to isolate a different, more appropriate set of design decisions. In essence, restructuring is redesign in the presence of more accurate information about pending changes.

Maintaining structure, however, is itself a complex and costly activity. Indeed, it is just like making a functional change in a structurally inadequate system: global search and change are required to maintain the behavioral relationships between a modified interface and the references to it throughout the program. This difficulty leads to the well-known phenomenon that software maintenance—be it for enhancement, correction, or restructuring—generally injects additional defects into a system as an undesirable side-effect of the basic maintenance activity. Belady and Lehman [8] characterize this in terms of a stratification of changes that leads to exponentially complex system structure. Collofello and Buck, as another example, have measured that when adding a product feature, 53% of the new defects were to existing features [14].

Belady and Lehman [8] conclude that *progressive* activities such as enhancement require continual *antiregressive* efforts to keep the inevitable exponential growth in complexity manageable. However, antiregressive activities in practice get ignored under financial and time pressures, and also because they are not usually as psychologically satisfying as progressive activities. Thus there is a general preference for quick fixes over those that retain or improve structural integrity.

1.1 An Alternative Approach

We use the power of the computer to overcome some of the inherent costs in making *manual* nonlocal changes. In particular, we believe that restructuring can reduce the cost of software maintenance if it is supported by a tool that performs the nonlocal aspects of restructuring in a way that assures that no errors are introduced.

In our approach the software engineer applies a local structural change to a syntactic form in the program, with the tool performing the (usually nonlocal) compensating changes necessary to maintain consistency throughout the program. Our tool ensures that an automated change is consistent by assuring that the meaning of the program has not changed.

This approach has two benefits. First, it uses automation to free the engineer from the highest costs of structural changes: nonlocal search and change, and later, debugging to repair inconsistencies in the distributed changes. Second, it leaves the engineer in control of the subjective activities of choosing the appropriate structure for the *re*design. This is critical since the selection of appropriate designs (or, in restructuring, redesigns) cannot be done automatically.

The abbreviated program in Figure 1 shows a simple example of such a transformation. When the engineer swaps push's formal parameters to conform to interface rules, the tool is responsible for swapping the arguments in its calls. If evaluation order of the arguments can affect the values of the arguments—as in the call h(myStack) in the second call on push—the tool is

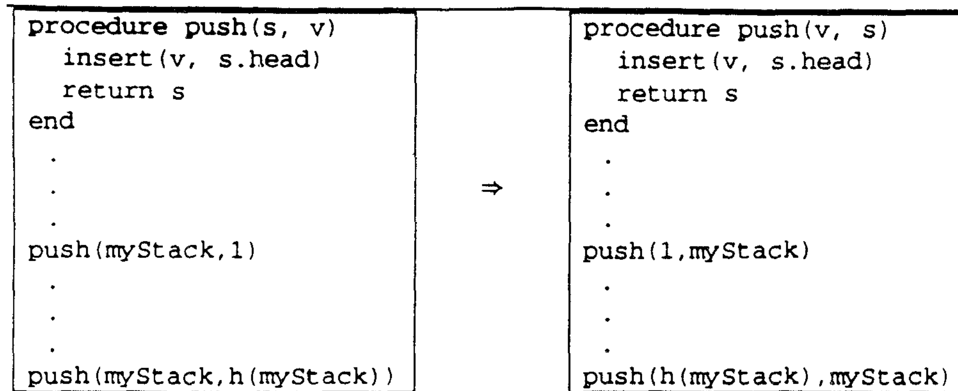

Fig. 1. Swapping the parameters of procedure push.

responsible for prohibiting the change and alerting the engineer to the problem. For a restructuring tool to be precise enough to be useful, then, it must be able to examine the definitions of functions (such as h) to determine properties such as side-effects to arguments.

A tool can perform the nonlocal aspects of other structural changes such as: replacing an expression with a variable that has its value; swapping the formal parameters in a procedure's interface and the respective arguments in its calls; adding a parameter to a procedure definition and the appropriate argument to its calls; replacing inline code with a call to a function that contains that code. These can be extended to module-level and class-level manipulations.

1.2 Restructuring and the Software Process

In conventional process models, the maintenance phase usually includes both restructuring and general maintenance activities. In these process models, restructuring is not explicitly identified. Thus one iteration of the maintenance phase is

$$I_M ; V_M ,$$

where I_M is the activity of implementing the enhancements, corrections, etc. that were the initial intention of the maintenance phase, and V_M is the validation of I_M.

Some models explicitly identify redesign:

$$D_R ; I_{RM} ; V_{RM} ,$$

where D_R is the activity that determines appropriate new structures intended to ease subsequent general maintenance; I_{RM} represents the combination of the implementation activities for restructuring and for maintenance, and V_{RM} represents the validation of I_{RM}.

Completely separating the restructuring and general maintenance activities leads to process models where the structure of the maintenance phase is

$$D_R ; I_R ; V_R ; I_M ; V_M .$$

That is, the activities of defining, implementing, and validating the restructuring of the system take place prior to the activities of implementing and validating general maintenance. Separating the restructuring activity from general maintenance activities (such as enhancement, correction, and retargeting) gives the software engineer intellectual leverage. During the restructuring activity, the engineer determines a new design that is more suitable for the upcoming modifications I_M.

Our approach augments the power of this intellectual separation with a restructuring tool, defining the new process model

$$DIV_R ; I_M ; V_M ,$$

where DIV_R combines the design, implementation, and validation activities of restructuring into a single activity. The implementation activity, I_R, is eliminated because the tool keeps the essence of the implementation constant in the face of the redesign, D_R. That is, the basic algorithms and representations of the system stay constant, although their location within modules may change. (Later, of course, I_M might change these algorithms and representations as a part of general maintenance.) The validation activity, V_R, is eliminated because the tool ensures that the meaning of the restructured program is unchanged from the original program; further validation is not needed.[1]

An added benefit of this process model is that the I_M and V_M activities are more focused than the I_{RM} and V_{RM} activities described above. Thus, I_M and V_M should be simpler because they no longer include restructuring activities. Also, I_M and V_M should be simpler because of the better structure created by DIV_R in anticipation of I_M.

These models do not suggest when restructuring should be performed. Nor is there much experience in making such decisions. The availability of a restructuring tool may help us to identify, over time, suitable policies for reducing maintenance costs through restructuring.

1.3 Overview

We demonstrate that automated assistance for restructuring is feasible for imperative programming languages. Section 2 demonstrates how automating the meaning-preserving activities of restructuring through transformation improves the manual process of restructuring. We define, in Section 3, a set of transformations that can restructure programs, also discussing an experiment that compares the manual restructuring of a matrix multiply program to the same restructuring using our prototype tool for restructuring

[1] Of course, this also implies that any defects in the original will still be present in the restructured program

Scheme programs. We also briefly describe the restructuring of Parnas' KWIC program, which demonstrates that the prototype's transformations are powerful enough to restructure a program from a functional decomposition to a data decomposition. In Section 4 we describe our model for meaning-preserving source-to-source transformation and its use to help correctly and efficiently implement our prototype. Section 5 examines related research. To conclude, in Section 6 we evaluate the current limits of our approach and the contributions made.

2. A RESTRUCTURING EXAMPLE

To give a feeling for the sorts of structural problems encountered during maintenance and how a tool can help, consider a simplified model of a computer system for community transit that actively tracks the distance traveled by each bus on the road. The bus module exports this value as variable miles_traveled. Every three minutes the bus sends its trip odometer reading to the central computer, which assigns it to miles_traveled. This information is used by the tracking module to display the location of the bus on a map for the dispatcher. The total accumulated miles are also used for scheduling preventive maintenance of the buses. The module structure of the initial system is shown in Figure 2.

When the transit authority decides to put buses onto ferries to service nearby islands, the job of the software engineer is to add ferries to the system without changing its nonferry behavior. In the updated system, miles_traveled can no longer be used for both scheduling maintenance and locating a bus, since not all miles traveled will be rolling miles. New abstractions are needed that distinguish rolled miles from those due to ferry trips. Further, it is necessary to determine which expressions currently using miles_traveled should be just rolled miles, and which should be the combined value. This determination cannot be made automatically, but it can be guided by our tool.

With a restructuring tool the engineer can locally specify structural changes without changing the meaning of the program. Further, by preceding the enhancement with restructuring, the changes needed to implement the enhancement will be local, and thus easier to perform.

First, the bus module variable miles_traveled needs to be renamed to rolled_miles to reflect more precisely its true meaning. With the tool, the engineer invokes the transformation **rename-variable** on the declaration of miles_traveled to change all references of the variable to rolled_miles. This is not a purely textual substitution. For example, to preserve meaning, the tool must verify that the new name does not conflict with any names in the modified scope.

Now a new abstraction, total_miles, must be created for eventually combining the values of rolled_miles and ferry trip miles. To begin, the engineer needs to introduce a function total_miles that returns the value of rolled_miles. This is done by invoking the transformation **extract-function** on a use of rolled_miles. Then the engineer invokes the transformation **scope-substitute-call** on the definition of total_miles, which finds each expression

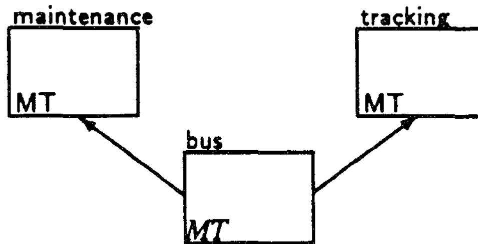

Fig. 2. Module export-import structure of the initial transit system (**MT** = miles_traveled, variable declaration in italics).

equivalent to the function—each reference to rolled_miles—and asks the engineer for approval to substitute a function call. The engineer approves the matches in the bus-tracking module and denies the ones in the bus maintenance module. Meaning is preserved by these substitutions because each call exactly represents the reference to rolled_miles that was replaced.[2]

Next, the engineer creates the ferry_miles variable using **create-variable**. Again, this is not just a syntactic change; the operation verifies that it does not mask or conflict with any existing variable declarations. This new variable is not referenced, so it does not affect the system's meaning. Up to this point, the program performs exactly the same function as before.

Finally, the engineer augments total_miles by adding code that sums rolled_miles with ferry_miles and initializes ferry_miles to zero. The remaining changes are enhancements that involve introducing the ferry module. Figure 3a shows the final module structure of the restructured and enhanced system.

The tool eased the structural change of splitting the two miles concepts by creating the total_miles function and finding all uses of the original expression, but let the engineer decide which uses represented the abstraction total_miles. The actual enhancement—adding ferry miles to rolled miles throughout the system—was simplified by encapsulating the total-miles concept in a function. This localized the travel concept: only one addition was required to incorporate ferry miles.

In the absence of a restructuring tool, a software engineer would likely perform the enhancement by making the fewest changes possible. To do this, the engineer could augment each expression referencing the original miles_traveled variable to include a reference to ferry miles. In the short term this minimizes the cost of maintenance and the chances of introducing an error, but also distributes several identical expressions throughout the program. Thus, a subsequent change to the miles concept would require finding and changing all those expressions again. In the restructured version, by contrast, there is only a single instance of the expression to change. Figure 3b

[2] If the search finds an instance that *updates* rolled_miles, and if the engineer wants it to be total_miles, there is probably an encapsulation violation in the system. That is, rolled_miles should be updated solely by readings from the odometer in the bus module. However, this could occur if there is already another kind of mile that is not strictly a rolling mile. The substitution by **scope-substitute-call** on such a match is prohibited because it is syntactically illegal to assign a value to a function call.

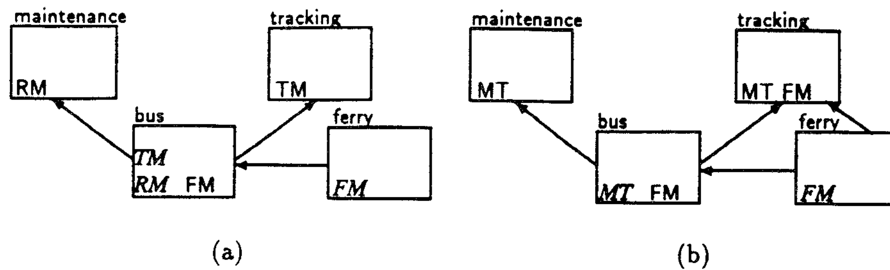

Fig. 3. Export-import structure of the enhanced system with restructuring (a) and without (b) (**RM** = rolled_miles, **FM** = ferry_miles, **TM** = total_miles, **MT** = miles_traveled, variable declaration in italics).

depicts the enhanced (but unrestructured) system, which shows the nonlocalized references to miles_traveled (MT) in the maintenance and tracking modules.

3. TRANSFORMATION-BASED RESTRUCTURING

The transformations in the tool have several basic properties. First, when a transformation is applied by the engineer, it is guaranteed either to succeed and produce a new program with the same meaning as the original program, or else to fail and leave the program unchanged. Second, the engineer applies a transformation to a syntactic construct. Third, to preserve the meaning of the program, the tool will typically make nonlocal changes to compensate for the local syntactic change. These properties assure preserving the program's meaning and free the engineer from the work involved in the updates. In essence, the tool is a semantics-preserving structure editor, allowing the engineer to focus on the design aspects of the restructuring.

A tool transformation, when applied by the engineer, makes a meaning-preserving structural change by modifying how the relationships between program components are expressed without affecting the inputs to operations that perform actual computation (such as +, −, or write). So, for example, the only legitimate changes to a procedure involve the manipulation of its name, its parameters, and the location of the computations within its body. None of these changes alone is likely to preserve meaning, so each implies particular updates to the calls on the procedure. Changing the procedure's name implies changing the names of calls to it; changing the order of parameters implies changing the order of arguments in the requisite calls, and moving an expression out of the procedure implies adding a parameter to the procedure (and an argument to each call) to pass the value of that expression in when called. Checks are typically performed by the tool to assure that these changes to the calls are sufficient to preserve meaning. The globalization paradigm, then, exploits the link between a definition of an abstraction and its uses. Since there are normally many uses for a single definition, it is natural that, for most transformations, the engineer's change is specified on the definition, and the uses are updated by the tool. However, in some cases

the roles may be reversed if the transformation is to disassociate the relationship of a particular use from its definition, say by inlining the abstraction at the point of the use.

3.1 Techniques for Representing Structure

The usefulness of a transformation can be shown by demonstrating how it changes modules to modify structure. Harold Ossher described the following basic structural mechanisms that support concise organization of information, reuse, and localization of changes [39].

Grouping identifies a set of program components as being part of an aggregate component. Denoting a group denotes all the components within it. Typical grouping constructs are the statement sequencing construct, the function body, and the module body. *Abstraction* allows identifying a (possibly grouped) component through a protocol that hides the details of the internals. Procedures and modules fall into this category. A procedure's protocol is a name and a list of parameters to be processed by the named function. A module's protocol is the union of its exported procedures' protocols (with implicit or explicit constraints on how they may be ordered). Abstraction is useful because it allows reuse of a potentially complex program component through a simple interface, and there is only one instance of the component's internals. This means that a single change to the definition automatically propagates to all uses without any further change—as long as the interface does not have to be changed as well. *Analogy and deviation* constructs a component by exploiting its similarity to an existing component, and then adding some things to account for differences. An example is the wrapping of a procedure with another to modify its output. Using inheritance to build a subclass is another. Closely related to deviation is *approximation*, which is defining a component that similar to what is desired, but not quite. The desired component can be created through deviation, and the approximation can be reused to help create other components. A superclass is an example of an approximate component that can be reused through the deviation process of subclassing.

There are two basic structural problems that can arise when considering a software change, both of which force the change of a single concept into nonlocal consistent updates on multiple program components. Our tool is capable of assisting with both of these.

The first problem occurs when two components that are not strongly related are contained in the same abstraction. This incidental relationship can lead to an intertwined implementation of the components [40, 50]. This implies that changing one component can undesirably affect the other; avoiding these undesirable effects may require compensating changes to the other component. Commingled components can be separated with meaning-preserving transformations that remove the offending abstraction, grouping, or deviation of the components.

The second problem is an unnecessarily low-level relationship between two conceptually related components. Consider, for example, two modules that must execute the same code. If the code is replicated in the two

modules, then repairing the code requires finding and changing the code in both modules. However if the code is encapsulated in a procedure, then procedure calls can be used to execute the code, and only one change is required to repair it. Such a relationship can be pulled into an abstraction using meaning-preserving transformations that abstract the implementations into a single function definition. In the transit example, for instance, the **extract-function** transformation created a procedure abstraction of the concept total_miles, which was implemented by returning the rolled_miles variable. This localized the addition of ferry miles to rolled_miles, since the only reference to rolled_miles was within total_miles.

3.2 Transformations

This section introduces many of the transformations useful for restructuring programs. (Although the transformations are relevant to most any imperative programming language, they are described with LISP's terminology, rather than, say, Pascal's. For example, the term *expression* refers to any syntactically complete component in the program.) These transformations were derived from experience in restructuring and have proved sufficient thus far. The next section shows how a Scheme matrix multiply program is relayered using them. Omitted from this list are transformations that manipulate data structures, such as grouping scalar variables into a record, or converting a scalar variable into a pointer or list variable [24].

Moving an Expression. Moving a program component is perhaps the most common transformation—usually as part of another transformation, but also on its own via **move-expr**. When moving an expression, there are no compensating transformations, just checks to ensure that the change preserves meaning. In particular, the bindings of the variables referenced in the expression cannot change; the movement will change the order of evaluation of some expressions, which must not change the values returned from expressions; finally, the moved expression must be evaluated in the same circumstances as before.

Moving an object closer to others, specifically when it is being moved between scopes, is regrouping. Also, moving an object next to another will allow them to be grouped by a subsequent transformation.

Renaming a Variable. Transformation **rename-variable** takes a variable binding and a new name and renames the variable and all its uses. The transformation must check each use site of the variable to make sure that the new name does not conflict with any existing names.

Renaming is important to structure, for example, because it may be desirable to group two objects of the same name that previously were in separate scopes. One must be renamed to make this possible. It was used in the transit example to give a more precise name to the variable miles_traveled to reduce ambiguity with respect to other names being used, such as ferry_miles.

Inlining an Expression. The transformation **var-to-expr** replaces the uses of a variable definition with the defining expression. The engineer selects the

assignment to be inlined and the tool handles finding and inlining the uses. An alternative version of the transformation takes a single variable use and inlines only the single use, not others from the same definition. Since the expression is moving, it must satisfy all the conditions for a move to each of the variable uses. Additionally, if the result will allow multiple evaluations of the expression, the transformation must assure that there are no side-effects in the expression, since repeating the side-effects could change the meaning of the program. Similarly, if there are multiple uses to be inlined, the expression cannot, in general, have side-effects. Finally, if a use has two potential definitions—this can occur with a conditional assignment—the transformation is prohibited.

A variation of **var-to-expr** is **binding-to-expr**. Given a variable binding by the engineer, the transformation finds all the assignments to the variable and performs **var-to-expr** on each. When this is applied to a function definition's parameter, the tool must check that each call of the function passes the same expression for the argument being inlined. This is because each call represents an independent binding of the parameter, but when inlined all the binding expressions must be merged into the one inlined expression.

Both versions allow specifying whether the variable binding should be deleted (if possible). The flexibility is important because it is not always clear that it should be deleted. The engineer may have another use for the variable after the transformation is complete.

Inlining removes abstraction. It generates multiple instances of the abstracted program component, allowing individual instances to be modified without affecting others. After applying this, it is possible to change an inlined instance without affecting the other uses. The version that inlines function bindings not only inlines the parameter, removing its abstraction, but it also narrows the interface of the function abstraction.

Abstracting an Expression. Given an expression by the engineer and a name and location for a binding, **expr-to-binding** performs roughly the inverse of **binding-to-expr**, moving an expression into a scope binding, assigning the result of the expression to the new binding variable, and putting a reference to the variable in the old location of the expression. This can be successful only if the expression at the new location would have the same value as in its original location. It also requires that the newly defined variable binding not conflict with the scope of any existing bindings of the same name.

This transformation has a broader impact when taking an expression from inside a function body and abstracting it to become a parameter of the function. This requires that the abstracted expression be passed as an argument in all calls on the function. The benefit of this is that the resulting function is parameterized and hence more general.

Abstracting a Function. The transformation **extract-function** turns a sequence of expressions into a function, and if so desired, it replaces the

abstracted statements with a call on the function. This is a more general version of expression abstraction; however, since a function can take parameters, variables that would be undefined in the new location have their values passed into the procedure as arguments. The engineer provides the expressions, a name for the function, the expressions or variables to be made parameters, the names of the parameters, and the location of the new function. Transformation **extract-function** is implemented by first creating the function inline, and then doing an **expr-to-binding**. For this transformation to succeed requires that the right parameters get abstracted so that moving the function to a new location does not leave any free variables. The tool can find these parameters if asked by the engineer.

This has the same benefits as expression abstraction, but with more potential for reuse, and hence localization. Also, by abstracting a new function from a significant part of an existing one, the new function can be treated as an approximation of the original. In the transit example, **extract-function** was used to localize the reference to rolled_miles with the function total_miles.

Procedure abstraction is reversed with **inline-function**; it can remove undesirable grouping and abstraction.

Scope-Wide Function Replacement. The transformation **scope-substitute-call** replaces repeated sequences of the code of an existing function with calls on the function. This is often used after function extraction.

Matching repeated sequences is complicated by the fact that when trying to match a function to an expression, the function's parameters must be matched against actual code. That is, to match a function to inline code, an inferencer must try to select which parts match the function body and which parts should be passed as parameters. This requires something like logical inference, although this is not straightforward since two references to a function parameter can have different values (due to side-effects). This means that function parameters cannot be treated strictly as logical variables.

Even without parameters, finding the repeated codings requires a program equivalence test, which in general is infeasible [16]. However, there are conservative techniques that are fast but can still yield nontrivial matches [57] (See Section 4.3). As a backup, a heuristic technique can be used, such as comparing the usage of an expression with the usage of the function call that is being substituted. For example, if each produces a value of the same type, and if each uses variables of the same types, then there is some likelihood of a match. Of course, a heuristic requires guidance by the engineer.

User approval of each substitution is also required because two semantically identical expressions are not necessarily instances of the same abstraction. For example, in the transit example, instances of rolled_miles were to be replaced with calls to the semantically identical miles_traveled function. However, the references to rolled_miles in the maintenance module were to remain unchanged, since they did not (conceptually) represent total miles traveled, but only those miles that resulted in wear and tear on mechanical

components. This required allowing the engineer to filter matches. Comparing the usage of matching expressions, as suggested in the previous paragraph, works well here since semantically identical expressions that are the same abstraction will tend to be used in a similar way.[3]

Scope-wide function replacement has the same localization benefits as function extraction, but it can help recover structure *after* redundant code has been introduced. Transformation **extract-function** can be used only to avoid future redundant coding.

This transformation is a slight departure from the paradigm of local change by the engineer compensated by the tool. There is no local change, and so there is no necessary compensation—hence the need to prompt the engineer for each substitution. What is really changing is the engineer's perceived structure of the system and how it is best represented. The tool helps by finding all the candidates, making the substitutions, and assuring that the substitutions preserve meaning.

3.3 Restructuring a Matrix Multiply Program

To show how these transformations are used to restructure, we transform a matrix multiply program [17] shown in Figure 4a. Matrices in this program are represented as vectors of (equal length) vectors, although the representation is hidden from the multiplication function through the use of auxiliary functions.

The restructuring centers around three local functions and an implicit, inlined function embedded in the main function. Extracting a form of these functions may prepare for later functional changes or for reuse by other programs. The first two local functions, matrix-rows and matrix-columns, respectively report the number of rows and of columns in a matrix. The third local function, match-error, reports an error if the two matrices do not match in size. This function also has an embedded constant, 'matrix-multiply, used for reporting the name of the function that received the incorrectly sized matrices. All of these functions, if at the top level, could be reused to implement, for example, a matrix-add function. Finally, the part of the inner loop of the matrix multiply that computes the innerproduct of a row of one matrix and a column of another is an operation that could be extracted and invoked as a separate function.

The restructuring tasks are:

—Modify match-error to accept a parameter that is the symbol name of the function that received the mismatched matrices.

[3]Alan Demers (personal communication, 1991) suggested this technique for eliminating spurious equivalent matches. The problem posed was the removal of the embedded uses of the file number 1 in C programs. Since C programs use 1 for incrementing integers, incrementing structure pointers, and as a file number for the standard output, there are too many matches for the engineer to interactively filter. However, restricting the tool to selecting matches that only use the number 1 in write is sufficient to differentiate those uses that should be abstracted as file numbers.

```
(define 1+ (lambda (x) (+ x 1)))

(define make-matrix (lambda (rows columns)
  (do ((m (make-vector rows))
       (i 0 (1+ i)))
      ((= i rows) m)
      (vector-set! m i (make-vector columns)))))

(define matrix? (lambda (x)
  (and (vector? x)
       (> (vector-length x) 0)
       (vector? (vector-ref x 0)))))

(define matrix-ref (lambda (m i j)
  (vector-ref (vector-ref m i) j)))

(define matrix-set! (lambda (m i j x)
  (vector-set! (vector-ref m i) j x)))

(define matrix-multiply (lambda (m1 m2)
  (letrec
    ((match-error
      (lambda (what1 what2)
        (error 'matrix-multiply
               "~s and ~s are incompatible operands"
               what1 what2)))
     (matrix-rows
      (lambda (x) (vector-length x)))
     (matrix-columns
      (lambda (x)
        (vector-length (vector-ref x 0)))))

    (let* ((nr1 (matrix-rows m1))
           (nr2 (matrix-rows m2))
           (nc2 (matrix-columns m2))
           (r   (make-matrix nr1 nc2)))
      (if (not (= (matrix-columns m1) nr2))
          (match-error m1 m2))
      (do ((i 0 (1+ i)))
          ((= i nr1) nil)
        (do ((j 0 (1+ j)))
            ((= j nc2) nil)
          (do ((k 0 (1+ k))
               (a 0 (+ a (* (matrix-ref m1 i k)
                            (matrix-ref m2 k j)))))
              ((= k nr2) (matrix-set! r i j a))
            nil)))
      r))))
```

(a)

Fig. 4. Matrix multiply before restructuring.

—Move the three local functions to the top level; then remove the letrec that contained them.

—Make the inlined innerproduct function a callable, top-level function that computes the dot product of a row of one matrix and a column of another. The parameters to the function are to be the two matrices and the

```
(define 1+ (lambda (x) (+ x 1)))

(define make-matrix (lambda (rows columns)
  (do ((m (make-vector rows))
       (i 0 (1+ i)))
      ((= i rows) m)
      (vector-set! m i (make-vector columns)))))

(define matrix? (lambda (x)
  (and (vector? x)
       (> (vector-length x) 0)
       (vector? (vector-ref x 0)))))

(define matrix-ref (lambda (m i j)
  (vector-ref (vector-ref m i) j)))

(define matrix-set! (lambda (m i j x)
  (vector-set! (vector-ref m i) j x)))

(define match-error
 (lambda (what1 what2 header)
  (error header
         "~s and ~s are incompatible operands"
         what1 what2)))

(define matrix-rows
 (lambda (x) (vector-length x)))

(define matrix-columns
 (lambda (x) (vector-length (vector-ref x 0))))

(define inner-product (lambda (m1 i m2 j)
  (do ((k 0 (1+ k))
       (a 0 (+ a (* (matrix-ref m1 i k)
                    (matrix-ref m2 k j))))
       (len (matrix-rows m2) len))
      ((= k len) a) nil)))

(define matrix-multiply (lambda (m1 m2)
  (let* ((nr1 (matrix-rows m1))
         (nr2 (matrix-rows m2))
         (nc2 (matrix-columns m2))
         (r (make-matrix nr1 nc2)))
    (if (not (= (matrix-columns m1) nr2))
      (match-error m1 m2 'matrix-multiply)
      nil)
    (do ((i 0 (1+ i)))
        ((= i nr1) nil)
      (do ((j 0 (1+ j)))
          ((= j nc2) nil)
        (matrix-set!
         r i j (inner-product m1 i m2 j))))
    r)))
```

(b)

Fig. 4. (b) Matrix multiply after restructuring.

respective row and column. The code in the inner loop that computes this in the original program is to be replaced by a call to this new function.

The result of restructuring matrix multiply is shown schematically in Figure 5. Solid boxes are abstractions, and dotted boxes are groupings. An

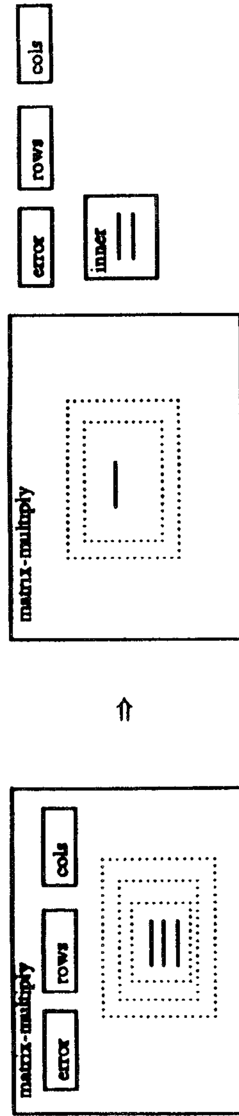

Fig. 5. Grouping and abstraction diagram of matrix multiply's restructuring.

abstraction can only see those abstractions at the same level, above it, or those immediately inside it. On the left, then, there is only one abstraction accessible at the top level, while on the right there are five. Four program components that were not externally accessible before—one of them not abstracted—become independent abstractions. The ease of reuse of these four new top-level functions will be greater because they span a larger scope. As a secondary consequence, the internal structure of the main object will be simpler.

The computer-aided restructuring of the matrix multiplication program is relatively straightforward. The first step in the restructuring is to add a parameter to generalize match-error. The command

```
(expr-to-binding ⟨'matrix-multiply⟩ 'header :scope ⟨match-error⟩)
```

extracts the 'matrix-multiply constant from the original body and makes it a parameter, named header, of function match-error, and updates each call on match-error to pass 'matrix-multiply as a parameter. (In the text here we have altered the commands slightly by denoting components of the matrix multiply program with descriptive tags surrounded by angle braces ⟨ ⟩. These components are normally selected with pattern-matching directives.) The command checks that the abstracted value has the same value in its new context as it did in the old and that header is an acceptable name in the scope created by match-error (that is, there is no existing parameter or local variable of the error function with that name). In this case, the command succeeds and transforms the match-error function to

```
(match-error (lambda (what1 what2 header)
  (error header "~s and ~s are incompatible operands" what1 what 2)))
```

and updates the (only) call to

```
(match-error m1 m2 'matrix-multiply)
```

The second step is to move the three local functions to the top level and to remove the letrec that encloses them:

```
(move-expr ⟨match-error⟩ :before ⟨matrix-multiply⟩)
(move-expr ⟨matrix-rows⟩ :before ⟨matrix-multiply⟩)
(move-expr ⟨matrix-columns⟩ :before ⟨matrix-multiply⟩)
(ungroup ⟨letrec⟩)
```

The first three commands check to make sure that there are no name conflicts in the new scope; in this case they succeed and move the three functions to the top level, in front of the definition of matrix-multiply. The last transformation removes the now empty letrec from the body of the matrix multiply.

The last part of the restructuring, extracting the inlined inner product, is the hardest. There are three impediments to performing the restructuring directly.

First, the variable nr2 is used in the inner product computation but is defined by the enclosing main function; to allow the extraction of the inner product, the value of nr2 must be made available in the newly

extracted function. Second, the value that the inner loop computes is stored in the variable r; it must be returned as the value of the extracted function. Third, the inner product is not parameterized; before extraction, it must be explicitly parameterized by the two matrices (m1 and m2) and by the row and column indices (i and j).

To handle the first problem, nr2 must be split into another equivalent expression (called len):

```
(var-to-expr ⟨second nr2 reference⟩)
(expr-to-binding ⟨result of previous⟩ 'len :scope ⟨do⟩)
```

The tool checks that the new names will not conflict with others in the designated scope and that the recomputation of the binding produces the same value as before and causes no extra side-effects. The checks succeed, and the command transforms the inlined inner loop of matrix-multiply to:

```
(do ((k 0 (1 + k))
     (a 0 (+a (* (matrix-ref m1 i k) (matrix-ref m2 k j))))
     (len (matrix-rows m2) len))
    ((= k len) (matrix-set! r i j a))
 nil)
```

To handle the second problem, the independent part of the result expression must be moved out of the enclosing do loop.

```
(pop-out ⟨do⟩ ⟨a⟩)
```

This command moves the entire return result of the do loop, except for the second parameter (the reference to a in the matrix-set! command), outside of the do loop:

```
(matrix-set! r i j
  (do ((k 0 (1 + k))
       (a 0 (+a (* (matrix-ref m1 i k) (matrix-ref m2 k j))))
       (len (matrix-rows m2) len))
      ((= k len) a)
  nil))
```

Now the inner loop can be extracted.

```
(extract-function ⟨do⟩ 'inner-product
                  :old-new-name-pairs' (⟨m1⟩ ⟨i⟩ ⟨m2⟩ ⟨j⟩)
                  :before ⟨matrix-multiply⟩)
```

The parameter **:old-new-name-pairs** is the list of variables or expressions to be abstracted as parameters to the new function. New names can be supplied as well, but here are defaulted to their current names. Note that if the engineer had tried to apply this transformation without moving out nr2 and r, the value of the computed inner product, the tool would have aborted the extract-function transformation, returning the error message:

Variables nr2, r would be unbound in new context.

This completes the restructuring of the program, which is shown in Figure 4b.

3.4 An Experiment

We asked six programmers to perform the restructuring of matrix multiply by hand [25]. Even on this program, which fits on one page, four of the programmers made errors on their first attempt. We observed that the programmers tended to interleave logically independent activities, apparently in an attempt to reduce the number of editing steps. We believe this to be one source of restructuring errors. Their approach to restructuring tended to be a copy/paste/edit paradigm, with some instances of cut/paste/edit. This was used, for example, in extracting the inner-product function. The copy/paste/edit approach proved easier because it is easy to compare the edited copy with the original, but it is potentially slower because it requires an additional editing step to go back and delete the original unedited text. Cut/paste/edit proved slower in practice because extra editing steps were required to recover the original copy. In either approach, based on our observations, we believe that the physical distance between the group of changes comprising a restructuring step increased the chance of errors.

Because our tool combines all the intermediate editing tasks for the programmer, it is not necessary to interleave logically independent changes to speed restructuring. Combining the steps also relieves the programmer of reasoning about the textually separated changes that comprise a restructuring step. Finally, our tool interface requires the engineer to select an object and a transformation on it, and the object, in practice, is the same object that would be cut or copied if manually modified. This implies that the restructuring interface is sufficiently similar to an editor that it should be easy to use.

3.5 Discussion

Is our set of transformations sufficiently powerful to allow localizing appropriate design decisions in a restructured program? To localize any property means being able to colocate any subset of program components within a module. A proof of whether a set of transformations can do this requires precise knowledge of the transformations, the programming language, and perhaps even the program. Here we take a more informal approach. The transformations shown above have been derived in the process of restructuring programs by hand and with the tool. Some of these have also been suggested in the literature [11, 27]. Although the current set may not be complete, it is not difficult to add new transformations as needed (see Section 4.4).

An additional program we restructured with the tool comes from Parnas' case study of modular structure for the KWIC program [24]. KWIC takes a list of lines of text and produces a list of all circular shifts of those lines, in sorted order. Parnas used KWIC to demonstrate the structural principle of information hiding as a criterion for module decomposition [41]. We successfully used our tool to restructure a poor modularization of KWIC to a structure preferred by Parnas.

Parnas' study compared two modularizations of KWIC, a *functional* and a *data* decomposition. The functional decomposition models control flow, mak-

ing no attempt to isolate data representations. For example, the representation of the line storage is accessed directly by the input task module and the circular shift task module. The data decomposition, conversely, isolates the representation inside line access routines that comprise a line storage module. Parnas' analysis showed that a data decomposition of KWIC will be easier to modify than a functional decomposition.

Parnas emphasizes that the two decompositions can share all representation choices and algorithms—only what is isolated in a module need be different. Using our prototype restructuring tool, we successfully restructured a Scheme implementation of a functional decomposition of KWIC to a data decomposition, showing that Parnas' claim is systematically supported by our tool. Restructuring with the tool before performing other changes yielded two major benefits. First, the tool performed the necessary nonlocal changes to preserve meaning. Second, as a consequence of the new data decomposition of KWIC, many likely subsequent changes—those to representations and algorithms, for example—are now confined to a module, whereas before they were not. These subsequent changes are not themselves automated, but as Parnas argued, they are local, and hence easier to reason about.

4. MODEL AND IMPLEMENTATION

Our tool's implementation must preserve meaning and perform nonlocal updates, while at the same time preserving the readability of the program. These properties can be managed by using three program representations: the source,[4] the control flow graph (CFG) [2], and the program dependence graph (PDG) [22, 31]. (See Appendix A for definitions of the CFG and PDG.) The source includes information about scopes, for instance, which is unavailable in either the CFG or PDG. The CFG contributes information about the ordering of statements. The PDG acts as the arbiter of meaning of a program [30]. The PDG also simplifies the application of distributed changes in the program source, since these are confined to a local neighborhood in the PDG.

Consistently managing these three representations is difficult [26]. Earlier approaches, such as those used in program version merging [29, 56] have modeled the relationships between program, CFG, and PDG as mappings between representations. Our approach additionally relates the *functions* that transform each of these representations. The data mappings from the AST to and from the CFG and PDG are used for composing piecewise-local program source transformations into a single global transformation. Function mappings are first used to reason about the correctness of global transformation implementations, and then to make them efficient. In particular, a program source transformation must map to a sequence of meaning-preserving local substitution rules on the CFG and PDG. The

[4] We use the terms "program," "source," "text," and "Abstract Syntax Tree" (AST) casually and interchangeably. Although the representations are not identical, the mappings between them are straightforward—especially for Scheme programs, our prototype's domain.

resulting substitution transformations then can be applied directly to the CFG and PDG, which is faster than reconstituting them from the source.

There are at least three ways to handle the relationships among these representations, as shown in Figure 6. (The figure, and the following discussion, are simplified by introducing the notion of a combined CFG/PDG, rather than by discussing the two separately. In practice, the representations are indeed separate, but for these purposes the reasoning about and management of the two are essentially identical.)

Figure 6a depicts an approach in which the program P is translated by the CFG/PDG construction function m_d into its CFG/PDG form, G. The transformation δg is applied to this representation to produce the new CFG/PDG G', which is then converted (i.e., unparsed) back to the source form P'. This allows using the nice mathematical properties of the CFG/PDG for reasoning about the correctness of the implementation of the transformation function, as well as for the natural and efficient application of semantically oriented algorithms when the transformation is applied. This approach is problematic because it requires restructuring in absence of syntactic constraints (and can even lead to syntactically illegal programs). For example, a pure PDG collapses scope information from *permitted* relationships to *actual* relationships. This absence of scope information might allow a statement in a source reconstituted from the PDG to be located in a different scope than it was originally, because the statement does not use any variables defined in its immediately enclosing scope. In this case, CFG or source information is needed to assure that this problem does not arise. Another problem is the cost of reconstituting the AST after each transformation to permit further transformations by the user. Although the fact that program reconstitution from a PDG is NP-complete is not considered a serious problem in practice [30], even an algorithm linear in the program's length may be too slow for interactive transformation.

A second approach, shown in Figure 6b, has the user apply the source transformation δp. Since δp directly manipulates the source, precise syntactic control is assured, and scoping semantics are easily checked. However, the CFG/PDG notation is still used to show that the transformation is implemented correctly, and G is used at runtime to ensure that necessary semantic properties hold in P. In this approach, then, the benefits of both representations are achieved, but as suggested by the dotted arrow, CFG/PDG G' is reconstructed from P' after δp is performed. However, the cost of reconstructing the CFG/PDG G' from scratch is prohibitive.[5]

A third approach, then, shown in Figure 6c, suggests performing both δp and δg (which are associated by m_f) directly on each representation, and bypassing the cost of the batch application of m_d or m_d^{-1} to keep P' and G' consistent. (Of course, the initial construction costs of G from P using m_d must be paid, but subsequent transformations do not incur this cost.[6])

[5]For example, Larus' Curare system requires at least $O(|program|^3)$ to construct a PDG that incorporates reasonably precise alias information (personal communication).

[6]Function m_d or m_f is referred to as m when the type of the input—data or function—is clear.

Three steps are needed to realize this solution. First, a way is needed to map a source transformation to an associated CFG/PDG transformation. This mapping must find δg isomorphic to δp without applying δp. This avoids constructing G' from scratch and comparing $m(\delta p(P))$ to $\delta g(m(P))$. Second, a way is needed to show that the resulting CFG/PDG transformation *a priori* preserves meaning (thus showing the source transformation preserves meaning). Third, during transformation, a way is needed of mapping back and forth between AST and CFG/PDG data elements so that semantic queries on the AST can be executed using the CFG/PDG. These runtime mappings of data can be efficiently handled using relations that associate AST vertices with CFG/PDG vertices. We focus on the first two points, as the third is straightforward.

Although the following discussion focuses on the use of CFG/PDG transformations to preserve program meaning, the scenario in Figure 6a suggests that a program transformation is not a meaning-preserving one unless it is successful in all the representations. For example, the corresponding source transformation must be syntactically legal, and any semantic properties better represented in the AST, primarily scoping, must be verified there. For example, any newly introduced variable must not conflict with existing variables in the program.

4.1 Constructing m_f

The formal basis for constructing a mapping from functions on the program source to functions on the CFG/PDG is in its infancy [12, 46, 53]. Lacking a formal basis, we developed a practical technique that relies on the transformation builder's knowledge of the programming language, the CFG, and the PDG to relate transformations. There are two practical difficulties in mapping transformations between representations. First, source transformations are global and thus difficult to reason about. Second, a way of relating a change in one representation to a change in another is required. We handle the first by defining a procedure skeleton that connects a collection of local source transformations with semantic relationships defined in the PDG. We handle the second by providing a simple set of CFG/PDG subgraph substitution rules. As defined by the globalization skeleton, a global program source transformation is mapped to a sequence of these substitution rules. The substitutions preserve meaning of a CFG/PDG, so they not only aid in the mapping, but at the same time assure that the source transformation preserves meaning with respect to the PDG. The result of using these is a set of pairs $(\delta p, \delta g)$—one for each transformation—yielding an effective m_f. First we present the skeleton, and then the substitution rules.

4.2 Decomposing δp Using G

Our paradigm for global transformation—that a tool user applies a local change *local_trans* to an expression **e** in P, and the tool applies the necessary compensating changes via *compensation_trans* to the program components affected by **e**—is described by the following procedure skeleton, which

we call the *globalization skeleton*:

```
procedure delta_p(e)
  for u_i ∈ uses(e)do
    compensation_trans(u_i)
  local_trans(e)
end
```

where **uses** retrieves the set of references to the value produced by **e**. As long as the uses are computed before transformation begins, the sequencing of the compensations is unimportant, since each use is in a syntactically independent location. Each of *local_trans* and *compensation_trans* are typically simple movements, copies, and substitutions, along with some deletion and creation of syntax to represent the new structure (see Section 4.4).

Two issues must be resolved for this skeleton to be useful: the computation of **uses** and showing that the resulting transformation preserves meaning. The functions **uses(e)** can be computed by $m^{-1}(fs(m(\mathbf{e})))$, where fs retrieves the flow dependence successors of a CFG/PDG vertex, which in this case are the uses of **e**. To assure that a transformation preserves meaning, the transformation builder shows that the transformation obeys some basic properties—described in the next section—and implements a check function for those properties that must be checked at runtime (see Section 4.4).

The skeleton, with this definition for **uses**, aids in implementing a transformation and its check by delineating several aspects of multiple-representation transformation: between implementation time and run-time tasks, between data mappings and transformation mappings, and between tasks best performed in the AST versus the PDG. For example, the local program transformations *local_trans* and *compensation_trans* are implemented by the transformation builder (who also maps them to a CFG/PDG transformation, as described in the next section), and m_d and m_d^{-1} are used by the tool at runtime to access semantic relationships naturally represented in the CFG/PDG.

4.3 Meaning-Preserving Graph Substitution Rules

The program transformation initially derived with the aid of the globalization skeleton is completed and demonstrated correct by mapping *compensation_trans*($\mathbf{u}_i$) and *local_trans*(**e**) to the PDG as a composition of PDG subgraph substitution rules, yielding δg. The PDG substitution rules are designed to preserve flow dependence and not change the operations in value-changing vertices.[7] Preserving dependence between two vertices preserves the semantic link between them [42]. Not changing the mutating operations preserves the actual values passed along the preserved dependences.

[7]These rules are described solely in terms of the PDG, rather than the combined CFG/PDG, because the substitutions on the CFG portion are straightforward because of its structural similarities to the AST and PDG.

This notion of equivalent meaning is rigorously applied by the Sequence-Congruence algorithm [56], which computes equivalence classes of equivalent programs or subprograms. Members of an equivalence class, over the course of a program execution, produce equivalent sequences of visible states. Although the definition of the Sequence-Congruence algorithm uses a variant of the PDG called a Program Representation Graph (PRG), with appropriate modifications the algorithm still applies to PDGs. The key difference between the PDG and PRG is that the PRG uses normalized variables in the style of static single-assignment (SSA) form [15] (eliminating antidependences and output dependences) to make the Sequence-Congruence algorithm faster. The PRG form might be useful for restructuring. However, the normalization process introduces new variables that appear to either complicate mapping between the source and the PRG or else compromise the readability of the restructured source.

By definition, subgraphs of a PDG may be modified by the substitution of sequence-congruent vertices without changing a PDG's meaning. For example, creating common subexpressions [3] in a PDG by replicating a vertex and its incoming edges (the result by definition is sequence congruent to the original) and splitting the outgoing edges between the two copies preserve meaning. Thus PDGs can be transformed by performing replacement of sequence-congruent vertices in the PDG. This notion is used in motivating the PDG subgraph substitution rules, and the algorithm itself is used in substitutions where it is necessary to identify equivalent subgraphs.

Briefly, two of the substitution rules can be described as follows. The transitivity rule represents the fact that x := y has the same meaning as temp := y; x := temp, if temp is not otherwise used. The distributivity rule states, roughly, that if an expression's result is assigned to a variable that is used in multiple locations, then a copy of that expression (i.e., it is sequence congruent) may be evaluated to produce the value for one of those locations. In Figure 7 the copying of the + vertex and moving an edge over to it is an example of applying the distributivity rule. There are also rules for renaming variables, introducing (or removing) indirection to values, and modifying control flow [24]. Most of these are strict equivalence rules, so their inverses apply as well.

4.4 Example: Using the Model to Design **var-to-expr**

How are the globalization skeleton and the substitution rules used? Consider the transformation **var-to-expr**, which replaces each use of a variable definition with a copy of the defining expression. The user selects the assignment to be inlined, and the tool handles finding and inlining the uses. Since the expression is moving to its uses, the transformation must check that none of the inputs to the expression is changed by the move. Additionally, if the result will allow multiple evaluations of the expression, the transformation must assure that there are not side-effects in the expression. Finally, if a use has multiple potential definitions—this can occur with a conditional assignment—the transformation is aborted.

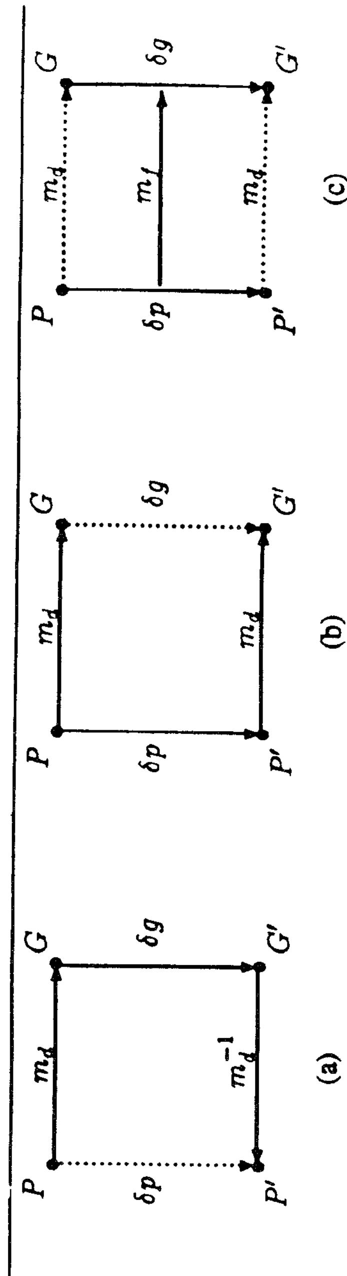

Fig. 6. Diagrams of transformation (dotted arrows are implied mappings).

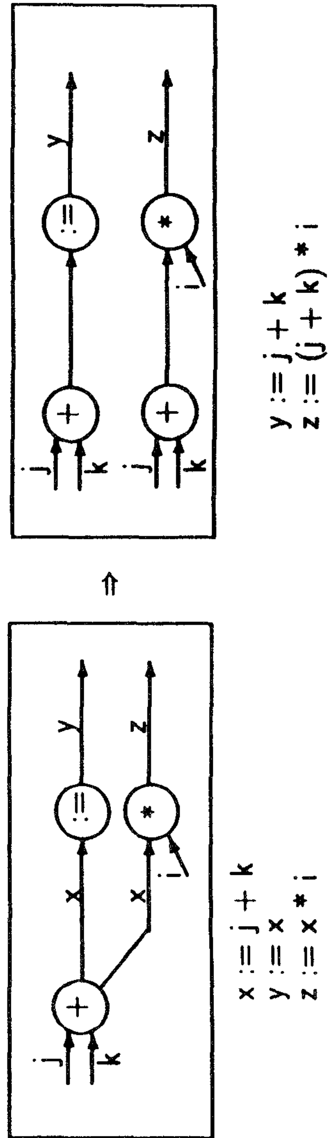

Fig. 7. A flow diagram for an application of **var-to-expr**.

The **var-to-expr** specialization of the globalization skeleton, where [v, e] is the expression that defines v, is:

```
procedure var-to-expr([v, e])
  for u_i ∈ uses([v, e]) do
    substitute(copy(e), u_i)
  remove([v, e])
end
```

In addition to these local AST transformations connected by the *uses* relation in the PDG, the source transformation must assure scoping and syntactic properties. For example, moving e must not move a variable reference in e out of the scope in which it is visible. On the syntactic side, each inlining of e results in replacing a variable reference with a direct evaluation of e. In a program there is no way one evaluation of e could be used in different syntactic locations without an explicit variable to transmit e's value, although in the PDG this would be perfectly legal. This constraint is achieved *a priori* by designing the transformation on the AST, rather than the PDG.

It remains to be justified with substitution rules that **var-to-expr** is meaning preserving (refer to Figure 7). First the distributivity rule justifies replicating e (e.g., j + k on the left in Figure 7), moving outgoing edges of the original e to its copies and relabeling them (e.g., inlining j + k and removing the references to x because the result is now directly transmitted).[8] The null label for the edges is dictated by the source transformation's inlining of e. Second, the rule for modifying control flow specifies when it is legal to move e to replace $\mathbf{u}_i$ that are in a conditional.

Some runtime checks are required to assure that the substitution rules are really being obeyed by the AST transformation; these are the same checks described in the definition of **var-to-expr** above. For example, the use of the distributivity rule requires the runtime check verifying that there are not multiple definitions of the variable reaching the vertices being modified.

The resulting transformations can be used to transform the AST, CFG, and PDG simultaneously. The transformations are efficient because they are incremental, modifying only the data necessary, and they use the best of the three representations for every query. In particular, all the components to be modified (including edge types that we have omitted here due to space considerations) are found within one PDG edge of the locally modified expression.

4.5 Implementation

The prototype consists of over 30,000 lines of Common Lisp code, of which approximately half is the CFG/PDG package defined by Larus for his Curare system [32].

The implementation of our prototype tool closely follows the definition of the model. There is a module for each of the AST and CFG/PDG, and data

[8] Because an assignment is being deleted in the program text, it appears that the transitivity rule is being applied, but in the graph the associated edge is not deleted, just relabeled.

mappings between them are implemented with hash tables. The PDG module is borrowed from Larus' Curare implementation [32]. The transformations on the AST and PDG are implemented as procedures with side-effects. The maps between a transformation on the AST to its equivalent on the CFG/PDG are implemented by bundling calls on the two into a single procedure.

There are several layers of function in the prototype, as shown in the intermodule dataflow diagram in Figure 8. In the lowest layer is the independent implementation of the individual representations, the modules for the AST and CFG/PDG. The layer on top of them connects the two into a unified interface, enabling familiar syntactic access to information naturally represented in the CFG/PDG. This is aided by the data-mapping functions. Using the unified interface, the next layer defines a set of syntactic queries and transformations, as well as semantic queries. These are used for defining meaning-preserving transformations and are frequently reused to implement a new transformation. On a par with the meaning-preserving transformations is the mechanism that keeps the AST and CFG/PDG consistent with each other.

The current implementation does not support incremental update to the CFG/PDG for all transformations, just for variable renaming, moving expressions, and changes to simple grouping constructs. The remainder use a mechanism based on event-mediator integration [51, 52]. Also, the CFG/PDG implementation does not support first-class functions, but recent research in flow analysis for LISP languages suggests that suitable solutions are possible [47].

5. RELATED WORK

5.1 The Laws of Programming

C. A. R. Hoare et al. [27] demonstrated that imperative programming languages obey powerful and intuitive algebraic laws that permit source-to-source transformation. For example, there is a law that says a variable reference can be replaced by its defining expression. For example, given expression E, and expression F using x, F(x), then x := E; F(x) is equivalent to F(E).

A law as simple as this applies only to languages with restrictions on input/output, pointers, recursion, and procedure call to achieve a degree of referential transparency that normally is not present in imperative languages. In the example above, E and F, by definition of the language used by Hoare et al., are known not to have side-effects. This is important because the order of evaluation is potentially changed by substituting E in place of x. Also if x is referenced multiple times in F(x), then E is evaluated multiple times in F(E). If E were allowed to contain side-effects, the substitution would cause repeated side-effects, disallowing application of the law. Further, in an imperative programming language with global variables and procedure call, meaning-preserving substitution depends on how global variables are referenced in the procedure calls made in the expression. In this case the properties of an expression cannot be locally determined.

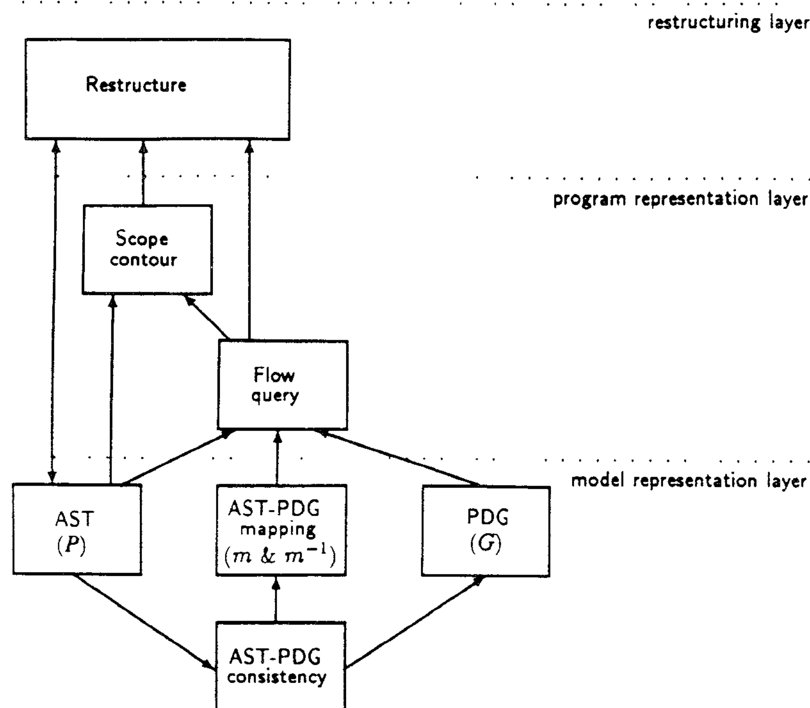

Fig. 8. The prototype's intermodule dataflow diagram.

However, it was this idea that suggested to us that there could be a rigorous approach to restructuring programs. The idea was applied manually to restructure a prototype operating system for a shared-memory parallel processor, easing an enhancement that allowed running different programs on different processors. The experiment was a success, but the referential transparency problems of the rules prevented straightforward automation of the manual process. To overcome this problem we generalized the technique to the PDG, which has more of the referential transparency needed to reason algebraically about a program.

5.2 Imposing Block Structure

A program using gotos can be automatically transformed into a program using only the structured flow graph operators *sequence*, *branch*, and *loop* [10]. The intent is to improve a program's structure to lower the cost of maintenance, a shared motivation with our work.

Most of the solutions involve simulating gotos with structured operators. One possibility is to use a large case statement inside a loop [55]. This is done by putting each jump-free section of code in the original program into a case

in the case statement. The case label can be the constant whose name is the original goto to that piece of code. Fall-through gotos are given an invented tag. Then at the end of each case a flag is set to the label of the goto that would have been jumped to next in the original program. When the top of the case is entered again on the next loop iteration, it selects the case corresponding to that label. This proves unsatisfying when the gotos are tangled, as the result is not much prettier than the original. Some approaches try to preserve the original structure of the program during goto removal [43]. These techniques use control flow graphs, an early precursor to, and important subrepresentation of, PDGs.

Automatic restructuring systems such as SUPERSTRUCTURE [37] and RECODER [21] have successfully exploited reorganizing program structure by removing or block structuring gotos to aid in the maintainability of goto-laden programs. These tools are batch oriented, avoiding the need for user input. Although this approach has shown some benefit in experiments, it was also observed that it distorts structure in programs with carefully designed use of gotos [21].

Although useful as a first step for programs with gotos, restructuring on control statements has limited applicability. The relationship among data, functions, and types are of interest in restructuring, but these are not addressed. Also, these batch-oriented techniques do not address aspects of structure that are not easy to quantify, such as restructuring toward a particular enhancement.

5.3 Transformational Programming

Introduced in the 1970's, transformational programming, also known as derivational programming, feeds a functional specification of an intended computation to a transformation system that, with guidance from a "programmer," rewrites the specification into an efficient program [11]. Thus most of the development effort is focused on the specification rather than programming, and there is a guarantee that the program satisfies the specification. The use of transformation is the key similarity with the restructuring work. Some basic transformations are:

—**Unfolding** substitutes a function's definition in place of its uses. Unfolding can expose identities that allow other transformations to happen, leading ultimately to its inverse, *Folding*.

—**Abstraction** substitutes a variable for every instance of an expression and defines that variable to be the value of that expression. This exposes similarities in the code.

—**Instantiation** substitutes a value or expression for a parameter. This permits splitting a problem into cases, such as a base case and recursive case.

The analogues of these transformations in the restructuring work are generalized to handle the semantics of imperative programming languages.

Recognizing the possibility to fold has the same equivalence problem as **global_substitute_function** and appears to use a similar pattern-matching technique.[9]

A significant detriment to transformational programming is the large number of transformations that must be applied to derive a program. This is not as severe a problem in restructuring because restructuring does not need to transform through successive language levels and also transform within a language. Transformation between language levels is essentially the choice of a lower-level implementation of a higher-level specification. Also, transformational programming makes more use of the algebraic knowledge of types, which, although important for equivalence, does not affect locality.

Another problem is the large catalogue of transformations that must be available for deriving programs. The catalogues are large enough to make it difficult to find appropriate transformations to apply. The reason this occurs is that there are large, potentially unbounded, numbers of appropriate implementations (choice of representation and algorithm) for a specification. Local transformations must also be supported. This is not a serious problem in the restructuring domain because the catalogue is practically bounded by the size of the language's syntax.

Recent work in derivational programming has attempted to alleviate the tedium of choosing and applying transformations by building up higher-level transformations from primitives [6, 20]. For example, Feather developed a technique that uses a pattern to express the goal of a transformation. Using a goal pattern, a tool can select the appropriate primitive transformations to compose to achieve the goal. Such techniques might be applicable to restructuring.

Other work has focused on trying to lower the costs of redevelopment by automating *re*derivation of a program from a modified specification [19]. The basic idea is to reuse the transformation sequence from the initial development to automate the programming tasks of maintenance. There are doubts about the success of these techniques because it appears that the ordering of transformations is brittle with respect to changes in the specification [38]. To avoid this problem and the others cited above, Narayanaswamy and Cohen's system uses declarative annotations that help the compiler choose transformations to derive an efficient program. This frees the programmer from transformation tasks, and an annotation needs to be changed only when the usage of the annotated program component changes. The specification and implementation are still separated, but the downside is that the specification language must be lower level for this technique to produce efficient programs.

5.4 Knowledge Representation Enhancement

A package of tools for performing structural enhancements of a knowledge representation system [5] has the same motivations as our research, but in a

[9]The presence of referential transparency in this domain does not help. Program equivalence is undecidable due to looping, or equivalently, recursion, which is what functional languages use to iterate.

narrower domain. The tools exploit the highly structured, declarative domain model of a knowledge base to infer the changes to assumptions caused by a structural enhancement. A tool locates the representations that use these assumptions, so the programmer can update them. In some cases, the system can perform the update as well, but lets the user provide additional input if more of a change is desired.

The changes supported are on types and attributes and include changing the supertype of a type or moving an attribute between types. The updates to the model are made to not only the declarations and the code that uses them, but also to the existing data modeled by the knowledge representation—an aspect not addressed in the restructuring work. Changing the type of an attribute can require translating the original values of the attribute into values of the new type. This can be automated when sufficient information about the relationship between the types is available.

It is not the intention of the tool's operations to preserve meaning per se; the propagation of changes proceeds only one step to direct the programmer to the directly affected locations. Thus the changes in structure and the change in meaning they allow are intended to go hand in hand. Balzer's approach is in contrast to our style of restructuring, which propagates the changes of the compensating transformations if necessary to preserve meaning. This difference is philosophically significant, as it is widely agreed that global changes are subject to error [8, 41], although at least with Balzer's work the user is directed to the sites potentially requiring change. It also requires significantly more work on the part of the engineer. Finally, if the user compensations are not meaning preserving, and they are global, then interfaces of the changed object are not being made any more robust; future change is not likely to be any more localized. Although structural enhancements may not require locality of change because of the tool, functional enhancements can receive no aid from the restructuring tool, and so will require manual global change. This suggests, then, that the most reasonable action in the user-driven compensations is to make meaning-preserving compensations that localize future changes.

The semantics of type structure and attribute structure can be handled by the techniques described in this paper, although they are not aspects that are so readily exemplified in Scheme [24].

5.5 Bounding the Effects of Changes

We aid maintenance activities by constraining (i.e., prohibiting) the semantic effects of changes, thus eliminating the chance for introducing errors during restructuring. Another approach is to constrain the changes themselves. In particular, it may be useful if a programmer can limit the semantic effects of enhancements to a particular subsystem, thus assuring that errors are not introduced in the rest of the system. Gallagher and Lyle [23] have proposed a slicing tool that computes the bounds on the effects of software changes. (Loosely, a *slice* for an output variable is all the statements in a program that may affect the value of the variable [54]. A tool using their technique can compute a union slice for the variables of a code fragment to be modified, and

then protect the slice of the statements *not* in that slice by limiting the editing commands that can be executed. After computing the union slice and the complement slice, the union slice is partitioned into statements that also belong to the complement slice (i.e., affect the values of output variables in the complement slice), called *dependent*, and those that are in the union slice only, called *independent*. They then constrain editing in the following fashion: Any independent statement may be deleted, and no dependent one can be deleted. A statement can be inserted if it does not modify any variables in the complement slice. Inserting statements to control dependent statements is not allowed. (Note that a change to a statement can be modeled as a deletion followed by an insertion.)

Although this technique bounds the effects of changes in a program, it has limitations. First, the slice is not the most natural unit of code. Slices in general do not correspond to functions or modules, but instead to the flow of data and control throughout a program. Realistically, a programmer may prefer to prescribe bounds that correspond to natural functional units. Second, the technique does not allow many changes. For instance, a slice typically shares dependences with other slices, and these expressions cannot be modified. Also, slices tend to be large [56], implying that the effects of changes are not tightly bounded. In this respect the technique may be inherently limited, because it is purely analytical—it does not exploit semantic information to actively preserve the meaning (e.g., by transformation) of subsystems selected by the user.

5.6 Vertical Migration and Good System Structure

One concern about "good" structure is that it can incur high execution cost overhead. Stankovic [48, 49] investigated the tradeoffs between good structure and performance, developing an execution cost model based on structure, and a structuring technique called *vertical migration* to improve performance in selected portions of code. To help identify modules that would benefit from vertical migration, Stankovic implemented an analysis tool based on the cost model. He also described a subset of transformations for improving program structure.

The cost model views a system as several layers of virtual machine, and each layer can call only the layer below it. The cost of a call from one layer into the next is broken into three parts, a *prologue* that performs some setup or perhaps checking on entry to the layer, the execution of the function called, and an *epilogue* executed on exit from the layer. For example, calling into a layer could cross an address space boundary, requiring a context switch in the prologue and again in the epilogue when returning. This model exposes two causes for unnecessary execution overhead. One, when one layer makes multiple calls from one layer to the next, the prologue and epilogue are executed on each call. Two, if a layer wishes to call a function two layers down, it must go through the intervening layer, incurring the cost of its epilogue and prologue.

Vertical migration is the process of moving function from one layer down to the next layer, in the process removing unnecessary overhead. For example,

if the layer-1 call P_{11} makes two successive calls P_{01} and P_{02} to layer 0, then a single function P'_{11} can be created in layer 0 that incorporates the two calls, and layer 2 may call it directly rather than calling P_{11}. This cuts the two calls into layer 0 down to one, and removes the call into layer 1 altogether. Additional savings are possible if the generality of the separate P_{01} and P_{02} functions is not required, and the implementation can be optimized in their absence.

One downside to vertical migration is that it violates the layering methodology by allowing layer $n + 1$ to call into $n - 1$ directly. Good documentation techniques can help overcome this problem. Another downside is that if generality is optimized away, then enhancements requiring this generality will require additional reimplementation effort. Of course, this is acceptable if the improvement in performance is highly desired.

Stankovic performed experiments showing that improving a system's structure through manual transformation, and then selectively migrating function to improve performance, yielded better overall structure and improved execution time in comparison to the original system.

Stankovic claims that restructuring cannot be automated because the choice of appropriate structure requires human judgment. What he failed to distinguish was the automation of transformations and the choice of what transformations are applied where. Our approach automates the former without sacrificing human control over the latter. On the other hand, the vertical migration transformations, although meaning preserving, cannot be automated in the style presented here. This is because vertical migration eliminates execution of prologue and epilogue code, which cannot be described with the existing PDG substitution rules.

5.7 Program Understanding

Program understanding—also sometimes called reverse engineering—uses techniques such as graphical display of program structure [13], inferring abstractions [44], or assessing modularity [18, 45] to extract program information in a more understandable or reusable form [34].

Improving a programmer's understanding of a system makes its existing structure clearer—and hence better—just by making it better understood [4, 8]. Although learning has limits on clarifying structure—because of the increasing amount of time required for increasingly complex programs—it requires no change to the system, which is advantageous in the short term.

More importantly, a program-understanding tool can help an engineer navigate and understand a system that may need to be restructured. Also, after restructuring it can accelerate the reeducation of programmers about the system's new structure.

At another level, an intelligent tool might be able to automate the choices of transformations (see Section 5.3) or the actual desired structure. The latter will be very difficult because what constitutes "good" structure is difficult to quantify and is dependent on future changes that are often unknown and perhaps not describable to a tool. The Programmer's Apprentice [44], a knowledge-based inferencing tool, generates plans from programming clichés

that allow them to be reused in developing new code. However, the programmer still must choose when use of the plan is appropriate.

Many program-understanding tools, such as slicers for debugging [1, 54] use PDGs or other flow analysis representations. This provides significant leverage for providing several tools at low cost, since they can share complex components.

6. CONCLUSION

Maintenance is the most expensive component of the software process [35]. The structure of a program significantly influences the cost of its maintenance. Restructuring a program can isolate a design decision in a module so that changing it will not require costly nonlocal changes. Our approach to improving the cost-effectiveness of restructuring enables the software engineer to locally specify source-to-source structural changes, while automating the nonlocal, consistent changes that complete the restructuring. This frees the engineer from the implementation and validation tasks associated with the restructuring, accelerating it and preventing the introduction of new errors due to inconsistencies in the changes.

6.1 Critique

To demonstrate the validity of these ideas, we have shown how to build such a tool using a novel semantic model and have used the tool in an experiment and examples. Restructuring, however, is not yet a mature technology.

Transformational Restructuring Is Low-Level. After the software engineer chooses the best structure for the next maintenance step, the transformational approach still requires the engineer to choose the transformations that migrate the program from its existing structure to the new structure. One solution may be to use techniques related to goal-directed program derivation [20] (see Section 5.3), which for restructuring would allow the engineer to specify the goal structure and let the system infer the transformations.

A more straightforward yet higher-level approach would be to integrate analysis tools that use the AST and PDG to compute structural relationships; these can then be visually displayed for manipulation by the engineer. For example, the *uses* relation defined by Parnas can provide information to help define a virtual-machine structure for a system [40]. Such a relation requires significant computation, so it is best computed by a tool. Where a function should be placed based on this information is best left to the engineer; visual display of the *uses* relation may simplify the task.

The Relationship Between Restructuring and Preserving Meaning Is Unclear. The preservation of meaning is a central theme of this restructuring work. Three aspects of this may deserve further attention. First, is preserving meaning needed as an automation technique for restructuring? Balzer's tool described in Section 5.4 leads the engineer to the locations requiring update and performs some manipulations, but does not preserve meaning. This is more flexible than our approach, but less automatic and

provides fewer guarantees. Which approach will engineers prefer? Our experiment, described briefly in Section 3.4, suggests that a guarantee of preserved meaning is valuable, but it is not proved, since the subjects were unaided except for a text editor. Second, if meaning is to be preserved, must it be the *implementation* meaning of the program? For example, an approach based on preserving specification meaning would allow more flexibility. One concern is that specification semantics may not be sufficiently constrained to allow automated assistance. Third, is preserving meaning during restructuring too restrictive to be useful? This reflects on the other two points, but also questions whether global enhancement must be automated to successfully reduce maintenance costs. This calls Parnas' module work into question, but the addition of a tool to automate nonlocal enhancement may qualitatively change the basis of his assumptions.

Need Evidence that Automated Restructuring is Cost-Effective. The experiment in Section 3.4 showed restructuring was haphazard and error prone when performed manually. Although this experiment was on a small program—where no benefit might be seen—the differences were observable. Likewise, the restructurings of matrix multiply (Section 3.3) and the KWIC indexing program (Section 3.5) using the prototype have strengthened the case for tool-aided restructuring. Certainly more thorough experiments are required, but only long-term use will provide sure evidence of the value of tool-aided restructuring. In either case, interactive performance, a windowing interface (now nearly complete), and a more complete and robust set of transformations are required for further progress.

Another approach would be to discover a model that relates the cost of maintenance to assisted restructuring. Such a model might be in the style of that developed by Belady and Lehman [8]. A more precise analysis would require a micromodel of software change.

Restructuring Must Be Generalized to Handle Large Programs. The greatest potential for restructuring lies in managing the structure of large programs. At the scale of thousands or millions of lines of code, the asymptotic term of exponential structure complexity of a program will dominate, with great financial impact.

Modules and classes are two language structures that are useful for building large programs, but are not handled in the current tool. Transformations for restructuring at the module level and in a class hierarchy have been designed [24]. These transformations need to be incorporated in a tool, justified in the model, and their success measured in use.

Performance issues are also crucial. Our substitution rules for incremental update of the PDG promise good performance. Increasing the scale will test this claim. Also important are environment issues such as storage of semantic information between restructuring sessions. Finally, the ability to transform in the presence of interdependences computed with conservative dependence analysis may prove to be a problem. Improved dataflow analysis techniques may be necessary.

More importantly, manual restructuring at a large scale is likely impossible because there are so many nonlocal relationships to keep consistent. The manual bookkeeping promises to be overwhelming, so only an automating tool can flawlessly and tirelessly aid a software engineer in restructuring a program with a guarantee of preserving meaning.

6.2 Contributions

We have shown how to build an interactive, source-to-source, meaning-preserving program-restructuring tool for imperative programs that allows the engineer to locally specify a nonlocal change. The transformations do not just remove gotos; the technique supports a broad class of transformations for localizing design decisions. Further, we have proved the novel idea that program structure can be managed by transforming the abstractions of a program without affecting its basic computations. There are several supporting contributions.

Using Meaning-Preserving Transformation to Automate Restructuring. Automating restructuring is enabled by requiring that structural changes not change the runtime behavior—or even the underlying algorithms and data representations—of the program: preserving meaning is a precise, easy-to-understand, global consistency constraint that precludes introducing errors during restructuring, and still allows changing structure. Using a transformational approach, Section 3 defined a set of structural changes exploiting this constraint. The transformations as a group can manipulate structures spanning several common types.

The transformations are not new [11, 36, 48], but their style of application and purpose are. A transformation is applied to a single syntactic construct, and the tool makes the compensating changes in the rest of the program to preserve its original meaning. This style of transformation removes the engineer from error-prone activities without sacrificing control over the resulting structure, unlike prior restructuring tools [21, 37].

In addition to revealing the hazards of manual restructuring, the experiment also confirmed that the tool automates those exact activities that are error prone: making consistent, physically dispersed changes. It also showed that the tool's style is consistent with observed manual-restructuring techniques, which should improve usability. In particular, the common manual technique of applying copy-paste to a construct, and then editing it and its uses, is supported by the tool. The tool allows the engineer to apply the first part of the action, with the tool completing the editing and compensations to the uses.

Development of a Practical Model for Defining Meaning-Preserving Source-To-Source Transformations. The model introduced in Section 4 defined a small set of local, meaning-preserving subgraph substitution rules and scope manipulation rules. By relating the CFG and PDG to the program source via the mappings of a commutative diagram, the rule set locally describes a transformation's physically distributed textual changes. This

simplified understanding the meaning-preserving properties of program transformations.

The globalization skeleton, derived from the commutative diagram, provides guidance for mapping PDG transformations to program transformations during implementation. Only searches are dynamically mapped between the program and the PDG; all updates to the program and PDG can be performed directly on each. Consequently, a source-to-source transformation is applied directly to the program, leaving unchanged those syntactic features of the program that are not explicitly manipulated. This is in contrast to standard PDG unparse techniques [28, 32].

A Working Implementation of a Restructuring Tool. Successfully implementing a restructuring tool validated the claim that a restructuring transformation can be invoked locally by the engineer and compensated by a tool to preserve meaning. The tool was successfully used on two programs: the matrix multiply used in the experiment and the functional decomposition of KWIC. The implementation also supports the claim that the model is a powerful tool. In particular, the model's abstractions and the globalization skeleton helped define the tool structure and reason about the correctness of its transformations. The implementation also tested incremental update of the PDG in two instances, evidence that a restructuring tool can be efficient.

APPENDIX A

Control Flow Graphs and Program Dependence Graphs

The CFG is a set of vertices that are the primitive operations of a program and a set of directed edges that represent the flow of control between the vertices. The primitive operations, derived from the program's expressions, are called statements, and are triples of the form (*operation*, *result*, *arguments*), where *operation* is either operator call, function call, or a predicate; *result* is a variable to hold the result, and *arguments* are the variables that contain the inputs to the operation. A predicate statement will have two outgoing edges, one with the label **true**, the other **false**, indicating that the respective successor statement is conditionally evaluated based on the success or failure of the predicate. Nonpredicate statements are linked by unlabeled edges, indicating straight-line sequential execution.

A program dependence graph [22, 31] is a set of vertices that represent the primitive operations in the program (i.e., the statements of the CFG) and a set of directed edges that connect the vertices. An edge e representing the flow of data between two operations u and v is called a flow dependence and is denoted $e = FD(u, v)$. There is such an edge if and only if, in the CFG, the result operand of u is an argument to v, and there exists a traversal of the CFG from u to v such that there does not exist a vertex w whose result operand is the same as u's [42]. If the dataflow dependence is due to a variable s being set in u and used in v, the edge is labeled by the variable

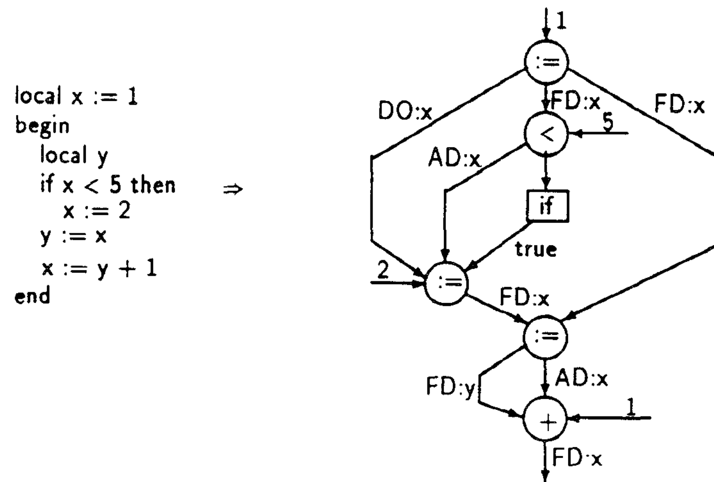

Fig. 9. A program and its program dependence graph.

definition carrying the flow, $e_s = FD(u, v)_s$. By convention, if the variable is not a program variable, it may be omitted from a figure. A control dependence edge represents an on-off switch for its destination vertex. It denotes the success or failure of the predicate vertex at its source. A control dependence edge is labeled **true** or **false** to denote whether the destination vertex is activated on success or failure of the predicate vertex. Thus a true branch of predicate vertex p to v is denoted $CD(p, v)_{\text{true}}$. By definition, in the program only one of the paths denoted by e_{true} or e_{false} can execute on evaluation of the predicate denoted by p.

Some operation vertices generate a constant dataflow dependence. In the figure they are denoted as a constant value on the flow dependence with no source vertex. Another special class of vertices are the input and output vertices, which denote the read and write statements of the program. These are different from other vertices because the value on an output edge is not entirely dependent on the values on the input edge. They are treated conservatively as modifying a common global variable (i.e., the file system).

A PDG may also support some additional edge types: antidependences, def-order dependences, and output dependences. Each, for a different case, implies that the source vertex is necessarily executed before the destination vertex in the program, even though there is no explicit flow of data or control between them. The PDG representation used in the prototype [32] has antidependences (edges denoted by **AD** in Figure 9) and def-order dependences [30] (denoted by **DO**).

ACKNOWLEDGMENTS

We thank James Larus for providing us with the Curare system. Kevin Sullivan, David Garlan, and Harold Ossher helped us with earlier versions of

this paper. Wuu Yang gave us helpful comments on Sequence-Congruence and on the relationship between the PDG and PRG. Kenneth Zadek provided insight on static single-assignment form. Comments by the referees, especially on Section 4, helped improve the paper.

REFERENCES

1. AGRAWAL, H., AND HORGAN, J. R. Dynamic program slicing. In *Proceedings of the SIGPLAN '90 Conference on Programming Languages Design and Implementation* (June 1990). *SIGPLAN Not. 25*, 6 (1990).
2. AHO, A. V., SETHI, R., AND ULLMAN, J. D. *Compilers, Principles, Techniques, and Tools.* Addison-Wesley, Reading, Mass., 1986.
3. ALLEN, F. E., AND COCKE, J. A catalogue of optimizing transformations. In R. Rustin, editor, *Design and Optimization of Compilers.* Prentice-Hall, Englewood Cliffs, NJ, 1972.
4. ARNOLD, R. S. An introduction to software restructuring. In *Tutorial on Software Restructuring.* Society Press (IEEE), Washington, D.C., 1986.
5. BALZER, R. Automated enhancement of knowledge representations. In *Proceedings of the 9th International Joint Conference on Artificial Intelligence* (Aug. 1985), 203–207.
6. BARSTOW, D. On convergence toward a database of program transformations. *ACM Trans. Program. Lang. Syst. 7*, 1 (Jan. 1985), 1–9.
7. BELADY, L. A., AND LEHMAN, M. M. A model of large program development. *IBM Syst. J. 15*, 3 (1976), 225–252.
8. BELADY, L. A., AND LEHMAN, M. M. Programming system dynamics or the metadynamics of systems in maintenance and growth. Res. Rep. RC3546, IBM, 1971. Page citations from reprint in M. M. Lehman, L. A. Belady, Editors, *Program Evolution: Processes of Software Change*, Ch. 5, APIC Studies in Data Processing, No. 27, Academic Press, London, 1985.
9. BOEHM, B. W. *Software Engineering Economics.* Prentice-Hall, Englewood Cliffs, N.J., 1981.
10. BÖHM, C., AND JACOPINI, G. Flow diagrams, Turing machines and languages with only two formation rules. *Commun. ACM 9*, 5 (May 1966), 366–371.
11. BURSTALL, R. M., AND DARLINGTON, J. A transformation system for developing recursive programs. *J. ACM 24*, 1 (Jan. 1977), 44–67.
12. CARTWRIGHT R., AND FELLEISEN, M. The semantics of program dependence. In *Proceedings of the SIGPLAN '89 Conference on Programming Languages Design and Implementation* (July 1989). *SIGPLAN Not. 24*, 7 (1989).
13. CLEVELAND, L. A program understanding support environment. *IBM Syst. J. 28*, 2 (1989).
14. COLLOFELLO, J. S., AND BUCK, J. J. Software quality assurance for maintenance. *IEEE Comput.* (Sept. 1987), 46–51.
15. CYTRON, R., FERRANTE, J., ROSEN, B., WEGMAN, M., AND ZADECK, K. An efficient method of computing static single assignment form. In *Proceedings of the 16th Symposium on Principles of Programming Languages* (Jan. 1988), 25–35.
16. DOWNEY, P. J., AND SETHI, R. Assignment commands with array references. *J. ACM 25*, 4 (Oct. 1978), 652–666.
17. DYBVIG, R. K. *The Scheme Programming Language.* Prentice-Hall, Englewood Cliffs, N.J., 1987.
18. EMBLEY, D. W., AND WOODFIELD, S. N. Assessing the quality of abstract data types written in Ada. In *Proceedings of the 10th International Conference on Software Engineering* (Apr. 1988), 144–153.
19. FEATHER, M. S. Specification evolution and program (re)transformation. In *Proceedings of the 5th RADC Knowledge-Based Software Assistant Conference* (Sept. 1990).
20. FEATHER, M. S. A system for assisting program transformation. *ACM Trans. Program. Lang. Syst. 4*, 1 (Jan. 1984), 1–20.
21. FEDERAL SOFTWARE MANAGEMENT SUPPORT CENTER. Parallel test and productivity evaluation of a commercially supplied COBOL restructuring tool. Tech. Rep., Office of Software Development and Information Technology, Washington, D.C., 1987.

22. FERRANTE, J., OTTENSTEIN, K. J., AND WARREN, J. D. The program dependence graph and its use in optimization. *ACM Trans Program. Lang. Syst. 9*, 3 (July 1987), 319–349.
23. GALLAGHER, K. B., AND LYLE, J. R. Using program slicing in software maintenance. *IEEE Trans. Softw. Eng. 17*, 8 (Aug. 1991), 751–761.
24 GRISWOLD, W G. Program restructuring to aid software maintenance. Ph.D. dissertation, Univ. of Washington, Dept. of Computer Science and Engineering, Seattle, Wash., 1991. Tech. Rep. No. 91-08-04.
25. GRISWOLD, W. G., AND NOTKIN, D Computer-aided vs. manual program restructuring. *ACM SIGSOFT Softw. Eng. Notes 17*, 1 (Jan. 1992).
26. GRISWOLD, W. G., AND NOTKIN, D. Semantic manipulation of program source. Tech. Rep 91-08-03, Univ. of Washington, Dept of Computer Science and Engineering, Seattle, Wash., 1991.
27. HOARE, C. A. R., HAYES, I. J., JIFENG, H., MORGAN, C. C., ROSCOE, A. W., SANDERS, J. W., SORENSEN, I. H., SPIVEY, J. M., AND SUFRIN, B. A. Laws of programming. *Commun ACM 30*, 2 (Aug. 1987), 672–686.
28 HORWITZ, S , REPS, T., AND BINKLEY, D. Interprocedural slicing using dependence graphs. *ACM Trans. Program Lang Syst. 12*, 3 (Jan. 1990), 26–60.
29. HORWITZ, S., PRINS, J., AND REPS, T. Integrating noninterfering versions of programs. *ACM Trans. Program. Lang. Syst.* (July 1989), 345–387.
30. HORWITZ, S., PRINS, J , AND REPS, T. On the adequacy of program dependence graphs for representing programs. In *Proceedings of the 15th Symposium on Principles of Programming Languages* (Jan 1988), 146–157.
31. KUCK, D. J., KUHN, R. H., LEASURE, B., PADUA, D. A., AND WOLFE, M. Dependence graphs and compiler optimizations. In *Proceedings of the 8th Symposium on Principles of Programming Languages* (Jan. 1981), 207–218.
32 LARUS, J R. Restructuring symbolic programs for concurrent execution on multiprocessors. Ph.D. dissertation, UC Berkeley Computer Science, 1989. Also Tech. Rep No. UCB/CSD89/502.
33. LEHMAN, M. M. On understanding laws, evolution and conservation in the large-program life cycle. *J. Syst. Softw. 1*, 3 (1980). Page citations from reprint in *Program Evolution: Processes of Software Change*. APIC Studies in Data Processing, No. 27, Academic Press, London, 1985.
34 LEWIS, T. *IEEE Computer* (Jan. 1990). Special Issue on Software Engineering.
35. LIENTZ, B., AND SWANSON, E. *Software Maintenance Management: A Study of the Maintenance of Computer Application Software in 487 Data Processing Organizations*. Addison-Wesley, Reading, Mass., 1980.
36. LOVEMAN, D. B. Program improvement by source-to-source transformation *J. ACM 24*, 1 (Jan. 1977), 121–145.
37. MORGAN, H. W. Evolution of a software maintenance tool. In *Proceedings of the 2nd National Conference EDP Software Maintenance* (1984), 268–278.
38. NARAYANASWAMY, K., AND COHEN, D. An assessment of the AP5 programming language—theory and experience. Tech Rep., Information Sciences Inst., Univ. of Southern California, Los Angeles, 1991.
39. OSSHER, H. L. A mechanism for specifying the structure of large, layered programs In *Research Directions in Object-Oriented Programming*. MIT Press, Cambridge, Mass., 1987
40. PARNAS, D. L. Designing software for ease of extension and contraction. *IEEE Trans. Softw Eng. SE-5*, 2 (Mar. 1979), 128–138.
41 PARNAS, D. L. On the criteria to be used in decomposing systems into modules. *Commun. ACM 15*, 12 (Dec. 1972), 1053–1058.
42. PODGURSKI, A., AND CLARKE, L. A. A formal model of program dependences and its implications for software testing, debugging, and maintenance. *IEEE Trans. Softw. Eng. SE-16*, 9 (1990), 965–979
43. RAMSHAW, L. Eliminating go to's while preserving program structure. *J. ACM 35*, 4 (Oct. 1988), 893–920.
44. RICH, C., AND WATERS, R. C. The programmer's apprentice: A research overview. *IEEE Comput.* (Nov. 1988), 11–25.

45. SCHWANKE, R. W. An intelligent tool for re-engineering software modularity. In *Proceedings of the 13th International Conference on Software Engineering* (May 1991), 83–92.
46. SELKE, R. P. Transforming program dependence graphs. Tech. Rep. TR90-131, Dept. of Computer Science, Rice Univ., Houston, Tex., 1990.
47. SHIVERS, O. Control flow analysis in scheme. In *Proceedings of the SIGPLAN '88 Conference on Programming Languages Design and Implementation* (July 1988). *SIGPLAN Not. 23*, 7.
48. STANKOVIC, J. Good system structure features: Their complexity and execution time cost. *IEEE Trans. Softw. Eng. SE-8*, 4 (July 1982), 306–318.
49. STANKOVIC, J. A. Structured systems and their performance improvement through vertical migration. Ph.D. dissertation, Brown Univ., 1979. Dept. of Computer Science Tech. Rep. CS-41.
50. STEVENS, W., MYERS, G., AND CONSTANTINE, L. Structured design. *IBM Syst. J. 13*, 2 (1974).
51. SULLIVAN, K., AND NOTKIN, D. Reconciling environment integration and component independence. *ACM Trans Softw. Eng. Method. 1*, 3 (July 1992), 229–268.
52. SULLIVAN, K. J., AND NOTKIN, D. Reconciling environment integration and component independence. In *Proceedings of the SIGSOFT '90 4th Symposium on Software Development Environments* (Dec. 1990).
53. VENKATESH, G. A. The semantic approach to program slicing. In *Proceedings of the SIGPLAN '91 Conference on Programming Languages Design and Implementation* (June 1991). *SIGPLAN Not. 26*, 6.
54. WEISER, M. Program slicing. *IEEE Trans. Softw. Eng. SE-10*, 4 (July 1984), 352–357.
55. WILLIAMS, M. H., AND OSSHER, H. L. Conversion of unstructured flow diagrams to structured form. *Comput. J. 21*, 2 (1977), 161–167.
56. YANG, W. A new algorithm for semantics-based program integration. Ph.D. dissertation, Univ. of Wisconsin, 1990. Computer Sciences Tech. Rep. No. 962.
57. YANG, W., HORWITZ, S., AND REPS, T. Detecting program components with equivalent behaviors. Tech. Rep. 840, Computer Sciences Dept., Univ. of Wisconsin, Madison, 1989.

Received December 1992; revised November 1992 and January 1993; accepted January 1993