DIME: A direct manipulation environment for evolutionary development of software

Arun Lakhotia
The Center for Advanced Computer Studies
The University of Southwestern Louisiana
Lafayette, LA 70506, USA
+1 318 482–6766
arun@cacs.usl.edu

Abstract

*This paper presents an overview of the DIME environment (***DI***rect* **M***anipulation* **E***nvironment) being developed by the author. The paper presents the DIME vision, its catalogue of evolutionary transformations—program transformations used by programmers during software maintenance—and scenarios of how they may be used by a programmer during software maintenance. The DIME system will provide for programmers what programmers provide for other computer users: a simple, intuitive, yet powerful way to transform data with the click of a mouse. It will place at the programmer's finger-tips—figuratively speaking—rigorous, formal transformations for creating, composing, analyzing, and modifying the architecture of a software system. Using DIME a programmer will radically overhaul the architecture of a software system just by point-and-click and drag-and-drop with the guarantee that the external behavior of the system is unchanged.*

1 Introduction

Software is different from other engineering artifacts in that a *new software system is created by modifying actual components* of the old systems. Thus, while the mechanical, electrical, and electronic components of a new submarine are created from totally new material—rather than from scavenging similar components from an old submarine—its software components most likely are scavenged from an old submarine.

The DIME environment (**DI**rect **M**anipulation **E**nvironment), currently under development, will provide capability to modify software using a mouse-based, direct manipulation, user interface. The DIME environment will place at the programmers' finger-tips—figuratively speaking—rigorous, formal transformations for creating, composing, analyzing, and modifying the architecture of a software system. Using DIME a programmer will be able to radically overhaul the architecture of a software system just by point-and-click and drag-and-drop with the guarantee that the external behavior of the system will remain unchanged.

The theoretical foundation of DIME consists of a catalogue of *evolutionary transformations*—transformations performed by programmers during software evolution. This catalogue of transformation, compiled by analyzing the author's journal containing his record of programming activities over a span of three years, is quite similar to the catalogue of restructuring transformations developed earlier by Griswold [12, 14]. The innovation of the DIME project lies in attaching the evolutionary transformations to mouse-clicks. This calls for the development of additional formalism to infer parameters of the transformations from both the context in which a transformation is applied and some minimal hints provided by the user.

The rest of this paper is organized as follows. Section 2 introduces the DIME vision. Section 3 presents our catalogue of evolutionary transformations. Section 4 highlights issues in developing the direct manipulation interface. A comparison of DIME with other research efforts is presented in Section 5. Our concluding remarks, which include the status of the DIME project, its expected impact, and some anticipated challenges are presented in Section 6.

2 Vision

The DIME environment will revolutionize the way programmers develop and manipulate programs. By integrating the robustness of formal methods with the ease of use of Internet web browsers, the Macintosh, and the Windows95 desktop, DIME will place sophisticated program transformations at the programmers' finger-tips, literally available at the click of a mouse.

The desktops of Macintosh and Windows95 provide (a) iconic representation of files and directories, (b) point-and-click interface for moving, copying, deleting, renaming, and printing files and directories, (c) multiple views to present the contents of a directory, (d) point-and-click to navigate through directory, and (e) mechanisms to search for files and directories based on various parameters.

DIME will provide similar point-and-click access to the components of a software system. The directory, source

code files, functions, variables, statements, types, makefiles, etc.—the components of a software system—will be represented as icons. A program component or a relation between components may be presented in different "views." For instance, a function may either be shown as an icon or be shown by its name along with some summary information, such as its size, date of last change, etc. It could also be shown as its complete code. A call relation may be shown as a directed graph or as a nested tree structure.

A programmer will perform commonly used complex operations by just pointing-and-clicking. For instance, renaming a function will be akin to renaming a file. Below the icon of a function will be its name. To rename the function one will click on its name and type in the new name. The environment will automatically rename the function in all calls to the same function, based on scope rules.

Similarly, other mouse-based operations—such as select, cut/paste, drag/drop—will lead to meaningful manipulations of programs. One could extract a reusable piece of code hidden within a big function by selecting the relevant code fragment, dragging it, and dropping it on a file. The system will automatically identify the parameters to the new function and introduce a call to the new function. Similarly, one could make a global variable into a local variable by dragging the variable and dropping it into the declaration section of a function.

Just as web browsers provide mechanisms to move through documents by following links and traversal history for navigation through the web, so will DIME provide mechanisms for navigation through a software system. Using architecture modelling technologies [16, 20], DIME will present to the programmer an abstract view of the system. The programmer will navigate through the system by clicking on objects and relationships. Thus, a programmer may navigate through a program following task interaction paths, or function call paths, or data flow paths, to name a few.

The integration of the coarse-grain architecture analysis techniques and fine-grain formal methods will arm the programmer with a very powerful mechanism to radically overhaul the architecture of a software system. For instance, a programmer may first use the coarse-grain analysis to identify tightly-coupled components [13] and then use fine-grain analysis to de-couple those components. Similarly, coarse-grain analysis may be used to identify repeated patterns of code [3] which may be replaced by appropriate function calls using fine-grain analysis.

3 Evolutionary transformations

The main theoretical challenge of this project is the development of a catalogue of *transformations for evolving the structure* of a program. These are transformations used by programmers in their day-to-day activity to reorganize, restructure, or re-architect source code without influencing the external behavior of the program. Such transformations may be interspersed with other behavior modifying changes and are performed to alter the design of a system. These transformations are also used when overhauling the architecture of a software system.

In the rest of this section we summarize our catalogue of transformations for structural evolution. The transformations in this catalogue were identified by analyzing about 40,000 lines journal of the author's programming activities over the past three years. The programming activities recorded were undertaken in the development of three systems, most notably the WolfPack*. That our transformations are representative of operations actually performed by programmers is supported by the fact that, in principle, they are the similar to the restructuring transformations presented earlier by Griswold [12, 14]. We use the word *transformation* broadly as a "mathematical mapping." While most of the transformations actually modify a program's structure, some of them—in particular, wedge—only provides some insight into a program's structure without modifying it.

Transformation: *Fold*. The fold transformation creates a function for a given set of statements and replaces the statements by a call to this function.

The fold transformation, also sometimes called *lambda lifting*, was first developed by Burstall and Darlington in the context of functional programming [6] and subsequently studied for logic programs [30]. Griswold's `extract-function` transformation extends it to structured—single-entry, single-exit—imperative programs. We have extended this transformation to arbitrary imperative programs [21].

Once a sequence of statements that can be folded has been identified, folding them into a function poses some other challenges. One must not only decide which variables would be parameters to the functions and which variables would be its local variables—equivalent to determining which variables are universally quantified and which are existentially quantified—but also determine whether a parameter is passed by value or by reference.

Transformation: *Unfold*. The unfold transformation replaces a function call by the statements in the body of the called function.

This transformation is analogous to the unfold transformation in functional and logic programs [6, 30], with the difference that in the imperative domain this transfor-

* Visit `http://www.cacs.usl.edu/~arun/Wolf/`

```
1  Procedure Sale_Pay_Profit (days: integer;
                   cost: float; var sale: int_array;
                   var pay:  float; var profit: float;
                   process: boolean);
2   var i: integer;total_sale, total_pay: float;
3   begin
4     i:=0;
5     while i < days do begin
6          i := i + 1;
7          readln(sale[i])
8     end;
9     if process = True then begin
10      total_sale:=0;
11      total_pay:=0;
12      for i := 1 to days do begin
13          total_sale := total_sale + sale[i];
14          total_pay := total_pay + 0.1 * sale[i];
15          if sale[i] > 1000 then
16              total_pay := total_pay + 50;
17       end;
18       pay := total_pay / days + 100;
19       profit := 0.9 * total_sale - cost;
20    end;
21  end;
```

Figure **1** **Sample non-cohesive code.** This function uses the same input to compute different outputs. Its computation also depend on a flag passed as a parameter. This function id a representative of code with *interleaved* computations [27]. In the following figures this function is restructured into a collection of functions with an object-based architecture. Functions, representing methods, computing different outputs are extracted using meaning preserving transformations: Wedge, Split, and Fold.

mation must account for the various types of parameter-passing conventions.

Transformation: *Inline*. The inline transformation unfolds a function at all the places from where it is called.

Griswold's `var-to-expr` and `binding-to-expr` transformations correspond to our inline and unfold transformations, respectively [14].

Transformation: *Split*. The split transformation splits a single-entry, single-exit region into two regions, one containing all the computations relevant to a set of statements S and the other containing all the remaining computations. The transformation introduces new variables or renames variables and composes the two new regions such that the overall computation remains unchanged. When it is not feasible to split a region in such a way, the transformation leaves the region unchanged.

The split and fold transformations provide a method for extracting interleaved computations [27] into separate functions without changing the external behavior of the system [21]. This transformation uses program slicing [31] to identify computations that are related.

Transformation: *Move*. The move transformation moves a statement over the statements before or after it in the control flow, but within the same function.

The move transformation is analogous to Griswold's `move-expr` transformation. In optimizing compilers, a move transformation is used to move invariant computation out of a loop and to reorder computations so as to reduce the need for temporary variables [1, 23].

Transformation: *PushUp*. The pushup transformation pushes a statement from a function to its call-sites. The statement may be moved either to the point before the function is called, or the point after the completion of the call.

To ensure that the external behavior remains unchanged, the parameters of the affected function may have to be changed, such as when a computation involving a local variable is pushed up.

Transformation: *PushDown*. The pushdown transformation pushes a statement from the call-site into the body of a called function. In so doing, all the sites from where this function is called may also be affected.

Just like the pushup transformation, this transformation may influence the parameters of the called function in order to ensure that the external behavior of the system remains unchanged.

Transformation: *Rename*. The rename transformation changes the name of a program component, such as a function, a variable, a type, or a file. It changes the name at the definition of that component and also at all the places where that component is referenced.

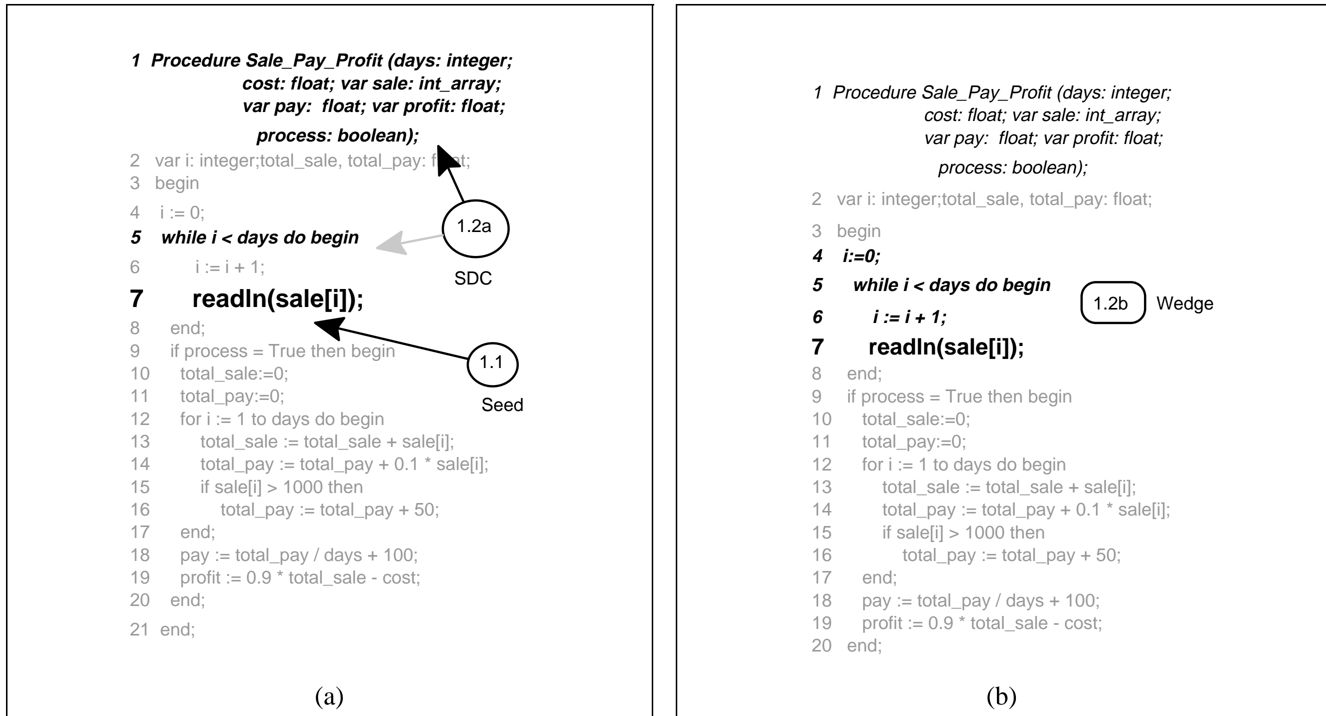

Figure 2 **Selection of code to be extracted into a function.** (1.1) The user selects the seed. (1.2a) The system highlights the two SDCs of the seed. Each SDC defines a single-entry, single-exit region in which to bound the slice. The user selects one SDC. (1.2b) The system identifies the statements influencing the seed within the region defined by the SDC. In this step, the user selects the *readln* statement as the seed with the intent to separate the user interface from the computation.

The rename transformation is analogous to Griswold's `rename-variable` transformation [14].

Transformation: *Reorder*. The reorder transformation changes the order of the parameters of a function. The reordering is applied to the function definition and the call-sites.

Transformation: *Rescope*. The rescope transformation changes the scope of a variable, for instance, it changes a global variable into a parameter. The rescope transformation may influence the parameters of a function both when a global variable is converted into a parameter or a local variable; or when a parameter or a local variable is converted to a global variable.

The reorder and rescope transformation in Griswold's catalogue may be performed by using his `move-expr` transformation. His catalogue does not have explicit transformation for this operation.

Transformation: *Group*. The group transformation collects a set of variables into a structure (record).

The group transformation would be most valuable in migrating FORTRAN IV programs to FORTRAN 9X or to other procedural languages. The transformation would also be valuable for re-architecting C and C++ programs, especially those written by domain-experts who are not trained computer scientists. The group transformation is analogous to the `vectorfy-bindings` and `listify-bindings` transformations of Griswold [14].

Transformation: *Ungroup*. The ungroup transformation decomposes a structure and creates a variable for each of its fields. The ungroup transformation may be used to decompose a structure when it does not represent a cohesive grouping.

4 Accessing evolutionary transformations using the mouse

The DIME project's major innovation is in providing access to rigorous, formal transformations at the click of a mouse. Thus, an important component of the project is how to associate the transformations with mouse-clicks.

The mouse-based operations of Macintosh and Windows95 desktops may be classified into two categories as follows:

- **Selection:** Select an object; add to selection; unselect an object
- **Action:** Move, Cut, Paste, Drag-Drop, User-defined operation

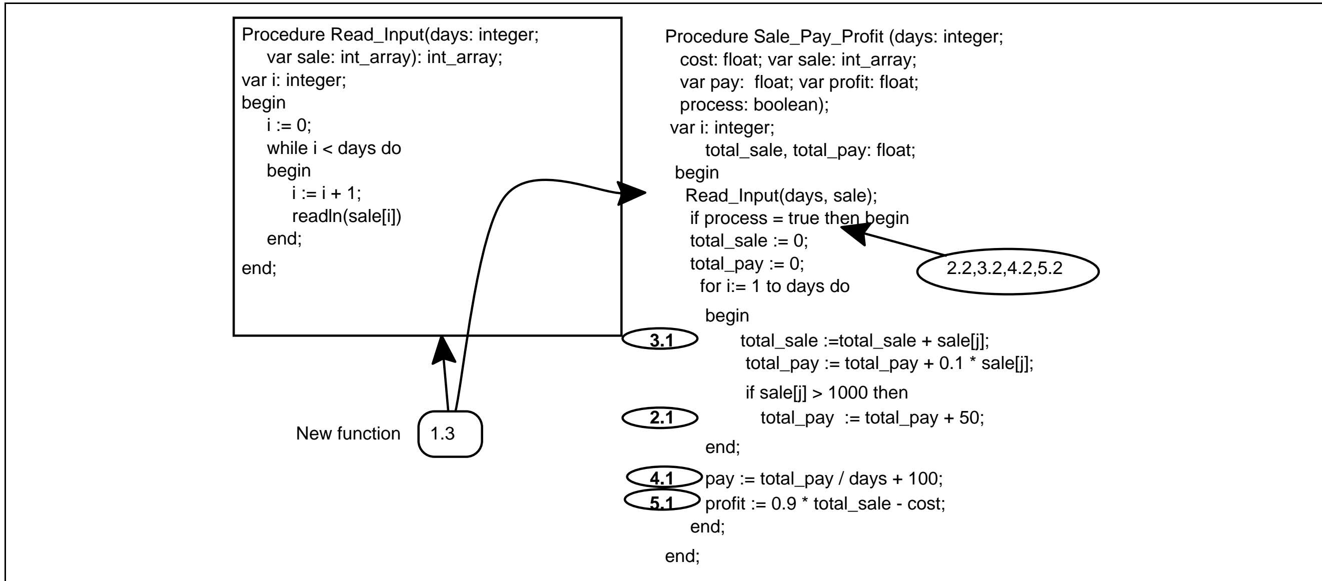

Figure **3** **Completion of function extraction, and input for subsequent steps** (1.3) The code selected in Figure 2 is extracted and converted into a function. The selected code is replaced by a call to this function. Since the selected code did not interleave with any other code the decision about where to place the call was straightforward. User selected for steps 2, 3, 4, and 5 and the SDC selected for these seeds are shown. The next figure contains the result of these steps.

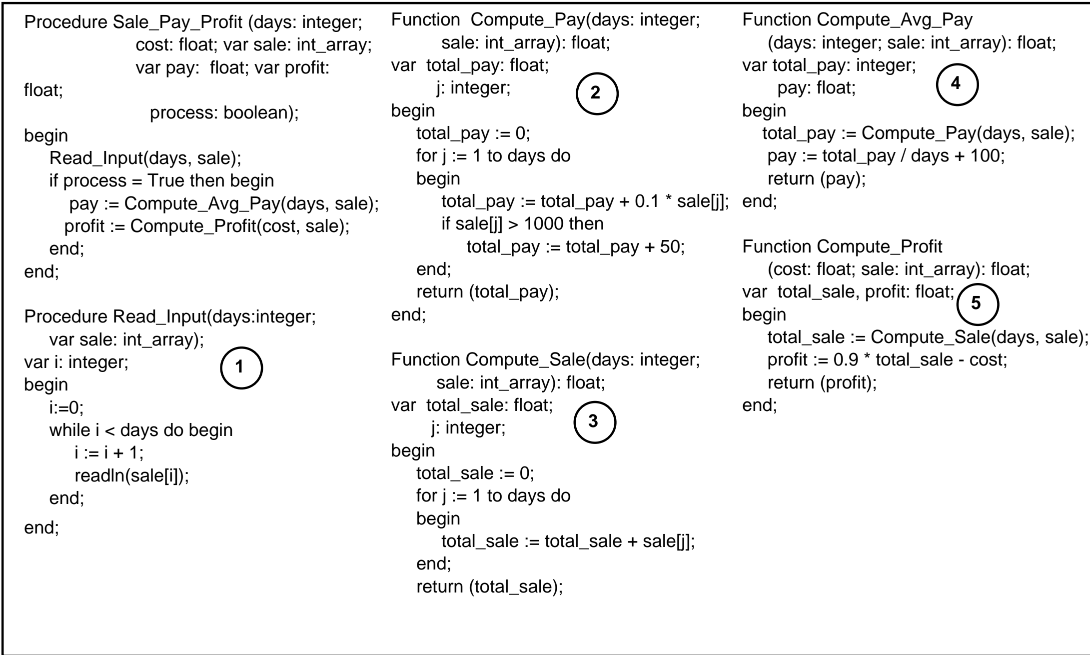

Figure **4** **Final result of restructuring program in Figure 1.** The annotations **1** to **5** indicate the restructuring steps, with respect to previous figures, in which the function was created. To create functions in steps **2** and **3** required separating interleaved computations. This was achieved by duplicating some code segment.

Both types of operations are performed using either mouse-clicks or menu selection. The action operations operate on objects which are identified using the objects in the *current* selection.

The need to access transformations using the mouse introduces a need for additional formalism that is not needed in other uses of formal transformations. To be easy to use, DIME should not require the user to specify all the pa-

rameters to a transformation completely. Instead, it should infer the parameters based on some *hints* or *seeds* provided by the user. Additional formalism is needed to identify these parameters automatically. The next transformation, an example of such formalism, has been designed explicitly for the purpose of identifying parameters for split and fold [21].

Transformation: *Wedge*. The wedge transformation bounds the slice in a single-entry, single-exit region called a *single definite control* (SDC) (See [8, 21] for details).

Figures 1 through 4 show the above transformations interactively extracting interleaved computations. The original function, given in Figure 1, has a very low cohesion [28]. In subsequent steps this function is restructured using a sequence of Wedge, Split, and Fold transformations. At each step, the computations related to a set of seed statements are extracted and converted into a function. The final program has an *object-based* architecture in that the computation for each data is hidden in a single function.

A restructuring step in Figures 1 through 4 consists of the following activities:

1. The user selects a set of seed statements.
2. With the help of the system, the user selects the computation to be extracted in a function:
   a. The system highlights the SDCs of the seed statements. The user picks an SDC.
   b. The system selects the computation to be extracted by applying the wedge transformation on the seed and the SDC.
3. The system replaces the selected code by a call to this new function.

The seed statements need not be contiguous code. If the selected code is interleaved with any other code, the system may have to duplicate some code. In such a case, the system generates the parameters of the new function and places the function call such that external behavior of the function is not changed due to the duplication of code. If the system cannot guarantee that the external behavior will remain unchanged, the system does not create the new function call.

Figures 2 (a) and (b) show the details of performing the above activities once. The user selects the *readln* statement as the seed and the procedure entry as the SDC. The result of the wedge transformation is shown in Figure 2(b). This example has been taken from Deprez [8]. Details about the intermediate steps and the formal definition of the transformations may be found in his thesis.

5 Related works

There has been significant amount of work in the use of program transformations for the development of program from specifications [10, 24], for evolution of specifications [11, 18, 19], and for specification-directed evolution of programs [9, 22] . All these works require the specification of a program to be explicitly represented, usually in some formal language. The transformations we propose may be classified as *structural* evolution transformation. Our transformations operate directly on the program. Since these transformations do not modify the behavior of a program, they treat the program as its own specification. The focus of the structural transformation is the structure of a program.

Structural evolution transformations have previously been studied by Griswold and Notkin [12, 14], Bull [5], Datta [7], and the REDO project [4]. Griswold and Notkin studied these transformations for Scheme, an imperative, yet structured, language. Bull, Datta, and the REDO project have developed general frameworks to describe such transformations. These frameworks are quite similar to Software Refinery (now called Reasoning5 Code Base Management System, CBMS [25][†]). They provide capability for expressing reengineering transformations.

Jain has formalized a method of constructing complex logic programs by annotating the control flow of simpler programs [17]. He proposes a software environment for constructing and maintaining logic programs using a catalogue of simple programs and a list of "behavior preserving" transformations. These transformations preserve *how* a program computes but alter *what* it computes. In contrast, we preserve *what* a program computes but may alter *how* it is computed.

Since we operate on the structure of a program, our work is related to research in syntax-directed editors [26, 29]. The structural operations provided by such editors, while aware of a program's structure, are not sensitive to its semantics. Therein lies the difference. Even though our transformations are oriented towards a program's structure, they are actually *semantics directed* because they are aware of the semantics of the program's structure.

Interest in syntax-directed editors has resurfaced due to the WWW phenomenon. The Netscape Composer editor is an example of a syntax-directed editor. Programming environments, such as Visual Basic, Visual C++, the Java Development Kit, and Symantec Cafe, also provide rudimentary syntax-directed editing using the mouse. Though these tools give an appearance of drag-and-drop editing, similar to what we propose, the operations they provide do not have any semantic content.

Simonyi's effort in Intentional Programming (IP) at the Microsoft Research is directed towards developing a new paradigm of programming [2]. Instead of encoding actions in the rigid syntax of a programming languages, as one

[†] Visit `http://www.reasoning.com`

currently does, a programmer will *encode intentions* in a syntax-free structure. The structure is syntax-free in that it may not correspond to any external ASCII representation. A program in the IP paradigm is a hierarchy of intentions. The higher-level intentions are defined in terms of other low level intentions. The lowest level of intentions map to the notion of statements in traditional languages. In the IP paradigm, since the intentions are explicitly modelled, they also provide a trace of a programmer's design decisions. To move existing code to the IP paradigm will require identifying and encoding its hierarchy of intentions, a task that would require significant effort.

Our structural evolution transformations correspond to transforming intentions in the IP paradigm. Our evolutionary model may therefore provide a bridge between the current paradigm and the IP paradigm. Using our transformations legacy code may be moved to the IP paradigm incrementally.

6 Conclusions

The DIME environment will aid in reengineering the design of existing systems and also in extracting reusable components from existing systems. DIME's architectural transformations will enable programmers to *surgically operate* on legacy systems and reengineer them into object-oriented architectures. For example, using DIME's drag-and-drop transformations a programmer would first separate the kernel of a software system from its (user) interface. Then using similar transformations she may collect the code segments operating on the same data structures. Then the programmer may throw away the old user interface and package the kernel as a reusable component.

The DIME environment is currently under development. Besides the work presented in this paper, we have so far developed the algorithms necessary for performing some of the transformations [8, 21].

There are several challenges in the path of executing the vision presented in this paper. The most formidable challenge is in developing algorithms for performing control and data flow analysis of programs in reasonable time and with reasonable precision. This is further made difficult by the need to incrementally update the analysis as the program is modified. Faced with these issues, Griswold and Notkin developed a formalism for incrementally updating the analyses [14] and developed an architectural design that reduced the cost of keeping the various analyses consistent [15]. Yet, they have concluded that the analysis cost for a large system is prohibitively expensive. They are now developing a tool that help a programmer in *planning* the restructuring task, but that does not perform the restructuring itself [13].

The DIME project is investigating a compromise between providing no automated support and fully automated support. In the approach being pursued, we split the test for the feasibility of a transformation—that the transformation will not change meaning—into two parts: local and global. The system will actually verify whether a transformation is feasible using local information—typically information within a procedure or a function. It will not verify, but only develop the constraints that should be satisfied globally, i.e., using information outside the function, in order for the transformation to be feasible. The unsolved, but may be simplified, global constraints will be presented to the programmer to verify.

7 Acknowledgments

The idea of using a direct manipulation user interface for program restructuring was triggered by Bruce Lewis of Army MICOM. The author thanks Jean-Christophe Deprez and Sharat Jenigiri for their contribution in the development of the concepts presented. This work was partially supported by a contract from the Department of Defense and a grant from the Department of Army, US Army Research Office. The contents of the paper do not necessarily reflect the position or the policy of the funding agencies, and no official endorsement should be inferred.

8 References

[1] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 1986.

[2] W. Aitken, B. Dickens, P. Kwiatkowski, O. de Moor, D. Richter, and C. Simonyi. Transformation in intentional programming. http://www.research.microsoft.com/ip/-overview/TrafoInIP.ps, Sept. 1997.

[3] B. S. Baker. On finding duplication and near-duplication in large software systems. In *Proceedings of the Second Working Conference on Reverse Engineering (WCRE'95), Toronto*, pages 86–95, Los Alamitos, CA, July 1995. IEEE Computer Society Press.

[4] J. Bowen, P. Breuer, and K. Lano. A compendium of formal techniques for software maintenance. *IEE/BCS Software Engineering Journal*, 8(5), September 1993.

[5] T. Bull. *Software maintenance by program transformation in a wide spectrum language*. PhD thesis, School of Engineering and Computer Science, University of Durham, Durham, UK, 1994.

[6] R. M. Burstall and J. Darlington. A transformation system for developing recursive programs. *J. ACM*, 24(1):44–67, Jan. 1977.

[7] A. Datta. *Automated Adaptation of Programs*. PhD thesis, Wright State University, 1992.

[8] J.-C. Deprez. A context-sensitive formal transformation for restructuring programs. Master's thesis, The Center for

Advanced Computer Studies, University of Southwestern Louisiana, Lafayette, Louisiana, Dec. 1997.

[9] N. Dershowitz and Z. Manna. The evolution of programs: Automatic program modification. *IEEE Trans. Softw. Eng.*, 3(6):377–385, Nov. 1977.

[10] M. S. Feather. A survey and classification of some program transformation approaches and techniques. In L. G. L. T. Meertens, editor, *Program Specification and Transformation*, pages 165–195. North-Holland, 1987.

[11] M. S. Feather. Detecting interference when merging specification evolutions. In *Proceedings, 5th International Workshop on Software Specification and Design, Pittsburgh, Pennsylvania*, pages 169–176. Computer Society Press of the IEEE, 1989.

[12] W. G. Griswold. *Program Restructuring as an Aid to Software Maintenance*. PhD thesis, University of Washington, July 1991.

[13] W. G. Griswold, M. I. Chen, R. W. Bowdidge, and J. D. Morgenthaler. Tool support for planning the restructuring of data abstraction in large systems. In *Proceedings of the ACM SIGSOFT'96 Symposium on the Foundations of Software Engineering (FSE-4), San Francisco, CA*, pages 33–45, Oct. 1996.

[14] W. G. Griswold and D. Notkin. Automated assistance for program restructuring. *ACM Transactions on Software Engineering*, 2(3):228–269, July 1993.

[15] W. G. Griswold and D. Notkin. Architectural tradeoffs for a meaning-preserving program restructuring tool. *IEEE Trans. Softw. Eng.*, 21(4):275–287, Apr. 1995.

[16] D. R. Harris, A. S. Yeh, and H. R. Reubenstein. Extracting architectural features from source code. *Automated Software Engineering*, 3:109–138, 1996.

[17] A. Jain. *Program Maps for Relating Structurally Enhanced Logic Programs*. PhD thesis, Case Western Reserver University, Department of Computer Engineering and Science, 1995.

[18] L. Johnson and M. S. Feather. Building an evolution transformation library. In *Proceedings of 12th International Conference on Software Engineering*, pages 238–248, 1990.

[19] W. L. Johnson and M. S. Feather. Using evolution transformations to construct specifications. In *Automating Software Design*, pages 65–92. AAAI Press, 1991.

[20] A. Lakhotia. A unified framework for software subsystem classification techniques. *Journal of Systems and Software*, 36:211–231, Mar. 1997.

[21] A. Lakhotia and J.-C. Deprez. Restructuring programs by tucking statements into functions. *Journal of Information and Software technology*, page to appear, 1999.

[22] R. Mili, M. Frappier, J. Desharnais, and A. Mili. A calculus of program modifications. *ACM Software Engineering Notes*, 22(3):157–168, May 1997.

[23] S. S. Muchnick. *Advanced Compiler Design & Implementation*. Morgan Kaufmann Publishers, San Francisco, CA., 1997.

[24] H. Partsch and R. Steinbruggen. Program transformation systems. *Computing Surveys*, 15(3):199–236, 1983.

[25] Reasoning Systems, Inc., Palo Alto, CA. *Refine User's Guide*, 1992.

[26] T. Reps. *Generating Language Based Environments*. MIT Press, 1983.

[27] S. Rugaber, K. Stirewalt, and L. Wills. Understanding interleaved code. *Automated Software Engineering*, 3(1-2):47–76, June 1996.

[28] W. P. Stevens, G. J. Myers, and L. L. Constantine. Structured design. *IBM Systems Journal*, 13(2):115–139, 1974.

[29] G. Szwillus and L. Neal. *Structure-based editors and environments*. Academic Press, 1996.

[30] H. Tamaki and T. Sato. Unfold/fold transformations of logic programs. In *Proceedings of Second International Conference on Logic Programming, (Sweden)*, pages 127–138, 1984.

[31] F. Tip. A survey of program slicing techniques. *J. Program. Lang.*, 3:121–181, 1995.