Revision Recognition for Scientific Computing: Theory and Application

Andrew Strelzoff
University of California Santa Barbara
Department of Computer Science
Santa Barbara, California 93106
strelz@engineering.ucsb.edu

Linda Petzold
University of California Santa Barbara
Department of Computer Science
Santa Barbara, California 93106
petzold@engineering.ucsb.edu

Abstract

Scientific computing applications are rarely equipped with high quality graphic user interfaces (GUIs), in part because the interfaces cannot be revised quickly enough to keep pace with rapid changes in the underlying application. A critical step in user interface maintenance is revision recognition: the process of determining, given two versions of a program, which parts of the original program have been changed, deleted or preserved. For large, highly complex scientific applications this problem is a significant barrier to interface maintenance. In this paper we present the "reticent programmer model" which transforms the revision recognition problem into an optimization problem. The solutions obtained using this model are exact for small applications and are high quality estimates for medium to large applications. An example of how these solutions can be used to facilitate user interface maintenance is also presented.

Keywords

Revision recognition, graph mapping, software evolution, scientific computing.

1. Introduction

Maintaining user interfaces for scientific computing is a long standing problem. Fundamentally, the user interface programmer rarely has the mathematical background to understand the complex changes that take place in numeric software packages as they are revised. Developers of numeric software are usually unwilling to invest the time needed to become expert in interface programming, an area of programming very far from their basic expertise. The general assumption is that individuals who can bridge the gap are very rare. Thus research has concentrated upon **communication** between the **developers** of numeric software on the one hand, and interface **programmers** on the other.

Research into this problem is divided between assistance for the interface programmer in the form of tool-kits [3, 5] and visual environments [9, 5], and communication assistance (enforcement) in the form of tags developers place in their code [3] or software wrappers [7] developers build around their code. These tags or wrappers can then be scanned to provide the information needed to reformulate a user interface for a revised application. The problem with this approach is that it places a considerable burden upon the developer to wrap and re-wrap the underlying software with each revision. In our research we seek to determine what assistance can be provided to the interface programmer with only minimal participation of the numeric software developer.

The interface programmer's first goal when updating a user interface is to identify which code segments in the underlying application have remained largely unchanged. The supporting interface code for these sections can then be reused with only minor modification. The remainder of the code in the new version must then either be new, or changed and reorganized so radically that it would not be useful to try to preserve the interface code relating to these sections. We term this process **revision recognition**. For small codes, revision recognition can be solved by inspection. However, for large, complicated multi-module scientific computing codes the problem is so overwhelming that it is most often handled by **starting over from scratch**. We believe that this is one of the root causes of the generally poor quality of user interfaces for scientific computing.

In this paper we present a revision model which through a series of transformations reduces the revision recognition problem to a 0-1 binary optimization problem. The solutions derived in this way closely mirror those derived by hand. Several examples are given, and an example of how these solutions can be used to help maintain user interfaces is presented.

2. The “Reticent Programmer”

Revision recognition can be thought of abstractly as a graph mapping problem. Given two versions of a program, consider both programs as directed graphs in which nodes are Basic Blocks and edges are directions which program execution can take. A Basic Block is a series of code statements in a computer program such that program flow enters only at the first statement and exits only at the last statement. Each Basic Block in the old version may either be mapped to a Basic Block in the new version or must be in the “deleted set”. Each Basic Block in the new version must either be part of a mapping from old to new or must be in the “added set”.

If the older version has N Basic Blocks and the newer version has M Basic Blocks then the number of possible mappings is of the order of $\text{Max}(N, M)!/|M - N|!$. The number of basic blocks in an application is generally proportional to the number of lines of code, with a typical ratio being around 1.8 lines of code per basic block. A large scientific computing application with one hundred thousand lines of code will have about 60 thousand Basic Blocks. The space of all possible mappings is clearly enormous. Yet it is possible for the interface programmer to solve this problem by inspection for small to medium sized codes. This implies that although the space of all possible mappings is enormous, the “search space” which the programmer traverses by making comparisons back and forth between old code and new is relatively small and compact. Our aim here is to design an algorithm which explores basically the same subspace of reasonable solutions the programmer would look at, even for programs that are so large that their inspection by hand would be prohibitive.

To identify this subspace of reasonable mappings we need to make a conjecture about the behavior of programmers. Suppose a programmer is assigned the task of revising a software application and then explaining exhaustively the purpose and effect of each change. We assume that the programmer will make revisions in such a way as to minimize the amount of explanation necessary. Intuitively we can imagine the programmer with block diagrams of the program and its revision making notations: “these blocks are the same in both versions”, “this block was changed to fix an error”,“this block was added as a new level of control” and so on. Our assumption is essentially that the programmer will produce the new version which minimizes the total size and complexity of such “notes”. We call this assumption the *Reticent Programmer Conjecture*.

3. Minimum Information Distance Mapping

Abstractly we wish to choose the mapping from old version to new which allows us to construct the smallest possible machine which fully describes the mapping. *Kolmogorov Complexity* is defined as the size of the smallest machine which can produce a given string. ’Machine’ is then usually extended to mean ’computer program’. Thus for small or repetitive strings a few lines of code will suffice, while for large strings of a more complex structure a much larger program is required. The properties of the Kolmogorov Complexity have been extensively studied [10]. Many methods of approximating the Kolmogorov Complexity have been described in the literature [10]. The simplest method is by using compression software as described in [10]. By “zipping” a file we create a small machine consisting of the zipped file and the unzip utility, which can reproduce the original string. It has been shown that the size of the zipped file is then proportional to the Kolmogorov Complexity.

The Kolmogorov or information distance can also be approximated using compression software. Suppose we have two files A and B and we wish to find the distance of *A given B*. We zip B and then take B’s Huffman encoding dictionary and use it as a guide to compress A. If A and B are the same or very similar, then the size of A zipped with B’s dictionary will be close to the size of A zipped with its own ideal dictionary. If A and B are far apart then B’s dictionary will do a poor job of compressing A. Normalized for the sizes of A and B this method yields an approximation to the information distance [10]. We call this approximation $\text{Kd}(A, B)$.

The information distance between two versions of a code gives a relative measure of how closely related the two versions are to each other. We call this the **Simple Information Distance** between the two versions. This simple distance is not the maximum rate of compression possible. Suppose the old version of a code consisted of the blocks AB and the new version consisted of AB'. If we take *A given A* and *B’ given B* together we will achieve a greater rate of compression than if we took *AB’ given AB*. Our goal is to find the mapping between blocks in the older and newer versions which if we compress each mapping separately produces the greatest possible compression. We call this mapping the **Minimum Information Distance Mapping [MIDMapping]** and the rate of compression found the **Minimum Information Distance [MIDistance]**. We define the “cost” of either deleting or creating a block so that if we find that all the blocks of the old version have been deleted and all the blocks of the new version have been created then the MIDistance will be the same as the simple information distance. Similarly, we normalize the individual compression rates so that if the two versions are the same and all old blocks map to identical blocks in the new version then the MIDistance will be zero.

The MIDistance can be thought of as a measure of how much effort is needed to understand the new version if the old version is completely understood. If the two versions are the same then a perfect understanding of the old version leads to a perfect understanding of the new version. If the two programs are not essentially different versions of the same program, then studying one of the programs will only be useful to the degree that the two programs are generally related.

Suppose that the old version of a program has the two blocks (“abcdfff”, “x”), and the new version has the blocks (“abcd”,”fffx”). These sections of code would clearly be related but by mapping single blocks to single blocks our algorithm might miss this conclusion. We therefore consider the class not just of all block-to-block mappings but of all isomorphic subgraphs of which block-to-block is the smallest possible case. This is also useful computationally. For very large codes the set of all block-to-block mappings is very large. By setting a minimum subgraph size we can dramatically reduce the size of the resulting algebraic problem.

A second computational issue is block coloring. We can color the graph representations of the programs as branches and non-branches by assuming that branches in the old version can only be mapped to branches in the new version. This is important because the general problem of finding isomorphic subgraphs has been found to be $NP-Hard$ [8]. Because code graphs are “densely colored”, the complexity of our algorithm to find subgraph isomorphisms is on the order of the factorial of the “mean free path” (the average length of a section of blocks all of the same type). The length of the mean free path increases only gradually with increasing program length.

Coloring is interesting in another sense. Suppose we had two versions of a code both of which had no branches. Then the problem of finding the MIDMapping would be equivalent to the “Single Alignment” problem from bioinformatics. If the two versions consisted of nothing but branches then the MIDMapping problem would be equivalent to “Multiple Alignment”.

4. Transformations

In this section we outline the series of transformations needed to calculate the MIDMapping and MIDistance. Suppose we are given two versions V_1 and V_2 of an application. We begin by stripping the codes of comments and continuation markers. We then calculate the Simple Information Distance Kd(V_1, V_2).

4.1 Basic Block Graphs

The first transform is from code to a Basic Block graph. We use the procedure outlined in [1] for deriving the Basic Block graphs G_1 of size M and G_2 of size N from V_1 and V_2 respectively. Denote the blocks of G_1 by b_{i1} and the blocks of G_2 by b_{j2}. The blocks of both graphs are colored as branch or non-branch. It should be noted that Basic Block networks are directed and semi-ordered. A semi-ordered graph is one in which some siblings may be swapped with each other while others may not, depending on the type of parent node involved. If the parent is a Branch Block then the ordering of child blocks is arbitrary and therefore unordered. All other blocks are ordered among siblings from left to right. The network is usually not a tree but it does have a root: the beginning of the primary subroutine, and at least one leaf: an execution end-point of the primary subroutine.

4.2 Isomorphic Subgraphs

The second transformation is from two Basic Block graphs to a set of their isomorphic subgraphs. Given two versions of a program in Basic Block representation, we find the set of all subgraphs which have the same connectivity, parent-to-child and sibling-to-sibling relationships and are therefore isomorphic. The subgraphs found must match block for block in type, meaning that they have the same coloring.

A general overview of the algorithm is as follows. (I) We examine each block of the original of the code in turn. (II) We find the set of all possible 1-to-1 mappings between the chosen block and blocks in the revised code which are of the same type. These isomorphisms of size one are added to the solution set. (III) Around each isomorphism of size one we recursively build larger and larger isomorphisms by considering the set of subgraphs which could be created by adding one more connected block of the same type in both versions. Isomorphic subgraphs found are added to the solution set unless they have already been added previously. (IV) When no further isomorphisms can be found we are finished with this block in the original version of the code and we move on to the next block. When all the blocks in the first version of the code have been examined, we have collected all possible subgraph isomorphisms between the two versions.

It should be noted that this process is complicated by the fact that children of branches are unordered. Thus we must consider the permutations among the children of branches. If these children are also branches then we need to further consider permutations among these grandchildren, and so on. Several nested “IF” statements in a row can lead to dozens of isomorphic subgraph candidates.

There is one other consideration, loops. Suppose that in the old version of a code we find the loop A-B-C-A, while in the new version we find A-B-C-X-A. These loops are clearly related but we would not necessarily recognize this because the two loops are not isomorphic. There are two possibilities. We could consider loops to be in their own class, making all loops effectively isomorphic. Considering that large applications may have hundreds of loops the number of combinations would be astronomical. Instead, we choose to ignore the back-jump of a loop so that in the above example we would be comparing A-B-C with A-B-C-X, and correctly find that the A-B-C section was preserved.

Our solution set S consists of s pairs of isomorphic subgraphs $((I_{11}, I_{12}), ...(I_{s1}, I_{s2}))$ where the left subgraph is from V_1 and the right subgraph is from V_2. If we are going to limit our solution to isomorphic subgraphs of size L or greater, then all pairs of isomorphic subgraphs of size less than L are removed from the solution set. We call the resulting set Q, so that $Q_i = (I_{i1}, I_{i2})$. Q is the set of potential mappings between V_1 and V_2.

4.3 Computing the Kolmogorov Distances

We next derive the rate of compression for the isomorphic subgraph in Q. The underlying text of each subgraph is constructed by concatenating the text of each block "depth first". Thus, for a block containing the text "XXXX" with two child blocks with text "YYYY" and "ZZZZ" the constructed text would be "XXXXYYYYZZZZ". In the special case of a "join" where two or more edges lead into a block we will take the block to be the child of its rightmost parent. Denoting this depth first concatenation function CAT(I), where I is a subgraph, for each pair of isomorphic subgraphs q_i in Q we find

$$c_i = \text{Kd}(\text{CAT}(I_{i1}), \text{CAT}(I_{i2})).$$

The ideal rate of compression for the right hand subgraph text is also recorded as x_i.

$$x_i = \text{Kd}(\text{CAT}(I_{i2}), \text{CAT}(I_{i2})).$$

We note that X (the average of the x_i's) would be the compression rate if V_1 was identical to V_2.

4.4 Algebraic System

The next step is to reformulate the problem of deciding which mappings in Q maximize the compression of V_2 given V_1 as an algebraic system. This system consists of an objective function which is to be minimized, together with constraints upon the solution. Define the variables $(q_1, q_2, ..., q_s)$ such that if the mapping Q_i is part of the MIDMapping then q_i takes the value 1, otherwise q_i is 0. We introduce the variables $(y_1, y_2, ..., y_m)$, which take the value 1 if the corresponding block in Version 2 is created and 0 otherwise, and the variables $(z_1, z_2, ..., z_n)$ which take the value 1 if the corresponding block in Version 1 is deleted and 0 otherwise. The "costs" for the q_i are given by the corresponding compression rates, c_i. The costs for both y_i and z_i are set to Kd(V_1, V_2) (the Simple Information Distance). If our MIDMapping solution were that all blocks in V_1 have been deleted and all blocks in V_2 have been added then the MIDistance would be equal to Kd(V_1, V_2) (the Simple Information Distance).

The objective function can be written as

$$q_1 * c_1 + q_2 * c_2 + ... + q_s * c_s + (y_1 + ... + y_m + z_1 + ... + z_n) * D.$$

Constraints arise from the fact that some mappings preclude others from also being in the MIDMapping. Take left(b_{i1}) to be the set of all mappings in which the block b_{i1} in G_1 participates. Only one of these mappings can be part of the MIDMapping. The block could also be deleted. We formulate the constraints on b_{i1} as:

$$\sum \text{left}(b_{i1}) + z_i = 1.$$

For blocks in G_2 we can similarly formulate the constraints on b_{j2} as

$$\sum \text{right}(b_{j2}) + y_j = 1.$$

Thus there will be $N + M$ constraints on the system.

4.5 0-1 Optimization

The system we have constructed is an "0-1 optimization" problem. The state variables q_i, y_i, and z_i can take the values zero or one exclusively. The objective function, which is really the overall rate of compression, is the sum of these state variables times their costs.

The final step is the solution of the optimization problem. The algebraic constraints are generated in text, MPS format, a standard for encoding algebraic systems. We then use the 0-1 optimization software package "opbdp" [4]. The solution is a listing of the values 0,1 of the state variables. The set of q's, y's and z's set to 1 in this listing is the MIDMapping. The calculated value of the objective function is divided by the number of mapping pairs in the MIDMapping solution to get an average compression rate. The MIDistance compression rate can be compared to the ideal compression rate X, which it approaches for smaller, less complicated revisions.

5. Results

We begin this section by introducing our target applications. **LAPACK** is a well-known collection of commonly used FORTRAN numeric routines [2]. It is convenient for this study because previous releases 1a, 1b, 1.1, 2.0 and 3.0 are available, allowing us to trace through 15 years of revisions. The codes which we follow for this paper are SLANV, a Schur factorization routine, and SSYTF2, a

Application	Lines	Blocks	Edges	Branches
SLANV2 ver 2.0	82	20	31	7
SLANV2 ver 3.0	99	21	30	8
SSYTF2 ver 1.0	183	85	155	31
SSYTF2 ver 2.0	181	84	153	32
SSYTF2 ver 3.0	193	94	143	33

Table 1. Properties of the target software.

Bunch-Kaufman factorization routine. Table 1 summarizes the properties of SLANV and SSYTF2.

Table 2 shows the number of isomorphic subgraphs found between versions of SLANV and SSYTF2. We also include a set of isomorphic subgraphs derived from a hypothetical revision from SLANV version 2 to SSYTF2 version 1. We will also follow this example to show that given two codes which are not versions of the same application the MIDistance will not improve much over the Simple Distance.

Isomorphic Subgraphs	1	2	4
SLANV2 ver 2 to ver 3	225	203	58
SSYTF2 ver 1 to ver 2	3800	3231	3236
SSYTF2 ver 2 to ver 3	4228	2965	1215
SSYTF2 ver 1 to ver 3	4317	3001	1350
SLANV2 v2 to SSYTF2 v1	893	186	48

Table 2. The isomorphic subgraphs of SLANV2 and SSYTF2 by size. By limiting the subgraph size to 4 or larger, the number of state variables in the algebraic problem is reduced.

Table 3 shows Kd, the compression rate achieved by zipping the column application using the row application's encoding dictionary, for versions of SLANV2 and SSYTF2.

Table 4 shows the average ideal compression achieved by compressing the codes block by block. The MIDistance approaches the ideal compression rate with smaller and less complex revisions. For codes which are essentially unrelated, the MIDistance will approach the simple compression rate at shown in Table 3. Thus, for the problem MIDMapping$(SSYTF2v1, SSYTF2v3)$ we find a MIDistance close to the ideal of 28%, and for the problem MIDMapping$(SLANV2v1, SSYTF2v3)$ we find the MIDistance to be close to 45%.

Table 5 shows the MIDistance compression for the two sample problems. We used the opbdp [4] binary optimization package to find these solutions. The global minimum of the objective function is divided by the number of mappings found to arrive at the MIDistance, the

	SSYTF2v1	SSYTF2v3	SLANV2v1
SSYTF2v1	41%	45%	77%
SSYTF2v3	42%	43%	79%
SLANV2v1	78%	80%	40%

Table 3. Compression rates of column application given the row application. Thus, SSYTF2v3 compresses to a little less then one half of its size using its own Huffman encoding dictionary and to about 70 percent of its size using the dictionary of SLANV2, therefore Kd$(SSYTF2v3, SSYTF2v3) = 0.45$ and X$(SLANV2v1, SSYTF2v3) = 0.70$.

SSYTF2v1	SSYTF2v3	SLANV2v1
23%	28%	27%

Table 4. The ideal compression rate calculated by compressing the Basic Blocks of a code one at a time. Compression software does have some overhead and although care was taken to suppress extraneous labels and headers the theoretical rate of ideal compression is probably somewhat greater than shown.

average compression rate. In the case of the problem where the two applications are not versions of the same application, MIDMapping$(SLANV2v1, SSYTF2v3)$, the MIDistance was found to be 44%. This is an improvement over the simple compression rate of 80%, but does not approach the ideal compression for the new code, SSYTF2v3 of 27%. Any two numeric codes written in the same programming language share some of the same features, like double loops, and therefore are more compressed by considering block-to-block compression. Given two unknown codes the MIDMapping algorithm determines them to be versions of the same application if the MIDistance approaches the ideal compression rate and if the resulting MIDMapping shows a high degree of organization rather than the random pairing of matching features.

Figure 1 shows the result of the MIDMapping$(SLANV2v1, SSYTF2v3)$, with square boxes indicating matching isomorphic subgraphs and ovals indicating blocks that have either been deleted in SLANV2v1 or added in SSYTF2v3. SLANV2v1 and SSYTF2v3 are not versions of the same application so the mapping shows little organization. Parent-child relationships among mapped subgraphs are not preserved. There is no section of blocks in the old version which appear intact

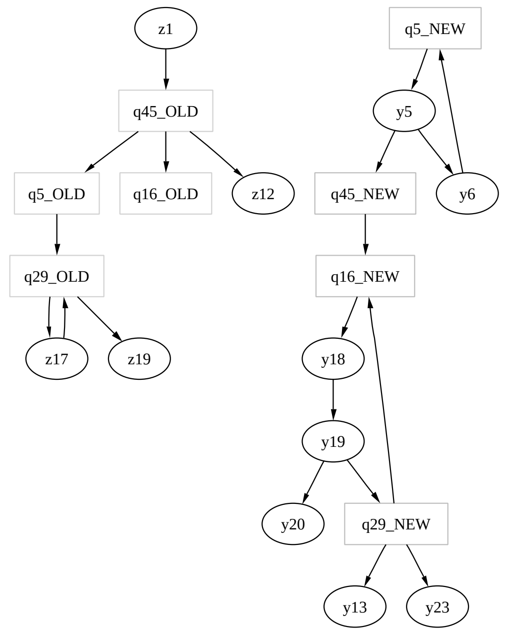

Figure 1. We would expect the MIDMapping of two closely related versions of an application to show a certain degree of organization indicating where revision had taken place. The MIDMapping$(SLANV2v1, SSYTF2v3)$ shows only a random association of blocks.

in the new version.

	Compression Rate
MIDistance(SSYTF2v1,SSYTF2v3)	0.32
MIDistance(SLANV2v1,SSYTF2v3)	0.44

Table 5. The MIDistances for our two sample problems.

Figures 2 and 3 show the solution to MIDMapping$(SSYTF2v1, SSYTF2v3)$. In contrast to the MIDMapping$(SLANV2v1, SSYTF2v3)$ solution there is a strong sense of organization. The core of the original version of SSYTF2v1 remains intact and it is possible by inspection to see where code slices have been added to the new version, SSYTF2v3. The filled black rectangles are code that has been preserved from the previous version.

6. Using the MIDMapping

This project is part of a larger project called JMPL - Java Math Package Launcher. The idea is to provide an environment for generating and maintaining graphic user interfaces for complex scientific applications. JMPL uses SIFT, a module we have developed for discovering user interface requirements [11] and MAUI XML-to-GUI [6] technology to generate a GUI skeleton. The technology developed in this paper is intended as a first step towards automating the maintenance of these generated interfaces.

6.1 Discussion

This study was done using applications coded in FORTRAN77, which is still an important research and development language for scientific computing. We believe that the methodology developed here is largely language independent and would give similar results for C or Pascal. Object oriented languages present significant new challenges. Generic class instantiation would have to be treated as a branch with edges for each possible instance type, leading to a “fan-out” of combinations of possible instantiations. Object oriented languages are also organized differently. Typically a programmer uses a hierarchical toolkit of objects such as the JSwing library. These object libraries form a convenient sublanguage for the programmer. The challenge is to recognize revisions in both the application and sublanguage simultaneously. We plan to pursue this research issue in future work.

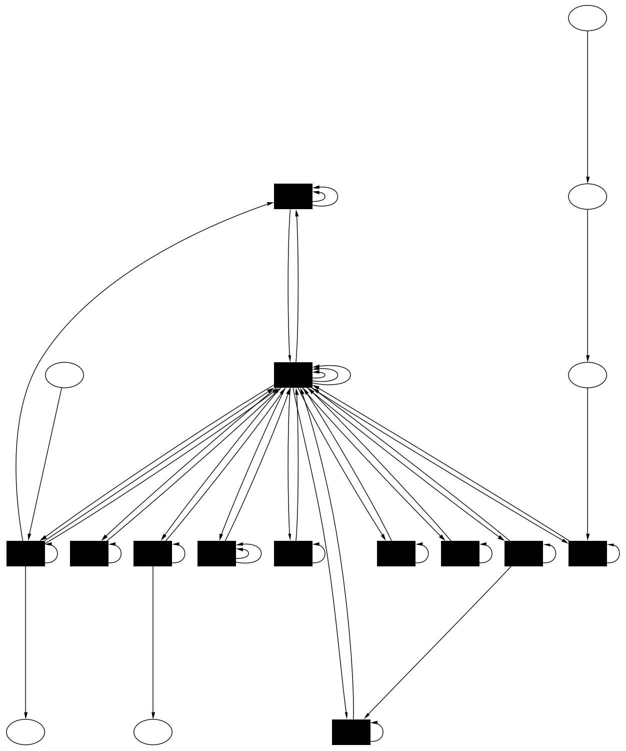

Figure 2. The MIDMapping solution for SSYTF2v1 shows a compact core of functionality which has been preserved in the current version. The black filled rectangles are subgraphs of size 4 which map to isomorphic subgraphs in the new version SSYTFv3.

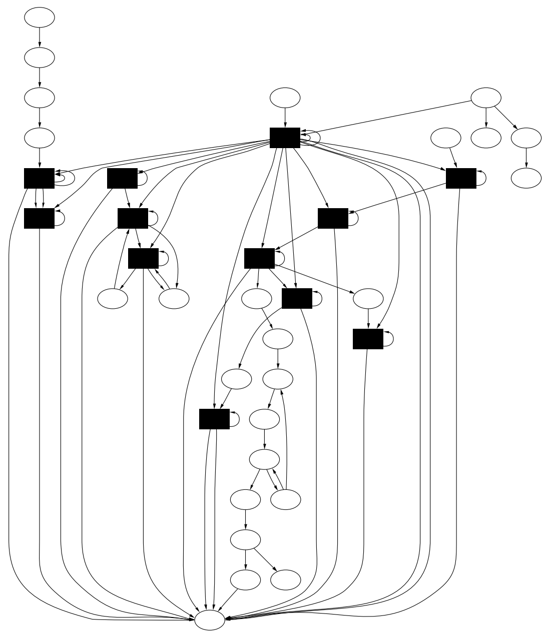

Figure 3. The MIDMapping solution for SSYTF2v3 shows the addition of several slices of new code.

7. Acknowledgments

This work was supported by grants: NSF/ITR ACI-0086061, NSF/KDI ATM-9873133, and DOE DE-FG03-00ER 25430.

References

[1] A.Aho. *Compilers - Principles, Techniques and Tools*. Addison-Wesley, 1988.

[2] E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. D. Croz, and A. Greenbaum. *LAPACK Users' Guide*.

[3] S. Balay, K. Buschelman, W. Gropp, D. Kaushik, L. McInnes, and B. Smith. *PETSc: The Portable, Extensible Toolkit for Scientific Computation*. Argonne National Lab.

[4] P. Barth. *A Davis-Putnam Based Enumeration Algorithm for Linear Pseudo-Boolean Optimization*. Max-Plank Institute.

[5] D. Beazley. *Software Wrapper and Interface Generator*, 2002. SWIG Home Page http://www.swig.org.

[6] P. Boggs, L. Lehoucq, K. Long, A. Rothfuss, E. Walsh, and R. Whiteside. A maui user's guide. http://csmr.ca.sandia.gov/projects/maui/docs/MauiTutorial/.

[7] D. Gannon. The information power grid and the problem of component systems for high performance distributed computing. In *Globus Retreat 2000*.

[8] M. Garey and D. Johnson. *Computers and Intractability. A Guide to the Thory of NP-Completeness*. W.H.Freeman, 1979.

[9] N. A. Group. *IRIX Explorer*, 2002. http://www.nag.com.

[10] M. Li and P.Viytanyi. *An Introduction to Kolmogorov Complexity and its Applications*. Springer Verlag, 1997.

[11] A. Strelzoff and L. Petzold. Automated user interface requirements discovery for scientific computing. In *11th IEEE Conference on Requirements Engineering*, 2003.