INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

ProQuest Information and Learning
300 North Zeeb Road, Ann Arbor, MI 48106-1346 USA
800-521-0600

SOFTWARE COMPREHENSION AND PROGRAM SLICING

Lava Kumar

A Major Report

in

The Department

of

Computer Science

Presented in Partial Fulfillment of the Requirements

for the degree of Master of Computer Science at

Concordia University

Montréal, Québec, Canada

November 2001

ABSTRACT

Software comprehension and program slicing

Lava Kumar

Program comprehension is a very difficult task but, at the same time crucial for system maintenance and development. As programs grow in size, this task becomes more and more daunting. It is, therefore, necessary to evolve program comprehension strategies, which allow us to minimize the amount of data that is to be observed and inspected as part of the comprehension process.

Program slicing has been widely accepted as a very powerful technique for programmers to comprehend very large and complex programs. This technique decomposes a large program into a smaller one that contains only statements relevant to the computation of a selected function and variable.

Various slicing techniques have been evolved over the past years, such as static slicing, dynamic slicing and hybrid slicing. Each technique exploits a different algorithm to compute slices based on some slicing criterion.

In this report, we present an alternate approach to hybrid slicing and its integration in the CONCEPT (**C**omprehension **O**f **N**et **CE**ntered **P**rograms and **T**echniques) project, which provides a set of tools and techniques to slice a program using various slicing approaches/methodologies. CONCEPT also utilizes information derived from program slicing algorithms to derive a program comprehension framework.

ACKNOWLEDGEMENTS

I would like to acknowledge Dr. Juergen Rilling, my supervisor, for all his patience and guidance throughout my project work. His supervision and support proved instrumental in helping me to complete this work in limited time.

I wish to thank my wife and children for encouraging me to pursue the second graduate degree while allowing themselves a lot of discomfort during this period. I would like to thank all the fellow students and others who have been directly or indirectly associated in helping me to prepare this report.

Table of Contents

List of Figures

1. Introduction

Program comprehension is a very difficult task but at the same time crucial for system development and maintenance. As programs grow large and complex, comprehension of these systems becomes more and more aggravating for programmers who maintain these programs.

Several studies have shown that a large amount of time and money is being spent on maintenance and modification of existing programs. It is, therefore, necessary to have a proper program comprehension strategy before maintenance or modification is done to a program. Poor design, unstructured programming methods and crisis-driven maintenance can contribute to poor code quality which in turn affects program comprehension. The essence of program comprehension is to identify program artifacts and understanding their relationships; this process is essentially pattern matching at various abstraction levels via mental pattern recognition by the software engineer and the aggregation of these artifacts to form more abstract system representations [44]. The comprehension of source code plays an important role in ensuring correctness during the maintenance process. Debugging and program tracing techniques are available for program comprehension; however, the volume of information presented to the developer becomes daunting as programs become large and complex.

The goal of program comprehension is to acquire sufficient knowledge about a software system so that it can evolve in a disciplined manner. There are varieties of support mechanisms for aiding program comprehension, which can be grouped into four categories: unaided browsing, leveraging corporate knowledge, experience, and

computer-aided techniques like reverse engineering. In this report, the focus is on reverse engineering and now it can be applied effectively in program comprehension [3, 18, 47]. One approach to improve the comprehension of programs is to reduce the amount of data that has to be observed and inspected. Program slicing is a program decomposition technique that transforms a large program into a smaller one that contains only statements relevant to the computation of a selected function. Applications of program slicing can be found in software testing, debugging and maintenance where program slicing essentially reduces the amount of data that has to be analyzed in order to comprehend a program or parts of its functionality.

Static slicing:

The notion of static program slicing originated in the seminal paper by Weiser [59,60]. Weiser defined a slice S as a reduced, executable program obtained from a program P by removing statements from P such that S replicates parts of the behaviour of P.

Dynamic slicing:

Korel introduced a major extension of program slicing, with Laski [24], called dynamic slicing. The dynamic slicing approach not only utilizes static source code information, but also dynamic information from program executions on some program input.

The dynamic slice preserves the program behaviour for a *specific* input, in contrast to the static approach, which preserves the program behaviour for the set of all inputs for which the program terminates. By considering only a particular program execution rather than all possible executions, dynamic algorithms may compute slices that are significantly smaller than the slices computed by the static slicing algorithms.

Hybrid slicing:

Hybrid program slicing algorithms were introduced to take advantage of both static and dynamic slicing properties. These algorithms use static information to lower the run time overheads and dynamic information is used for more accurate handling of dependencies [49]. Gupta and Souffa proposed in [15] to use both static and dynamic information for the computation of program slices for structured programs.

CONCEPT:

The Comprehension Of Net CEntered Programs and Techniques project was developed to design and develop an integrated environment. The CONCEPT project provides an open implementation platform utilizing a variety of comprehension concepts, tools and methods, as well as user experience and knowledge to streamline the comprehension process by focusing a programmer's attention on these program parts and tools that are relevant for the selected task.

1.1 Motivation and objective

The objective of this work was to integrate and implement static and hybrid slicing within the CONCEPT framework. CONCEPT was designed as an open comprehension framework to guide programmers during the challenging task of understanding large traditional and object-oriented programs and their executions.

In addition, a new alternative hybrid slicing technique referred to as "alternate hybrid slicing" is presented and integrated within the CONCEPT project.

1.2 Scope of the dissertation

The presented report consists of five sections including this section.

In section two, an overview and a survey of related literature and existing approaches of static, dynamic and hybrid slicing, as well as a general comparison of these algorithms is presented.

In section three, an overview of the CONCEPT environment is presented. In section four, an alternate technique to hybrid technique is introduced and discussed.

In section five, conclusions of the present work and proposed future directions related to this study are outlined.

2. Background

Program comprehension is a crucial part of system development and software maintenance. It is expected that a major share of systems development effort will go into modifying and extending pre-existing systems, about which programmer usually know little. Change to a system may be necessitated for adaptive, perceptive, corrective or preventive reasons. Understanding the system, incorporating the change, and testing the system to ensure that the change has no united effect on the system are the three facets of software maintenance [13,36,55]. For all of these maintenance activities, software comprehension plays a pivotal role. A commonly used technique to enhance the comprehensibility of software systems is through reverse engineering. This technique is used to analyze a subject system with the goals to: (a) identify the system's components and their inter-relationships (b) create representations of a system in another form at a higher level of abstraction and (c) understand the program execution and the sequence in which it occurred. Numerous theories have been formulated and empirical studies are conducted to explain and document the problem-solving behavior of software engineers engaged in program comprehension. Cognitive models have been introduced to describe the comprehension processes and knowledge structures used to form a mental representation of the program under examination [40].

Typically, a program performs a large set of functions/outputs. Rather than trying to comprehend all of a program's functionality, programmers will focus on selected functions (outputs) with the goal to identify which parts of the program are relevant for that particular function. Program slicing provides support during program comprehension, by capturing the computation of a chosen set of variables/functions at

some point (static slicing) in the original program or at a particular program execution position (dynamic slicing). This will lead to a smaller, simplified version of the original version of the program without changing the local semantics of the extracted slice.

2.1 Basic program slicing terminology

Program slicing terminology is based on the terminology of program dependence theory. Most of the slicing algorithms are represented by a directed graph, which captures the notion of data dependence and control dependence in programs. The program structure is represented by a flow graph $G = (N, A, s, e)$, where (1) N is a set of nodes, (2) A, a set of arcs, is a binary relation on N and (3) s and e are, respectively, unique entry and exit nodes. A node corresponds to an assignment statement, an input or output statement or the predicate of a conditional or a loop statement, in which case, it is called a *test* node. A *path* from the entry node s to some node k, $k \in N$ is a sequence $<n_1, n_2, n_q>$ of nodes such that $n_1 = s$, $n_q = k$ and $(n_i, n_{i+1}) \in A$, for all n_i, $1 \leq i < q$. A path that has actually been executed for some input will be referred to as an *execution trace*. A path is regarded feasible only if there exists some input data, which causes the path to be traversed during a particular program execution. A program trajectory has been defined as a feasible path that has actually been executed for some specific input. Notationally, an *execution trace* is an abstract list (sequence) whose elements are accessed by position in it, e.g., for trace T_x in Figure 2, $T_x(4)=4$, $T_x(5)=8$. Node Y at position p in T_x (e.g., $T_x(p)=Y$) will be written as Y^p and referred to as an *action*. Y^p is a *test* action if Y is a test node v^q denotes *variable v at position q*, i.e., variable (object) v before execution of node $T_x(q)$.

For example, T_x = <1,2,3,4,9,10,11,12> is the execution trace when the program in Figure 1 is executed on the input x: $MSRP$ = 25000; this execution trace is presented in Figure 2.

```
1. normal_profit = 100;
2. bonus_profit  = 1500;
3. cin >>MSRP;
4. if (MSRP > 30000)
   {
5.       bonus_profit     = MSRP + bonus_profit ;
6.       normal_profit    = MSRP + normal_profit ;
7.       Showroom_price  = MSRP + bonus_profit + normal_profit;
   }
8.  else
   {
9.       Showroom_price = MSRP + normal_profit ;
   }
10.  cout << normal_profit;
11.  cout << bonus_profit;
12.  cout << Showroom_price;
```

Figure 1: Sample program

```
1[1]    normal_profit = 100
2[2]    bonus_profit  = 1500;
3[3]    cin >>MSRP;
4[4]    if (MSRP > 30000)
8[5]    else
9[6]    Showroom_price = MSRP +normal_profit ;
10[7]   cout << normal_profit;
11[8]   cout << bonus_profit;
12[9]   cout << Showroom_price;
```

Figure 2: An execution trace of the sample program on input MSRP = 25000

A *use* of variable v is an action in which this specific variable is referenced. A *definition* of variable v is an action, which assigns a value to that variable. The following assumptions are made: $U(Y^p)$ is a set of variables whose values are used in action Y^p and $D(Y^p)$ is a set of variables whose values are defined in Y^p. Sets $U(Y^p)$ and $D(Y^p)$ are determined during program execution, especially for array and pointer variables because it is possible to identify the specific array elements that are used or modified by the action during program execution.

Static data dependence captures the situation in which one node assigns a value to an item of data and the other action uses that value. Data dependence is based on the concepts of a variable definition and use. Thus, a node j is data dependent on node i if there exists a variable v such that: (1) v is defined in node i, (2) v is used in node j and (3) there exists a path from i to j without an intervening definition of variable v. In the sample program of Figure 1, there exists data dependence between node 5 (using the variable *MSRP)* and node 3 (defining the variable *MSRP).*

Static control dependence is based on the concepts of post-dominance. Informally, this can be thought of as one program statement determining in some way, whether or not another statement will be executed. The control dependence is defined as [11]; let *Y* and *Z* be two nodes and *(Y, X)* be a branch of Y. Node Z post-dominates node *Y* iff *Z* is on every path from *Y* to the exit node *e*. Node *Z* post-dominates branch *(Y, X)* iff *Z* is on every path from *Y* to the program exit node *e* through branch *(Y, X)*. *Z* is *control dependent* on *Y* iff *Z* post-dominates one of the branches of *Y* and *Z* and does not post-dominate *Y*. The concept of post-dominance means that all execution paths in a control flow graph from a specific node i to the program end must pass through another node j before they reach the program end [19]. For example, in the sample program of Figure 1, there exists a control dependence between the node 5 and node 4, where node 5 is control dependent on node 4.

2.1.1 Backward program slicing

In backward slicing, slices are computed by gathering statements and control predicates through backward traversal of the program dependencies, starting at the slicing criterion [56]. The slices are mostly obtained by traversing the edges of graph towards the root node.

2.1.2 Forward program slicing

A forward program slice contains all statements and control predicates dependent on the slicing criterion. A statement is 'dependent' on the slicing criterion: 1) if the value computed at that statement depend on the values computed at the slicing criterion or 2) if

the values computed at the slicing criterion determine the fact if the statement under consideration is executed or not.

2.1.3 Removable Blocks

Korel introduced the notion of removable blocks in [29] and described it as the part of program text (code) that can be removed during slice computation. A block is described as the smallest component of the program text that can be removed (e.g. assignment statement, input and output statements, etc.). Test nodes (predicates of conditional statements) are not removable individually and therefore, they are considered part of a complex block where they can be removed if none other block in the complex block is said to be not removable. Intuitively, a block may be removed from a program if its removal does not "disrupt" the flow execution on some input x. Each block B has a regular entry to B and a regular exit from B referred to as *r-entry* and *r-exit*, respectively. In unstructured programs, because of jump statements, execution may enter a block directly without going through its *r-entry*; in this case, one can say execution enters the block through a *jump entry*. Similarly, execution can exit a block without going through its *r-exit*; in this case, the execution leaves a block through a *jump exit*. Let B_1, B_2, B_3 be a sequence of three blocks in a program. Block B_2 may be removed, if during execution of the program on some input x, the execution exits from block B_1 through its *r-exit*, enters block B_2 through its *r-entry*, leaves B_2 through its *r-exit*, and enters block B_3 through its *r-entry*. If block B_2 is removed and the resulting program is executed on the same input x, the program execution will, after leaving B_1 through *r-exit*, enter block B_3 directly through its *r-entry*. In this case, the flow of execution is not disrupted by the

removal of block B_2. Figure 3 shows the sample program represented in Korel's removable block concept.

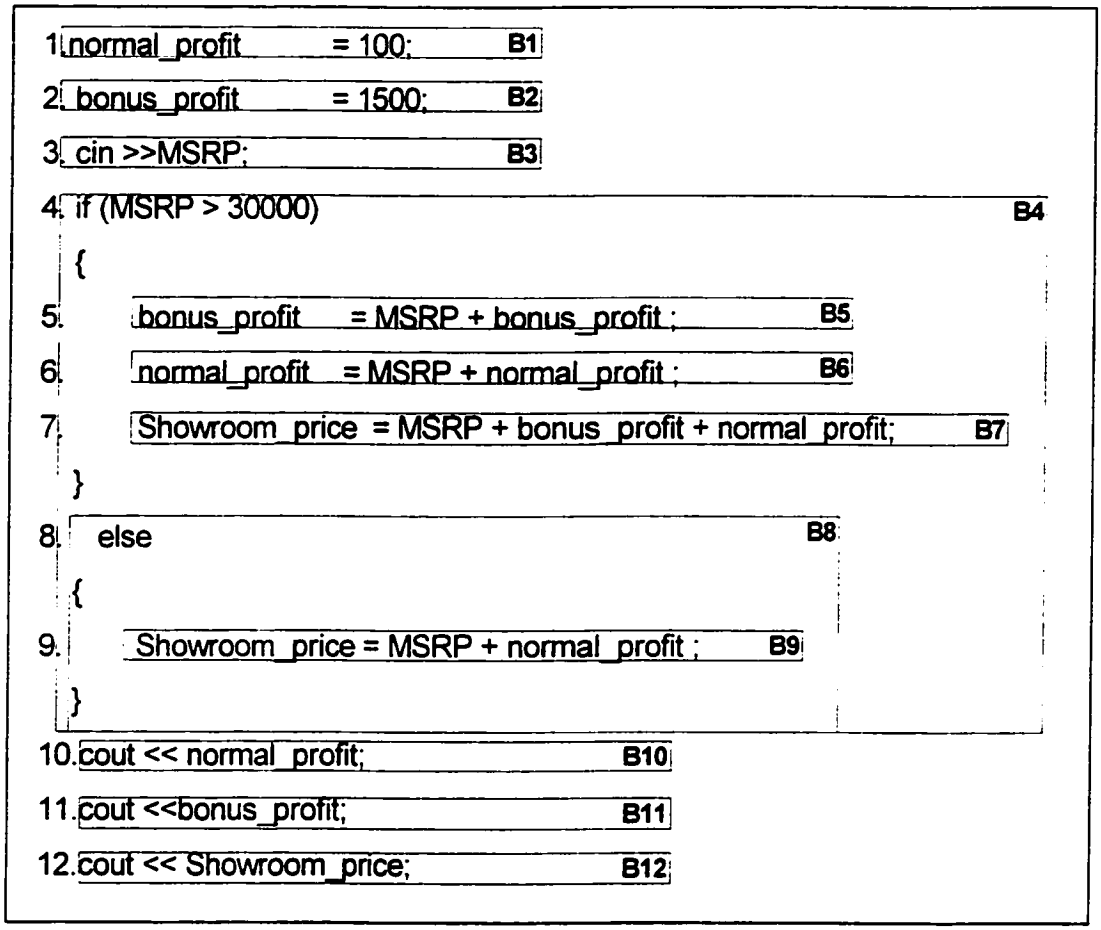


Figure 3: Sample program (Figure 1) with removable blocks

2.2 Static program slicing

Based on the original definition of Weiser[60], static program slice S consists of all statements in program P that may affect the value of variable v at some point p. The slice is defined for a slicing criterion $C=(x,V)$, where x is a statement in program P and V is a subset of variables in P. Given C, the slice consists of all statements in P that potentially affect variables in V at position x. Static slices are computed by finding consecutive sets of indirectly relevant statements, according to data and control dependencies.

```
1[1].    normal_profit = 100;
2[2].    bonus_profit  = 1500;
3[3].    cin >>MSRP;
4[4].    if (MSRP > 30000)
         {
5[5].      bonus_profit = MSRP + bonus_profit ;
6[6].      normal_profit =  MSRP + normal_profit ;
7[7].      Shoom_price  = MSRP + bonus_profit + normal_profit;
         }
8[8].    else
         {
9[9].      Showroom_price = MSRP +normal_profit ;
         }
12[10].  cout <<Showroom_price;
```

Figure 4: Static slice for variable *Showroom_price*

2.2.1 Advantages and disadvantages of static program slicing

Static program slicing [60] derives its information through the analysis of the source code. Its strength can be found particularly in the following areas:

(a) The computation of a static program slice is relatively rapid (compared to the dynamic program slice) as only the static analysis of the source code and no analysis of program execution is required

(b) It helps the user to gain a general understanding of the program parts that contribute to the computation of a selected function with respect to all possible program executions.

(c) It is not necessary to identify relevant input conditions.

However, static program slicing has some major drawbacks and they are as follows:

(a) For programs containing conditional statements, dynamic language constructs like polymorphism, pointers, aliases, etc., static slicing has to make conservative assumptions with respect to their run-time contribution that might be relevant for slice computation.

(b) Due to its static nature, static program slicing does not provide any information with respect to the analysis of program executions as slices are based on static information.

(c) In most cases, static program slicing produces larger program slices than the dynamic program slicing algorithms.

2.3 Dynamic program slicing

The goal of program slicing is to find the slice with the minimal number of statements, but this goal may not be always achievable in general static program slicing. A dynamic program slice overcomes the limitations of the static program slicing algorithms as it is based on a particular program execution (program input). A dynamic program slice, as originated by Korel and Laski [24], is an executable part of the program whose behaviour is identical, for the same program input, to that of the original program with respect to a variable of interest at some execution position. In the existing dynamic program slice

algorithms, the major goal is to identify those actions in the execution trace that contribute to the computation of the value of variable y^q by identifying data and control dependencies in the execution trace. However, it is also important to identify actions that do not contribute to the computation of y^q. The more such "non-contributing" actions that can be identified, the smaller will be the dynamic program slice computed by the algorithm. A slicing criterion of program P executed on program input x is a tuple $C=(x,y^q)$ where y^q is a variable at execution position q. A *dynamic program slice* of program P on slicing criterion C is any syntactically correct and executable program P' that is obtained from P by deleting zero or more statements. In addition, the *dynamic program slice* when executed on program input x, produces an execution trace T'_x for which there exists a corresponding execution position q' such that the value of y^q in T_x equals the value of $y^{q'}$ in T'_x. A dynamic program slice P' preserves the value of y for a given program input x. The goal to find the smallest slice may be difficult; however, it is possible to determine a safe approximation of the dynamic program slice that will preserve the computation of the values of variables of interest. Most of the existing algorithms of dynamic program slice computation use the notion of data and control dependencies to compute dynamic program slices. Dynamic program slicing algorithms presented in [3,13] do not compute correct slices for unstructured programs (shown in [31]) and/or procedural language constructs. In [21], an algorithm for the computation of inter-procedural slicing of structured programs was presented. However, this algorithm is limited to structured programming language constructs. In [31], a dynamic program slicing algorithm based on the notion of removable blocks was introduced. This algorithm computed correct executable slices for unstructured non-object-oriented programming

languages. Later, the algorithm was further refined in [47] for all language constructs found in major procedural programming languages. A forward computation of dynamic slices for structured programs was introduced in [29] that does not require the recording of an execution trace. A dynamic program slicing for object-oriented programs based on forward computation was introduced in [62] that computes non-executable program slices, but it is not based on the notion of removable blocks.

By *last definition LD* (v^k) of variable v^k in execution trace T_x, it means, action Y^p such that (1) $v \in D(Y^p)$ and (2) for all i, $p < i < k$ and all Z such that T_x (i)=Z, $v \in D(Z^i)$, in other words, action Y^p assigns a value to variable v and v is not modified between positions p and k. For example, the last definition of variable *Showroom_price* at node 12^9 in execution trace of Figure 2 is action 9^5.

Dynamic data dependence captures the situation where one action assigns a value to an item of data and the other action uses that value. For example, in the execution trace of Figure 2, 3^3 assign a value to variable *MSRP* and 4^4 uses that value.

Dynamic control dependence captures the influence between "test" actions and actions that have been chosen to be executed by these "test" actions. The concept of control dependence may also be extended to actions by using the concept of control dependence between nodes. Action Z^k is control dependent on action Y^p iff (1) $p < k$, (2) Z is control dependent on Y, and (3) for all actions X^i between Y^p and Z^k, $p < i < k$, X is control

dependent on *Y*. For example, action 9^6 is control dependent on action 4^4 as action 8^5 is control dependent to 4^4 in the execution of Figure 2. Figure 5 shows a dynamic program slice for variable *Showroom_price* at block 12, with input *MSRP* = 25000. A dynamic program slice can be regarded as a refinement of the static program slice. By applying dynamic analysis [24], it is easier to identify those statements in the program, that do not have influence on the variables of interest.

```
1[1].    normal_profit = 100;
3[2].    cin>>MSRP;
4[3].    if (MSRP > 30000 )
         {
         }
8[4].    else
         {
9[5].     Showroom_price := MSRP + normal_profit;
         }
12[6].    cout<<Showroom_price;
```

Figure 5: Dynamic slice for *Showroom_price* at block 12, with input MSRP = 25000

2.3.1 Advantages and disadvantages of dynamic program slicing

As already stated, dynamic program slicing is a further refinement of static program slicing, the following are considered the main advantages as compared to the later:

(a) Dynamic program slicing allows a reduction in the slice size and a more precise handling of arrays and pointer variables at runtime.

(b) Dynamic program slicing computation can utilize information about the actual program flow for a particular program execution, which leads to an accurate handling of dynamic and conditional language constructs and therefore, leads to smaller program slices.

(c) Allows for additional application in performance analysis and debugging.

There are few associated disadvantages to get the above benefits such as:

(a) In dynamic program slicing (compared with static slicing), it is necessary to identify relevant input conditions for which a dynamic program slice should be computed. A commonly used approach to identify such input conditions is referred to as an operational profile, which is a well-known concept that is frequently applied in testing and software quality assurance.

(b) The computation of dynamic slices is based on a particular program execution that incurs a high run time overhead due to the required recording of program executions and/or analysis of every executed statement.

2.4 Hybrid program slicing

Hybrid program slicing algorithms takes advantage of the best properties of both static and dynamic slicing to derive a compromising slicing solution. However, very few research work focus on this type of slicing. Gupta and Souffa in [15] used pre-set breakpoints history information in their static slicing to solve conditional predicates. They characterized the procedure as follows:

(a) The user sets the breakpoints and starts the execution.

(b) When the breakpoint is encountered, the user examines the values of variables at breakpoint.

(c) If the values are as expected, the user resumes the program execution. However, before resuming the execution, the user may disable some breakpoints or add new ones.

(d) If the values are incorrect, the user requests slicing information for selected variables to potential causes of errors.

Information on conditional predicates helped to reduce the size of the static program slice without having to generate complete execution trace of the program. It assumes that the user can identify the breakpoints at various points in the program. Limitations of this approach are that it only supports structured programs, assumes user's knowledge of program for breakpoints and it might not compute executable slice. It is generally agreed that mixing of static and dynamic program slicing is a good compromise between accuracy and time performances. However, the user should interfere as little as possible to compute the slice [49]. Gupta et al. in their paper [15], have presented an algorithm based on breakpoint history and some experimental results of their work. Schoenig and Ducass'e [49] hybrid backward slicing algorithm for *Prolog* and is only applicable to a limited subset of *Prolog* programs. They developed only preliminary prototype and there is no further research evidence on their algorithm or test results.

2.4.1 Comparison of hybrid program slicing with other program slicing approaches

Dynamic program slicing algorithms have the advantage of being able to handle dynamic language constructs. However, the computation of dynamic slices is based on a particular program execution that incurs a high run time overhead due to the required recording of

program executions and/or analysis of every executed statement. The time and space required for recording execution trace and traversing the same trace depends on the number of times each of these executed statements might be significant. For example, if the program has a "**for**" loop for **n** times and has **m** statements in the loop, then the **m** x **n** entries have to be stored/traversed and analyzed. Many statements are executed merely because they are part of the program flow but their execution might not be relevant at all for the computation of the selected function. In the above example, for instance, say there are only **l** statements that are relevant to the computation of the selected function. This will result in unnecessary tracing and traversing of ((**m-l**)+1) x **n** executed statements during slice execution.

As the source code size grows, space and time complexity increases drastically for the dynamic program slice computation due to number of executed statements that have to be stored and analyzed.

Static program slicing, on the other hand, only analyzes the source code and requires, therefore, less of an overhead during the slice computation. As already stated, it computes conservative program slice, which is an undesirable property. At the same time, the cost for computation is far lower than the dynamic program slicing.

In short, hybrid program slicing uses the cheaper computational properties of static program slicing presented in [15], and at the same time, improves the accuracy of the slice with minimal incremental cost. However, the algorithm presented in [15] is clearly not the choice for larger programs as it is impractical to set breakpoints for every conditional statement unless the user has a very good understanding of the code.

Definitely, the quality of the slice should be progressed towards dynamic program slice. To achieve better slicing algorithms, one of the logical steps is to give the user some choices to choose between size and accuracy. One choice is to reduce the size of the execution trace recording in dynamic program slicing by some means. One of the ways to reduce the recording trace is to use the condensed program such that unwanted statements are executed in the first place. Another approach is to suspend the execution trace recording at a given criterion.

2.4.2 Advantages and disadvantages of hybrid program slicing

As stated earlier, hybrid program slicing uses properties of both static and dynamic program slicing, taking advantage of both approaches:

(a) It allows for a reduction of space and time complexity for the computation.

(b) It computes a slice of higher or equal precision than the static slice.

(c) It helps programmers to carry out multiple executions at a low cost with various values for the same variable to understand behaviour of the source code.

However, to bring the two major program slicing techniques together, hybrid program slicing has to carry out additional work and it poses a few disadvantages:

(a) Additional run time requirements compared to static program slicing.

(b) Not all the hybrid program slicing guarantees the accuracy of dynamic slicing as it depends on the hybrid program slicing algorithm and criterion.

3. Integration of program slicing within CONCEPT

3.1 CONCEPT a comprehension framework

The CONCEPT project was developed to provide an open software comprehension and maintenance framework [47]. The CONCEPT framework is developed using a program slicing tool presented in [32,33]. It provides a platform for the development of advanced program slicing algorithms, slicing related features as well as applications and visualization techniques for both functional and object-oriented programs. The motivation for the project is to provide an open environment that supports a variety of cognitive models and visualization & algorithmic comprehension techniques to guide users during various program comprehension tasks. For example, understanding and analyzing of existing source code and the comprehension of program executions etc. Providing higher levels of visual abstraction might not be enough to guide programmers during the complex tasks such as software comprehension of large software systems [5,21]. Most program comprehension tools represent more a collection of somewhat independent tools that provide certain analysis or visual abstraction approaches. Users (in particular novice users) are frequently confronted with a significant initial learning curve caused by the large set of less intuitive functions and their associated information. Within the software engineering community, well-known concepts of good software design are module cohesion and module coupling. In the CONCEPT environment, we try to overcome limitations of current comprehension tools with respect to their functionality and learnability by applying the concepts of coupling and cohesion on the functional level of program comprehension tools. One of the CONCEPT design goals is to maximize the cohesion and minimize the coupling of the available functionality within the tool.

Functional cohesion means a collection of tools that form from a user and task perspective, a set of coherent functions providing them with the functionality required to master a particular task and its associated information (Figure 6).

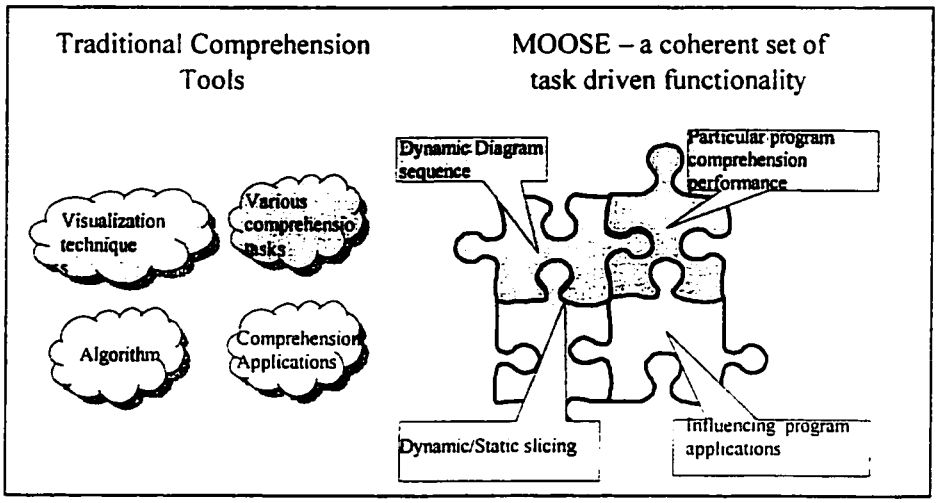


Figure 6: Task-oriented functional cohesion from a user perspective

3.1.1 CONCEPT - Architecture

The CONCEPT architecture (Figure 7) is based on five major components: (1) task and user centred approach that will guide users during comprehension of specific tasks (slicing framework), (2) an algorithmic framework, providing analysis and metric functionality, (3) an application framework that provides a set of applications supporting various comprehension tasks, (4) the visualization support and (5) an underlying repository that provides a communication and interaction interface among all the parts of the environment.

Figure 7: CONCEPT program comprehension framework

The CONCEPT environment was designed with two major goals in mind. The first goal was to provide a suite of tightly integrated tools with a set of coherent functionality. The second goal was to create an open environment that can easily be extended with new tools, algorithms, and applications to meet future demands. These goals are achieved by creating a general framework that consists of several sub-frameworks as illustrated in Figure 8.

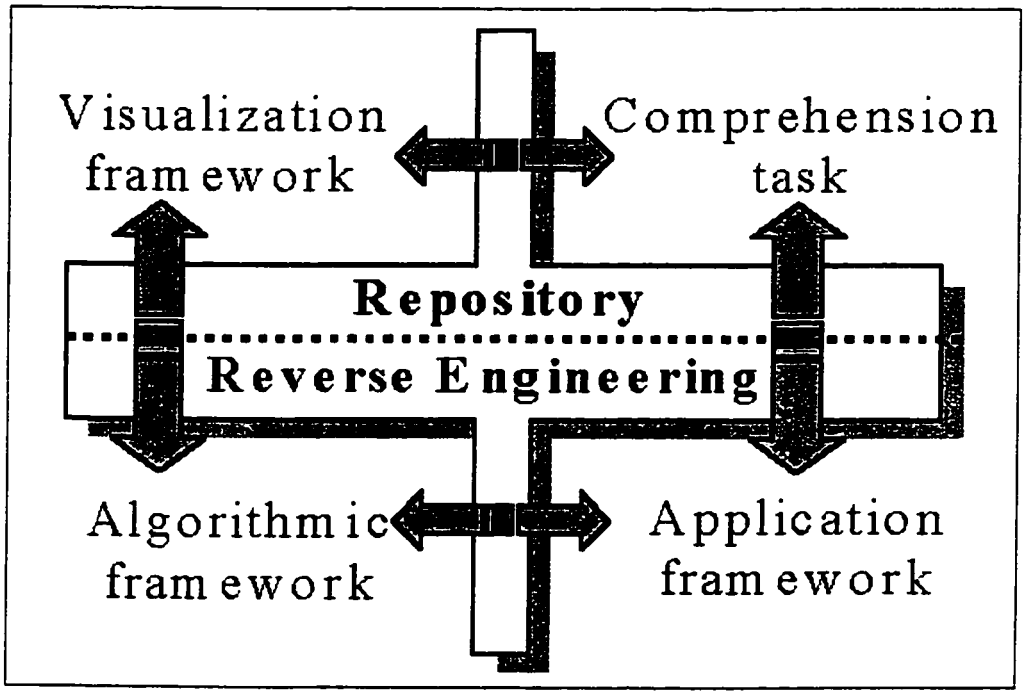


Figure 8: The open CONCEPT architecture with sub-frameworks

Figure 9 shows the abstract system design of the CONCEPT framework. The system design allows the addition of modules without having to modify the overall system as they exchange the data using the *adapter* layer. The new modules such as static and hybrid program slicing algorithms need to use the *adapter* for external data storage and retrieval. Within CONCEPT framework an algorithmic sub-framework has been

implemented that uses program slicing for algorithmic analysis of sub-programs. The algorithmic sub-framework is an integrated part of CONCEPT framework providing an open architecture that can be expanded to use different algorithms utilizing static and dynamic information. The information is derived through reverse engineering and is stored in framework repository. *Parser* layer interfaces the source code through adapter for visualization support within CONCEPT framework.

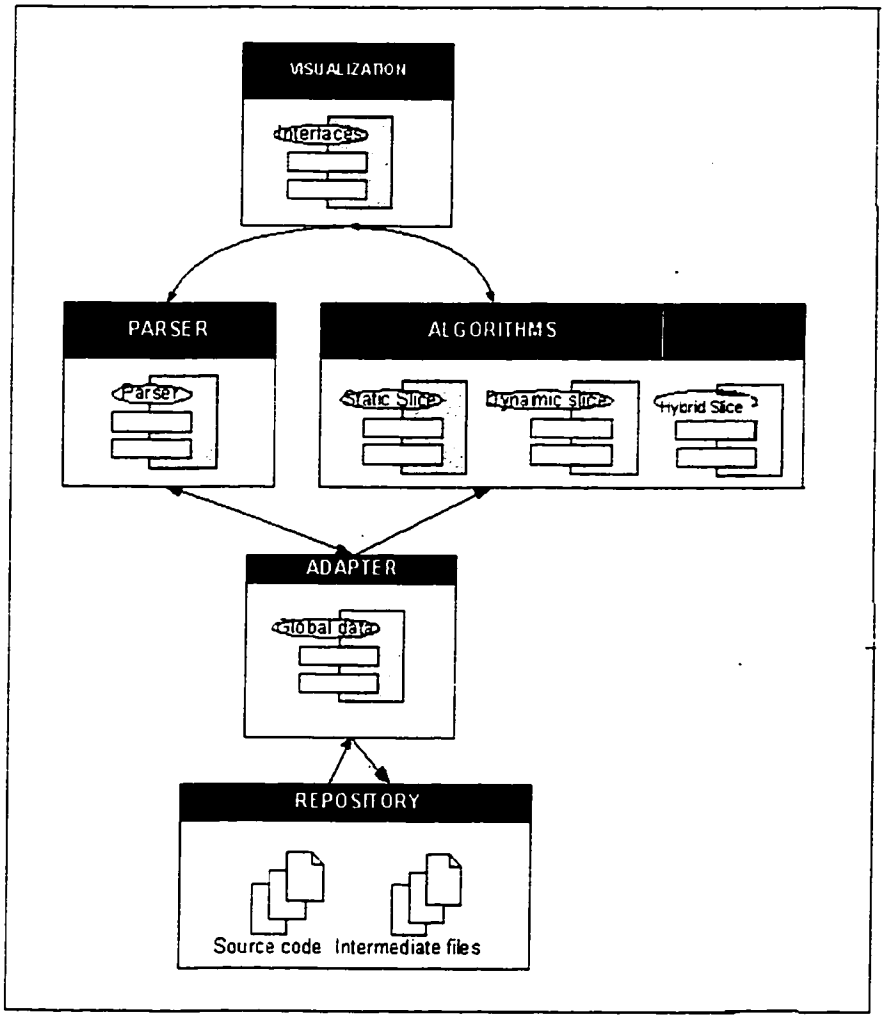


Figure 9: CONCEPT system design

3.1.2 CONCEPT - A user-centered approach

Evidence culled from day-to-day experience tends to indicate that, in most cases, software engineering technology can meet business benefits and requirements and yet, still be quite challenging to use and learn. Many definitions of usability exist, often making usability a confusing concept. Generally speaking, usability of software refers to how easy it is to use and how easy it is to understand. For an inexperienced user, ease of use is coupled closely with ease of learning and does not necessarily imply a high performance in task completion. Whereas experienced users are interested in completing a wide range and number of tasks with minimal obstruction. A usable system would, therefore, be easy to use and learn over time while allowing to gain experience in the process [58]. Despite efforts made by managers to render the transition more "user-friendly", the associated help documentation and training material, although precise and perfectly describing the product, are often delivered in an esoteric and unreadable language. Thus rendering the product and its document inconsistent. This could contribute to the rejection of the product by its users. It also explains a large part of the frequently observed phenomenon of modifying the product or the documentation after it has been deployed.

A typical CONCEPT interactive interface is shown in the following figures.

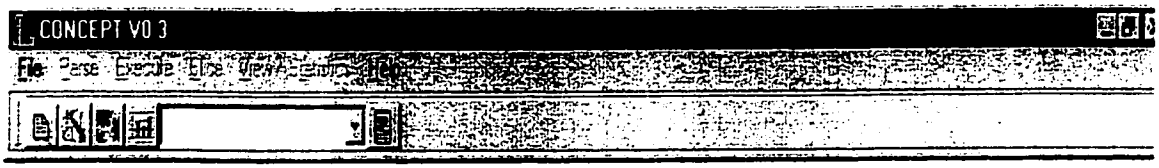
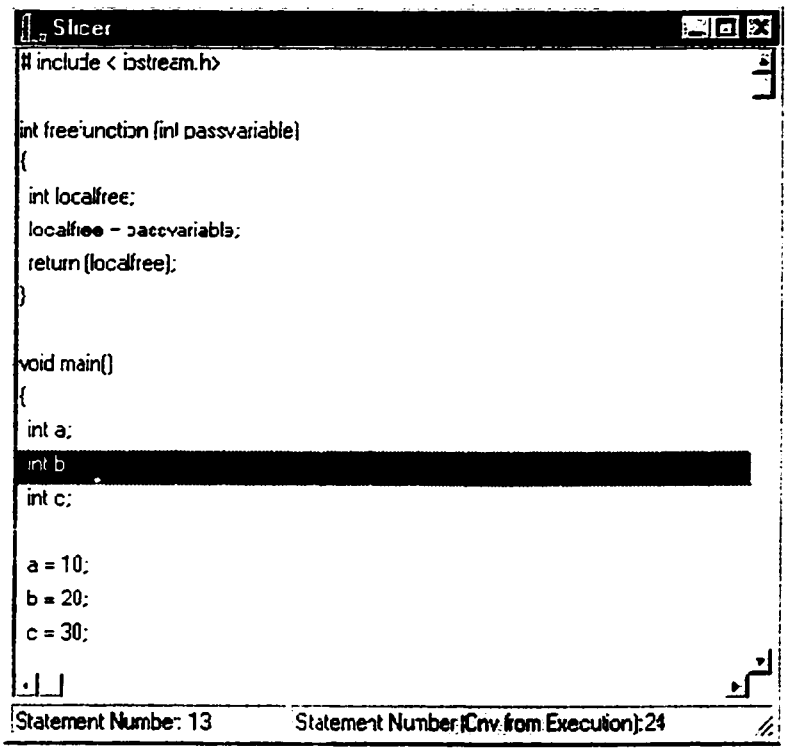


Figure 10 : CONCEPT parent Window -1



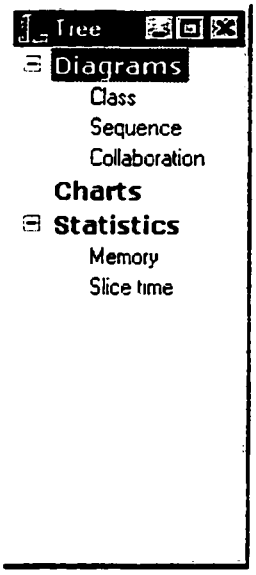


Figure 11: CONCEPT- 2

Figure12: CONCEPT-3

The user first interacts with CONCEPT framework using interface as shown in Figure 10. The user then loads the specific program source code into CONCEPT through File/Open menu options. This inturn will display the source code in the display screen (Figure 11). The user will now have to parse the source code using Parse option on the menu bar or press concerned button on main tool bar. CONCEPT will now create necessary repository files and display screen (Figure 12) to enable user to view various diagrams such as Class diagram (Figure 13), Sequence diagram (Figure 14) and collaboration diagram (Figure 15). User will also be able to create slices (Static, Dynamic, Hybrid and Alternate hybrid) through Slice menu option and respective sub options.

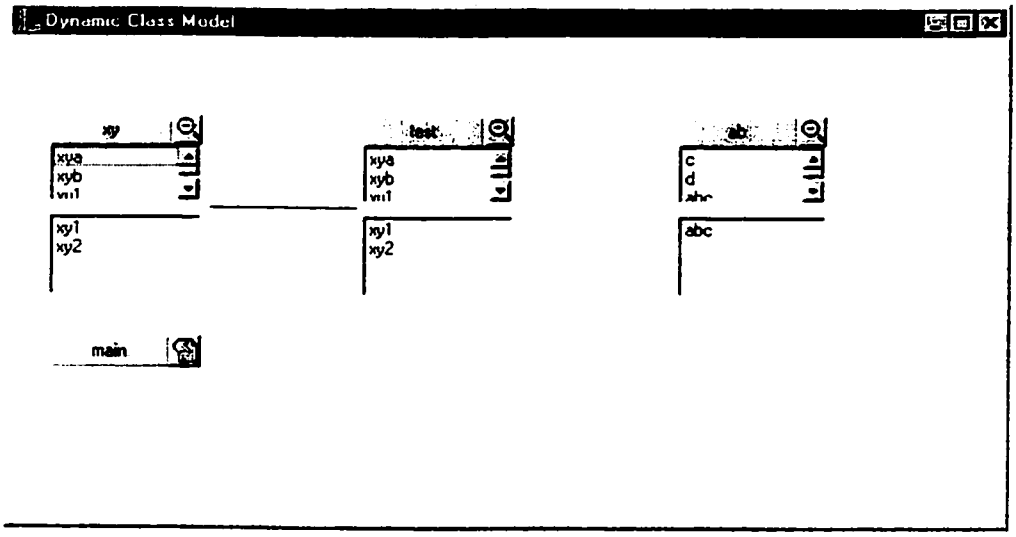


Figure 13: Class diagram as sliced and depicted by CONCEPT

In the CONCEPT class model slice (Figure 13), a class member function will be included in the slice if at least one statement in the member function is the part of the program slice. This also includes any cascading relationship among classes relevant to slice.

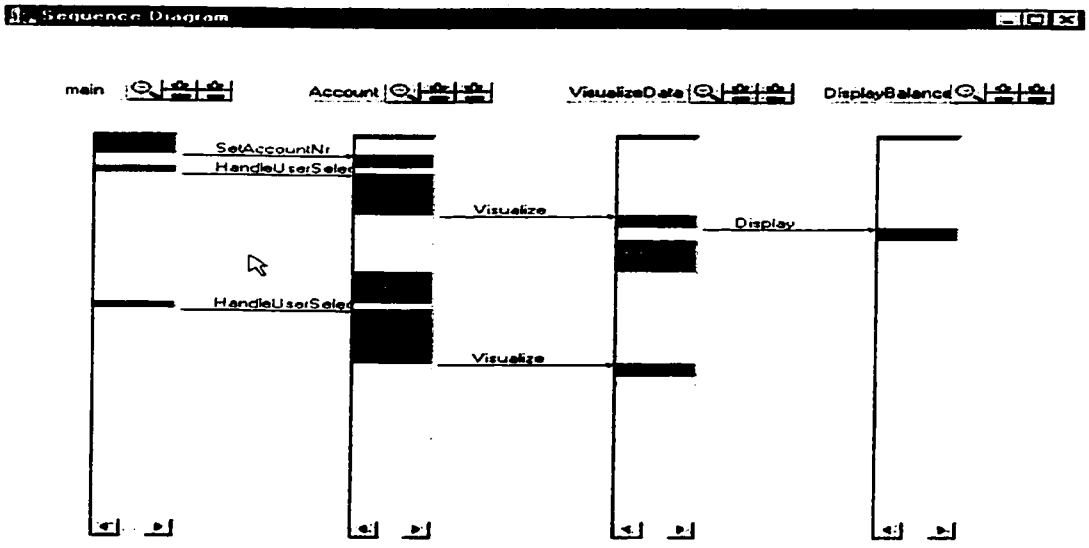


Figure 14: Sequence diagram as depicted by CONCEPT

The CONCEPT framework provides some of the traditional visual abstraction, e.g. call-graph with enhancements in the navigation process and newer approaches for static and dynamic views of the larger object oriented system and their execution. Other abstraction techniques extend the standard UML notation of a class model, sequence and collaboration diagram by applying a reengineering process to derive dynamic sequence diagrams (Figure 14). dynamic collaboration diagrams (Figure 15) and class diagrams (Figure 13).

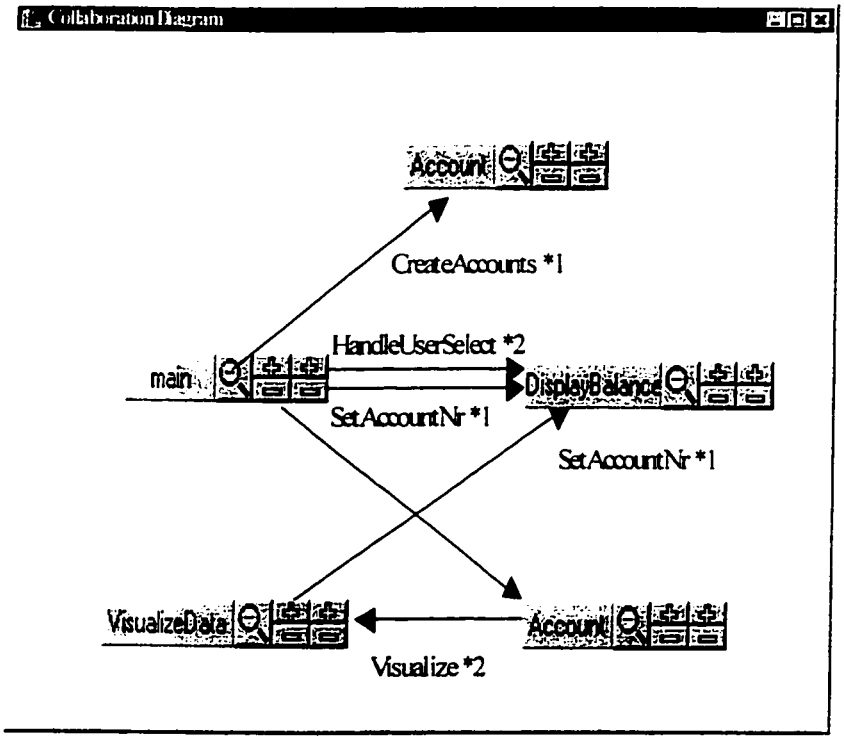


Figure 15: Collaboration diagram as depicted by CONCEPT

4. Alternate approach to hybrid slicing

After considering the advantages and disadvantages of various slicing techniques, which are presented in chapter 2 and summarized in the table below (Figure 16).While CONCEPT currently uses all the three techniques, neverthless, we have proposed and integrated in CONCEPT an alternative approach to hybrid slicing. This approach utilizes slicing algorithms already implemented in the CONCEPT framework and at the same time overcomes the major disadvantages of the hybrid algorithm presented in section 2.4. This new alternate hybrid algorithm reduces the run time overhead required by the original hybrid algorithm.

Static slicing	Dynamic slicing	Hybrid slicing
Advantages: Cheap as no analysis of an execution trace is necessary. Provides a general way to understand the program. No operational profile is necessary. *Disadvantages:* No dynamic execution trace is available for dynamic analysis of program executions. Outputs a large slice.	*Advantages:* Provides information about actual program flow. Facilitates testing and debugging. Outputs a smaller slice. *Disadvantages:* Requires high run time overhead. Requires an operational profile.	*Advantages:* It allows for a reduction of the space and time requirements for the computation. It will help the programmer to carry out multiple executions at a low cost. Outputs a smaller slice. *Disadvantages:* Additional run-time overhead is required as compared to static program slice.

Figure 16: Comparison of various slicing techniques

In this new approach, an executable static slice will first be computed. In the next step, the repository file such as "Executed_statement.txt" will be examined and based on the dynamic information available in this file, the system will highlight the statements in the new alternate hybrid slice that are part of static slice and executed.

The different steps involved in the computation of alternate hybrid slice are enumerated below:

1. The original program will be parsed.

2. An executable static slice will be computed.

3. The static slice will be executed keeping track of executed statements.

4. The static slice will be combined with the executed statements as available in the Executed_Statements.txt file and new alternate hybrid slice will be generated.

5. The slice will be displayed.

The process can be well illustrated on the source code shown in Figure 17.

```
1. void main(){
2. i = 0;
3.  while (i < 2) {
4.    if (i = = 0)
5.          a = 5 ;
6.   else
7.          a = 10;
8.   i++;
   }
}
```

Figure 17: Source code of Sample_1 program

Figure 18 shows the execution trace for the Sample_1 program (Figure 17) combining the data and control dependencies for variable "a". T_x = < 1,2,3,4,5,6,7,8 >

Statement number	Execution count	Code
1	1	void main()
2	1	i = 0;
3	2	while (i < 2)
4	1	if (i = = 0)
5	1	a = 5;
6	2	else
7	2	a = 10;
8	2	i++;

Figure 18: Executed_statements file for Sample_1 program by CONCEPT

Based on the slicing techniques discussed in section 2, the slices of the Sample_1 program (Figure 17) using different slicing approaches are shown in Figure 19.

Static slicing	Dynamic slicing	Alternate hybrid slicing

1. void main(){
2. i = 0;
3. while (i < 2)
{
4. if (i = = 0)
5. a = 5 ;
6. else
7. a = 10;
8. i++;
}
}
	1. void main(){
	2. i = 0;
	3. while (i < 2)
	{
	4. if (i = = 0)
	6. else
	7. a = 10;
	8. i++;
	}
	}
		1. void main(){
		2. i = 0;
		3. while (i < 2)
		{
		4. if (i = = 0)
		5. a = 5 ;
		6. else
		7. a = 10;
		8. i++;
		}
		}

Figure 19: Slices obtained for Sample_1 program using different techniques

It can well be observed from Figure 19 that the slice created by the alternate hybrid approach is the same as the one computed by the static slicing approach and in this case there has not been any tangible gain by using this technique. The dynamic slicing approach will provide in this case, a smaller and more precise slice for a given slicing criterion compared to the static or alternate hybrid slicing algorithms. However, for the Sample_2 program (Figure 20) the alternate hybrid slicing algorithm will compute a smaller slice than the static algorithm. The slice will be identical to the dynamic slice.

```
1. void main(){
2. i = 0;
3.    if (i = = 0)
4.          a = 5 ;
5.   else
6.          a = 10;
7.  i++;
}
```

Figure 20: Source code for Sample_2 program

Figure 21 shows the execution trace for the sample_2 program shown in Figure 20.

T_r = < 1,2,3,4,7 >

Statement number	Execution count	Code
1	1	void main()
2	1	i = 0;
3	1	if (i = = 0)
4	1	a = 5;
7	1	i ++;

Figure 21: Executed_statements file for Sample_2 program

Once we compute the alternate hybrid slice, the traditional static and dynamic slice for the for Sample_2 program, we can compare the different slices (Figure 22).

Static slicing	Dynamic slicing	Alternate hybrid slicing

1. void main(){
2. i = 0;
3. if (i = = 0)
4. a = 5 ;
5. else
6. a = 10;
7. i++;
}
	1. void main(){
	2. i = 0;
	3. if (i = = 0)
	4. a = 5 ;
	7. i++;
	}
		1. void main(){
		2. i = 0;
		3. if (i = = 0)
		4. a = 5 ;
		7. i++;
		}

Figure 22: Slices obtained for Sample_2 program using different techniques

In Figure 22, one can observe that the alternate slicing approach computes the same slice as the dynamic slicing approach, however, without requiring the recording of a complete dynamic execution trace and thereby reducing the required runtime overhead.

4.1 Advantages and disadvantages of the alternate slicing approach

(a) The computation of the alternate program slicing approach is relatively rapid (compared to dynamic program slicing) as only the static analysis of the source code is conducted and no detailed analysis of program execution is required.

(b) The alternate hybrid slice assists the user to gain a general understanding of the

program parts that contribute to the computation of a selected function and a particular program execution.

(c) For specific program flow and slicing criterion discussed above, this approach may compute a much smaller slice than that computed by static slicing approach.

Nevertheless, the alternate program slicing has some drawbacks, which are elaborated below:

(a) For programs containing dynamic language constructs like arrays, pointers, aliases, etc., the alternate hybrid slicing has to make some conservative assumptions with respect to their run time contribution similar to the one made by static slicing and therefore, frequently computes larger slice than dynamic slicing algorithm.

(b) As this approach does not take into consideration all the information regarding execution trace, the slice created may not provide all the information necessary for a detailed analysis of program execution.

5. Conclusions and future work

5.1 Conclusions

In this report, we have presented and discussed general program slicing techniques which include static, dynamic and hybrid slicing techniques. A new alternate technique to hybrid slicing has also been proposed, discussed and integrated in the CONCEPT project.

The results of the static and alternate hybrid slicing obtained through CONCEPT are very encouraging and produce almost correct slice as anticipated for all language constructs found in object oriented programming language.

5.2 Future work

As part of the future work, it is proposed to include *criterion-based hybrid slicing algorithm* in the CONCEPT framework and also to combine criterion-based slicing with the new alternative hybrid slicing approach. New slicing related concepts as well as new visualization techniques should be derived to take advantage of the algorithms. In addition, integration of *forward* program slicing algorithms within hybrid slicing framework has to be developed to investigate additional usability aspects of CONCEPT environment.

5.3 Analytical analysis to be conducted

It is now planned to conduct an experimental analysis of the various slicing algorithms presented in this report using a variety of programs to illustrate the benefits and

limitations of each approach.

It is anticipated that hybrid program slicing run time will be greater than the static program slicing, which may be acceptable to achieve better accuracy. The hybrid program slice run time will also be much lower than dynamic program slicing because many non-contributing statements have been removed using static program slicing.

However, further analytical experiments have to be carried out to compare the hybrid program slicing algorithm with static and dynamic program slicing. Through this analytical analysis, certain properties will be evaluated which will allow further study of slicing algorithms in the context of their accuracy, limitations, time and space complexity and behavior for different types of programs and program executions. The properties we plan to use are the following:

5.3.1 Correctness

Correctness of the slice is defined for each of the language constructs that are handled properly by the algorithm. This property has to be tested with sample programs with different language constructs, for example, conditional statements, loops, class constructs etc. for the proposed algorithms with the existing algorithms.

5.3.2 Accuracy

The goal of slicing is to compute the smallest executable subprogram from the original program. This property is referred to as *accuracy* of the program slicing algorithm. Again, this property has to be tested with sample programs with the new algorithm as well as with other existing algorithms.

5.3.3 Time complexity

Time complexity is dependent on the execution length and size of the program to be sliced. This property can be analytically verified using the computation time for different algorithms with same slicing criterion.

5.3.4 Space complexity

Space complexity is dependent on the amount of data used for the analysis in any algorithm at any one time. This property can be verified by memory requirements during the computation of slice using different algorithms with the same slicing criterion.

Further experiments need to be carried out in the above category with the same conditions across the experiments. In other words, the comparison with different algorithms shall use the same sample program, slicing criterion and where applicable, same execution length. This data could be useful in optimizing the slicing algorithms within the CONCEPT framework.

Program	Computation Time for Hybrid slicing	Computation Time for static slicing	Computation Time for dynamic slicing
Program1	*X1* seconds	*Y1* seconds	*Z1* seconds
...	...	...	.....
Program*x*	*Xx* seconds	*Yx* seconds	*Zx* seconds

Figure 23: Computation time for slicing the sample programs

Program	Memory resources for Hybrid slicing	Memory resources for static slicing	Memory resources for dynamic slicing
Program1	*X1* kB	*Y1* kB	*Z1* kB
...	...	...	.....
Programx	*Xx* kB	*Yx* kB	*Zx* kB

Figure 24: Memory resources for slicing the sample programs

Using the above information, CONCEPT users can determine the particular type of slicing algorithm to be used in a the particular case, based on the time and space availability of each algorithm.

References

1. Agrawal, H., "Towards automatic debugging of computer programs", *Technical Report SERTC-TR-40-P,* Purdue University, 1989.

2. Agrawal, H. and Horgan, J, "Dynamic program slicing", *In Proceedings of the ACM SIGPLAN'90 Conference on Programming Language Design and Implementation,* SIGPLAN Notices, 25(6), pp. 246-256, 1990.

3. Agrawal, H., DeMillo, R., and Spafford, E., "Debugging with dynamic slicing and backtracking", *Software – Practice and Experience*, 23(6), pp. 589-616, 1993.

4. Agrawal, H., "On Slicing programs with jump statements", *In Proceedings of the ACM SIGPLAN'94 Conference on Programming Language Design and Implementation*, pp. 112-135, 1994.

5. Binkley D. and Gallagher K., "Program Slicing", *Adv. in Computers, 43, Academic Press*, pp. 1-52, 1996.

6. Chen J.L., Wang F.J., and Chen Y.L., "Slicing object oriented Programs", *In Proceedings of the APSEC'97*, pp. 395-404, Hongkong, China, December 1997.

7. Cheng, J., "Slicing concurrent programs a graph-approach", *In Proceedings of the First International Workshop on Automated and Algorithmic Debugging (1993), P. Fritzson, Ed., Vol. 749 of Lecture Notes in Computer Science, Springer-Verlag*, pp. 232-245, 1993

8. Choi, J.-D., Miller. B., and Netzer, R., "Techniques for debugging parallel programs with flowback analysis", *In ACM Transactions on Programming Languages and Systems,* 13(4), pp. 491-530, 1991.

9. Desmarais, M.C., Liu, J. "Exploring the applications of user-expertise assessment for intelligent interfaces". *In Proceedings of InterCHI'93, Bridges between worlds (Amsterdam, 24-29 April)*, pp. 308-313, 1993.

10. Duesterwald, E., Gupta, R., and Soffa, M., "Distributed slicing and", *In Proceedings of the fifth workshop on Languages and Compilers for Parallel Computing, New Haven, Connecticut,* partial re-execution for distributed programs pp. 329-337, 1992.

11. Ferrante, K., Ottenstein, K, and Warren J. "The Program Dependence Graph and its use in Optimization", *In ACM Transactions on Programming Languages and Systems*, 9(5), pp.319-349, 1987.

12. Gallagher, K. and Lyle, J., "Using program slicing in software maintenance", *IEEE Transactions on Software Engineering*, 17(8), pp. 751-761, August 1991.

13. Gopal, R., "Dynamic program slicing based on dependence relations", *In Proceedings of the Conference on Software Maintenance,* pp. 191-200, 1991.

14. Gupta, R., Harrold, M., and Soffa, M., "An approach to regression testing using slicing", *In Proceedings of the Conference on Software Maintenance*, pp. 299-306, 1992.

15. Gupta, R., Soffa, M. and Howard J., "Hybrid Slicing: Integrating Dynamic Information with Static Analysis", *In ACM Transactions on Software Engineering and Methodology*, 6(4), pp. 370-397, October 1997.

16. Harman M. and Gallagher K., editors, *Journal of Information and Software Technology Special Issue on Program Slicing*, volume 40. Elsevier, 1998.

17. Hart, J.M, "Experience with Logical code analysis in software reuse and reengineering", *In AIAA computing in Aerospace*, 10, pp. 1243-1262, San Antonio, Texas, March 28-30, 1995.

18. Hendley R., et al.: "Case Study - Narcissus: Visualizing Information", *In Proceedings of the IEEE Information Visualization 95*, pp. 90-96, 1995.

19. Horwitz S., Reps, T., and Binkley, D., "Interprocedural slicing using dependence graphs", *In ACM Transactions on Programming Languages and Systems*, 12(1), pp. 26-61, 1990.

20. Horwitz, S. and Reps, T., "The use of program dependence graphs in software engineering", *In Proceedings of the 14th International Conference on Software Engineering*, pp. 392-411, Melbourne, Australia, 1992.

21. Kamkar M., "Interprocedural Dynamic Slicing with Applications to Debugging and Testing", *Ph.D. Thesis, Linköping University*, 1993.

22. Kamkar M., Fritzson, P., and Shahmehri, N., "Three approaches to interprocedural dynamic slicing", *Microprocessing and Microprogramming*, (38), pp. 625-636, 1993.

23. Karanth B.A "Utilizing notation of removable block to enhance program slicing algorithm", *Master thesis ,Concordia University, Montreal.* June 2001.

24. Korel, B. and Laski, J., "Dynamic program slicing", *In Proc. Letters*, 29(3), pp. 155-163, Oct. 1988.

25. Korel, B., "PELAS – Program Error Locating Assistant System", *In IEEE Transactions on Software Engineering*, 14(9), pp. 1253-1260, Sept. 1988.

26. Korel, B. and Laski, J., "Dynamic program slicing", *In Proc. Letters*, 29(3), pp. 187-195, 1990.

27. Korel, B. and Ferguson, R., "Dynamic slicing of distributed programs", *Applied Mathematics & Computer Science Journal*, 2(2), pp. 199-215, 1992.

28. Korel, B., "Identifying faulty modifications in software maintenance", *Proceedings of the 1st International Workshop on Automated and Algorithmic Debugging*, pp. 341-356, Linköping, Sweden, 1993.

29. Korel, B. and Yalamanchili, S., "Forward Derivation of Dynamic Slices". *In Proceedings of the International Symposium on Software Testing and Analysis*, pp. 66-79, Seattle, 1994.

30. Korel, B., "Computation of Dynamic Slices for Programs with Arbitrary Control-flow", *The 2nd International Workshop on Automated and Algorithmic Debugging*, pp. 1-41, St. Malo, France, 1995

31. Korel, B., "Computation of dynamic slices for unstructured programs", *In IEEE Transactions on Software Engineering*, 23(1), pp. 17-34, 1997.

32. Korel B. and Rilling, J., "Application of Dynamic Slicing in Program Debugging", *Third International Workshop on Automated Debugging (AADEBUG'97)*, pp. 59-74, Linköping, Sweden, May 1997.

33. Korel, B. and Rilling, J., "Program Slicing in Understanding of Large Programs". *In Proceedings of the 6th IWPC '98*, pp. 145-152, Ischia, Italy, June 1998.

34. Korel, B. and Rilling, J., "CASE and Dynamic Program Slicing in Software Maintenance", *Special issue of the International Journal of Computer Science and Information Management*, (June 1998)

35. Krishnaswamy A., "Program slicing: An application of object-oriented Program Dependency Graphs", *Technical report TR94-108, Computer Science Department, Clemson University*, 1994.

36. Kung D. Gao J. et al., "Developing an object-oriented software testing and maintenance environment"; *In Commmunications of the ACM*, Vol. 38, Issue 10 (1995), pp. 75-87.

37. Larsen L.D. and Harrold M.J., "Slicing Object oriented software", *Proceeding of the 18th International conference on Software engineering*, March,1996.

38. Law R.C.H., "Object-Oriented Program Slicing" *Ph.D. thesis, University of Regina, Regina, Canada*, 1994.

39. Lyle, J. and Weiser, M., "Experiments on slicing-based debugging tools", *Proceedings of the 1st Conference on Empirical Studies of Progrmes.*", pp. 187-197, 6/ 1986.

40. Mayerhauser A, Vans A. M., "Program Understanding Behavior During Adaptation of Large Scale Software", *Proceedings of the 6th International Workshop on Program Comprehension IWPC* '98, pp. 164-172, Ischia, Italy, June 1998.

41. Ottenstein, K., and Ottenstein, L., "The program dependence graph in a software development environment", *In Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments, SIGPLAN,19(5),* pp.177-184, 1984.

42. Reps, T., and Horwitz S., "Semantics-based program integration", *In Proceedings of the Second ACM European Symposium on Programming (ESOP'88),* pp. 133-145, Nancy, France, March 1988.

43. Reps, T. and Bricker, T., "Semantics-based program integration illustrating interference in interfering versions of programs", *In Proceedings of the Second International Workshop on Software Configuration Management,* pp. 46-55, Princeton, New Jersey, Oct. 1989.

44. Richner, T. and Stéphane Ducasse, "Recovering High - Level Views of Object - Oriented Applications from Static and Dynamic Information", *In Proceedings of ICSM'99, September. IEEE Computer Society Press*, pp. 13-22, 1999.

45. Rothermel G. and Harrold, M. J., "Selecting tests and identifying test coverage requirements for modified software", *In Proceedings of the ACM SIGSOFT International Symposium on Software Testing and Analysis,* pp. 169-184, August 1994.

46. Rilling, J., "Maximizing functional cohesion of comprehension environment by Integrating user and task knowledge", *Computer Science Department, Concordia University, Montreal.*

47. Rilling, J., "Investigation Of Dynamic Slicing And Its Application In Program Comprehension", *Ph.D. Thesis; Illinois Institute of Technology*, July 1998.

48. Rilling, J. and Seffah, A., "Enhancing the Usability and Learnability of Software Visualization Techniques through Task Wizards and Software Agents", *2001 International Conference on Imaging Science, Systems, and Technology (CISST'200),* June 25-28, 2001, Las Vegas, Nevada, USA.

49. Schoenig S. and Ducass'e. M., "A hybrid backward slicing algorithm producing executable slices for Prolog". *In Proceedings of the 7th Workshop on Logic Programming Env.* Pages 41--48, Portland, USA, December 1995.

50. Seffah, A., "Training Software Developers in Critical Skills" *IEEE Software Magazine,* June 1999.

51. Seffah, A. and Rilling, J., "Investigating the Relationship between Usability and Conceptual Gaps for Human-Centric CASE Tools", *IEEE Symposium on Human-Centric Computing Languages and Environments, Stresa,* Italy, September 2001.

52. Shimomura T., "The program slicing technique and its application to testing, debugging and maintenance", *Journal of IPS of Japan, 9(9)*, pp. 1078-1086, Sept. 1992.

53. Steindl, C., "Intermodular slicing of object-oriented programs". *In International Conference on Compiler Construction (CC'98)*, 1998.

54. Steindl, C., "Static Analysis of Object-Oriented Programs", *9th ECOOP Workshop for Ph.D. Students in OO-Programming, Lisbon, Portugal*, June 14, 1999.

55. Storey, M., Wong, K. and Muller, H.A., "How do program understanding tools affect how programmers understand programs?" *Proceedings of the Fourth Working Conference on Reverse Engineering*, p. 12-21, Netherlands, October 1997.

56. Tip F., "A survey of program slicing techniques", *Journal of Programming Languages, 3(3)*, pp. 121-189, September 1995.

57. Tip F., Choi J.D., Field J. and Ramalingam G. "Slicing Class Hierarchies in C++," *Proceedings of the 11th Annual conference on Object-Oriented Programming, systems, Languages, and Applications*, pp.179-197, October,1996.

58. Walker, R.J., Murphy, G.C., Freeman-Benson, B., Wright, D., Swanson, D., Isaak, J., "Visualizing Dynamic Software System Information through High-level Models". *Proceedings of OOPSLA'98, pp. 271-283*, Vancouver, October 1998. Published as SIGPLAN Notices 33(10), October 1998.

59. Weiser M., "Programmers use slices when debugging", *Communications of ACM, 25*, pp. 446-452, 1982.

60. Weiser M, "Program slicing" ,*IEEE Transactions on Software Engineering 10(4)* , pp352-357,1984

61. White L. and Leung, H., "Regression testability", *IEEE Micro*, pp. 81-85, April 1992.

62. Zhao J., Cheng J. and Ushijima K, "Static Slicing of Concurrent Object-Oriented Programs", *Proceedings of the 20th IEEE Annual International Computer Software and Applications conference* , pp.312-320, August, 1996, IEEE Computer Society Press.

63. Zhao.J, " Dynamic Slicing of Object-Oriented Programs," *Technical-Report SE-98-119*, pp.17-23, Information Processing Society of Japan, May,1998.