Program Slicing for Refactoring: Static Slicer using Dynamic Analyser

Amogh Katti
Poojya Doddappa Appa College of Engineering, Gulbarga, Karnataka, India.

Sujatha Terdal
Poojya Doddappa Appa College of Engineering, Gulbarga, Karnataka, India.

ABSTRACT

Refactoring is the process of changing the code of the software such that its internal design is improved without altering its observable behavior. Method Extraction is the process of separating out a subset of method's statements into another method and replacing their occurrence in the original method with a call to this new method. Method extraction is a classical problem to improve the modularity of the system and is used in extracting methods from long procedural programs. It can also be used in extracting aspects from object oriented code. Thus it makes the software easier to understand, maintain and reusable. In the earlier days of method extraction, programmer selected a random set of statements for extraction which was made more sensible by specifying the variables of interest and separating the statements concerning them into a method. Thus, program slicing became part of method extraction. Many slicing algorithms exist in the literature; they first convert the program into some alternative representation and then apply some correctness preserving transformations on it to produce slice and its complement. This process was identified to be expensive and an algorithm was proposed to act directly on the source code. It statically analyzes the source code to produce the slice but fails to handle dynamic constructs like aliasing and polymorphism effectively. To overcome this limitation we propose a new slicing algorithm that dynamically analyzes source code to produce static slices. It exploits the behavior preservation requirement of refactoring and uses the data collected during testing, which we perform prior to refactoring, for slicing. This algorithm suits better to the refactoring domain.

General Terms

Refactoring, Extract Method Refactoring, Program Slicing

Keywords

Program Slicing for Refactoring, Static Slicer using Dynamic Analyser

1. INTRODUCTION

Refactoring is the process of changing a software system in such a way that it does not alter the external behavior of the code yet improves its internal structure. Refactoring Improves the design of software and it makes software easier to understand.

The extract method refactoring takes a set of statements out of a method and places them inside a new method replacing the statements' occurrence in the old method with a call to this new method. This refactoring has many benefits like increased comprehension, maintenance and reuse.

The concept of program slicing was originally introduced by Mark Weiser [1]. He claimed a slice to be the mental abstraction people make when they are debugging a program. A slice consists of all the statements of a program that may affect the values of some variables in a set V at some point of interest p. The pair (p,V) is referred to as a slicing criterion.

Many slicing algorithms have been proposed in the literature and used in their original or slightly modified form for method extraction [2, 3, 4, 5, 7]. All these algorithms first convert the program into some alternative representation and then apply some correctness preserving transformations on it to produce slice and its complement. Converting a program into an alternative form and then extracting slice is quite heavy in terms of both computation and resource usage.

This was identified and an algorithm was proposed, by Mathieu Verbaere using inference rules in [6] that acts directly on the source code. This algorithm statically analyzes the source code and does not handle dynamic constructs like aliasing and polymorphism effectively.

To overcome this limitation we propose a new slicing algorithm that dynamically analyzes the source code to produce static slices. As testing is an inalienable part of refactoring (we test the code to observe behavior, refactor and test it again to make sure that the refactoring has not altered the behavior), we can collect data usage details at runtime during this testing. We use multiple test cases to make sure that all program statements have been executed, which is required as we need to collect data usage details in the whole program for computing static slices. This collected data is analyzed to find data dependencies and then we collect program statements/predicates that form the slice. Since this algorithm exploits the testing performed prior to refactoring and does not use any program representation it best suits refactoring.

2. BACKGROUND

In this section we discuss refactoring, dynamic data flow analysis, aspects and AspectJ and program slicing. We will provide a few definitions and customize them if necessary to suit our approach, discuss the concepts necessary to understand the slicing algorithms and refactoring and our slicing algorithm's implementation concerns.

2.1 Refactoring

"Refactoring is the process of changing a software system in such a way that it does not alter the external behavior of the code yet improves its internal structure", [8]. According to Flower, [8], "Refactoring Improves the Design of Software and Refactoring Makes Software Easier to Understand".

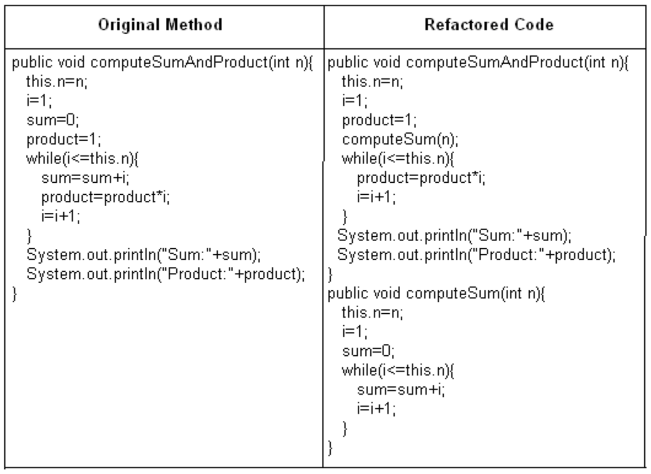
A few refactorings are:

- **Composing Methods:** Extract Method, Inline Method, Inline Temp, Replace Temp with Query, etc.
- **Moving Features Between Objects:** Move method, Move Field, Extract Class, Inline Class, etc.
- **Organizing Data:** Self Encapsulate Field, Replace Data Value with Object, Change Value to Reference, etc.
- **Simplifying Conditional Expressions:** Decompose Conditional, Consolidate Conditional Expression, Remove Control Flag, etc.
- **Making Method Calls Simpler:** Rename Method, Add Parameter, Remove Parameter, etc.
- **Dealing with Generalization:** Pull up Field, Pull up Field, Pull up Constructor Body, etc.
- **Big Refactorings:** Tear Apart Inheritance, Convert Procedural Design to Objects, Separate Domain From Presentation and Extract Hierarchy.

2.1.1 *The "extract method" refactoring*

The extract method refactoring takes a set of statements out of a method and places them inside a new method replacing the statements' occurrence in the old method with a call to this new method. For example, Fig 1 shows a method, computeSumAndProduct(int n), that calculates the sum and product of first n natural numbers. We can extract the statements computing the sum into the method computeSum(int n) and replace the sum computation in the former procedure with a call to this new method.

This refactoring has many benefits like increased comprehension, maintenance and reuse.



Original Method

```
public void computeSumAndProduct(int n){
   this.n=n;
   i=1;
   sum=0;
   product=1;
   while(i<=this.n){
      sum=sum+i;
      product=product*i;
      i=i+1;
   }
   System.out.println("Sum:"+sum);
   System.out.println("Product:"+product);
}
```

Refactored Code

```
public void computeSumAndProduct(int n){
   this.n=n;
   i=1;
   product=1;
   computeSum(n);
   while(i<=this.n){
      product=product*i;
      i=i+1;
   }
  System.out.println("Sum:"+sum);
  System.out.println("Product:"+product);
}
public void computeSum(int n){
   this.n=n;
   i=1;
   sum=0;
   while(i<=this.n){
      sum=sum+i;
      i=i+1;
   }
}
```

Fig 1 An example for Extract Method Refactoring

2.2 Dynamic Data Flow Analysis

Dynamic data flow analysis is used in software testing to detect data flow anomalies: define-define, define-undefine and undefine-reference. However, we are interested in performing dynamic data dependency analysis as part of our program slicer.

Andrew Cain, Jean-Guy Schneider, Doug Grant and Tsong Yueh Chen defined a set of requirements any data analysis approach must support in order to perform a complete analysis of programs written in modern object-oriented programming languages like Java, in [9]:

1. The approach must allow at least the tracking of actions for the definition, reference and destruction of all variables under investigation.

2. The approach must be able to handle any type of variable, independent of scope, type or visibility.

3. The approach must support targeted analysis of source, thereby allowing analysis of individual parts of a system and also allowing analysis of systems that use third party components.

4. The output generated by the approach must enable programmers to identify the location and type of any anomalies produced.

5. The approach must enable automated analysis.

For Java there are two ways that we can retrieve the required information at runtime: program transformation or debugging services. Program transformation can be performed at a number of levels, either by inserting probes into the original source, providing a compiler which produces instrumented byte code, or by instrumenting byte code produced from an existing compiler. Alternatively the debugging services provided in the Java Platform Debugger Architecture can be used to watch for access and modification of Java fields.

We discuss the pros and cons of each of the above dynamic data flow analysis approaches and make a choice for our slicing algorithm implementation in the implementation section.

2.3 Aspects and AspectJ

2.3.1 *Aspects*

Cross-cutting concerns are aspects of a program which affect (crosscut) the main concerns. These concerns often cannot be cleanly decomposed from the rest of the system in both design and implementation and result in either scattering or tangling of the program or both [10, 11]. Here the code is scattered or tangled, making it harder to understand and maintain. It is scattered when one concern (like logging) is spread over a number of modules (e.g., classes and methods). That means to change logging can require modifying all affected modules. Modules end up tangled with multiple concerns (e.g., account processing, logging, and security). That means changing one module entails understanding all the tangled concerns.

For example, consider a banking application for transferring an amount from one account to another:

```
void transfer(Account fromAccount, Account toAccount, int amount) throws Exception
{
  if (!getCurrentUser().canPerform(OP_TRANSFER))   {
    throw new SecurityException();
  }
  if (amount < 0)    {
    throw new NegativeTransferException();
  }
  Transaction tx = database.newTransaction();
  try  {
     if (fromAccount.getBalance() < amount)   {
       throw new InsufficientFundsException();
     }
     fromAccount.withdraw(amount);
     toAccount.deposit(amount);

     tx.commit();
     systemLog.logOperation(OP_TRANSFER, fromAccount, toAccount, amount);
  }
  catch(Exception e) {
     tx.rollback();
     throw e;
  }
}
```

In the above example other interests have become tangled with the basic functionality of withdrawing from an account and then transferring to the other (sometimes called the business logic concern). Transactions, security, and logging all exemplify cross-cutting concerns. Also consider what happens if we suddenly need to change (for example) the security considerations for the application. In the program's current version, security-related operations appear scattered across numerous methods, and such a change would require a major effort.

Therefore, we find that the cross-cutting concerns do not get properly encapsulated in their own modules. This increases the system complexity and makes evolution considerably difficult. Aspect Oriented Programming (AOP) attempts to solve this problem by allowing the programmer to express cross-cutting concerns in stand-alone modules called aspects. Aspects can contain advice (code joined to specified points in the program) and inter-type declarations (structural members added to other classes).

One more application of AOP is it can be used for program analysis. It provides constructs to intercept program execution and we make use of them to intercept variable accesses for dependency analysis prior to slicing.

2.3.2 AspectJ Language Constructs [12]

To support AOP, AspectJ adds to the Java language the concepts:

Joinpoints: Points in a program's execution. For example, joinpoints could define calls to specific methods in a class

Pointcuts: Program constructs to designate joinpoints and collect specific context at those points

In particular, Field-access pointcuts are of interest to us and they capture read and write access to a class's field. Table 1 shows some examples.

Advices: Code that runs upon meeting certain conditions. For example, an advice could log a message before executing a joinpoint Pointcut and advice together specify weaving rules. Aspect, a class-like concept, puts pointcuts and advices together to form a crosscutting unit. The pointcuts specify execution points and the weaving rule's context, whereas the advices specify actions, operations, or decisions performed at those points. One can also look at joinpoints as a set of events in response to which an advice is executed.

Table 1. Field Access Pointcuts

Pointcut	Description
get(PrintStream System.out)	Execution of read-access to field out of type PrintStream in System class
set(int MyClass.x)	Execution of write-access to field x of type int in MyClass

2.4 Program Slicing

The concept of program slicing was originally introduced by Mark Weiser [1]. He claimed a slice to be the mental abstraction people make when they are debugging a program. A slice consists of all the statements of a program that may affect the values of some variables in a set V at some point of interest p. The pair (p,V) is referred to as a slicing criterion.

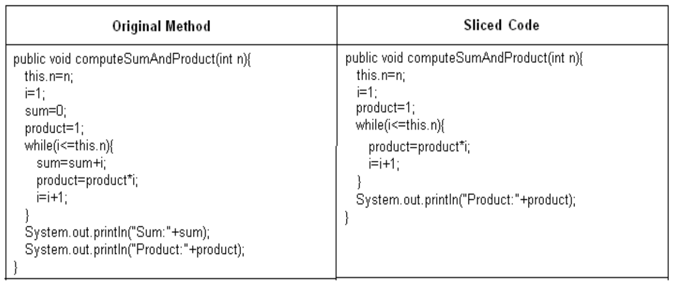
We use a slightly modified definition of a slice: instead of specifying a line number as part of slice criterion, we take the end of the method as point of interest as it would make sense for the extract method refactoring. So we have only the set of variables which are of interest to us as the slice criterion i.e. (V).

Since Weiser introduced the concept of slicing, many different notions of program slicing and approaches to compute them have been proposed. Tip makes an extensive survey of these different types of slices, algorithms to compute them, language constructs they support and applications of slicing in [13].

2.4.1 Static Vs Dynamic Slices

There are two varieties of program slicing: Static and Dynamic. In static slicing, static analysis of the source code is carried out to compute the slice. Whereas in dynamic slicing, we execute the source code for a particular input and compute the slice that is valid only for this particular execution. Slicing criterion for dynamic slicing includes a specific input for which the slice is valid along with other components in static slicing criterion. Figures 2 and 3 below show an example for static and dynamic slices respectively.

For n = -3, the statement of the then branch is not relevant and can be removed. The semicolon is kept in order to preserve the syntax correctness.



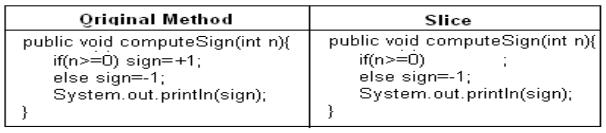
Original Method

```
public void computeSumAndProduct(int n){
   this.n=n;
   i=1;
   sum=0;
   product=1;
   while(i<=this.n){
      sum=sum+i;
      product=product*i;
      i=i+1;
   }
   System.out.println("Sum:"+sum);
   System.out.println("Product:"+product);
}
```

Sliced Code

```
public void computeSumAndProduct(int n){
   this.n=n;
   i=1;
   product=1;
   while(i<=this.n){
      product=product*i;
      i=i+1;
   }
   System.out.println("Product:"+product);
}
```

Fig 2 Static slice for the criterion ({product})



Original Method

```
public void computeSign(int n){
     if(n>=0) sign=+1;
     else sign=-1;
     System.out.println(sign);
}
```

Slice

```
public void computeSign(int n){
     if(n>=0)              ;
     else sign=-1;
     System.out.println(sign);
}
```

Fig 3 Dynamic slice for the criterion (n=-3, {sign})

We are interested in static slice computation since the slice has to be valid for all possible program executions. Slice computed using our approach is actually a kind of combination of a number of dynamic slices, since we execute the program for different test cases corresponding to different program paths prior to slice computation.

2.4.2 Forward Vs Backward Slicing

Slice computation can be accomplished in two ways: using Backward Slicing or using Forward Slicing. Backward slicing is the one which was introduced originally by Weiser: a slice contains all statements and control predicates that may have influenced a given variable at a given point of interest. By contrast, forward slices contain all statements and control predicates that may be influenced by the variable. Nevertheless, backward and forward slices are computed in a similar way. The only difference is the way the flow is traversed.

We are using dynamic dependence analysis in our slicing approach; we execute the program prior to dependence computation for each possible test case reflecting program paths. Program execution has to start from the beginning and precedes in the forward direction and this force us to adopt forward slicing approach.

2.4.3 Intra-procedural Vs Inter-procedural slices

Slices can span a single procedure or multiple procedures and are called Intra-procedural and Inter-procedural slices respectively. Intra-procedural slicing computes slices within one procedure. On the other hand, inter-procedural slicing can compute slices which span procedures and even different classes and packages when slicing object-oriented programs.

Our slicing algorithm supports both intra-procedural and inter-procedural slicing. Though method extraction is restricted to extracting from within a single procedure, inter-procedural slicing support can be helpful in extracting aspects from object oriented code [14]. Inter-procedural slice identification using dynamic analysis is much easier and does not require any method call to definition mapping as in case of slicing using static analysis (We show this in further chapters).

Inter-procedural slicing raises an important problem: when the same procedure is called at different places in the program, the context in which the call occurs is certainly different. Not taking into account this difference of context can lead to very inaccurate slices. Because program slicing for refactoring must be sufficiently accurate to be useful in a development process, it has to be inter-procedural and account for this difference of context. Solutions that handle this problem are said **context sensitive**.

Context sensitivity issue arises only if static analysis is used. In our approach the method is executed for all possible test cases corresponding to different program paths, and this execution of the method rather than just analyzing it statically, automatically solves the problem of context sensitivity.

3. STATIC SLICER USING DYNAMIC ANALYSER

In this section we propose our slicing algorithm that computes static slices by using dynamic analysis. We first introduce the terminology used in the approach. Then we describe the algorithm. And finally we discuss the features that our approach supports or could be enhanced to support.

3.1 Terminology

3.1.1 Variable Accesses

Variable accesses in a program can be of two types – read and write. A program statement performs some computation taking as input values of some variables, i.e. it reads some variables, and writes some variable as a result of its computation. We denote a variable v's read as Read(v) and write as Write(v).

3.1.2 Events, Listeners and Handlers

An event in a program's execution is a particular occurrence like, reading and writing of variables, calling and execution of methods, rising of an exception, etc. We can observe these events with the help of event listeners and take some action, with the help of handlers, when the events occur.

We make use of event listeners and event handlers to calculate data dependencies. Variable reading and writing are the events we are interested in; we attach listeners to these events and specify the handler code that executes when the event fires.

3.1.3 Data Dependency

Data dependence from statement s1 to statement s2 by a variable v, exists iff

- s1 assigns a value to *v*,
- s2 refers to *v*, and
- at least one execution path from s1 to s2 without re-defining v exists

For example, in the class input.PolyInterPro, shown in Fig 4, the variable input.PolyComp.sum is data dependent on input.PolyInterPro.c, input.PolyComp.sum and input.PolyComp.i. Hence statement 10 is data dependent on statements 1, 2, 4, 5, 7 and 9.

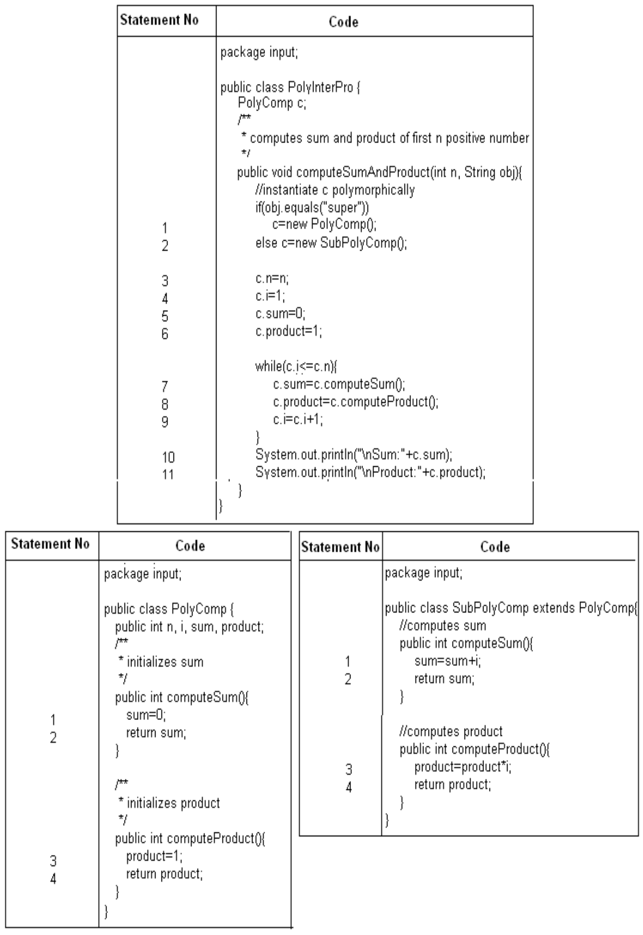
3.1.4 Testing

Developer is responsible to perform unit testing of the components (classes containing instance variables and methods) he develops. If we consider the testing of a method, two kinds of testing are performed on them: *black-box testing* in which we test the method's functionality without regard to its structure (code structure) and *white-box or structural testing* in which we design test cases making sure that each path through the method is executed [15].

Our slicing algorithm collects information about data flow in a program during execution. We exploit the structural testing performed on the program/method prior to refactoring.

3.2 Algorithm

We describe the algorithm below and we use the program shown in the Fig 4 to illustrate these steps. We are slicing the method *computeSumAndProduct(int n, String obj)* in the class *input.PolyInterPro.* input.*PolyComp* is a class on which *input.PolyInterPro* is dependent and *input.SubPolyComp* is a sub class of *input.PolyComp.* This program computes and displays the sum and product of first n positive integers and is made complex for the explanation of our algorithm. Program statements are numbered and this is used by our algorithm to find the statements that form the slice.



```
Statement No   Code
               package input;

               public class PolyInterPro {
                   PolyComp c;
                   /**
                    * computes sum and product of first n positive number
                    */
                   public void computeSumAndProduct(int n, String obj){
                       //instantiate c polymorphically
                       if(obj.equals("super"))
     1                     c=new PolyComp();
     2                 else c=new SubPolyComp();

     3                 c.n=n;
     4                 c.i=1;
     5                 c.sum=0;
     6                 c.product=1;

                       while(c.i<=c.n){
     7                     c.sum=c.computeSum();
     8                     c.product=c.computeProduct();
     9                     c.i=c.i+1;
                       }
    10                 System.out.println("\nSum:"+c.sum);
    11                 System.out.println("\nProduct:"+c.product);
                   }
               }
```

```
Statement No   Code
               package input;

               public class PolyComp {
                 public int n, i, sum, product;
                 /**
                  * initializes sum
                  */
                 public int computeSum(){
     1             sum=0;
     2             return sum;
                 }

                 /**
                  * initializes product
                  */
                 public int computeProduct(){
     3             product=1;
     4             return product;
                 }
               }
```

```
Statement No   Code
               package input;

               public class SubPolyComp extends PolyComp{
                  //computes sum
                  public int computeSum(){
     1               sum=sum+i;
     2               return sum;
                  }

                  //computes product
                  public int computeProduct(){
     3               product=product*i;
     4               return product;
                  }
               }
```

Fig 4 Example program for slicing

1. Listener Attachment

Attach *listeners* to read and write accesses to the program/method variables so that a *handler* code will be triggered when the access *events* occur.

For our example program, attach read and write listeners to all the variables – c, n, i, sum and product.

2. Program Execution and Data Collection

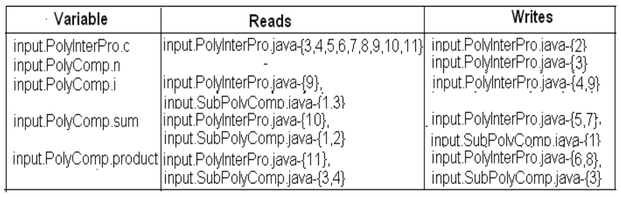
Perform structural testing of the code with a number of test cases ensuring that all program paths are executed at least once.

During the execution of the program while testing, variable accesses will trigger the handler code due to listener attachment. In the handler code collect the location, in terms of line number or statement number, of variable access.

This collection of variable access details will be repeated for each execution corresponding to the test cases identified. Then take the union of these access details to form complete variable access details for all the variables in the program/method under consideration.

For the example program shown in Fig 4, execute the method *computeSumAndProduct(..)* once with the arguments 4 and "sub" and next with the arguments 4 and "super". For the first test case we get read and write access details as shown in Fig 5.

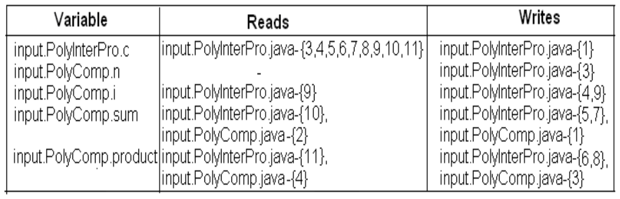
File names in which accesses occur are prefixed to the reads and writes.



Variable	Reads	Writes
input.PolyInterPro.c	input.PolyInterPro.java-{3,4,5,6,7,8,9,10,11}	input.PolyInterPro.java-{2}
input.PolyComp.n	-	input.PolyInterPro.java-{3}
input.PolyComp.i	input.PolyInterPro.java-{9}, input.SubPolyComp.java-{1,3}	input.PolyInterPro.java-{4,9}
input.PolyComp.sum	input.PolyInterPro.java-{10}, input.SubPolyComp.java-{1,2}	input.PolyInterPro.java-{5,7}, input.SubPolyComp.java-{1}
input.PolyComp.product	input.PolyInterPro.java-{11}, input.SubPolyComp.java-{3,4}	input.PolyInterPro.java-{6,8}, input.SubPolyComp.java-{3}

Fig 5 Variable Access Details for computeSumAndProduct(4,"sub")

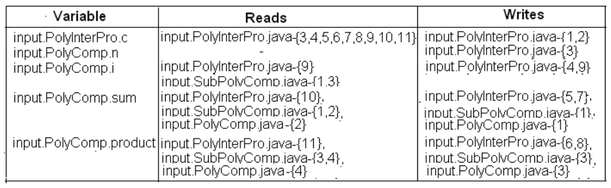
And for the second case the details are as shown Fig 6



Variable	Reads	Writes
input.PolyInterPro.c	input.PolyInterPro.java- {3,4,5,6,7,8,9,10,11}	input.PolyInterPro.java-{1}
input.PolyComp.n	-	input.PolyInterPro.java-{3}
input.PolyComp.i	input.PolyInterPro.java-{9}	input.PolyInterPro.java-{4,9}
input.PolyComp.sum	input.PolyInterPro.java-{10}, input.PolyComp.java-{2}	input.PolyInterPro.java-{5,7}, input.PolyComp.java-{1}
input.PolyComp.product	input.PolyInterPro.java-{11}, input.PolyComp.java-{4}	input.PolyInterPro.java-{6,8}, input.PolyComp.java-{3}

Fig 6 Variable Access Details for computeSumAndProduct(4,"super")

And the variable access details after the union of the above two are as shown in Fig 7.



Variable	Reads	Writes
input.PolyInterPro.c	input.PolyInterPro.java-{3,4,5,6,7,8,9,10,11}	input.PolyInterPro.java-{1,2}
input.PolyComp.n	-	input.PolyInterPro.java-{3}
input.PolyComp.i	input.PolyInterPro.java-{9} input.SubPolyComp.java-{1,3}	input.PolyInterPro.java-{4,9}
input.PolyComp.sum	input.PolyInterPro.java-{10}, input.SubPolyComp.java-{1,2}, input.PolyComp.java-{2}	input.PolyInterPro.java-{5,7}, input.SubPolyComp.java-{1}, input.PolyComp.java-{1}
input.PolyComp.product	input.PolyInterPro.java-{11}, input.SubPolyComp.java-{3,4}, input.PolyComp.java-{4}	input.PolyInterPro.java-{6,8}, input.SubPolyComp.java-{3}, input.PolyComp.java-{3}

Fig 7 Variable Access Details for after union

3. Data Dependency Calculation

For each of the variables, find dependencies with other variables. A variable v1 is dependent on variable v2 if v2 is read in a statement whose purpose is to perform a write to v1.

Use these dependencies to calculate transitive dependencies. A variable v1 is transitively dependent on variable v2 if v1 is dependent on a variable v' which is dependent on v2.

For the variables in our program, their dependencies and transitive dependencies with others is shown in Fig 8

Variable	Dependent on	Transitively Dependent on
c	-	-
n	{input.PolyInterPro.c}	-
i	{input.PolyInterPro.c, input.PolyComp.i}	{input.PolyInterPro.c, input.PolyComp.i}
sum	{input.PolyInterPro.c, input.PolyComp.sum, input.PolyComp.i}	{input.PolyInterPro.c, input.PolyComp.i, input.PolyComp.sum}
product	{input.PolyInterPro.c, input.PolyComp.i, input.PolyComp.product}	{input.PolyInterPro.c, input.PolyComp.i, input.PolyComp.product}

Fig 8 Dependencies for the variables in the earlier program

4. Slice Calculation

Collect all the statements where the slice variables are read or written. And then collect the definitions of the variables on which the slice variables are transitively dependent. All these statements form the slice.

Let us calculate the slice for the slice criterion – {sum}. The statements that form the slice are SubPolyComp.java-{1, 2}, PolyInterPro.java–{1, 2, 4, 5, 7, 9, 10} and PolyComp.java-{1, 2}. The statements are prefixed with the file names they occur.

3.3 Features

We investigate how our algorithm behaves in presence of different program constructs like different types of expressions, complex data types, structured jumps, aliases, polymorphism, etc. We restrict ourselves to the features of Java language, however. The capabilities and limitations of our algorithm are a function of the capabilities of the dynamic data flow analysis tools available (We discuss these tools/approaches in the next chapter).

Expressions and Local Variables

Our slicing algorithm records a dependency between two variables if they are written and read in the same statement. This would work for cases where the statements involve simple assignments, assignments involving prefix and postfix operators, assignments inside control constructs. It gets a little complicated when a statement involves a method call that performs some computation and returns a value. This returned value can be a local variable to the method. The tools available for dynamic data flow analysis today do not support listener attachment to local variable accesses and hence these events do not trigger handler execution. To support this, we need to customize these tools to support local variable access handling. However, we can overcome this limitation with the help of a static analysis supplement which introduces an instance variable wrapping a local variable for all the methods in a class. The code fragments below illustrate this. Now the instance variable corresponding to the local variable when accessed would trigger handler execution.

```
int aMethod()
{
        int x;
        x = 100;
        return x;
}
```

```
int aMethod()
{
        DfaIntWrapper x = new DfaIntWrapper ();
        x.value = 100;
        return x.value;
}

class DfaIntWrapper
{
        public int value;
}
```

Complex Data Types

Arrays are complex data types containing other data elements. One issue with arrays in the context of slicing is whether to log whole array in dependencies or only the array element accessed. Our choice determines the accuracy of slicing and we are considering the whole array as one variable in dependency analysis as this is the way analysis tools available to us treat arrays.

Classes are another complex data types containing instance variables. Since we are provided with facilities to attach listeners to instance variables we can attach them to individual instance variables or to the entire class, when the class itself is an instance variable of another class. This fine grained feature provides us accurate slices.

Jump Statements and Control Constructs

As said previously, we are concentrating on only identifying the statements that constitute the slice and not on slice extraction. Handling of jump statements like break and continue can be considered as part of semantic slice extraction and can be taken as future work.

Aliasing and Polymorphism

Aliasing refers to a situation in which one or more pointers/references are referring to the same variable. Using static and backward slicing it is not possible to determine what variable a reference is referring to prior to run time. Since we are slicing dynamically this problem is easily solved.

Method call resolution is a typical polymorphism example. Mapping between method calls to their definitions is very difficult if static slicing is performed as it requires the analysis of the entire source code to search for candidate methods that might be called. This is not at all a problem in dynamic slicing as the execution itself determines the mapping. We execute the program with test cases corresponding to each polymorphic path to take into account all the possible method destinations and collect all the methods as part of our static slice.

Other object oriented features like inheritance, packages, etc are also supported by our approach.

4. IMPLEMENTATION

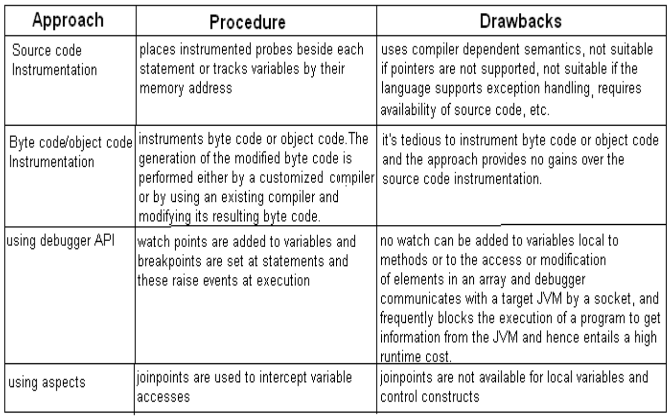
We use Java 1.5 for implementing our slicing algorithm and we have arrived at the conclusion to use AspectJ, to perform dynamic dependence analysis as part of our slicing algorithm, after evaluating the candidates for dynamic data flow analysis [9, 16]: program transformation, debugging services and aspects. The table below compares the approaches.

From Fig 10 it seems that the debugger API is a strong contender to support analysis. But we choose the aspect approach because it achieves good modularity, maintainability and reusability; it achieves complex handling of control elements such as multi-threading and exception in a well-organized way; the aspect handles objects using weak reference so as not to affect the lifetime of objects; and due to inefficiency of the debugger approach.

We now briefly describe the program elements implementing our algorithm:

- o Listener attachment to program variables is achieved using AspectJ’s field access pointcuts. We place our test files containing programs/methods in the *input* package of our project.

- User specified **method is executed** dynamically using the Java's reflection API. During execution, variable accesses trigger the **handler code** associated with the listeners attached and in the handler code we collect the location of accesses using AspectJ's reflection mechanism.



Approach	Procedure	Drawbacks
Source code Instrumentation	places instrumented probes beside each statement or tracks variables by their memory address	uses compiler dependent semantics, not suitable if pointers are not supported, not suitable if the language supports exception handling, requires availability of source code, etc.
Byte code/object code Instrumentation	instruments byte code or object code.The generation of the modified byte code is performed either by a customized compiler or by using an existing compiler and modifying its resulting byte code.	it's tedious to instrument byte code or object code and the approach provides no gains over the source code instrumentation.
using debugger API	watch points are added to variables and breakpoints are set at statements and these raise events at execution	no watch can be added to variables local to methods or to the access or modification of elements in an array and debugger communicates with a target JVM by a socket, and frequently blocks the execution of a program to get information from the JVM and hence entails a high runtime cost.
using aspects	joinpoints are used to intercept variable accesses	joinpoints are not available for local variables and control constructs

Fig 9 Comparison of approaches for dynamic data flow analysis

- We analyze these variable access details and **calculate dependencies** and Java's Collections Framework is heavily used here.
- **Slice calculation** is accomplished again with the help of Collections.
- We provide a simple Swing GUI for our application. We have used "GUI Genie" for generating the GUI.

5. CONCLUSION AND FUTURE WORK

Objective of our work was to design a slicing algorithm that suits reafactoring's needs and could be used as part of extract method refactoring.

We have observed that the approaches to slicing used alternative program representations and they were cumbersome due to the program conversion. We have taken the path suggested, computation of slice by directly analyzing the program, to improve this and identified the weakness of the present approach in handling run time constructs like aliasing, polymorphism, etc. We proposed our slicing approach that exploits of behavior preservation requirement of refactoring. We used the data collected during testing prior to refactoring to analyze dependencies. This dependency information is used by our algorithm to compute slice. Our algorithm handles run time constructs – aliasing and polymorphism – effectively as it uses dynamic analysis.

We have compared the tools available to perform dynamic data flow analysis and picked up aspectj for our implementation as it's the better one amongst. We have implemented our slicing algorithm using aspectj constructs for dependency analysis and a forward slicing algorithm in java.

Aspectj constructs return line number as the location of variable accesses and hence our implementation is line number dependent. This can be overcome either by customizing the aspectj constructs to return statement identifier/pointer as the location of variable accesses or using the Java Debugger Interface (JDI), which allows us analyze each statement being executed by setting breakpoints, for implementation.

Due to the unavailability of constructs to intercept local variable accesses, control construct execution, they were neglected by our algorithm. We can customize aspectj to support local variable access join-points and its reflection capabilities to include statement identifier, block/control construct identifier, etc. for each join-point. Once these features are available we can perform semantic preserving slice extraction just by sequestering the statements and control constructs as slice.

Once the support for intercepting local variables is available we can use the modified form of our slicing algorithm that takes into account local variables in inter-procedural slicing; the steps below mention the modifications to our original algorithm:

- Attach listeners to local variable accesses as well as formal parameters.
- Data collection phase remains the same but we collect data for all local variable accesses as well.
- In dependence calculation, fuse (take union) dependencies of formal parameters and actual parameters. Also take the union of dependencies of return parameter of the called procedure and its place holder in the calling procedure.
- Slice calculation phase remains the same.

Also, this algorithm being light weight - as it does not use any additional data structure for program representation - can be used as a general purpose slicing algorithm.

6. REFERENCES

[1] Mark Weiser. *Program slicing.* IEEE Transactions on Software Engineering, 10(4):352-357, 1984.

[2] Karl J. Ottenstein, Linda M. Ottenstein. *The program dependence graph in a software development environment.* Software Development Environments (SDE), pages 177-184, 1984.

[3] Filippo Lanubile, Giuseppe Visaggio. *Extracting Reusable Functions by Flow Graph-Based Program Slicing.* IEEE TRANSACTIONS ON SOFTWARE ENGINEERING, VOL. 23, NO. 4, APRIL 1997.

[4] Raghavan Komondoor, Susan Horwitz. *Semantics-Preserving Procedure Extraction.* In Proc. of 27th ACM Symp. on Principles of Programming Languages (POPL), (Boston, Massachusetts, January 2000).

[5] Arun Lakhotia, Jean-Christophe Deprez. *Restructuring programs by tucking statements into functions.* Information & Software Technology 40(11-12): 677-689 (1998).

[6] Mathieu Verbaere. *Program Slicing for Refactoring.* Master's thesis, 2003.

[7] Ran Ettinger. Refactoring via Program Slicing and Sliding. Ph D thesis 2006.

[8] M. Fowler. Refactoring: Improving the Design of Existing Programs. Addison-Wesley, 1999.

[9] A. Cain, J.-G. Schneider, D. Grant, and T. Y. Chen. *Runtime data analysis for Java programs*. In Proceedings of ECOOP 2003 Workshop on Advancing the State-of-the-Art in Runtime Inspection (ASARTI 2003), July 2003.

[10] Andre Restivo. *The Case for Aspect Oriented Programming.* in Proceedings of the 1st Conference on Methodologies for Scientific Research (CoMIC'06), p.84-92, 2006.

[11] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J. Loingtier and J. Irwin. *Aspect Oriented Programming.* Proceedings of the 11th annual European Conference for Object-Oriented Programming, vol.1241 of LNCS, pp.220-242(1997).

[12] AspectJ: http://www.eclipse.org/aspectj

[13] Frank Tip. *A survey of program slicing techniques*. Journal of programming languages, 3:121-189, 1995.

[14] MIGUEL JORGE TAVARES PESSOA MONTEIRO. Refactorings to Evolve Object-Oriented Systems with Aspect-Oriented Concepts. Ph D thesis, 2005.

[15] Ian Sommerville. *Software Engineering,* Pearson Education, 2004.

[16] Takashi Ishio, Shinji Kusumoto, Katsuro Inoue. Program Slicing Tool for Effective Software Evolution Using Aspect-Oriented Technique. IWPSE 2003: 3-12