Leaner Object-Oriented Slicing

Rob Law

School of Hotel and Tourism Management, The Hong Kong Polytechnic University, Hong Kong

Summary
This paper introduces the concept of Leaner Object-Oriented Slicing, an extension of Object-Oriented Program Slicing [4]. Leaner Object-Oriented Slicing can perform extra code reduction from an object-oriented slice, reducing the amount of information that a programmer must examine. This paper also provides a discussion on the implementation issues of a Leaner Object-Oriented Slicing based debugging tool.

Key words:
Software Engineering, Object-Oriented Programs, Debugging

Introduction

An object-oriented slice of an object-oriented program with respect to a class c is defined to consist of c and all base classes of c that could affect (either directly or transitively) the operation of an instance of c [4]. In other words, the bug which causes the incorrect operation of an instance of c is in the object-oriented slice with respect to c. Object-Oriented Program Slicing is further defined as the procedure used to compute an object-oriented slice.
The following example demonstrates how a complex debugging process can be simplified by applying Object-Oriented Program Slicing.

```
#include <iostream.h>
#include <string.h>
class BillingItem {
protected:
  char name[25];
  int  cost;
public:
  virtual void display() = 0;
};
class Product : public BillingItem {
  int qty_sold;
public:
  Product(char *nm, int qty)
    { qty_sold = qty; strcpy(name, nm); }
  void display() {cout << cost  << ' ' << name << "s were
sold ";}
};
class Service : public BillingItem {
  int manhours;
public:
  Service(char *nm, int mh, int cst)
    { manhours = mh; strcpy(name, nm); cost = cst; }
  void display() { cout << manhours; }
};
class Installation : public Service {
public:
  Installation(char *nm, int hrs, int cst) :
Service(nm,hrs,cst) {}
  void display ()
  {cout << "Installed Item: " << name;
   cout << "\nLabour: ";
   Service::display();
   cout << " hours";
   cout << "\nCost: $" << cost << "\n\n"; }
};
main() {
 Product pdsold("toaster", 4);
 pdsold.display();
}
```

Figure 1 - A C++ Program

Question: Where is the bug in the program shown in Figure 1?
The solution drawing of the problem in Figure 1 is surely beyond the ability of most existing computer debugging tools, intelligent tutoring systems, and programming environments. However, the debugging process can be simplified by applying the Object-Oriented Program Slicing technique to generate an object-oriented slice with respect to class *Product.* The C++ code of the object-oriented slice is shown in Figure 2.

```
#include <iostream.h>
#include <string.h>
class BillingItem {
protected:
  char name[25];
  int cost;
```

```
public:
  virtual void display() = 0;
};
class Product : public BillingItem {
  int qty_sold;
public:
  Product(char *nm, int qty)
     { qty_sold = qty; strcpy(name, nm); }
  void display() {cout << cost << ' ' << name << "s were
sold";}
};
main() {
  Product pdsold("toaster", 4);
  pdsold.display();
}
```

Figure 2 - An Object-Oriented Slice

The C++ program in Figure 2 returns the same computation as the C++ program in Figure 1 with respect to class *Product*. However, there is 50% reduction in the number of C++ statements that a programmer needs to examine for fault. In other words, a programmer only needs to scrutinize the class definitions *of BillingItem* and *Product* instead of definitions of all classes. This information reduction provides a solid advantage for a programmer to locate bugs. The code reduction percentage would be more significant for large real life object-oriented systems. From their experiments with human subjects, Lyle, Weiser, as well as Law and Maguire have obtained significant statistical evidence that programmers can locate bugs faster with less amount of code to examine [3,4,5,8].

2. Leaner Object-Oriented Slicing

An object-oriented slice contains a subset of code from the original program. The concept of Leaner Object-Oriented Slicing is an extension of Object-Oriented Program Slicing. A leaner object-oriented slice [LOOS] of an object-oriented slice with respect to a class *c* is defined as a program segment which consists of *c* and all derived classes of *c*. We also define Leaner Object-Oriented Slicing as the procedure to compute a LOOS. The operation of instances of classes in a LOOS can all be affected by *c*. The most feasible application of Leaner Object-Oriented Slicing is the further reduction of irrelevant information from an object-oriented slice.
To formally define the concept of Leaner Object-Oriented Slicing, the following four sets of classes are required:

ISS - An inheritance slicing set which consists of a class and its base classes. That is, an ISS is the original object-oriented slice.
CC - A set of classes which forms an inheritance net. The bottom class of this inheritance net produces a correct response.
IC - A set of classes which contain the classes with incorrectly defined data members and/or function members.
NISS - The relative complement of CC with respect to ISS. That is, NISS=ISS-CC. NISS is the LOOS to be returned. In other words, no class in NISS generates a correct response.

Theorem: An element IC_i of IC in ISS is also in NISS, for i = 1,2,...n where n is the index of the last element in IC.
Proof:
Suppose there is an element IC_i in ISS but not in NISS. This implies that IC_i can only be found in CC. If IC_i is in CC, then IC_i and all its derived classes respond incorrectly. However, this contradicts the definition of CC which states that the bottom class in the hierarchy net does respond correctly. Thus, the theorem is proved. □
To explain the concept of Leaner Object-Oriented Slicing further, consider the object-oriented slice in Figure 3 pictured next.

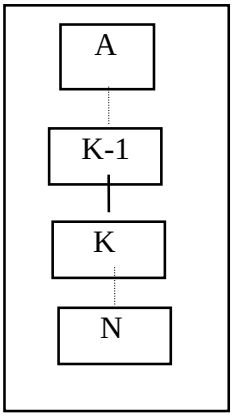

Figure 3 - Pictorial View of An Object-Oriented Slice

Figure 3 consists of an object-oriented slice with respect to class N. Classes K-1 and K are intermediate classes in the inheritance hierarchy and class A is the pure base class.
Having received an object-oriented slice, a user can perform a bottom-up search to look for the occurrence of the first class definition which produces the first incorrect response. That is, it is known that class N in Figure 3 does not generate a correct response. This incorrect response could be from the incorrect definition of class N and/or one or more of N's base classes. Suppose class K is the

first class in the hierarchy to incorrectly respond, (or class K-1 is the first one to produce a correct response), the search will stop at class K.
A LOOS with respect to class K is thus obtained, consisting of classes from K to N. This is, the incorrect data member and/or function member definitions which cause the incorrect output of the object of class N should be in this LOOS. In other words, no class in this LOOS responds correctly. The incorrect response of a class could be inherited from its superclass(es) in this LOOS and/or from a bad definition within the class. Of paramount importance, this LOOS further reduces the amount of information that a programmer must examine.
Figure 4 shown next demonstrates the code of a LOOS of class *Product* in Figure 2.

```
#include <iostream.h>
#include <string.h>
class Product : public BillingItem {
 int qty_sold;
public:
 Product(char *nm, int qty)
  { qty_sold = qty; strcpy(name,nm); }
 void display() { cout << cost << ' ' << name << "s were
sold"; }
};
main() {
 Product pdsold("toaster",4);
 pdsold.display();
};
```

Figure 4 - A Leaner Object-Oriented Slice

. A Leaner Object-Oriented Slicing System

C++_LOOS [C++ Leaner Object Oriented Slicer] was implemented to compute and return leaner object-oriented slices from C++ programs. C++_LOOS adopts a fast and direct approach to generate output to aid programmers in diagnosing faults in C++ programs. This will allow programmers to locate bugs more rapidly. C++ is selected because of its growing popularity in the past few years. The main reason for C++'s growing acceptance is the compatibility of C and C++. In a recent study, Hashemi and Leach found that C programmers could easily adapt to the C++ environment [1].
We should mention that C++_LOOS is not intended to act as a conventional debugger. In order to use a conventional debugger, a user needs to know the syntax of the debugger commands and the entities on which the debugger operates. Additionally, the user must be able to determine the detailed steps or operations which will provide a meaningful insight into the rationale for the failure of the underlying program. This is a non-trivial task.
4. Computing Leaner Object-Oriented Slices
C++_LOOS deals with single-file C++ programs. To handle C++ programs in multiple files, a user needs to merge these files into a single file by executing a preprocessor command. The computation of an object-oriented slice can be summarized in the following three stages.

Stage I
1. Read in each line of a source program.
2. Each line is stored as an instance of an object of type "source_line".
3. The constructor function of the source_line object uses two static members (head, tail) and a non-static member (next) to implement a singly linked list. This list is the only link between source lines in the code.
4. The source_line constructor classifies each line into one of the following five types:
   (i) Preprocessor Directive
   (ii) Class Definition
   (iii) Structure Definition
   (iv) Class Extension
   (v) Others (default if none of the above)

Stage II
1. Having read and represented all lines as source_line objects, the list is broken into sections of text via a class of object "tsl" which stands for Typed Source Lines.
2. The tsl is a container class implementing a linked list of text sections (segments). There is another container class of objects named "section" which is required in text section recognition. The section class is explained in point 5.
3. The tsl container has a default constructor that creates a null object. All elements of this object are going to be placed in the container by the function member "append". The list of elements thus formed is constructed by a source line test function "sltest".
4. To perform the actual appending, the program attempts to append a given source_line object to the current text section (each section of text is contained in an instance of an abstract class "section"). A function member of each section determines if the given line may be appended to the current section or not. If the appending is legal (the given line belongs to the current section), the function member of the section returns a pointer to itself. Otherwise, a NULL pointer will be returned. A NULL pointer indicates that the given line cannot be appended to the current section. Upon receiving a NULL pointer, the source line testing function "sltest" of the tsl class invokes another function member "get_new_section" to

determine the type of the next code section and construct a new instance of the class "section" which will contain the first line of this new section. A section contains the information of the first source_line object and the class source_line object.

5. The class "section" is an abstract class with a pure virtual function "append". This pure virtual function allows a derived class to encapsulate the logic that it uses to determine when the end of its section occurs. A new type of text section may be added by creating another object which is derived from "section".
6. Upon completion of the "sltest" function, the source code is represented as a two-dimensional singly linked list with one dimension being the sections and the other one being the individual lines.
7. A many-to-many relation dimension is constructed for all Class Definition source_line objects to hold the inheritance hierarchy.
8. The list of sections is searched to find any Class Extension types. The Class Extension type sections are then appended to the section of text where the class definition is contained. This searching is accomplished by going through the many-to-many relation dimension to find a node by node_name. Each class derived from "section" has an overloaded operator "+=" to perform the appending.

Stage III

1. All section pointers are stored in an array.
2. There is a function member of each node in the many-to-many relation dimension called "get_relation" that returns a pointer to a list of all the nodes that are related to a given node. This list is then used to generate an array of section pointers (text sections).
3. The section file_scope_list is then read and its contents are appended to the array of section pointers.
4. The section pointer array list is then sorted according to the order in the original text file and duplicates are removed. This array will then contain (in sorted order) pointers to "section" objects.
5. To generate a LOOS, a "print" function member of the "section" object is invoked to print the node's children.

To find a LOOS, C++_LOOS reads in a file which contains an object-oriented slice. A user is then required to enter a class name. A LOOS is then computed and returned to an output file specified by the user.

In a LOOS, C++_LOOS keeps all function definitions that are not members of any inheritance hierarchy. It is safer to retain these separate functions than to remove them completely. Additionally, in the presence of multiple inheritance hierarchies, C++_LOOS removes all hierarchies except the one which contains a class to be sliced. Real-life C++ systems always consist of multiple inheritance hierarchies. Therefore, C++_LOOS will have a larger code reduction for larger C++ systems.

C++_LOOS does not check for C++ syntax errors. Most available C++ compilers can provide useful information to help a programmer remove syntax errors.

5. Conclusion

The reduction in debugging time provided by C++_LOOS will be of great interest to most C++ programmers. Recent studies indicate that the time programmers spend on debugging is 50% of the time that they spend on program development [7]. By utilizing C++_LOOS to debug a computer program, especially a program written by others, the C++ programmers, regardless of his/her computer background and programming habits, can directly use an isolation debugging approach to locate a bug. In other words, a novice programmer, as well as an experienced programmer, can perform a simple mapping by using C++_LOOS to point directly to the specific program entities which are incorrect. The programmer's difficulty is not correcting the bug itself but in finding it [2,6]. In other words, fault localization is more beneficial than correction in the context of debugging. C++_LOOS fits itself exactly in this important but currently unfulfilled debugging area for C++ object-oriented programs. With the increasing popularity of object-oriented programming, C++_LOOS has the potential to be a useful debugging tool.

References

[1] R. Hashemi and R. Leach, Issues in Porting Software from C to C++, *Software-Practice and Experience* **22**(7) (1992) 599-602.

[2] I.R. Katz and J.R. Anderson, Debugging: An Analysis of Bug-Location Strategies, *Human-Computer Interaction* **3** (1987-1988) 351-399.

[3] R.C.H. Law, Evaluating the Program Slicing Technique, *SIAST TODAY* **4**(6) (1993) 6.

[4] R.C.H. Law and R.B. Maguire, Debugging of Object-Oriented Software, in: *Proceedings of 1996 Conference on Software Engineering & Knowledge Engineering*, Lake Tahoe, Nevada (1996), 77-84.

[5] J.R. Lyle, *Evaluating Variations on Program Slicing for Debugging* (U.M.I. Dissertation Information Service, Ann Arbor, Michigan, 1992).

[6] P.W. Oman, C.R. Cook, and M. Nanja, Effects of Programming Experience in Debugging Semantic Errors, *The Journal of Systems and Software* **9** (1989) 197-207.

[7] R. Ward, Beyond Design: The Discipline of Debugging, *Computer Language* **5**(4) (1988) 37-39.

[8] M. Weiser, Programmers Use Slices When Debugging, *Communications of ACM* **25**(7) (1982) 446-452.

Rob Law received his PhD (1994), MSc (1990), and BASc (1988) in Computing

Science from universities in Canada. He is presently an Associate Professor of Information Technology at the Hong Kong Polytechnic University's School of Hotel & Tourism Management.