CODE SLICING

Usman Ghani

Lecturer

Computer Science

CODE SLICING

- Code slicing
- Static and dynamic slicing
- Other types of slicing
- Applications of program slicing

Code Slicing

- A technique for simplifying programs by focusing only on selected aspects of semantics.
- During the slicing process those parts of the program which can be determined to have no effect upon the semantics of interest are deleted.
- Slicing has various application areas including testing and debugging, re-engineering, program comprehension and software measurement.

Code Slicing

- For example, during debugging, there is little point in the (human) debugger analyzing sections of the source code which cannot have caused the bug. Slicing avoids this situation by simply removing these parts of the program and focusing attention on those parts which may contain a fault.

Code Slicing

- The syntactic dimension and the semantic dimension.
- The semantic dimension focuses on describing that which is to be preserved.
- The program’s static behavior is preserved by static slicing criteria and its dynamic behavior is preserved by dynamic criteria

Code Slicing

This section reviews five semantic paradigms for slicing:

- Static
- Forward
- backward
- dynamic and conditioned
- two syntactic paradigms: syntax-preserving and amorphous.

Over the years, slicing has been successfully applied to many software development problems including testing, reuse, maintenance and evolution. This section describes the main forms of program slicing and some of the applications to which slicing has been put.

Static Slicing

- A slice is constructed by deleting those parts of the program that are irrelevant to the values stored in the chosen set of variables at the chosen point. The chosen points of interest are usually identified by annotating the program with line numbers which identify each primitive statement and each branch node.

Static Slicing

- The point of interest is usually indicated by adding a line comment to the program.
- In general slices are constructed for a set of variables, but in this chapter only slices constructed for a single variable will be considered.
- Thus, assuming that you have been given a variable v and a point of interest n, a slice will be constructed for v at n.
- This is not restrictive, since the slice with respect to a set of variables V can be formed from the union of the slices on each variable in V.

Dynamic Slicing

- A dynamic slice with respect to three pieces of information.
- Two of these are just the same as we find in static slicing the variable whose value appears to be wrong and the point of interest within the program.
- The third is a mere sequence of input values for which the program was executed. Collectively, this information is referred to as the “dynamic slicing criterion”.

Dynamic Slicing

- We can construct a dynamic slice for a variable v, at a point n, on an input i.
- To describe the input sequence i, we shall enclose the sequence of values in angled brackets.
- Thus represents a sequence of three input values the first of which is 1, the second of which is 4 and the last of which is 6

Is Dynamic Always Better than Static Slicing?

- Dynamic slices are very attractive as an aid to debugging, and in particular, they are superior to static slices for this application.
- We would forgive the reader for concluding that static slicing is rendered obsolete by dynamic slicing.
- However, there is still the need to have static slicing for some applications in which the slice has to be found for every possible execution.

Is Dynamic Always Better than Static Slicing?

- For example, assume we are interested in reusing the part of a program which implements a particularly efficient and well-tested approach to some problem.
- More often than not, in such situations (particularly with legacy code), the code we want to reuse will be intermingled with all sorts of other unrelated code which we do not obviously want.
- In this situation then static slicing becomes ideal as a technique for extracting the part of the program we require, while leaving behind the part of the program we are not interested in.

Is Dynamic Always Better than Static Slicing?

- This observation highlights the inherent trade-off between the static and dynamic paradigms.
- Static slices are typically larger but will definitely cater for every possible execution of the original program.
- Dynamic slices are typically much smaller, but they will only cater for a single input.

Conditioned Slicing

- The static and dynamic paradigms represent two extremes – either we say nothing about the input to the program (static slicing) or we say everything (dynamic slicing). Conditioned slicing then allows us to bridge this gap.
- This approach to slicing is called the “conditioned approach”, because the slice is conditioned by knowledge about the condition in which the program is to be executed. Conditioned slicing addresses the kind of problems the maintainers face when trying to understand the large legacy systems.

Forward Slicing

- Forward slicing uses functions that normally process input values. The objective of this method is to fi nd all areas of code that depend on the values of the input variables. Ripple effect analysis is applied; given a variable and slicing range, the statements in that range that can be potentially affected by the variable are defi ned. Statements can be affected in terms of data fl ow, control dependency, or the recursive process; variables on the left and right of the statement are included.

Backward Slicing

- In backward slicing, the process starts with the results of the existing program. Areas of code that contribute to values of output variables are a part of the slice. Given a variable and range, this method of slicing returns statements that can potentially affect the value of the variable; variables only on the right side of the equation are used. Tools are available to facilitate slicing, but the process is still diffi cult and time consuming.

Amorphous Slicing

- All approaches to slicing discussed so far have been “syntax preserving”. This means, they are constructed by the sole transformation of statement deletion. The statements which happen to remain in the slice are therefore a syntactic subset of the original program from which the slice was originally constructed. By contrast, amorphous slices are constructed using any program transformation which simplifi es the program and which preserves the effect of the program with respect to the slicing criterion

Amorphous Slicing

- This syntactic freedom facilitates amorphous slicing's ability to perform greater simplifications with the result that amorphous slices are never larger than their syntax-preserving counterparts. In most cases they are considerably smaller. Amorphous slicing allows a slicing tool to "wring out" the semantics of interest, thus aiding analysis, program comprehension and reuse. As a matter of fact, it actually does this correctly. However, the syntax preserving slice does not appear to make this very clear. This is because all that can be achieved by statement deletion is the removal of the fi nal statement.

Applications of Program Slicing

- Slicing has quite an array of applications. Any area of software engineering and software development in which it is useful to extract subprograms based upon arbitrary semantic criteria are candidate applications of slicing.