Software Engineering 4

Static Analysis Techniques

Andrew Ireland

School of Mathematical and Computer Science
Heriot-Watt University
Edinburgh

Outline

- Focus on static analysis for verification.
- Emphasis on analysis across the development life-cycle.
- Highlight key testing methods & techniques.

Static Analysis: Definition

- Static analysis is any form of analysis that does not require a system to be operated.
- Static analysis complements dynamic analysis, where system operation is central. Testing is often thought of in terms of purely dynamic analysis — this is not the case.
- When applied to code, typically referred to as white-box, glass-box, structural or implementation based techniques.

Static Analysis: Methods

- Reviews & inspections.
- Software metrics.
- Flow analysis.
- Formal verification.

Reviews & Inspections

- Reviews and inspections are one of the most widely applicable forms of static analysis — for the simple reason that they are sometimes the only techniques available in the early stages of development.
- An inspection is a review with the more focused goal of detecting defects.
- Fagan Inspections: One of the most widely used styles of inspection. Developed within IBM since the 1970's and is used widely throughout the industry as a process for detecting defects across the software development life-cycle.
- NASA has a Software Formal Inspection Process Standard (NASA-STD-2202-93) and an associated Software Formal Inspection Guidebook (see references).

Fagan Inspections

Planning: Preconditions for inspection defined; selection of participants; scheduling issues.

Overview: Background material distributed to participants; assign inspection roles to participants, *i.e.* author, reader, tester, moderator.

Preparation: Individual preparation based upon the background material.

Inspection: Focus on defect identification, *i.e.* discussion of solutions should be discouraged.

Rework: The author addresses the defects.

Follow-up: Verification of patches by some or all of the inspection team, ensuring no secondary defects have been introduced.

Software Metrics

- Some examples:
  - Graph theoretic complexity based on flow graph model
  - Module accessibility
  - Number of comments
- Pros and Cons:
  - Fully automatic
  - No focus on potential defect locations

Cyclomatic Complexity

- First proposed by Tom McCabe – logical complexity measure based upon graph theory.
- Basic idea:
  - Translate source code into a flow graph G (directed graph)
  - The cyclomatic complexity CC for a graph G equals:

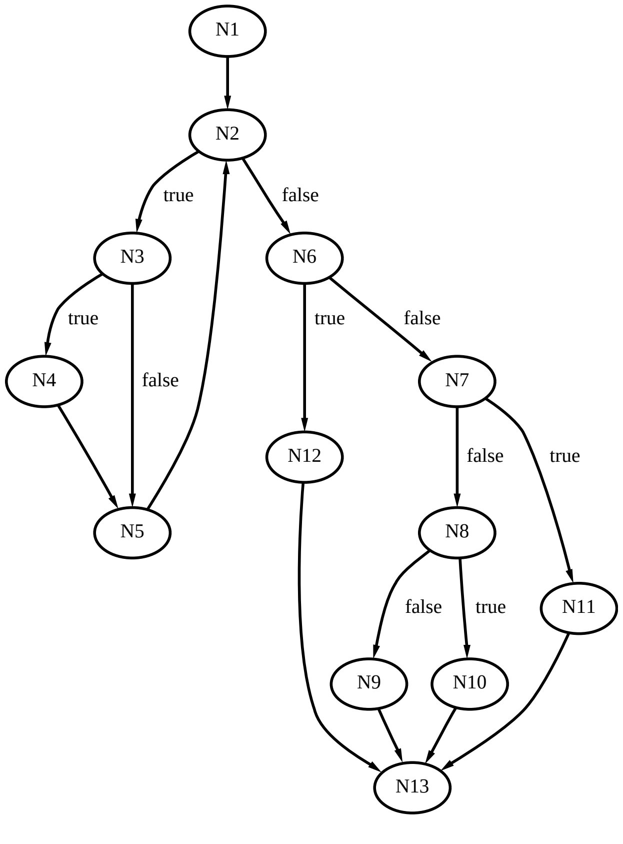
    $$CC(G) = (Edges - Nodes) + 2$$
- Empirical studies have established a correlation between cyclomatic complexity and the number of errors within source code.

Cyclomatic Complexity: An Example

```
typedef enum gradetype {A, B, C, F};

gradetype autograde(int student[], int model[], int num){
int score; gradetype grade;
int correct = 0; int index = 0;
while (index < num){
      if (student[index] == model[index]) correct++;
      index++;}
     score = (correct * 100) / num;
     if (score >= 80) grade = A;
     else
        if (score >= 60) grade = B;
        else
           if (score >= 50) grade = C;
           else
              grade = F;
     return grade;}
```


Flow Graph for `autograde`



Cyclomatic Complexity for `autograde`

- Cyclomatic Complexity:

  $CC(G) = (Edges - Nodes) + 2$

  $6 = (17 - 13) + 2$

- Note that while such measures tell you where the complexity within your system lies they do not tell you precisely where to look for potential defects.
- Complexity metrics were applied to the Channel Tunnel Software: 3 million lines of code – approach not fully endorsed by the safety/control community.

Data Flow Analysis

- Aim is to identify data flows that do not conform to sound programming practices, *e.g.* variables are not read before they are written; ineffective code.
- A purely symbolic form of analysis, *i.e.* no specific data values are considered.
- Based upon a number of relationships between variables and expressions.
- Process involves annotating a program flow graph with each data object definition (D), usage (U) and elimination (E).
- Analysis involves flow graph traversal, *e.g.* DD paths suggest redundancy, DE paths are most likely to be bugs.

Control Flow Analysis

- Aim is to detect poorly structured code, *e.g.* multiple exists from a loop, dead code etc
- Process again typically involves translating the program into a flow graph.
- By a process of repeated reduction inaccessible code and certain classes of non-termination can be identified.

Program Slicing

- Program slicing involves focusing on a particular subset of variables within a given program.
- The parts of the program that are relevant to the subset of variables denotes a program slice.
- Some applications:
  - Program testing & re-testing: provides focus with respect to test case design and the selection of regression tests.
  - Program comprehension: slicing provides a useful aid to understanding code where no documentation exists.
- Tool support, *e.g.* CodeSurfer (see references).

Types of Program Slicing

- **Backward:** For a given subset of variables V and statement S, then the backward program slice contains all statements that proceed S and which reference variables within the subset.
- **Forward:** For a given subset of variables V and statement S, then the forward program slice contains all statements that follow S and which reference variables within the subset.
- **Static:** A static program slice is calculated symbolically, *i.e.* takes no account of concrete data values.
- **Dynamic:** A dynamic program slice is calculated based upon particular data values.

Note that forward and backward slices can be calculated either statically or dynamically.

An Example Of Program Slicing

Program	**A Program Slice**
`read(X);`	`read(Y);`
`read(Y);`	`read(Y);`
`Q := 0;`	
`R := X;`	`R := X;`
`while R >= Y do`	`while R >= Y do`
`begin`	`begin`
`   R := R - Y;`	`   R := R - Y;`
`   Q := Q + 1`	
`end;`	`end;`
`print(Q);`	
`print(R);`	`print(R);`

Above illustrates backward slicing with respect to the last occurrence of `R`.

Information Flow

- Exploits software annotations, *i.e.* meta-data that asserts properties that should hold at particular points during program execution.
- SPARK based example:

```
procedure Exchange(X, Y:in out Float)
--# derives X from Y &
--#         Y from X;
is
   T:Float;
begin
   T:=X; X:=Y; Y:=T;
end Exchange;
```

Note: `derives` defines a dependency relation between variables that is checked against the code automatically by the SPARK Examiner static analyser.

Formal Verification

- Formal verification enables us to check that code is compliant with the specification.
- Logical assertions embedded within the code provides a basis for program specification.
- Mathematical logic provides the "glue" that enables us to show that code is compliant with the specification.

Code & Assertions

$\Downarrow$

Pure Assertions

$\Downarrow$

Proof or Failure

Correctness Specifications

Functional based specifications: Express that under certain assumptions (preconditions) that the software will deliver a particular result (postcondition) – termination may or may not be guaranteed.

Property based specifications: Express that certain properties that are intended of a system actually hold, *e.g.* freedom from deadlocks, mutual exclusive access to critical data, freedom from run-time exceptions.

Note: an integer overflow run-time error led to the loss of Ariane 5; "Buffer overflows have been the most common form of security vulnerability in the last ten years" (DARPA Information Survivability Conference, 2000)

A Data Filter: Exception Freedom Example

Proving that exceptions such as division by zero and numeric overflow can never occur during the execution of the software.

```
subtype Index_Type is Integer range 0 .. 9;
type Data_Set_Type is array (Index_Type) of Integer;
Result: Integer;
...
Result:= 0;
for I in Index_Type loop
   if Data_Set(I) >= 0 and Data_Set(I) <= 100 then
      Result:= Result+Data_Set(I);
   end if;
end loop;
```

A Closer Look ...

- Consider:

```
Result:= Result+Data_Set(I);
```

- What we know:

```
Data_Set(I) >= 0  and Data_Set(I) <= 100
Result >= -32768 and Result <= 32767
```

Note: assuming 16-bit two's complement integer representation.

Integer Overflow?

- We must prove that `Result+Data_Set(I)` does not exceed the bounds of `Result`, *i.e.*

```
Result >= -32768 and Result <= 32767
```

- From what we know a proof is not possible.
- Either the code is buggy or we need to discover more information, *e.g.* tighter bounds on `Result`:

```
Result >= 0 and Result <= I * 100
```

A Data Filter: Loop Invariants

```
subtype Index_Type is Integer range 0 .. 9;
type Data_Set_Type is array (Index_Type) of Integer;
Result: Integer;
...
Result:= 0;
for I in Index_Type loop
   --# assert Result >= 0 and Result <= I * 100;
   if Data_Set(I) >= 0 and Data_Set(I) <= 100 then
      Result:= Result+Data_Set(I);
   end if;
end loop;
```

Within the context of SPARK, the programmer annotates their code with the tighter bounds on `Result`, *i.e.* they assert a loop invariant of the form:

```
   --# assert Result >= 0 and Result <= I * 100;
```


Informal Proof (upper bound on `Result`)

- From the start of the code to the loop invariant we know:

  $\texttt{I}_1$ `= 0   and   ` $\texttt{Result}_1$ `= 0`

- We have to prove:

  $\texttt{Result}_1$ `   <=   ` $\texttt{I}_1$ `* 100`

  `      0   <=   <= 0 * 100`

  `      0   <=   0`

  `         true`

Informal Proof (upper bound on `Result`)

- On the $\texttt{N}^{th}$+1 iteration we know:

$$\texttt{Data\_Set(I}_N\texttt{) >= 0} \quad \texttt{and} \quad \texttt{Data\_Set(I}_N\texttt{) <= 100}$$

$$\texttt{Result}_N \texttt{ >= 0} \quad \texttt{and} \quad \texttt{Result}_N \texttt{ <= I}_N \texttt{ * 100}$$

- We have to prove:

$$\texttt{Result}_N \texttt{ + Data\_Set(I}_N\texttt{) <= (I}_N\texttt{+1) * 100}$$

$$\texttt{Result}_N \texttt{ + Data\_Set(I}_N\texttt{) <= I}_N\texttt{*100 + 100}$$

$$\texttt{Result}_N \texttt{ <= I}_N\texttt{*100 and Data\_Set(I}_N\texttt{) <= 100}$$

$$\texttt{true and true}$$

$$\texttt{true}$$

Exercise: Prove that the lower bound on `Result` is not violated.

Cost Benefit Analysis

- Formal verification greatly increases confidence in a compliance argument.
- While batch or retrospective verification is impractical, correctness by construction has been shown to work.
- Development time need not be increased, and there is evidence to shown that significant reductions in maintenance costs can be achieved.
- High degree of skill required by the programmer, *e.g.* in the Data Filter example where did the intermediate assertion come from:

```
--# assert Result >= 0 and Result <= I * 100;
```

- Greater automation is required for wider acceptance.

Summary

- Static analysis – a range of techniques that cover a wide spectrum of the software development life-cycle.
- Level of analysis is application dependent, *e.g.* are you building an aircraft flight monitoring system or a plugin for a web browser?
- A key advantage of static analysis is that it can be applied before coding is complete.

References

- "Advances in software inspection", Fagan, M.E. IEEE Trans. Software Eng., 12(7), 744-51, 1986. See also Michael Fagan Associates: `http://www.mfagan.com/`
- "Software Formal Inspections Guidebook", *Office of safety and mission assurance*, NASA-GB-A302, 1993. See also: `http://satc.gsfc.nasa.gov/fi/fipage.html`

References

- "A Software Complexity Measure", McCabe, T. IEEE Trans. Software Engineering, Vol 2, No 6, 1976.
- "Software Complexity: Measures & Methods", Zuse, H. DeGruyter, NY, 1990.
- "Program Slicing", IEEE Trans. Software Eng., vol SE-10, 1984.
- "The Science of Programming", Gries, D. New York, Springer-Verlag, 1981.
- "High Integrity Software: The SPARK Approach to Safety and Security", Barnes, J. Addison-Wesley, 2003.

References

- Quantitative Software Management Inc.

  `http://www.qsm.com`
- Software Productivity Research Inc.

  `http://www.spr.com`
- CodeSurfer: A software maintenance, understanding and inspection tool for C and C++, based upon program slicing. GrammaTech, Inc. NY, US, `http://www.grammatech.com/`