Software Design (F28SD2)
Static Analysis Techniques

Andrew Ireland
Department of Computer Science
School of Mathematical and Computer Sciences
Heriot-Watt University
Edinburgh

Outline

- Focus on static analysis for verification.
- Emphasis on analysis across the development life-cycle.
- Highlight key testing methods & techniques, as well as aspects of formal verification.

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.
- 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 methods.

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

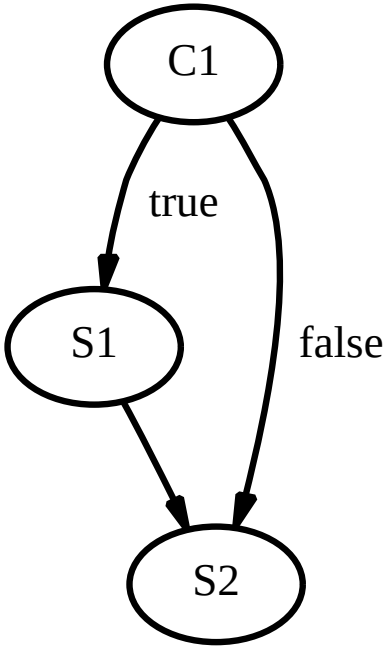
Cyclomatic Complexity

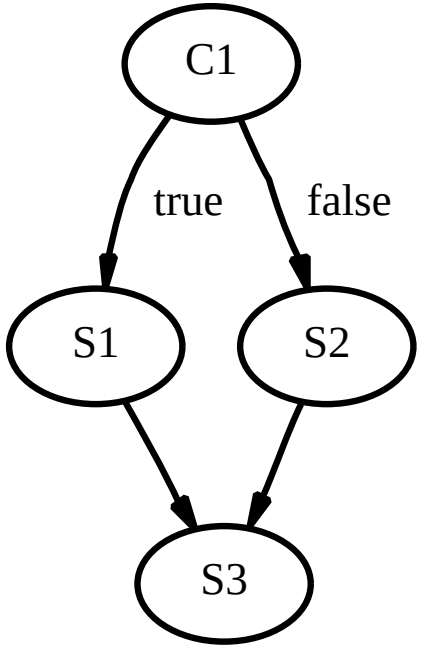
- First proposed by Tom McCabe – logical complexity measure based upon graph theory.
- Basic idea:
  - Translate source code into a flow graph G (directed graph)
  - The cyclomatic complexity CC for a graph G equals:

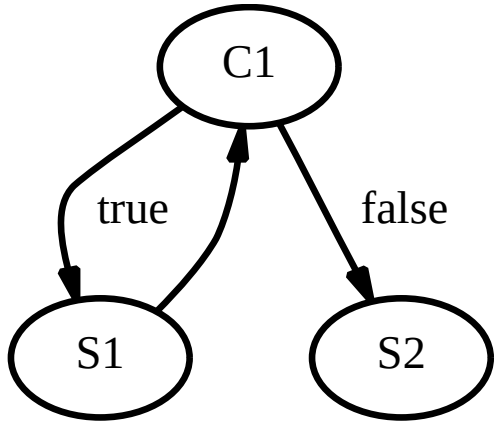
$$CC(G) = (Edges - Nodes) + 2$$

- Note: there must exist unique **start** and **end** nodes.
- Empirical studies have established a correlation between cyclomatic complexity and the number of errors within source code.

Flow Graph Construction





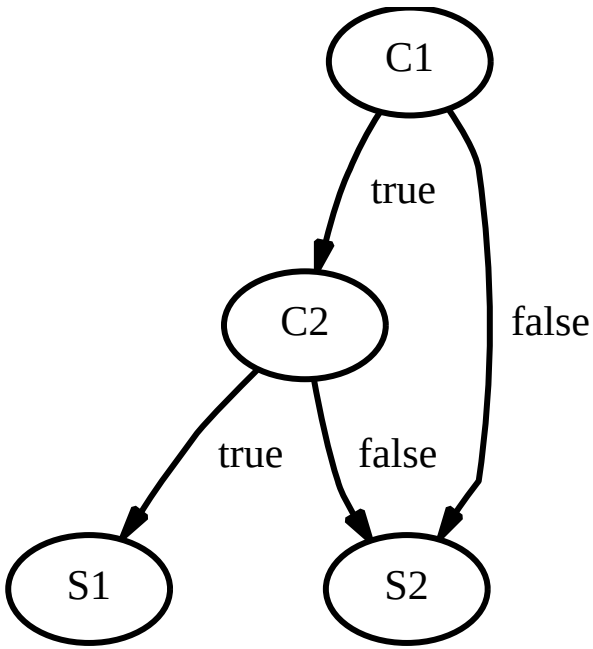


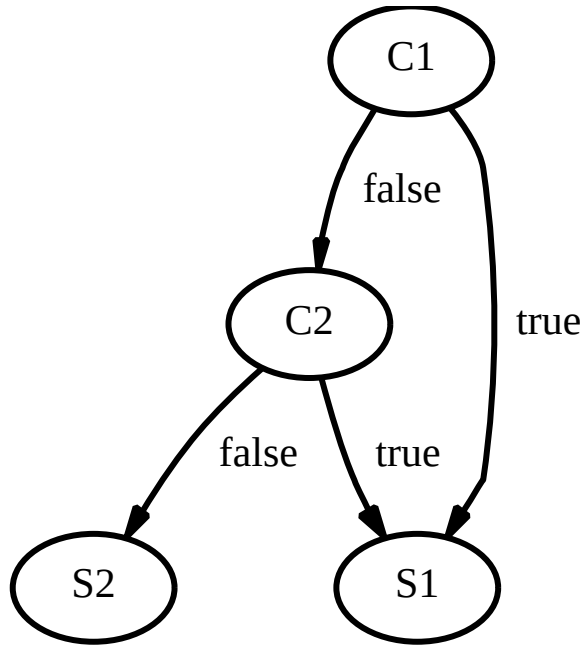
`if C1 S1; S2`

`if C1 S1 else S2`

`while C1 do S1; S2`

Flow Graph Construction






```
if (C1 && C2) S1; else S2
```

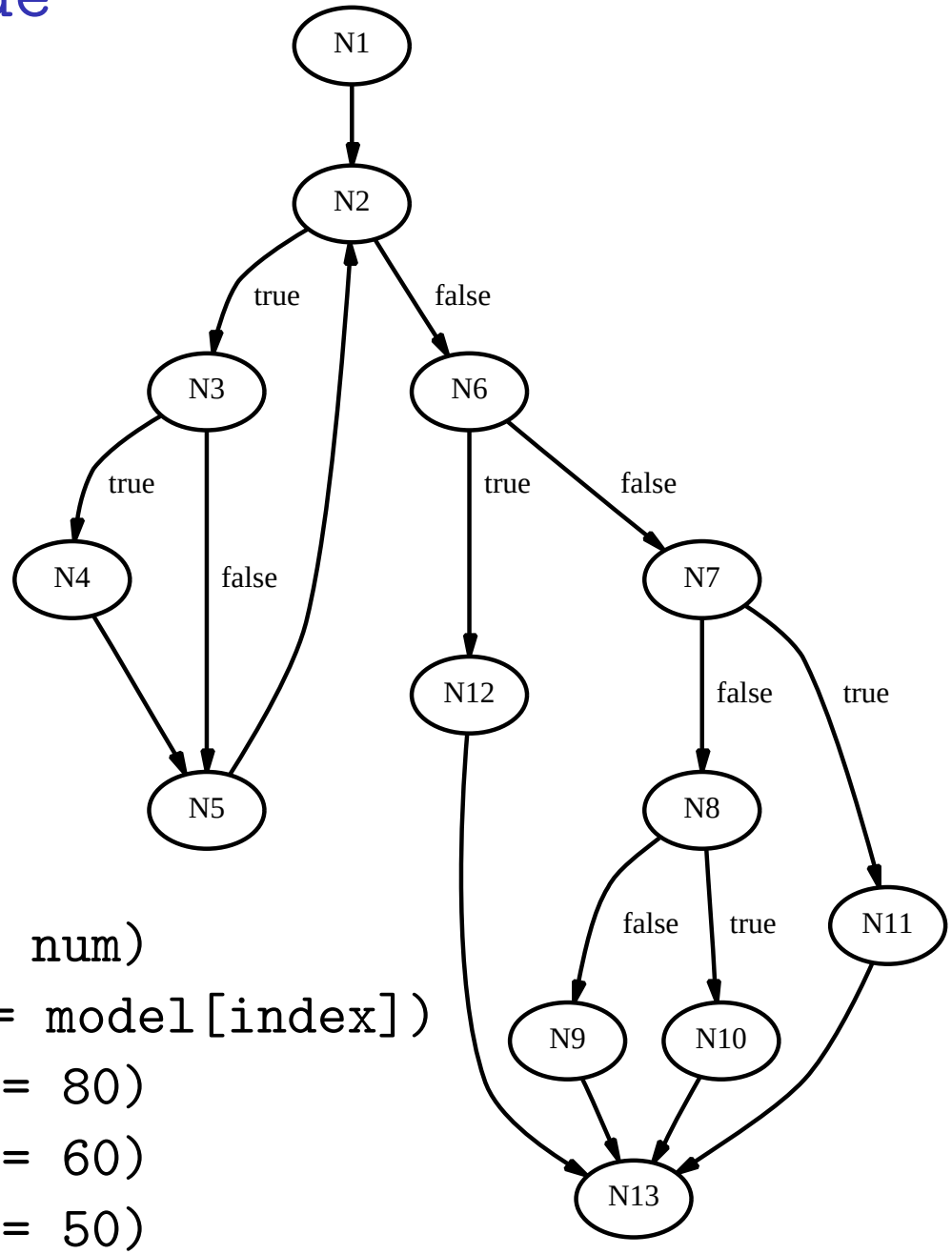
```
if (C1 || C2) S1 else S2
```

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




```
N2              (index < num)
N3  (student[index] == model[index])
N6              (score >= 80)
N7              (score >= 60)
N8              (score >= 50)
```

Cyclomatic Complexity for `autograde`

- Cyclomatic Complexity:
$$\begin{aligned} CC(G) &= (Edges - Nodes) + 2 \\ 6 &= (17 - 13) + 2 \end{aligned}$$
- 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.

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.
- For tool support see references.

Types of Program Slicing

- **Backward:** For a given statement S, a backward slice through a program contains all statements that **effect** whether control reaches S and also all statements that **effect** the value of variables that occur in S.
- **Forward:** For a given statement S, a forward slice through a program contains all statements that are **affected** by S.
- **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

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

A Program Slice

```
read(X);
read(Y);

R := X;
while R >= Y do
begin
   R := R - Y;

end;

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. Note also that SPARK is derived from the Ada programming language.

Formal Methods

- Formal methods use mathematical notations and logic (proof) to establish correctness.
- Formal methods are applicable to:
  - Designs, *e.g.* Z, VDM, Event-B, Alloy.
  - Code, *e.g.* SPARK, ESC/Java2, Spec#
- Formal methods are also used to bridge the gap between abstract designs and concrete code, *e.g.* B-Method.
- Mathematical logic provides the "glue" that enables us to establish correctness.

Correctness

Function based correctness: Express that under certain assumptions (preconditions) that the software will deliver a particular result (postconditions).

Property based correctness: Express that certain properties that are intended of a system actually hold. Typically properties are generic, so the programmer can select them "off the shelf" *e.g.* freedom from deadlocks, mutual exclusive access to critical data, freedom from run-time exceptions.

Assertion Based Formal Verification

- Typically function/property based specifications are represented as **assertions** embedded within the code.
- The assertions and code are then compiled into **verification conditions** (VCs) – logical conjectures.
- If the VCs can be proved, then the code is guaranteed to be correct with respect to the specification (assertions).

A Function Based Example - Bubble Sort in SPARK

```
subtype Index_Type is Integer range 0 .. 9;
type Array_Type is array (Index_Type) of Integer;
```

- **Precondition:** No preconditions.

```
--# pre true;
```

- **Postcondition:** On termination `Table` is ordered from `0` upwards and all elements are preserved.

```
--# post Ordered(Table, 0, Index_Type'Last) and
--#      Perm(Table, Table~);
```

where:

- `Table~` denotes the initial value of `Table`
- `Index_Type'Last` denotes the upper bound of `Index_Type` (and `Index_Type'First` denotes the lower bound).

Bubble Sort in SPARK

```
procedure Bubble_Sort(Table: in out Array_Type)
   is
      T: Integer;
   begin
      for I in Index_Type range 1 .. Index_Type'Last loop
         for J in reverse Index_Type range I .. Index_Type'Last
            if Table(J-1) > Table(J) then
               T:= Table(J-1);
               Table(J-1):= Table(J);
               Table(J):= T;
            end if;
         end loop;
      end loop;
   end Bubble_Sort;
```

Bubble Sort in SPARK: Auxiliary Assertions

- **Outer loop invariant:** `Table` is ordered from `0` to `I-2`. Elements from `0` to `I-2` are less than or equal to all the elements from `I-1` upwards. All elements are preserved and `I` remains within its type.

```
--# assert Ordered(Table, 0, I-2) and
--#        Partitioned(Table, 0, I-2, Index_Type'Last)
--#        Perm(Table, Table~) and
--#        1 <= I and I <= Index_Type'Last;
```

- **Inner loop invariant:** Extends the outer invariant to include that element `J` has the minimum value of the elements from `J` upwards. `I` remains within its type. `I` is less than or equal to `J`. The upper bound on `J` is `Index_Type'Last`.

```
--# assert Ordered(Table, 0, I-2) and
--#        Partitioned(Table, 0, I-2, Index_Type'Last)
--#        Perm(Table, Table~) and
--#        Minimum(Table, J, Index_Type'Last) and
--#        1 <= I and I <= Index_Type'Last and
--#        I <= J and J <= Index_Type'Last;
```

Property Based: Overflow Exceptions

- Overflow errors occur when a program stores data into a chunk of memory that is too small for it
- An integer overflow run-time error led to the loss of Ariane 5, *i.e.* $7billion of development over 10 years.
- "On 19 Oct 2000, hundreds of flights were grounded or delayed because of a software problem in the Los Angeles air-traffic control system. The cause was attributed to a controller in Mexico typing 9 (instead of 5) characters of flight-description data, resulting in a buffer overflow." (The Risks Digest: `http://cataless.ncl.ac.uk/Risks`)

Property Based: Buffer Overflows

- Hackers deliberately target any weaknesses in software, such as the potential for buffer overflows, in order to undermine the way that systems operate. Minimising such software vulnerabilities is therefore an important challenge.
- "Buffer overflows have been the most common form of security vulnerability in the last ten years" (DARPA Information Survivability Conference, 2000)

Cost Benefit Analysis

- Formal verification greatly increases confidence in a compliance argument.
- While retrospective verification is impractical, correctness by construction has been shown to work, *i.e. starting from design, software construction and correction go hand-in-hand.*
- Development time need not be increased, and there is evidence to show that significant reductions in testing and maintenance costs can be achieved.
- For wider acceptance greater tool automation is required, along with high-level guidance that informs decision making during design.

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 plug-in for a web browser?
- A key advantage of static analysis is that it can be applied before any code exists!

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`
- "Safety-Critical Computer Systems", Storey, N. Addison-Wesley, 1996.

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/`
- StaticSlicer: A simple Open Source tool that calculates static slices. `http://someslice.sourceforge.net`

References

- Z http://czt.sourceforge.net
- VDM: http://www.vdmportal.org/twiki/bin/view
- Event-B: http://www.event-b.org/
- Alloy: http://alloy.mit.edu/alloy/
- SPARK: http://www.altran-praxis.com/spark.aspx
- ESC/Java2: http://kindsoftware.com/products/opensource/ESCJava2/
- Spec#: http://research.microsoft.com/en-us/projects/specsharp/
- B-Method: http://www.bmethod.com