SOFTWARE COMPLEXITY METRICS IN GENERAL AND IN THE CONTEXT OF ISO 26262
SOFTWARE VERIFICATION REQUIREMENTS

MIROSLAW STARON, ASSOCIATE PROFESSOR, SOFTWARE ENGINEERING
CHALMERS | UNIVERSITY OF GOTHENBURG
Motivation for our research – safe cars

• The number of functions that are software steered grows as well
  – Autonomous driving >> 50 pure software functions

• Exponential growth of vehicle’s software size
  – The number of ECUs grows exponentially (2 ECUs in 1970 to over 130 in 2016)
  – The amount of software grows exponentially

• We face new challenges
  – How to verify and validate all the software?
  – How to increase sw dev. speed if the sw. complexity grows?

Source: www.software-center.se,
Outline of the talk

• Software complexity
  – Basic concepts
  – New scenarios for software use
  – New data sets available

• Overview of ISO 26262
  – Basic concepts
  – Software in ISO 26262
  – Software verification requirements

• Challenges for verifying and validating
  – ISO 26262 verification requirements linked to software verification techniques
  – Combining techniques to increase the level of verification and validation
SOFTWARE COMPLEXITY
Complexity in the software of modern cars

- **Software complexity**
  - The degree of connectivity between entities in a program

- **Metrics (examples)**
  - Cyclomatic complexity metric (McCabe)
  - Software science metrics (Halstead)
  - Software Structure Metrics (Henry and Kafura)
  - Metrics Suite for Object Oriented Design (Chidamber and Kamerer)
  - Branching complexity (Sneed)
  - Data access complexity (Card)
  - Data complexity (Chapin)
  - Data flow complexity (Elshof)
  - Decisional complexity (McClure)
Some of the most common complexity metrics, cont.

<table>
<thead>
<tr>
<th>Name of the Measure</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>McCabe’s cyclomatic complexity (1976)</td>
<td>The number of linearly independent paths in the control flow graph of code. This can be calculated by counting the number of control statements in the code</td>
</tr>
<tr>
<td>Halstead measures (1977)</td>
<td>7 measures completely based on number of operators and operands</td>
</tr>
<tr>
<td>Fan-out (Henry and Kafura 1981)</td>
<td>Number of unique invocations found in a given function</td>
</tr>
<tr>
<td>Fan-in (Henry and Kafura 1981)</td>
<td>Number of calls of a given function elsewhere in the code</td>
</tr>
<tr>
<td>Coupling measures of Henry and Kafura (1981)</td>
<td>Based on size, fan-in, and fan-out</td>
</tr>
<tr>
<td>Chidamber and Kemerer OO measures (1994)</td>
<td>Inheritance level and several size measures for class</td>
</tr>
<tr>
<td>Size measures</td>
<td>Lines of code, number of statements, etc.</td>
</tr>
<tr>
<td>Readability measures, e. g. Tenny (1988), Buse and Weimer (2010)</td>
<td>Line length, indentations, length of identifiers, etc.</td>
</tr>
</tbody>
</table>
How often are they used in industry?

Survey done by V. Antinyan, M. Staron, A. Sandberg, J. Hansson, in submission
Complexity of decision algorithms in practice (automotive)

Implications

• One control path => at least one test case
  – 511 for each path
  – test all combinations (theoretical) – anything between 511 and 1.5 * 10^{22}
  – In practice >> 1 trillion 10^{12} test cases is required due to co-dependency of test cases

• One control path => at least one fault injection
  – 511 injections

• One test case => one mutation
  – 511 – 1.5 * 10^{22} mutations
OVERVIEW OF ISO 26262
ROAD VEHICLES — FUNCTIONAL SAFETY
ISO 26262 – Functional Safety – Road vehicles

ISO 26262
– Chapter 6: Product development: software level
- Chapter 8, clause 9: Verification

Software complexity in ISO 26262
Chapter 6

- Data complexity
  - Data structures, classes, packets

- Code/control flow complexity
  - Algorithms, state machines, block diagrams

---

### Table 4 — Mechanisms for error detection at the software architectural level

<table>
<thead>
<tr>
<th>Methods</th>
<th>ASIL</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>A</td>
</tr>
<tr>
<td>1a. Range checks of input and output data</td>
<td>++</td>
</tr>
<tr>
<td>1b. Plausibility check</td>
<td>+</td>
</tr>
<tr>
<td>1c. Detection of data errors</td>
<td>+</td>
</tr>
<tr>
<td>1d. External monitoring facility</td>
<td>o</td>
</tr>
<tr>
<td>1e. Control flow monitoring</td>
<td>o</td>
</tr>
<tr>
<td>1f. Diverse software design</td>
<td>o</td>
</tr>
</tbody>
</table>

---

**Notes:**

- Plausibility checks can include using a reference model of the desired behaviour, assertion checks, or comparing signals from different sources.
- Types of methods that may be used to detect data errors include error detecting codes and multiple data storage.
- An external monitoring facility can be, for example, an ASIC or another software element performing a watchdog function.

---

### Table 6 — Methods for the verification of the software architectural design

<table>
<thead>
<tr>
<th>Methods</th>
<th>ASIL</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>A</td>
</tr>
<tr>
<td>1a. Walk-through of the design</td>
<td>++</td>
</tr>
<tr>
<td>1b. Inspection of the design</td>
<td>+</td>
</tr>
<tr>
<td>1c. Simulation of dynamic parts of the design</td>
<td>+</td>
</tr>
<tr>
<td>1d. Prototype generation</td>
<td>o</td>
</tr>
<tr>
<td>1e. Formal verification</td>
<td>o</td>
</tr>
<tr>
<td>1f. Control flow analysis</td>
<td>+</td>
</tr>
<tr>
<td>1g. Data flow analysis</td>
<td>+</td>
</tr>
</tbody>
</table>

---

**Notes:**

- In the case of model-based development these methods can be applied to the model.
- Method 1c requires the usage of executable models for the dynamic parts of the software architecture.
- Control and data flow analysis may be limited to safety-related components and their interfaces.
Overview of V&V requirements from ISO 26262

Software design and implementation

• Walkthrough
• Inspection
• Semi-formal verification
• Control-flow analysis
  – McCabe cyclomatic complexity
• Data-flow analysis
• Static code analysis
• Semantic code analysis

```c
void main(int a) {
    if (a == 0) {
    } else {
    }
}
```
COMBINING FAULT INJECTION AND MUTATION TESTING


Fault injection

Principles of mutation testing

• Exchange a piece of code into a different one
• Observe whether the change results in test cases failures
Mutation Testing

Principles of mutation testing

Figure: http://muclipse.sourceforge.net
Mutation testing
Overview of major techniques/tools
Summary

• Two take-aways
  – As the number of software functions (usage scenarios) increase in cars => complexity of the software increases
  – Testing for all possible execution paths becomes almost impossible => we need to test for subsets and understand how good our testing is

• Further directions
  – Software reliability growth modelling and latent defect inflow prediction
  – Combining formal verification with software testing
  – Using machine learning/search-based software testing to find the best testing combination for a given software functionality
Overview of V&V requirements from ISO 26262
Software design and implementation

• Walkthrough
• Inspection
• Semi-formal verification
• Control-flow analysis
• Data-flow analysis
• Static code analysis
• Semantic code analysis
Benefits of combining

• Assessment of the quality of software
  – We know if the software can handle problems with failures during the operation

• Assessment of the quality of the ”process” – or testing
  – We know if the test cases test the faulty programs
  – We know if we can trust the testing

• Where do we go from here
  – Software reliability assessment
Overview of V&V requirements from ISO 26262

Software design and implementation

- Walkthrough
- Inspection
- Semi-formal verification
- Control-flow analysis
- Data-flow analysis
- Static code analysis
- Semantic code analysis

- Efficiency
  - 125 source statement/hour during individual preparation
  - 90-125 statements/hour can be inspected during inspection meeting

- Inspection is therefore an expensive process
  - Inspecting 500 lines costs about 40 man/hours effort – about €2000