ISO 26262 SOFTWARE VERIFICATION REQUIREMENTS AND THEIR INFLUENCE ON AUTONOMOUS CARS

MIROSLAW STARON, ASSOCIATE PROFESSOR, SOFTWARE ENGINEERING
CHALMERS | UNIVERSITY OF GOTHENBURG
Outline of the talk

• Autonomous driving
  – Basic concepts
  – Software in autonomous driving
  – New scenarios for software use

• 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
AUTONOMOUS DRIVING
Software in the automotive domain

• Software is today at the heart of automotive development.
• A typical premium car has up to 150 ECUs, connected by several system buses to realize over 2000 functions
• 90% of all innovations are driven by electronics and software
• Many functions within automotive development are safety critical
What is an autonomous car?

NHSTA levels of autonomy

• Level 0: The driver completely controls the vehicle at all times.
• Level 1: Individual vehicle controls are automated
  – ESP, ABS.
• Level 2: At least two controls can be automated in unison
  – Adaptive cruise control in combination with lane keeping.
• Level 3: The driver can fully cede control of all safety-critical functions in certain conditions
  – The car senses when conditions require the driver to retake control and provides a "sufficiently comfortable transition time" for the driver to do so.
• Level 4: The vehicle performs all safety-critical functions for the entire trip, with the driver not expected to control the vehicle at any time.

Source: https://en.wikipedia.org/wiki/Autonomous_car
Exemplary functions of autonomous cars

- Adaptive Cruise Control
- Lane Keeping Assistance
- Active Lane Change Assistance
- Active Traffic Light Assistance
- Traffic Jam Chauffeur
- Highway Chauffeur
- Platooning
- Highway Pilot
- Overtaking Pilot

Source: http://www.adaptive-ip.eu/
Complexity in autonomous cars

- Scenario complexity
- Problem domain
- Data complexity
- Sensor complexity
- Algorithm complexity
- Decision complexity
- Software domain
- Tooling complexity
- Process domain

GÖTEBORGS
UNIVERSITET

WWW.STARON.NU
Top 3 obstacles for autonomous driving

• Liability for damage
• Resistance by individuals to forfeit control of their cars
• Software reliability
  – Definition
    ▪ *Software Reliability is the probability of failure-free software operation for a specified period of time in a specified environment*
  – Challenges
    ▪ Measurement of reliability
    ▪ Assessment of processes
    ▪ Trust in the evidence
    ▪ Formalization of reliability
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

ISO 26262 – an Overview

3. Concept phase
   3a. Item definition
   3b. Initiation of safety lifecycle
   3c. Hazard analysis and preliminary risk assessment
   3d. Assign functional safety requirements

4. Product development at system level
   4a. Initiation of product development at system level
   4b. Specification of technical safety requirements
   4c. System design
   4d. Item integration and testing
   4e. Safety validation
   4f. Functional safety assessment
   4g. Release for production

5. Product development at hardware level
   5a. Initiation
   5b. Specification of hardware safety requirements
   5c. Hardware architectural metrics
   5d. Evaluation of violations of safety goal due to random hardware failures
   5e. Hardware integration and testing

6. Product development at software level
   6a. Initiation
   6b. Specification of software safety requirements
   6c. Software architectural metrics
   6d. Software unit design and implementation
   5e. Unit testing
   6f. Software integration and testing
   6g. Verification of software safety requirements

7. Production and operation
   7a. Production
   7b. Operation, service and decommissioning
## ISO 26262 - Road vehicles -- Functional safety

<table>
<thead>
<tr>
<th>ISO 26262 Chapter</th>
<th>Reference to recommendation</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>Hardware-software integration and testing</td>
</tr>
<tr>
<td></td>
<td>• Table 5 — Correct implementation of technical safety requirements at the hardware-software level.</td>
</tr>
<tr>
<td></td>
<td>• Table 8 — Effectiveness of a safety mechanism’s diagnostic coverage at the hardware-software level.</td>
</tr>
<tr>
<td></td>
<td>System integration and testing</td>
</tr>
<tr>
<td></td>
<td>• Table 10a — Correct implementation of functional safety and technical safety requirements at the system level</td>
</tr>
<tr>
<td></td>
<td>• Table 13b — Effectiveness of a safety mechanism's failure coverage at the system level</td>
</tr>
<tr>
<td></td>
<td>Vehicle integration and testing</td>
</tr>
<tr>
<td></td>
<td>• Table 15 — Correct implementation of the functional safety requirements at the vehicle level</td>
</tr>
<tr>
<td></td>
<td>• Table 18 — Effectiveness of a safety mechanism's failure coverage at the vehicle level</td>
</tr>
<tr>
<td>5</td>
<td>Hardware integration and testing</td>
</tr>
<tr>
<td></td>
<td>• Table 11 — Hardware integration tests to verify the completeness and correctness of the safety mechanisms implementation with respect to the hardware safety requirements</td>
</tr>
<tr>
<td>6</td>
<td>Software unit testing</td>
</tr>
<tr>
<td></td>
<td>• Table 10 — Methods for software unit testing</td>
</tr>
<tr>
<td></td>
<td>Software integration and testing</td>
</tr>
<tr>
<td></td>
<td>• Table 13 — Methods for software integration testing</td>
</tr>
</tbody>
</table>
ISO 26262 & Software V&V

• Chapter 6 requires a number of verification activities based on
  – Work product
  – ASIL level
Overview of V&V requirements from ISO 26262

Software architectural design

• Walkthrough of the design
• Inspection of the design
• Simulation of dynamic parts of the design
• Prototype generation
• Formal verification
• Data flow analysis

<table>
<thead>
<tr>
<th>Data faults</th>
<th>Are all program variables initialised before their values are used?</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Have all constants been named?</td>
</tr>
<tr>
<td></td>
<td>Should the upper bound of arrays be equal to the size of the array or Size -1?</td>
</tr>
<tr>
<td></td>
<td>If character strings are used, is a de limiter explicitly assigned?</td>
</tr>
<tr>
<td></td>
<td>Is there any possibility of buffer overflow?</td>
</tr>
</tbody>
</table>
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
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 {
    }
}
```
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
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

Size and complexity (color) or components from the tool Sonar
Overview of V&V requirements from ISO 26262

Software unit and integration testing

• Requirements-based test
• Interface test
• Fault injection test
• Resource usage test
• Back-to-back comparison test between model and code (if applicable)
COMBINING FAULT INJECTION AND MUTATION TESTING


Fault injection

Principles of fault injection

• Introduce a known fault into an existing design/operation
• Observe whether the fault propagate over the system
• Observe if the faults result in failures
Fault Injection
Overview of major techniques/tools
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
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
IMPACT ON AUTONOMOUS CARS
Example scenario
Pedestrian detection

- What do we see?
- Is there a pedestrian in the image?
- What are the other objects?
- What is our context?
- What is the scenario?
- What should we do?
- Act upon the decision
Pedestrian detection

Complexity and certainty

- **Image recognition**
- **Object identification**
- **Decision making**
- **Actuation**

**Complexity**

- Depends on
  - External/weather conditions
  - Sensor quality
  - Image processing algorithm

- Depends on
  - Scenario
  - Speed
  - Hazard cl.

- Depends on
  - Hw. components

**Certainty**
Complexity of decision algorithms in practice

Implications

• One control path => at least one test case
  – 511 for each path
  – $1.5 * 10^{22}$ to test all combinations (theoretical)

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

• One test case => one mutation
  – 511 – $1.5 * 10^{22}$ mutations
Summary

• Two take-aways
  – As the number of software functions (usage scenarios) increase in autonomous 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