Automated Testing of Autonomous Driving Assistance Systems

Lionel Briand

TrustMeIA, 2019
Acknowledgments

Raja
Ben Abdessalem

Shiva Nejati

Annibale
Panichella
Introduction
Cyber-Physical Systems

- A system of collaborating computational elements controlling physical entities
- Increasingly autonomous
- ML-based, e.g., DNN in perception layer
- Not just recommendations, act on the physical environment
- Safety implications
CPS Development Process

Model-in-the-Loop Stage

Functional modeling:
- Controllers
- Plant
- Decision

Continuous and discrete Simulink models

Model simulation and testing

Software-in-the-Loop Stage

Architecture modelling
- Structure
- Behavior
- Traceability

System engineering modeling (SysML)

Analysis:
- Model execution and testing
- Model-based testing
- Traceability and change impact analysis
- ...

(partial) Code generation

Hardware-in-the-Loop Stage

Deployed executables on target platform

Hardware (Sensors ...)
Analog simulators

Testing (expensive)
Formal Verification

- **Significant Research Formal verification**: Exhaustive, guarantees (deterministic or statistical) [Sheshia et al. 2017]

- Models of system, environment, and properties in formalisms for which there are **efficient decision procedures**.

- **Challenges**: Undecidability, assumptions, scalability.

Seshia et al., Towards Verified AI
Limitations of Formal Verification

- Complexity and heterogeneity of CPS
- Non-Boolean properties
- Models capture continuous dynamics
- Not always easy or feasible to translate into low-level logic-based languages
- Models contain constructs that are not easy to handle such as non-algebraic arithmetics, third-party code (S-Functions).
Testing

• Focus on falsification (testing) and explanation.
  • Automated testing with a focus on safety violations
  • Explanation of safety violations ➔ Risk assessment

• Testing takes place at different phases of development in CPS: MiL, SiL, HiL.

• Testing does not prove satisfiability: It finds safety violations or provide assurance cases (evidence)
Testing Cyber-Physical Systems

- **MiL and SiL testing**: Computationally expensive (simulation of physical models)

- **HiL**: Human effort involved in setting up the hardware and analog simulators

- **Number of test executions tends to be limited** compared to other types of systems

- Test input space is often **extremely large**, i.e., determined by the complexity of the physical environment

- **Traceability** between system testing and requirements is mandated by standards
Testing Advanced Driving Assistance Systems
Problem Definition and State of Practice
Advanced Driver Assistance Systems (ADAS)

- Automated Emergency Braking (AEB)
- Lane Departure Warning (LDW)
- Pedestrian Protection (PP)
- Traffic Sign Recognition (TSR)
Advanced Driver Assistance Systems (ADAS)

Decisions are made over time based on sensor data
Automotive Environment

- Highly varied environments, e.g., road topology, weather, building and pedestrians …

- Huge number of possible scenarios, e.g., determined by trajectories of pedestrians and cars

- ADAS play an increasingly critical role in modern vehicles

- Systems must comply with functional safety standards, e.g., ISO 26262

- A challenge for testing
ISO 26262

- Defines the vehicle safety as the absence of unreasonable risks that arise from **malfunctions of the system**.

- Requires **safety goals**, necessary to mitigate the risks.

- Provides requirements and recommendations to avoid and control random **hardware failures** and **systematic system failures** that could violate safety goals.
SOTIF

• ISO/PAS standard: Safety of the intended functionality (SOTIF).

• Autonomy: Huge increase in functionalities relying on advanced sensing, algorithms (ML), and actuation.

• SOTIF accounts for limitations and risks related to nominal performance of sensors and software:

  • The inability of the function to correctly comprehend the situation and operate safely; this also includes functions that use machine learning algorithms;

  • Insufficient robustness of the function with respect to sensor input variations or diverse environmental conditions.
SOTIF: Scenes and Scenarios

Domain Conceptual Model (Terminology) [9]
Temporal view of scenes, events, actions and situations in a scenario
Verification in SOTIF

- **Decision algorithm verification**: “... ability of the decision-algorithm to react when required and its ability to avoid unwanted action.”

- **Integrated system verification**: “Methods to verify the robustness and the controllability of the system integrated into the vehicle ...”
## Decision Algorithm Verification

### Table 6: Decision Algorithm verification

<table>
<thead>
<tr>
<th>Methods</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>A</strong> Verification of robustness to interference from other sources, e.g. white noise, audio frequencies, Signal-to-Noise Ratio degradation (e.g. by noise injection testing)</td>
</tr>
<tr>
<td><strong>B</strong> Requirement-based test (e.g. classification, sensor data fusion, situation analysis, function)</td>
</tr>
<tr>
<td><strong>C</strong> Verification of the architectural properties including independence, if applicable</td>
</tr>
<tr>
<td><strong>D</strong> In the loop testing (e.g. SIL / HIL / MIL) on selected SOTIF relevant use cases and scenarios</td>
</tr>
<tr>
<td><strong>E</strong> Vehicle level testing on selected SOTIF relevant use cases and scenarios</td>
</tr>
<tr>
<td><strong>F</strong> Inject inputs into the system that trigger potentially hazardous behaviour</td>
</tr>
<tr>
<td><strong>NOTE</strong> For test case derivation the method of combinatorial testing can be used [5]</td>
</tr>
</tbody>
</table>
**Table 8: Integrated system verification**

<table>
<thead>
<tr>
<th>Methods</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Verification of robustness to Signal-to-Noise Ratio degradation (e.g. by noise injection testing)</td>
</tr>
<tr>
<td>B</td>
<td>Requirement-based Test when integrated within the vehicle environment (e.g. range, precision, resolution, timing constraints, bandwidth)</td>
</tr>
<tr>
<td>C</td>
<td>In the loop testing (e.g. SIL / HIL / MIL) on selected SOTIF relevant use cases and scenarios</td>
</tr>
<tr>
<td>D</td>
<td>System test under different environmental conditions (e.g. cold, damp, light, visibility conditions)</td>
</tr>
<tr>
<td>E</td>
<td>Verification of system ageing affects. (e.g. accelerated life testing)</td>
</tr>
<tr>
<td>F</td>
<td>Randomized input tests a)</td>
</tr>
<tr>
<td>G</td>
<td>Vehicle level testing on selected SOTIF relevant use cases and scenarios</td>
</tr>
<tr>
<td>H</td>
<td>Controllability tests (including reasonably foreseeable misuse)</td>
</tr>
</tbody>
</table>

a) Randomized input tests can include erroneous patterns e.g. in the case of image sensors adding flipped images or altered image patches; or in the case of radar sensors adding ghost targets to simulate multi-path returns.
How to perform such testing efficiently and effectively?
Research is needed
MiL Testing of ADAS
Recent Project

- Developing an automated testing technique for ADAS
- To help engineers efficiently and effectively *explore* the complex scenario space of ADAS
- To *identify* critical, failure-revealing test scenarios
- **Characterization of input conditions** that lead to most critical scenarios, e.g., safety violations
Automated Emergency Braking System (AEB)

Objects’ position/speed

Vision (Camera)

Sensor

“Brake-request” when braking is needed to avoid collisions

Brake Controller

Decision making
Example Critical Situation

• “AEB properly detects a pedestrian in front of the car with a high degree of certainty and applies braking, but an accident still happens where the car hits the pedestrian with a relatively high speed”
Testing ADAS

On-road testing

- Time-consuming
- Expensive
- Unsafe

Simulation-based (model) testing

A simulator based on physical/mathematical models
Testing via Physics-based Simulation

- ADAS (SUT)
- Simulator (Matlab/Simulink)
- Model (Matlab/Simulink)

- Physical plant (vehicle / sensors / actuators)
- Other cars
- Pedestrians
- Environment (weather / roads / traffic signs)

Test input

Test output
time-stamped output
ADAS Testing Challenges

• Test input space is **multidimensional, large, and complex**

• *Explaining failures and fault localization* are difficult

• Execution of **physics-based simulation models** is computationally expensive
Our Approach

• We use decision tree classification models

• We use multi-objective search algorithm (NSGAII)

• **Objective Functions:**
  1. Minimum distance between the pedestrian and the field of view
  2. The car speed at the time of collision
  3. The probability that the object detected is a pedestrian

• Each search iteration calls simulation to compute objective functions
Individual A Pareto dominates individual B if A is at least as good as B in every objective and better than B in at least one objective.

- A multi-objective optimization algorithm (e.g., NSGA II) must:
  - Guide the search towards the global Pareto-Optimal front.
  - Maintain solution diversity in the Pareto-Optimal front.
Search-based Testing Process

Input data ranges/dependencies + Simulator + Fitness functions

Test input generation (NSGA II)
- Select best tests
- Generate new tests

Evaluating test inputs
- Simulate every (candidate) test
- Compute fitness functions

(candidate) test inputs
Fitness values

Test cases revealing worst case system behaviors
Search: Genetic Evolution

Initial input
Fitness computation
Selection
Breeding
Better Guidance

- Fitness computations rely on simulations and are very expensive
- Search needs better guidance
Partition the input space into homogeneous regions
Genetic Evolution Guided by Classification

- Initial input
- Fitness computation
- Classification
- Selection
- Breeding

38
Search Guided by Classification

Input data ranges/dependencies + Simulator + Fitness functions

Test input generation (NSGA II)

- Build a classification tree
- Select/generate tests in the fittest regions
- Apply genetic operators

Evaluating test inputs

- Simulate every (candidate) test
- Compute fitness functions

(candidate) test inputs

Fitness values

Test cases revealing worst case system behaviors + A characterization of critical input regions
NSGAII-DT vs. NSGAII

NSGAII-DT outperforms NSGAII
Generated Decision Trees

The generated critical regions consistently become smaller, more homogeneous and more precise over successive tree generations of NSGAII-DT
Usefulness

• The characterizations of the different critical regions can help with:

(1) **Debugging** the system model

(2) **Identifying possible hardware changes** to increase ADAS safety

(3) **Providing proper warnings** to drivers
Testing for Feature Interactions
Integration of Autonomous Features in ADAS

Automated Emergency Braking (AEB)

Lane Departure Warning (LDW)

Pedestrian Protection (PP)

Traffic Sign Recognition (TSR)
Actuator Commands:
- Steering
- Acceleration
- Braking

Undesired Feature Interactions
(Early) Function Modeling

Software Under Test (SUT)

Integration Component

Executable Function Models (Matlab/Simulink)
## Example of Rules

### Conditions

1. if \( TTC < TTC_{th} \) \& (\( Dist(P/Car) < Dist_{th} \)) \& (object detected is pedestrian)

2. if (none of TSR rules are activated) \& (speed of the car < speed of the leading car)

3. if (speed of the car > speed limit)

4. if \( TTC < TTC_{th} \) \& (object detected is not pedestrian)

5. if \( Dist(P/Car) < Dist_{th} \)

6. if (none of PP rules are activated) \& (there is a STOP sign)

### Features

- PP
- ACC
- TSR
- AEB
- PP
- TSR
Main challenge

• Resolving conflicts:
  • Difficult task
  • Requires extensive domain expertise and a thorough analysis of system requirements

• Problems:
  • There might be mistakes, inaccuracies and bugs in the developed rules
Problem Statement

How to automatically detect undesired feature interactions in self-driving systems at early stages of development?
Testing Function Models

Physics-based Simulators

SUT

sensor/camera data

Actuator commands

environment (road, weather, etc)

actuators

other cars

(ego) car

pedestrians

Time Stamped Vectors
Search-Based Solution

- We do not know a priori which safety requirements may be violated. Neither do we know in which branches of IntC the violations may be detected. Therefore, we search for any violation of system safety requirements that may arise when exercising any branch of IntC.

- A combination of three test objectives:
  - Coverage-based
  - Failure-based
  - Unsafe Overriding
Coverage-based Test Objective

Goal: Exercising as many branches of the integration component as possible
Branch Distance

- Many decision branches in IntC
- Branch coverage of IntC
- **Fitness**: Approach level and branch distance $d$ (standard for code coverage)
- $d(b, tc) = 0$ when $tc$ covers $b$
Failure-based Test Objective

Goal: Revealing violations of system-level requirements

Example:
- Requirement: No collision between pedestrians and cars
- Generating test cases that minimize the distance between the car and the pedestrian
Fitness: Failure Distance

- **Fitness functions** based on the trajectory vectors for the ego car, the leading car and the pedestrian, generated by the simulator.

- **PP fitness**: Minimum distance between the car and the pedestrian during the simulation time.

- **AEB fitness**: Minimum distance between the car and the leading car during the simulation time.
# Distance Functions

<table>
<thead>
<tr>
<th>Feature</th>
<th>Requirement</th>
<th>Failure distance functions ( (FD_1, \ldots, FD_5) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( PP )</td>
<td>No collision with pedestrians</td>
<td>( FD_1(i) ) is the distance between the ego car and the pedestrian at step ( i ).</td>
</tr>
<tr>
<td>( AEB )</td>
<td>No collision with cars</td>
<td>( FD_2(i) ) is the distance between the ego car and the leading car at step ( i ).</td>
</tr>
<tr>
<td>( TSR )</td>
<td>Stop at a stop sign</td>
<td>Let ( u(i) ) be the speed of the ego car, at time step ( i ), once it reaches a stop sign. If there is no stop sign, then ( u(i) = 0 ). We define ( FD_3(i) = 0 ) if ( u(i) \geq 20 \text{km/h} ). Otherwise, we define ( FD_3(i) = \frac{1}{u(i)} ). If there is no stop sign, we have ( FD_3(i) = 1 ).</td>
</tr>
<tr>
<td>( TSR )</td>
<td>Respect the speed limit</td>
<td>Let ( u'(i) ) be the difference between the speed of the ego car and the speed limit at step ( i ) if a speed limit sign is detected. If there is no speed limit sign ( u'(i) = 0 ). We define ( FD_4(i) = 0 ) if ( u(i) \geq 20 \text{km/h} ). Otherwise, we define ( FD_4(i) = \frac{1}{u'(i)} ). If there is no speed limit sign, we have ( FD_4(i) = 1 ).</td>
</tr>
<tr>
<td>( ACC )</td>
<td>Respect the safety distance</td>
<td>( FD_5(i) ) is the absolute difference between the safety distance ( sd ) and ( FD_2(i) ).</td>
</tr>
</tbody>
</table>

When any of the functions yields zero, a safety failure corresponding to that function is detected.
Unsafe Overriding Test Objective

Goal: Finding failures that are more likely to be due to faults in the integration component rather faults in the features

Reward failures that could have been avoided if another feature had been prioritized by the integration logic
Combining Test Objectives

• **Goal:** Execute every branch of the integration component such that while executing that branch, the component unsafely overrides every feature \( f \) and its outputs violate every safety requirement related to \( f \).

For every time step \( i \), every branch \( j \) and every requirement \( l \):

**If** branch \( j \) is not covered:

\[
\Omega_{j,l}(i) = \text{Branch}_j(i) + \max(\text{Overriding}) + \max(\text{Failure})
\]

**ELSE If** feature \( f \) is not unsafely overrides:

\[
\Omega_{j,l}(i) = \text{Overriding}_f(i) + \max(\text{Failure})
\]

**ELSE**

\[
\Omega_{j,l}(i) = \text{Failure}_l(i)
\]

\[
\Omega_{j,l} = \min\{\Omega_{j,l}(i)\}_{0 \leq i \leq T}
\]
Our Hybrid Test Objectives

One hybrid test objective $\Omega_{j,l}$ for every branch $j$ and every requirement $l$:

1. $\Omega_{j,l}(tc) > 2$ → $tc$ does not cover Branch $j$
2. $2 \geq \Omega_{j,l}(tc) > 1$ → $tc$ covers branch $j$ but $F$ is not unsafely overridden
3. $1 \geq \Omega_{j,l}(tc) > 0$ → $tc$ covers branch $j$ and $F$ is unsafely overridden but req $l$ is not violated
4. $\Omega_{j,l}(tc) = 0$ → A feature interaction failure is likely detected
Search Algorithm

• Optimal test suite covers all search objectives, i.e., for all IntC branches and all safety requirements

• The number of test objectives is large:

$$\text{# of requirements} \times \text{# of branches}$$

• Computing test objectives is computationally expensive

• Not a Pareto front optimization problem

• Objectives compete with each others, e.g., cannot have the car violating the speed limit after hitting the leading car in one test case
MOSA: Many-Objective Search-based Test Generation

Not all (non-dominated) solutions are optimal for the purpose of testing.

These points are better than others.

Panichella et. al. [ICST 2015]
Case Study

• Two Case Study systems from IEE

• Both systems consist of four self-driving features
  
  • Adaptive Cruise Control (ACC)
  
  • Automated Emergency Braking (AEB)
  
  • Traffic Sign Recognition (TSR)
  
  • Pedestrian Protection (PP)

• But, they use different rules to integrate feature actuator commands

• RQ: Does our Hybrid test objectives reveal more feature interaction failures compared to baseline test objectives (coverage-based and failure-based)?
Hybrid test objectives reveal significantly more feature interaction failures (more than twice) compared to the baseline alternatives.
Feedback from Domain Experts

• The failures we found were due to undesired feature interactions

• The failures were not previously known to them

• We identified ways to improve the feature integration logic to avoid failures
Problem: How to automatically detect undesired feature interactions in self-driving systems at early stages of development?

Context: Executable Function models and Simulated Environment

Approach: A search-based testing approach

- Hybrid test objectives (coverage-based, failure-based, unsafe overriding)
- A tailored many-objective search algorithm

We have evaluated and validated our approach: Promising results
Repairing Feature Interaction Failures
Problem Statement

- How to **automatically repair** (some of the errors in) the feature integration logic?
Types of errors in the developed Rules

- The conjunctive *conditions* on the left side of the rules may be wrong

- Issues in determining the *order* of conflict resolution rules

- There is a partial order over rules

- Different rule orderings lead to different system behaviors and it is *not clear what order should be used*

<table>
<thead>
<tr>
<th>Rule</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. if ((TTC &lt; TTC_{th}) \land (Dist(P/Car) &lt; Dist_{th}) \land \text{(object detected is pedestrian)})</td>
<td>(\rightarrow) PP</td>
</tr>
<tr>
<td>2. if (none of TSR rules are activated) \land (speed of the car &lt; speed of the leading car)</td>
<td>(\rightarrow) ACC</td>
</tr>
<tr>
<td>3. if (speed of the car &gt; speed limit)</td>
<td>(\rightarrow) TSR</td>
</tr>
<tr>
<td>4. if ((TTC &lt; TTC_{th}) \land \text{(object detected is not pedestrian)})</td>
<td>(\rightarrow) AEB</td>
</tr>
<tr>
<td>5. if (Dist(P/Car) &lt; Dist_{th})</td>
<td>(\rightarrow) PP</td>
</tr>
<tr>
<td>6. if (none of PP rules are activated) \land (there is a STOP sign)</td>
<td>(\rightarrow) TSR</td>
</tr>
</tbody>
</table>
Our Solution

- **A Search-based Fault Localization and Repairing Approach for the Decision Rules of Self-Driving Systems**

- **Many-objective search**: Minimize severity of failures until all test cases pass
Program Repair

Goal: automatically repair software systems

- Repair Tests: Failing test cases (demonstrate a defect) + Passing test cases (satisfy system behavior)

- Program repair process:
  - Step 1 - Fault Localization: identifies the locations of faults
  - Step 2 - Patch generation: modifies the software in the faulty code locations
  - Step 3 – Program validation: checks if the synthesized patch has actually repaired the software
State of the Art: Challenges

- Our errors span several lines of code
- Each test case covers most paths over simulation steps
- High suspiciousness for most statements
Our Approach

- **Goal:** Identify errors in the decision rules for self-driving systems and automatically repair these errors

**Fault Localization (FL)**
Focus on faulty statements that are related to the most severe failures

**Patch Generation**
Define mutation operators to modify the decision rules
FL - Illustrations

Faulty Program

if condition1(s1)
    if condition11 (s2)
        statement condition11(s3)
    end
else if condition2(s4)
    statement condition2(s5)
else if condition3(s6)
    if condition11 (s7)
        statement condition31(s8)
    end
else if condition4(s9)
    statement condition4(s10)
else if condition5(s11)
    statement condition5(s12)
else if condition6(s13)
    statement condition6(s14)
: :

Decision rule 4: Conjunctive conditions

Test Suite

tc1

tc2

tc3

Simulation time

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>...</th>
<th>t</th>
<th>...</th>
<th>T</th>
</tr>
</thead>
<tbody>
<tr>
<td>s1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>s2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>s3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>s4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>s5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>s6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>s7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>sn</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

x : if s is executed

Features

FL - Illustrations
**FL - Illustrations**

![Diagram showing a test suite with a faulty program and a simulation time table with statements and status marked for each test case.](image)

**Faulty Program**

```plaintext
if condition1(s1)
  if condition11 (s2)
    statement condition11(s3)
  end
else if condition2(s4)
  statement condition2(s5)
else if condition3(s6)
  if condition11 (s7)
    statement condition31(s8)
  end
else if condition4(s9)
  statement condition4(s10)
else if condition5(s11)
  statement condition5(s12)
else if condition6(s13)
  statement condition6(s14)
```

**Simulation time**

<table>
<thead>
<tr>
<th>Statements</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>...</th>
<th>t_j</th>
<th>...</th>
<th>T</th>
</tr>
</thead>
<tbody>
<tr>
<td>s1</td>
<td></td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>s2</td>
<td></td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>s3</td>
<td></td>
<td></td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>s4</td>
<td></td>
<td></td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>s5</td>
<td></td>
<td></td>
<td></td>
<td>x</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>s6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>x</td>
<td></td>
<td></td>
</tr>
<tr>
<td>s7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>sn</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>x</td>
</tr>
</tbody>
</table>

**Status**

<table>
<thead>
<tr>
<th>Test Case</th>
<th>tc1</th>
<th>tc2</th>
<th>tc3</th>
</tr>
</thead>
<tbody>
<tr>
<td>tc1</td>
<td>x</td>
<td></td>
<td></td>
</tr>
<tr>
<td>tc2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>tc3</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
FL - Illustrations

Test Suite

Faulty Program

```
if condition1(s1)
    if condition11 (s2)
        statement condition11(s3)
    end
else if condition2(s4)
    statement condition2(s5)
else if condition3(s6)
    if condition11 (s7)
        statement condition31(s8)
    end
else if condition4(s9)
    statement condition4(s10)
else if condition5(s11)
    statement condition5(s12)
else if condition6(s13)
    statement condition6(s14)
... ...
```

Simulation time

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>...</th>
<th>t</th>
<th>...</th>
<th>T</th>
</tr>
</thead>
<tbody>
<tr>
<td>s1</td>
<td></td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>s2</td>
<td></td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>s3</td>
<td></td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>s4</td>
<td></td>
<td></td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>s5</td>
<td></td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>s6</td>
<td></td>
<td></td>
<td></td>
<td>x</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>s7</td>
<td></td>
<td></td>
<td></td>
<td>x</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>sn</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>x</td>
<td>x</td>
</tr>
</tbody>
</table>

x : if s is executed

Status

<table>
<thead>
<tr>
<th></th>
<th>tc1</th>
<th>tc2</th>
<th>tc3</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>X</td>
</tr>
</tbody>
</table>
FL - Illustrations

Faulty Program

```
if condition1(s1)
  if condition11 (s2)
    statement condition11(s3)
  end
else if condition2(s4)
  statement condition2(s5)
else if condition3(s6)
  if condition11 (s7)
    statement condition11(s8)
  end
else if condition4(s9)
  statement condition4(s10)
else if condition5(s11)
  statement condition5(s12)
else if condition6(s13)
  statement condition6(s14)
: :
```

Rule executed at the time of failure

\[ \pi_{tc3} \]

Test Suite

Simulation time

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>...</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>...</th>
<th>T</th>
</tr>
</thead>
<tbody>
<tr>
<td>s1</td>
<td></td>
<td>x</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>s2</td>
<td></td>
<td>x</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>s3</td>
<td></td>
<td>x</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>s4</td>
<td>x</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>s5</td>
<td>x</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>s6</td>
<td></td>
<td></td>
<td></td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>x</td>
</tr>
<tr>
<td>s7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>x</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>sn</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>x</td>
<td></td>
</tr>
</tbody>
</table>

Status

<table>
<thead>
<tr>
<th>tc1</th>
<th>tc2</th>
<th>tc3</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td></td>
<td>X</td>
</tr>
</tbody>
</table>
Ranking

- We define the suspiciousness of each statement $s$ by considering both failure severity and faulty rules

$$Score(s) = \sum_{tc_j \in TS} I_{tc_j}(s)$$

$$I_{tc_j} = \begin{cases} 
0 & \text{if } s \notin \text{ faulty rule } \pi_{tc_j} \\
\omega_{tc_j} & \text{otherwise}
\end{cases}$$

$\omega_{tc_j}$: severity of failure exposed by $tc_j$

- We select a statement among the most suspicious ones in a randomized way

<table>
<thead>
<tr>
<th>Rules</th>
<th>Score</th>
</tr>
</thead>
<tbody>
<tr>
<td>s1</td>
<td>0.9</td>
</tr>
<tr>
<td>s2</td>
<td>0.9</td>
</tr>
<tr>
<td>s3</td>
<td>0.9</td>
</tr>
<tr>
<td>s6</td>
<td>0.6</td>
</tr>
<tr>
<td>s7</td>
<td>0.6</td>
</tr>
<tr>
<td>s8</td>
<td>0.6</td>
</tr>
<tr>
<td>s4</td>
<td>0</td>
</tr>
<tr>
<td>s5</td>
<td>0</td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
</tbody>
</table>
Probabilistic Fault Localization

• For each failure, select in a randomized manner a statement among most suspicious ones

• Select different statements in a probabilistic manner (suspiciousness) across search interactions

• Generate and apply a patch
Generate a Patch

- Mutation operators
  1. **Modify** operator
  2. **Shift** operator
- Apply a sequence of randomly selected operators
- Evaluate the patch (run test simulations)
Modify

- **Modify** operator: modify the condition of the faulty statement by changing a threshold value, altering the direction of a relational operator (e.g., > to <) or swapping arithmetic operations (e.g., + to - )

```plaintext
if condition1(s1)
    if condition11 (s2)
        statement condition11(s3)
    end
else if condition2(s4)
    statement condition2(s5)
else if condition3(s6)
    if condition11 (s7)
        statement condition31(s8)
    end
else if condition4(s9)
    statement condition4(s10)
else if condition5(s11)
    statement condition5(s12)
else if condition6(s13)
    statement condition6(s14)
:
:
```

Example

1. \( \text{if} \) \( \text{TTC} < \text{TTC}_{th} \) \&\& \( \text{Dist}(P/C) < \text{Dist}(P/C)_{th} \) \&\& object detected is pedestrian\)
2. Apply PP (braking force = 100%)
3. else if (speed of the car > speed limit)
4. Apply TSR (braking force = 40%)
5. else if (TTC < TTC_{th} \&\& object detected is not pedestrian)
6. Apply AEB (braking force = 60%)
7. else if ...
   ..... 
8. n:    ..... 

Shift

- **Shift operator**: Change the order of rules

```plaintext
if condition1(s1)
    if condition11 (s2)
        statement condition11(s3)
    end
else if condition2(s4)
    statement condition2(s5)
else if condition3(s6)
    if condition11 (s7)
        statement condition31(s8)
    end
else if condition4(s9)
    statement condition4(s10)
else if condition5(s11)
    statement condition5(s12)
else if condition6(s13)
    statement condition6(s14)
    ...
```

```plaintext
if condition1(s1)
    if condition11 (s2)
        statement condition11(s3)
    end
else if condition2(s4)
    statement condition2(s5)
else if condition3(s6)
    statement condition3(s7)
    statement condition31(s8)
endif
else
    condition5(s11)
    statement condition31(s12)
    statement condition2(s5)
else if condition3(s6)
    statement condition3(s7)
    statement condition31(s8)
else if condition4(s9)
    statement condition4(s10)
else if condition5(s11)
    statement condition5(s12)
else if condition6(s13)
    statement condition6(s14)
    ...
```
Our Search-based Repairing Approach

Evaluate the Patch

Faulty Program

Test Cases

Faulty Program

Archive

Localize Fault

Generate a Patch

Evaluate the Patch

yes

select a patch from the archive

update the archive

update the archive

yes

good?

Best Patched Programs

Repair Program

select a patch from the archive

update the archive

update the archive

yes

good?
Preliminary evaluation

• RQ: How effective and efficient is our approach in localizing and repairing faults?
Results

• Able to repair the decision rules (change the order of the rules and change some threshold values)

• Able to find different correct versions of the decision rules

Our approach takes **on average 5 hours** to repair the decision rules of self-driving systems
Summary

• We proposed an automated approach to repair errors in the decision rules for self-driving systems
  
  • A many-objective search-based testing approach

• Our approach effectively and efficiently localizes and repairs errors in decision rules

• Limitations: Cannot repair all kinds of errors, e.g., adding or removing rules
Conclusions
Other CPS Domains

- Similar ISO standards
- With the rise of DNNs, they will develop SOTIF-like standards
- Most CPS are safety critical
- Most CPS follow similar development phases
- Different domain models, e.g., notion of scenes and scenarios
- Different, dedicated simulators
Reflections

- Search-based solutions are versatile
- Help relax assumptions compared to exact approaches
- Help decrease modeling requirements
- Scalability, e.g., easy to parallelize
- Require massive empirical studies for validation
- **Search is rarely sufficient by itself**
Reflections

• Combine with **machine learning** to make the search more effective and efficient

• Combine search with **solvers**, e.g., SMT, model checking, Constraint programming, on model fragments.

• **Grey box approaches** to search, i.e., use information from the Simulink models to better guide search.

• **Impact of deep learning** components in autonomous systems’ testing and verification
The Impact of DNNs

- Increasingly so, it is easier to learn behavior from data using machine learning, rather than specify and code

- Some ADAS components rely on deep learning ...

- Thousands of weights learned (Deep Neural Networks)

- No explicit code, no specifications

- Verification, testing?

- State of the art includes adequacy coverage criteria and mutation testing for DNNs
Selected References


• Ben Abdessalem et al., "Testing Advanced Driver Assistance Systems using Multi-objective Search and Neural Networks, ASE 2016

• Ben Abdessalem et al., "Testing Vision-Based Control Systems Using Learnable Evolutionary Algorithms”, ICSE 2018

• Ben Abdessalem et al., "Testing Autonomous Cars for Feature Interaction Failures using Many-Objective Search”, ASE 2018
Automated Testing of Autonomous Driving Assistance Systems

Lionel Briand

TrustMeIA, 2019