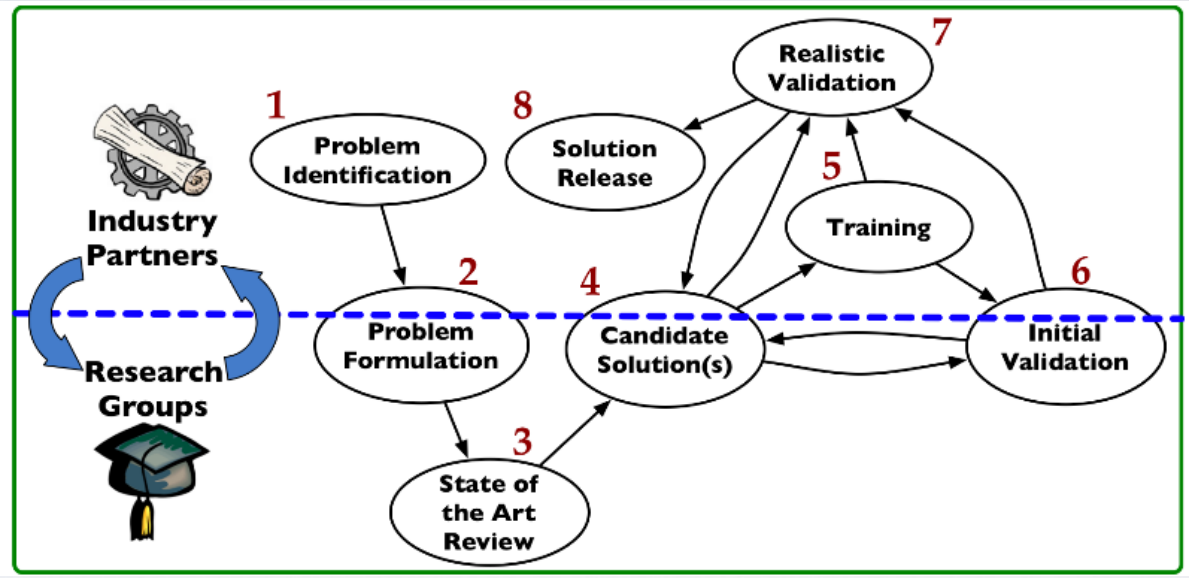
Automated Testing of Autonomous Driving Assistance Systems

Lionel Briand

SEMLA, Montreal, 2018

Collaborative Research @ SnT



securityandtrust.lu

- Research in context
- Addresses actual needs
- Well-defined problem
- Long-term collaborations
- Our lab is the industry

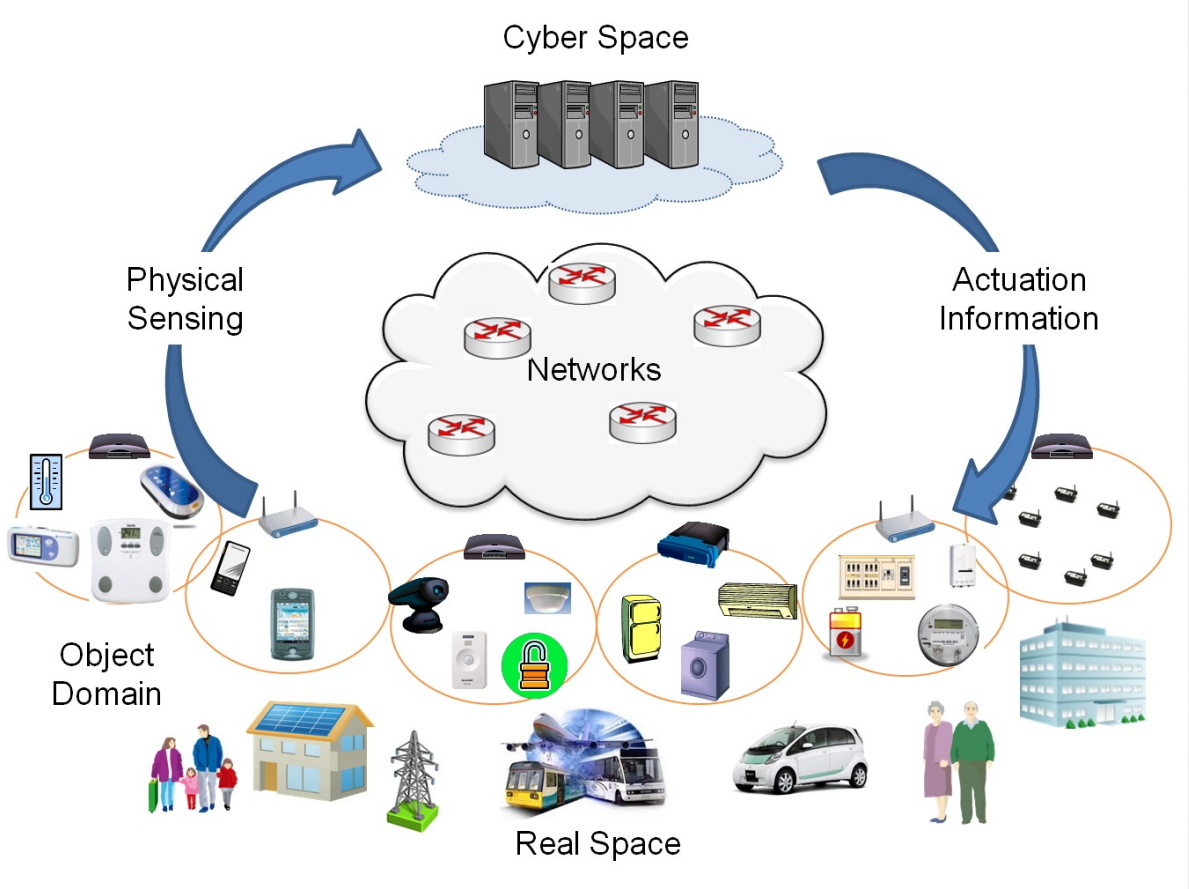
Software Verification and Validation @ SnT Centre

- Group established in **2012**
- **Focus:** Automated, novel, cost-effective V&V solutions
- ERC Advanced Grant
- ~ 25 staff members
- Industry and public partnerships

Introduction

Cyber-Physical Systems

- **A system of collaborating computational elements controlling physical entities**

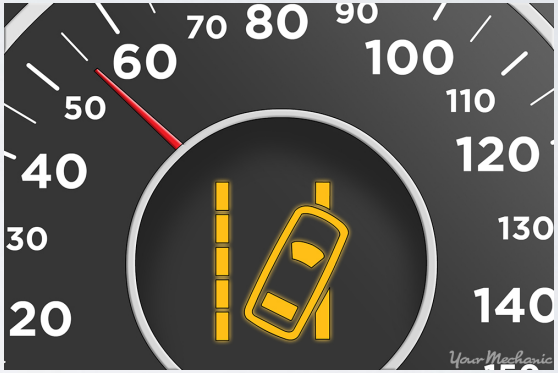


Autonomous CPS

- Read sensors, i.e., collect data about their environment
- Make predictions about their environment
- Make (optimal) decisions about how to behave to achieve some objective(s) based on predictions
- Send commands to actuators according to decisions
- Often mission or safety critical

Advanced Driver Assistance Systems (ADAS)

Automated Emergency Braking (AEB)



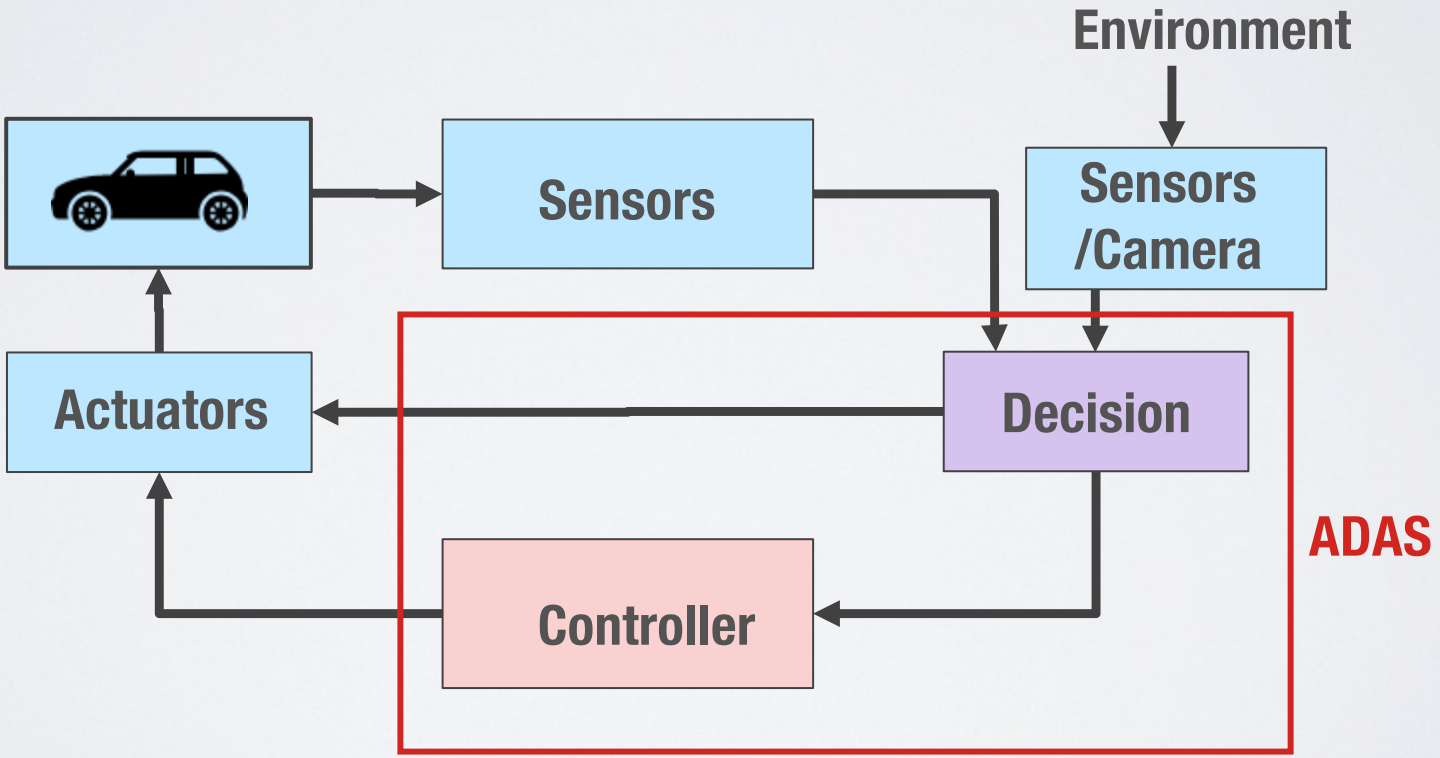
Lane Departure Warning (LDW)

Pedestrian Protection (PP)

Traffic Sign Recognition (TSR)

Advanced Driver Assistance Systems (ADAS)

Decisions are made over time based on sensor data



A General and Fundamental Shift

- Increasingly so, it is easier to learn behavior from data using machine learning, rather than specify and code
- Deep learning, reinforcement learning …
- Example: Neural networks (deep learning)
- Millions of weights learned
- No explicit code, no specifications
- Verification, testing?

Testing Implications

- **Test oracles?** No explicit, expected test behavior
- **Test completeness?** No source code, no specification

CPS Development Process

Model-in-the-Loop Stage

Functional modeling:
- Controllers
- Plant
- Decision

Continuous and discrete Simulink models

Model simulation and testing

MATLAB SIMULINK

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

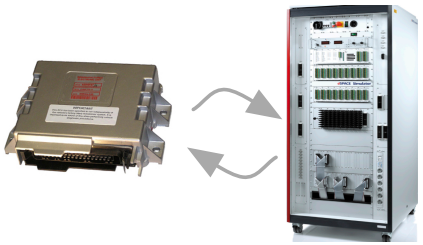
C SysML

Hardware-in-the-Loop Stage

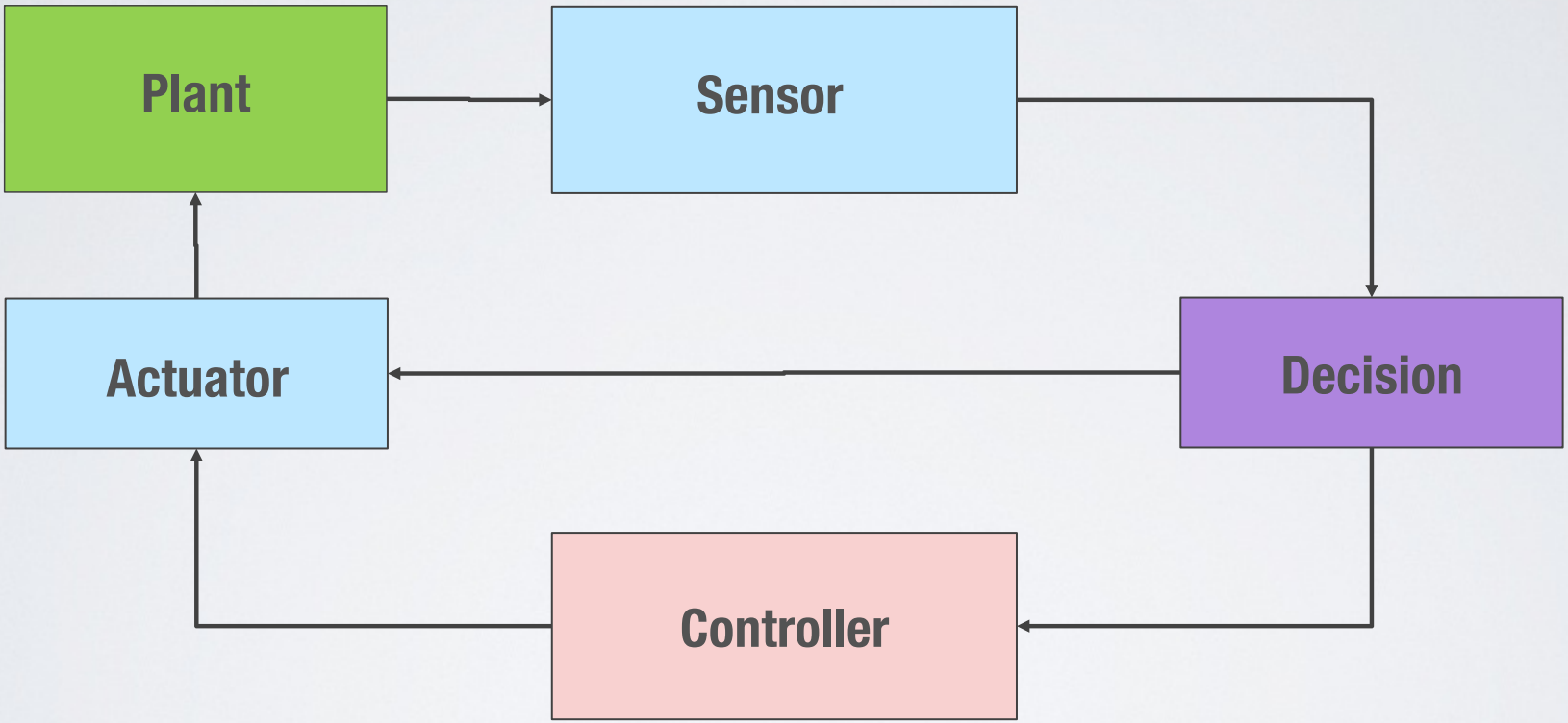
Deployed executables on target platform

Hardware (Sensors ...)
Analog simulators

Testing (expensive)



MiL Components



Opportunities and Challenges

- Early functional models (MiL) offer opportunities for early functional verification and testing
- But a challenge for constraint solvers and model checkers:
  - Continuous mathematical models, e.g., differential equations
  - Discrete software models for code generation, but with complex operations
  - Library functions in binary code

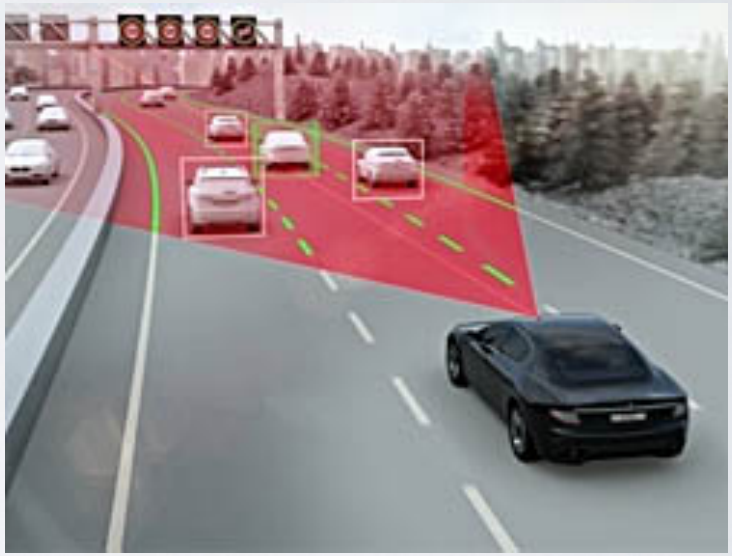
Automotive Environment

- 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
- A challenge for testing

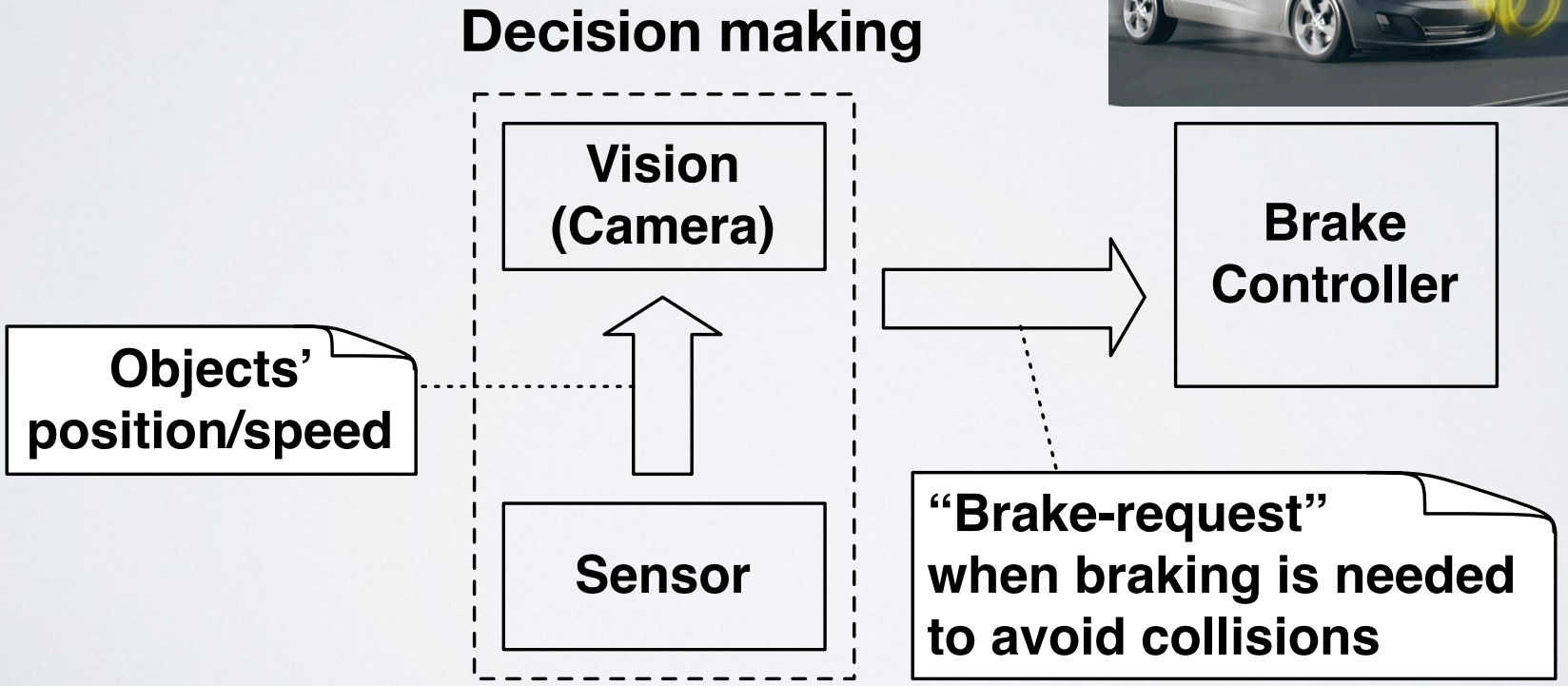
Testing Advanced Driver Assistance Systems

Objective

- Testing ADAS
  - Identify and characterize most critical/risky scenarios
  - Test oracle: Safety properties
  - Need scalable test strategy due to large input space



Automated Emergency Braking System (AEB)



Example Critical Situation

"AEB detects a pedestrian in front of the car with a high degree of certainty, but an accident happens where the car hits the pedestrian with a relatively high speed"

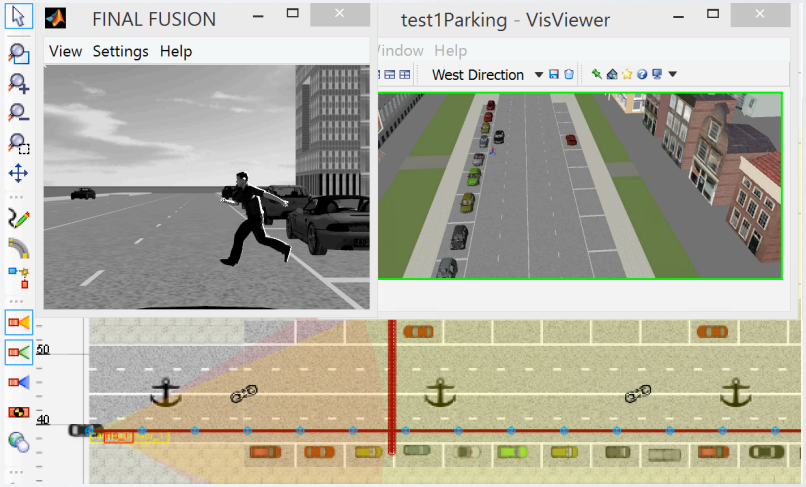
Testing ADAS

On-road testing

Time-consuming

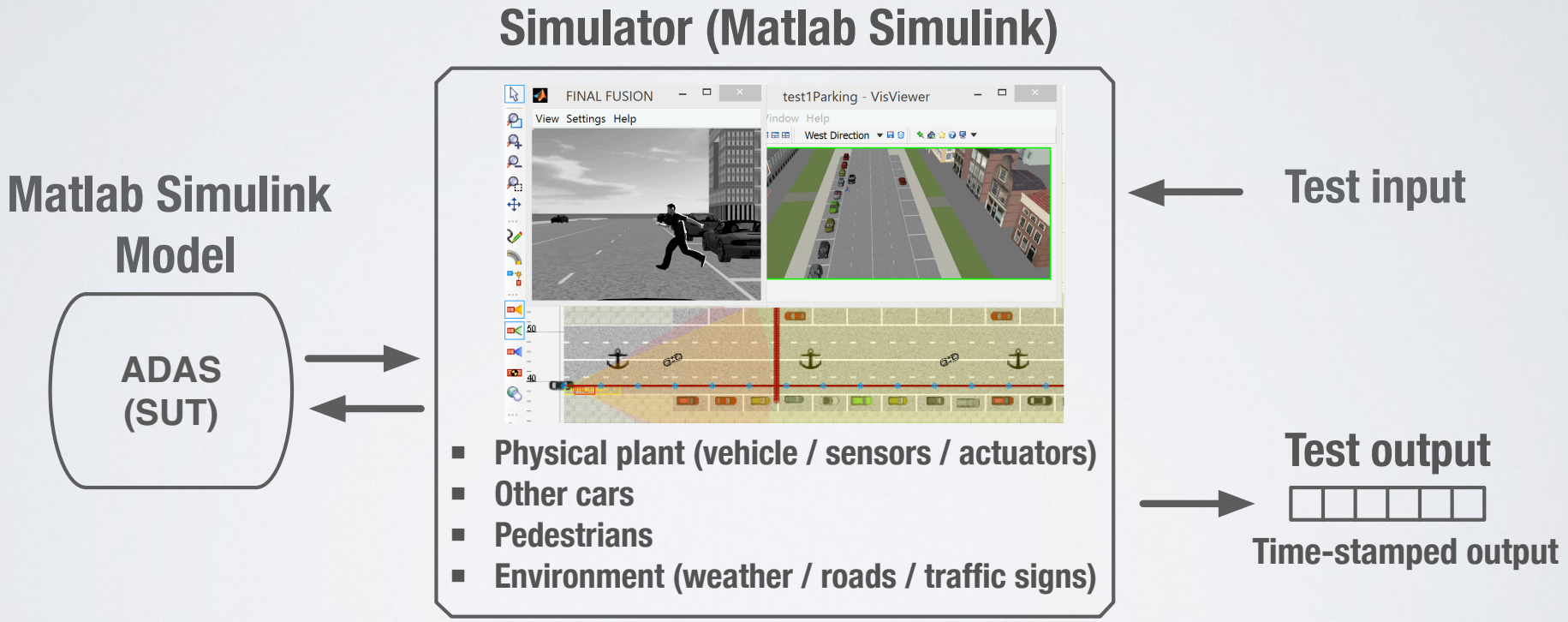
Expensive

Simulation-based (model) testing

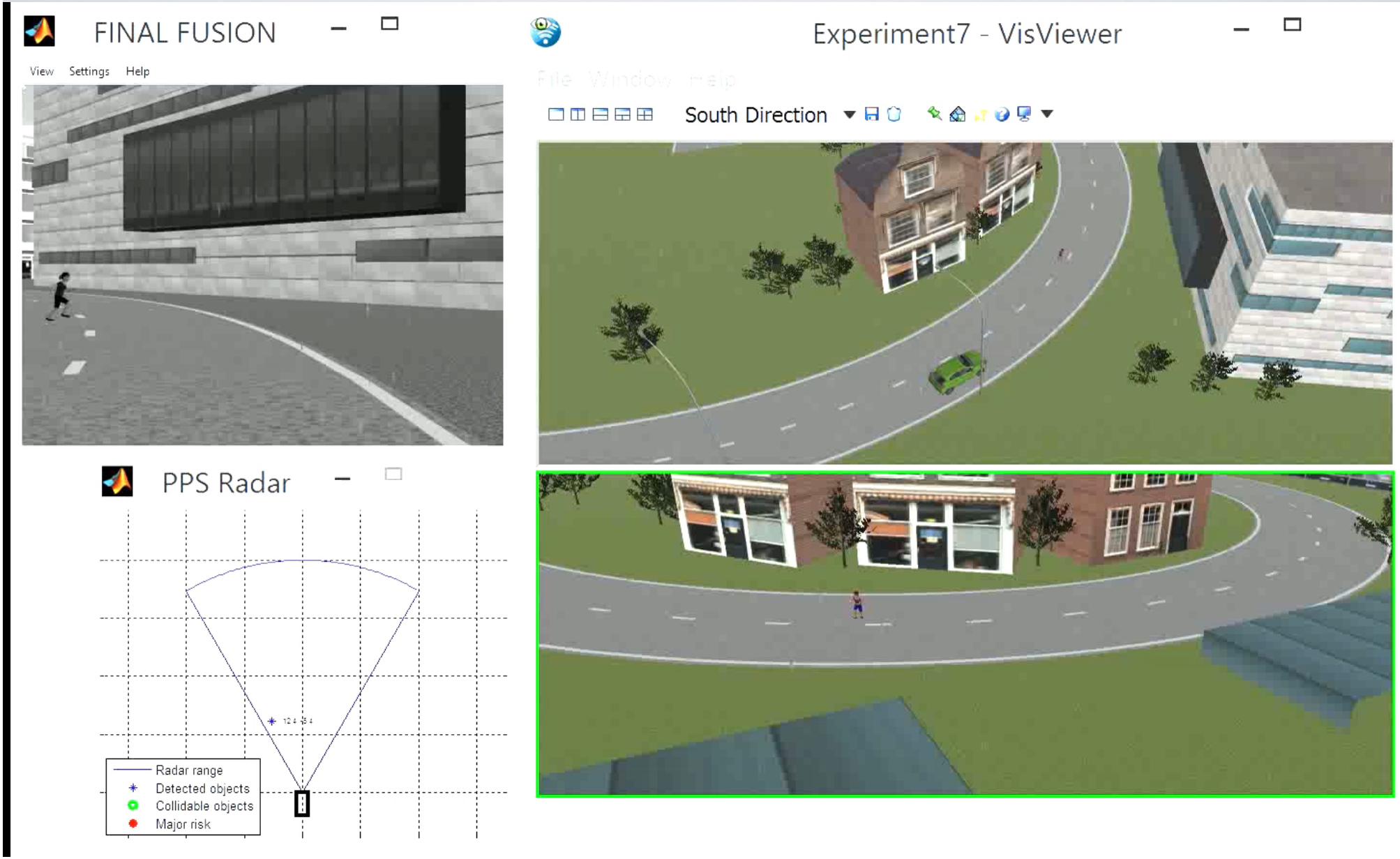


A simulator based on physical/mathematical models

Model Testing ADAS



Physics-Based Simulations



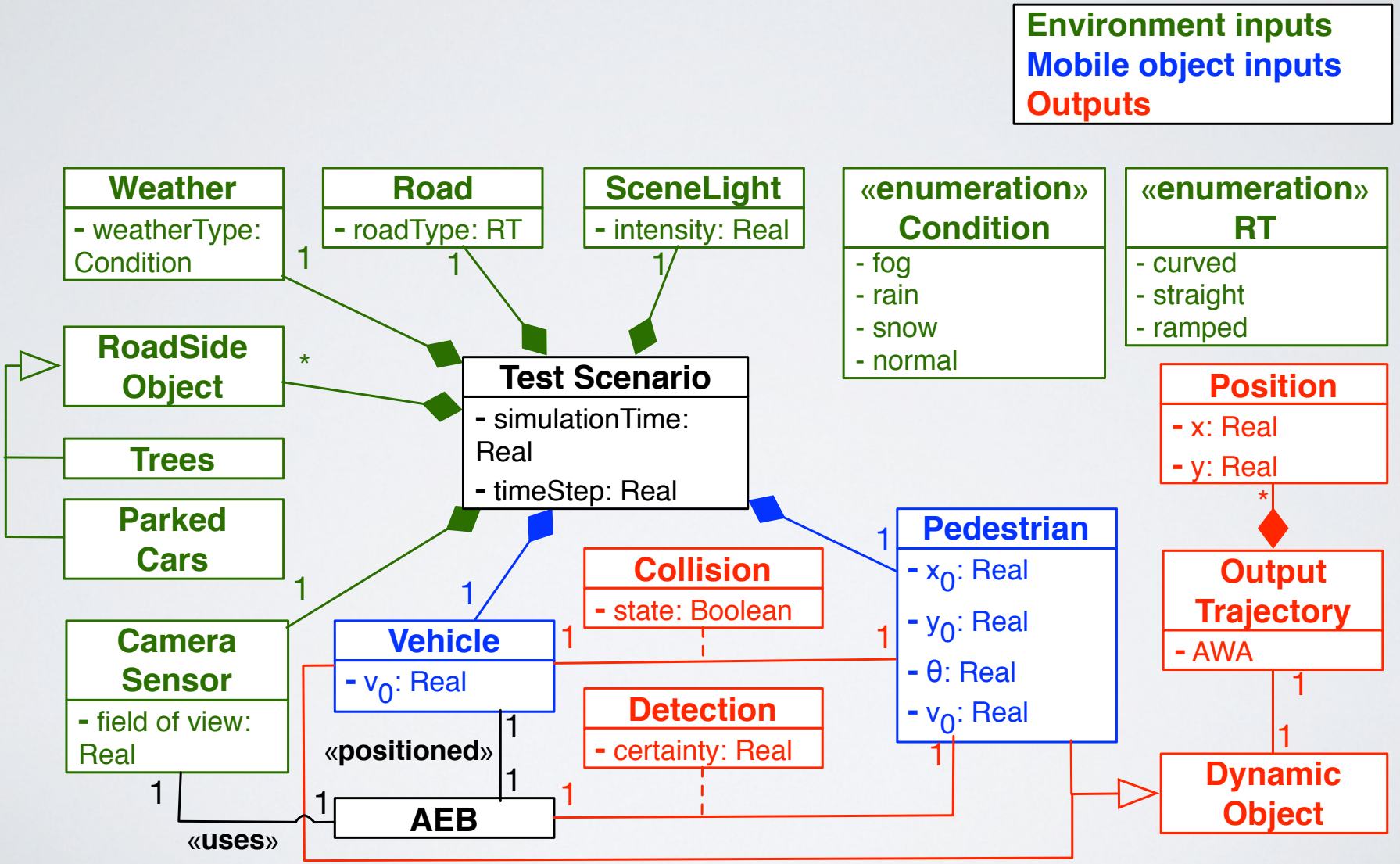
Our Goal

- Developing an automated testing technique for ADAS
  - To help engineers efficiently and effectively explore the complex test input space of ADAS
  - To identify critical (failure-revealing) test scenarios
  - Characterization of input conditions that lead to most critical situations

ADAS Testing Challenges

- Test input space is large, complex and multidimensional
- Explaining failures and fault localization are difficult
- Execution of physics-based simulation models is computationally expensive

Test Inputs/Outputs



Our Solution: Learnable Evolutionary Algorithms

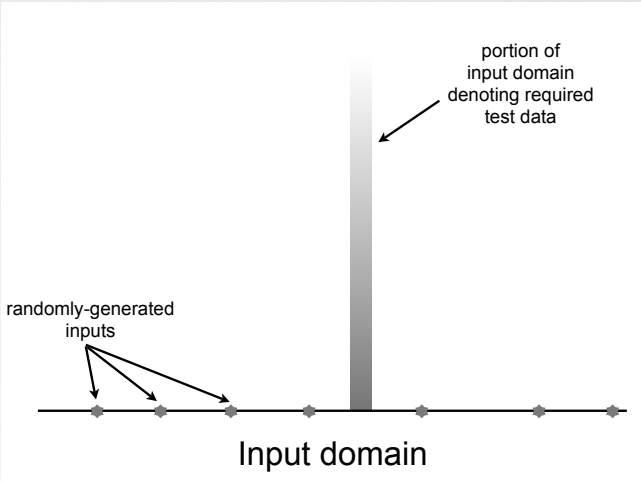
Learn regions likely to contain most critical (failure) test scenarios

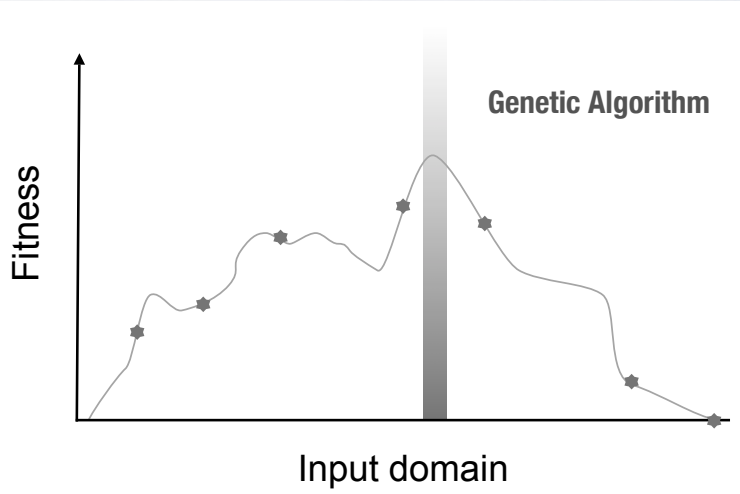
Search for critical test scenarios in the critical regions, and help refine classification models

➔ Machine-learning classification models are used to characterize failures and guide the search towards critical test scenarios faster

Search-Based Software Testing

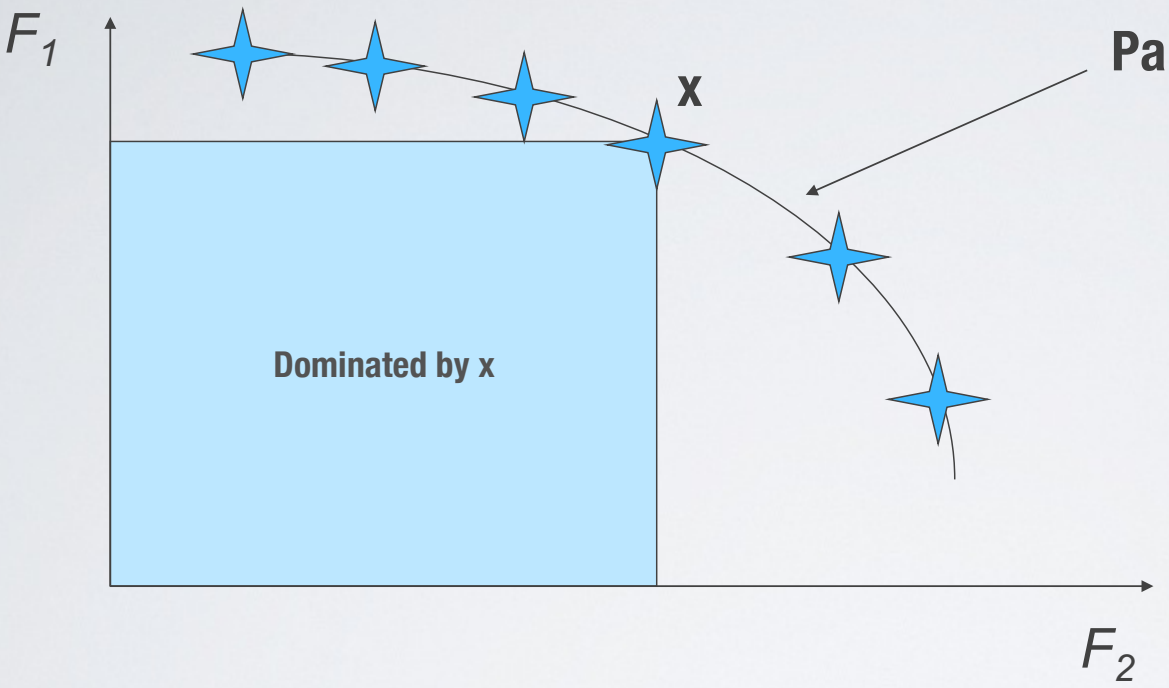
- Express test generation problem as a search problem
- Search for test input data with certain properties, i.e., constraints
- Non-linearity of software (if, loops, …): complex, discontinuous, non-linear search spaces (Baresel)
- Many search algorithms (metaheuristics), from local search to global search, e.g., Hill Climbing, Simulated Annealing and Genetic Algorithms





"Search-Based Software Testing: Past, Present and Future"
Phil McMinn

Multiple Objectives: Pareto Front



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.

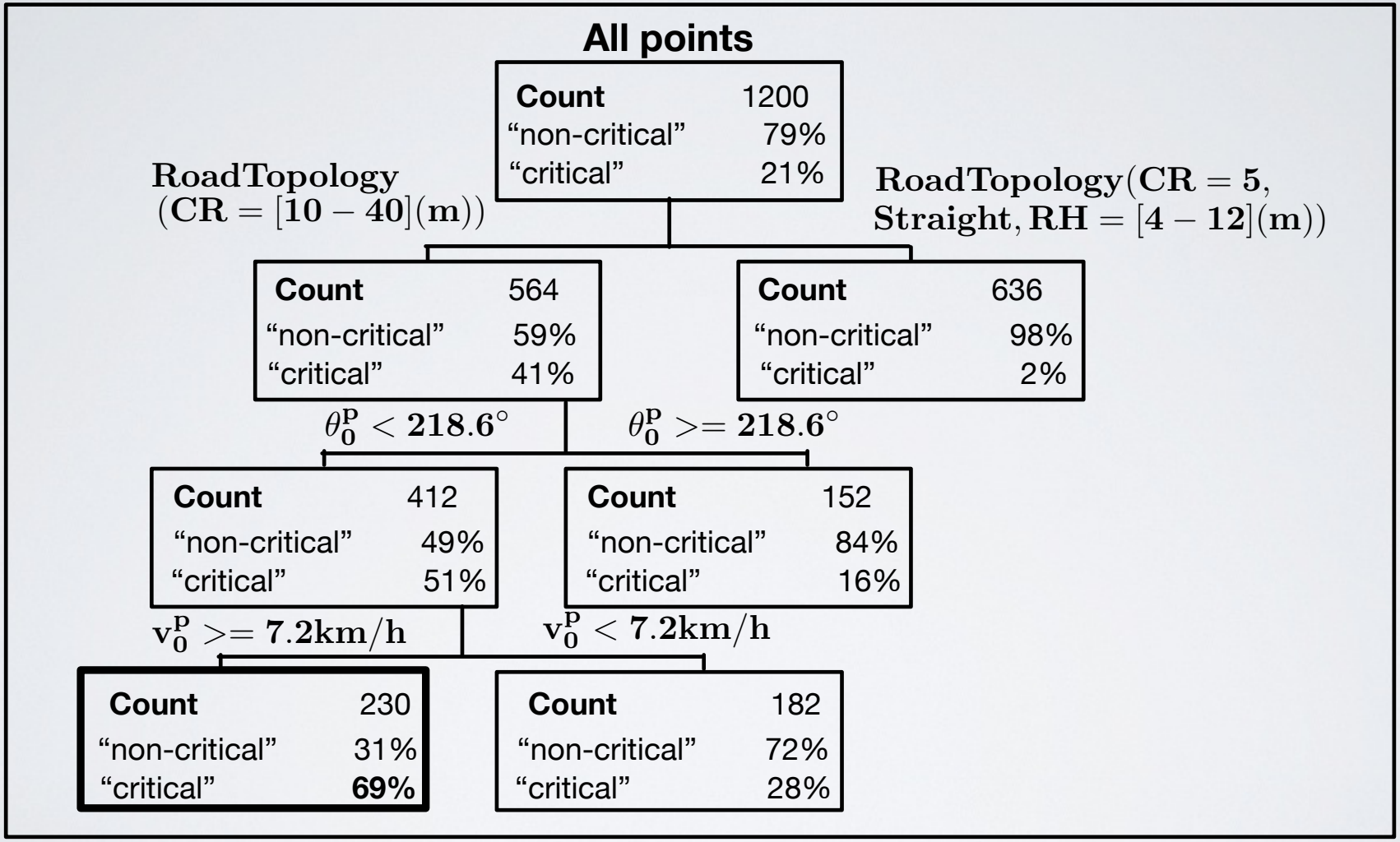
- A multi-objective optimization algorithm (e.g., NSGA II) must:
  - Guide the search towards the global Pareto-Optimal front.
  - Maintain solution diversity in the Pareto-Optimal front.

Our ADAS Testing

- 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
- Input values required to perform the simulation:

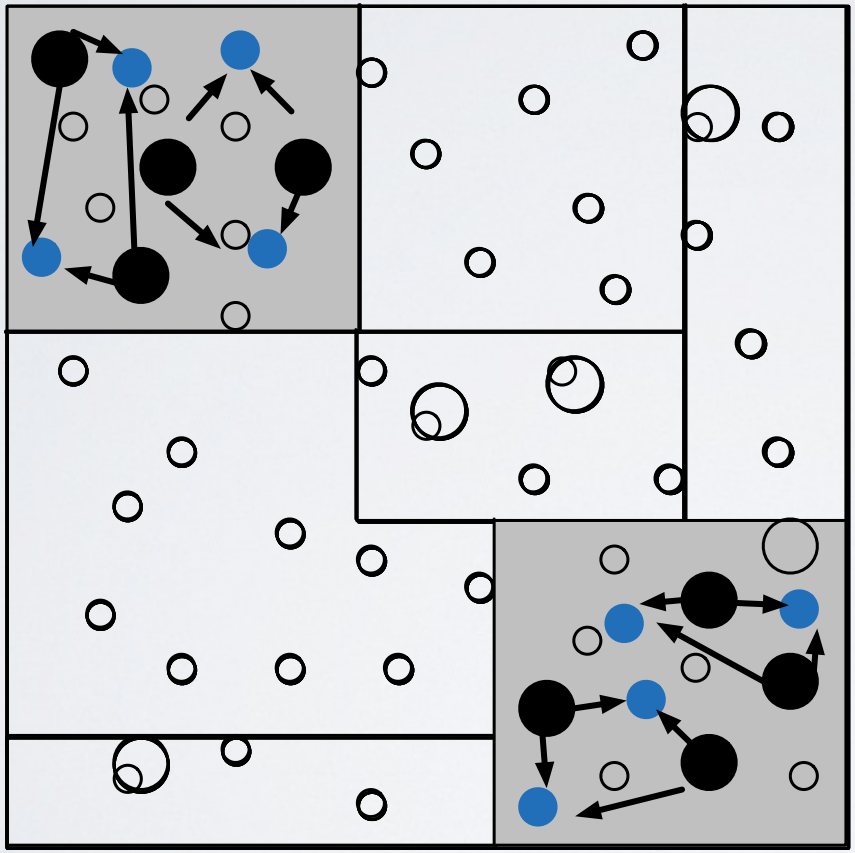
Precipita-tion | Fogginess | Road shape | Visibility range | Car-speed | Person-speed | Person-position | Person-orientation

Decision Trees



Partition the input space into homogeneous regions

Genetic Evolution guided by Classification



Initial input ✓

Fitness computation ✓

Classification ✓

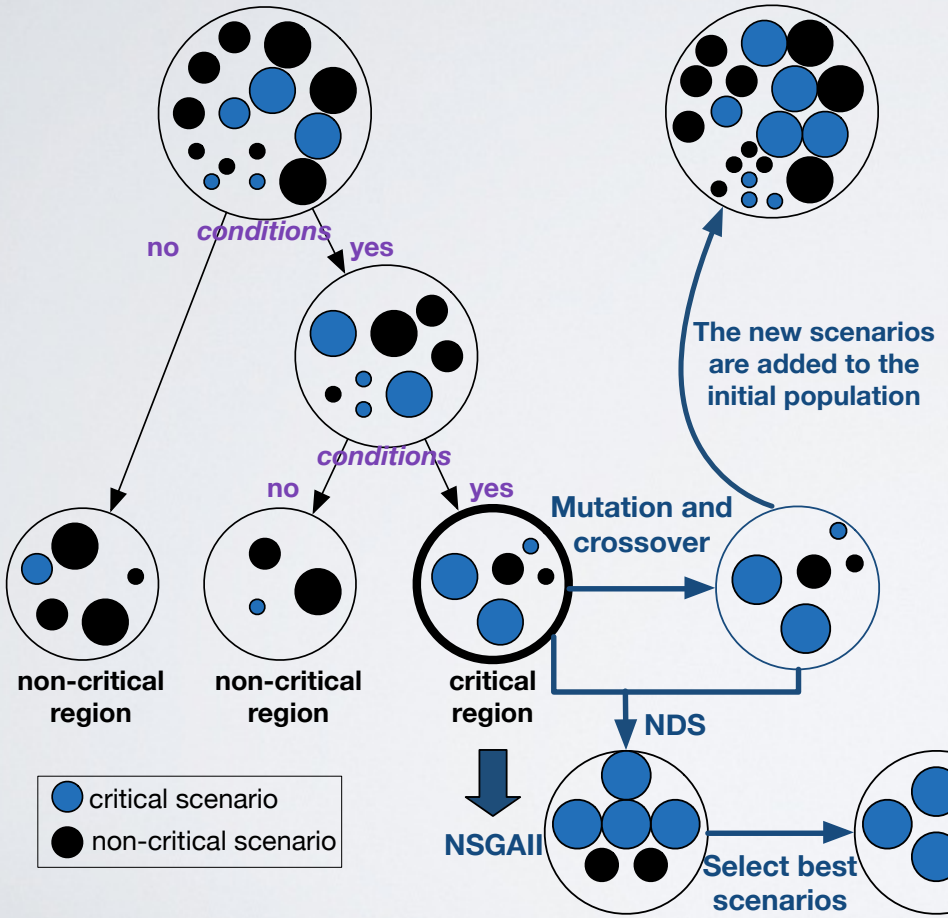
Selection ✓

Breeding

NSGAII-DT

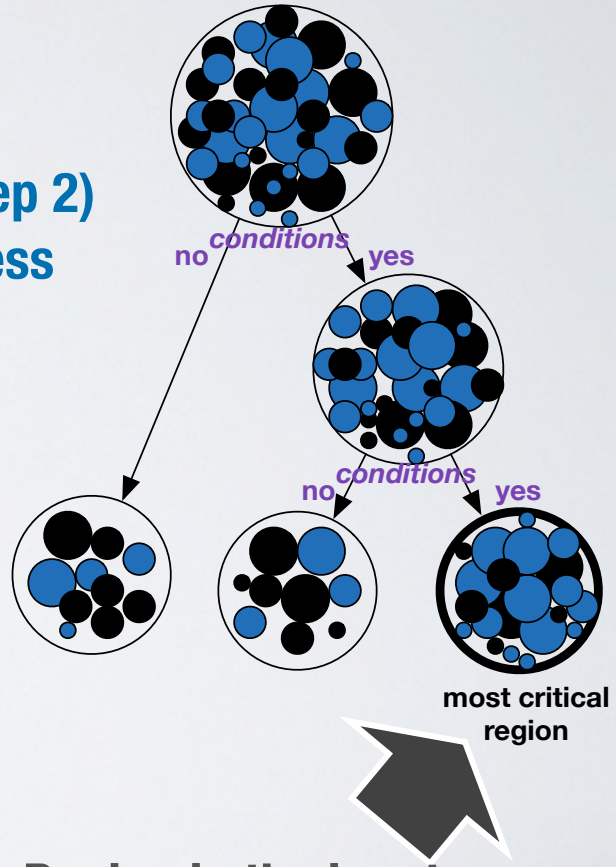
1. Generate an initial representative set of input scenarios and run the simulator to label each scenario as critical or non-critical

2. Build a decision tree model



3. Run the NSGAII search algorithm for the elements inside each critical leaf

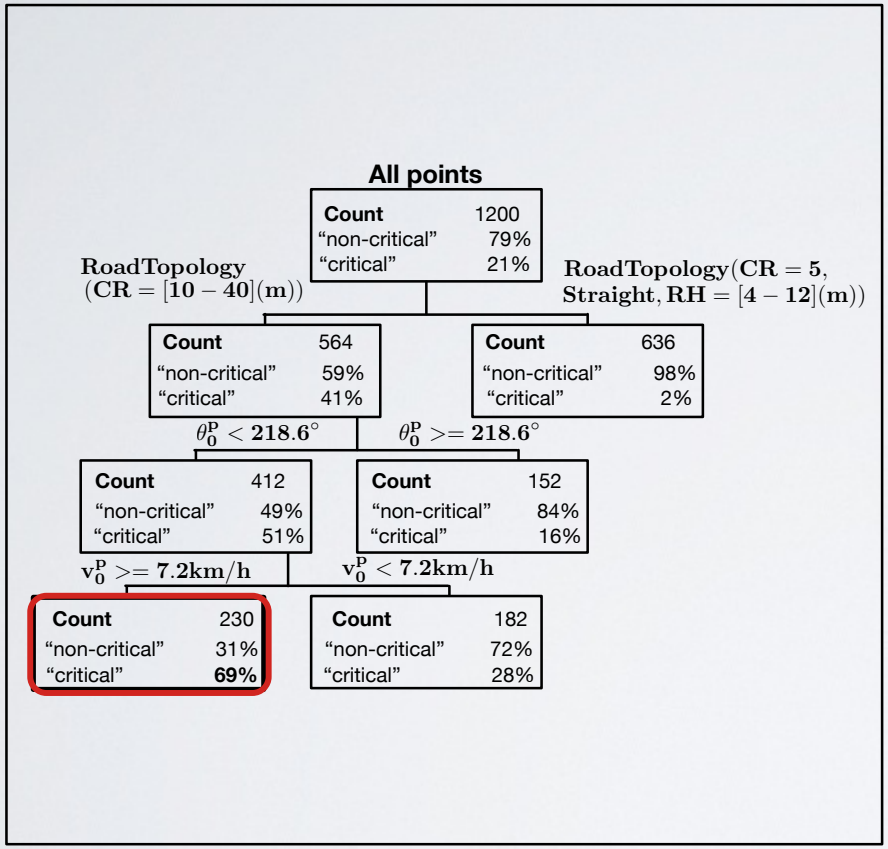
4. Rebuild the decision tree (step 2) or stop the process



Region in the input space that is likely to contain more critical scenarios

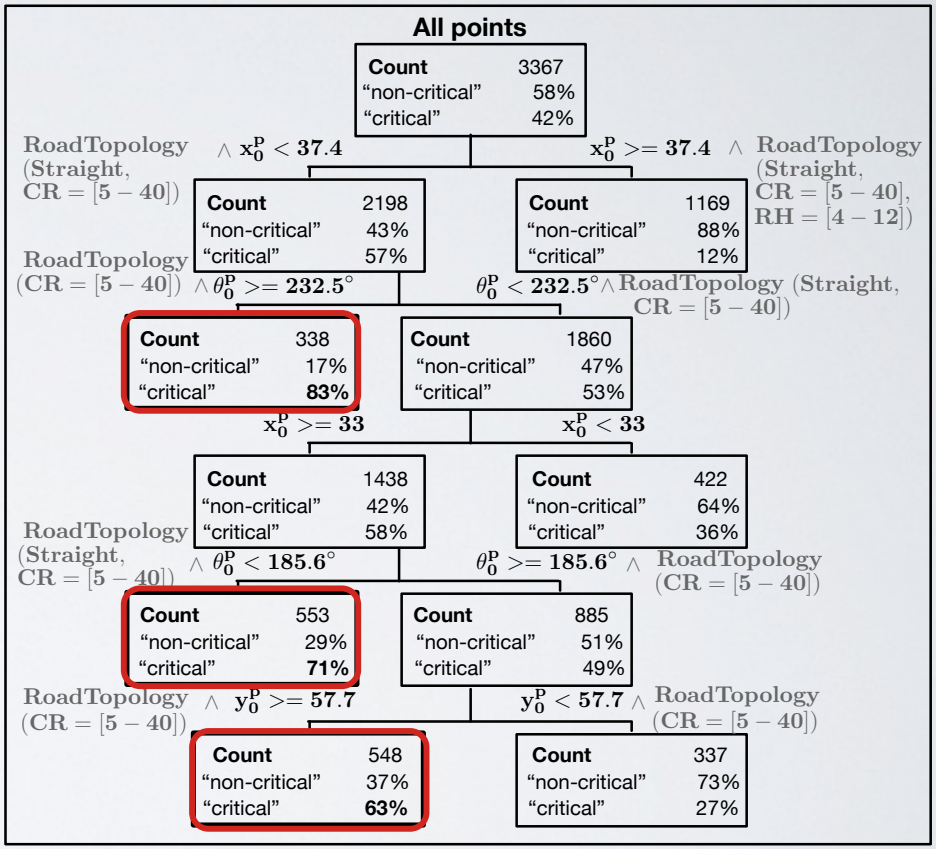
Iterative Process

Initial Classification Model



We focus on generating more scenarios in the critical region, respecting the conditions that lead to that region

Refined Classification Model

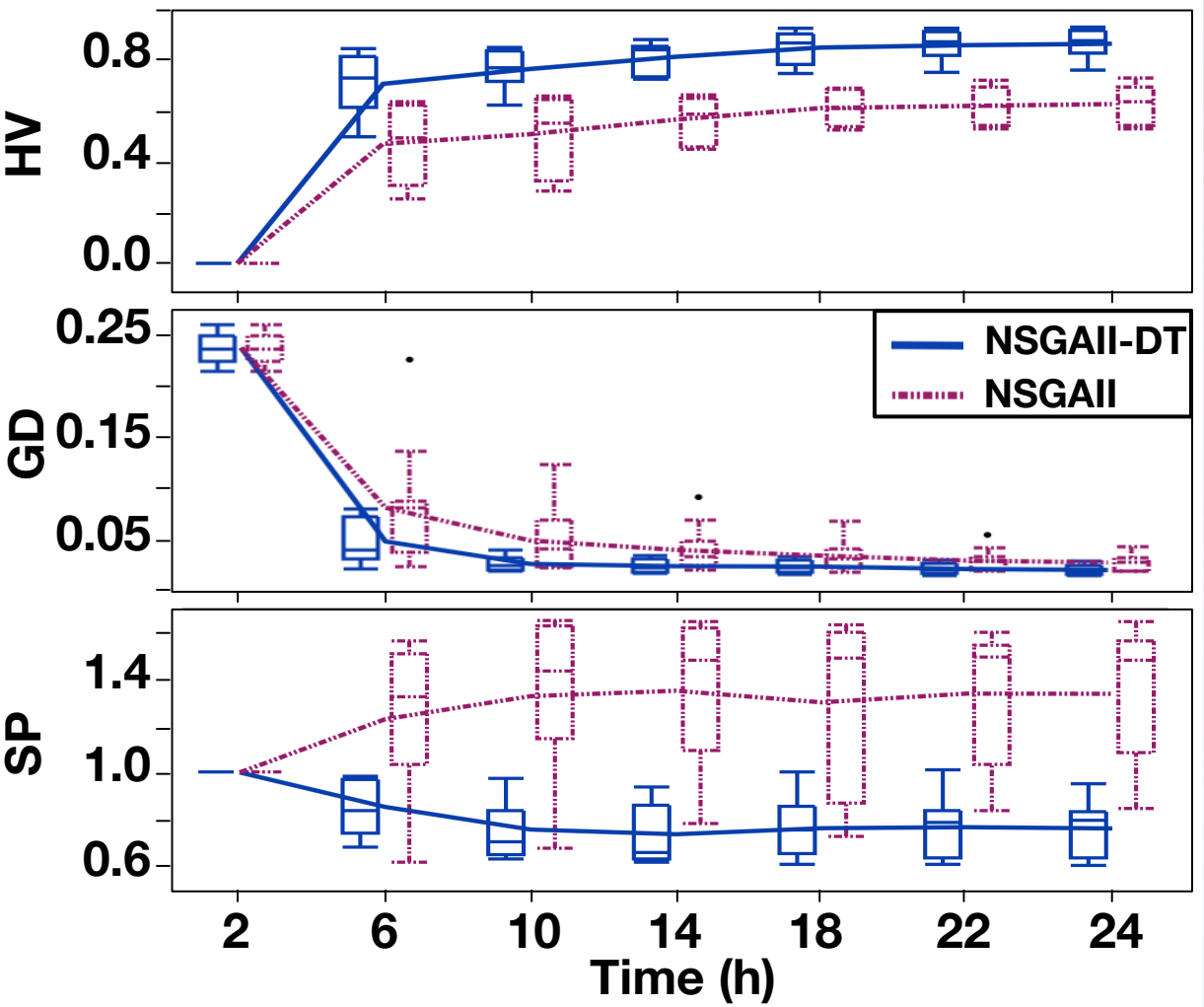


We get a more refined decision tree with more critical regions and more homogeneous areas

Research Questions

- RQ1: Does the decision tree technique help guide the evolutionary search and make it more effective?
- RQ2: Does our approach help characterize and converge towards homogeneous critical regions?
- Failure explanation
- Usefulness (feedback from engineers)

RQ1: NSGAII-DT vs. NSGAII

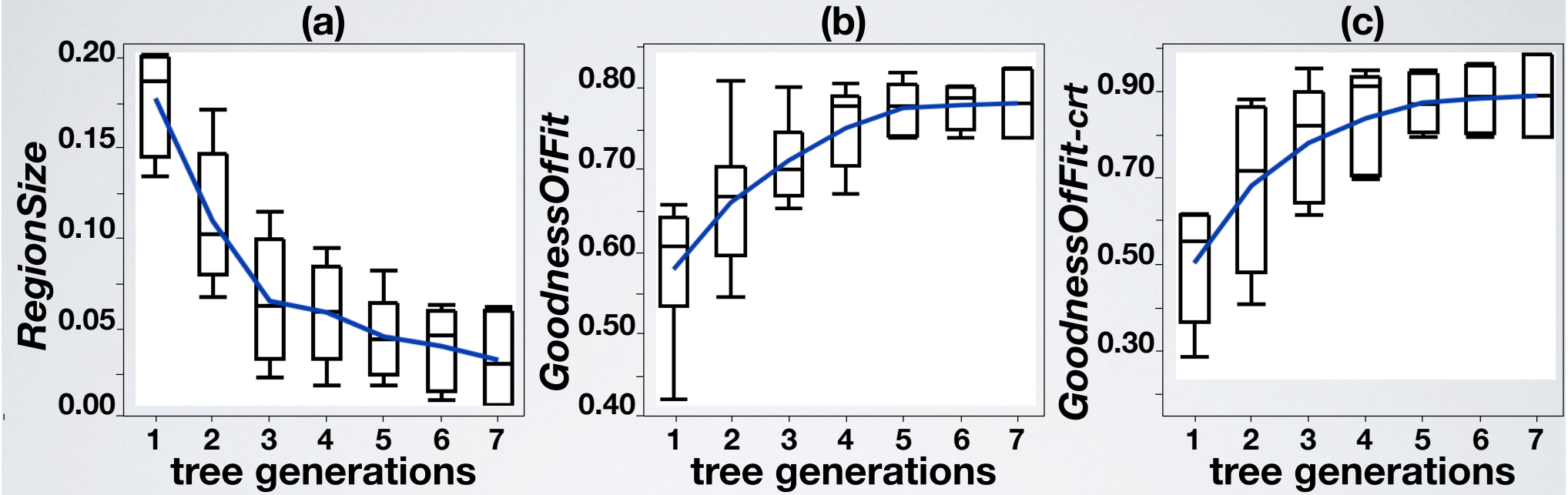


NSGAII-DT outperforms NSGAII

RQ1: NSGAII-DT vs. NSGAII

- NSGAII-DT generates 78% more distinct, critical test scenarios compared to NSGAII

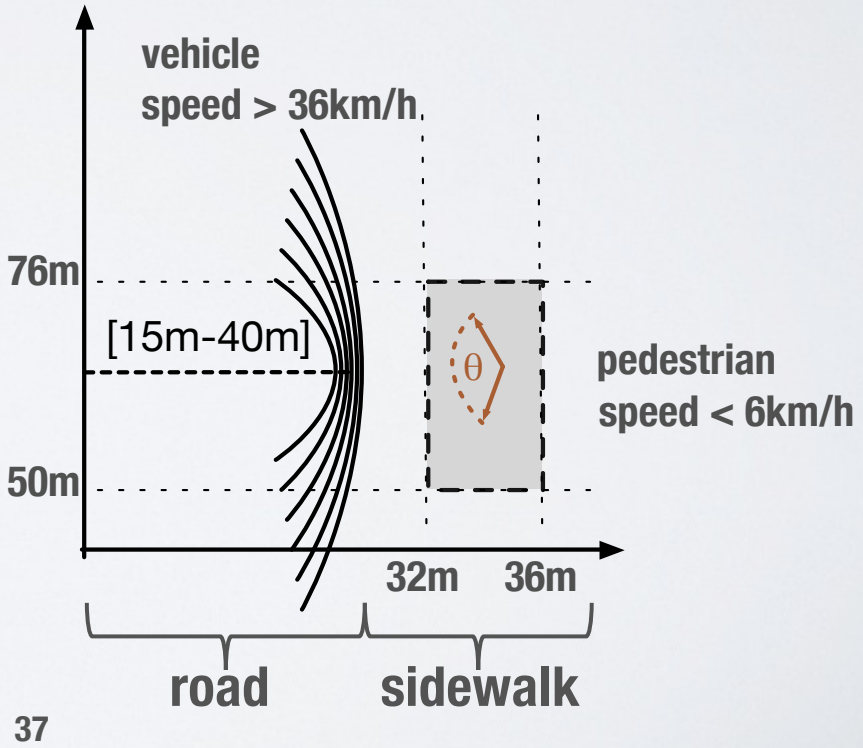
RQ2: NSGAII-DT (evaluation of the generated decision trees)



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

Failure explanation

- A characterization of the input space showing **under what input conditions the system is likely to fail**
- Visualized by decision trees or dedicated diagrams
- **Path conditions in trees**



Usefulness

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

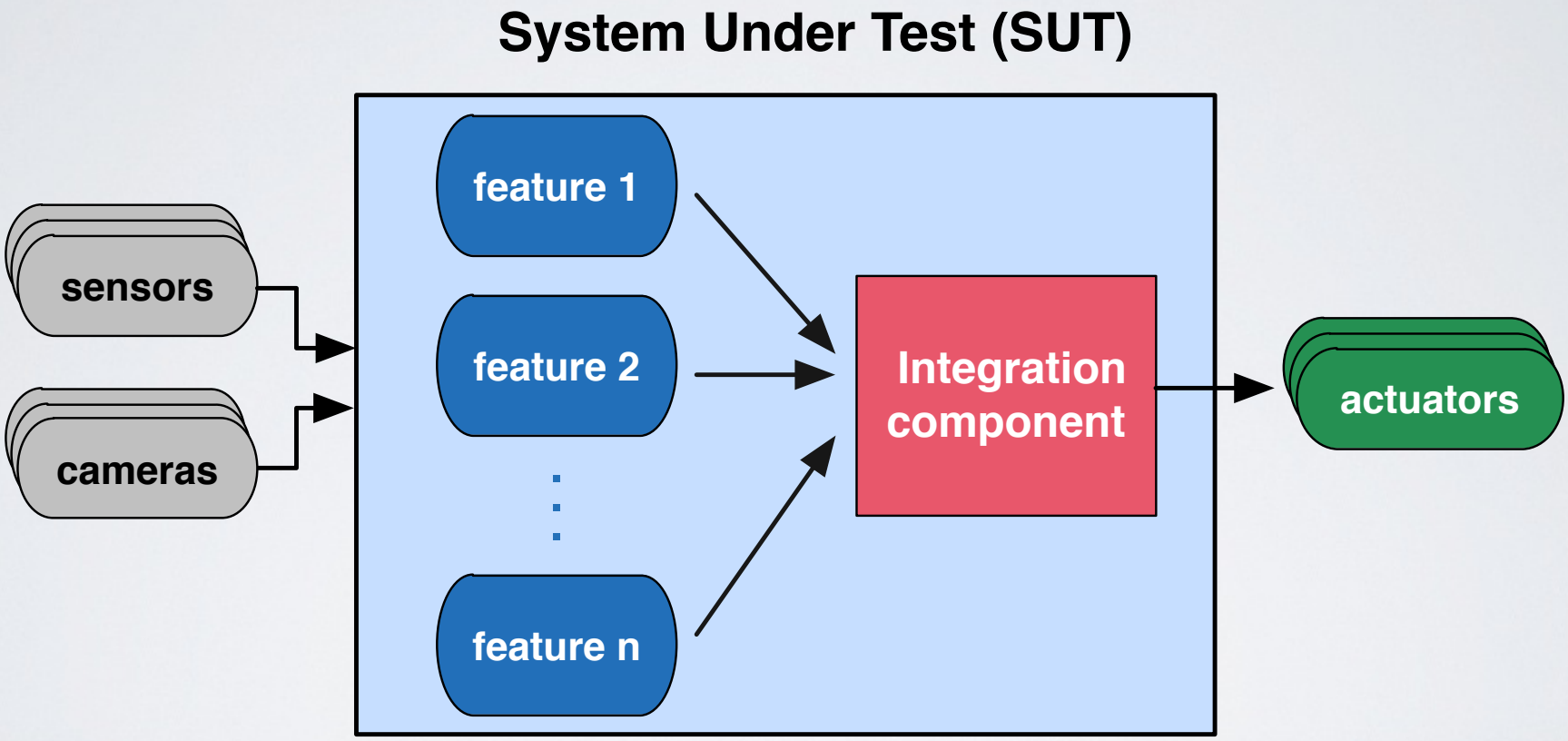
  (1) Debugging the system model (or the simulator)

  (2) Identifying possible hardware changes to increase ADAS safety

  (3) Providing proper warnings to drivers

Automated Testing of Feature Interactions Using Many Objective Search

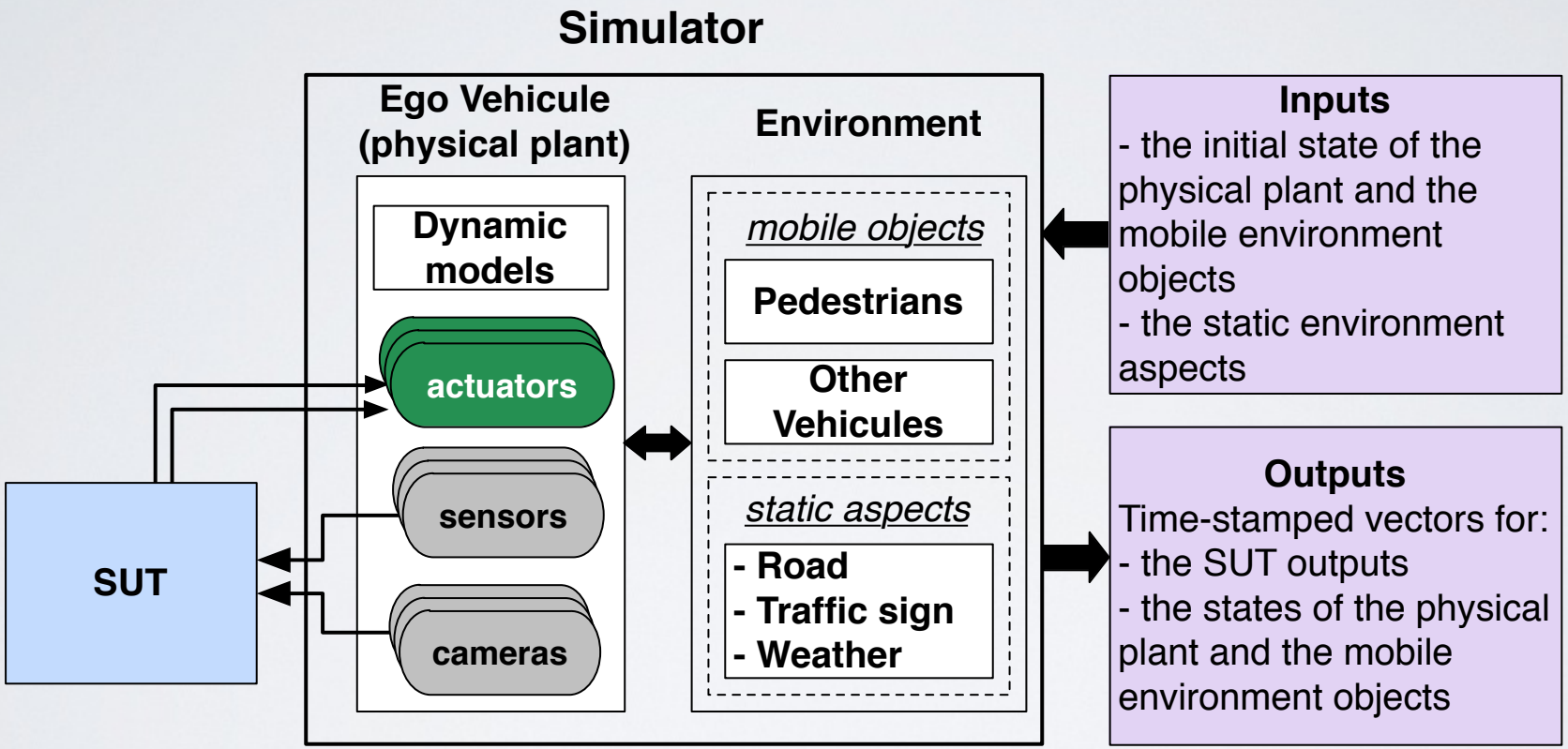
System Integration



Case Study: SafeDrive

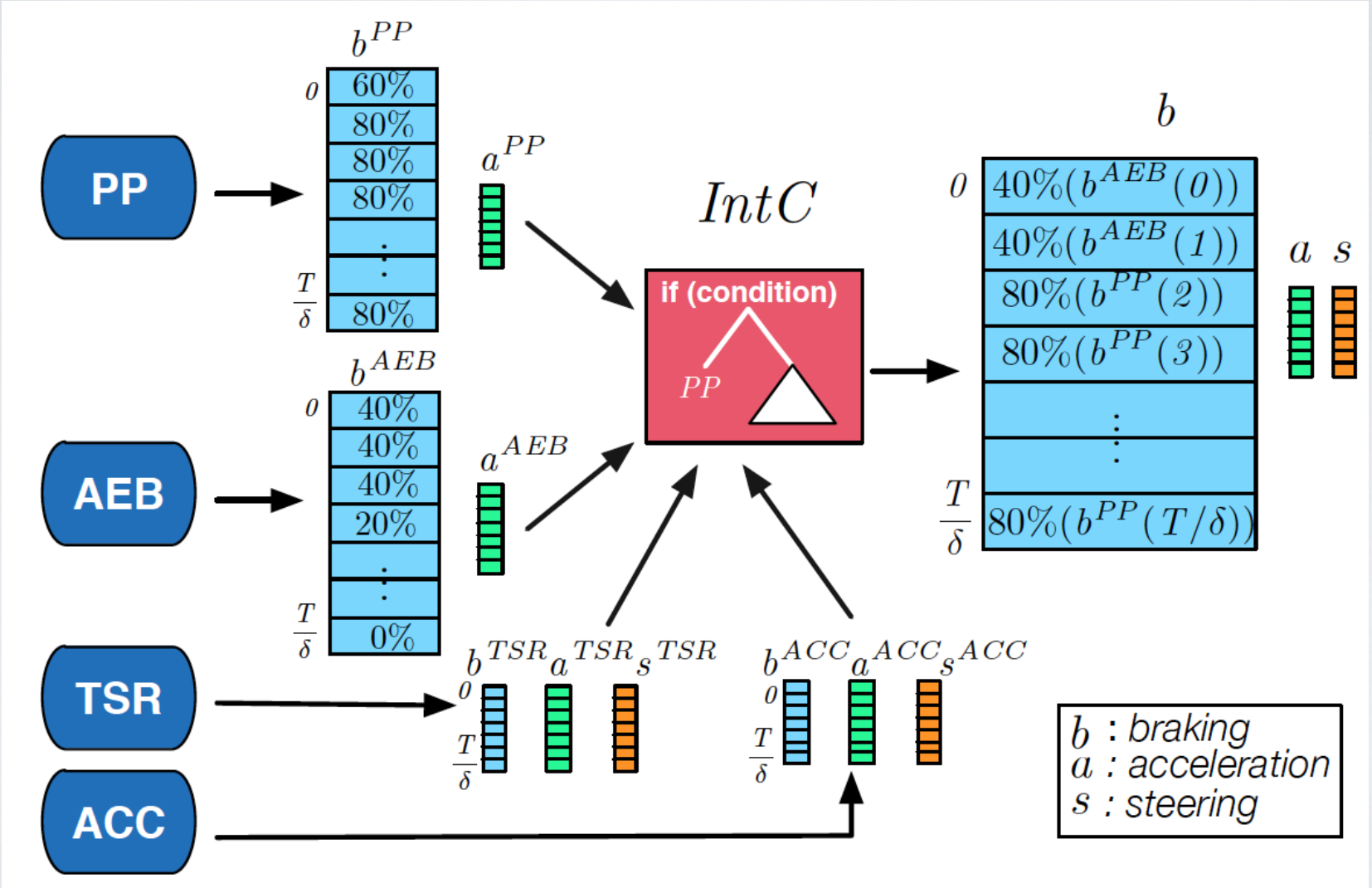
- Our case study describes an automotive system consisting of four advanced driver assistance features:
  - Cruise Control (ACC)
  - Traffic Sign Recognition (TSR)
  - Pedestrian Protection (PP)
  - Automated Emergency Breaking (AEB)

Simulation



Feedback loop

Actuator Command Vectors



Safety Requirements

Feature	Requirement	Failure distance functions $(FD_1, \ldots, FD_5)$
PP	No collision with pedestrians	$FD_1(i)$ is the distance between the ego car and the pedestrian at step i.
AEB	No collision with cars	$FD_2(i)$ is the distance between the ego car and the leading car at step i.
TSR	Stop at a stop sign	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 $u(i) \geq 20km/h$. Otherwise, we define $FD_3(i) = \frac{1}{u(i)}$. If there is no stop sign, we have $FD_3(i) = 1$.
TSR	Respect the speed limit	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 $u(i) \geq 20km/h$. Otherwise, we define $FD_4(i) = \frac{1}{u'(i)}$. If there is no speed limit sign, we have $FD_4(i) = 1$
ACC	Respect the safety distance	$FD_5(i)$ is the absolute difference between the safety distance sd and $FD_2(i)$.

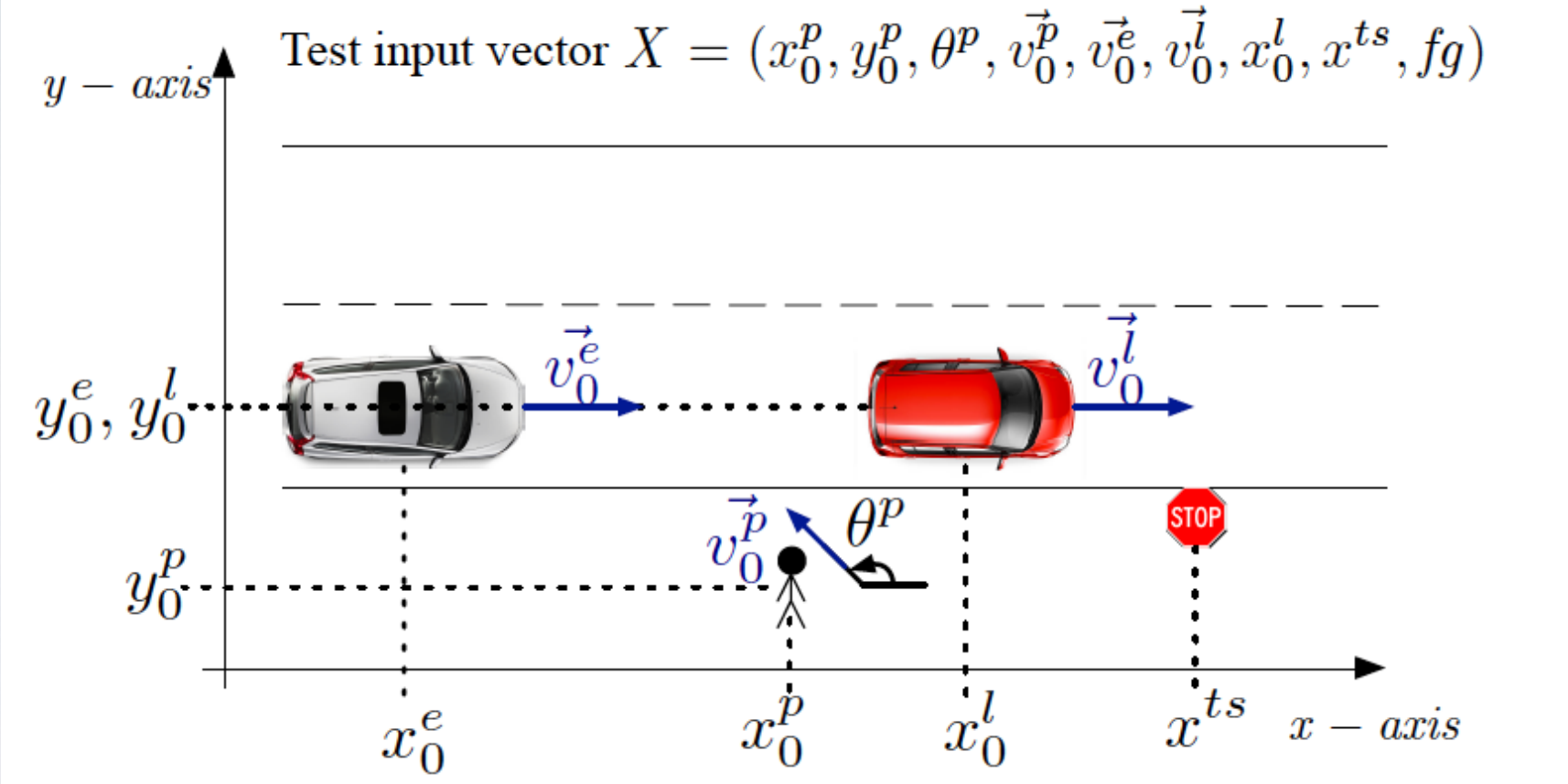
Features

- Behavior of features based on machine learning algorithms processing sensor and camera data
- Interactions between features may lead to violating safety requirements, even if features are correct
- E.g., ACC is controlling the car by ordering it to accelerate since the leading car is far away, while a pedestrian starts crossing the road. PP starts sending braking commands to avoid hitting the pedestrian.
- Complex: predict and analyze possible interactions at the requirements level in a complex environment
- Resolution strategies cannot always be determined statically and may depend on environment

Objective

- **Automated and scalable testing** to help ensure that resolution strategies are safe
- Detect **undesired feature interactions**
- **Assumptions:** IntC is white-box (integrator is testing), features were previously tested

Input Variables



Search

- **Input space is very large**
- **Dedicated search algorithm** (many objectives) directed/guided by test objectives (fitness functions)
- **Fitness (distance) functions:** reward test cases that are more likely to reveal integration failures leading to safety violations
- Combine **three types of functions**: (1) safety violations, (2) unsafe overriding by integration component (IntC), (3) coverage of the decision structure of IntC
- **Many test objectives** to be satisfied by the test suite

Failure Distance

- Goal: Reveal **safety requirements violations**
- **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

Feature	Requirement	Failure distance functions ($FD_1, \ldots, FD_5$)
PP	No collision with pedestrians	$FD_1(i)$ is the distance between the ego car and the pedestrian at step i.
AEB	No collision with cars	$FD_2(i)$ is the distance between the ego car and the leading car at step i.
TSR	Stop at a stop sign	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 $u(i) \geq 20km/h$. Otherwise, we define $FD_3(i) = \frac{1}{u(i)}$. If there is no stop sign, we have $FD_3(i) = 1$.
TSR	Respect the speed limit	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
ACC		stance

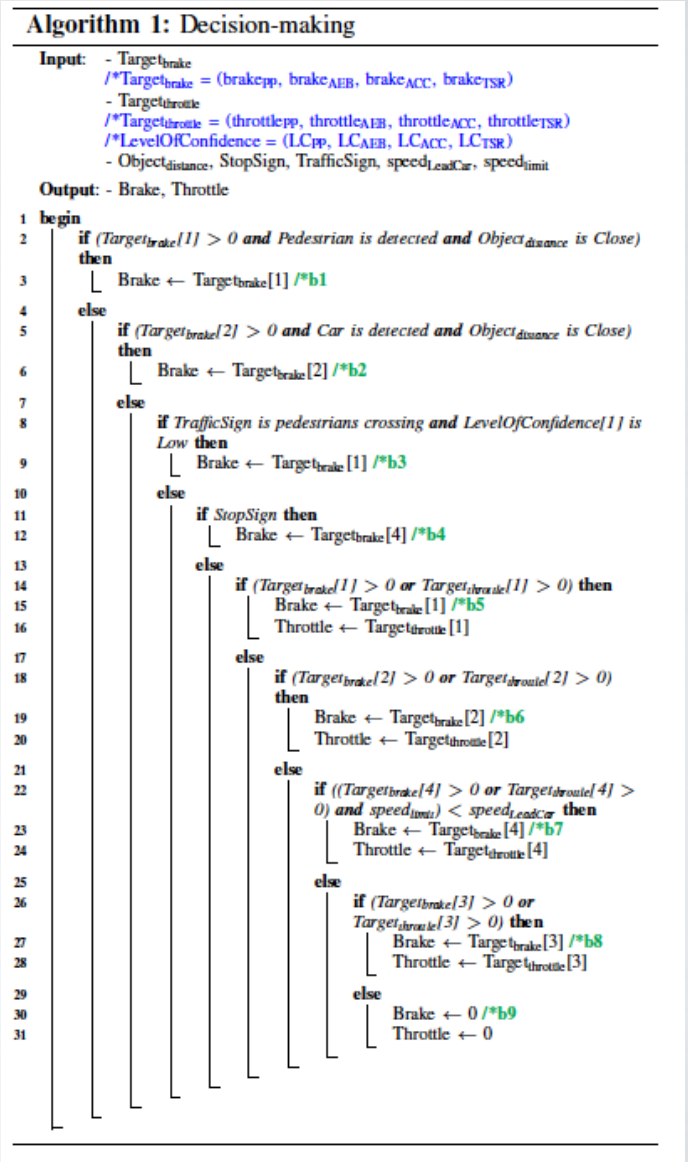
When any of the functions yields zero, a safety failure corresponding to that function is detected.

Unsafe Overriding Distance

- **Goal:** Find faults faults in integration component
- Reward test cases generating integration outputs **deviating from the individual feature outputs**, in such a way as to possibly lead to safety violations.
- **Example:** A feature *f* issues a braking command while the integration component issues no braking command or a braking command with a lower force than that of *f* .

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*



Algorithm 1: Decision-making

Input: - $Target_{brake}$
/*$Target_{brake}$ = ($brake_{PP}$, $brake_{AEB}$, $brake_{ACC}$, $brake_{TSR}$)
- $Target_{throttle}$
/*$Target_{throttle}$ = ($throttle_{PP}$, $throttle_{AEB}$, $throttle_{ACC}$, $throttle_{TSR}$)
/*LevelOfConfidence = (LC_{PP}, LC_{AEB}, LC_{ACC}, LC_{TSR})
- $Object_{distance}$, StopSign, TrafficSign, $speed_{LeadCar}$, $speed_{limit}$

Output: - Brake, Throttle

```
1  begin
2      if (Target_brake[1] > 0 and Pedestrian is detected and Object_distance is Close) then
3          Brake ← Target_brake[1] /*b1
4      else
5          if (Target_brake[2] > 0 and Car is detected and Object_distance is Close) then
6              Brake ← Target_brake[2] /*b2
7          else
8              if TrafficSign is pedestrians crossing and LevelOfConfidence[1] is Low then
9                  Brake ← Target_brake[1] /*b3
10             else
11                 if StopSign then
12                     Brake ← Target_brake[4] /*b4
13                 else
14                     if (Target_brake[1] > 0 or Target_throttle[1] > 0) then
15                         Brake ← Target_brake[1] /*b5
16                         Throttle ← Target_throttle[1]
17                     else
18                         if (Target_brake[2] > 0 or Target_throttle[2] > 0) then
19                             Brake ← Target_brake[2] /*b6
20                             Throttle ← Target_throttle[2]
21                         else
22                             if ((Target_brake[4] > 0 or Target_throttle[4] > 0) and speed_limit) < speed_LeadCar then
23                                 Brake ← Target_brake[4] /*b7
24                                 Throttle ← Target_throttle[4]
25                             else
26                                 if (Target_brake[3] > 0 or Target_throttle[3] > 0) then
27                                     Brake ← Target_brake[3] /*b8
28                                     Throttle ← Target_throttle[3]
29                                 else
30                                     Brake ← 0 /*b9
31                                     Throttle ← 0
```

Combining Distance Functions

- **Goal: Execute every branch of IntC such that while executing that branch, IntC unsafely overrides every feature *f* and its outputs violate every safety requirement related to *f*.**

$$\Omega_{j,l}(i) = \begin{cases} \overline{BD}_j(i) + Max(\overline{UOD}) + Max(\overline{FD}) & \text{(1) If } j \text{ is not covered } (\overline{BD}_j(i) > 0) \\ \overline{UOD}_f(i) + Max(\overline{FD}) & \text{(2) If } j \text{ is covered, but } f \text{ is not unsafely overridden } (\overline{BD}_j(i) = 0 \wedge \overline{UOD}_f(i) > 0) \\ \overline{FD}_l(i) & \text{(3) Otherwise } (\overline{BD}_j(i) = 0 \wedge \overline{UOD}_f(i) = 0) \end{cases}$$

$$\Omega_{j,l} = Min_{i=0}^{\delta}\Omega_{j,l}(i)$$

$\Omega_{j,l}(tc) > 2$ — Indicates that *tc* has not covered branch *j*

$2 \geq \Omega_{j,l}(tc) > 1$ — Branch covered but did not cause unsafe override of *f*

$1 \geq \Omega_{j,l}(i) > 0$ — Branch covered, unsafe override, but did not violate requirement *l*

Search Algorithm

- Best test suite covers all search objectives, i.e., for all IntC branches and all safety requirements
- Not a Pareto front optimization problem
- Objectives compete with each others for each test case
- Example: cannot have the ego car violating the speed limit after hitting the leading car in one test case
- Tailored, many-objective genetic algorithm
- Must be efficient (test case executions are very expensive)

Search Algorithm

Algorithm 1: FITEST

Input: Ω: Set of objectives
N: Initial population size
Result: A: Archive

1 **begin**
2 $P \longleftarrow$ RANDOM-POPULATION(N)
3 $W \longleftarrow$ CALCULATE-OBJECTIVES(P, Ω)
4 $[\Omega_c, T_c] \longleftarrow$ GET-COVERED-OBJECTIVE(P,W)
5 $A \longleftarrow T_c$
6 $\Omega \longleftarrow \Omega - \Omega_c$
7 **while** *not (stop_condition)* **do**
8 $Q \longleftarrow$ RECOMBINE(P,N)
9 $Q \longleftarrow$ CORRECT-OFFSPRINGS(Q)
10 $W \longleftarrow$ CALCULATE-OBJECTIVES(Q, Ω)
11 $[\Omega_c, T_c] \longleftarrow$ GET-COVERED-OBJECTIVE(P,W)
12 $A \longleftarrow A \cup T_c$
13 $\Omega \longleftarrow \Omega - \Omega_c$
14 $\mathbb{F}_0 \longleftarrow$ ENVIRONMENTAL-SELECTION($P \cup Q, \Omega$)
15 $P \longleftarrow F_0$
16 $N \longleftarrow |F_0|$
17 **return** A

Randomly generated TCs

Compute fitness

Archive covering tests

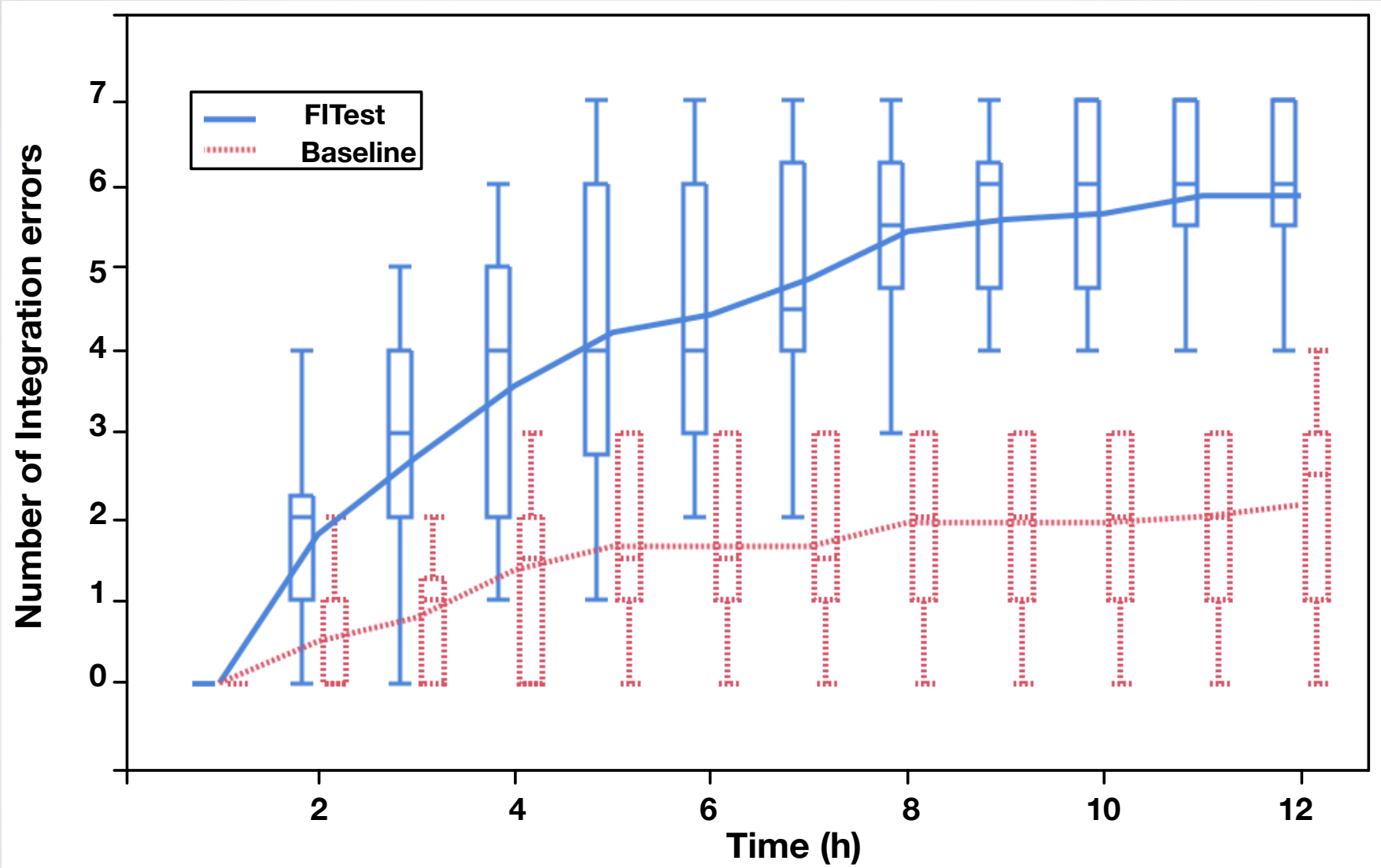
Tests are evolved

Crossover, mutation

Correct constraint violations

Fittest tests selected

Evaluation


FITest
Baseline
Number of Integration errors
0
1
2
3
4
5
6
7
2
4
6
8
10
12
Time (h)

Discussion

Observations

- We are unlikely to have precise and complete requirements, we face great diversity in the physical environment, including many possible scenarios.
- It is possible, however, to define properties characterizing unacceptable situations (safety)
- Notion of test coverage is elusive: No specification or code/models for some key (decision) components based on ML
- We have executable/simulable functional models (e.g., Simulink) at early stages

Conclusions

- We proposed solutions based on:
  - Efficient and realistic (hardware, physics) simulation
  - Metaheuristic search, e.g., evolutionary computing
  - Guided by fitness functions derived from properties of interest (e.g., safety requirements)
  - Machine learning, e.g., to speed up search, provide explanations to engineers
- No guarantees though

Generalizing

- Our work **easily generalizes** to many other cyber-physical systems
- Can a similar strategy be applied in **other domains** to test for bias or any other undesirable properties (e.g., legal), when system behavior is driven by machine learning?
- **Executable models** of environment and users?

Summary

- Machine learning plays an increasingly prominent role in autonomous systems
- No (complete) requirements, specifications, or even code
- Some safety and mission-critical requirements
- Neural networks (deep learning) with millions of weights
- **How do we gain confidence in such software in a scalable and cost-effective way?**

Related Testing Research

- Testing of hybrid controllers
- Testing timeliness requirements
- Testing for deadline misses (schedulability)
- HiL acceptance testing prioritization
- Testing for security vulnerabilities
- Find publications on: svv.lu

Acknowledgements

- Raja Ben Abdessalem
- Shiva Nejati
- Annibale Panichella
- IEE, Luxembourg

References

- R. Ben Abdessalem et al., "Testing Advanced Driver Assistance Systems Using Multi-Objective Search and Neural Networks”, IEEE ASE 2016
- R. Ben Abdessalem et al., "Testing Vision-Based Control Systems Using Learnable Evolutionary Algorithms”, IEEE/ACM ICSE 2018

Automated Testing of Autonomous Driving Assistance Systems

Lionel Briand

SEMLA, Montreal, 2018