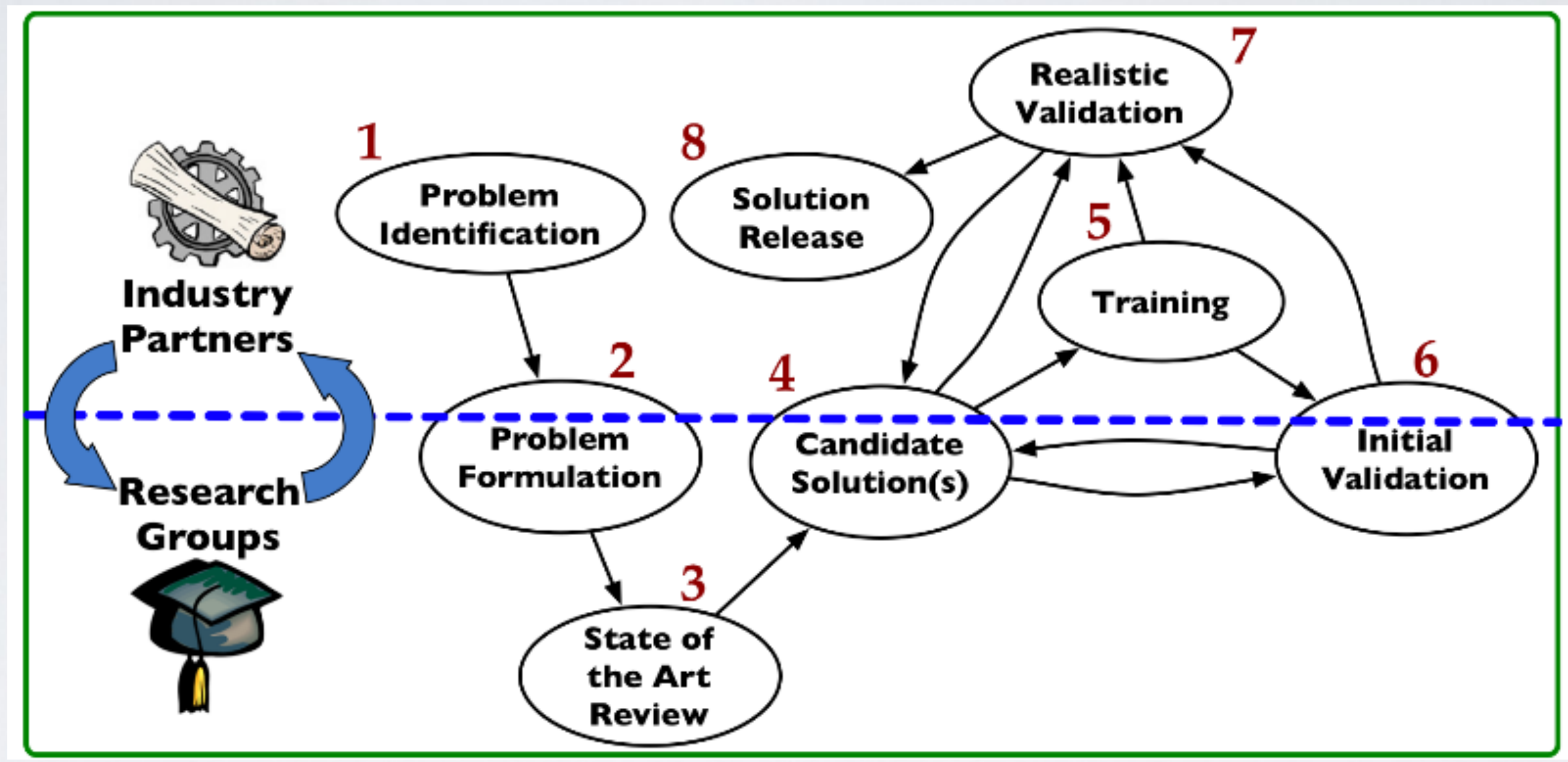
Automated Testing of Autonomous Driving Assistance Systems

Lionel Briand

VVIoT, Sweden, 2018

Collaborative Research @ SnT



- 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

Autonomous Systems

- May be embodied in a device (e.g., robot) or reside entirely in the cyber world (e.g., financial decisions)
- Gaining, encoding, and appropriately using knowledge is a bottleneck for developing intelligent autonomous systems
- Machine learning, e.g., deep learning, is often an essential component

Motivations

- Dangerous tasks
- Tedious, repetitive tasks
- Significant improvements in safety
- Significant reduction in cost, energy, and resources
- Significant optimization of benefits

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

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 …
- Assumption: data captures desirable behavior, in a comprehensive manner
- Example: Neural networks (deep learning)
- Millions of weights learned
- No explicit code, no specifications
- Verification, testing?

Many Domains

- CPS (e.g., robotics)
- Visual recognition
- Finance, insurance
- Speech recognition
- Speech synthesis
- Machine translation
- Games
- Learning to produce art

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
- ...

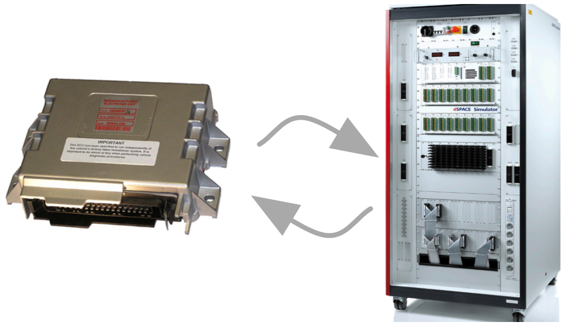
(partial) Code generation

Hardware-in-the-Loop Stage

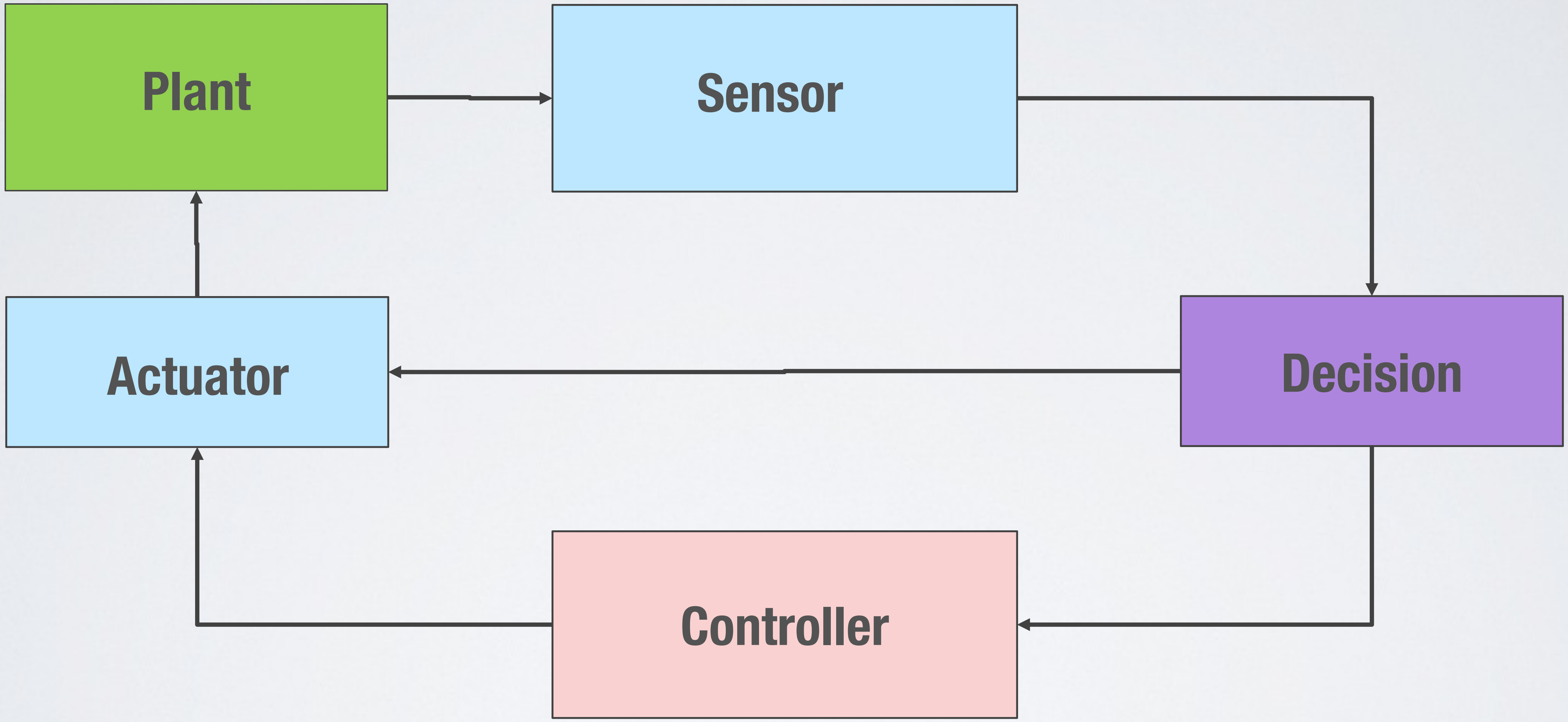
Deployed executables on target platform

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**

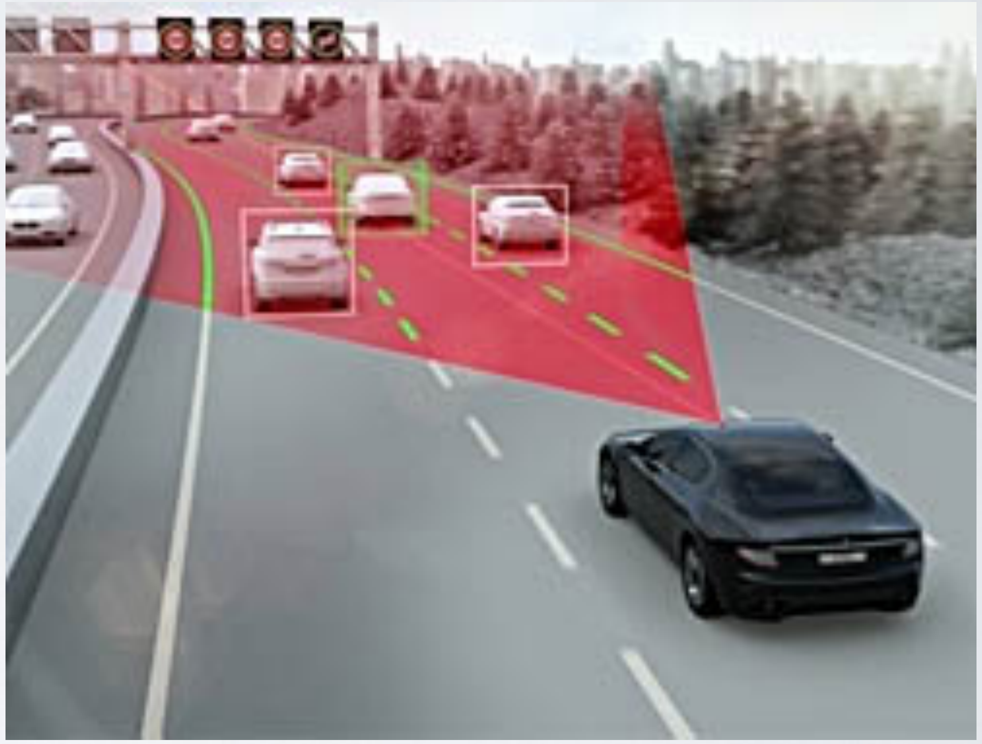
Automotive Environment

- 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

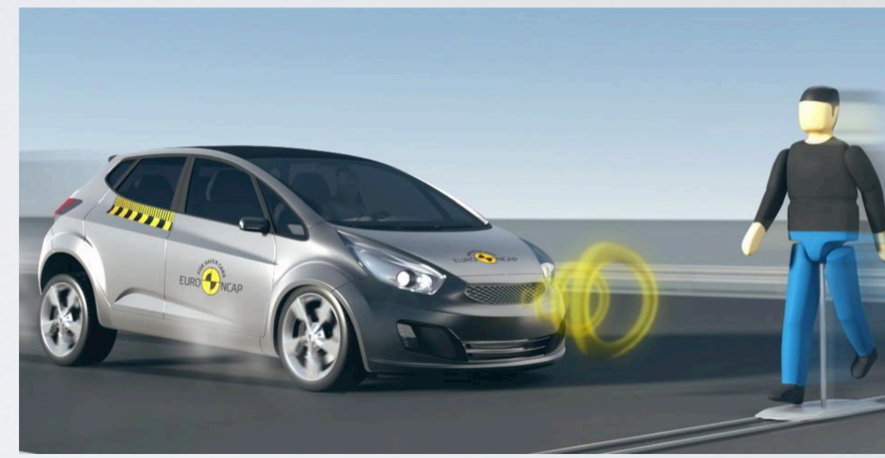
Testing Advanced Driver Assistance Systems

Objective

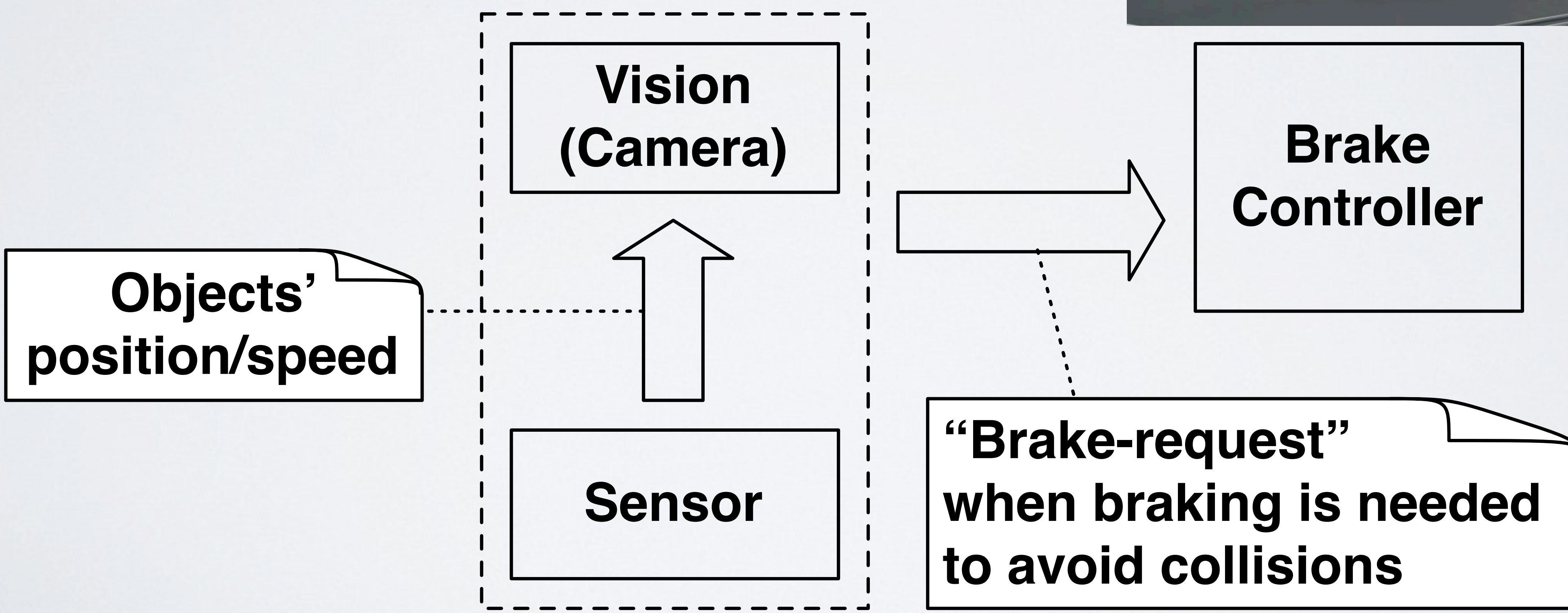
- Testing ADAS
  - Identify and characterize most critical/risky scenarios
  - Test oracle: Safety properties
  - Need scalable test strategy due to large input space



Automated Emergency Braking System (AEB)



Decision making

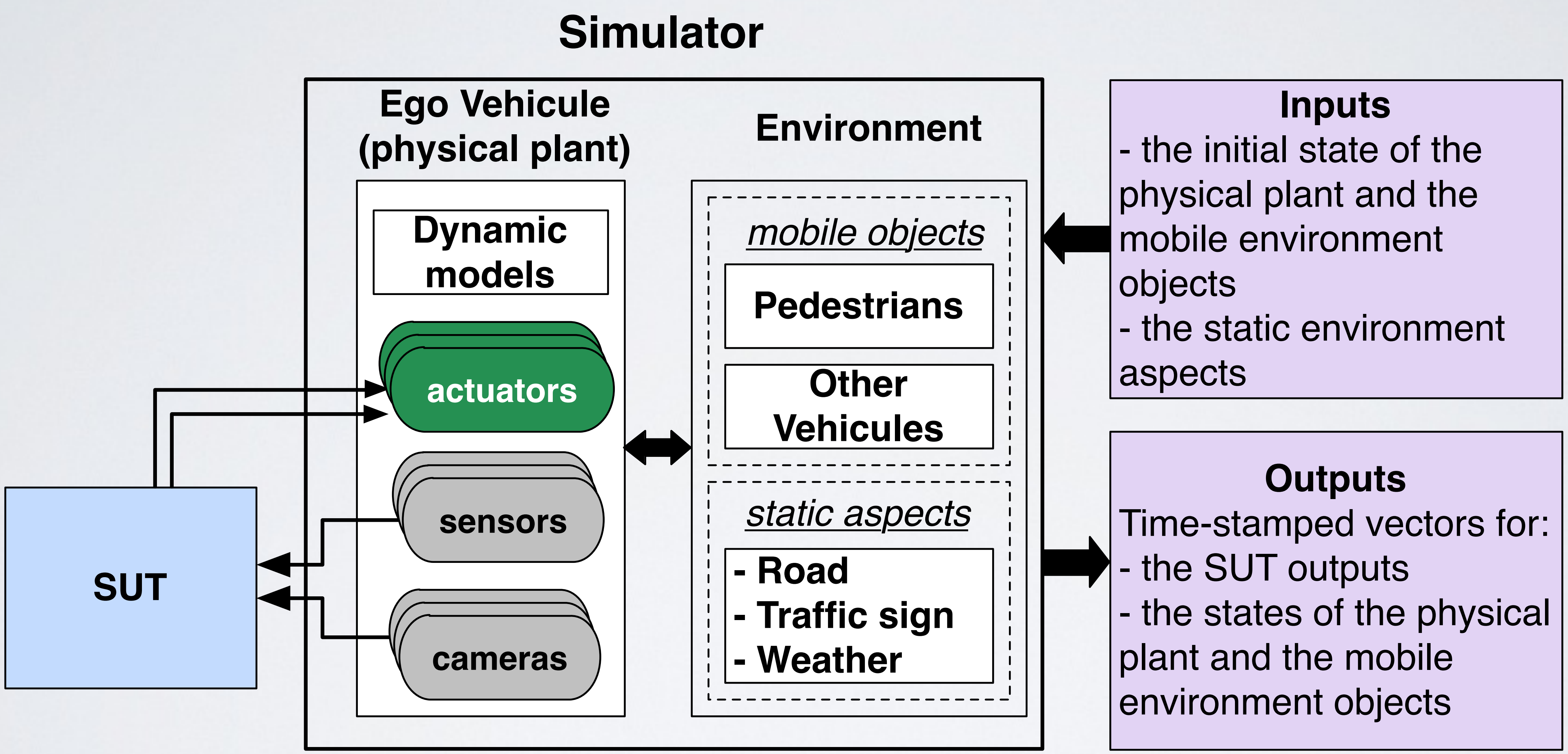


Example Critical Situation

- "AEB properly <u>detects a pedestrian</u> in front of the car with a <u>high degree of certainty</u> and applies braking, but an accident still happens where the car hits the pedestrian with a relatively <u>high speed</u>"

Testing via Physics-based Simulation

Simulation



Feedback loop

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

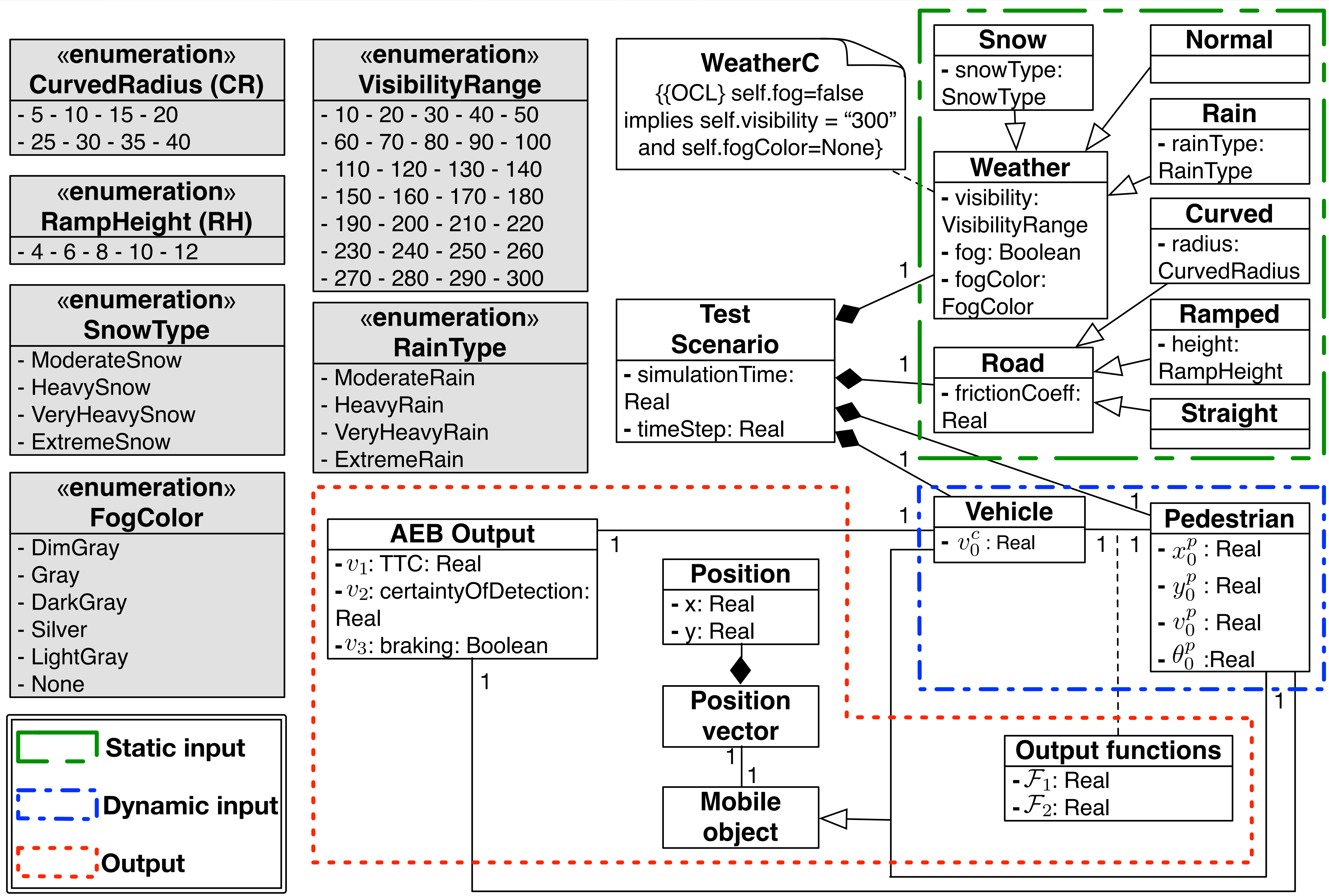
ADAS Testing Challenges

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

Our Approach

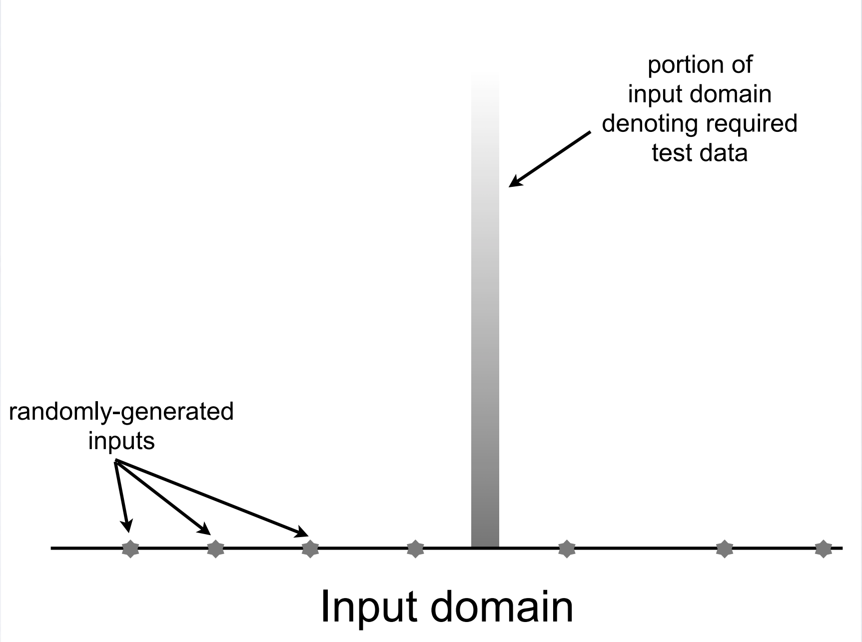
- Effectively combine evolutionary computing algorithms and decision tree classification models
  - Evolutionary computing is used to search the input space for safety violations
  - We use decision tress to guide the search-based generation of tests faster towards the most critical regions, and characterize failures
  - In turn, we use search algorithms to refine classification models to better characterize critical regions of the ADAS input space

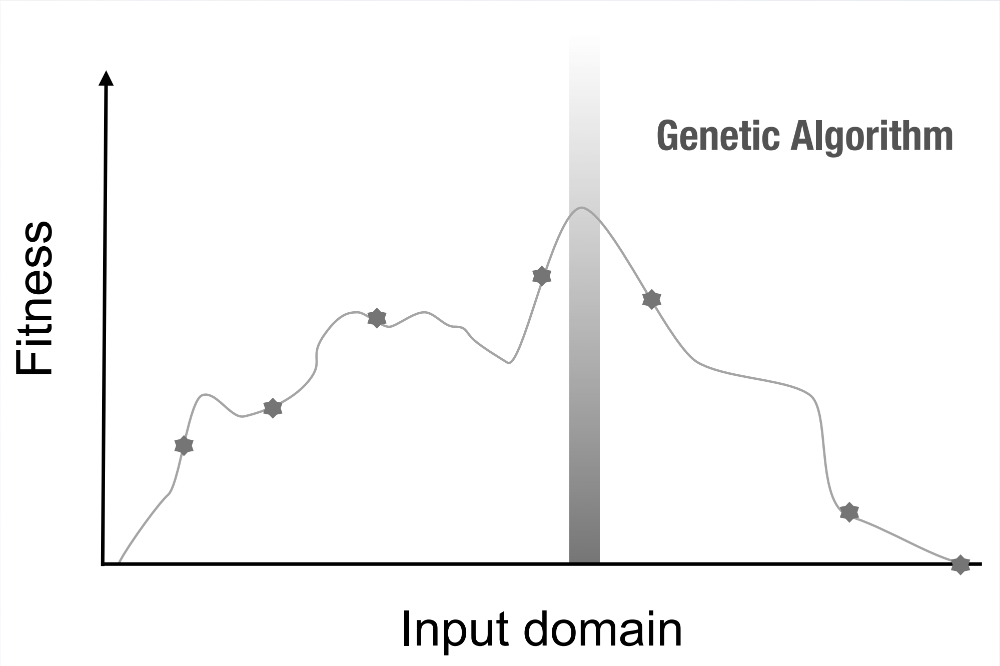
AEB Domain Model



Search-Based Software Testing

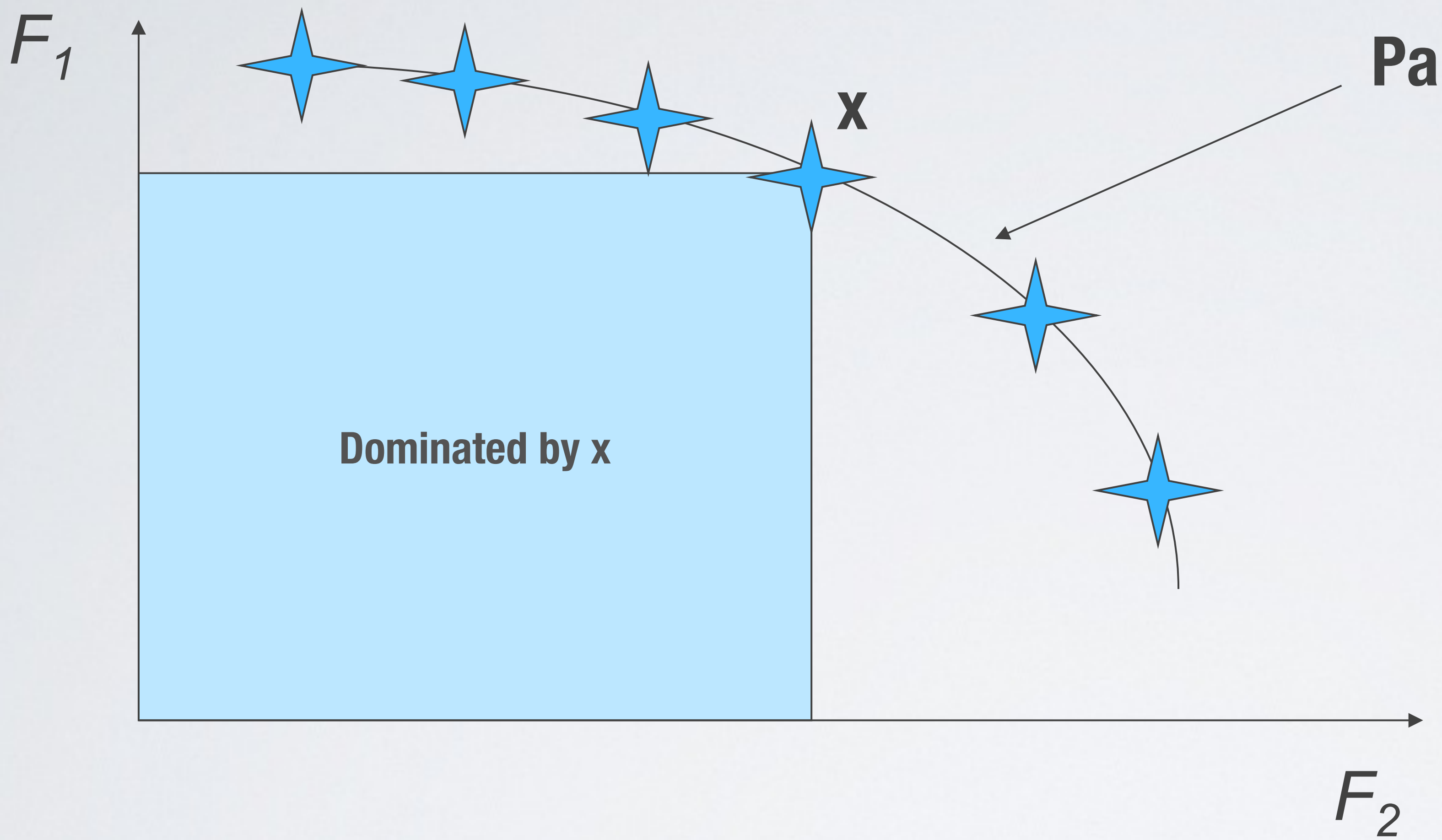
- Express test generation problem as a **search problem**
- Search for **test input data** with certain properties, i.e., constraints
- **Non-linearity** of software (if, loops, …): complex, discontinuous, non-linear search spaces (Baresel)
- Many search algorithms **(metaheuristics)**, from local search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





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

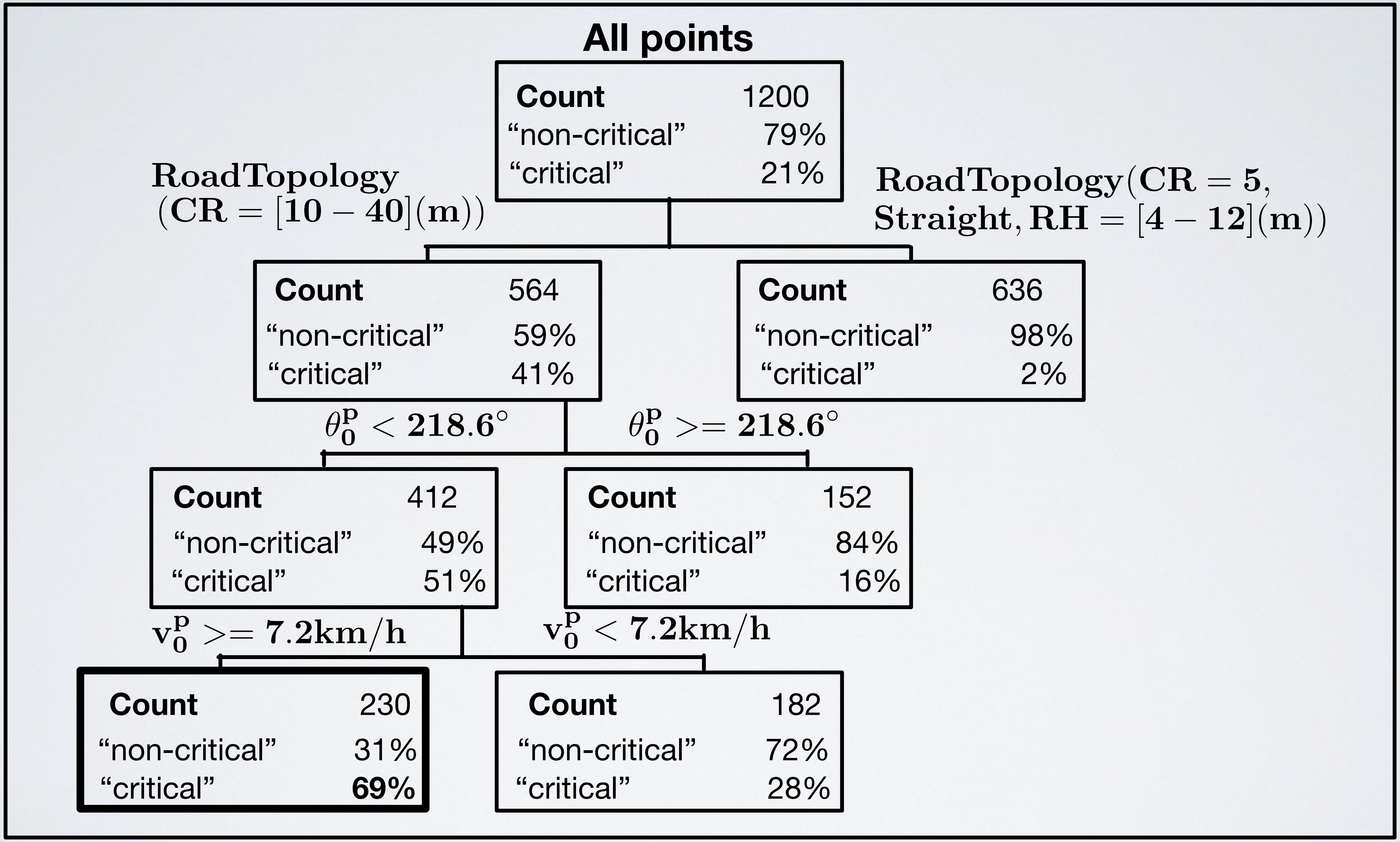
Multiple Objectives: Pareto Front



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.

Decision Trees



Partition the input space into homogeneous regions

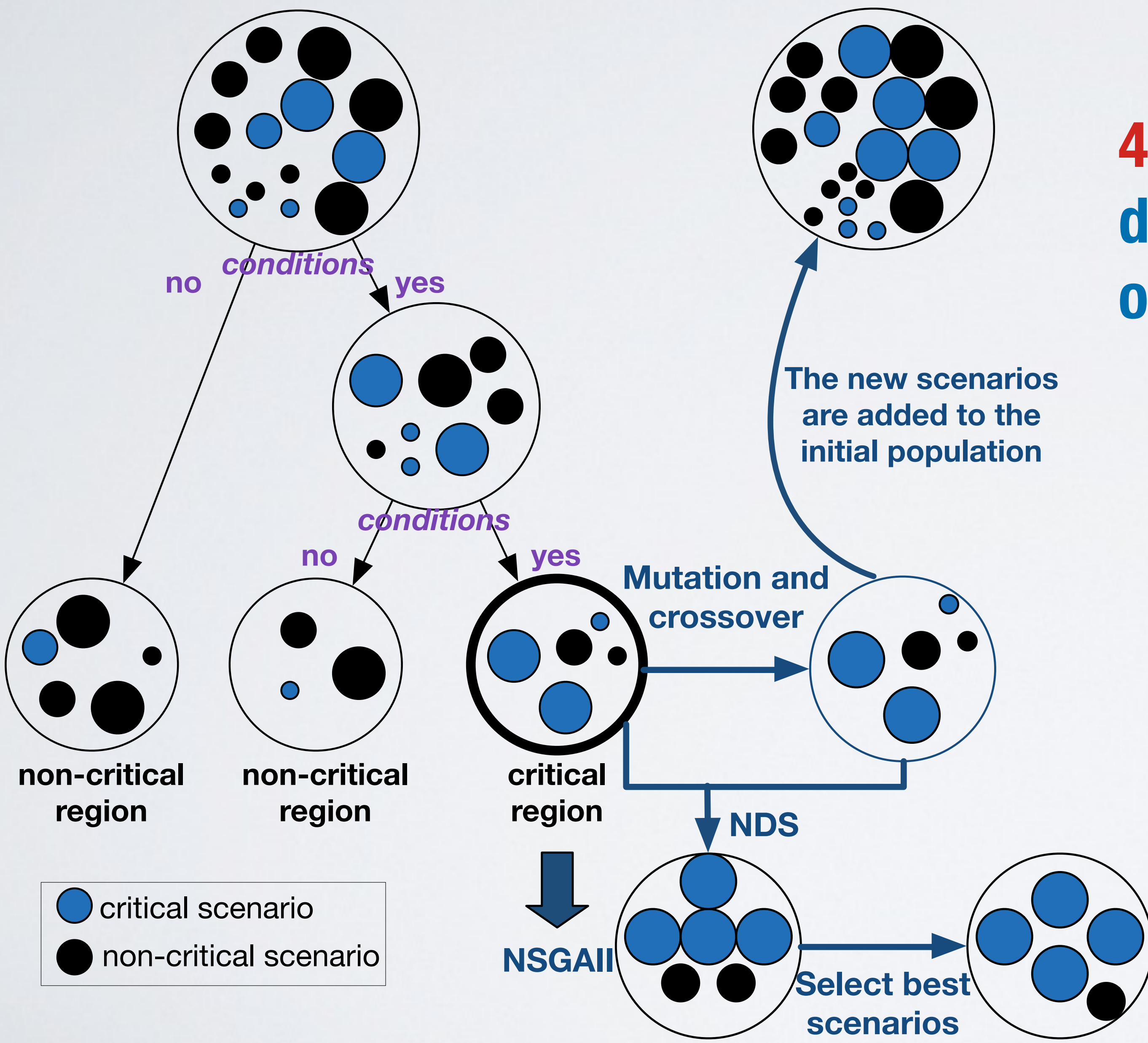
Search Algorithm (NSGAII-DT)

- We use multi-objective search algorithm (NSGAII)
    - Three objectives (CB): Minimum distance between the pedestrian and the field of view, the car speed at the time of collision, and the probability that the object detected in front of the car is a pedestrian
- Inputs are vectors of values containing static and dynamic variables: precipitation, fogginess, road shape, visibility range, car-speed, person-speed, person-position (x,y), person-orientation
- Each search iteration calls simulations to compute fitness
- We use decision tree classification models to predict scenario criticality

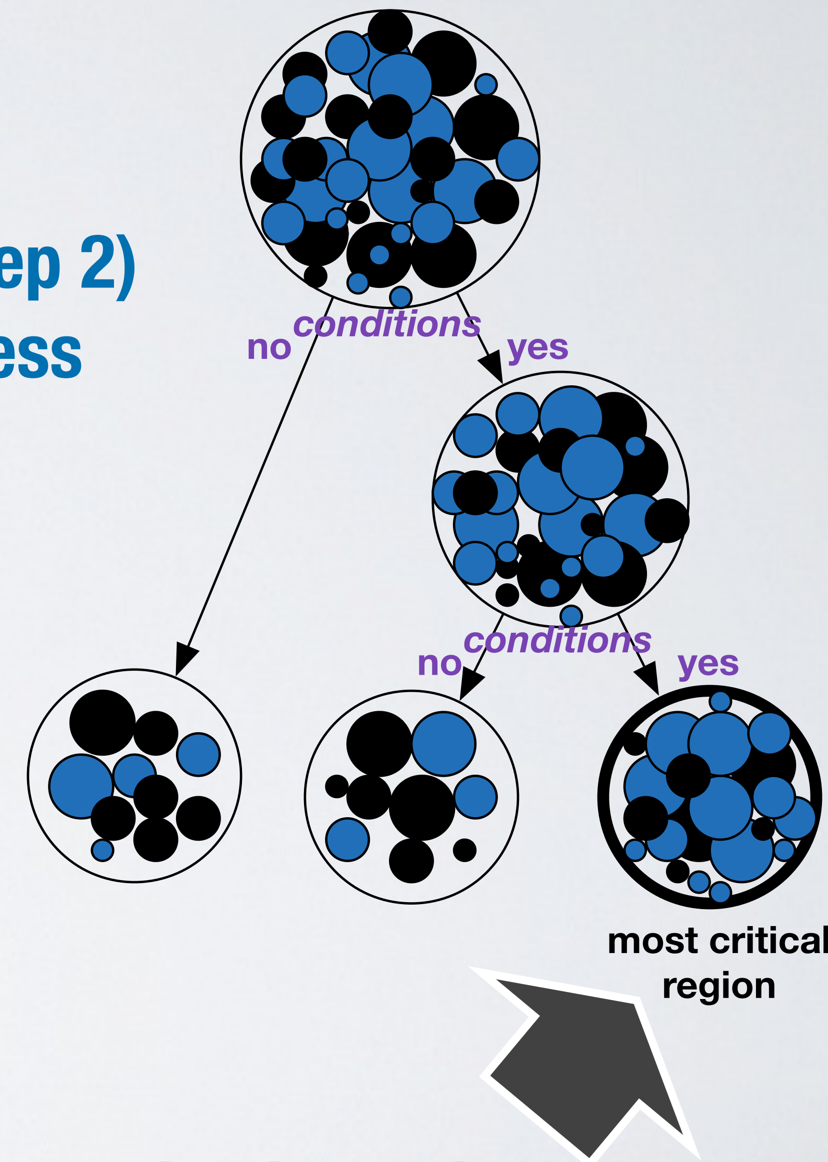
NSGAII-DT

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



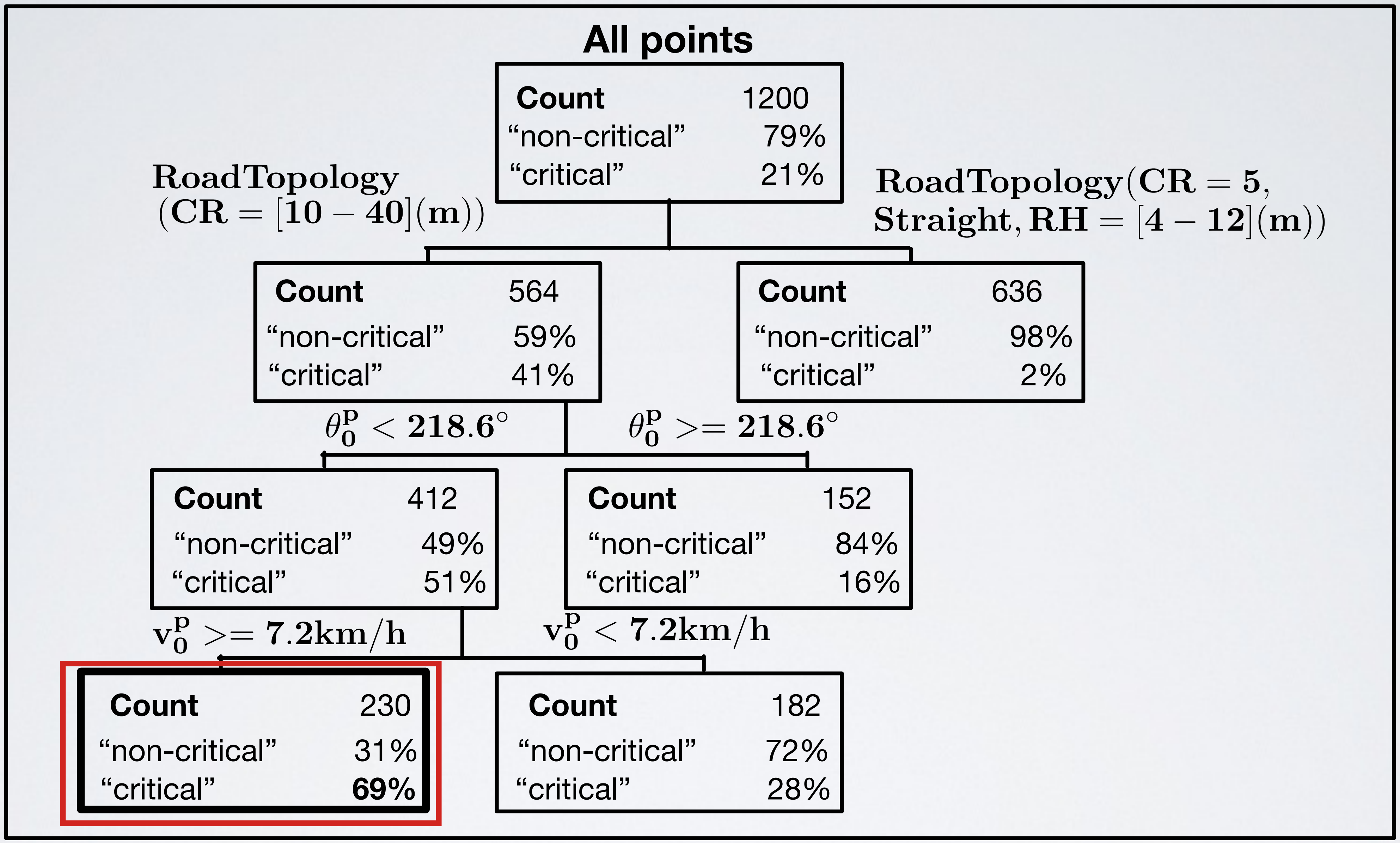
4. Rebuild the decision tree (step 2) or stop the process



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

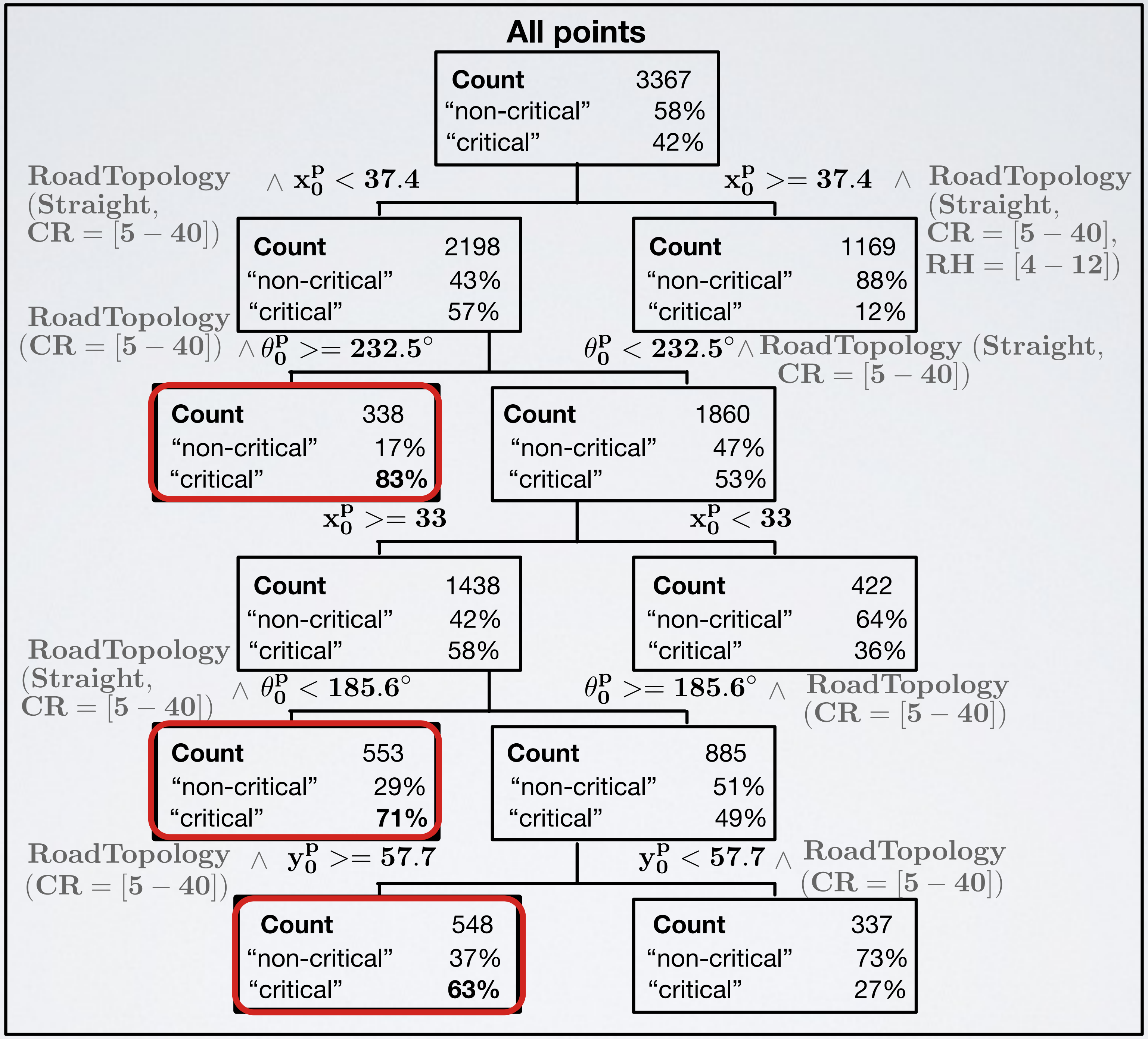
Region in the input space that is likely to contain more critical scenarios

Initial Classification Model



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

Refined Classification Model

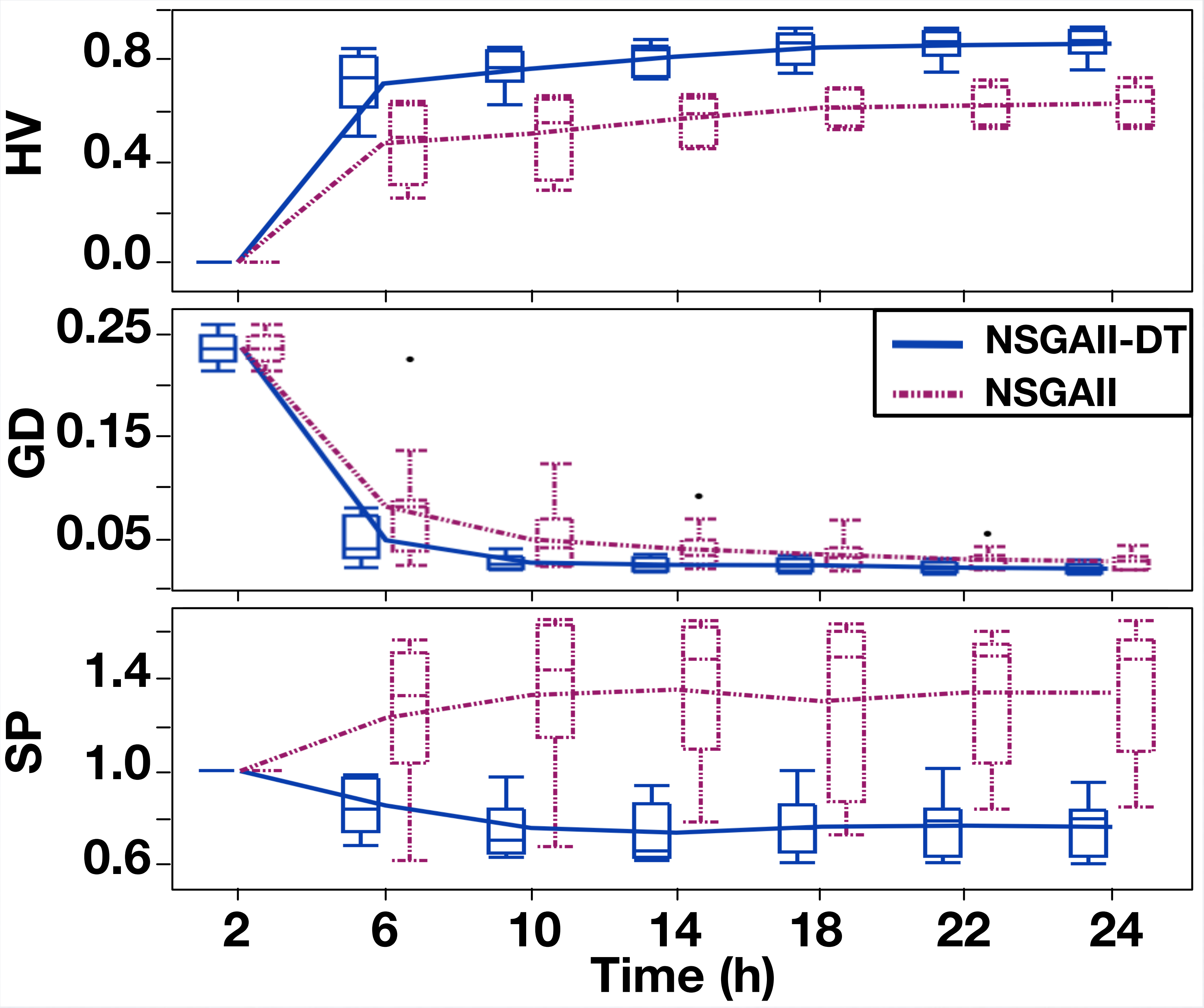


We get a more refined decision tree with more critical regions and more homogeneous areas

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

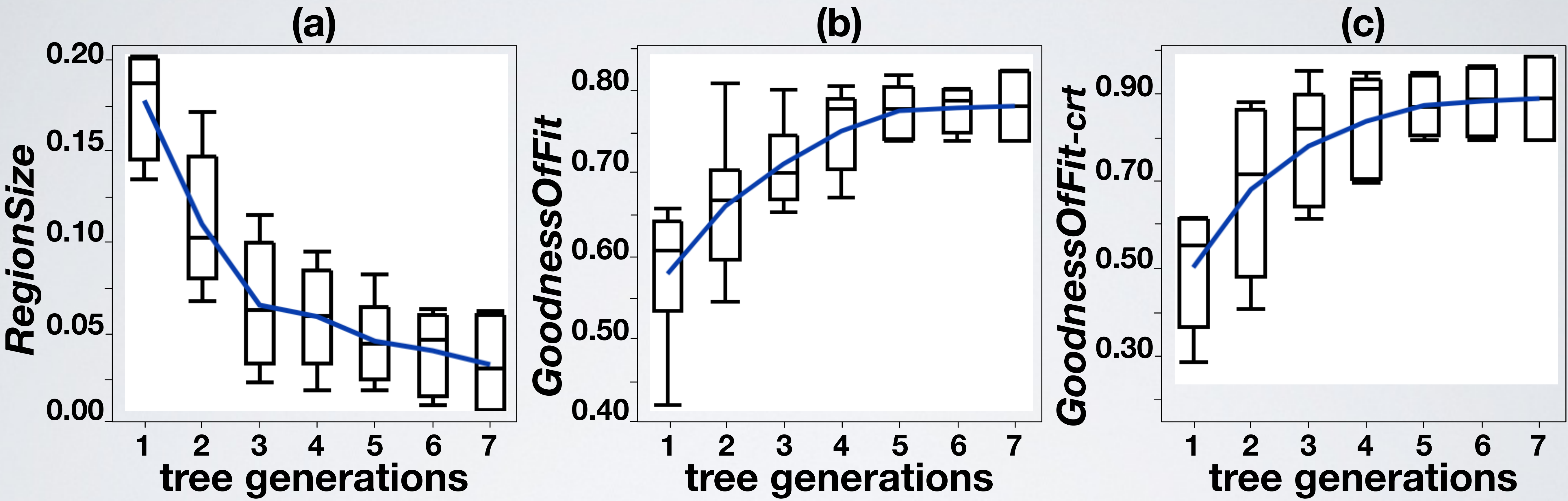


NSGAII-DT outperforms NSGAII

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)



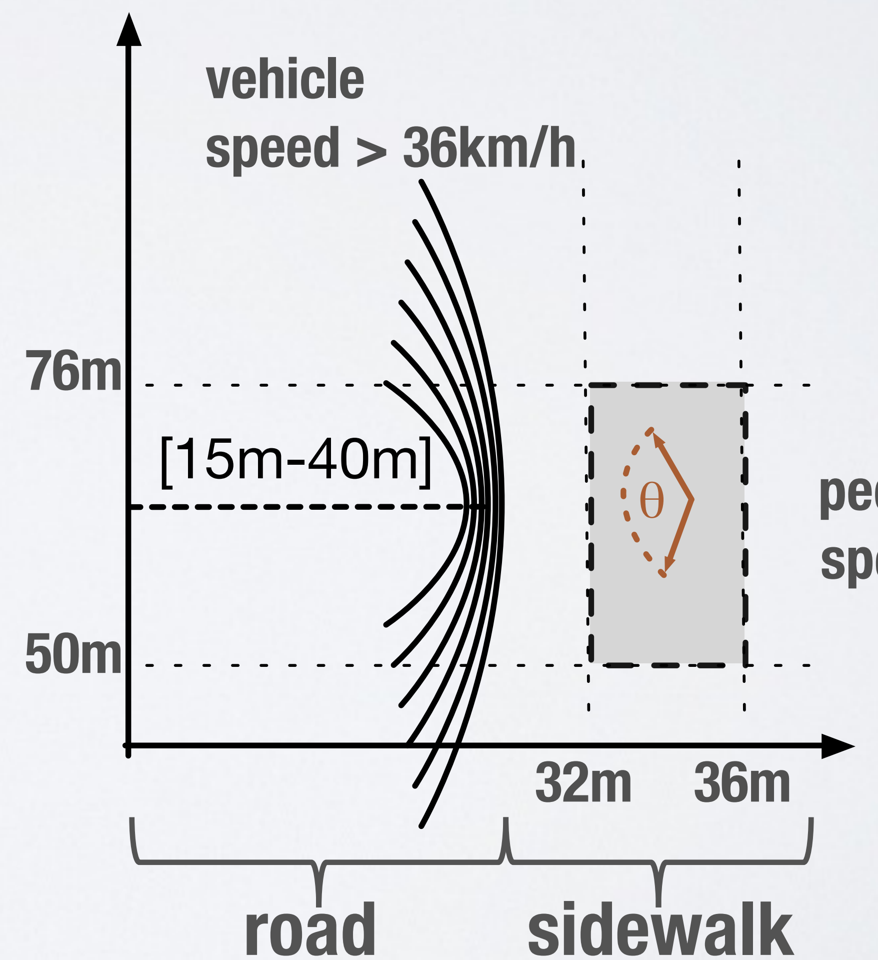
The generated critical regions consistently become smaller, more homogeneous and more precise over successive tree generations of NSGAII-DT

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)

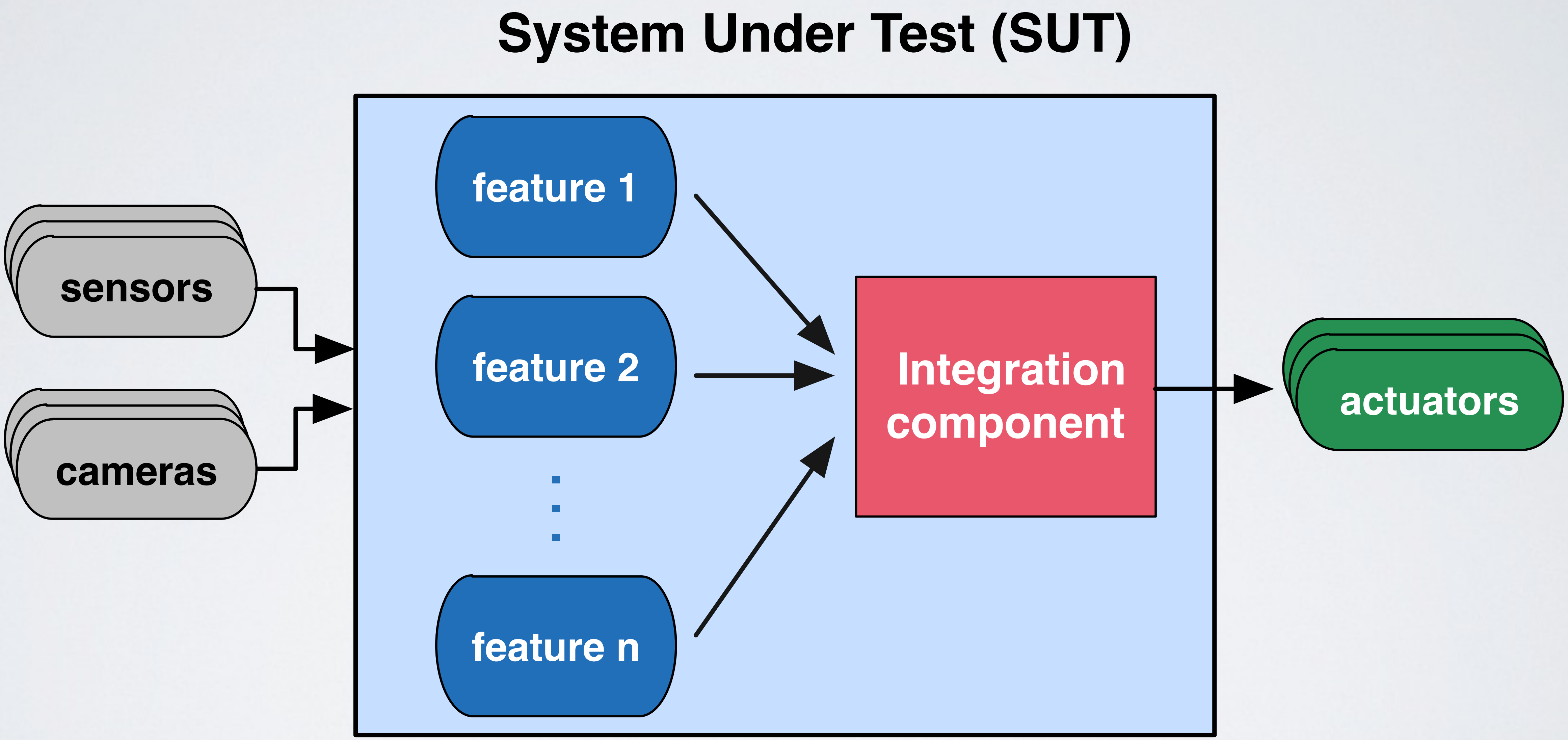
  (2) **Identifying possible hardware changes** to increase ADAS safety

  (3) **Providing proper warnings** to drivers

Automated Testing of Feature Interactions Using Many Objective Search

a sense for innovation IEE

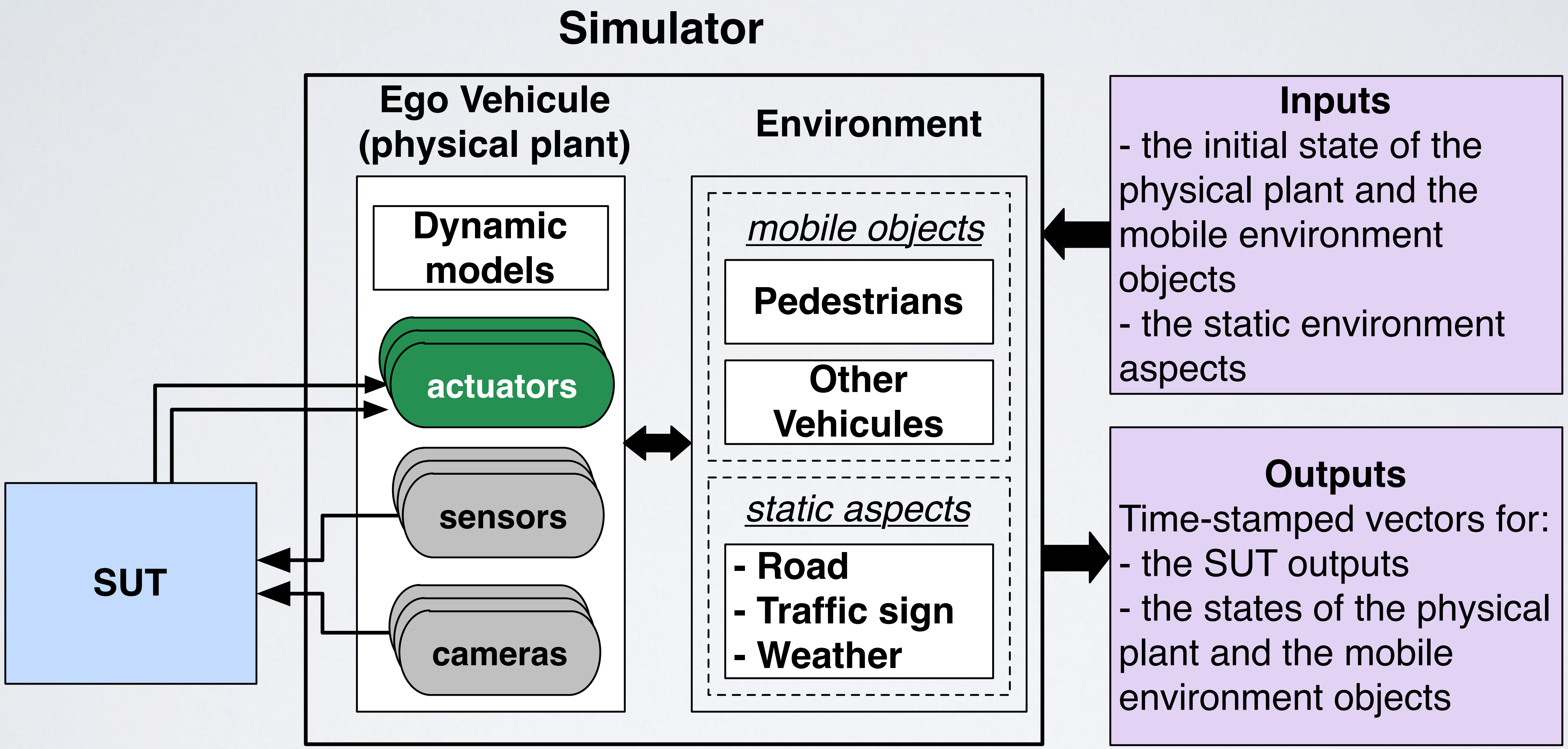
System Integration



Case Study: SafeDrive

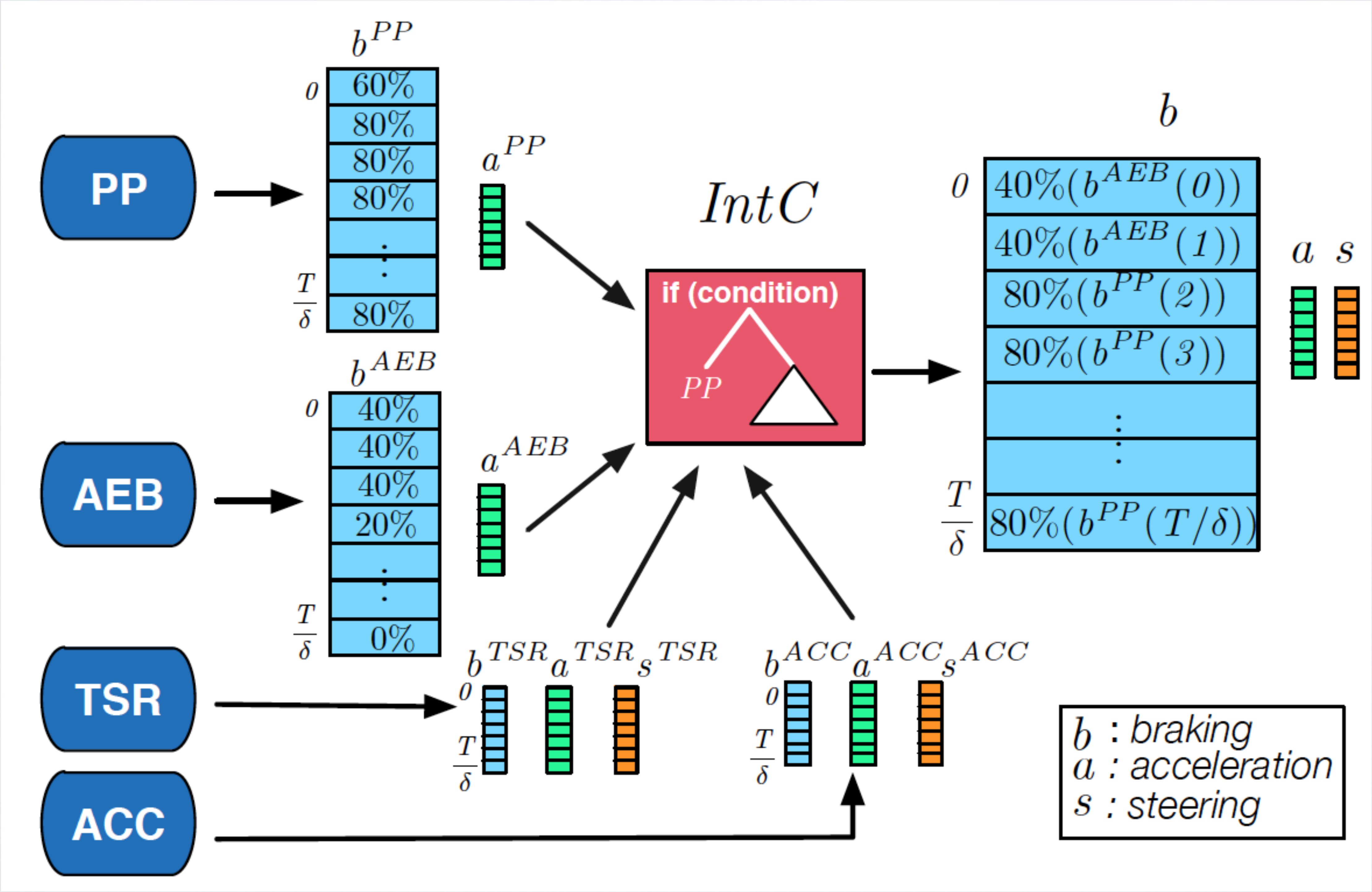
- Our case study describes an automotive system consisting of **four advanced driver assistance features**:
  - Cruise Control (ACC)
  - Traffic Sign Recognition (TSR)
  - Pedestrian Protection (PP)
  - Automated Emergency Breaking (AEB)

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 $FD_2(i)$.

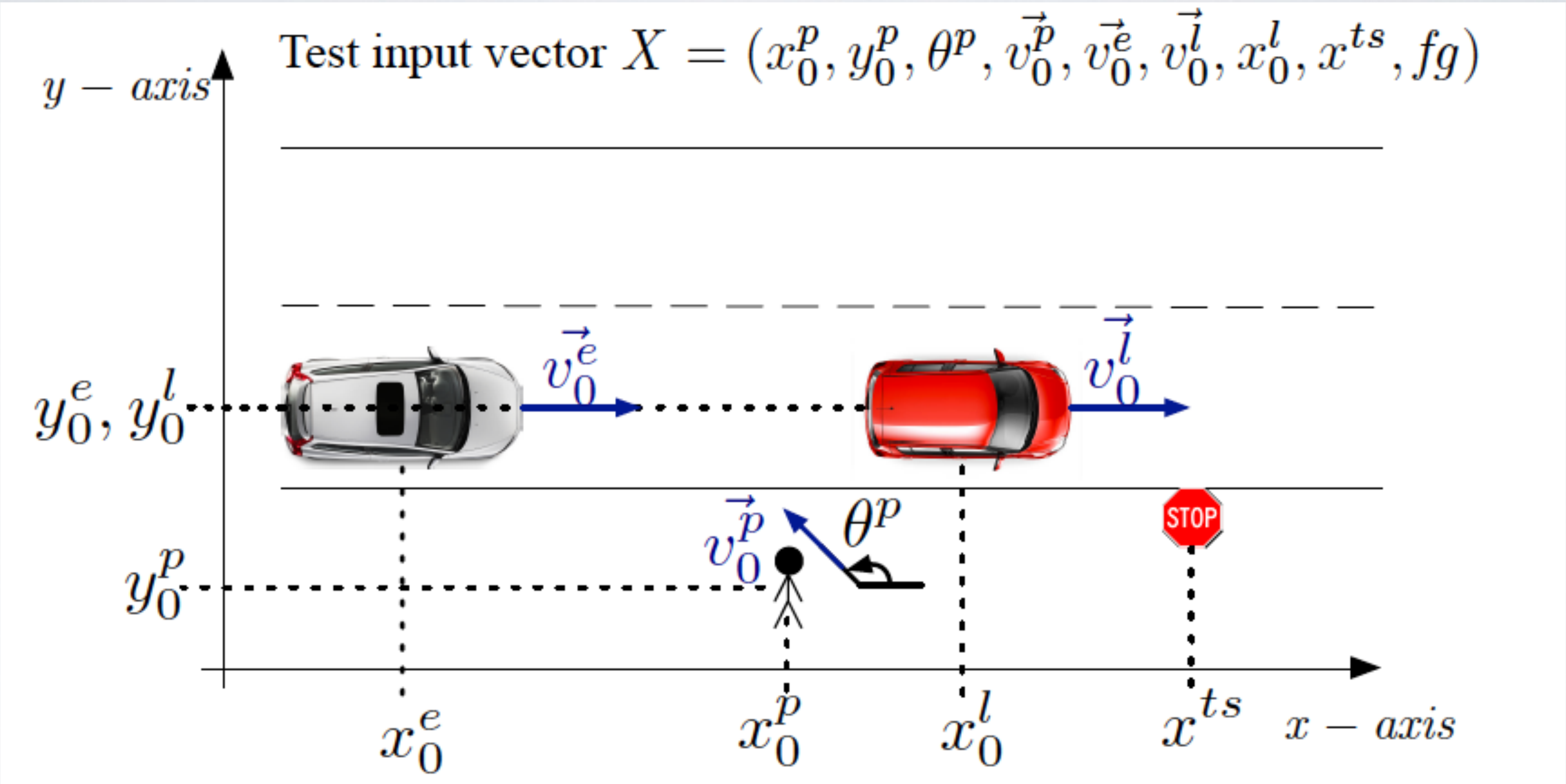
Features

- 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
- **Extremely large input space** since environmental conditions and scenarios can vary a great deal

Input Variables



Search

- Input space is 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 IntC, (3) coverage of the decision structure of integration component
- Many test objectives to be satisfied by the test suite

Failure Distance

- 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 ... define ... have
ACC		...stance

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

Unsafe Overriding Distance

- **Goal:** Find faults more likely to be due to 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

- **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
begin
    if (Target_brake[1] > 0 and Pedestrian is detected and Object_distance is Close)
    then
        Brake ← Target_brake[1] /*b1
    else
        if (Target_brake[2] > 0 and Car is detected and Object_distance is Close)
        then
            Brake ← Target_brake[2] /*b2
        else
            if TrafficSign is pedestrians crossing and LevelOfConfidence[1] is
            Low then
                Brake ← Target_brake[1] /*b3
            else
                if StopSign then
                    Brake ← Target_brake[4] /*b4
                else
                    if (Target_brake[1] > 0 or Target_throttle[1] > 0) then
                        Brake ← Target_brake[1] /*b5
                        Throttle ← Target_throttle[1]
                    else
                        if (Target_brake[2] > 0 or Target_throttle[2] > 0)
                        then
                            Brake ← Target_brake[2] /*b6
                            Throttle ← Target_throttle[2]
                        else
                            if ((Target_brake[4] > 0 or Target_throttle[4] >
                            0) and speed_limit) < speed_LeadCar then
                                Brake ← Target_brake[4] /*b7
                                Throttle ← Target_throttle[4]
                            else
                                if (Target_brake[3] > 0 or
                                Target_throttle[3] > 0) then
                                    Brake ← Target_brake[3] /*b8
                                    Throttle ← Target_throttle[3]
                                else
                                    Brake ← 0 /*b9
                                    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 the branch *j*

$2 \geq \Omega_{j,l}(tc) > 1$ Branch covered but did not caused 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
- 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)**

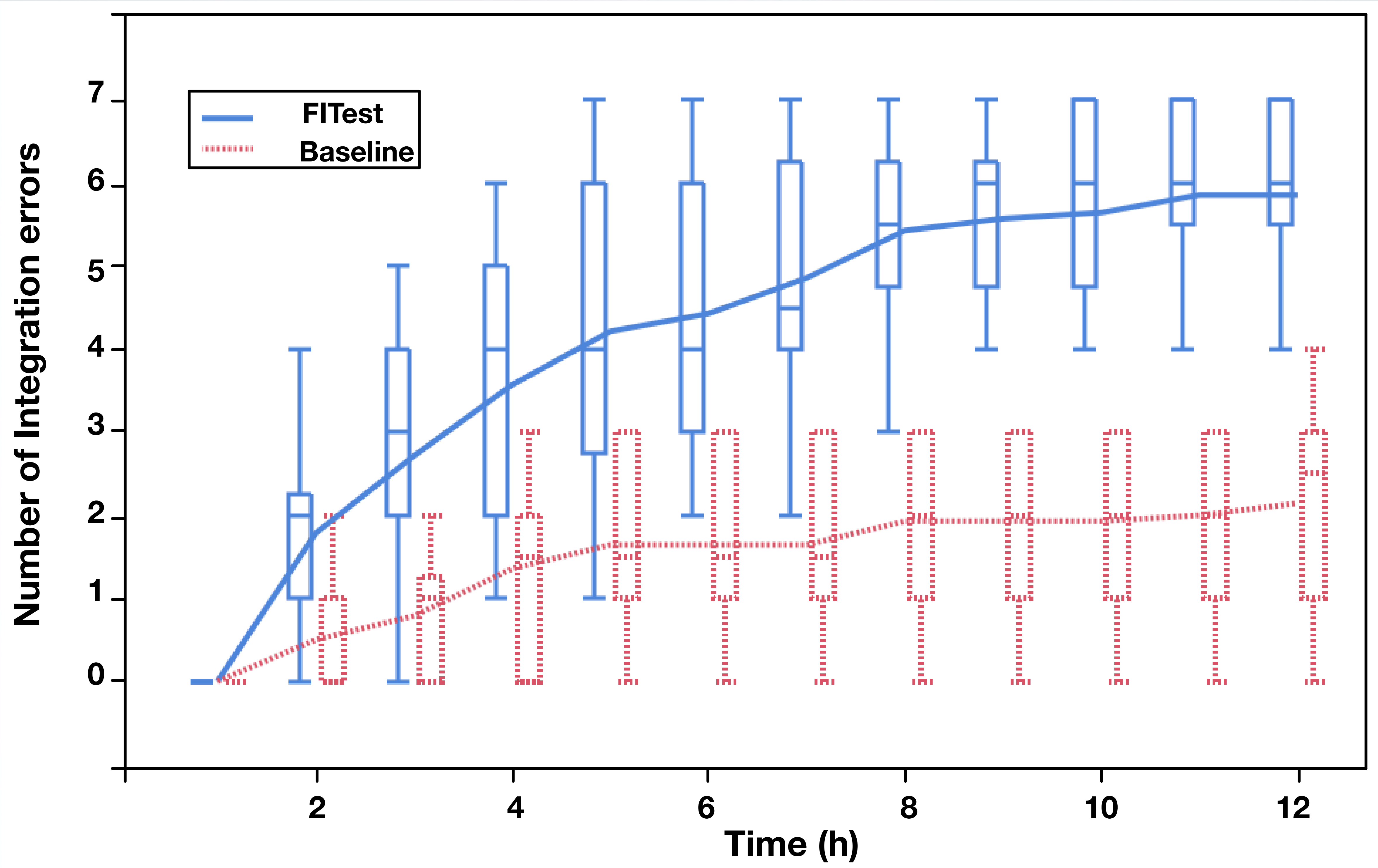
Search Algorithm

Algorithm 1: FITEST

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

begin
- $P \longleftarrow$ RANDOM-POPULATION(N) — Randomly generated TCs
- $W \longleftarrow$ CALCULATE-OBJECTIVES(P, Ω) — Compute fitness
- $[\Omega_c, T_c] \longleftarrow$ GET-COVERED-OBJECTIVE(P,W) — Archive covering tests
- $A \longleftarrow T_c$
- $\Omega \longleftarrow \Omega - \Omega_c$
- **while** *not (stop_condition)* **do** — Tests are evolved
  - $Q \longleftarrow$ RECOMBINE(P,N) — Crossover, mutation
  - $Q \longleftarrow$ CORRECT-OFFSPRINGS(Q) — Correct constraint violations
  - $W \longleftarrow$ CALCULATE-OBJECTIVES(Q, Ω)
  - $[\Omega_c, T_c] \longleftarrow$ GET-COVERED-OBJECTIVE(P,W)
  - $A \longleftarrow A \cup T_c$
  - $\Omega \longleftarrow \Omega - \Omega_c$
  - $\mathbb{F}_0 \longleftarrow$ ENVIRONMENTAL-SELECTION($P \cup Q, \Omega$) — Fittest tests selected
  - $P \longleftarrow F_0$
  - $N \longleftarrow |F_0|$
- **return** A

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 will rarely have precise and complete requirements, 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
- Failure is not clear cut: It is a matter of risk, trade-off …
- 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
- No guarantees though

Generalizing

- Examples presented from (safety-critical) cyber-physical systems, e.g., safety requirements

- 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?

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 Systems

Lionel Briand

VVIoT, Sweden, 2018