Making Model-Driven Verification Practical and Scalable - Experiences and Lessons Learned

Lionel Briand

Interdisciplinary Centre for ICT Security, Reliability, and Trust (SnT)
University of Luxembourg, Luxembourg

MODELSWARD, Rome, February 20, 2016

Acknowledgements

PhD. Students:

- Vahid Garousi
- Marwa Shousha
- Zohaib Iqbal
- Reza Matinnejad
- Stefano Di Alesio
- Raja Ben Abdessalem

Others:

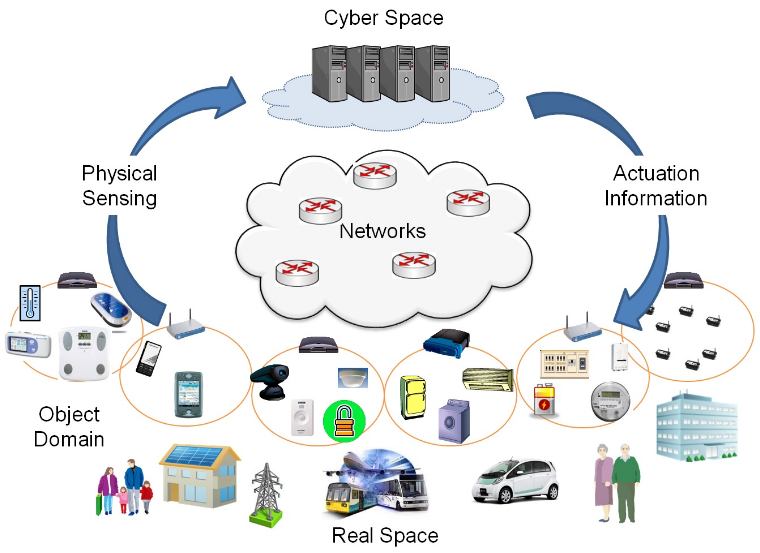
- Shiva Nejati
- Andrea Arcuri
- Yvan Labiche

Verification, Testing

- The term “verification” is used in its wider sense: Defect detection.
- Testing is, in practice, the most common verification technique.
- Testing is about systematically, and preferably automatically, exercise a system such as to maximize chances of uncovering (important) latent faults within time constraints.
- Other forms of verifications are important too (e.g., design time, run-time), but much less present in practice.
- Decades of research have not yet significantly and widely impacted engineering practice.

Cyber-Physical Systems: Challenges

- Increasingly complex and critical systems
- Complex environment
- Hybrid discrete and continuous behavior
- Combinatorial and state explosion
- Complex requirements, e.g., temporal, timing, resource usage
- Uncertainty, e.g., about the environment



Scalable? Practical?

- *Scalable:* Can a technique be applied on large artifacts (e.g., models, data sets, input spaces) and still provide useful support within reasonable effort, CPU and memory resources?

- *Practical:* Can a technique be efficiently and effectively applied by engineers in realistic conditions?
  - realistic ≠ universal

Focus

- *Formal Verification (Wikipedia):* In the context of **hardware** and **software** systems, **formal verification** is the act of **proving or disproving** the correctness of intended algorithms underlying a system with respect to a certain **formal** specification or property, using **formal** methods of mathematics.

- *Our focus:* How can we, in a **practical, effective and efficient** manner, uncover as many (critical) faults as possible in **software systems**, within **time constraints**, while **scaling** to artifacts of realistic size.

Metaheuristics

- *Heuristic search (Metaheuristics):* Hill climbing, Tabu search, Simulated Annealing, Genetic algorithms, Ant colony optimisation ….
- *Stochastic optimization*: General class of algorithms and techniques which employ some degree of randomness to find optimal (or as optimal as possible) solutions to hard problems
- Many verification and testing problems can be re-expressed as (hard) optimization problems

Talk Outline

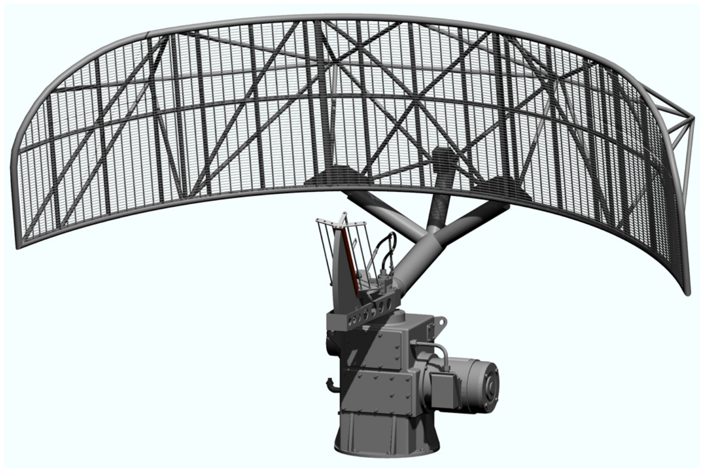
- Selected project examples, with industry collaborations
- Similarities and patterns
- Lessons learned

Testing Software Controllers

References:

- *R. Matinnejad et al., "Automated Test Suite Generation for Time-continuous Simulink Models", IEEE/ACM ICSE 2016*
- *R. Matinnejad et al., "Effective Test Suites for Mixed Discrete-Continuous Stateflow Controllers", ACM ESEC/FSE 2015 (Distinguished paper award)*
- *R. Matinnejad et al., "MiL Testing of Highly Configurable Continuous Controllers: Scalable Search Using Surrogate Models", IEEE/ACM ASE 2014 (Distinguished paper award)*
- *R. Matinnejad et al., "Search-Based Automated Testing of Continuous Controllers: Framework, Tool Support, and Case Studies", Information and Software Technology, Elsevier (2014)*

Dynamic Continuous Controllers



Electronic Control Units (ECUs)

Comfort and variety

More functions

Safety and reliability

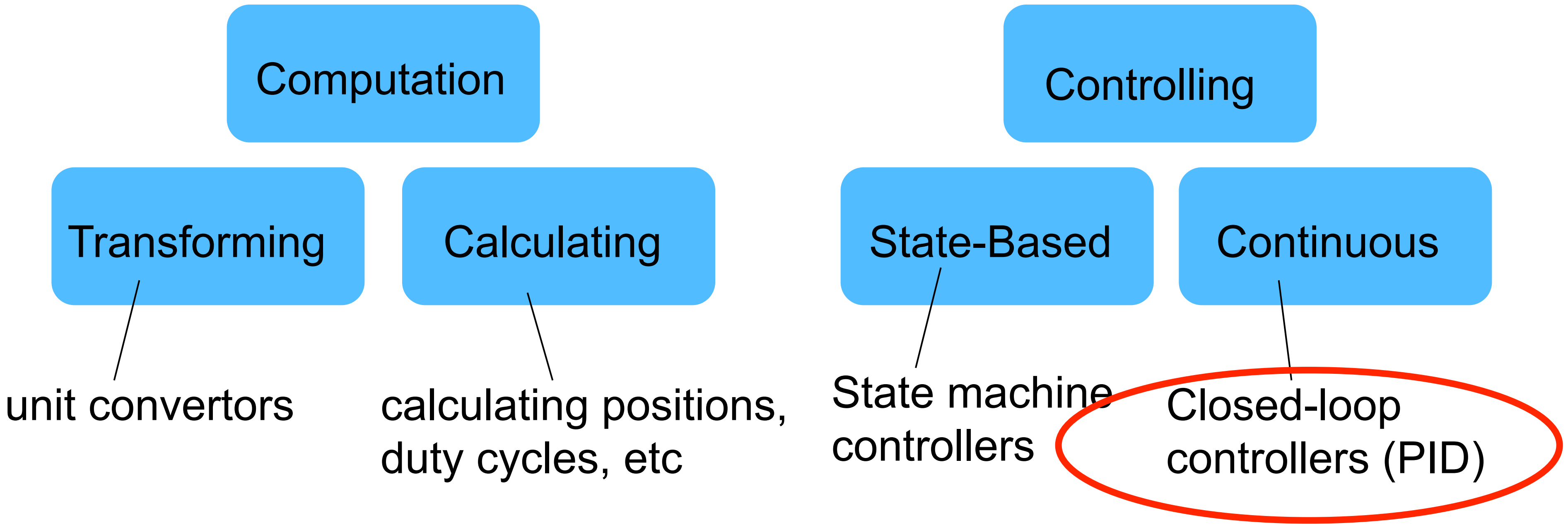
DELPHI
Automotive Systems

Faster time-to-market

Greenhouse gas emission laws

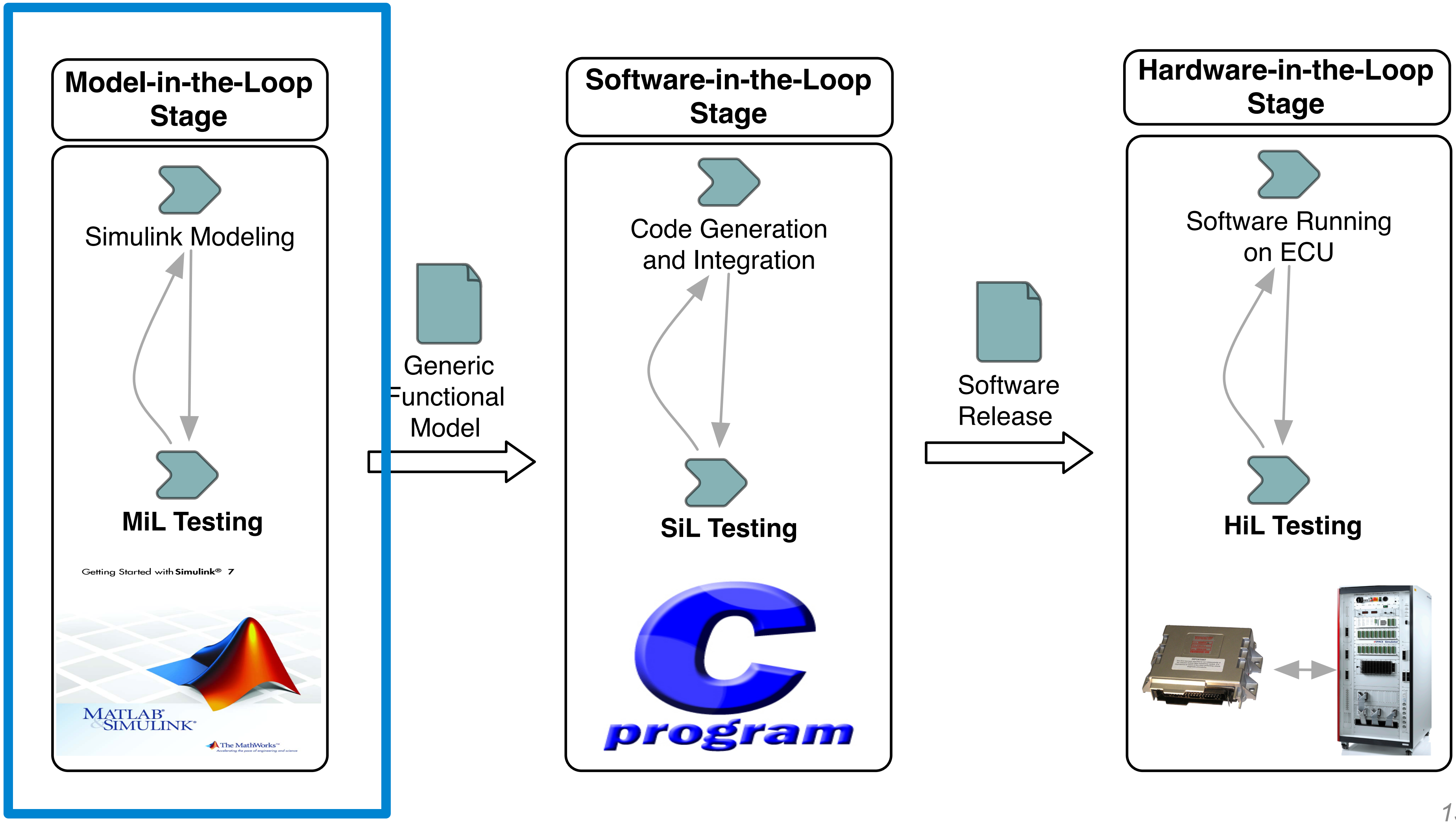
Less fuel consumption

A Taxonomy of Automotive Functions

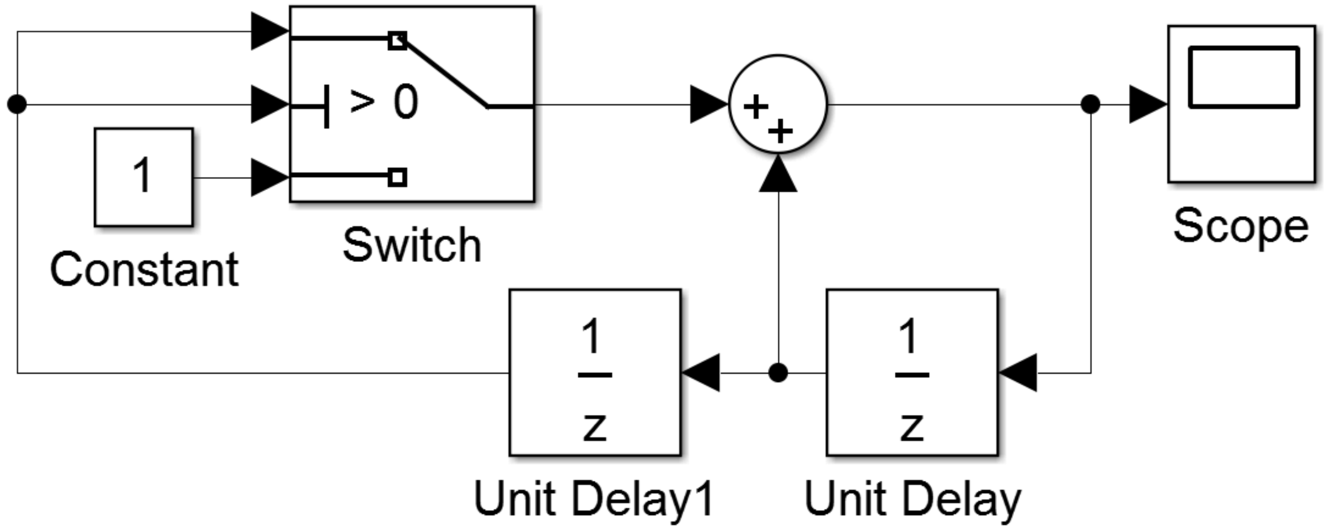

Computation
Controlling
Transforming
Calculating
State-Based
Continuous
unit convertors
calculating positions, duty cycles, etc
State machine controllers
Closed-loop controllers (PID)

Different testing strategies are required for different types of functions

Development Process

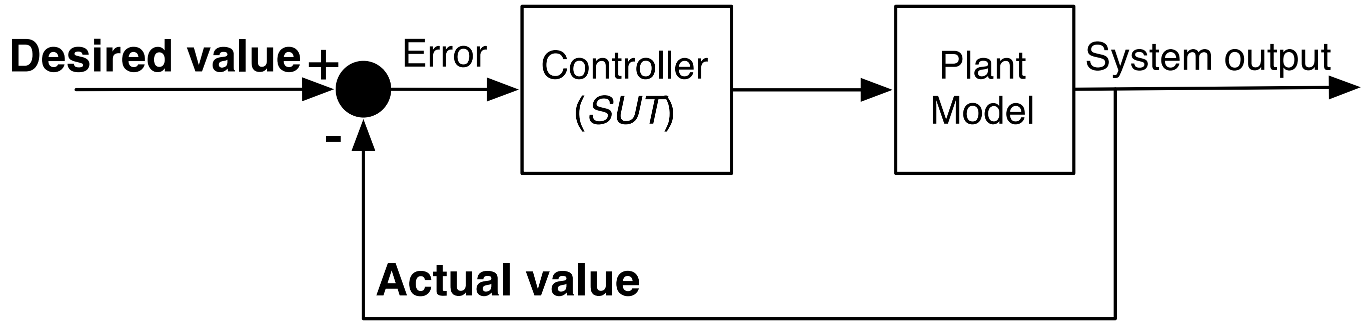


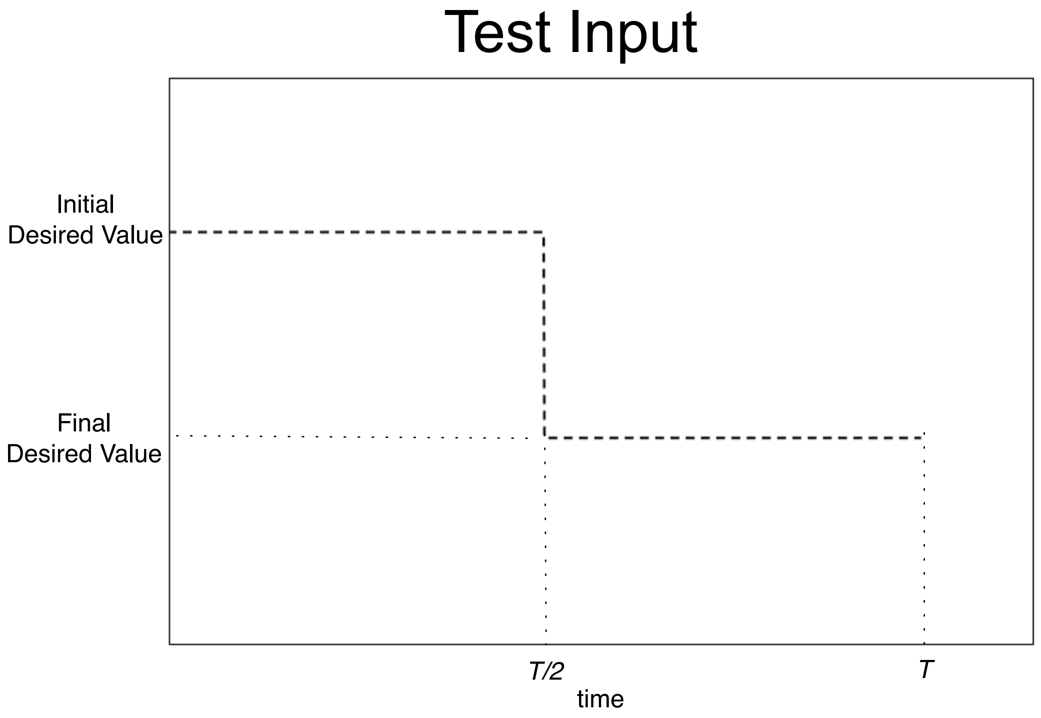
MATLAB/Simulink model

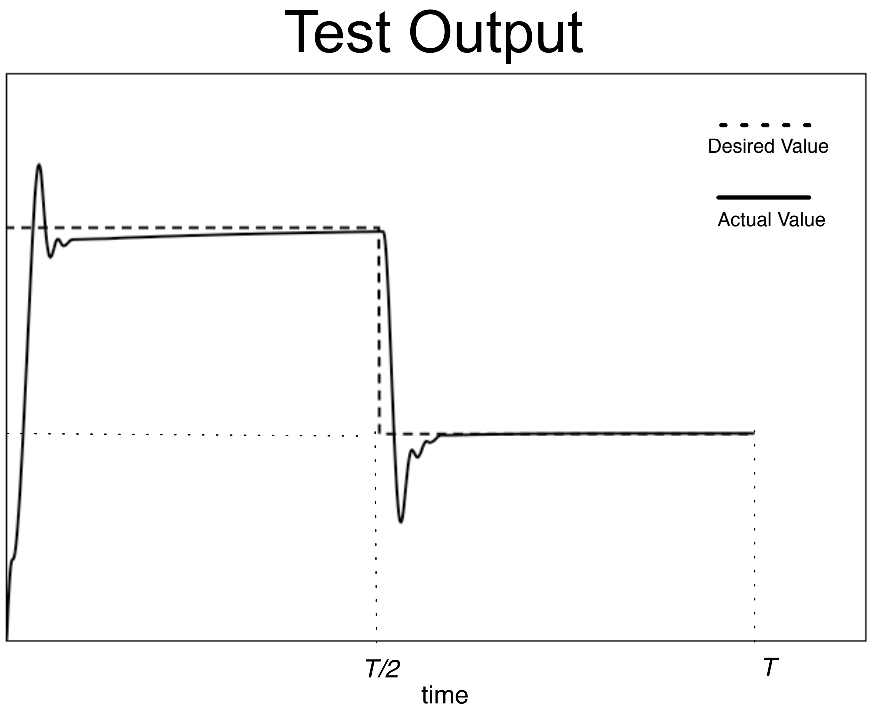


Fibonacci sequence: 1,1,2,3,5,8,13,21,…

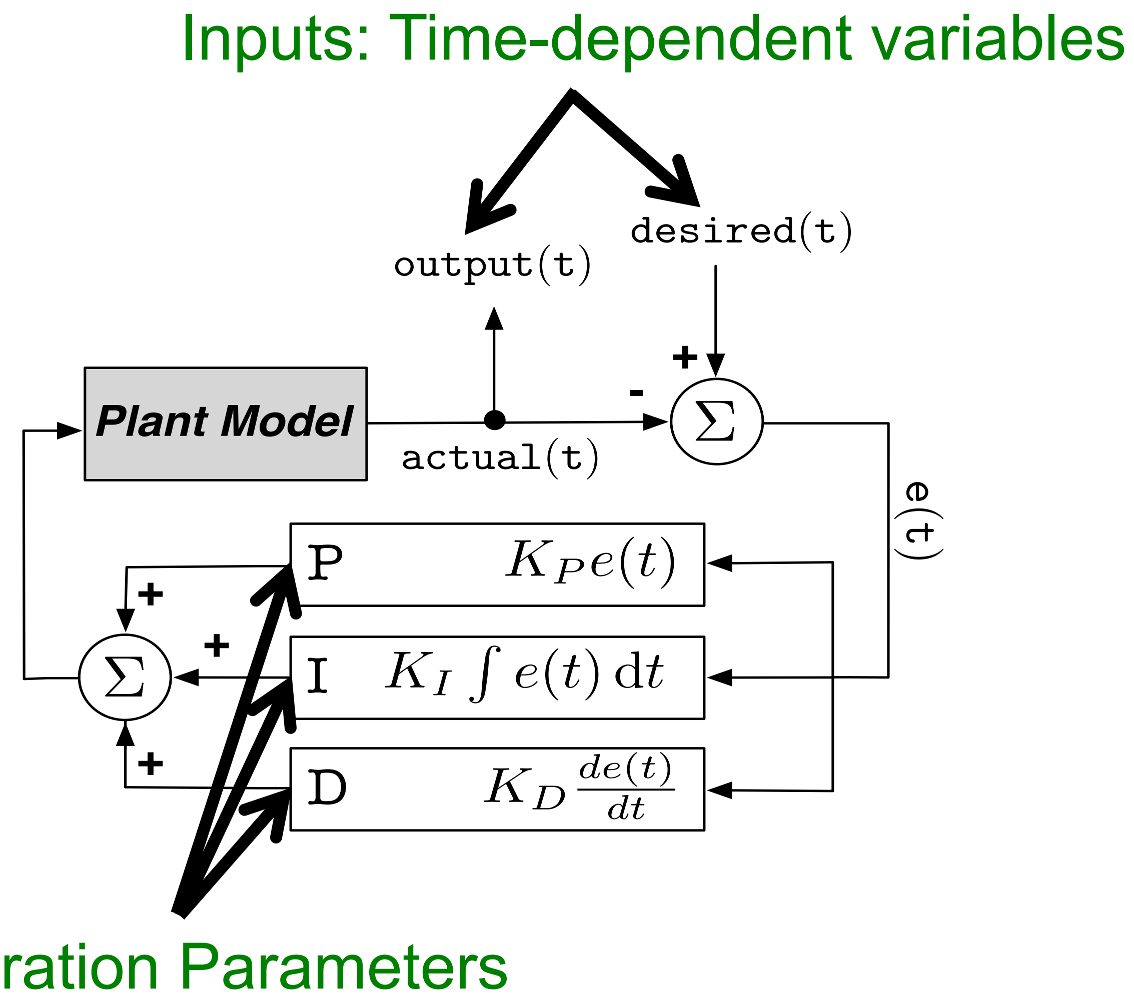
Controller Input and Output at MIL



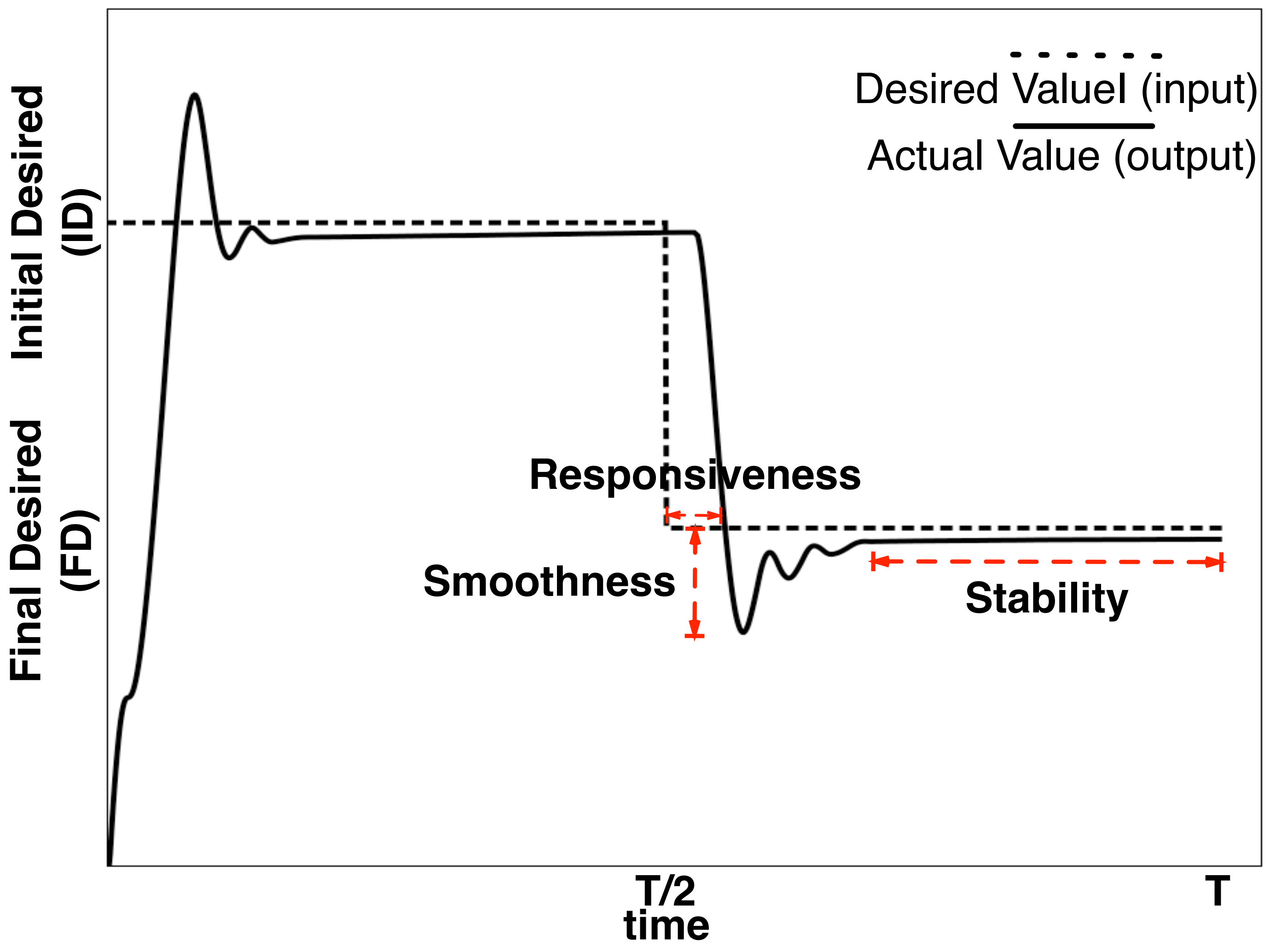




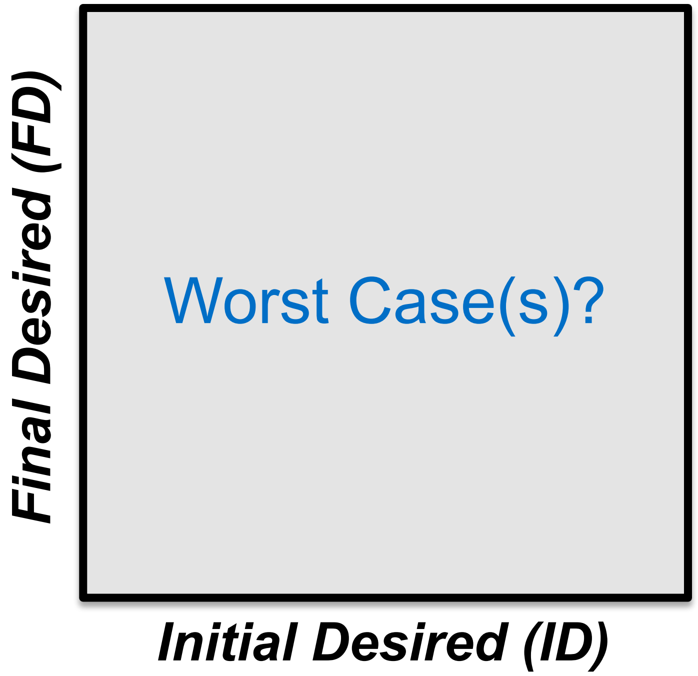
Controllers at MIL



Requirements and Test Objectives

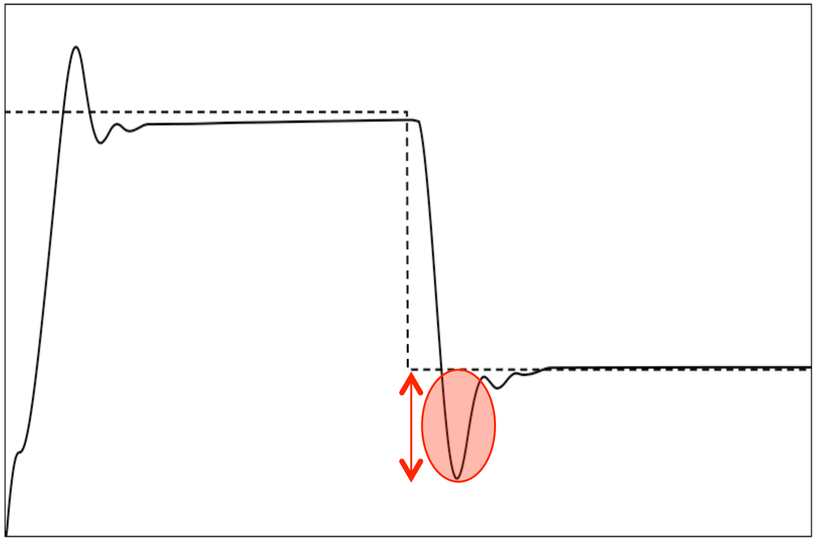


Test Strategy: A Search-Based Approach

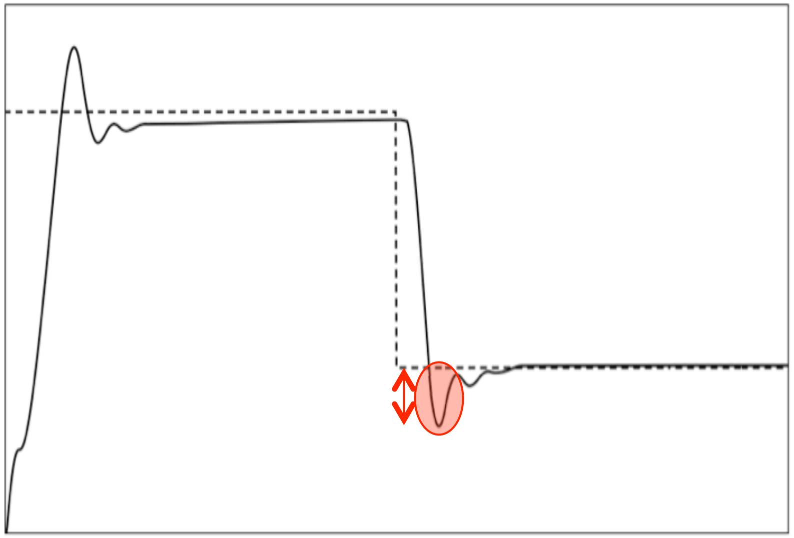


- Continuous behavior
- Controller's behavior can be complex
- Meta-heuristic search in (large) input space: Finding worst case inputs
- Possible because of automated oracle (feedback loop)
- Different worst cases for different requirements
- Worst cases may or may not violate requirements

Smoothness Objective Functions: $O_{Smoothness}$



Test Case A



Test Case B

$O_{Smoothness}$(Test Case A) **>** $O_{Smoothness}$(Test Case B)

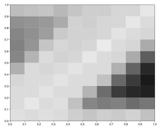
We want to find test scenarios which maximize $O_{Smoothness}$

Search Elements

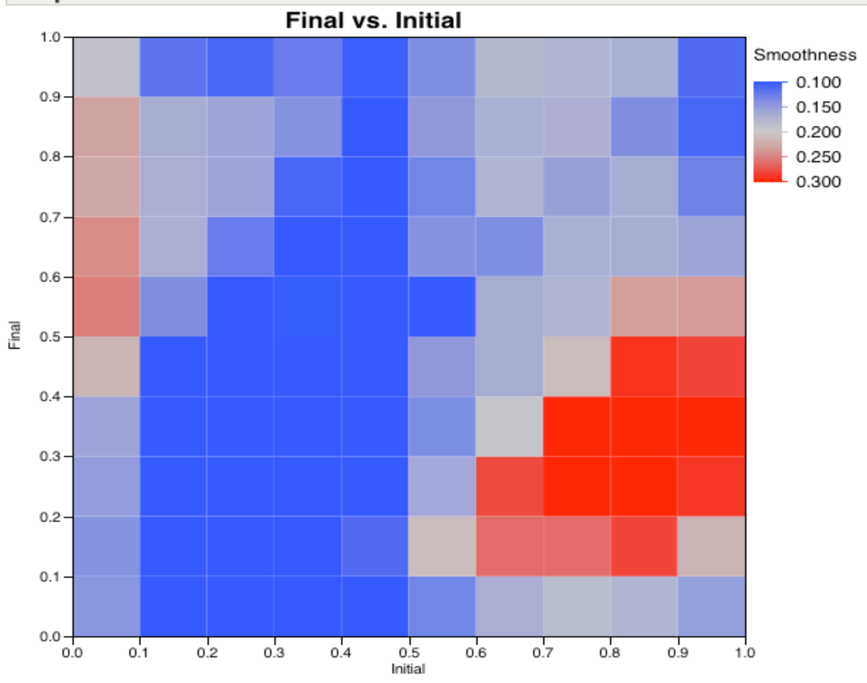
- **Search Space:**
  - Initial and desired values, configuration parameters
- **Search Technique:**
  - (1+1) EA, variants of hill climbing, GAs …
- **Search Objective:**
  - Objective/fitness function for each requirement
- **Evaluation of Solutions**
  - Simulation of Simulink model => fitness computation
- **Result:**
  - Worst case scenarios or values to the input variables that (are more likely to) break the requirement at MiL level
  - Stress test cases based on actual hardware (HiL)

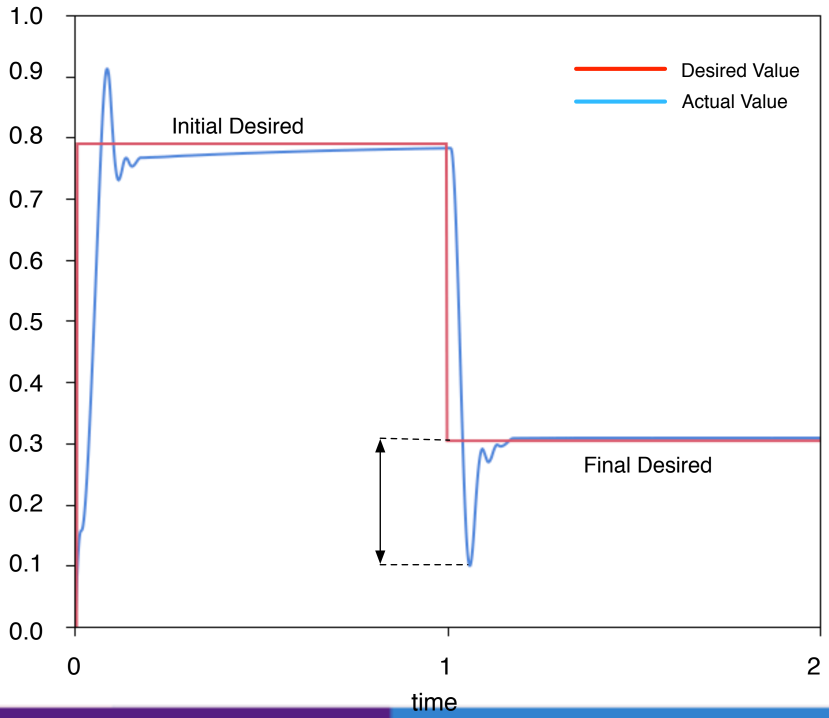
Solution Overview (Simplified Version)

Objective Functions based on Requirements + Controller-plant model → 1. Exploration → HeatMap Diagram ⇒ Domain Expert ⇒ List of Critical Regions → 2. Single-State Search → Worst-Case Scenarios



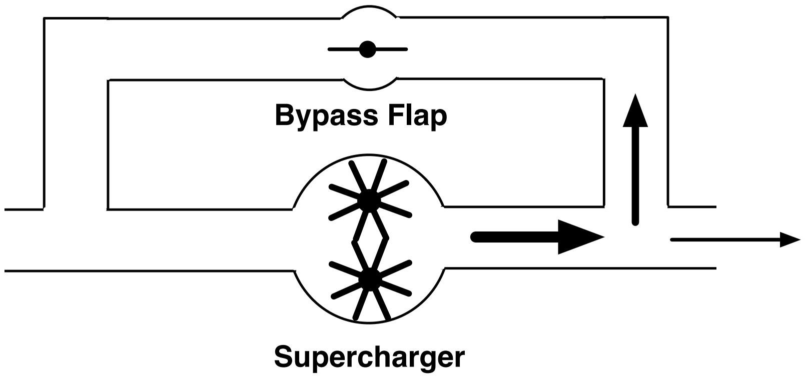
Graph Builder



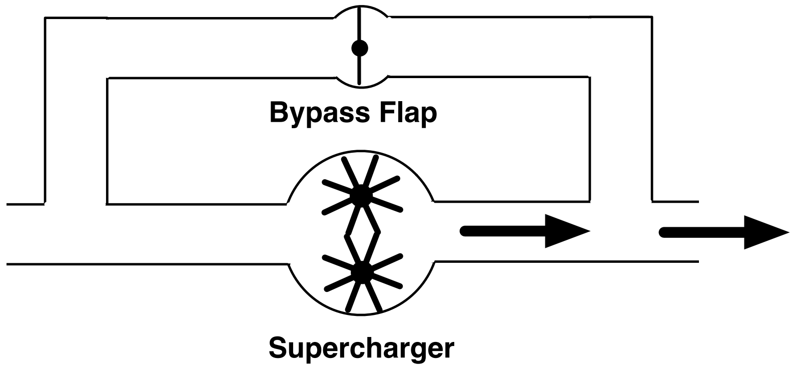


Automotive Example

- **Supercharger bypass flap controller**
  - ✓Flap position is bounded within [0..1]
  - ✓Implemented in MATLAB/Simulink
  - ✓34 sub-components decomposed into 6 abstraction levels
  - ✓The simulation time T=2 seconds

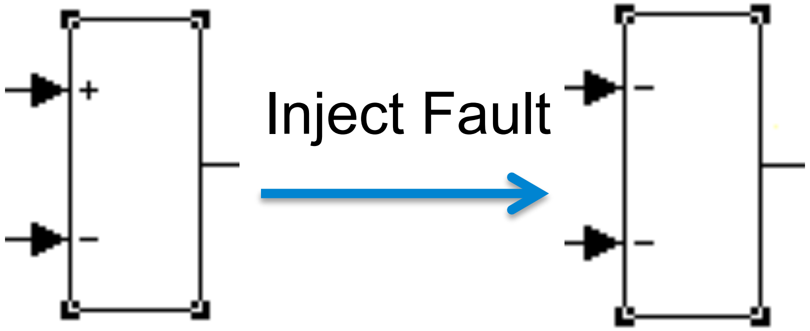


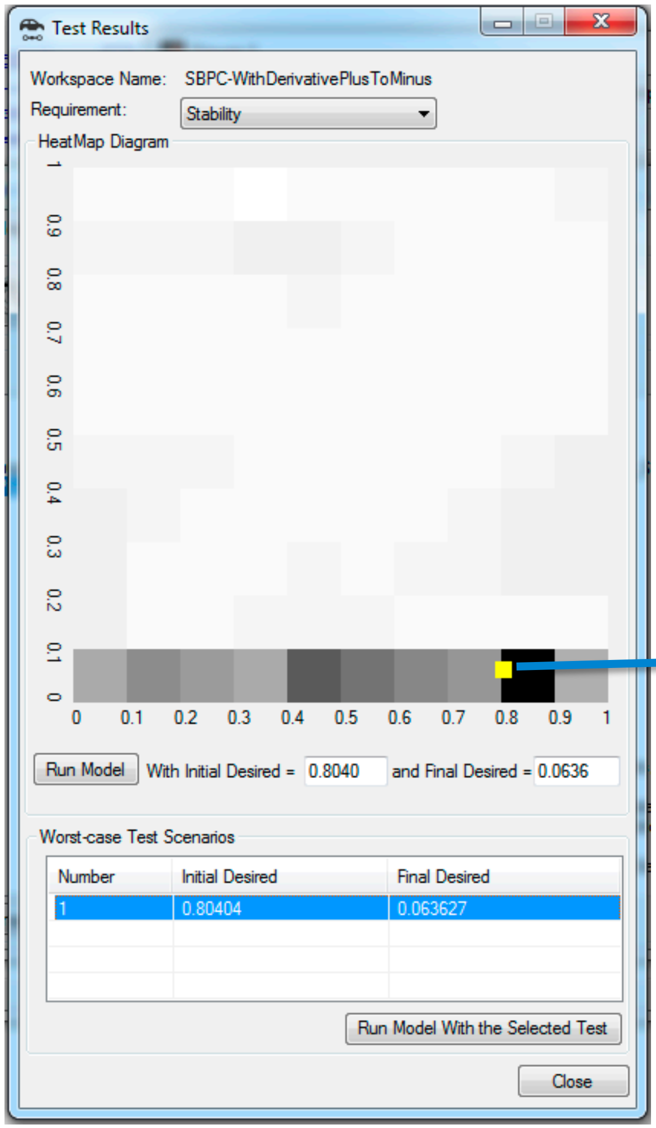
Flap position = 0 (open)



Flap position = 1 (closed)

Finding Seeded Faults





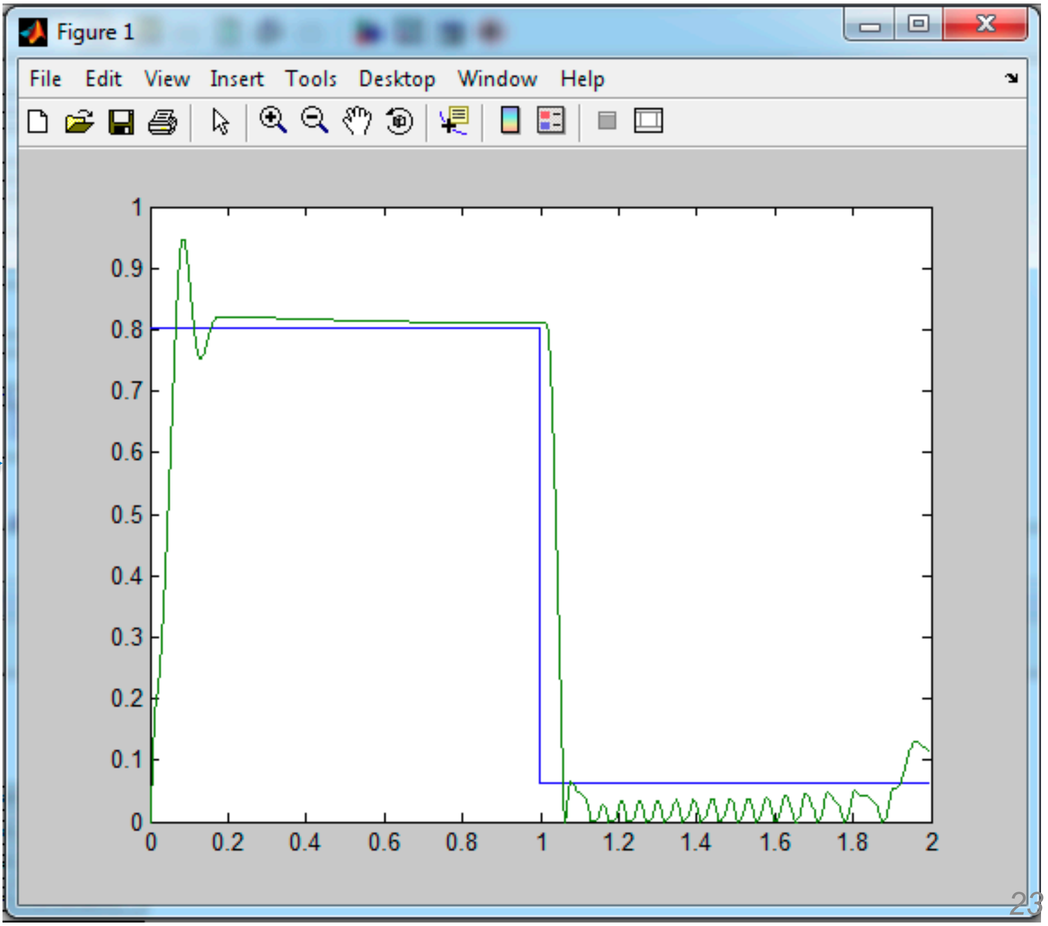
Workspace Name: SBPC-WithDerivativePlusToMinus

Requirement: Stability

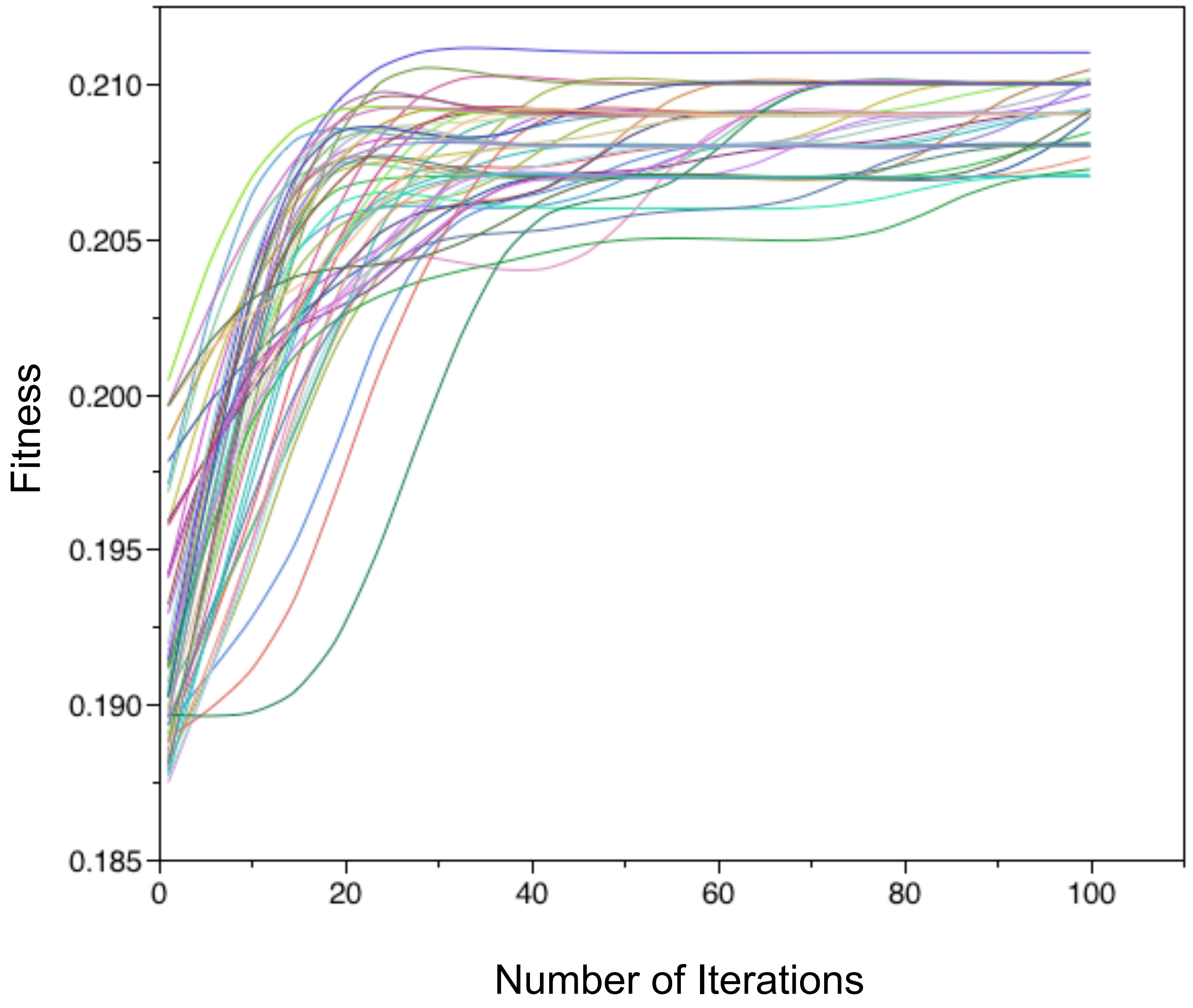
With Initial Desired = 0.8040 and Final Desired = 0.0636

Worst-case Test Scenarios

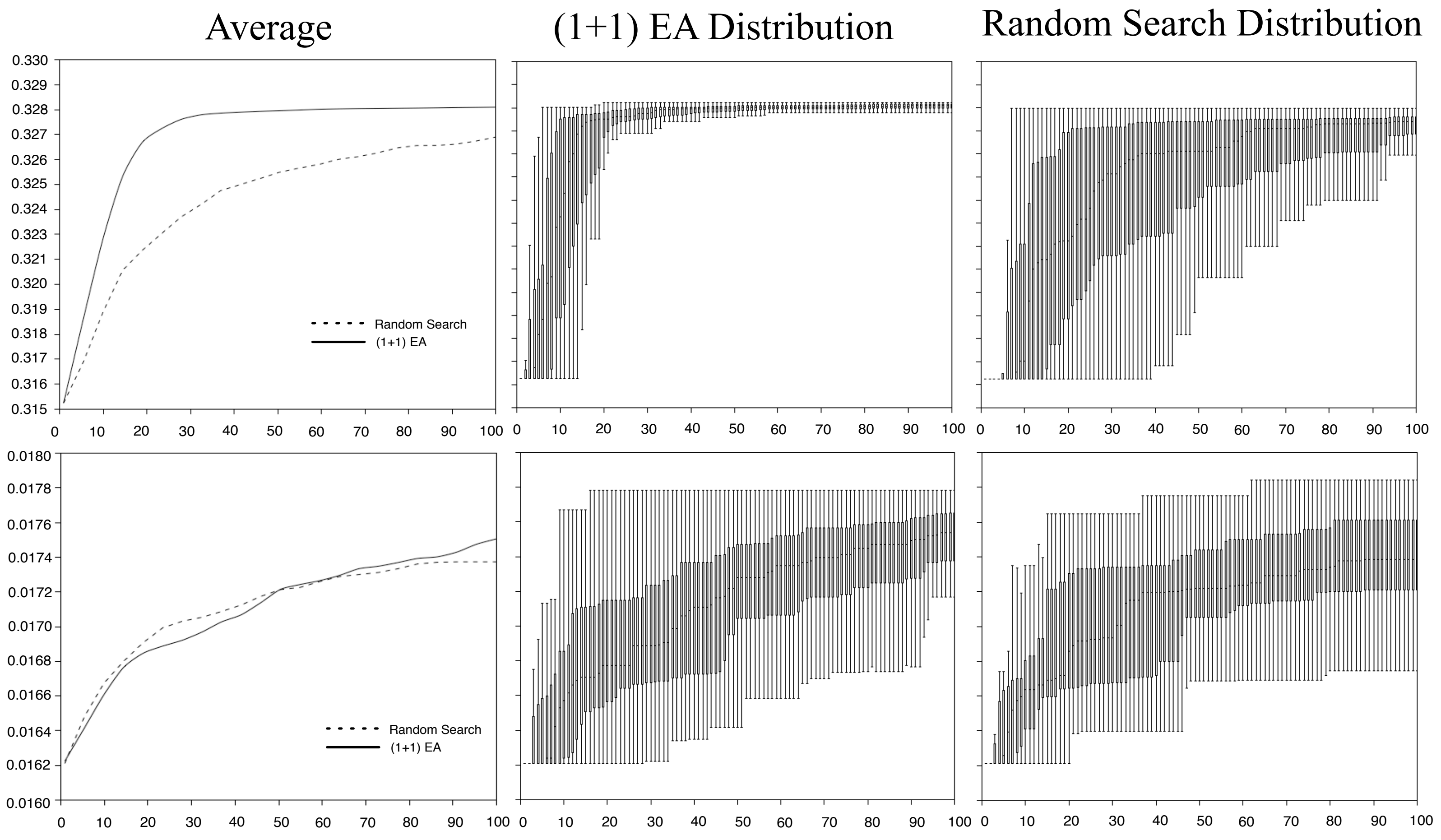
Number	Initial Desired	Final Desired
1	0.80404	0.063627



Analysis – Fitness increase over iterations



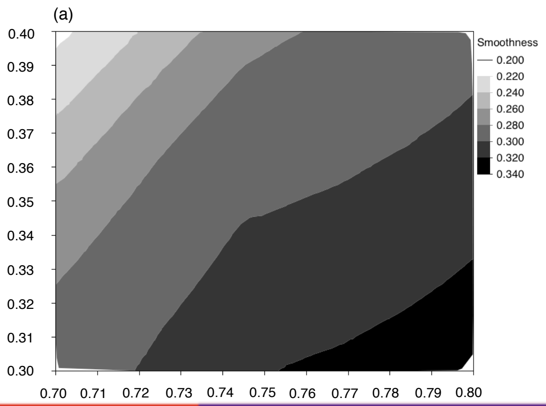
Analysis II – Search over different regions

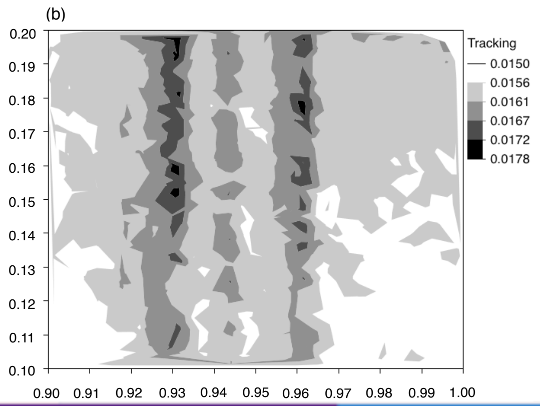


Number of Iterations

Conclusions

- We found much worse scenarios during MiL testing than our partner had found so far, and much worse than random search (baseline)
- These scenarios are also run at the HiL level, where testing is much more expensive: MiL results -> test selection for HiL
- But further research was needed:
  - Simulations are expensive
  - Configuration parameters
  - Dynamically adjust search algorithms in different subregions (exploratory <-> exploitative)

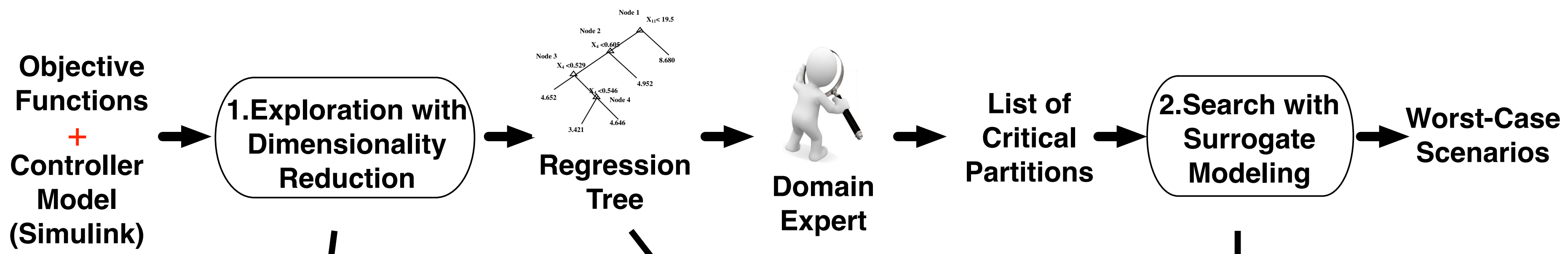




Testing in the Configuration Space

- MIL testing for all feasible configurations
- The search space is much larger
- The search is much slower (Simulations of Simulink models are expensive)
- Results are harder to visualize
- Not all configuration parameters matter for all objective functions

Modified Process and Technology

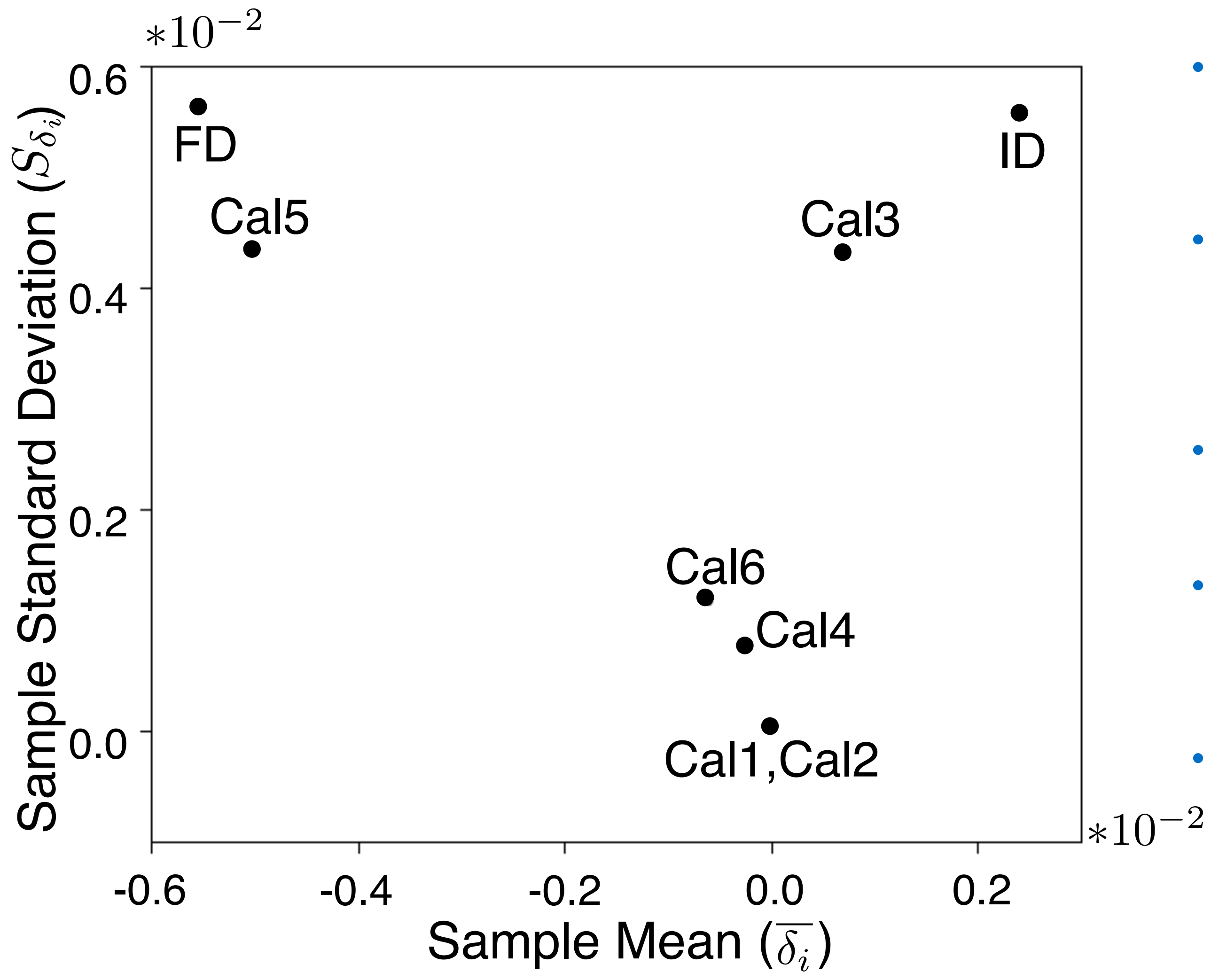


Dimensionality reduction to identify the significant variables (Elementary Effect Analysis)

Visualization of the 8-dimension space using regression trees

Surrogate modeling to predict the objective function and speed up the search (Machine learning)

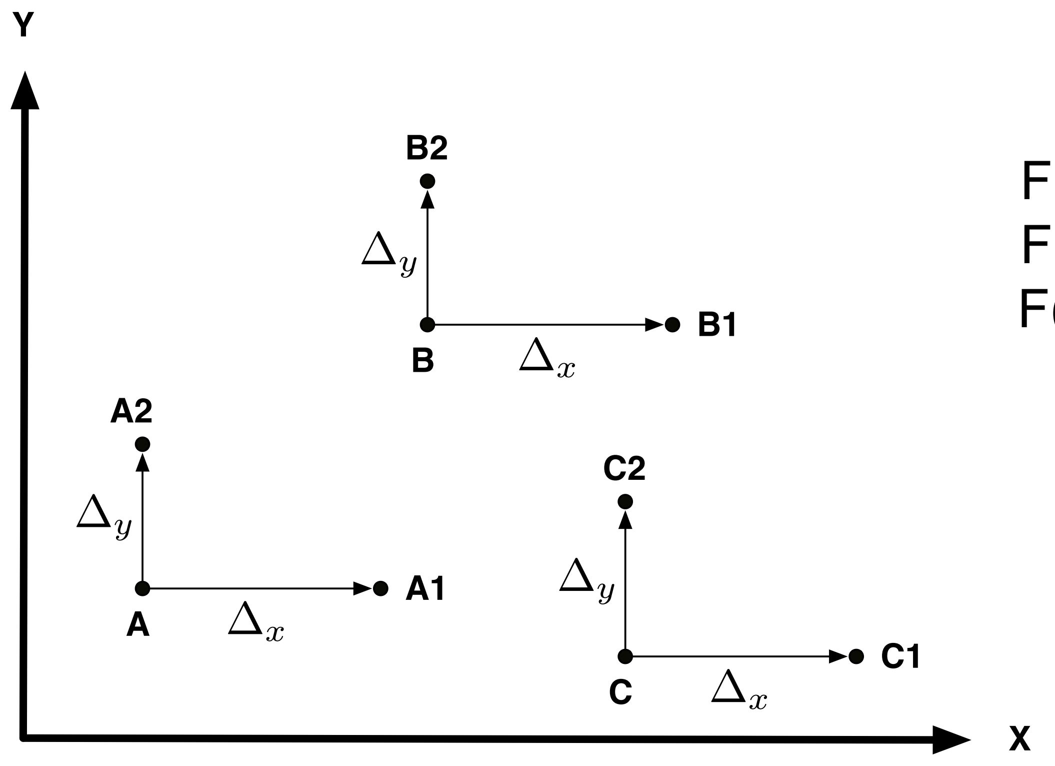
Dimensionality Reduction



- Sensitivity Analysis: Elementary Effect Analysis (EEA)
- Identify non-influential inputs in computationally costly mathematical models
- Requires less data points than other techniques
- Observations are simulations generated during the Exploration step
- Compute sample mean and standard deviation for each dimension of the distribution of elementary effects

Elementary Effects Analysis Method

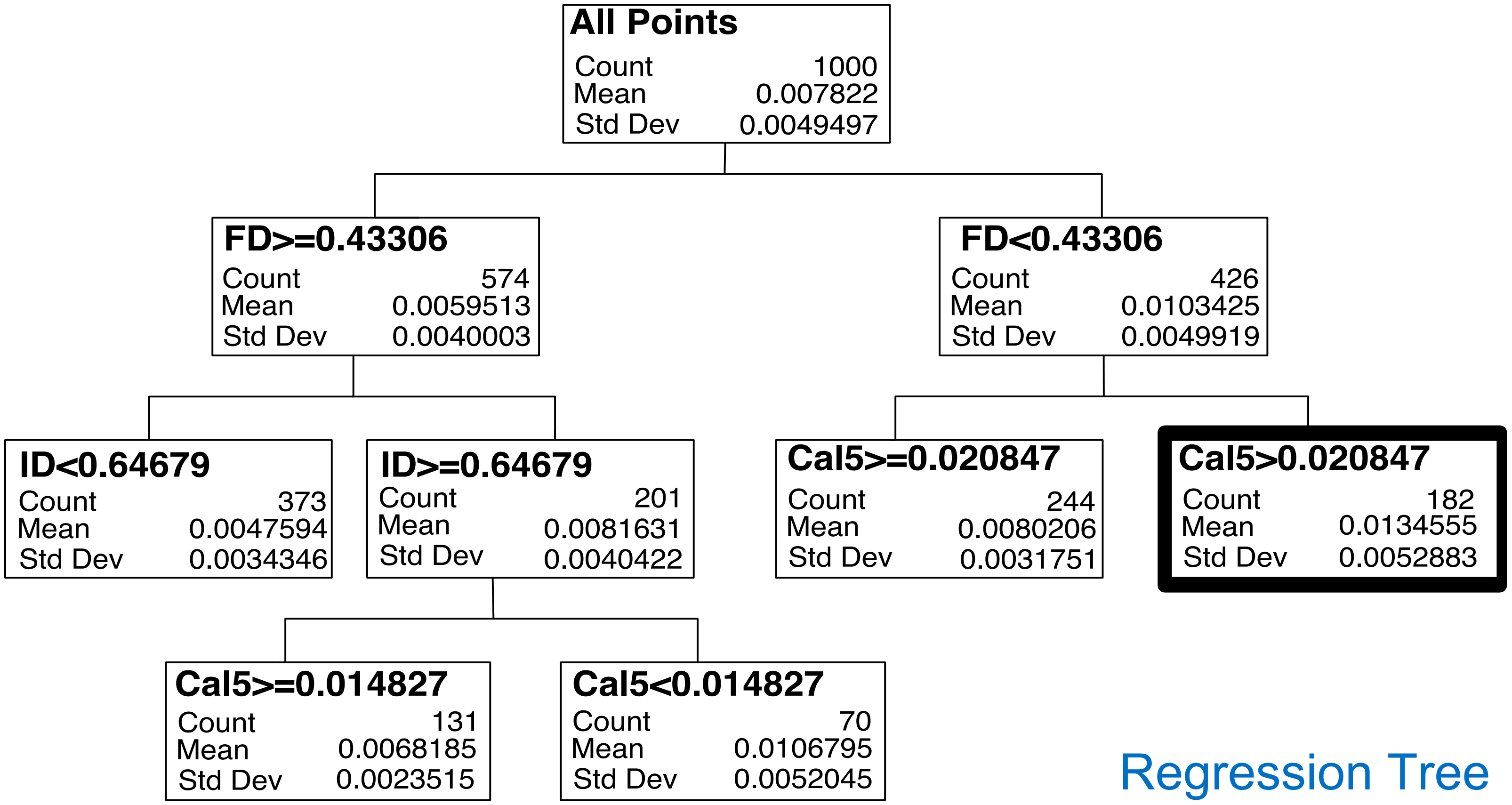
✓ Imagine function F with 2 inputs, x and y:



Elementary Effects

for X	for Y
F(A1)-F(A)	F(A2)-F(A)
F(B1)-F(B)	F(B2)-F(B)
F(C1)-F(C)	F(C2)-F(C)
…	…

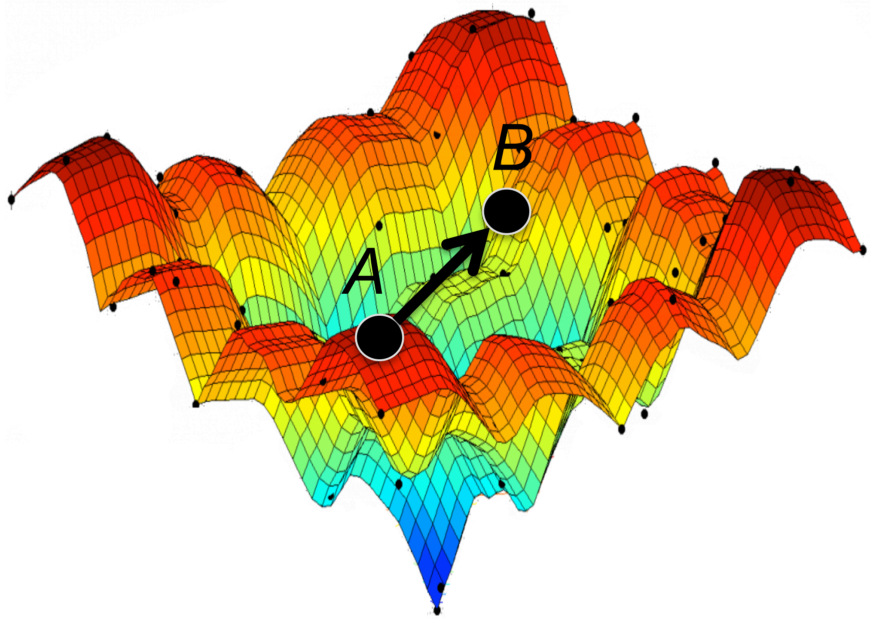
Visualization in Inputs & Configuration Space



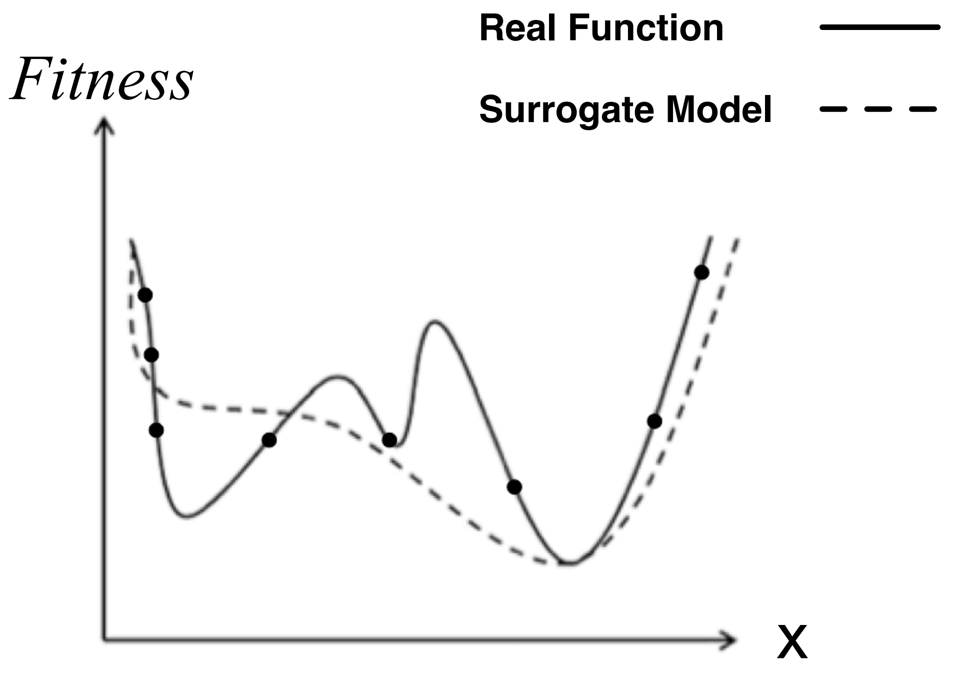
Regression Tree

Surrogate Modeling (1)

- Goal: To predict the value of the objective functions within a critical partition, given a number of observations, and use that to avoid as many simulations as possible and speed up the search



Surrogate Modeling (2)



- Any supervised learning or statistical technique providing fitness predictions with confidence intervals

1. Predict higher fitness with high confidence: Move to new position, no simulation
2. Predict lower fitness with high confidence: Do not move to new position, no simulation
3. Low confidence in prediction: Simulation

Experiments Results (RQ1)

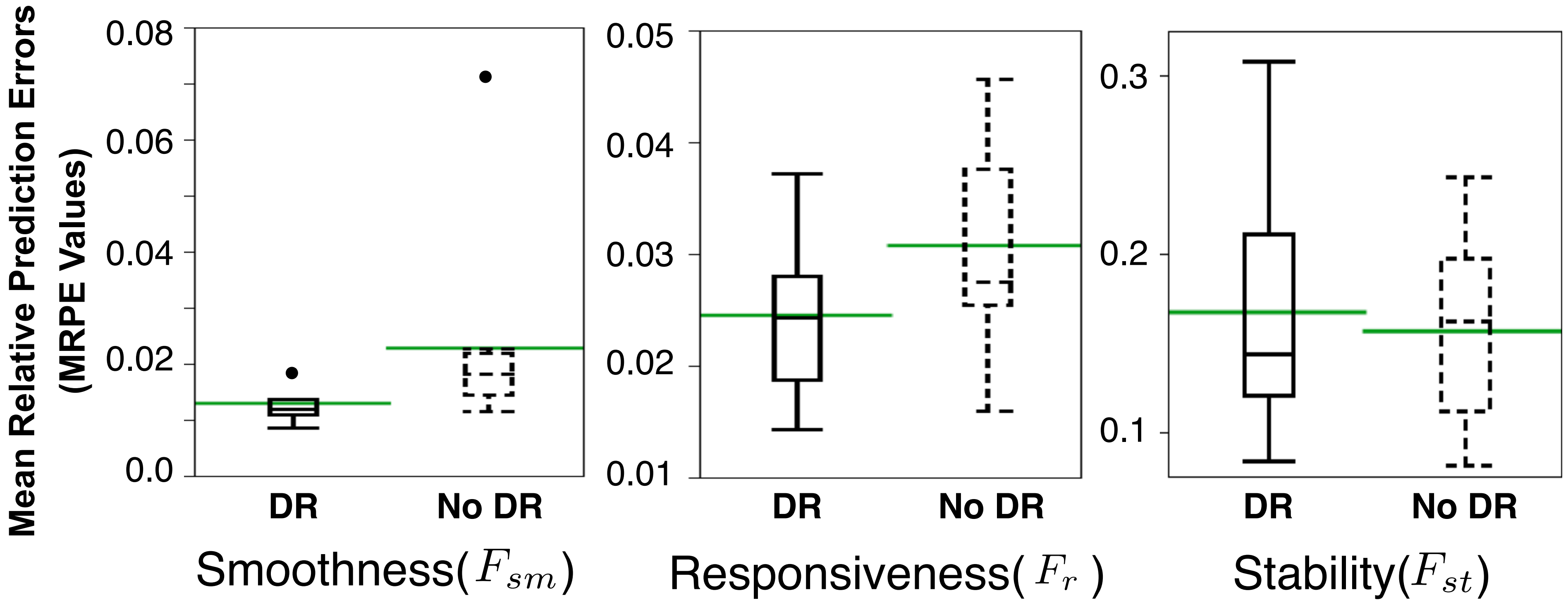
- ✓ The best regression technique to build Surrogate models for all of our three objective functions is Polynomial Regression with n = 3
  - ✓ Other supervised learning techniques, such as SVM

Mean of R^2/MRPE values for different surrogate modeling techniques

	LR	ER	PR(n=2)	PR(n=3)
	R^2/**MRPE**	R^2/**MRPE**	R^2/**MRPE**	R^2/**MRPE**
F_{sm}	0.66/**0.0526**	0.44/**0.0791**	0.95/**0.0203**	0.98/**0.0129**
F_r	0.78/**0.0295**	0.49/**1.2281**	0.85/**0.0247**	0.85/**0.0245**
F_{st}	0.26/**0.2043**	0.22/**1.2519**	0.46/**0.1755**	0.54/**0.1671**

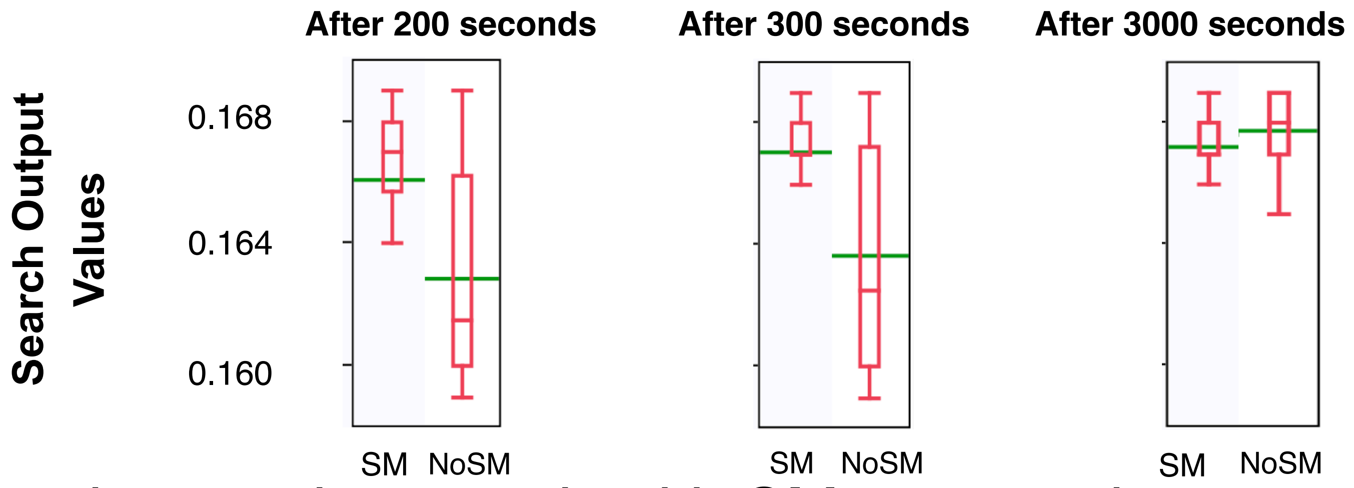
Experiments Results (RQ2)

✓ Dimensionality reduction helps generate better surrogate models for Smoothness and Responsiveness requirements

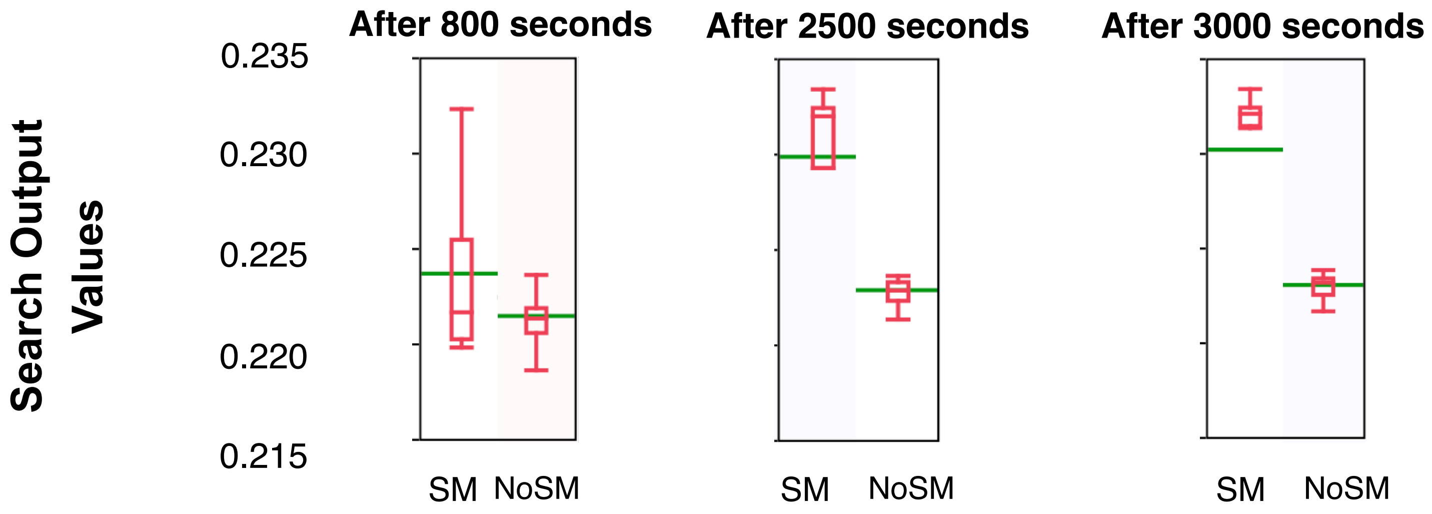


Experiments Results (RQ3)

✓ For responsiveness, the search with SM was 8 times faster



✓ For smoothness, the search with SM was much more effective

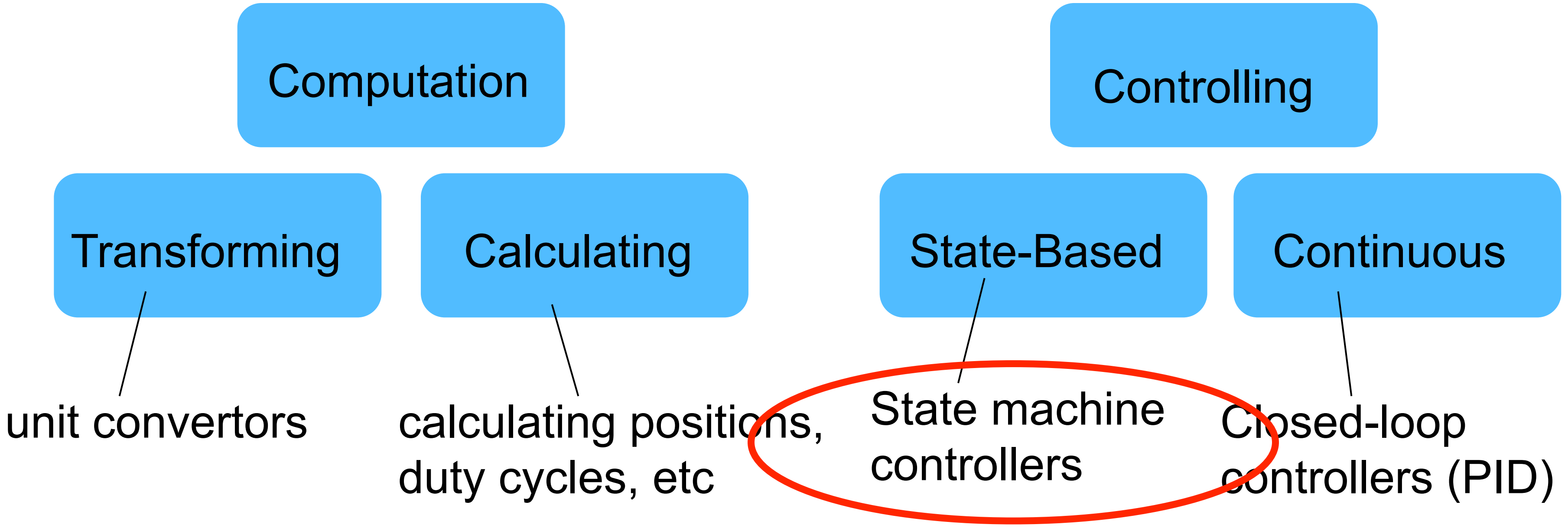


Experiments Results (RQ4)

✓ Our approach is able to identify critical violations of the controller requirements that had neither been found by our earlier work nor by manual testing.

	MiL-Testing different configurations	MiL-Testing fixed configurations	Manual MiL-Testing
Stability	2.2% deviation	-	-
Smoothness	24% over/undershoot	20% over/undershoot	5% over/undershoot
Responsiveness	170 ms response time	80 ms response time	50 ms response time

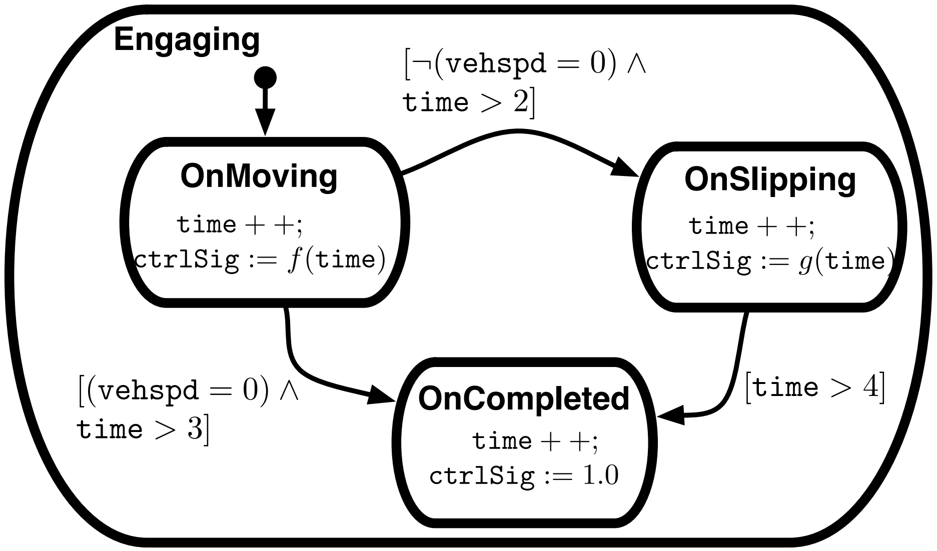
A Taxonomy of Automotive Functions

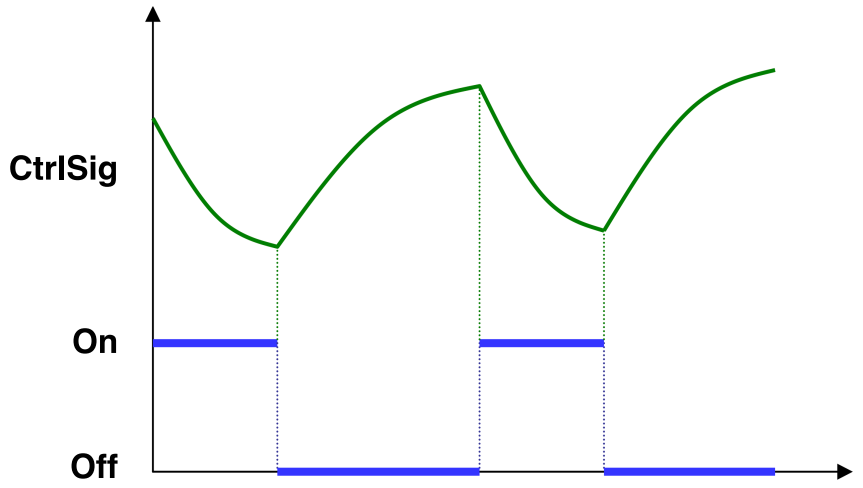

Computation
Controlling
Transforming
Calculating
State-Based
Continuous
unit convertors
calculating positions, duty cycles, etc
State machine controllers
Closed-loop controllers (PID)

Different testing strategies are required for different types of functions

Differences with Close-Loop Controllers

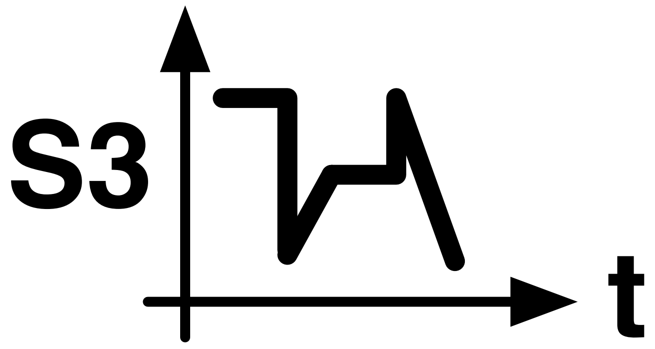
- Mixed discrete-continuous behavior: Simulink stateflows
- Much quicker simulation time
- No feedback loop -> no automated oracle
- The main testing cost is the manual analysis of output signals
- Goal: Minimize test suites
- Challenge: Test selection
- Entirely different approach to testing

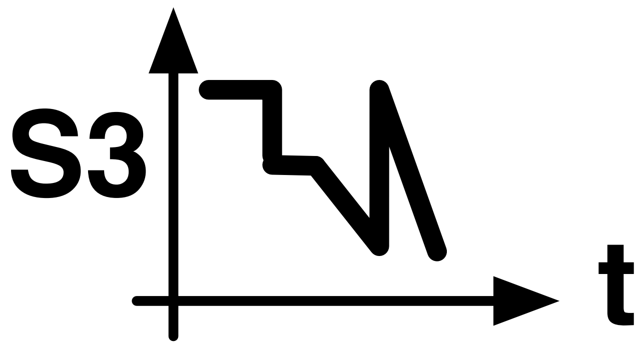

Engaging
$[\neg(\texttt{vehspd} = 0) \wedge \texttt{time} > 2]$
OnMoving
$\texttt{time}++;$
$\texttt{ctrlSig} := f(\texttt{time})$
OnSlipping
$\texttt{time}++;$
$\texttt{ctrlSig} := g(\texttt{time})$
$[(\texttt{vehspd} = 0) \wedge \texttt{time} > 3]$
OnCompleted
$\texttt{time}++;$
$\texttt{ctrlSig} := 1.0$
$[\texttt{time} > 4]$



Selection Strategies Based on Search

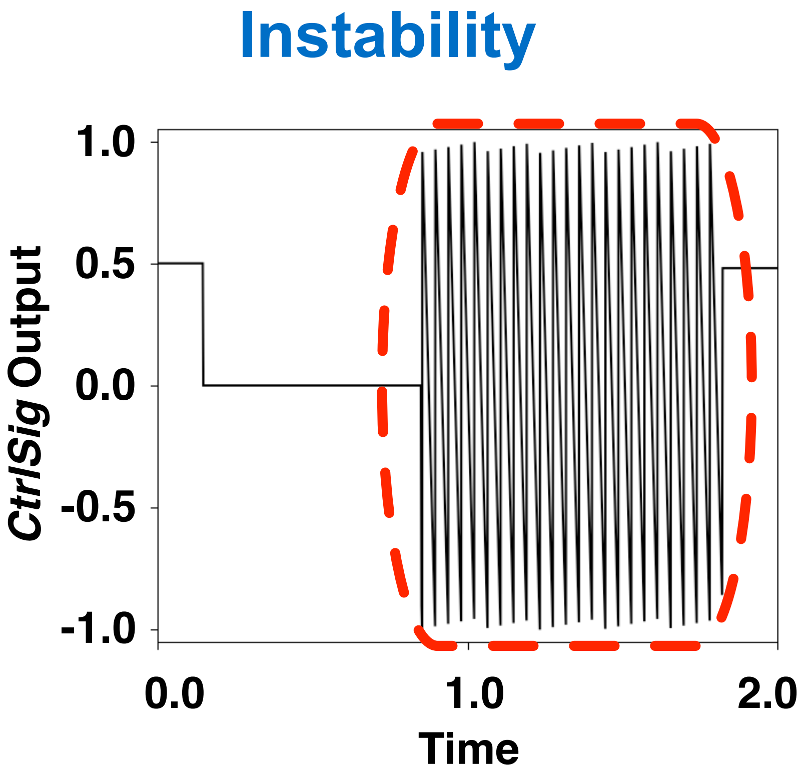
- Input diversity
- White-box Structural Coverage
  - State Coverage
  - Transition Coverage
- Output Diversity
- Failure-Based Selection Criteria
  - Domain specific failure patterns
  - Output Stability
  - Output Continuity

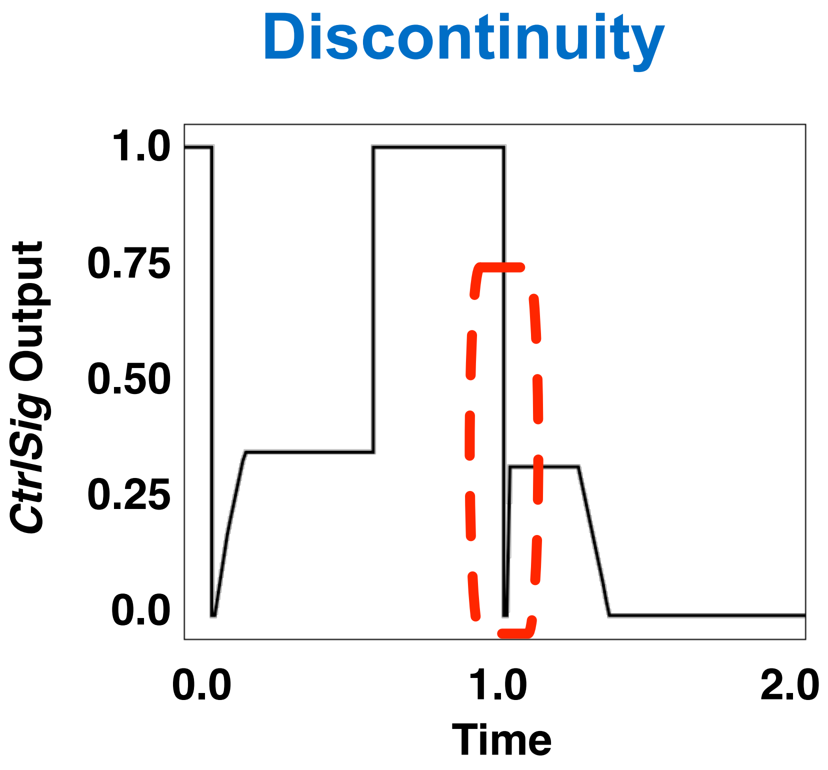




Failure-based Test Generation

- Maximizing the likelihood of presence of specific failure patterns in output signals
- Failure patterns elicited from engineers





Summary of Results

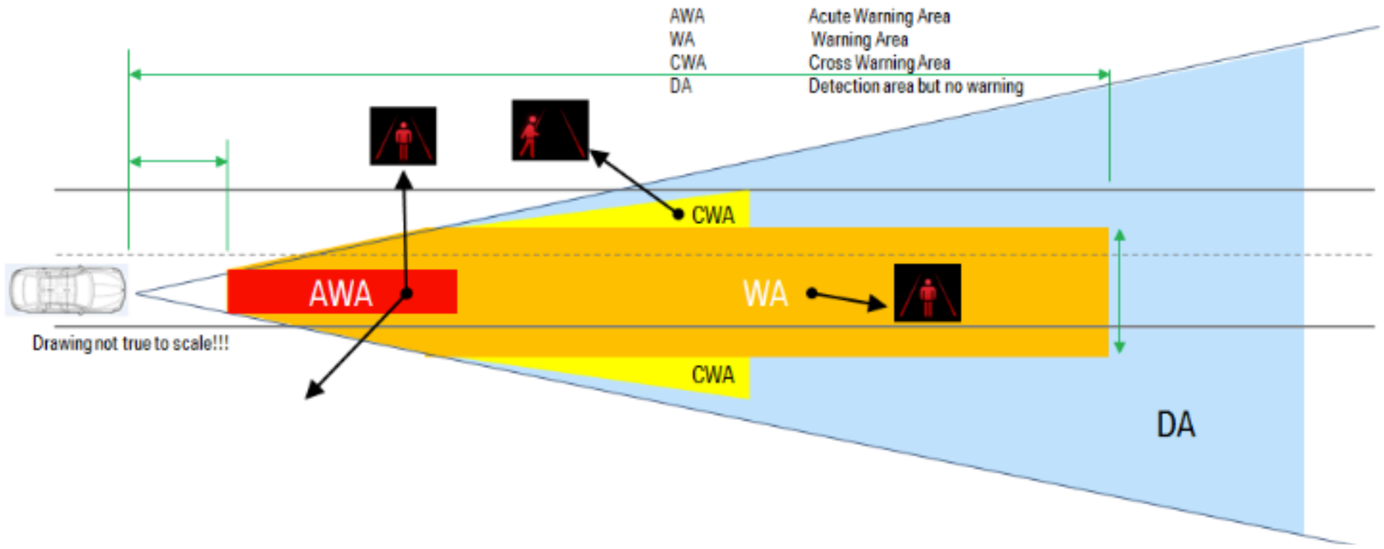
- The test cases resulting from state/transition coverage algorithms cover the faulty parts of the models
- However, they fail to generate output signals that are sufficiently distinct from the oracle signal, hence yielding a low fault revealing rate
- Output-based algorithms are more effective

Automated Testing of Vision Systems Through Simulation

- *With Raja Ben Abdessalem, Shiva Nejati*
- *In collaboration with IEE, Luxembourg*

Night Vision (NiVi) System

- The NiVi system is a camera-based assistance system providing improved vision at night

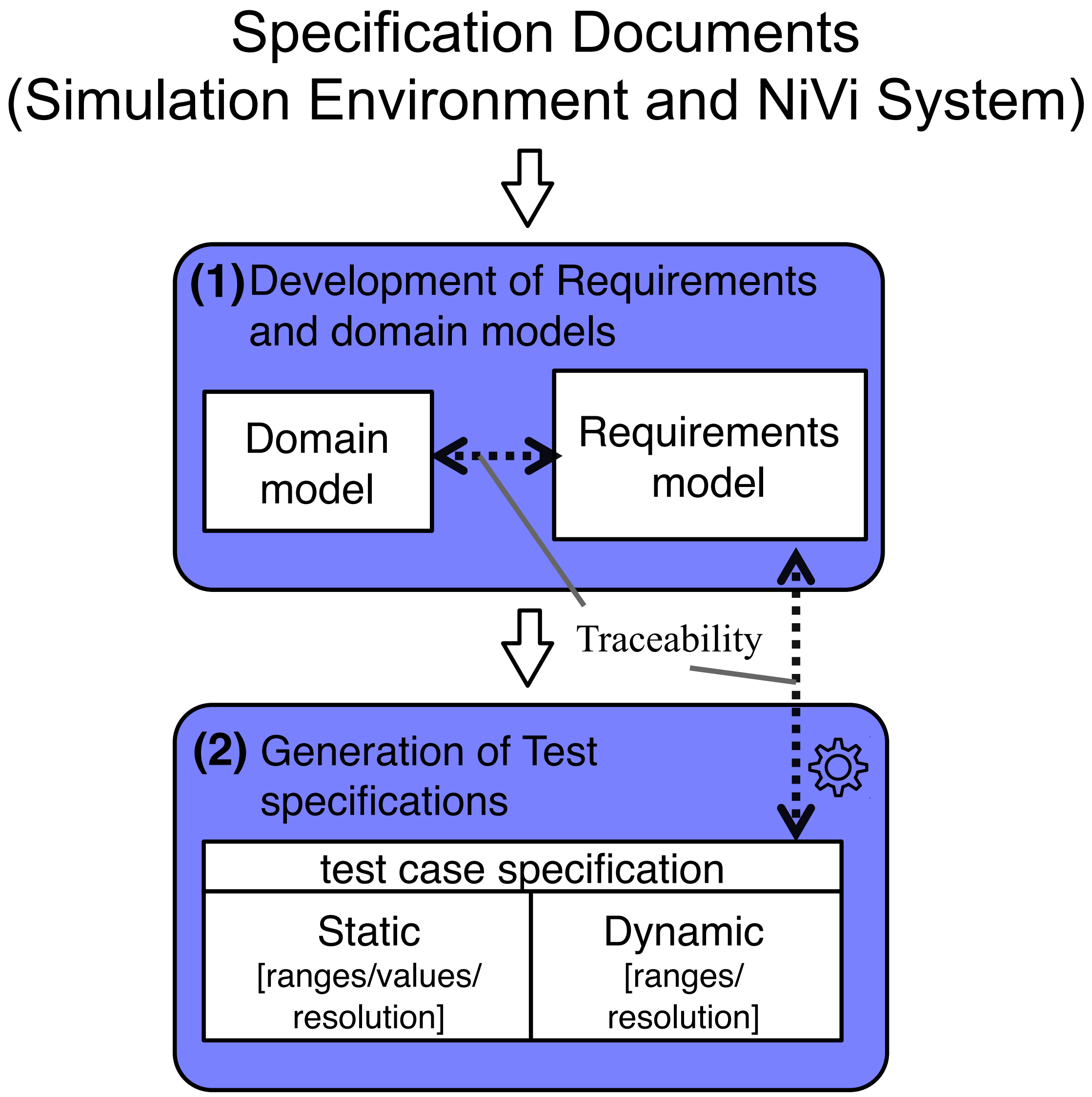


Testing Vision Systems

- Testing vision systems requires complex and comprehensive simulation environments
  - Static objects: roads, weather, etc.
  - Dynamic objects: cars, humans, animals, etc.

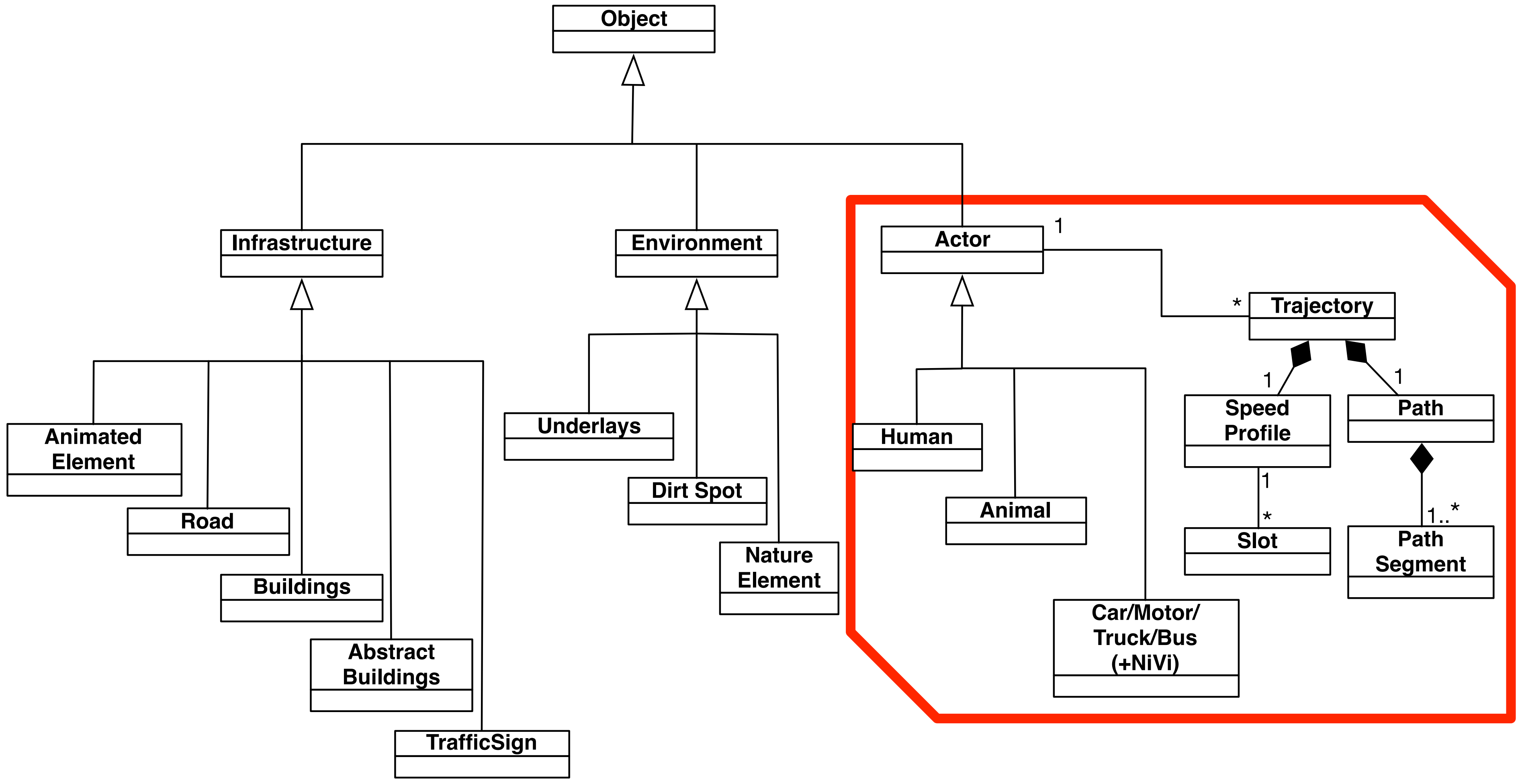
- A simulation environment captures the behaviour of dynamic objects as well as constraints and relationships between dynamic and static objects

Overview

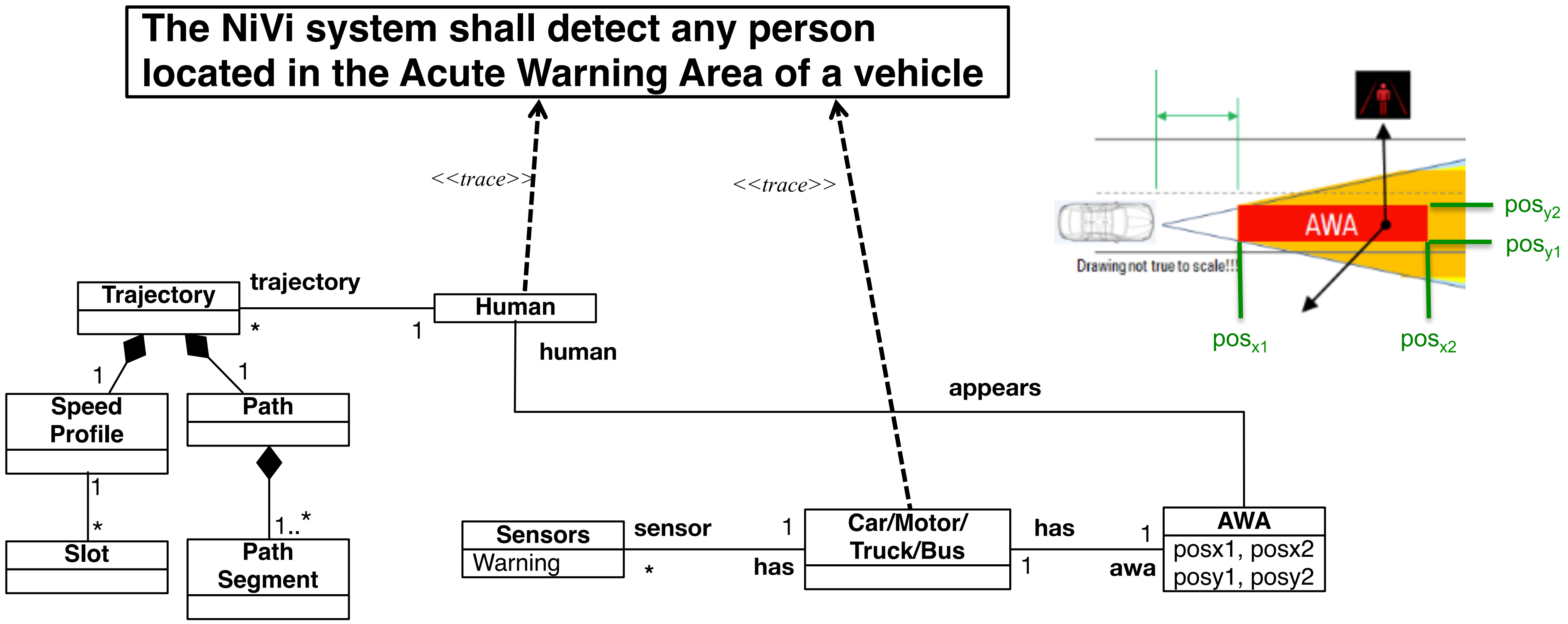


Specification Documents
(Simulation Environment and NiVi System)
(1) Development of Requirements and domain models
Domain model
Requirements model
Traceability
(2) Generation of Test specifications
test case specification
Static [ranges/values/ resolution]
Dynamic [ranges/ resolution]

NiVi and Environment Domain Model

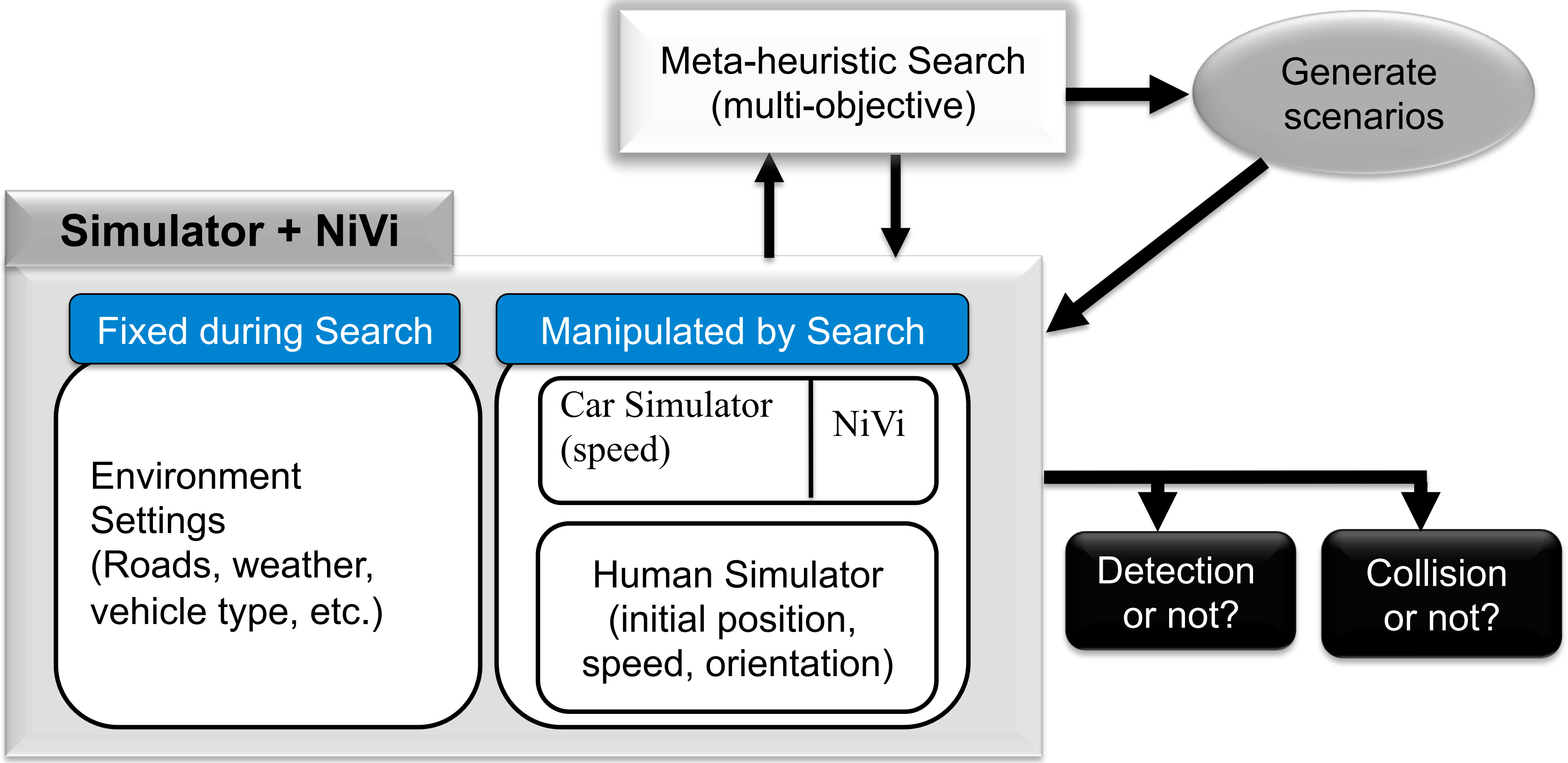


Requirements Model



$$\exists t.\ car.awa.pos_{x1}[t] < car.awa.human.trajectory.pos_x[t] < car.awa.pos_{x2}[t] \wedge$$
$$car.awa.pos_{y1}[t] < car.awa.human.trajectory.pos_y[t] < car.awa.pos_{y2}[t]$$
$$\Rightarrow car.sensor.warning == true$$

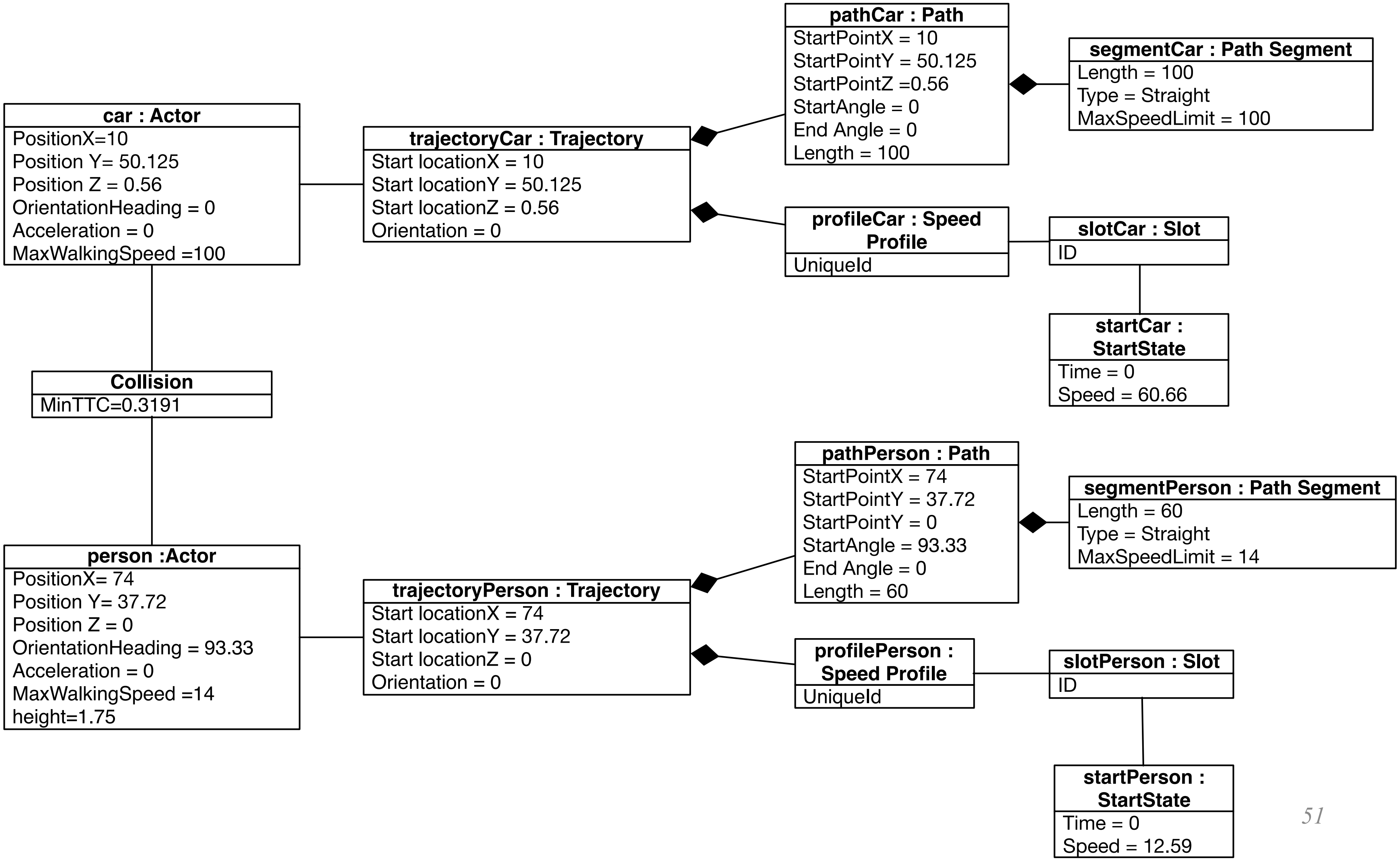
MiL Testing via Search


Meta-heuristic Search (multi-objective)
Generate scenarios
Simulator + NiVi
Fixed during Search
Manipulated by Search
Environment Settings (Roads, weather, vehicle type, etc.)
Car Simulator (speed)
NiVi
Human Simulator (initial position, speed, orientation)
Detection or not?
Collision or not?

Test Case Specification: Static (combinatorial)

	Type of Road	Type of vehicle	Type of actor
Situation 1	Straight	Car	Male
Situation 2	Straight	Car	Child
Situation 3	Straight	Car	Cow
Situation 4	Straight	Truck	Male
Situation 5	Straight	Truck	Child
Situation 6	Straight	Truck	Cow
Situation 7	Curved	Car	Male
Situation 8	Curved	Car	Child
Situation 9	Curved	Car	Cow
Situation 10	Curved	Truck	Male
Situation 11	Curved	Truck	Child
Situation 12	Curved	Track	Cow
Situation 13	Ramp	Car	Male
Situation 14	Ramp	Car	Child
Situation 15	Ramp	Car	Cow
Situation 16	Ramp	Truck	Male
Situation 17	Ramp	Truck	Child
Situation 18	Ramp	Truck	Cow
Situation 19 Situation 20	Straight	Car+ Cars in parking Car + buildings	Male

Test Case Specification: Dynamic



Multi-Objective Search

- Objective functions:
  - Distance to Car “D(P/Car)”, Time To Collision “TTC”, and Distance to AWA “D(P/AWA)”
- The goal is to identify scenarios that minimize our three objectives at the same times in different environment situations
- Identify automatically most important risky environment situations
  - e.g., ramped roads, curved roads, blocked field of views, and animal as the object to detect
- Challenge: Simulation time => surrogate modeling?
- Found many failures in NiVi

Minimizing CPU Shortage Risks During Integration

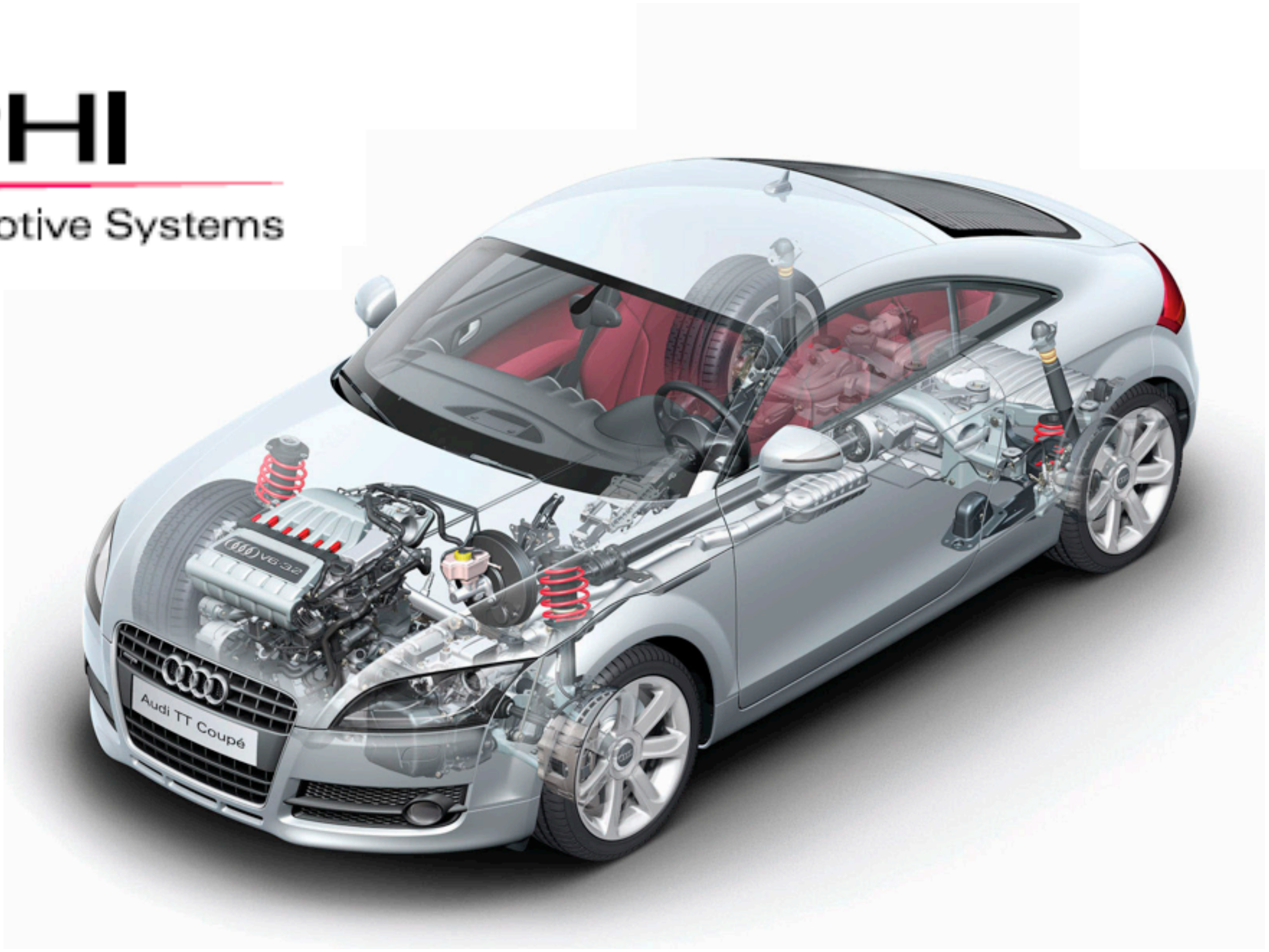
References:

- *S. Nejati et al., “Minimizing CPU Time Shortage Risks in Integrated Embedded Software”, in 28th IEEE/ACM International Conference on Automated Software Engineering (ASE 2013), 2013*
- *S. Nejati, L. Briand, “Identifying Optimal Trade-Offs between CPU Time Usage and Temporal Constraints Using Search”, ACM International Symposium on Software Testing and Analysis (ISSTA 2014), 2014*

Automotive: Distributed Development

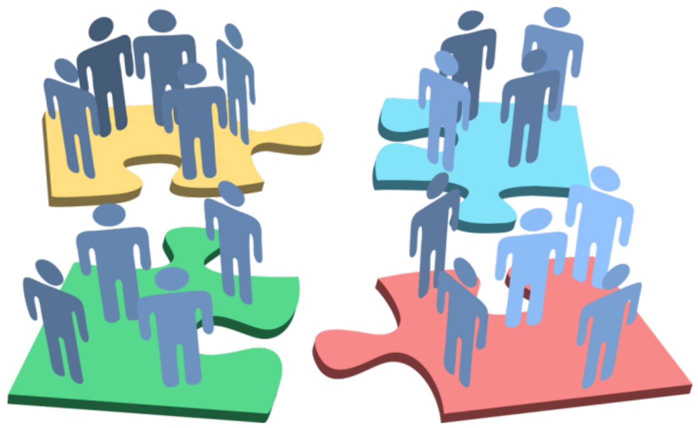
SNT
securityandtrust.lu

DELPHI
Automotive Systems



Software Integration

Stakeholders



Car Makers

- Develop software optimized for their specific hardware
- Provide part suppliers with runnables (exe)

Part Suppliers

- Integrate car makers software with their own platform
- Deploy final software on ECUs and send them to car makers

Different Objectives

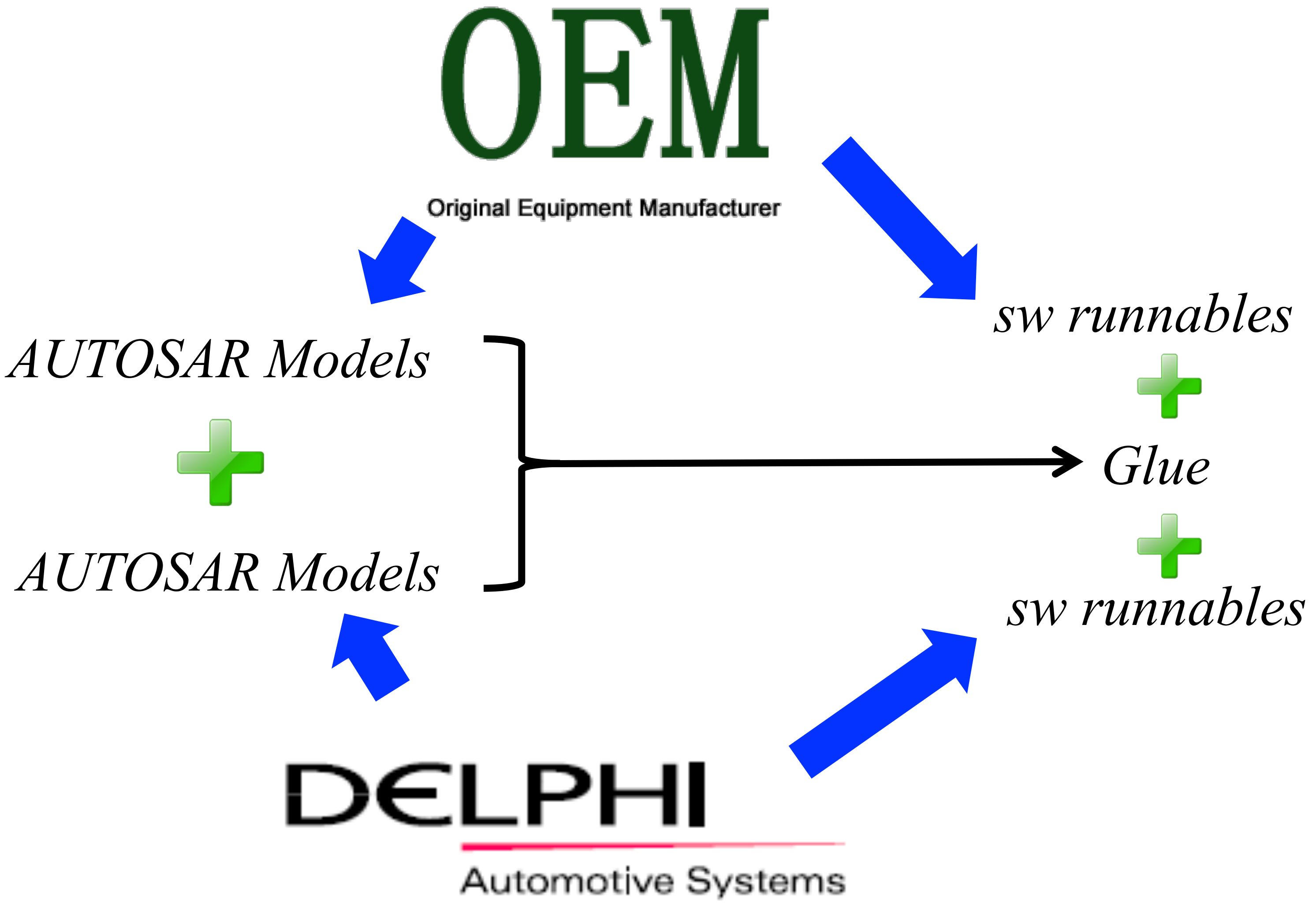
Car Makers

- Objective: Effective execution and synchronization of runnables
- Some runnables should execute simultaneously or in a certain order

Part Suppliers

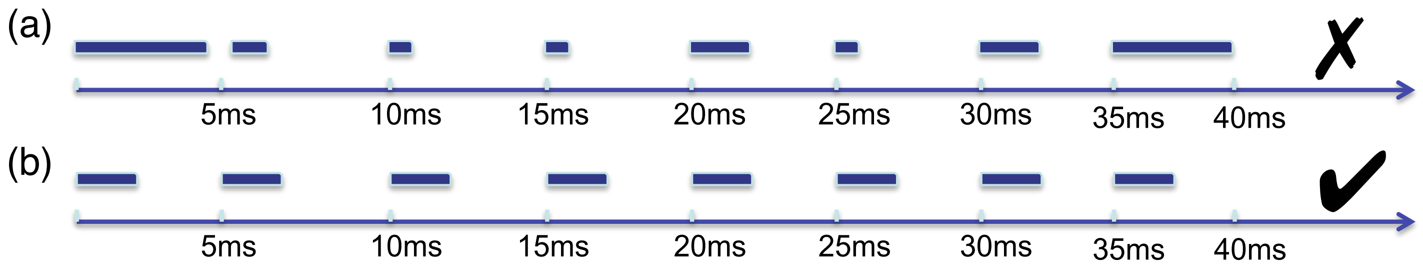
- Objective: Effective usage of CPU time
- Max CPU time used by all the runnables should remain as low as possible over time

An overview of an integration process in the automotive domain

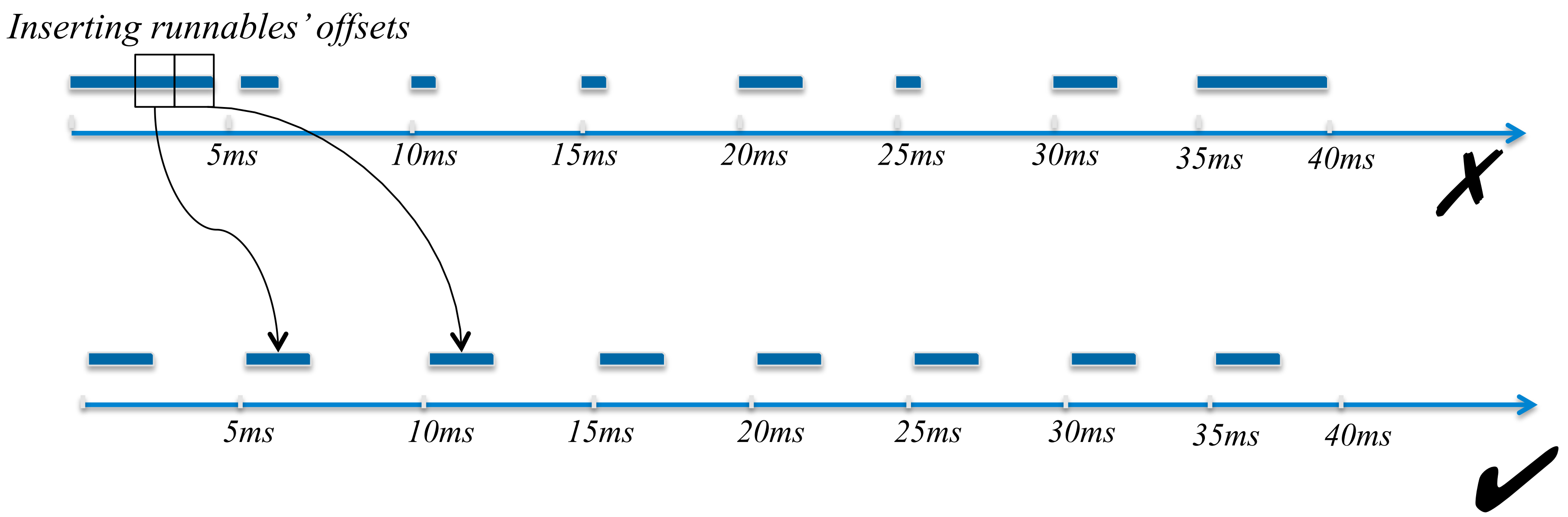


CPU time shortage

- Static cyclic scheduling: predictable, analyzable
- Challenge
  - Many OS tasks and their many runnables run within a limited available CPU time
    - The execution time of the runnables may exceed their time slot
- Goal
  - Reducing the maximum CPU time used per time slot to be able to
    - Minimize the hardware cost
    - Reduce the probability of overloading the CPU in practice
    - Enable addition of new functions incrementally



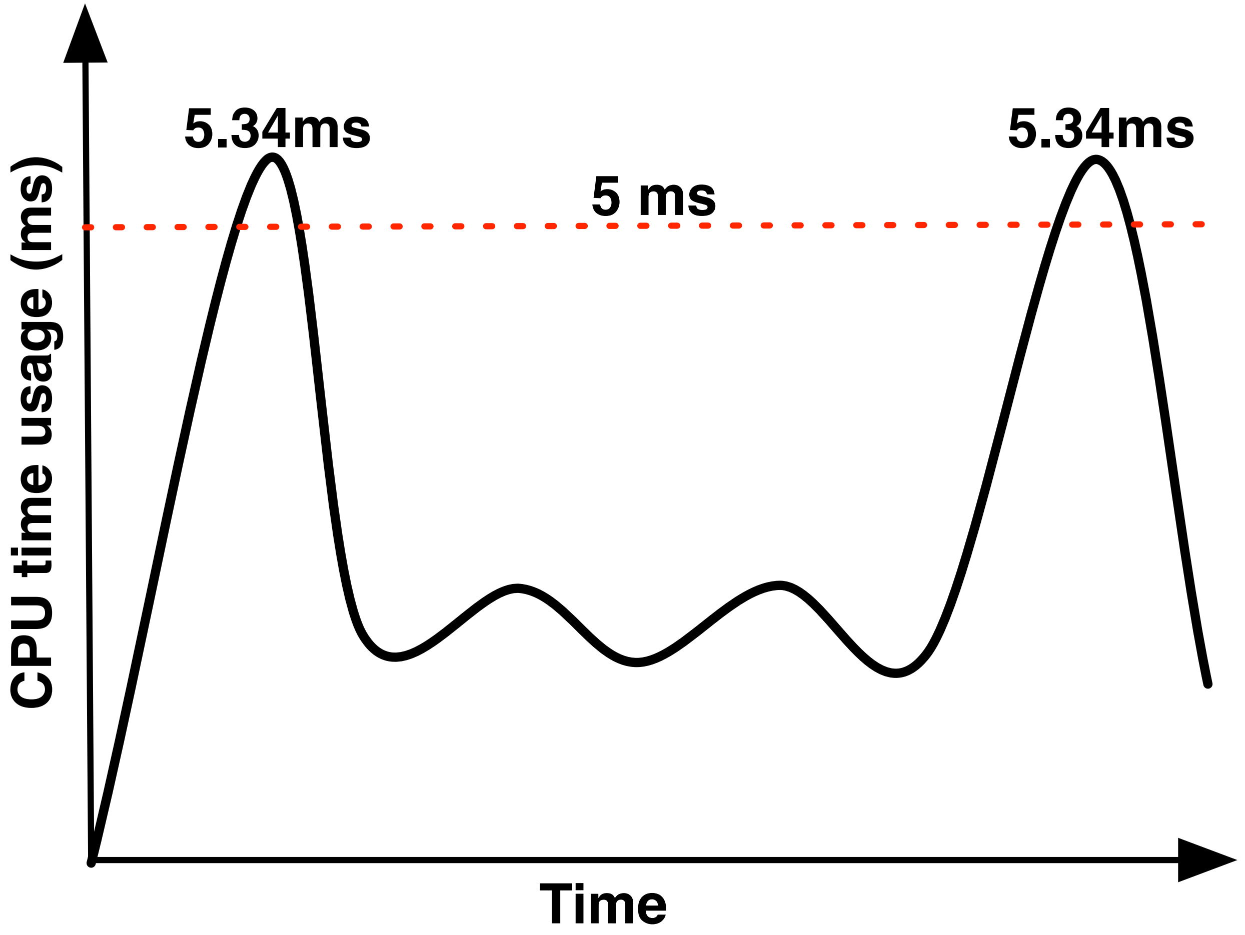
Using runnable offsets (delay times)



Offsets have to be chosen such that

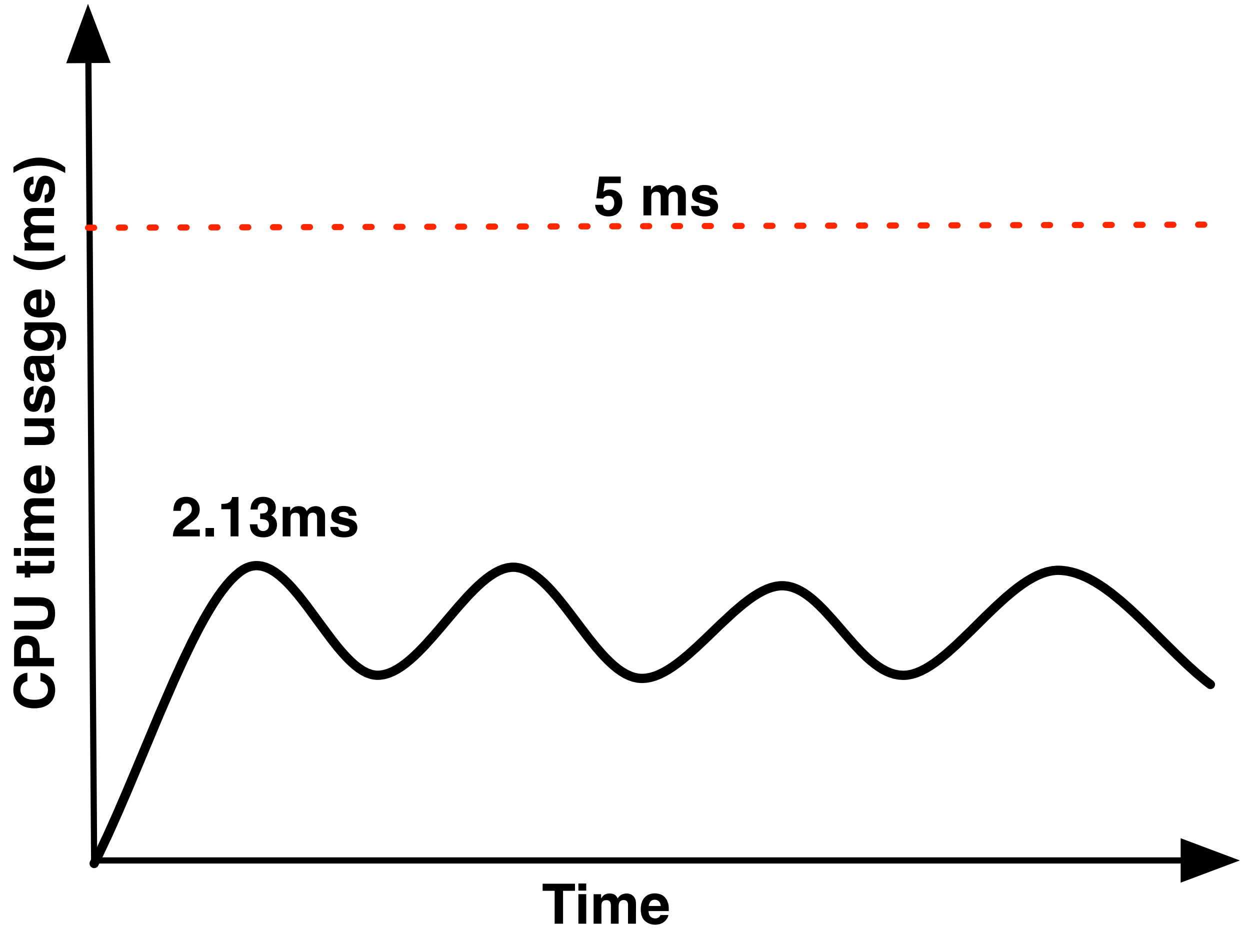
- the maximum CPU usage per time slot is minimized, and further,
  - the runnables respect their period
  - the runnables respect their time slot
  - the runnables satisfy their synchronization constraints

Without optimization



CPU time usage exceeds the size of the slot (5ms)

With Optimization



CPU time usage always remains less than 2.13ms, so more than half of each slot is guaranteed to be free

Single-objective Search algorithms

Hill Climbing and Tabu Search and their variations

Solution Representation

a vector of offset values: o0=0, o1=5, o2=5, o3=0

Tweak operator

o0=0, o1=5, o2=5, o3=0 → o0=0, o1=5, o2=10, o3=0

Synchronization Constraints

offset values are modified to satisfy constraints

Fitness Function

max CPU time usage per time slot

Summary of Problem and Solution

Optimization

while satisfying synchronization/ temporal constraints

Explicit Time Model

for real-time embedded systems

Search

meta-heuristic single objective search algorithms

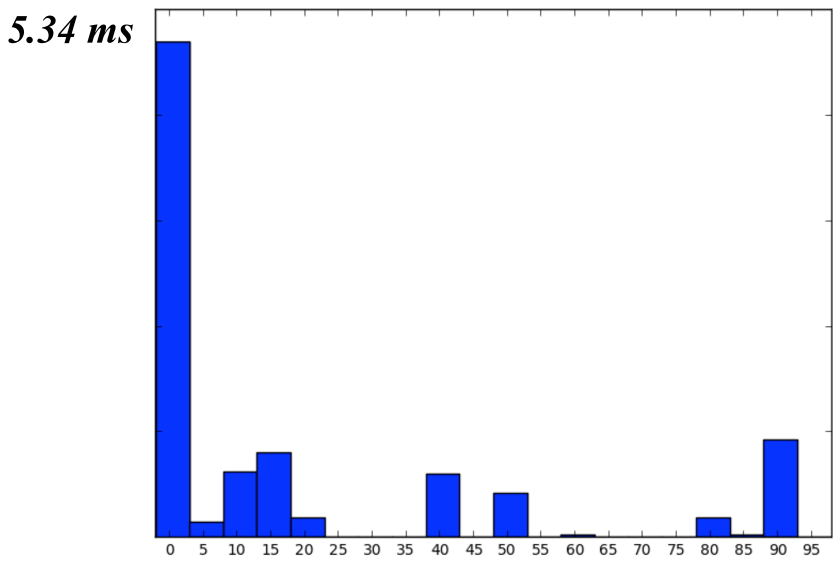
10^27

an industrial case study with a large search space

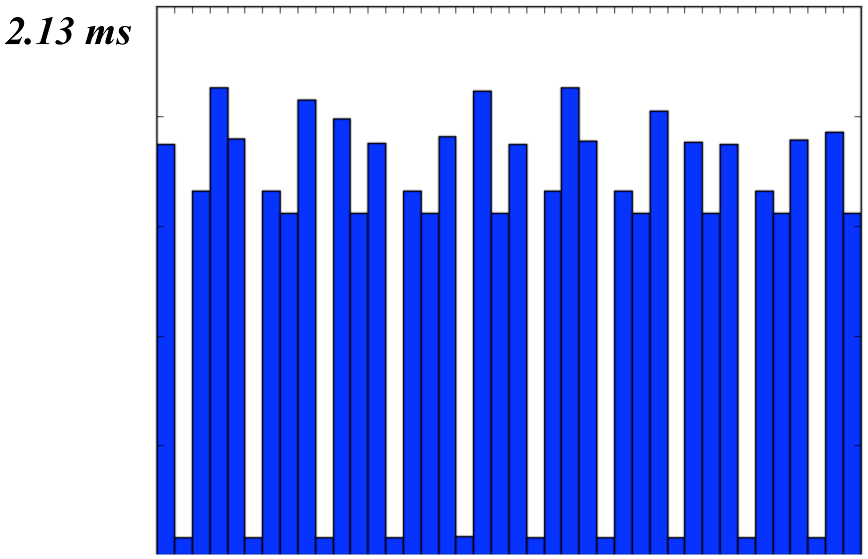
Search Solution and Results

- The objective function is the max CPU usage of a 2s-simulation of runnables
- The search modifies one offset at a time, and updates other offsets only if timing constraints are violated
- Single-state search algorithms for discrete spaces (HC, Tabu)

Case Study: an automotive software system with 430 runnables, search space = 10^27



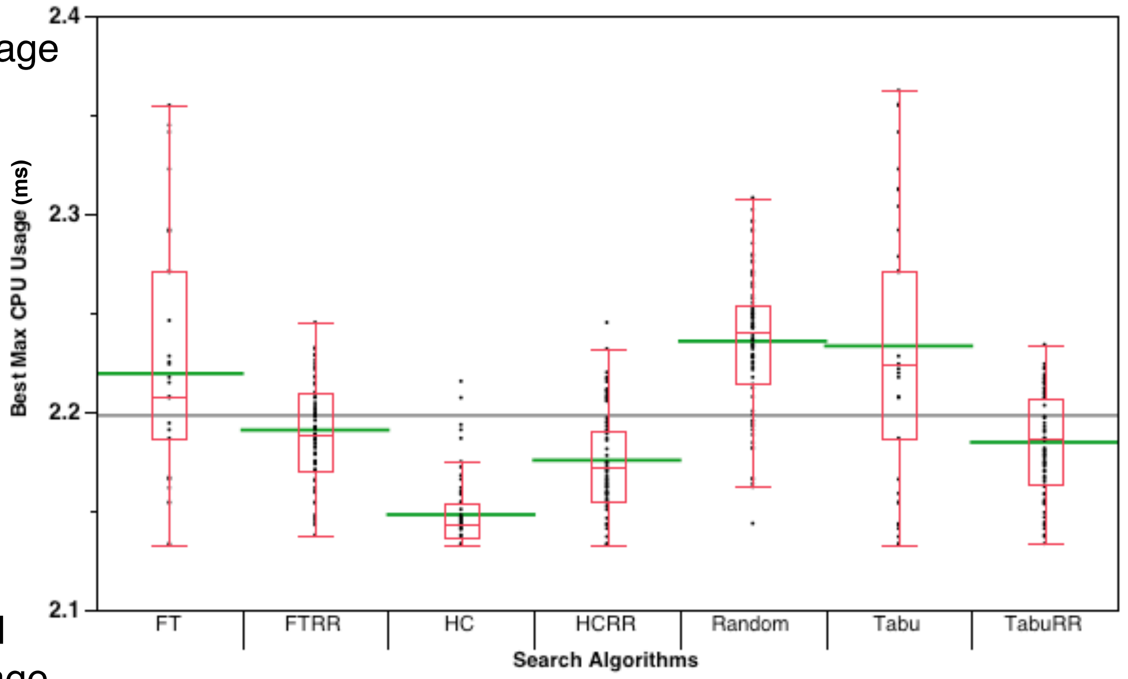
Running the system without offsets



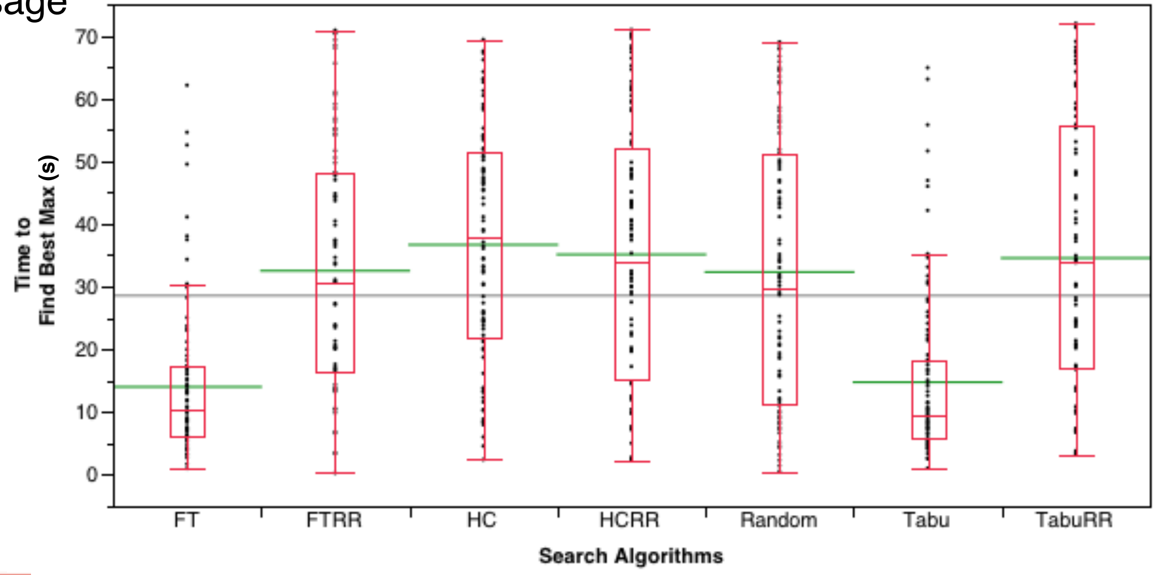
Optimized offset assignment

Comparing different search algorithms

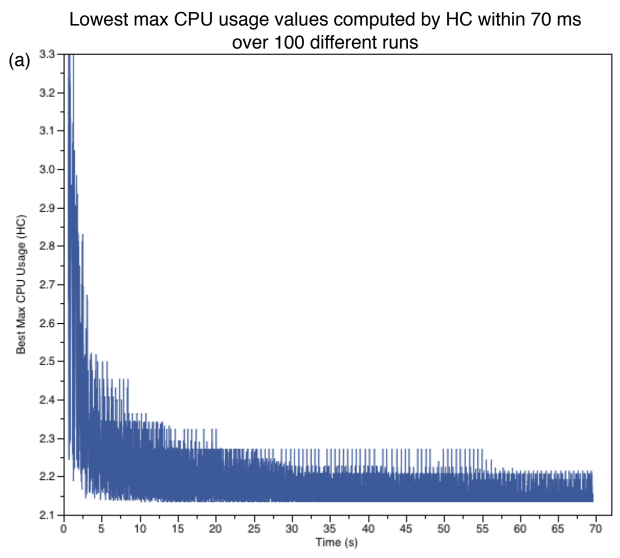
Best CPU usage

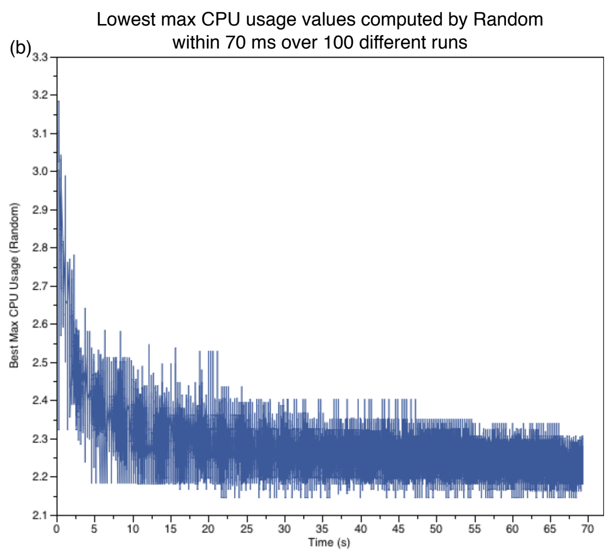


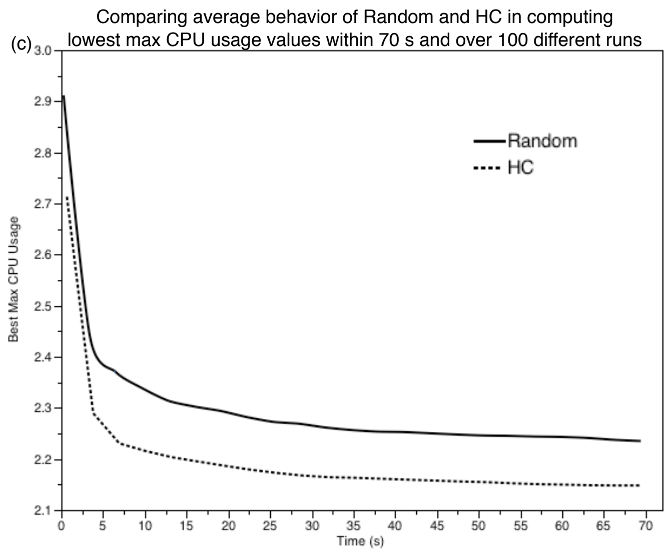
Time to find
Best CPU usage



Comparing our best search algorithm with random search







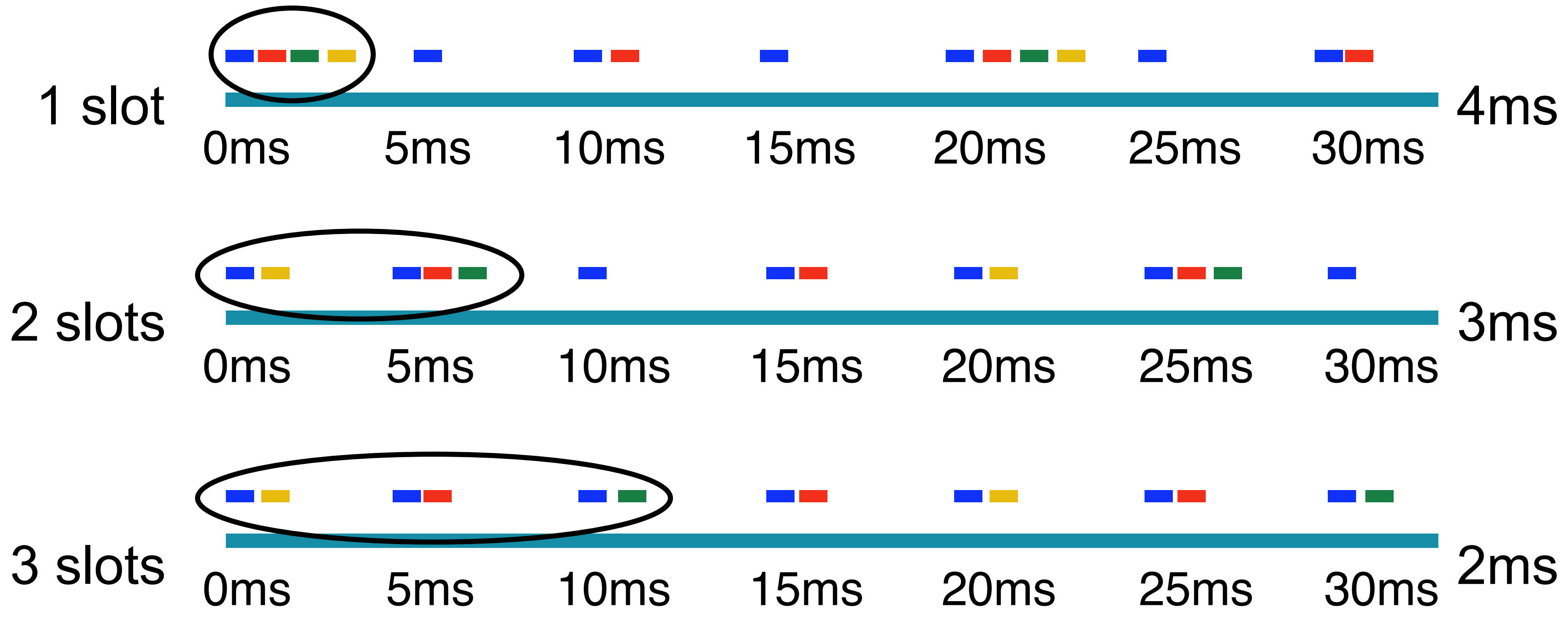
HC *Random* *Average*

Trade-off between Objectives

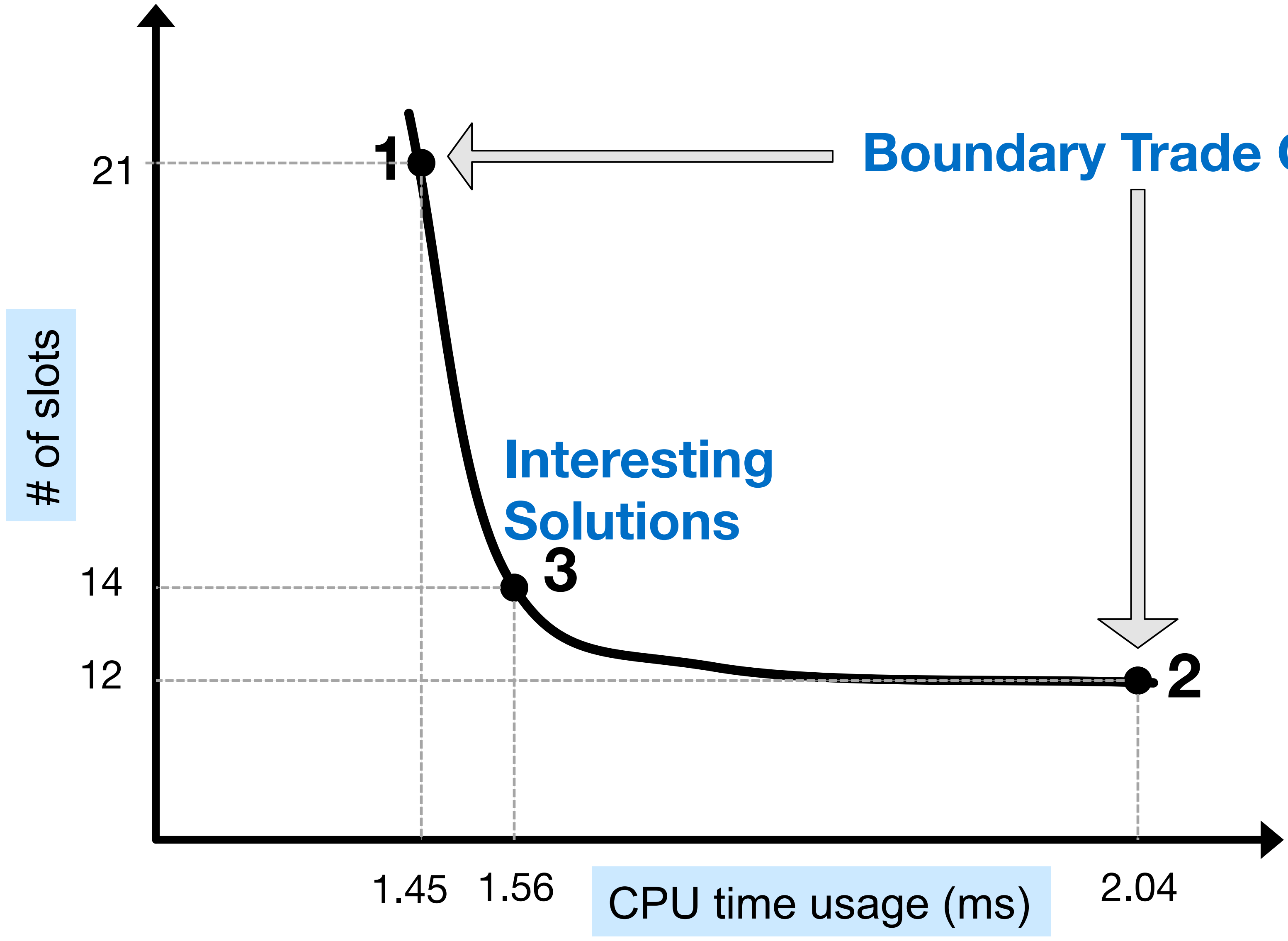
Car Makers r_0 🟦 r_1 🟥 r_2 🟩 r_3 🟨 **Part Suppliers**

Execute r_0 to r_3 close to one another.

Minimize CPU time usage



Trade-off curve

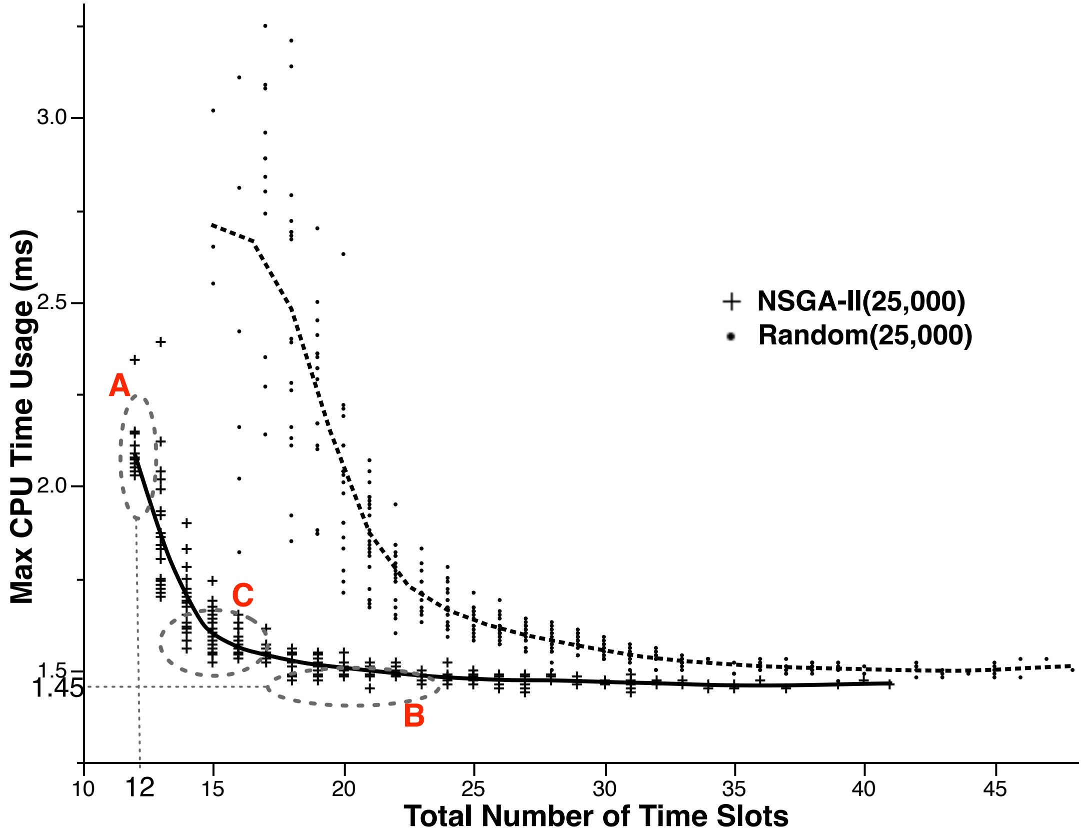


Multi-objective search

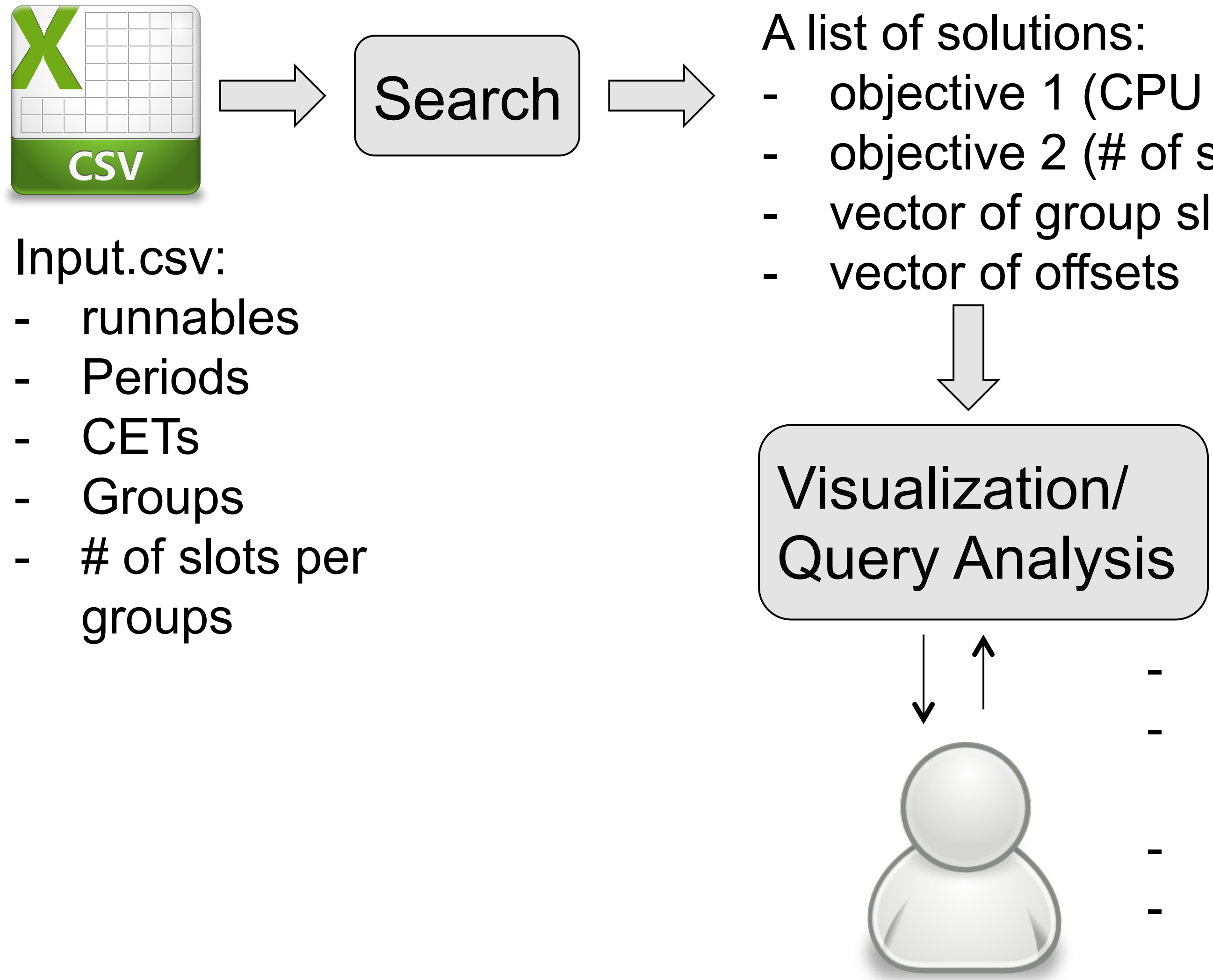
- Multi-objective genetic algorithms (NSGA II)
- Pareto optimality
- Supporting decision making and negotiation between stakeholders

Objectives:

- (1) Max CPU time
- (2) Maximum time slots between "dependent" tasks



Trade-Off Analysis Tool



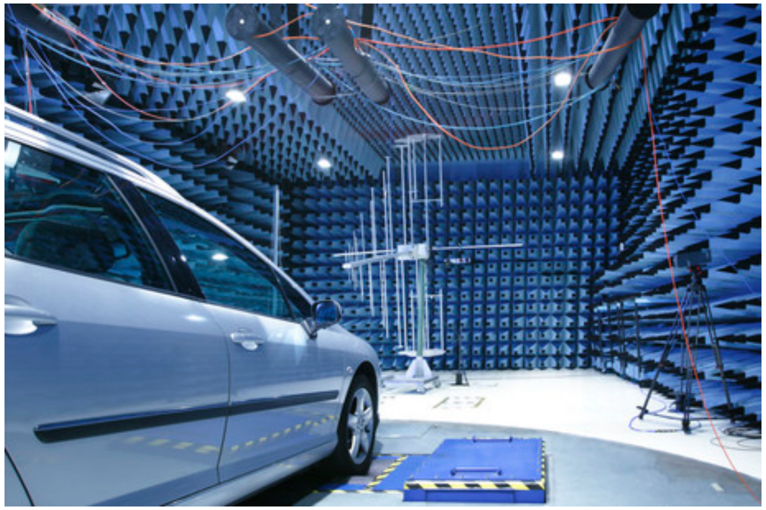
Input.csv:
- runnables
- Periods
- CETs
- Groups
- # of slots per groups

A list of solutions:
- objective 1 (CPU usage)
- objective 2 (# of slots)
- vector of group slots
- vector of offsets

- Visualize solutions
- Retrieve/visualize simulations
- Visualize Pareto Fronts
- Apply queries to the solutions

Conclusions

- Search algorithms to compute offset values that reduce the max CPU time needed
- Generate reasonably good results for a large automotive system and in a small amount of time
- Used multi-objective search → tool for establishing trade-off between relaxing synchronization constraints and maximum CPU time usage



Schedulability Analysis and Stress Testing

References:

- *S. Di Alesio et al., "Stress Testing of Task Deadlines: A Constraint Programming Approach", IEEE ISSRE 2013, San Jose, USA*
- *S. Di Alesio et al., "Worst-Case Scheduling of Software Tasks – A Constraint Optimization Model to Support Performance Testing, Constraint Programming (CP), 2014*
- *S. Di Alesio er al. "Combining Genetic Algorithms and Constraint Programming to Support Stress Testing", ACM TOSEM, 25(1), 2015*

Real-time, concurrent systems (RTCS)

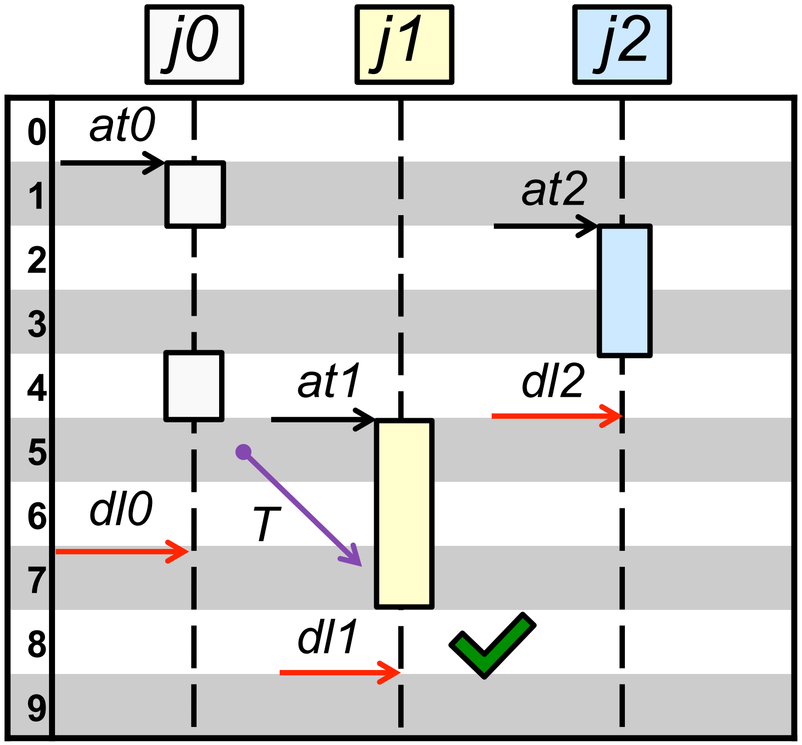
- Real-time, concurrent systems (RTCS) have concurrent interdependent tasks which have to finish before their deadlines
- Some task properties depend on the environment, some are design choices
- Tasks can trigger other tasks, and can share computational resources with other tasks
- How can we determine whether tasks meet their deadlines?

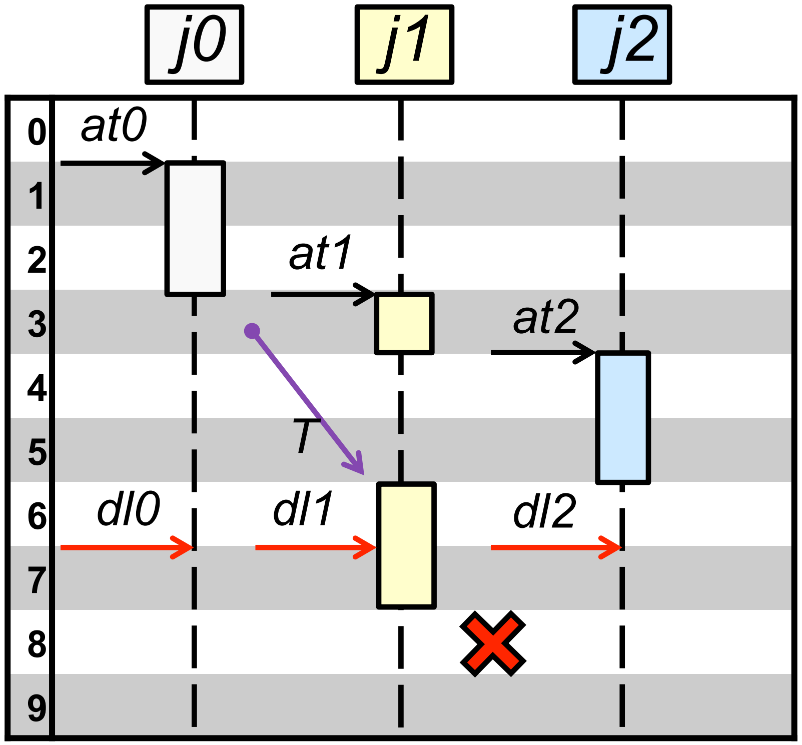
Problem

- Schedulability analysis encompasses techniques that try to predict whether all (critical) tasks are schedulable, i.e., meet their deadlines
- Stress testing runs carefully selected test cases that have a high probability of leading to deadline misses
- Stress testing is complementary to schedulability analysis
- Testing is typically expensive, e.g., hardware in the loop
- Finding stress test cases is difficult

Finding Stress Test Cases is Difficult

j0, j1 , j2 arrive at at0 , at1 , at2 and must finish before dl0 , dl1 , dl2





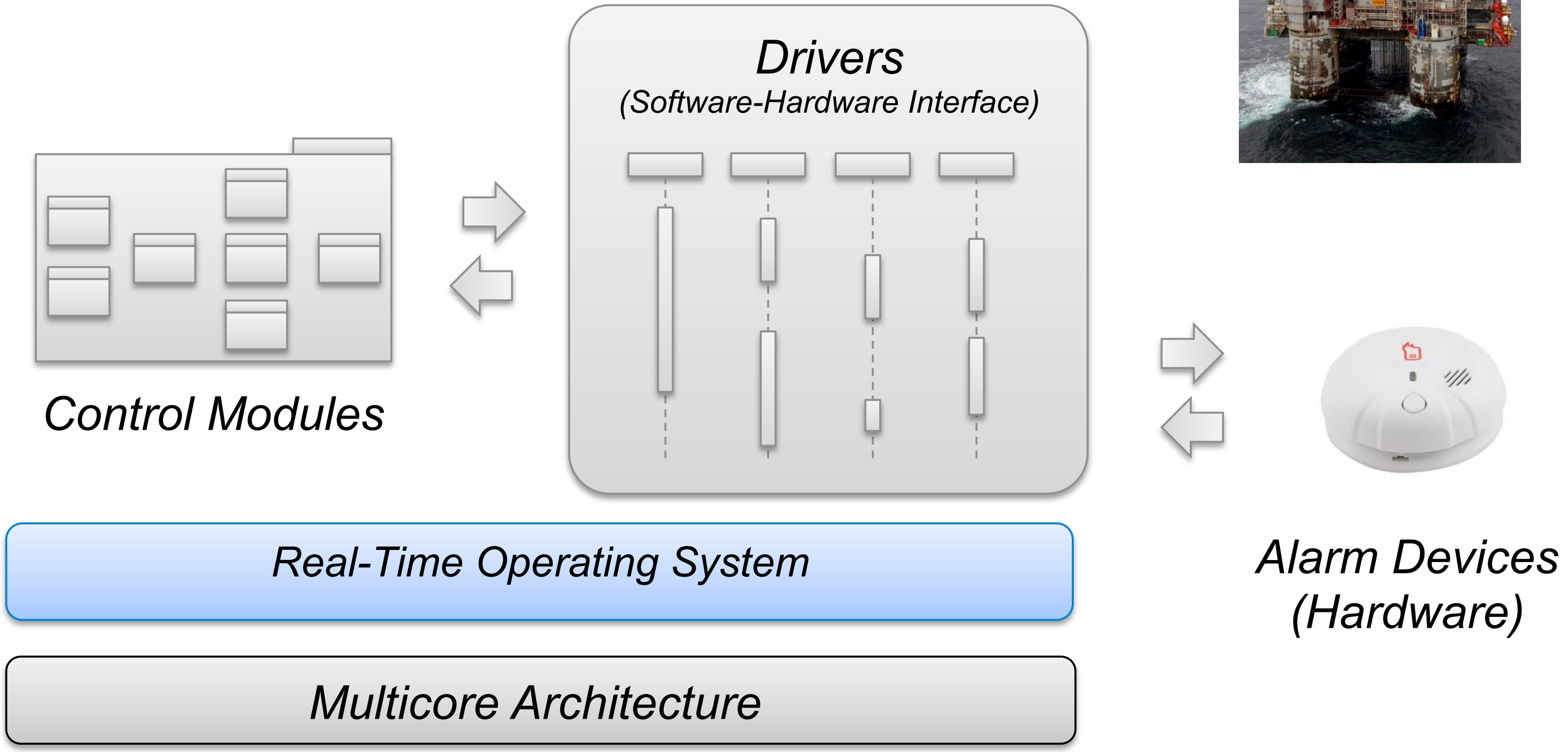
J1 can miss its deadline dl1 depending on when at2 occurs!

Challenges and Solutions

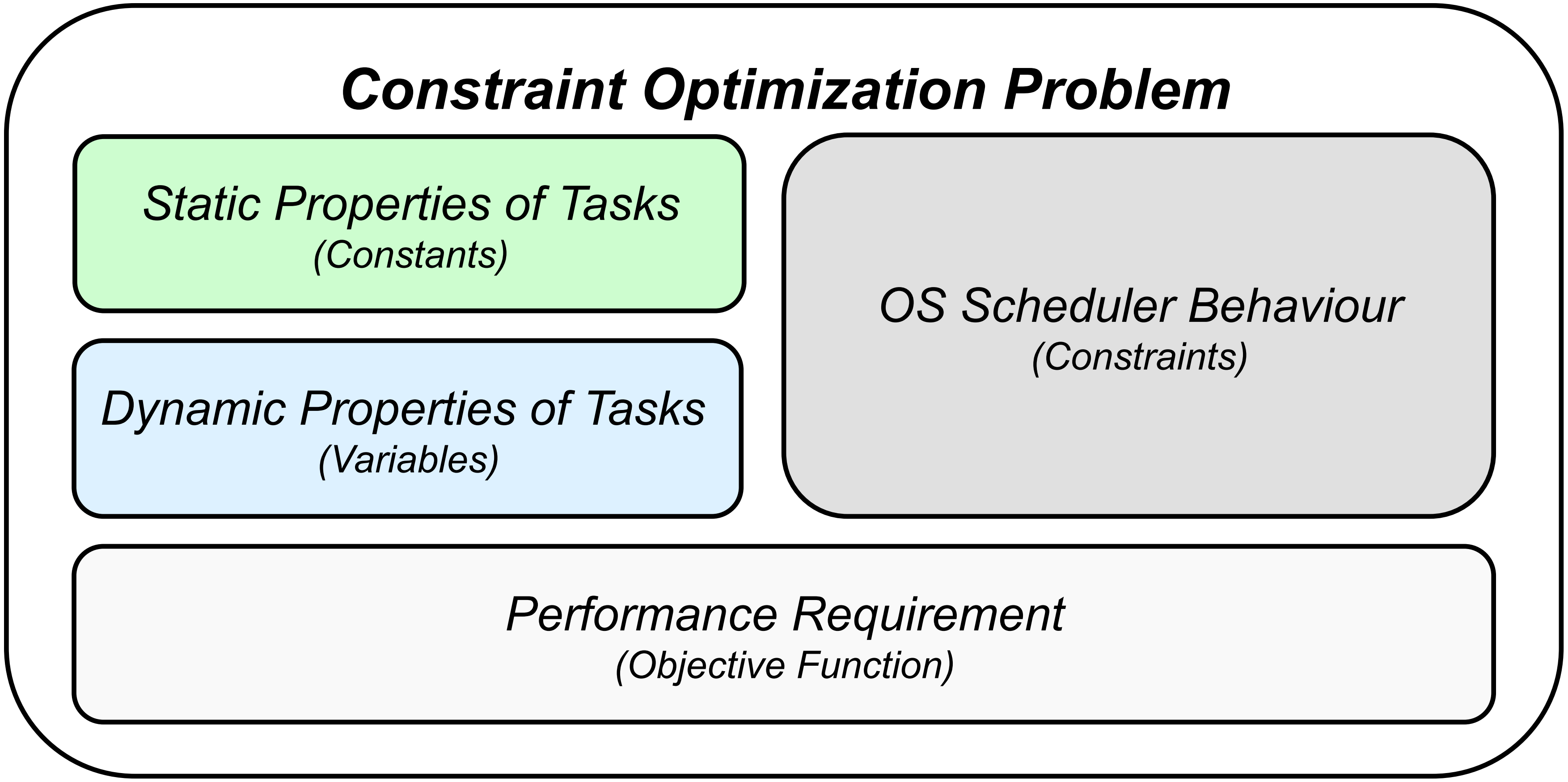
- Ranges for arrival times form a very large input space
- Task interdependencies and properties constrain what parts of the space are feasible
- We re-expressed the problem as a constraint optimisation problem
- Constraint programming (e.g., IBM CPLEX)

Context

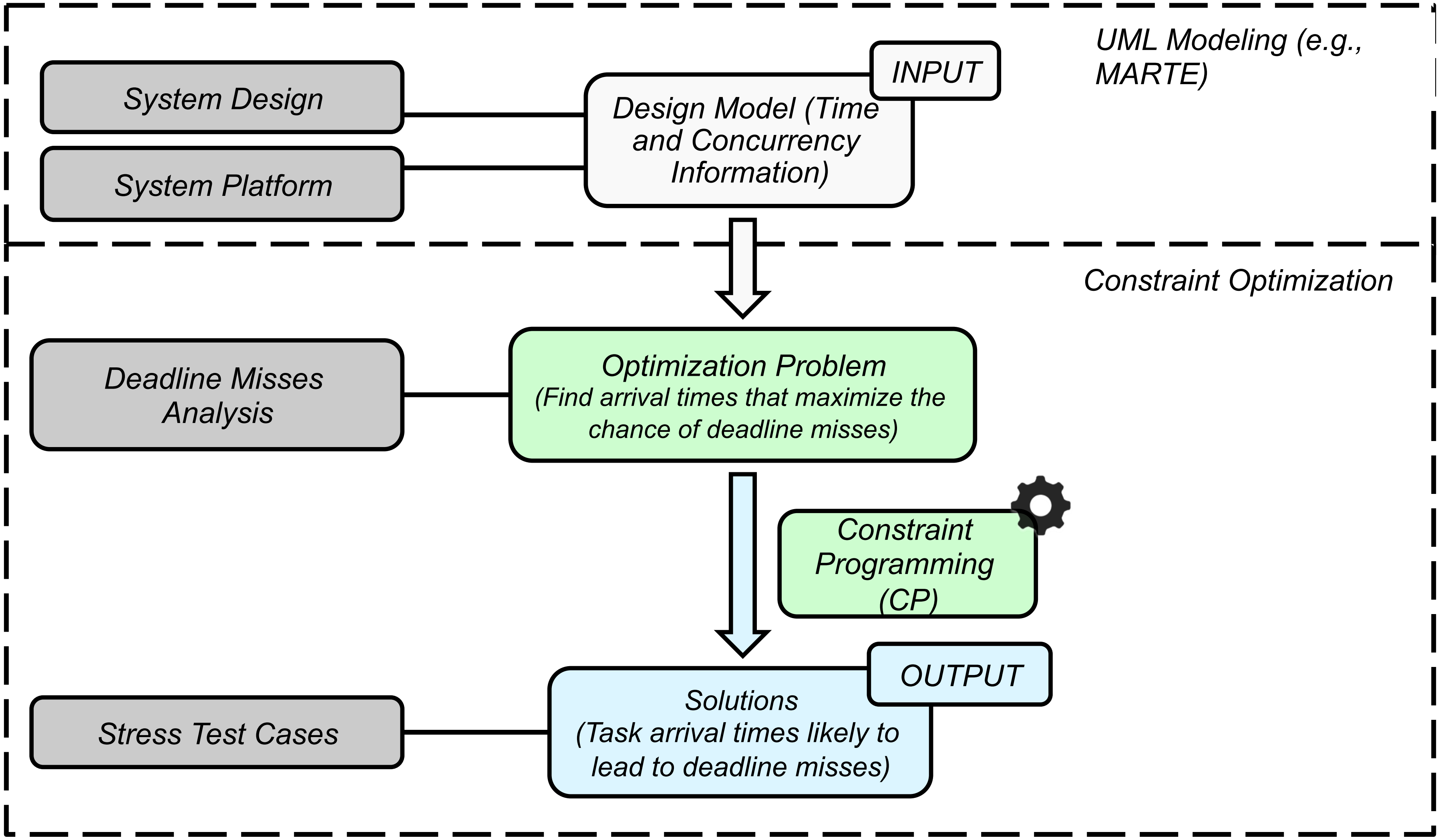
System monitors gas leaks and fire in oil extraction platforms



Constraint Optimization



Process and Technologies

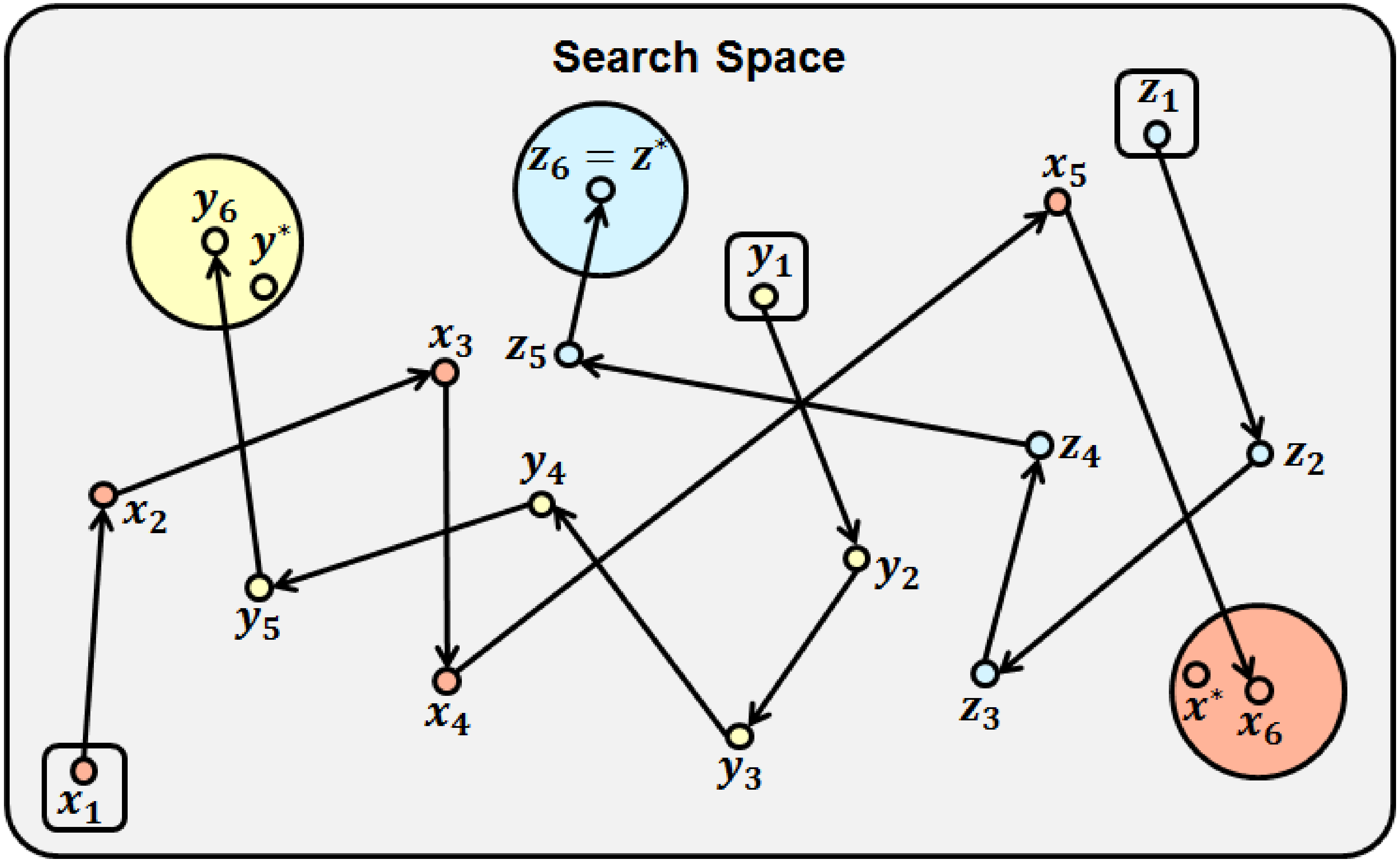

UML Modeling (e.g., MARTE)
System Design
System Platform
INPUT
Design Model (Time and Concurrency Information)
Constraint Optimization
Deadline Misses Analysis
Optimization Problem
(Find arrival times that maximize the chance of deadline misses)
Constraint Programming (CP)
OUTPUT
Stress Test Cases
Solutions
(Task arrival times likely to lead to deadline misses)

Challenges and Solutions

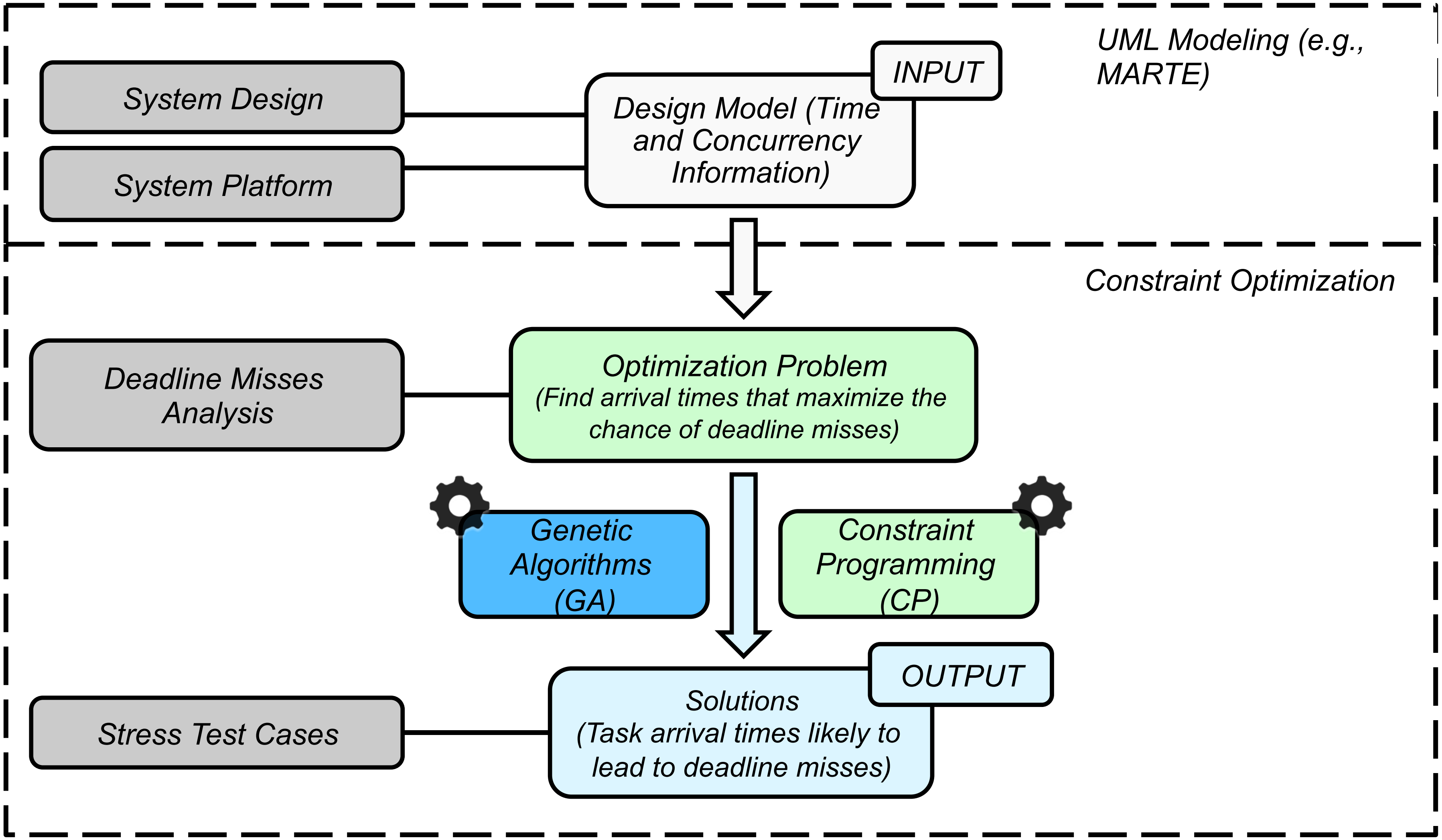
- Scalability problem: Constraint programming (e.g., IBM CPLEX) cannot handle such large input spaces (CPU, memory)

- Solution: Combine metaheuristic search and constraint programming
  – metaheuristic search (GA) identifies high risk regions in the input space
  – constraint programming finds provably worst-case schedules within these (limited) regions
  – Achieve (nearly) GA efficiency and CP effectiveness

Combining GA and CP



Process and Technologies


UML Modeling (e.g., MARTE)
System Design
System Platform
INPUT
Design Model (Time and Concurrency Information)
Constraint Optimization
Deadline Misses Analysis
Optimization Problem (Find arrival times that maximize the chance of deadline misses)
Genetic Algorithms (GA)
Constraint Programming (CP)
OUTPUT
Stress Test Cases
Solutions (Task arrival times likely to lead to deadline misses)

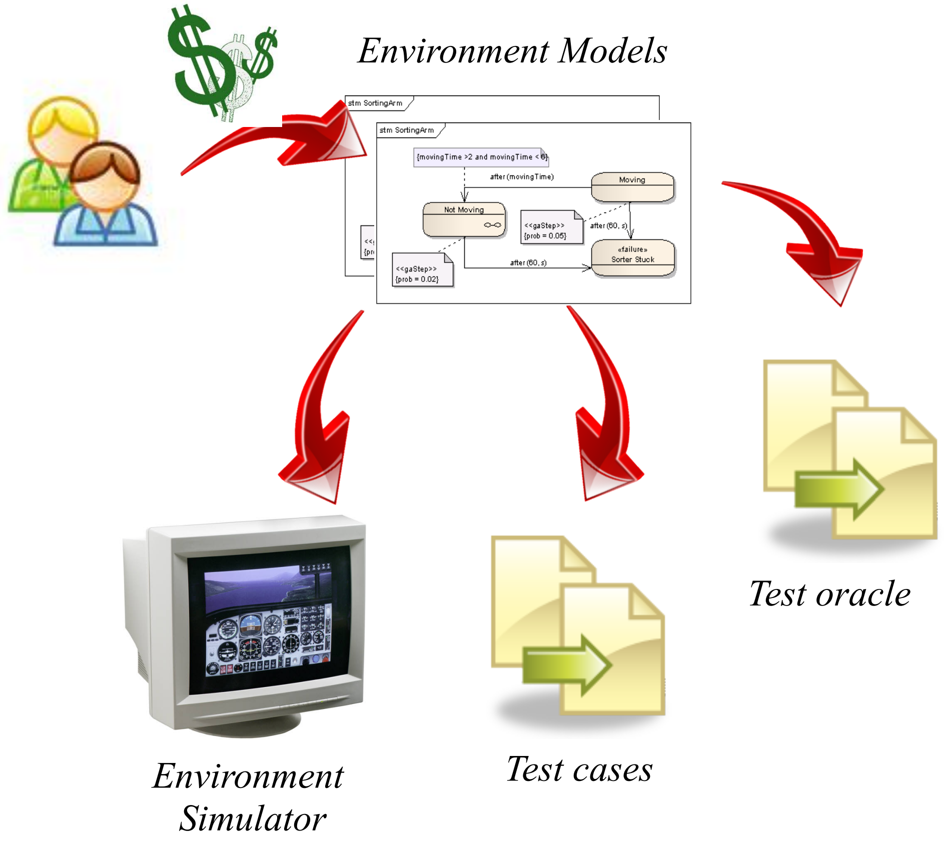
Environment-Based Testing of Soft Real-Time Systems

References:

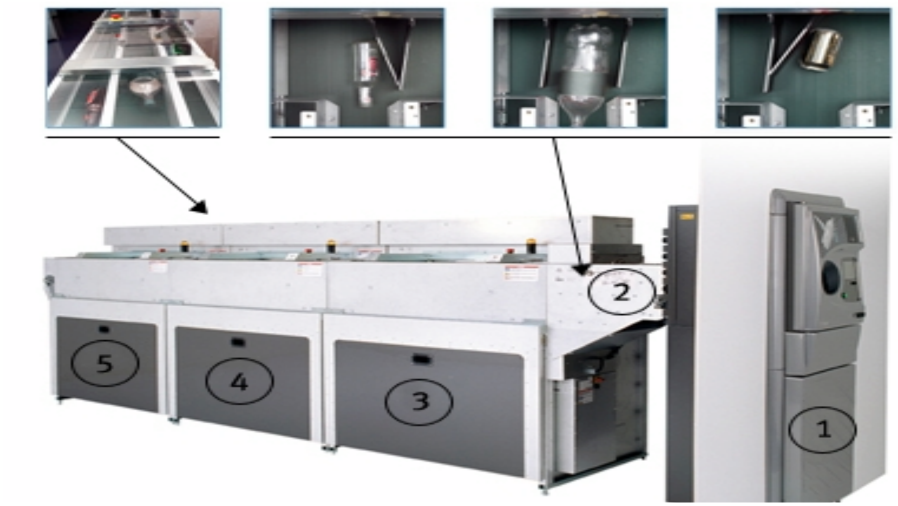
- *Z. Iqbal et al., "Empirical Investigation of Search Algorithms for Environment Model-Based Testing of Real-Time Embedded Software", ACM ISSTA, 2012*
- *Z. Iqbal et al., "Environment Modeling and Simulation for Automated Testing of Soft Real-Time Embedded Software", Software and System Modeling (Springer), 2014*

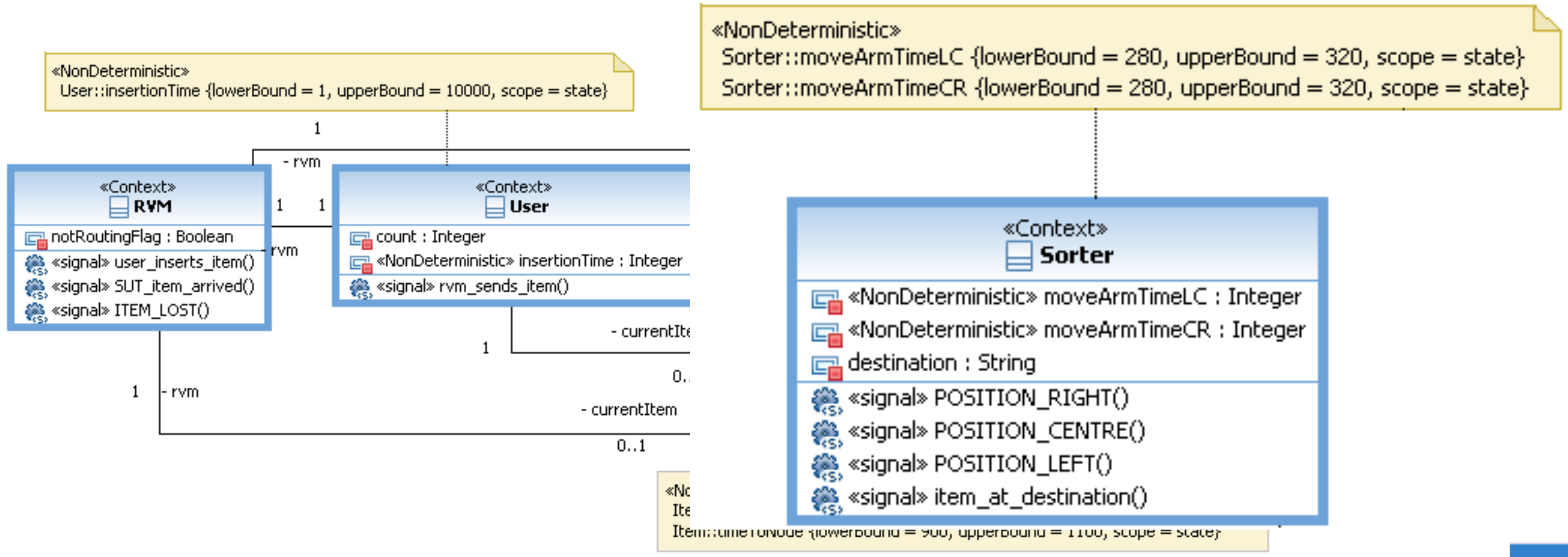
Objectives

- Model-based system testing
  - Independent test team
  - Black-box
  - Environment models



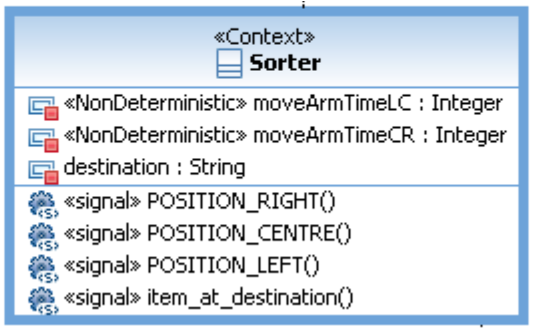
Environment: Domain Model





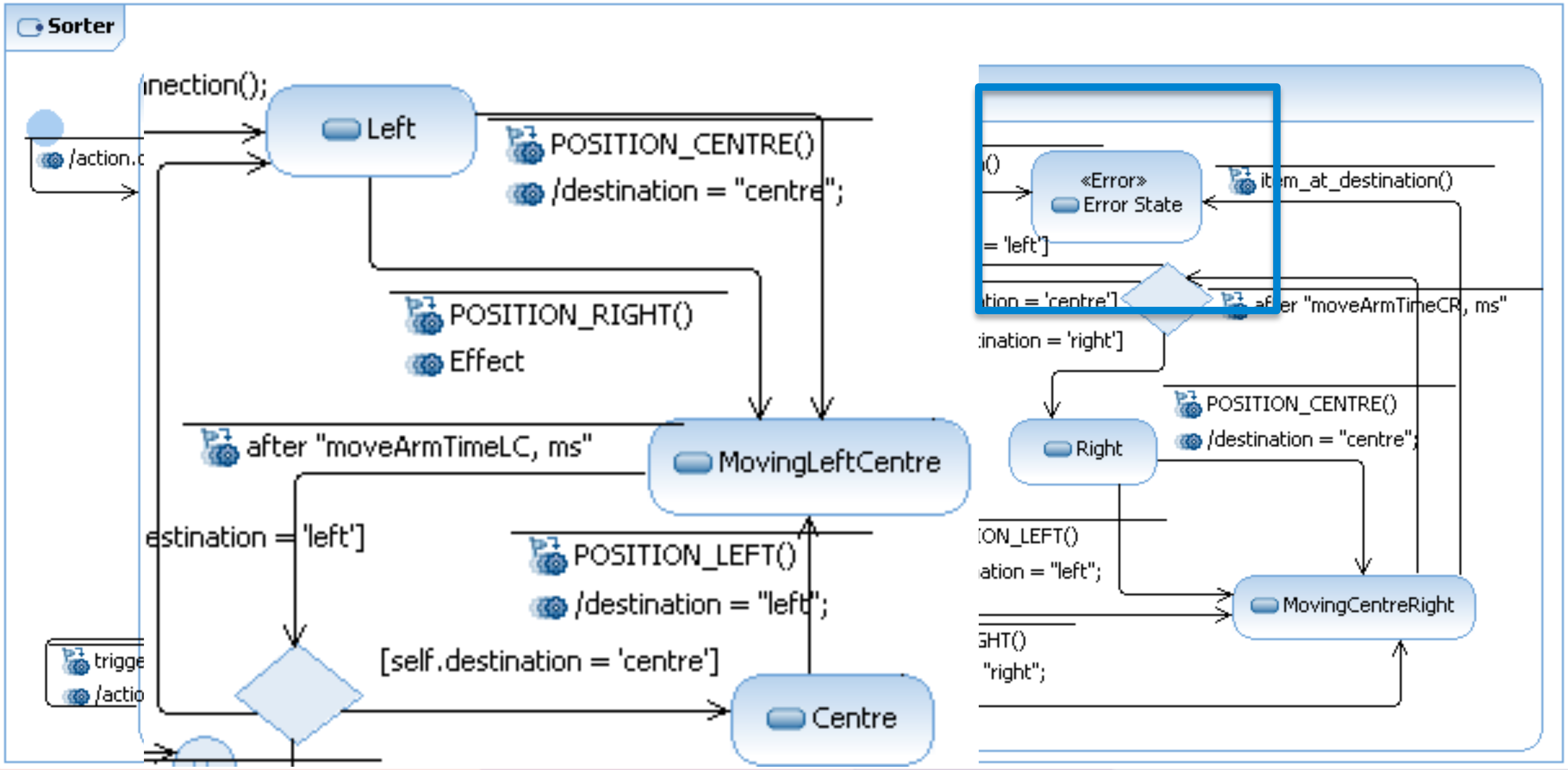
«NonDeterministic»
User::insertionTime {lowerBound = 1, upperBound = 10000, scope = state}
«NonDeterministic»
Sorter::moveArmTimeLC {lowerBound = 280, upperBound = 320, scope = state}
Sorter::moveArmTimeCR {lowerBound = 280, upperBound = 320, scope = state}
«Context»
RVM
notRoutingFlag : Boolean
«signal» user_inserts_item()
«signal» SUT_item_arrived()
«signal» ITEM_LOST()
«Context»
User
count : Integer
«NonDeterministic» insertionTime : Integer
«signal» rvm_sends_item()
«Context»
Sorter
«NonDeterministic» moveArmTimeLC : Integer
«NonDeterministic» moveArmTimeCR : Integer
destination : String
«signal» POSITION_RIGHT()
«signal» POSITION_CENTRE()
«signal» POSITION_LEFT()
«signal» item_at_destination()
- rvm
1
1
1
- rvm
- currentIte
1
0..
1
- rvm
- currentItem
0..1
«No
Ite
Item::timeToNode {lowerBound = 900, upperBound = 1100, scope = state}

Environment: Behavioral Model


«Context»
Sorter

«NonDeterministic» moveArmTimeLC : Integer
«NonDeterministic» moveArmTimeCR : Integer
destination : String

«signal» POSITION_RIGHT()
«signal» POSITION_CENTRE()
«signal» POSITION_LEFT()
«signal» item_at_destination()



Test Case Generation

- Test objectives: Reach “error” states (critical environment states)
- Test Case: Simulation Configuration
  - Setting non-deterministic properties of the environment, e.g., speed of sorter’s left and right arms
- Oracle: Reaching an “error” state
- Metaheuristics: search for test cases getting to error state
- Fitness functions
  - Distance from error state
  - Distance from satisfying guard conditions
  - Time distance
  - Time in “risky” states

Stress Testing focused on Concurrency Faults

Reference:

M. Shousha et al., "A UML/MARTE Model Analysis Method for Uncovering Scenarios Leading to Starvation and Deadlocks in Concurrent Systems", IEEE Transactions on Software Engineering 38(2), 2012

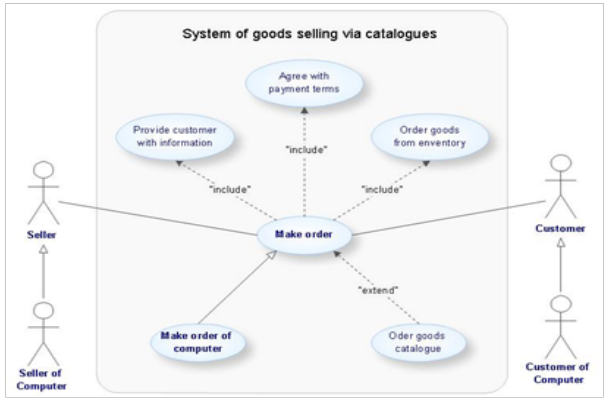
Stress Testing of Distributed Systems

Reference:

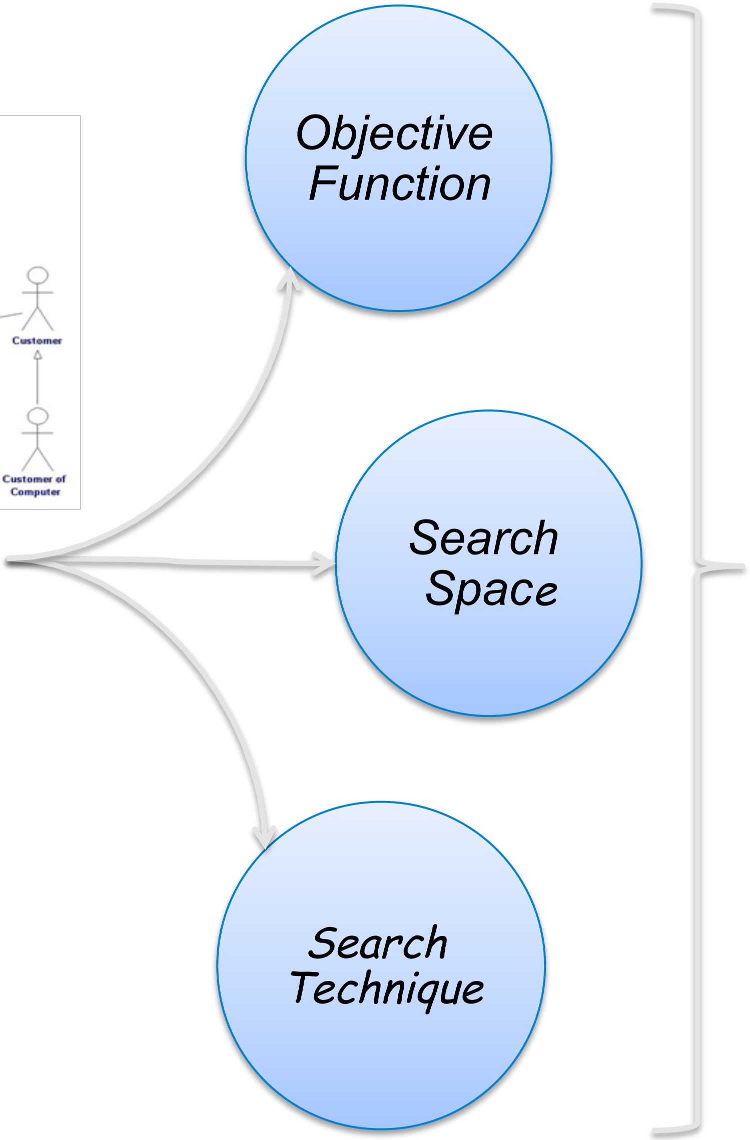
V. Garousi et al., "Traffic-aware Stress Testing of Distributed Real-Time Systems Based on UML Models using Genetic Algorithms", Journal of Systems and Software (Elsevier), 81(2), 2008

General Pattern: Using Metaheuristic Search

Model



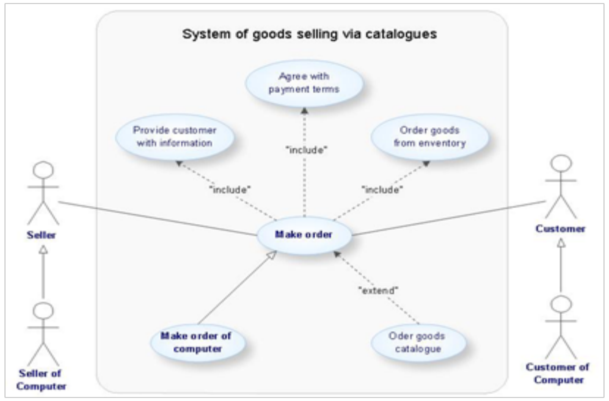
Problem



- Problem = fault model
- Model = system or environment
- Search to optimize objective function(s)
- Metaheuristics, constraint programming
- Scalability: A small part of the search space is traversed
- Model: Guidance to worst case, high risk scenarios across space
- Reasonable modeling effort based on standards or extension
- Heuristics: Extensive empirical studies are required

General Pattern: Using Metaheuristic Search

Model



+

Problem

Objective Function

Search Space

Search Technique

→ Simulator

- Simulation can be time consuming
- Makes the search impractical or ineffective
- Surrogate modeling based on machine learning
- Simulator dedicated to search

Scalability

Project examples

- Scalability is the most common verification challenge in practice

- Testing closed-loop controllers, vision system
  - Large input and configuration space
  - Smart heuristics to avoid simulations (machine learning)
- Schedulability analysis and stress testing
  - Large space of possible arrival times
  - Constraint programming cannot scale by itself
  - CP was carefully combined with genetic algorithms

Scalability: Lessons Learned

- Scalability must be part of the problem definition and solution from the start, not a refinement or an after-thought
- Meta-heuristic search, by necessity, has been an essential part of the solutions, along with, in some cases, machine learning, statistics, etc.
- Scalability often leads to solutions that offer “best answers” within time constraints, but no guarantees
- Scalability analysis should be a component of every research project – otherwise it is unlikely to be adopted in practice
- How many papers research papers do include even a minimal form of scalability analysis?

Practicality

Project examples

- Practicality requires to account for the domain and context

- Testing controllers
  - Relies on Simulink only
  - No additional modeling or complex translation
  - Within domains, differences have huge implications in terms of applicability (open versus closed loop controllers)
- Minimizing risks of CPU shortage
  - Trade-off between between effective synchronisation and CPU usage
  - Trade-off achieved through multiple-objective GA search and appropriate decision tool
- Schedulability analysis and stress testing
  - Near deadline misses must also be identified

Practicality: Lessons Learned

- In software engineering, and verification in particular, just understanding the real problems in context is difficult
- What are the inputs required by the proposed technique?
- How does it fit in development practices?
- Is the output what engineers require to make decisions?
- There is no unique solution to a problem as they tend to be context dependent, but a context is rarely unique and often representative of a domain or type of system

Discussion

- Metaheuristic search for verification
  - Tends to be versatile, tailorable to new problems and contexts
  - Can cope with the verification of continuous behavior
  - Entails acceptable modeling requirements
  - Can provide “best” answers at any time
  - Scalable, practical

  But

  - Not a proof, no certainty
  - Effectiveness of search guidance is key and must be experimented and evaluated
  - Models are key to provide adequate guidance
  - Search must often be combined with other techniques, e.g., machine learning

Discussion II

- Constraint solvers (e.g., Comet, ILOG CPLEX, SICStus)
  – Is there an efficient constraint model for the problem at hand?
  – Can effective heuristics be found to order the search?
  – Better if there is a match to a known standard problem, e.g., job shop scheduling
  – Tend to be strongly affected by small changes in the problem, e.g., allowing task pre-emption
  – Often not scalable, e.g., memory
- Model checking
  – Detailed operational models (e.g., state models), involving (complex) temporal properties (e.g., CTL)
  – Enough details to analyze statically or execute symbolically
  – These modeling requirements are usually not realistic in actual system development. State explosion problem.
  – Originally designed for checking temporal properties through reachability analysis, as opposed to explicit timing properties
  – Often not scalable

Talk Summary

- Focus: Meta-heuristic Search to enable scalable verification and testing.
- Scalability is the main challenge in practice.
- We drew lessons learned from example projects in collaboration with industry, on real systems and in real verification contexts.
- Results show that meta-heuristic search contributes to mitigate the scalability problem.
- It has also shown to lead to applicable solutions in practice.
- Solutions are very context dependent.
- Solutions tend to be multidisciplinary: system modeling, constraint solving, machine learning, statistics.

Making Model-Driven Verification Practical and Scalable - Experiences and Lessons Learned

Lionel Briand

Interdisciplinary Centre for ICT Security, Reliability, and Trust (SnT)
University of Luxembourg, Luxembourg

MODELSWARD, Rome, February 20, 2016

SVV lab: svv.lu
SnT: www.securityandtrust.lu