MISSIONE 4
ISTRUZIONE
RICERCA

FORESEEN

Deliverable D4.1 - Report on the process to build a formal model for vehicular behavior traces and on-line tests generation

Finanziato dall'Unione europea NextGenerationEU
Ministero dell'Università e della Ricerca
Italiadomani PIANO NAZIONALE DI RIPRESA E RESILIENZA
Università di Pisa

MISSIONE 4
ISTRUZIONE
RICERCA

FORESEEN

FORmal m**E**thod**S** for attack d**E**t**E**ction in autonomous drivi**N**g systems

PRIN 2022 PNRR

Project number: P2022WYAEW
CUP: I53D23006130001

Deliverable D4.1: **Report on the process to build a formal model for vehicular behavior traces and on-line tests generation**

Project Start Date: 30/11/2023 **Duration**: 24 months

Coordinator: *University of Pisa*

Deliverable No	D4.1
WP No:	WP3
WP Leader:	RU-MOL
Tasks:	T3.1, T3.2, T3.3, T3.5 - Leader RU-MOL
Due date:	M9-20
Delivery date:	July 31, 2025
Authors:	RU-MI, RU-MOL, RU-PA, RU-PI

Dissemination Level:

PU	Public	X
PP	Restricted to other programme participants (including the Commission Services)	
RE	Restricted to a group specified by the consortium (including the Commission Services)	
CO	Confidential, only for members of the consortium (including the Commission Services)	

Contents

List of Acronyms

CACC	Cooperative Adaptive Cruise Control
CCS	Calculus of Communicating Systems
CPA	Correlation Power Analysis
CPS	Cyber Physical System
CSV	Comma Separated Values
CWB-NC	Concurrency Workbench of New Century
HTML	HyperText Markup Language
INTO-CPS	Integrated Toolchain for model-based design of Cyber Physical Systems
JSON	JavaScript Object Notation
SMC	Statistical Model Checking
V2E	Vehicle to Edge
V2V	Vehicle to Vehicle

1 Introduction

This deliverable describes the methodology defined in WP3 to build a formal model for vehicular behavioral traces, starting from co-simulations logs produced in WP2; and to identify on-line tests for attack detection, starting from labeling of data related to the position of cars (e.g, too far or too close), and labeling of the trace (e.g., no attack, attack to the leader, or attack to car i). Moreover, stochastic model checking has been used to analyse, for example, the probability of an attack having a significant effect on the distance between cars, within a very short period of time.

In particular, the formal model of behavioral traces is built starting from processes expressed in the CCS process algebras in Task T3.1; then in Task T3.2 patterns for detection of attacks are identified and in T3.3 they are formalized in mu-calculus temporal logic; finally in Task T3.5 explainability of the proposed approach is analysed.

2 Background

In the following sections, we provide the necessary background to support the methodology presented in this project. We introduce the key concepts and formal foundations related to process modeling, symbolic trace abstraction, and formal verification through model checking.

2.1 Model Checking for Behavioral Verification

Formal verification techniques, and in particular model checking [Clarke99], offer a systematic approach to verifying properties of system behavior by exhaustively exploring all possible execution paths of a formal model. This technique proves especially useful in safety-critical contexts such as cooperative vehicular systems, where ensuring adherence to specific behavioral constraints is essential.

In our methodology, the behavior of each vehicle is encoded as a process in the Calculus of Communicating Systems (CCS) [Mil89], a well-established process algebra that allows for the representation of concurrent systems. These CCS models are then subject to model checking, using modal mu-calculus [Koz83] as the specification language for properties of interest.

The mu-calculus is a powerful formalism capable of expressing a wide range of temporal and modal properties, including reachability, safety, liveness, and synchronization conditions. For example, properties can express constraints such as: “A vehicle never enters a critical state of proximity for more than two consecutive time steps” or “Acceleration and deceleration patterns alternate consistently under normal driving conditions”.

Model checking is the automatic verification of whether a transition system satisfies a temporal logic formula. By systematically exploring all reachable states of the system, it checks the validity of the specified properties and ensures that the modeled behavior adheres to expected rules.

In our methodology, model checking enables the automated validation of behavioral properties over the entire symbolic model derived from real vehicular traces, serving as a rigorous bridge between simulation data and formal safety assurance. Specifically, it allows us to:

- Validate that symbolic traces follow safe behavioral patterns (e.g., avoiding critical low distances).
- Detect violations of coordination rules (e.g., ensuring that all vehicles synchronize at each time step).
- Formally verify that the modeled platoon dynamics comply with the intended operational modes.

2.1.2 Calculus of Communicating Systems (CCS)

The Calculus of Communicating Systems (CCS) is a process algebra introduced by Robin Milner to formally model and reason about concurrent systems. In CCS, system behavior is described in terms of processes that perform actions, possibly in parallel and with synchronization. Each process is defined recursively, using a set of syntactic constructors. The basic syntax of CCS is as follows:

```
P ::= nil              -- the inactive process
    | a.P              -- action prefix (a followed by P)
    | P + Q            -- choice (non-deterministic)
    | P | Q            -- parallel composition
    | P \ L            -- restriction (hiding of actions in L)
    | A                -- process identifier (with definition A ≡ P)
```

where:

- `a` is an action, which can be either observable (e.g., `send`, `receive`) or internal (denoted by τ).
- `P` and `Q` are processes.
- `L` is a set of actions to be hidden (restricted).

Example

A simple process that sends a message and then terminates is written as:

$$P = send.nil$$

Two processes that communicate via complementary actions a and $\bar{a}$ can synchronize:

$$P = a.nil \qquad Q = \bar{a}.nil \qquad P \mid Q \rightarrow \tau$$

In our context, each vehicle is modeled as a CCS process, where actions encode symbolic labels of acceleration, speed, and distance. The full platoon is modeled as the parallel composition of all vehicles, possibly with synchronization enforced via additional coordination actions (e.g., `sink`, `go`).

2.1.3 Mu-Calculus for Property Specification

To specify and verify properties over CCS models, we use the mu-calculus, a highly expressive logic that extends modal logic with least (μ) and greatest (ν) fixed points. This makes it particularly well-suited for expressing properties over potentially infinite behaviors, such as safety, liveness, and reachability.

Syntax of mu-calculus

Given a set of actions `Act` and propositional variables `X`, the syntax of modal mu-calculus formulas is:

```
ϕ ::= tt | ff          -- constants
    | X                -- propositional variable
    | ϕ ∧ ϕ | ϕ ∨ ϕ    -- conjunction/disjunction
    | [a]ϕ | <a>ϕ      -- box and diamond modalities
    | μX.ϕ | νX.ϕ      -- least and greatest fixed point
```

- `[a]ϕ`: For all executions of action `a`, `ϕ` holds afterward.
- `<a>ϕ`: There exists an execution of action `a` after which `ϕ` holds.
- `μX.ϕ`: The least fixed point — used to express eventuality (e.g., "the possibility of reaching a good state").
- `νX.ϕ`: The greatest fixed point — used for invariants (e.g., "always avoid unsafe state").

Satisfaction Semantics

The satisfaction of a formula `ϕ` by a state `s` of a transition system (denoted `s ⊨ ϕ`) is defined inductively:

- every state satisfies `tt`; no state satisfies `ff`.
- `s ⊨ ϕ₁ ∨ ϕ₂` if `s ⊨ ϕ₁` or `s ⊨ ϕ₂`.
- `s ⊨ ϕ₁ ∧ ϕ₂` if `s ⊨ ϕ₁` and `s ⊨ ϕ₂`.
- `s ⊨ [K]ϕ` if for every action in the set `K` enabled from `s`, the resulting state also satisfies `ϕ`.
- `s ⊨ ⟨K⟩ϕ` if there exists at least one action in `K` such that the resulting state satisfies `ϕ`.

This semantic interpretation allows us to rigorously define what it means for a CCS model to satisfy a behavioral property.

Example

- Safety: "It is always possible to avoid action `fail`"

  `νX.([fail]ff ∧ [-]X)`

- Reachability: "Eventually action `brake` will occur"

  `μX.(<->tt ∧ <brake>X)`

In our setting, the mu-calculus is used to express properties over CCS models derived from discretized traces, allowing us to verify if symbolic behaviors conform to expected patterns (e.g., vehicles keep safe distances, alternate acceleration phases, or synchronize correctly).

3 Tool for indexing and data overview

To better understand and analyze the vast amount of data generated in our previous work, we designed and implemented a robust indexing utility. This utility was developed to streamline data exploration, enhance accessibility, and facilitate deeper insights into the simulation traces and parameters. The utility is made of two components a web-app that's used to explore the simulation traces and parameters and a Python program used to generate the web-app's data structures.

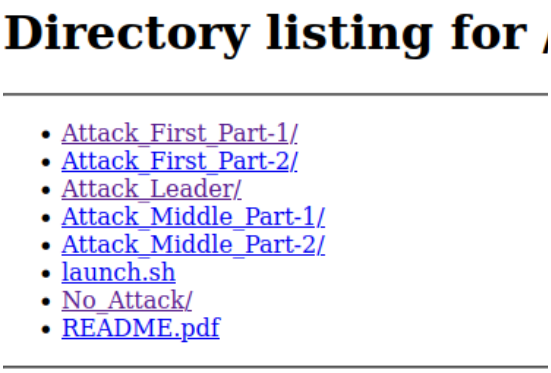

Figure 1: Listing of all roots

The **Python program** generates the files `index.html`. In the simulations' root folder, the aforementioned file contains a *table of contents* containing references to all simulation traces and their parameters. The table itself is generated by iterating through all traces' folder and parsing the `config.mm.json` file to get the parameters that were used by INTO-CPS to run the simulation and generate the simulation trace.

The **web-app**, composed by the HTML files generated by the Python program, requires a running web server to operate properly, a script called `launch.sh` is made available to quickly start a web server on port 9000.

As it is shown in Figure 1, once the web browser is pointed to the web server, we are greeted by a listing of all simulation roots. We can click on one of them to access its table of contents. The table of contents, shown in Figure 2, is a table in which the rows represent simulation traces. For each trace we show its name, with a hyperlink to the trace's root folder; hyperlinks to the generated CSV file and the configuration JSON file; and all the trace's parameters that were used by the simulator. Each parameter's column has a text input box that can be used to filter out simulation traces that do not match with the user's query, for instance we can show only traces that have an attack of type 1 by typing "1" in the column `{Car1}.CarIstance_1.attack`.

Folder	data	config	{Car1}. CarInstance_1. attack	{Car1}. CarInstance_1. attack_amplitude	{Car1}. CarInstance_1. attack_time	{Car1}. CarInstance_1. high_frequency	{Car1}. CarInstance_1. initial_position	{Car1}. CarInstance_ initial_veloci
3_0.08_30_172_0_0_0.1_4_0_0_0_172_0_0_0.1_16_0_0_0_0_0_2_0.8_0_-1.5_0.628_0_0_-0.01_0_0_0.5_5_5_10_4_4_4_4_4_4_4_4_4_4_10	results.csv	config_mm.json	3	0.08	30	172	0	0
3_0.08_30_172_0_0_0.1_4_0_0_0_172_0_0_0.1_16_0_0_0_0_0_2_0.8_0_-1.5_0.5_0_0_-0.01_1_0_0.5_5_3_10_4_4_4_4_4_4_4_4_4_4_10	results.csv	config_mm.json	3	0.08	30	172	0	0
1_0.08_30_172_0_0_0.1_4_0_0_0_172_0_0_0.1_16_0_0_0_0_0_1.7_0.8_0_-1.7_0.628_0_0_-0.01_0_0_0.5_5_5_10_4_4_4_4_4_4_4_4_4_4_10	results.csv	config_mm.json	1	0.08	30	172	0	0
1_0.08_30_172_0_0_0.1_4_0_0_0_172_0_0_0.1_16_0_0_0_0_0_2_0.8_0_-1.7_0.5_0_0_-0.01_1_0_0.5_5_3_8_4_4_4_4_4_4_4_4_4_4_10	results.csv	config_mm.json	1	0.08	30	172	0	0
2_0.08_30_172_0_0_0.1_4_0_0_0_172_0_0_0.1_16_0_0_0_0_0_1.7_0.8_0_-1.7_0.628_0_0_-0.01_0_0_0.5_5_5_8_4_4_4_4_4_4_4_4_4_4_10	results.csv	config_mm.json	2	0.08	30	172	0	0
2_0.08_30_172_0_0_0.1_4_0_0_0_172_0_0_0.1_16_0_0_0_0_0_2_0.8_0_-1.7_0.5_0_0_-0.01_2_0_0.5_5_3_8_4_4_4_4_4_4_4_4_4_4_10	results.csv	config_mm.json	2	0.08	30	172	0	0

Figure 2: The table of contents

Clicking the trace's name we are redirected to the trace's root. In this web page we have a series of plots, an example of them is shown in Figure 3 showing the evolution of the cars' physical quantities against time. More specifically, we plot the following graphs

- acceleration over time, as read by the sensors and as the actual simulation value
- speed over time, as read by the sensors and as the actual simulation value
- x position, as read by the sensors and as the actual simulation value
- inter-vehicular distance, as read by the sensors and as the actual simulation value

The plots are made at runtime using the JavaScript library Apache Echarts[1] by reading the CSV file generated by the simulator. The plots are interactive, the user can move the cursor over the traces and read the exact value of the trace in a certain instant. It is also possible to hide traces and zoom in and out of the plot to see the evolution of the system in greater detail.

[1] https://echarts.apache.org

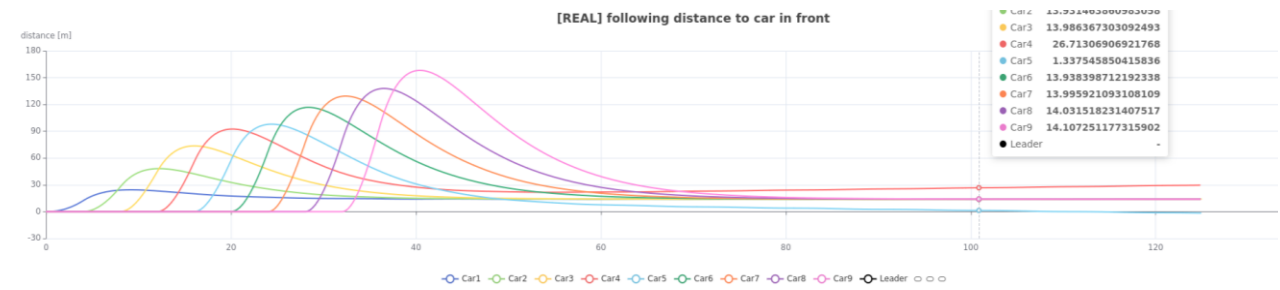

Figure 3: A graph that can be found in a trace's root

At the bottom of the page, another table shows again the trace parameters, this time arranged by row instead of by column.

4 Tool for labeling data

Since we have generated a quite large amount of data in the form of simulation traces, it was quite difficult to manually find the most interesting traces, in terms of attack severity. So we developed a Python program to generate a summary of all traces and their outcome, in the form of a CSV file.

The main idea is to label, for each trace, the behavior of each car and then aggregate them to give a general label to the trace. First, we defined 5 levels of severity ordered by the most severe to least severe as:

$$COLLISION > VIOLATION > TOOCLOSE > TOOFAR > OK$$

For each simulation step t and car i, we assign a label $L(i,t)$ computed in function of the inter-vehicular distance d_i kept between car i and the vehicle in front, as show:

$$L(i,t) = \begin{Bmatrix} OK & if\, d_i \in (12{,}18) \\ COLLISION & if\, d_i \leq 4 \\ TOOFAR & if\, d_i \geq 18 \\ TOOCLOSE & if\, d_i \leq 12 \\ VIOLATION & if\, d_i \leq 10 \end{Bmatrix}$$

Finally, the global label for a car i and for the whole trace, using the previously mentioned ordering, is assigned as follows:

$$L_i = \max_t \{L(i,t)\}$$
$$L = \max_i \{L_i\}$$

Then we imported the CSV files in LibreOffice Calc, a spreadsheet program, and aggregated the results. The following table shows the aggregated results:

Label class	No attack	Attack leader	Attack on car 1 P1	Attack on car 1 P2	Attack on car 4 P1	Attack on car 4 P2
OK	*100.00%*	*33.33%*	*33.33%*	*50.00%*	*33.33%*	*75.00%*
TOO FAR	*0.00%*	*0.00%*	*0.00%*	*0.00%*	*0.00%*	*0.00%*
TOO CLOSE	*0.00%*	*0.00%*	*0.00%*	*50.00%*	*0.00%*	*25.00%*
VIOLATION	*0.00%*	*0.00%*	*0.00%*	*0.00%*	*0.00%*	*0.00%*
COLLISION	*0.00%*	*66.67%*	*66.67%*	*0.00%*	*66.67%*	*0.00%*
# TRACES	*96*	*576*	*288*	*384*	*288*	*384*

The P1 columns refer to attacks on the car's sensors and P2 to the actuators; the leader only has attacks on sensors. Obviously, there is no issue with the *no attack* scenario. We also observe that the attacks on sensors are by far the most dangerous as they result in collisions; while attacks on actuators mainly result in a reduced inter-vehicular distance.

5 Formal model definition and properties specification

The process of formal modeling starts from the analysis of raw numerical traces obtained from simulated vehicular scenarios, i.e., cooperative platooning. These traces contain sampled values of continuous variables such as acceleration, speed, and inter-vehicle distance, computed at successive time instants.

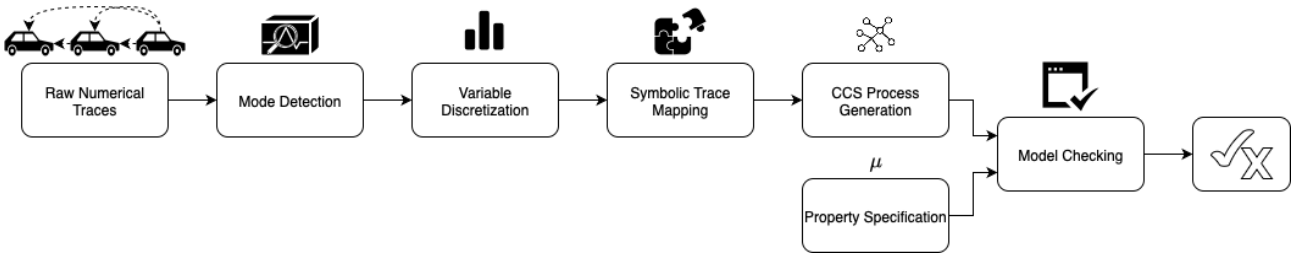

Figure 4: Complete Workflow of Our Methodology

To enable formal verification and behavioral reasoning, the pipeline transforms these numerical traces into symbolic process models. Figure 4 shows the workflow of our methodology. It outlines the complete transformation pipeline, starting from raw numerical traces and leading to a fully analyzable formal model.

The main stages of the methodology are as follows:

- **Operational Mode Identification**: The first step involves detecting the operational mode of the platoon system (mode 0, 1, or 2), which corresponds to different vehicle coordination strategies. This is achieved by analyzing the statistical profile of the leader vehicle over a short time window (e.g., average and standard deviation of acceleration) and comparing it with known mode signatures.

- **Discretization of Continuous Variables**: For each continuous variable, the numerical domain is partitioned into a finite number of disjoint intervals. Each interval is then associated with a symbolic label (e.g., low, medium, high). In particular:
    - Acceleration and speed are discretized based on percentile thresholds calculated from the trace itself.
    - Distance is discretized using predefined physical safety margins.
- **Mapping of Numerical Traces to Symbolic Traces**: Once the labels are defined, the entire numerical trace is transformed into a symbolic sequence, where each time instant is represented by a tuple of labels corresponding to acceleration, speed, and distance.
- **Formal Translation into CCS**: The symbolic trace of each vehicle is then translated into a process using the Calculus of Communicating Systems (CCS). Each symbolic step becomes an action in the CCS syntax. For cooperative scenarios, processes can also include synchronization points (e.g., sink, go) to model simultaneous progression among vehicles.
- **Specification of Properties in Mu-Calculus**: To enable formal reasoning, behavioral properties can be expressed using modal mu-calculus, a logic suitable for model checking over labeled transition systems. This allows verification of safety, liveness, or coordination constraints directly over the generated models.

This pipeline enables a systematic transition from raw numerical logs to a fully formal, analyzable representation of cooperative vehicular behavior.

5.1 Mode Detection

In cooperative vehicular systems such as platooning, vehicles can operate under different driving strategies, referred to as operational modes. Each mode defines a specific behavioral pattern for how the leader vehicle drives, which in turn influences the responses of all follower vehicles. In our framework, three operational modes are supported: mode 0, mode 1, and mode 2.

Automatic mode detection is a critical step in the modeling pipeline, as it directly influences the choice of discretization thresholds used in subsequent steps. Each mode corresponds to a different driving profile, and thus requires specific interval boundaries to accurately label numerical values.

Operational Modes Description

Each operational mode is designed to simulate a distinct type of leader behavior in the platoon:

- **Mode 0** involves a period of constant acceleration at the beginning of the simulation, followed by a smooth oscillatory motion. This mode represents a predictable and stable driving style, suitable for evaluating vehicle responses under regular conditions.
- **Mode 1** starts with a gradual linear increase in acceleration (a ramp phase), which smoothly transitions into the same oscillatory pattern used in mode 0. This mode models a more progressive, yet still regular, behavior of the leader.
- **Mode 2** is characterized by a periodic cycle alternating between phases of acceleration, short oscillations, and deceleration. This behavior repeats at fixed time intervals and simulates a more reactive or "aggressive" driving style, where the leader frequently accelerates and slows down.

These modes are not directly encoded in the trace files and must be inferred by analyzing the data itself — in particular, the behavior of the leader vehicle.

Statistical Signature Extraction

To automatically classify a trace into one of the predefined operational modes, we extract a statistical signature based on the behavior of the leader. Specifically, we compute the mean and standard deviation of the leader's real acceleration over an initial time window (typically the first 20 seconds of the trace).

This provides a simple yet informative numerical fingerprint of the leader's behavior at the beginning of the scenario, which is representative of the mode being simulated.

For each mode, reference signatures are precomputed using clean example traces. These signatures are stored and used for comparison during classification.

Mode Classification Algorithm

Mode classification is performed by comparing the signature of the input trace with the reference signatures for all known modes. We use Euclidean distance in the two-dimensional space defined by mean and standard deviation of the leader's acceleration.

The mode whose reference signature is closest to the input trace is selected as the most likely match.

This approach is intentionally simple and interpretable, ensuring that mode classification remains fast and explainable.

5.2 Trace Discretization

The second key step in our modeling pipeline involves transforming continuous numerical variables into discrete symbolic labels. This process, known as discretization, is essential for enabling formal reasoning over system behavior, as it allows us to define models over a finite and well-defined set of actions.

Discretization is applied to each relevant variable in the trace: acceleration, speed, and distance. The strategy differs depending on the type of variable and the operational mode detected in the previous step.

Discretization Objectives and Principles

The main goals of discretization are:

- To reduce the complexity of the raw traces while preserving essential behavioral differences.
- To map numeric ranges to interpretable qualitative categories.
- To enable symbolic modeling, where each trace step becomes a finite label used in formal processes.

The discretization ensures that:

- The full domain of each variable is covered (no value is left unlabelled).
- Each interval has a unique corresponding label.
- Labels are mutually exclusive, forming a partition of the domain.

Accelerations and Speeds: Percentile-Based Labeling

For both acceleration and speed, the discretization is based on the empirical distribution of values observed in each trace. We compute the following percentiles: 5th, 25th, 75th, and 95th, separately for each vehicle and for each variable, over a filtered subset of physically valid values (e.g., ignoring extreme or invalid samples).

This yields six possible symbolic labels:

- **non_physical_min** (below physical threshold)
- **extreme_low** (below 5th percentile)
- **low** (between 5th and 25th)
- **medium** (between 25th and 75th)
- **high** (between 75th and 95th)
- **extreme_high** (above 95th percentile)
- **non_physical_max** (above physical threshold)

Each raw value is mapped to its corresponding label depending on the calculated percentiles. Since different modes have different behavioral ranges, mode-specific thresholds are used.

Distances: Threshold-Based Labeling

Distance between vehicles is discretized using fixed predefined thresholds, based on the system's design constraints rather than distributional properties. In the platooning context, the nominal target distance between vehicles is 14 meters. Based on acceptable deviations, we define the following distance categories:

- **critical_low** (distance < 12.0 m) – critically short inter-vehicle distance
- **low** (12.0 m ≤ distance < 13.5 m) – short but acceptable distance
- **optimal** (13.5 m ≤ distance < 14.5 m) – ideal distance for platooning
- **high** (14.5 m ≤ distance ≤ 16.0 m) – slightly extended but still safe
- **critical_high** (distance > 16.0 m) – overly large inter-vehicle separation

This discretization captures meaningful deviations from the expected safe inter-vehicle distance and supports the detection of abnormal or unstable formations.

Mode-Specific Boundaries

While distance discretization uses fixed thresholds, acceleration and speed labeling is mode dependent. Each operational mode has its own statistical characteristics, which influence the percentile values. To ensure consistency, the boundaries used for discretization are:

- Automatically computed during preprocessing
- Stored per mode in a reference dictionary
- Used uniformly across all vehicles in each trace

This guarantees that the symbolic interpretation of high speed or medium acceleration, for example, is consistent with the mode's expected dynamics. After discretization, each time step of each vehicle's trace is represented as a symbolic tuple, and an example of this symbolic trace mapping is reported in Table 1.

Table 1: Example of symbolic trace after discretization of acceleration, speed, and inter-vehicle distance.

Time	Car ID	Acceleration	Speed	Distance to Preceding Vehicle
0.0	CAR_X	medium	low	nominal
0.1	CAR_X	high	medium	nominal
0.2	CAR_X	extreme_high	high	too_close
…	…	…	…	…

5.3 Formal Model Generation

This step generates a formal model by using the symbolic representations of the vehicular traces obtained in the previous discretization phase. For each vehicle involved in the platoon, the acceleration, speed, and distance variables are mapped into symbolic labels. These labels are then translated into CCS processes, representing the behavior of each vehicle over time.

The CCS model is defined over a finite set of symbolic actions, one for each discretized variable at each time point. Each process represents the behavior of a single vehicle, encoded as a sequence of synchronized or unsynchronized actions.

Symbolic Trace Encoding

Once the operational mode is detected and discretization bounds are applied, each row of the numerical trace is mapped to a symbolic triple of the form:

`[label_acceleration].[label_speed].[label_distance]`

where each label corresponds to the symbolic category assigned to the respective value. An example is shown in Table 2, representing a symbolic trace for one vehicle over three-time steps.

Table 2: Example of a discretized symbolic trace for a single vehicle.

Time	Acceleration	Speed	Distance to Preceding Vehicle
t_0	medium	low	optimal
t_1	high	medium	low
t_2	extreme_high	high	critical_low

Each row is translated into a CCS process step. The resulting process captures the step-by-step behavior of the vehicle, representing its acceleration, speed, and distance as a sequence of symbolic actions.

A separate CCS process is constructed for each vehicle. The process is defined as a chain of states, one per time instant. Each state executes a compound action (concatenation of labels), and transitions to the next process in the sequence.

Let us consider vehicle `CAR1`, with its symbolic trace shown in Table 2. In the following we report the corresponding CCS specification.

```
proc CAR1_T0 = acc_medium.speed_low.dist_optimal.CAR1_T1

proc CAR1_T1 = acc_high.speed_medium.dist_low.CAR1_T2

proc CAR1_T2 = acc_extreme_high.speed_high.dist_critical_low.nil
```

This process models the complete evolution of the vehicle from start to termination, based solely on its behavior in the trace.

Synchronization of Cooperative Vehicles

In cooperative settings such as platooning, vehicles operate in coordination. To model such behavior, we introduce a synchronization mechanism using dedicated CCS actions (sink, go). These actions are used to align the execution of different vehicle processes at each simulation time step.

To implement synchronization:

- Each vehicle process emits a synchronization action (`sinkX`, `goX`) before progressing.
- An auxiliary process `SINK` coordinates all `sink/go` actions in order and loops infinitely.
- A top-level process `ALL` puts all vehicle processes in parallel and hides synchronization actions.

Example:

```
proc SINK = 'sink0.'sink1.'sink2.'go0.'go1.'go2.SINK

proc ALL = (CAR3_T0 | CAR4_T0 | CAR5_T0 | SINK) \ {sink0, sink1, sink2, go0, go1, go2}
```

This structure ensures that vehicles advance in lock-step, reflecting real-time coordination as in the simulated platoon. The generated CCS models can now be used for formal verification. Properties describing safe distances, behavioral regularities, or synchronization invariants can be expressed in modal mu-calculus and checked against the model. This modeling approach enables a full pipeline from raw simulation data to verifiable formal specifications.

5.4 Property Specification

Cooperative vehicular systems such as platooning rely on strict coordination and safe distance keeping between vehicles. Given the high degree of automation involved, ensuring that the system adheres to behavioral constraints is essential, particularly in the presence of faults or attacks.

Compared to black-box machine learning approaches, the use of explicit symbolic properties allows for interpretable diagnostics and verifiable safety guarantees. By inspecting whether each property holds over the

system traces, engineers can identify precise points of failure, explain them in domain-specific terms, and define corrective measures that are formally justified.

However, raw numeric traces alone are often difficult to interpret due to signal noise and temporal variability. To overcome this, we apply a symbolic discretization process to map continuous variables — such as acceleration, speed, and inter-vehicular distance — into qualitative labels. This abstraction enables clearer analysis and supports the definition of properties over symbolic traces that highlight behaviorally meaningful patterns. In this way, symbolic modeling acts as a bridge between low-level data and high-level reasoning, enabling both formal verification and practical anomaly detection.

To better understand when and how such violations can occur, we analyze the evolution of key behavioral variables — acceleration, speed, and inter-vehicular distance — using both raw data and their symbolic (discretized) counterparts.

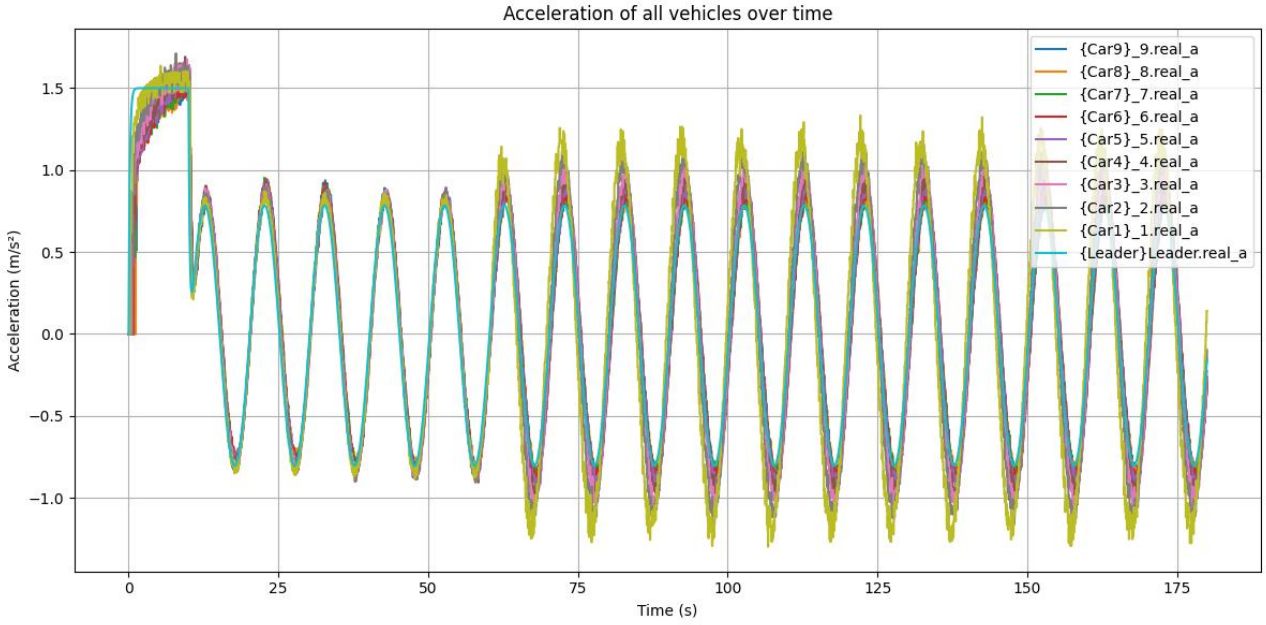

Figure 5: Acceleration profiles of the platoon under an example attack.

Errore. L'origine riferimento non è stata trovata. shows the acceleration profile of all vehicles in a scenario affected by an example attack. Around timestamp 60 seconds, `Car1` exhibits an abnormal drop in acceleration, while the `leader` vehicle maintains a stable behavior. This divergence is indicative of an unintended or malicious event. Such anomalies motivate the definition of high-level properties that can be used to detect, explain, or prevent undesired behaviors.

From the analysis of both normal and attack scenarios, we derive a set of relevant properties. These properties are described informally in natural language and reflect the key safety and coordination rules that the system should follow. **Errore. L'origine riferimento non è stata trovata.** shows the derived properties.

Table 3: Set of symbolic properties for safety derived from trace analysis.

Property ID	Description
P1 – Distance Violation	A vehicle should not remain in a *critical low* distance state for more than five consecutive time steps.
P2 – Acceleration Misalignment	If a follower vehicle changes its acceleration and the leader remains stable, this must not cause the distance to become *critical low*.
P3 – Consistent Speed Transitions	Transitions from high to low speed must be followed by a corresponding reduction in acceleration.
P4 – Synchronous Behavior	All vehicles in the platoon should undergo synchronous or coordinated changes in acceleration under normal operating conditions.
P5 – Anomaly Detection	A sudden and isolated change in acceleration of a follower vehicle, not justified by traffic or leader behavior, must be flagged as potentially unsafe.

For instance, in one of the analyzed attack traces, `Car1` abruptly changes its acceleration at timestamp 60, while the `leader` remains in a steady state. This causes the distance to shrink dangerously, violating Property **P2** – Acceleration Misalignment. This concrete example shows how symbolic representations help detect subtle coordination failures before they escalate into safety hazards.

These properties capture both invariant requirements (e.g., **P1**) and causal dependencies (e.g., **P2** and **P5**) observed in the system's behavior. While we express them in informal terms, they are grounded in the symbolic traces derived from real simulations and serve as the basis for further formal verification and anomaly detection tasks. These properties are designed to be generic and data-driven: they apply to any discretized vehicular trace and can be used both for validation during model checking and as diagnostic rules during trace inspection. In particular, these properties could also be embedded into runtime monitors, enabling automatic validation of vehicle behavior during simulations or live operations. Their symbolic structure ensures that the conditions they capture are both interpretable and enforceable, contributing to the transparency, reliability, and trustworthiness of cooperative driving systems.

We resort to the Concurrency Workbench of New Century (CWB-NC) [Clev96] formal verification environment.

5.5 Explainability

The proposed methodology enhances explainability by transforming raw numerical traces into symbolic sequences. Each continuous feature — acceleration, speed, and inter-vehicular distance — is mapped to a finite set of semantically meaningful labels (e.g., critical_low, optimal, extreme_high) using either percentile-based discretization or rule-driven thresholds. This transformation bridges the gap between low-level sensor data and high-level reasoning, enabling domain experts to interpret vehicle behavior without inspecting raw numeric logs. Compared to black-box machine learning approaches, this symbolic abstraction provides a transparent and interpretable representation of system dynamics. Abnormal behavior patterns — such as sudden changes in acceleration or prolonged critical distances — can be detected and explained in human-readable terms, supporting both offline analysis and operational monitoring.

Moreover, behavioral constraints can be formalized directly over symbolic labels, using either informal rules or temporal logics such as μ-calculus. This provides several advantages:

- *Traceability*: property violations can be precisely located in the trace and linked to concrete symbolic events.
- *Debugging and validation*: engineers can analyze transitions over time (e.g., a change from medium to extreme_low acceleration) to understand deviations from nominal behavior.
- *Property specification*: formulating behavioral properties is facilitated, as symbolic labels provide a meaningful vocabulary for expressing conditions (e.g., “a vehicle should not remain in critical_low distance for more than five steps”).

5.6 On-line test generation

In addition, the symbolic representation enables automatic runtime monitoring. Rule-based monitors can operate over symbolic sequences and detect potential violations of coordination or safety constraints in real time. For example, a simple automaton could trigger a warning when a follower vehicle changes acceleration while the leader remains stable and the distance drops to critical_low. Finally, this framework is compatible with hybrid approaches. Statistical anomaly detectors or learning-based classifiers can operate in tandem with symbolic rules. For instance, clusters of anomalous behavior identified by machine learning can be mapped back to symbolic traces for interpretation, validation, and formal verification.

By enabling interpretable diagnostics, trace-based reasoning, and formally verifiable properties, our method serves as a foundational tool for ensuring the safety and trustworthiness of cooperative vehicular systems.

6 Stochastic model checking for analysis on critical traces

The approach based on statistical model checking [Legacy10] uses the formalism of timed automata to model the platoon. We use the framework UPPAAL SMC [Behr04].

A timed automaton consists of a set of states and transitions, and real-valued variables for measuring time between transitions, named *clocks*. The general form of a transition consists of a guard, a synchronization and an assignment to clocks and variables. States are also named locations. Invariants can be added to locations to specify timing constraints on leaving the locations. The behavior of the automaton evolves from the initial location. One of the main advantages of using statistical model checking is that we can study the behavior of the system under a wide range of parameters. For instance, we can assume the acceleration time of a vehicle being drawn from a uniform distribution with lower and upper bound of 5 and 10, respectively. Such an approach allows us to test the probability of a certain property being true in case – generally speaking – the accelation of the car is in the continuous range from 5 to 10.

Consider the simple system, composed by a car and a driving pattern. The system moves according to a given driving scenario and imposes the acceleration on the car.

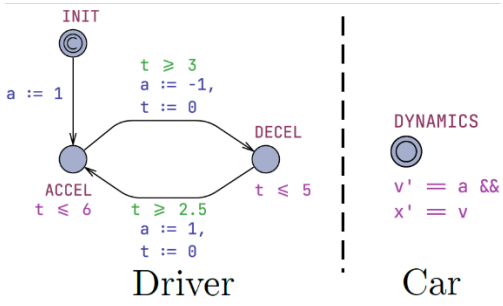

Figure 6: A simple car system

The car is represented by the automaton `Car`. The acceleration imposed on Car is represented by the automaton `Driver`. The acceleration is defined by a global variable `a`, set by the `Driver`. The clock variable `t` defines interleaving time between transitions.

The automaton `Car` consists of a single location DYNAMICS. The car's speed and position are modeled by two clocks `x` and `v`, which are used to model the physics via the Lagrangian derivative notation. In particular, the state has an invariant that consists of the conjunction of the following two formulae: the acceleration is always equal to the derivative of the velocity; and the velocity is always equal to the derivative of the position.

The automaton `Driver` consists of three locations (INIT, ACCEL and DECEL). The driver's behavior is modeled as follows. INIT is the initial location. The first transition is enabled and set the acceleration to

`a` = 1 m/sec^2. The location INIT is marked with a C that stands for *committed*, time is not allowed to pass when a process is in a committed location. It forces the automaton to transition to state ACCEL and assigns `a:=1` immediately.

Then, the invariant (`t<=6`) assigned to location ACCEL and the guard (`t>=3`) assigned to the transition exiting such location, guarantee that the transition between ACCEL and DECEL is executed in the time range `[3,6]`. The execution of the transition assigns a new value equal to `-1` m/sec^2 to the global variable `a (a:=-1)` and the clock `t` is reset to `0`. A similar behavior is exhibited when the system is in location DECEL (state invariant (`t<=5`); and guard (`t>=2.5`)). The Driver alternates an acceleration period of length T_A in the range `[3,6]` with `a` = 1 m/sec^{2}and a deceleration period of length T_D in the range `[2.5,5]` with `a` = -1 m/sec^2.

During the evolution of the systems, whenever a transition is executed, all clocks are evaluated. In particular, all clocks are updated, and the car's automaton updates the velocity and position of the car.

6.1 Modeling a platoon

Let us consider a reduced version of the platoon with the leader only 3 follower cars, we model the system using 7 templates of automata, as shown in Figure 7.

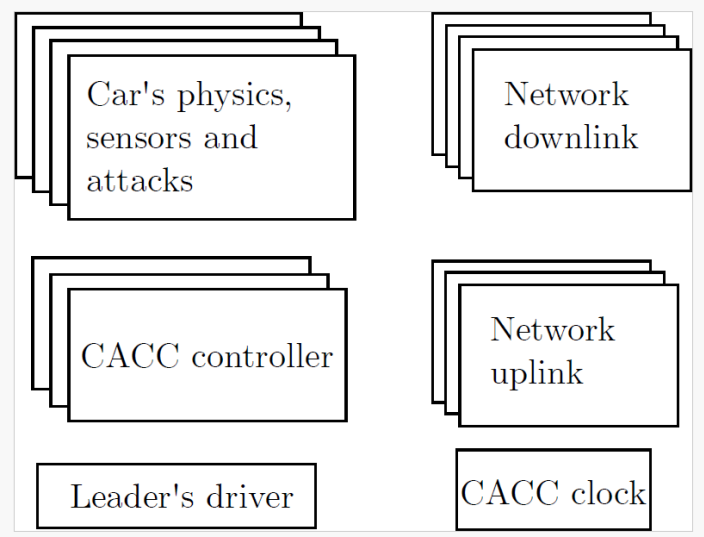

Figure 7: Components of a platoon.

Templates of automata

Car's physics, sensors and attacks, this template implements the physics and the attacks on a single car. We have 4 instances of this template – one for the leader and three for the followers. Shown in Figure 8.

Leader's driver, this template implements the driving behavior of the leader car.

CACC controller, this template implements the CACC controller, there are 3 instances of this template – one for each follower.

CACC clock, this automaton ties together the CACC instances.

Network uplink and downlink, these templates are used to implement the network delays between the CACC controllers and the platoon.

Modeling attacks. The system model is extended with the injection of attacks by adding the effects of the attacks on the behaviour of the system. We consider attacks that add spurious signals to data sent by the car to other vehicles (V2V platoon) or to controller at the edge (V2E platoon). Attacks that affect data sent back by the controller to cars can be modeled in a similar way.

Clocks modeling data that have been altered by an attack are used to model attacks in the platoon. There are six clocks for each car:

`a_p[i], v_p[i], x_p[i]` that represent the real physical quantities; and

`a[i], v[i], x[i]` that are used to model attacks in the platoon. They represent the value actually sent by the car to the other vehicles or to the remote controller.

The attack is characterized by a starting time and by an amplitude parameter. Let `attack_time` be the time at which the attack starts.

The global variable `A` represents the data alteration attack with amplitude `A`.

Let `N_CARS` be the number of cars in the platoon. A vector `crashed[N_CARS]` is used, whose i-th element is set to true after the i-th car has either rear-ended or has been rear-ended.

Let us consider the case of a low-frequency sinusoidal attack of 1 Hertz of frequency and of amplitude A in [-0.1, +0.1], applied to real acceleration value a(t) --- the coefficient $2\pi/10$ is the frequency expressed in radians per second. In the example, we assume the acceleration, velocity and position values are modified coherently, assuring that the relation x'=v, v'=a holds true. Additionally, for the velocity, a small bias coefficient $(5/\pi)A$ is introduced. The problem is made interesting by the signal's construction, which is designed to be difficult for an external observer to detect.

The `Car` model is shown in Figure 8. There are four states:

- WAIT. Location `WAIT` represents the car waiting for its time to start moving after `start_time` seconds. STOP ALL LOCAL CLOCKS is the state invariant in which all clocks modeling the physics are stopped by imposing their prime derivatives to zero, that is `x'==0 && v'==0 && etc`.

- DYNAMICS. Location `DYNAMICS` models the time evolution of the vehicle given the input signal desired acceleration `u`. NOMINAL DYNAMICS is the state invariant that models the physics of the running car. Being in location `DYNAMICS`, if `attack_time=+infinity` then no attack takes place as the guard `t_sim>=attack_time` of the transition from location DYNAMICS to DYNAMICS_ATTACK is always false. When an attack starts, such transition updates clocks according to the effects of the attack on velocity and position.
- DYNAMICS_ATTACK. Location `DYNAMICS_ATTACK` is the same as the former but with the attack. ATTACK DYNAMICS is the state invariant in which the clock rate of attacked values are altered according to the attack.
- CRASH. The location CRASH represents a crash with another vehicle STOP ALL LOCAL CLOCKS is the state invariant and it is the same as the location WAIT.

More details are reported in [Bern25].

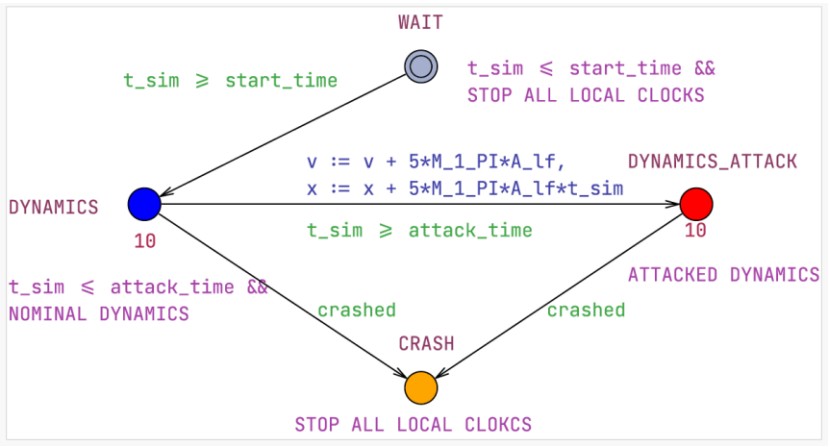

Figure 8: Car model in presence of possible attacks

6.2 Modeling system's properties

Using UPPAAL SMC we can compute, for example, the probability of an attack having a *significant* effect on the distance between two adjacent cars within the first 10 seconds. This might be useful in understanding how much time a threat detector might have to properly detect the threat and handle it.

We formalized the statement with the following formula, where the attack starts at 30s, and the property must hold for any pair of consecutive (adjacent) vehicles:

$$P\,(\forall t\colon 30\text{s} \le t \le 40\text{s} \;\Rightarrow \forall i\; \varepsilon_i < 0.15)$$

where ε_i represents the *difference* between the desired distance of 11 meters and the actual distance between vehicle *i* and vehicle *i-1* , t is the simulation time and imply operator ⇒ is used to check the condition $\forall i\; \varepsilon_i < 0.15$ within the time interval $t \in [30, 40]$.

In the UPPAAL query language, it becomes the following, where the desired distance is `11 meters` and `4 meters` is the length of a vehicle:

```
Pr[t_sim <= 40] ([] t_sim >= 30 imply
    forall(i : int[1, 3]) fabs(((x_p[i-1] - x_p[i] - 4) - 11) /11) < 0.15)
```

where:

- The `Pr[clock <= T](expr)` instructs the UPPAAL query engine to compute the probability of expr being true while the condition on a clock `clock` holds true
- The `[]` keyword says the property holds for the entire considered duration
- The imply keyword implements the *material conditional* ⇒
- The `forall(values) expr(i)` construct checks the `expr(i)` for all i in the list `values`
- The `fabs(x)` function computes the absolute value of a floating-point number

We made the simulation parametric in the attack amplitude A and we varied such parameter between –0.1 and 0.1 with step 0.002. We ran the query setting a confidence interval of 95%. The results are shown in the graph in Figure 9.

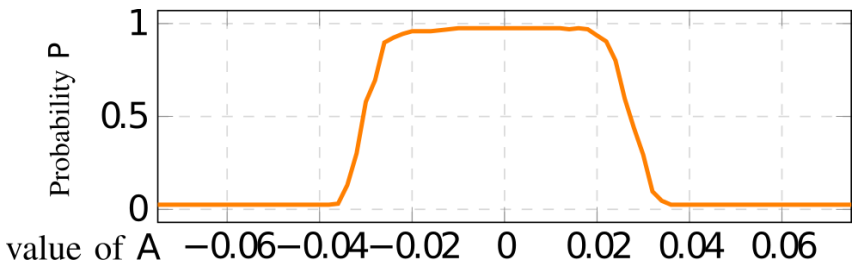

Figure 9: Probability of error distance $\varepsilon_i < 0.15$ within the time interval $t \in [30, 40]$ for any vehicle

For very small values of A the probability is 0, meaning that the attack has no effect in the short period; whereas larger values make the system more prone to leave the safe zone and can be easily detected.

7 Roadmap for future work

This deliverable reports on the methodology defined in WP3 to build formal models from behavioral traces and the identification of properties to detect attacks in the platoon. In the next tasks of WP3, the methodology will be validated using scenarios on simple cases; and successively, the methodology will be validated on the platooning use case.

Bibliography

[Clarke01] Clarke, E. M., Grumberg, O., and Peled, D. A. "Model Checking", MIT Press, 2001. [Online]. Available: http://books.google.de/books?id=Nmc4wEaLXFEC

[Mil89] Milner, R. "Communication and concurrency", ser. PHI Series in computer science. Prentice Hall, 1989.

[Koz83] D. Kozen, D. "Results on the propositional mu-calculus," Theor. Comput. Sci., vol. 27, pp. 333–354, 1983. [Online]. Available: http://dx.doi.org/10.1016/0304-3975(82)90125-6

[Clev96] R. Cleaveland and S. Sims, "The NCSU concurrency workbench," in Computer Aided Verification, 8th International Conference, Proc. CAV'96, New Brunswick, NJ, USA, 1996, pp. 394–397. [Online]. Available: https://doi.org/10.1007/3-540-61474-5n 87

[Legay10] Legay, B. Delahaye, and S. Bensalem. "Statistical model checking: An overview," in Runtime Verification (H. Barringer, Y. Falcone, B. Finkbeiner, K. Havelund, I. Lee, G. Pace, G. Ro¸su, O. Sokolsky, and N. Tillmann, eds.), (Berlin, Heidelberg), pp. 122–135, Springer Berlin Heidelberg, 2010.

[Behr04] Behrmann, G. and David, A. and Larsen, K. G. "A Tutorial on UPPAAL", Formal Methods for the Design of Real-Time Systems: International School on Formal Methods for the Design of Computer, Communication, and Software Systems, Bertinoro, Italy, September 13-18, 2004, Revised Lectures, Springer, 2004, pp. 200–236. DOI=10.1007/978-3-540-30080-9_7

[Bern25] Bernardeschi, C., Fagiolini, A., Pagani, D., and Quadri, C. "Statistical Model Checking for the Analysis of Attacks in Connected Autonomous Vehicles", Proc. 10th IEEE International Workshop on Cyber-Physical Systems Security, CPS-Sec 2025, Sept. 2025, France (to appear)