MISSIONE 4
ISTRUZIONE
RICERCA

FORESEEN

Deliverable D5.1: Report on the use case and requirement definitions

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 mEthodS for attack dEtEction in autonomous driviNg systems
PRIN 2022 PNRR

Project number: P2022WYAEW

CUP: I53D23006130001

Deliverable D5.1: Report on the use-case and requirement definitions

Project Start Date: 30/11/2023 Duration: 24 months
Coordinator: *University of Pisa*

Deliverable No	**D5.1**
WP No:	WP3
WP Leader:	RU-MI
Tasks:	T3.7 Validation through co-simulation on use case
Due date:	M 21-24
Delivery date:	February 23, 2026
Authors:	RU-MI, RU-PI, RU-PA, RU-MOL

Dissemination Level:

PU	**Public**	X
PP	**Restricted to other program 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
CAAL	Concurrency Workbench, Aalborg Edition
CAN	Controller Area Network
CCS	Calculus of Communicating Systems
CSV	Comma-Separated Values
CWB-NC	Concurrency WorkBench of the New Century
DSRC	Dedicated Short-Range Communications
FMU	Functional Mock-up Unit
GPS	Global Positioning System
P2P	Peer to Peer
V2E/V2N	Vehicle to Edge/ Vehicle to Network (synonyms for the purpose of this deliverable)
V2V	Vehicle to Vehicle

1. Introduction

This deliverable reports on the tools developed to support the project's methodology based on formal methods for detecting attacks in connected autonomous vehicle systems, and on the initial validation of this approach. In particular, the *Discretization Tool* generates abstract traces based on the results of Work Package 3 (Task 3.1), and the *CCS Converter Tool* generates CCS process inputs for the CWB-NC model checker. These tools are integrated into a comprehensive verification workflow and also support the parallel execution of model checker queries. The deliverable illustrates the strategy used to define the properties to be checked against the system model, based on the iterative process of comparing nominal and attack traces. A set of properties that could be used in the case of a platoon for verification in the CWB-NC framework are presented. These properties are based on inter-vehicle distance and the relationship between vehicle acceleration and inter-vehicle distance; they are formalized in mu-calculus and operate over discretized vehicle state traces. Finally, the deliverable reports on the analysis of modeling different road environments and discusses the conditions under which the methodology is applicable.

2. Tool to generate CWB-NC processes from traces

This chapter describes the solution implemented through two complementary Python tools that together form a complete transformation pipeline. The first tool, `discretization_with_mode_bounds.py`, addresses the problem of converting continuous sensor measurements into meaningful discrete symbolic labels. The second tool, `ccs_converter_sync_step_conf.py`, takes these symbolic representations and constructs formal process algebra specifications suitable for model checking with CWB-NC.

These tools are not merely technical utilities but represent a carefully designed abstraction strategy that preserves the essential behavioral characteristics of the physical system while making it amenable to exhaustive formal analysis. The pipeline handles multiple types of sensor data including acceleration, velocity, inter-vehicle distances measured by both radar and GPS, and relative velocity

differences between consecutive vehicles. Each of these continuous measurements must be discretized in a way that respects physical constraints, maintains semantic meaning, and produces finite-state models of manageable size for verification tools.

2.1. The Discretization Tool

The discretization tool serves as the first critical transformation step, converting the continuous numerical outputs from vehicular platoon simulations into discrete symbolic labels that can be processed by formal verification tools. Model checkers fundamentally operate on finite-state transition systems where each state represents a distinct symbolic configuration, making direct verification over continuous state spaces infeasible.

The discretization process must therefore carefully partition the continuous measurement domains into a finite set of meaningful symbolic categories that preserve the essential behavioral characteristics needed for verification.

The tool handles multiple types of measurements simultaneously, each requiring its own discretization strategy. Vehicle acceleration must be labeled in a way that distinguishes between different driving behaviors from emergency braking to aggressive acceleration, while respecting the physical limits of vehicle dynamics. Velocity measurements require categorization that captures whether vehicles are stopped, moving slowly, cruising, or traveling at high speeds. Inter-vehicle distances, measured by both radar and GPS sensors, need discretization schemes that reflect safety-critical thresholds where collision risk becomes significant or where platoon cohesion begins to deteriorate. Additionally, the relative velocity differences between consecutive vehicles provide crucial information about whether the following distances are increasing, decreasing, or remaining stable.

A sophisticated feature of this tool is its adaptive threshold mechanism. Rather than using fixed discretization boundaries that might not align well with the actual data distribution, the tool first performs operational mode detection to identify which operational regime the simulation data represents. Different operational modes might reflect different control parameters, environmental conditions, or mission profiles, each producing characteristic distributions of acceleration, speed, and distance measurements. Once the operational mode is identified, the tool applies mode-specific

percentile-based thresholds that have been learned from reference datasets. This ensures that discretization boundaries are appropriately positioned relative to the typical behavior in each mode, producing more meaningful and consistent symbolic abstractions.

The input format for the discretization tool consists of CSV files generated by the simulation environment, with columns following specific naming patterns that identify both the vehicle and the measurement type. Each row represents a timestep in the simulation, typically at 0.1-second intervals, containing columns with names like `{Leader}.LeaderInstance.acceleration`, `{F1}.CarInstance_1.speed`, `{F2}.CarInstance_2.real_x` for radar position measurements, and `{F3}.CarInstance_3.position_x` for GPS position data. The tool automatically discovers all vehicles present in the dataset by parsing these column names, then processes each vehicle's trajectory independently while also computing relative measurements between consecutive vehicle pairs.

Operational Mode Detection

The operational mode detection mechanism represents a key innovation in making the discretization process adaptive rather than fixed. The system learns characteristic signatures from reference files representing different operational modes, where mode_0 might represent baseline operational conditions, mode_1 an alternative control strategy, and mode_2 yet another configuration with different parameters. For each reference mode, the system computes percentile-based thresholds at the 5th, 25th, 75th, and 95th percentiles for both acceleration and velocity measurements across all vehicles. These percentiles define the boundaries between symbolic categories in a data-driven way that respects the actual distribution of measurements in each operational regime.

When processing new simulation data, the tool first analyzes the input to infer which operational mode it most closely resembles by comparing its statistical characteristics against the learned signatures. This inference is performed automatically, without manual specification of the mode. Once the mode is determined, the tool applies the corresponding mode-specific discretization thresholds, ensuring that the symbolic labels have consistent meaning across different runs of the same operational mode while allowing different modes to use different boundaries. This adaptive approach is particularly valuable when verifying systems under varying conditions or when comparing behavior across different control strategies.

Discretization Schemes

The discretization schemes for different measurement types reflect both the physical semantics of each variable and the requirements of formal verification. For acceleration, the tool produces eight distinct symbolic categories that span the full range of physically realistic vehicle behavior. The percentiles p5, p25, p75, and p95 are computed from the detected operational mode's reference data, ensuring that the boundaries adapt to the characteristic acceleration patterns of each mode. The physical constraints are hardcoded at -4.0 m/s^2 for the minimum acceleration, representing the emergency braking limit of typical vehicles, and +3.0 m/s^2 for the maximum acceleration, representing realistic vehicle acceleration capability. Values outside these bounds are flagged as non-physical, which can indicate sensor errors or simulation artifacts that require attention.

More details on the detector and discretization schemes are available in the previous deliverables.

2.2. The CCS Converter

The second tool in the pipeline transforms the symbolically labeled CSV data into formal CCS (Calculus of Communicating Systems) specifications that can be loaded into the CWB-NC model checker. CCS is a well-established process algebra specifically designed for describing concurrent systems where multiple processes execute in parallel and synchronize through shared actions. This mathematical formalism is particularly well-suited for modeling vehicular platoons because each vehicle can be represented as an independent process that evolves over time while synchronizing with other vehicles through coordination actions.

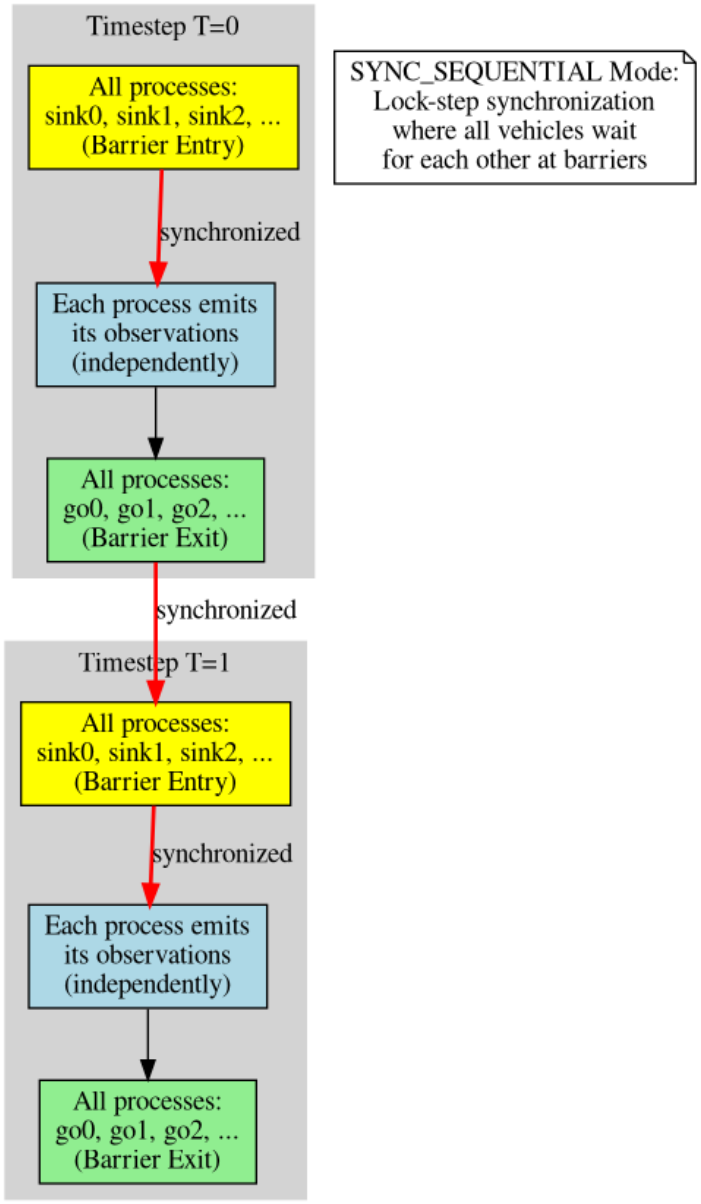
The converter provides extensive configuration capabilities that allow tailoring the generated models to specific verification needs and computational constraints. A fundamental design choice is which observable measurement types to include in the formal model. Configuration flags at the top of the tool's source code enable or disable acceleration labels, speed labels, radar distance measurements, GPS distance measurements, and relative velocity differences independently. This selective inclusion serves multiple purposes: reducing the state space by excluding unnecessary observables, focusing the model on measurements relevant to the properties under verification, and controlling the level of abstraction in the behavioral representation. For instance, if verifying a property that only concerns

relative distances between vehicles, one might disable acceleration and speed labels to simplify the model.

Beyond label selection, the converter implements sophisticated temporal processing to further reduce the state space while preserving essential temporal behavior. The tool can subsample the labeled time series by selecting only every N-th timestep, dramatically reducing the number of discrete time points in the formal model. Additionally, it can skip an initial portion of the data to eliminate transient startup behavior where vehicles are accelerating from rest to their cruising speeds. These temporal reductions are crucial for making verification computationally tractable, as the state space of the resulting model grows with both the number of timesteps and the number of distinct labels at each timestep.

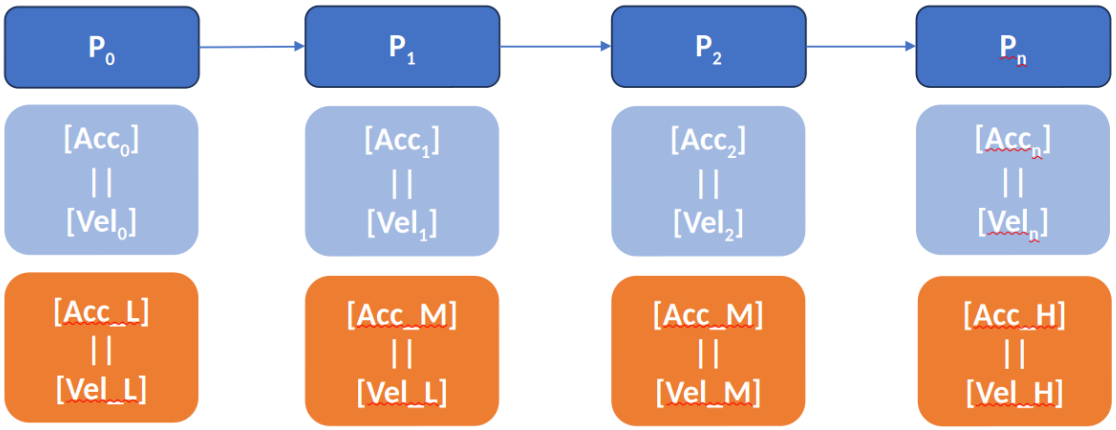
Synchronization Mechanisms

One of the most important design decisions in constructing formal models of concurrent systems concerns how to represent the coordination and timing relationships between parallel processes. The CCS converter supports three distinct synchronization strategies, each appropriate for different modeling assumptions about how vehicles coordinate their actions in the platoon. This section describes the implementation of the mechanisms already described in the Deliverable D4.1.



The sequential synchronization strategy, SYNC_SEQUENTIAL, implements lock-step coordination where all vehicles must synchronize at explicit barriers before and after each timestep. In this model, each vehicle first performs a synchronization action labeled `sink{X}` where X is the vehicle's index, effectively implementing a barrier entry. All vehicles must complete their sink actions before any can proceed. After synchronization, each vehicle emits its observable actions independently, then all vehicles perform another synchronization action labeled `go{X}` to exit the barrier. This pattern repeats at each timestep, ensuring that all vehicles advance through time in perfect coordination. The sequential strategy is appropriate for modeling discrete-time control systems where all vehicles sample sensors and compute control actions simultaneously at fixed time intervals, such as in time-triggered control

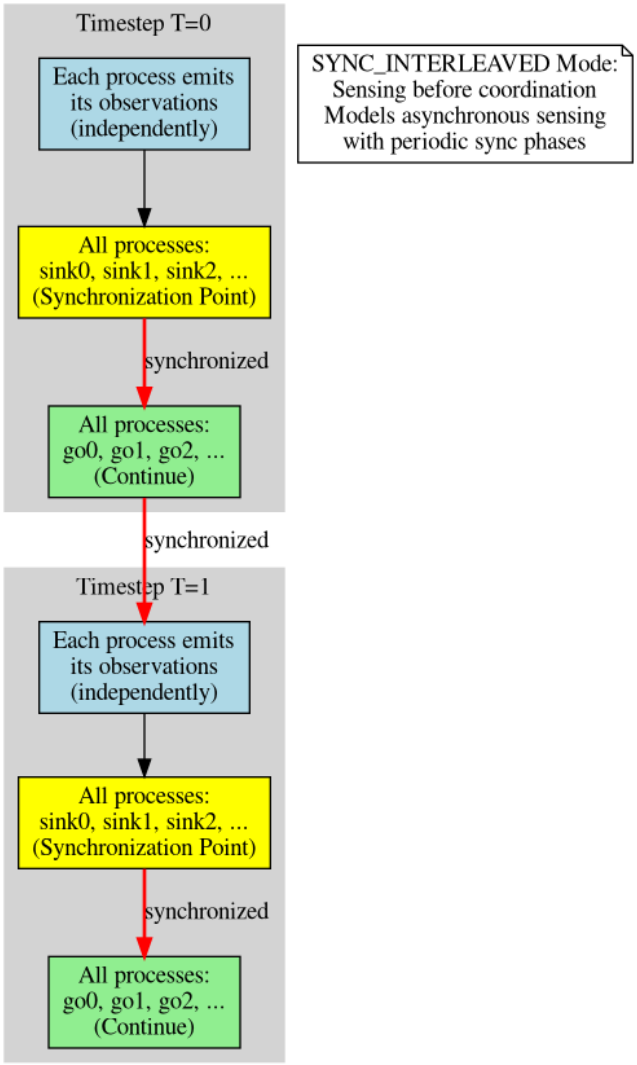
architectures or broadcast-based coordination protocols. Each vehicle is represented as a sequence of states evolving over time. Each time step is a state, labeled with discretized acceleration and velocity. The model is a sequence of these states, using parallel composition of features + sequential composition in time. As shown in the figure below.

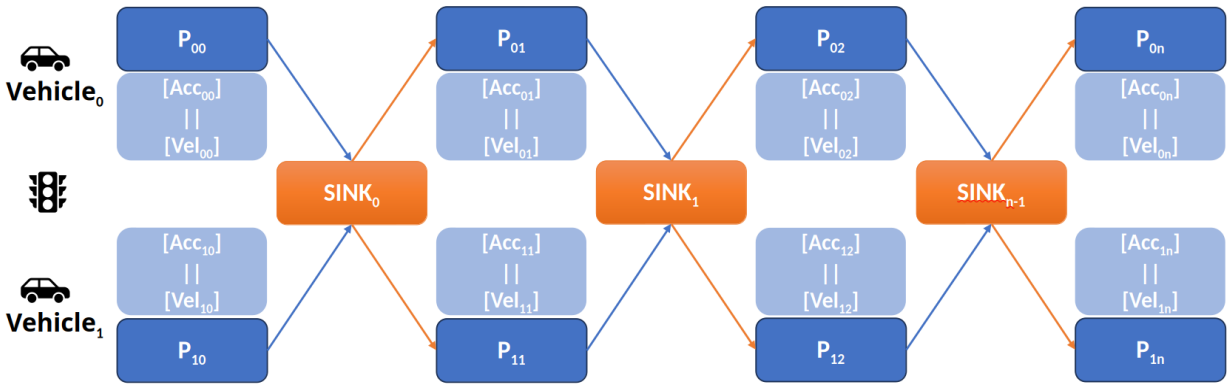


The interleaved synchronization strategy, SYNC_INTERLEAVED, represents a different coordination pattern where vehicles first emit their observations asynchronously, then synchronize before proceeding to the next timestep. Each vehicle emits its observable actions independently, then performs the `sink{X}` synchronization action, followed by the `go{X}` action to advance. This models scenarios where sensing happens asynchronously as data becomes available from sensors, but vehicles then coordinate at periodic intervals to exchange information or make collective decisions.

This strategy is useful for modeling systems with asynchronous sensing but periodic communication phases.

In such a mode, we combine all vehicle models in parallel. We introduce a sink process to synchronize all vehicles at each step. We ensure all vehicles “tick” together, thus representing platoon coordination.



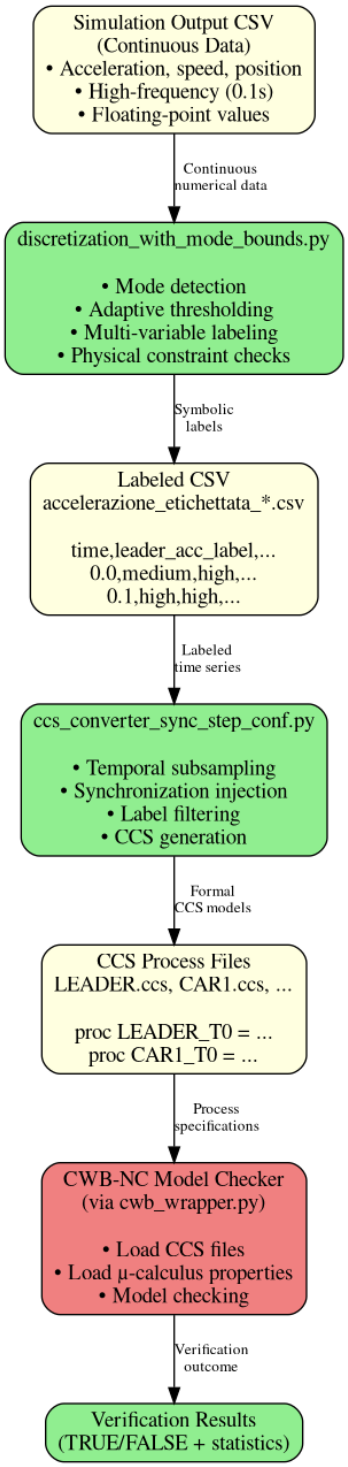


Process Generation and Output

The converter generates one CCS file for each vehicle, where each file contains a chain of process definitions representing that vehicle's behavior over time. Process names follow the pattern `{VEHICLE}_T{timestep}`, so the leader vehicle's processes are named LEADER_T0, LEADER_T1, LEADER_T2, and so forth. Each process definition consists of a sequence of action labels followed by a transition to the next process in the chain. At each timestep, the process may perform synchronization actions like sink0 or go0 depending on the configured synchronization mode, followed by observable actions representing the vehicle's current acceleration label, distance measurements to other vehicles, and any other enabled measurement types. The process then transitions to its successor, representing the next timestep. The final process in the chain transitions to nil, indicating the termination.

The converter implements sophisticated column-matching logic to identify which distance and relative velocity columns pertain to each vehicle, since these are relational measurements between vehicle pairs rather than individual vehicle properties. For each vehicle, it assigns a unique index that appears in its synchronization labels — the leader vehicle is assigned index 0, Car1 receives index 1, Car2 receives index 2, and so forth. This indexing scheme enables compositional verification where the complete system model is constructed as the parallel composition of all individual vehicle processes, with matching synchronization labels causing those processes to coordinate at the appropriate points in time.

The output directory contains one CCS file per vehicle, with filenames like LEADER.ccs, CAR1.ccs, CAR2.ccs, and so forth. Each file is self-contained with the complete process chain for that vehicle. Users invoke the converter by specifying the input labeled CSV file, the output directory for CCS files, and optional parameters for subsampling and row skipping. The tool provides informative console output indicating the discovered operational mode, the number of rows before and after each temporal processing stage, and the vehicle-to-index mapping used for synchronization labels.


Simulation Output CSV
(Continuous Data)
• Acceleration, speed, position
• High-frequency (0.1s)
• Floating-point values
Continuous numerical data
discretization_with_mode_bounds.py
• Mode detection
• Adaptive thresholding
• Multi-variable labeling
• Physical constraint checks
Symbolic labels
Labeled CSV
accelerazione_etichettata_*.csv
time,leader_acc_label,...
0.0,medium,high,...
0.1,high,high,...
Labeled time series
ccs_converter_sync_step_conf.py
• Temporal subsampling
• Synchronization injection
• Label filtering
• CCS generation
Formal CCS models
CCS Process Files
LEADER.ccs, CAR1.ccs, ...
proc LEADER_T0 = ...
proc CAR1_T0 = ...
Process specifications
CWB-NC Model Checker
(via cwb_wrapper.py)
• Load CCS files
• Load μ-calculus properties
• Model checking
Verification outcome
Verification Results
(TRUE/FALSE + statistics)

2.3. Integration and Application

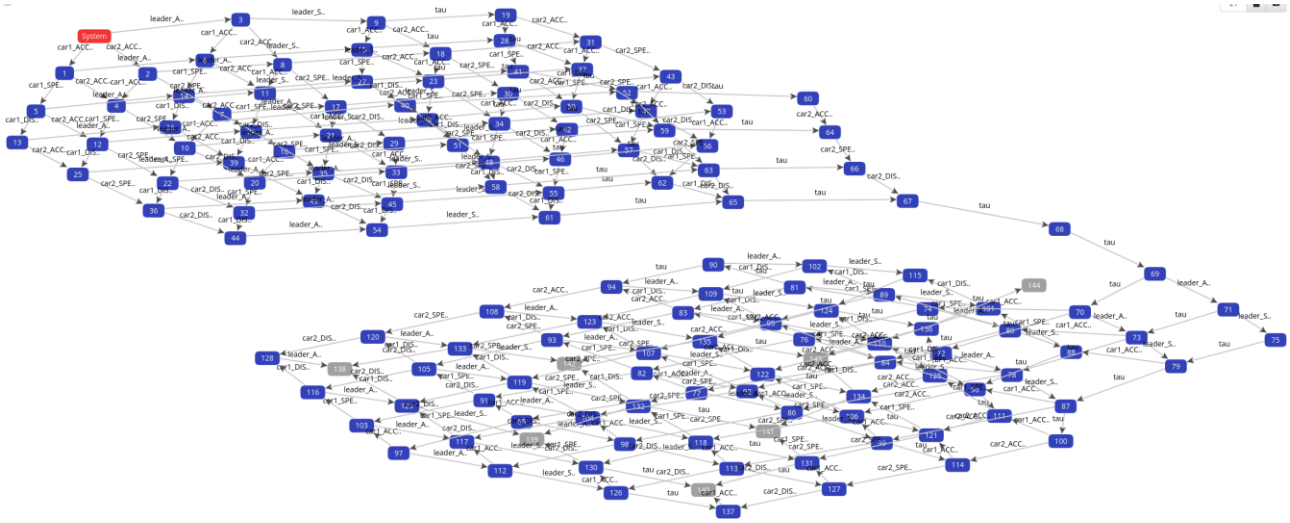
The power of these tools emerges not from their individual capabilities but from their integration into a complete verification workflow. The pipeline begins with simulation outputs from the vehicular platoon simulation environment, typically covering multiple scenarios representing different control parameters, environmental conditions, or cyber-attack configurations. Each simulation produces a CSV file containing continuous sensor measurements sampled at regular intervals throughout the simulation duration. These files serve as input to the discretization tool, which automatically detects the operational mode and applies appropriate adaptive thresholds to produce labeled symbolic data. The labeled CSV outputs then feed into the CCS converter, which applies temporal processing to reduce the state space and generate formal CCS specifications ready for model checking. The resulting CCS files, along with μ-calculus property specifications written separately by the verification engineer, are then loaded into the CWB-NC model checker to verify whether the system satisfies the desired properties.

From a theoretical perspective, the pipeline implements a classical abstraction transformation from infinite continuous state spaces to finite discrete state spaces that can be exhaustively explored by model checking algorithms.

2.4. CAAL traces compatibility

The aforementioned CCS generator tool is also able to generate traces for the *Concurrency Workbench, Aalborg Edition* (CAAL) [Andersen, J.R., 2015]. A graphical tool similar to CWB-NC, albeit differing a little bit in the mu-calculus syntax for its queries. We have observed that the execution time of the same queries on the same automata is much slower in the CAAL and its web-based interface makes it difficult to automate batch execution of queries over multiple automata.

In any case, the CAAL provides a tool that allows the graphical exploration of the graph of possible evolution of the systems, for instance this is the graphical view of the interleaved synchronization scheme:



We can see how the various sensor readings can be visited in any order; however, there's a sort of *bottleneck* between the two time-step, through which the system is forced to transition. Thus, two clusters are visible: one for the first time tick T = 0 and one for the second tick T = 0.1.

3. Mu-calculus queries for the CWB-NC

The figure below, taken from the CWB-NC's manual [Cleaveland, 2000], is the grammar of the mu-calculus, already presented in the D4.1, specific to writing queries for the CWB-NC.

tokens

```
"[a-zA-Z][a-zA-Z0-9'_-]*" => Ident
```

grammar

muFile	:	*propDeclList*	
propDecl	:	**prop** *ident* **=** *prop*	
prop	:	**tt**	(True)
		ff	(False)
		Ident	(Proposition variable)
		not *prop*	(Negation)
		prop **\/** *prop*	(Disjunction)
		prop **/** *prop*	(Conjunciton)
		< *actSet* **>** *prop*	(Possibility)
		<< *actSet* **>>** *prop*	(Weak possibility)
		[*actSet* **]** *prop*	(Necessity)
		[[*actSet* **]]** *prop*	(Weak necessity)
		min *ident* **=** *prop*	(Least fix point)
		max *ident* **=** *prop*	(Greatest fix point)
		(*prop* **)**	(Parentheses)
		AG *prop*	(CTL operators)
		AF *prop*	
		A (*prop* **U** *prop* **)**	
		A (*prop* **W** *prop* **)**	
		EG *prop*	
		EF *prop*	
		E (*prop* **U** *prop* **)**	
		E (*prop* **W** *prop* **)**	
actSet	:	*act*	
		actList	
		- actSet	

lists

List name	May be Empty?	Open Delimiter	Separator	Close Delimiter	Items
actList	yes	EMPTY_STR	,	EMPTY_STR	*act*
propDeclList	no	EMPTY_STR	EMPTY_STR	EMPTY_STR	*propDecl*

priorities

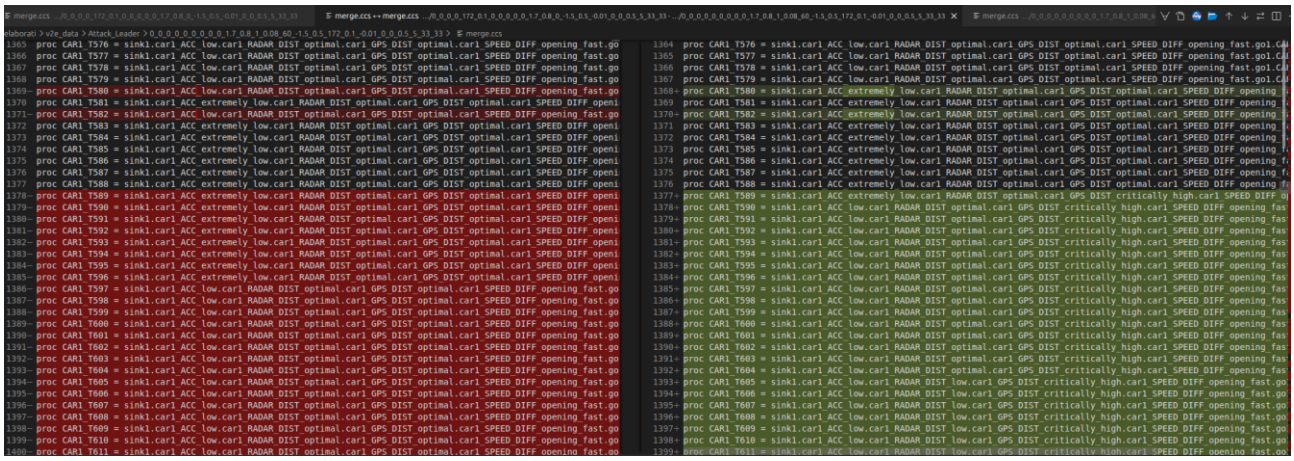
tt,ff > `[,[[,],]],<,<<,>,>>` > **not, AG, AF, EG, EF** >
/\ > \/ > =

4. Definition of properties to verify

To define the properties to check against our automata, we used an iterative process: we selected a pair of simulation traces, one with an attack and one without, while keeping all other parameters (e.g., driving scenario) unchanged. These two traces were converted to CWB-NC automata as usual.

Then, we made use of the diff tool to study the difference in behavior between the nominal automaton and the one under attack. The diff tool compares two files line by line and outputs the differences

between them. For better usability, we used the diff tool integrated in the Visual Studio Code, as shown in Figure below. This version of the tool provides a graphical front-end that makes it easy for the user to visualize the difference between the two files.



By studying the difference between the two, we derive properties to verify against the automata in the CWB-NC. Then we check them again against new pairs. If we find something that doesn't hold as expected, we modify the property appropriately.

4.1. For V2V

In the case of the vehicle to vehicle (V2V) communication – as previously explained in the past deliverables, – the CACC control law is distributed and the communication is P2P, that is, there's not a single entity that knows a reasonable truth about the current state of the platoon; and, in the case of checks that are run onboard the car, they do not have access to the full system state – for instance, the second car follower car in the platoon has access only to the physical values of itself, the car in front and the leader. In addition, we must rely only on these variables as reported by the car itself, and we cannot infer a different value by comparing the reading from other cars' sensors.

Thus, in generating the CWB-NC automaton, labels are restricted to values effectively available on board; similarly, properties for each vehicle are defined only over the labels associated with that vehicle.

4.2. For V2E

In the case of vehicle to edge (V2E) communication – as previously explained in the past deliverables, – the CACC control law is centralized, thus the centralized system at the network's edge has access to the complete platoon state and the same physical value may be reported from different sensors on different cars. For instance, the inter-vehicular distance can be reported by both cars, if 2 radars are employed, one at the front and one at the back.

Thus, we have a greater selection of physical values to generate the labels from and write the properties on.

Trace analysis reveals that attacks systematically impact inter-vehicular distance. In particular, one of the cars in the platoon stops following the one in front with the defined distance. However, due to the variability of the driving scenarios, it may happen that the inter-vehicular distance gets below the nominal threshold defined to produce the labels; thus, we must write the property in such a way that they tolerate the system going below the threshold for a limited time interval.

After some consideration and going through the iterative process, we came up with the following property:

```
prop is_d_safe_RC1 = max X = ([-]X /\ (
[car1_RADAR_DIST_low,car1_RADAR_DIST_critically_low][-][-][-][-][-][-][-][-][-][-][-][-][-][-]
[car1_RADAR_DIST_low,car1_RADAR_DIST_critically_low][-][-][-][-][-][-][-][-][-][-][-][-][-][-]
[car1_RADAR_DIST_low,car1_RADAR_DIST_critically_low][-][-][-][-][-][-][-][-][-][-][-][-][-][-]
[car1_RADAR_DIST_low,car1_RADAR_DIST_critically_low][-][-][-][-][-][-][-][-][-][-][-][-][-][-]
[car1_RADAR_DIST_low,car1_RADAR_DIST_critically_low][-][-][-][-][-][-][-][-][-][-][-][-][-][-]
[car1_RADAR_DIST_low,car1_RADAR_DIST_critically_low][-][-][-][-][-][-][-][-][-][-][-][-][-][-]
[car1_RADAR_DIST_low,car1_RADAR_DIST_critically_low][-][-][-][-][-][-][-][-][-][-][-][-][-][-]
[car1_RADAR_DIST_low,car1_RADAR_DIST_critically_low][-][-][-][-][-][-][-][-][-][-][-][-][-][-]
[car1_RADAR_DIST_low,car1_RADAR_DIST_critically_low][-][-][-][-][-][-][-][-][-][-][-][-][-][-]
[car1_RADAR_DIST_low,car1_RADAR_DIST_critically_low][-][-][-][-][-][-][-][-][-][-][-][-][-][-]
[car1_RADAR_DIST_low,car1_RADAR_DIST_critically_low][-][-][-][-][-][-][-][-][-][-][-][-][-][-]
[car1_RADAR_DIST_low,car1_RADAR_DIST_critically_low][-][-][-][-][-][-][-][-][-][-][-][-][-][-]
[car1_RADAR_DIST_low,car1_RADAR_DIST_critically_low][-][-][-][-][-][-][-][-][-][-][-][-][-][-]
[car1_RADAR_DIST_low,car1_RADAR_DIST_critically_low][-][-][-][-][-][-][-][-][-][-][-][-][-][-]
[car1_RADAR_DIST_low,car1_RADAR_DIST_critically_low][-][-][-][-][-][-][-][-][-][-][-][-][-][-]
[car1_RADAR_DIST_low,car1_RADAR_DIST_critically_low][-][-][-][-][-][-][-][-][-][-][-][-][-][-]
[car1_RADAR_DIST_low,car1_RADAR_DIST_critically_low][-][-][-][-][-][-][-][-][-][-][-][-][-][-]
[car1_RADAR_DIST_low,car1_RADAR_DIST_critically_low][-][-][-][-][-][-][-][-][-][-][-][-][-][-]
[car1_RADAR_DIST_low,car1_RADAR_DIST_critically_low][-][-][-][-][-][-][-][-][-][-][-][-][-][-]
[car1_RADAR_DIST_low,car1_RADAR_DIST_critically_low]ff
```

))

Mutatis mutandi, this property is repeated the same for the i-th car. This property checks that the car never presents a low following distance for more than 2 seconds.

5. Selected properties

Based on the iterative process of comparing nominal and attack traces, we have selected the following properties for verification in the CWB-NC framework. These properties are formalized in mu-calculus and operate over discretized vehicle state traces.

5.1. Distance-Based Properties

Radar Distance Safety - Basic Threshold

These properties are defined with the symbolic names is_d_safe_RC1 and is_d_safe_RC2.

Mu-calculus Formula:

```
max X = ([-]X /\ (
    [car*_RADAR_DIST_low,car*_RADAR_DIST_critically_low][-][-][-][-][-][-][-
][-][-][-][-][-][-][-]
    ... (20 consecutive patterns)
    [car*_RADAR_DIST_low,car*_RADAR_DIST_critically_low]ff
))
```

Meaning: The property verifies that the specified vehicle never maintains a radar distance in the "low" or "critically_low" range for 20 consecutive time steps, i.e., 2 seconds, assuming a sampling interval of 0.1 s. The greatest fixed-point operator (max X) combined with the pattern [-]X expresses the "always" temporal modality—the property must hold at every state in every execution trace.

Attack Detection: Violations indicate:

- Radar sensor false data injection (making distance appear larger than actual)
- Actuator command injection preventing proper deceleration
- Control law parameter tampering (e.g., reduced safety margins)

The 20-step window provides tolerance for brief legitimate proximity (driving fluctuations) while catching sustained dangerous spacing characteristics of attacks.

Radar Distance Safety - Critical Threshold

These properties are defined with the symbolic names is_d_qsafe_RC1 and is_d_qsafe_RC2.

Mu-calculus Formula:

```
max X = ([-]X /\ (
    [car*_RADAR_DIST_critically_low][-][-][-][-][-][-][-][-][-][-][-][-][-][-][-]
    ... (20 consecutive patterns)
    [car*_RADAR_DIST_critically_low]ff
))
```

Meaning: A stricter variant focusing exclusively on the "critically_low" threshold, allowing the platoon to drive closer together than the former.

Attack Detection: This property isolates the most severe safety violations. Combined with safety_distance_radar_ok_*, it enables differential diagnosis:

- Violation of _quasi_ok only: System at absolute safety limit but not yet in "low" zone
- Violation of both: Complete safety margin collapse, high attack likelihood

Radar Distance Safety with Speed Differential

These properties are defined with the symbolic names safety_distance_radar_ok_w_diff_speed_car1 and safety_distance_radar_quasi_ok_w_diff_speed_car1.

Mu-calculus Formula:

```
max X = ([-]X /\ (
    [car1_RADAR_DIST_low,car1_RADAR_DIST_critically_low][-][-]... (20
patterns)
    [car1_RADAR_DIST_low,car1_RADAR_DIST_critically_low][-
][car1_SPEED_DIFF_opening,car1_SPEED_DIFF_opening_fast]... (5 patterns)
))
```

Meaning: This property recognizes that temporarily low distances are acceptable when the relative velocity shows the vehicles are moving apart ("opening" or "opening_fast"). After 20 steps where distance is in the low range, the property allows an additional 5 steps of low distance provided the speed differential indicates the gap is increasing. This reflects normal platoon behavior where transient proximity is safe if vehicles are separated—the low distance will naturally resolve as they continue moving apart.

Safety Rationale: This property captures the physical reality that being closer than ideal spacing is tolerable when the trajectory shows imminent separation. Without considering speed differential, the basic distance properties might flag legitimate driving scenarios as violations. By incorporating relative velocity, we distinguish between:

- **Safe transient proximity**: Low distance + opening speed differential (acceptable, will self-correct)
- **Dangerous sustained proximity**: Low distance without opening speed differential (violation)

The property therefore reduces false positives during normal platoon dynamics such as minor speed oscillations, gentle acceleration of the front vehicle, or controlled following distance adjustments that temporarily reduce spacing before vehicles naturally separate.

GPS Distance Safety Properties

This family of six properties is defined with the symbolic names is_d_safe_GC1, is_d_qsafe_GC1, is_d_safe_GC2, is_d_qsafe_GC2, is_d_safe_w_dv_GC1, is_d_qsafe_w_dv_GC1.

These six properties are structural equivalents of the radar properties but operate on GPS-derived inter-vehicle distances instead of radar measurements. The key difference is the label position shift in the CCS state vector — GPS distance appears in one position to the right of radar distance.

Cross-Validation Mechanism: The parallel evaluation of radar and GPS properties enables sensor fusion attack detection:

- **Radar violation + GPS satisfied**: Potential radar-specific attack (jamming, spoofing)
- **GPS violation + Radar satisfied**: Potential GPS spoofing
- **Both violated**: Physical proximity danger or multi-sensor compromise
- **Neither violated**: Normal operation

This provides robustness against single-sensor attacks and helps disambiguate genuine safety events from cyber-attacks.

5.2. Acceleration-Distance Correlation Properties

Acceleration Control at Low Distance

This property is defined with the symbolic name: is acc safe w d unsafe RC1.

Mu-calculus Formula:

```
max X = ([-]X /\ (
  [car1_RADAR_DIST_low,car1_RADAR_DIST_critically_low][-][-][-][-][-][-][-
][-][-][-][-][-][-][-]
  ... (15 lines total = 1.5 seconds of low distance)

  [car1_RADAR_DIST_low,car1_RADAR_DIST_critically_low][-][-][-][-][-][-][-
][-][-][-][-][-][-][-][-car1_ACC_high,car1_ACC_critically_high]
  ... (15 lines checking high accel at end of each window)
  [car1_RADAR_DIST_low,car1_RADAR_DIST_critically_low]ff
```

Meaning: The property asserts that car1 never exhibits a 30-step sequence where: (1) the first 15 steps have low radar distance, and (2) the subsequent 15 steps maintain low distance while also exhibiting high positive acceleration.

Safety Rationale: High forward acceleration at close proximity violates the fundamental CACC safety principle: decelerate or maintain constant speed when following distance is minimal. This behavior should be impossible under correct control law execution.

Attack Detection: This property is particularly sensitive to:

- **Control law tampering**: Malicious modification of CACC algorithm gains or safety constraints
- **Command injection**: Forcing acceleration commands that override safety logic
- **Multi-stage attacks**: Phase 1 - sensor spoofing to make distance appear safe; Phase 2 - aggressive acceleration

The two-phase temporal pattern distinguishes attacks from transient anomalies (e.g., sensor noise causing brief erroneous readings).

Acceleration Control at Critical Distance

This property is defined with the symbolic name is_acc_safe_w_d_crit_RC1.

Mu-calculus Formula:

```
max X = ([-]X /\ (
    [car1_RADAR_DIST_critically_low][-][-]... (15 patterns)

[car1_RADAR_DIST_critically_low]...[car1_ACC_high,car1_ACC_critically_high]
(15 patterns)
))
```

Meaning: The most severe variant, requiring critically low distance throughout both phases.

Attack Detection: Under normal operation, this scenario should be physically unrealizable without collision. Any violation strongly indicates:

- **Simulation/replay attacks**: Attacker feeding pre-recorded safe sensor data while vehicle is actually in danger
- **Coordinated multi-vehicle attacks**: Both ego vehicle and front vehicle compromised to prevent collision during dangerous maneuver

Multi-Phase Acceleration Control

This property is defined with the symbolic name is_acc_safe_w_d_unsafe_RC1_v2.

Mu-calculus Formula:

```
max X = ([-]X /\ (
    [car1_RADAR_DIST_low,car1_RADAR_DIST_critically_low]... (15 patterns)

[car1_RADAR_DIST_low,car1_RADAR_DIST_critically_low]...[car1_ACC_high,car1_A
CC_critically_high] (10 patterns)

[car1_RADAR_DIST_low,car1_RADAR_DIST_critically_low][car1_SPEED_DIFF_opening
,car1_SPEED_DIFF_opening_fast]... (7 patterns)
))
```

Meaning: A three-phase property: (1) 15 steps of low distance, (2) 10 steps of high acceleration at low distance, (3) 7 steps where low distance persists despite opening speed differential.

Attack Detection: This extended temporal pattern captures attack sequences where:

1. Attacker causes aggressive acceleration at low distance (Phase 2)
2. This creates positive relative velocity as expected from physics (Phase 3)
3. However, distance fails to increase proportionally, indicating sensor manipulation or control lag

The property is robust against sophisticated attackers attempting to maintain plausible consistency across multiple sensor modalities (distance, velocity, acceleration). An attacker manipulating only one or two of these will violate the three-phase invariant.

6. Modeling of different road environments

This section presents the implementation and validation of a wheel slip model integrated into a vehicle platoon co-simulation system. The model incorporates a brakeless wheel component positioned between the controller and the vehicle dynamics model to enhance the realism of vehicle behavior under varying road surface conditions.

With the modeling that will be introduced, it is possible to reuse the methodology already employed in this project to study the behavior of the platoon under attack – even in slippery conditions – and to identify formal properties from the simulation traces.

In addition, some considerations on how to model the aerodynamic effects of windy conditions are presented in the last subsection.

6.1. Model Architecture

The enhanced vehicle model integrates a wheel slip component between the controller and the bicycle model. The controller computes the torque τ according to the relationship:

$$\tau = F \times R$$

Where F represents the force and R denotes the wheel radius. This torque is directly applied to the axle.

The simulation employs a vehicle mass of 2 Mg, with an axle load of 1 Mg distributed to the rear axle. The normal force F_z acting on the wheel is computed as the product of the axle load and gravitational acceleration.

The wheel slip behavior is modeled using the Pacejka Magic Formula [Bakker et al, 1987], which establishes the relationship between the longitudinal tire force F_x, the slip ratio κ and the normal force F_z:

$$F_x = f(\kappa, F_z) = F_z \cdot D \sin(C \arctan[B\kappa - E(B\kappa - \arctan(B\kappa))])$$

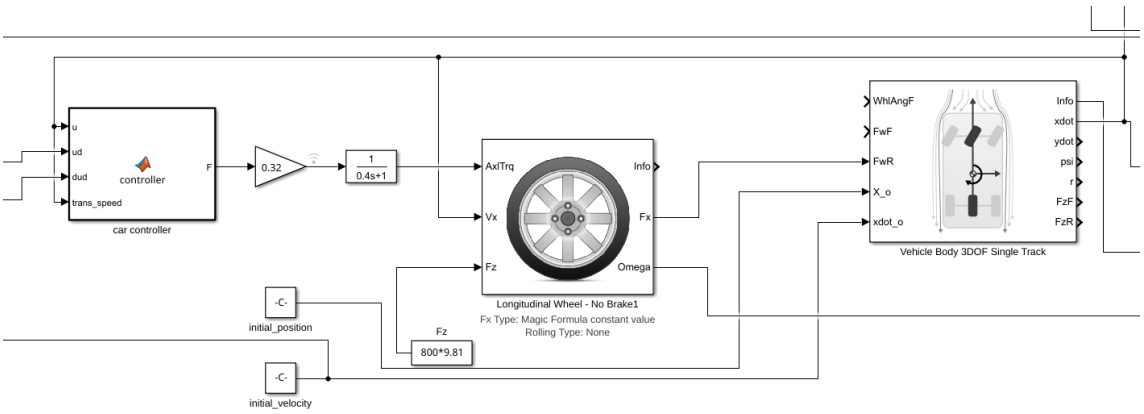
Where B, C, D, and E are empirical coefficients that characterize the tire-road interface properties.

The Magic Formula coefficients vary according to the road surface condition, as presented in Table below.

Table 1: Magic Formula coefficients for different road surfaces

Surface	B	C	D	E
Dry tarmac	10	1.9	1.00	0.97
Wet tarmac	12	2.3	0.82	1.00
Snow	5	2.0	0.30	1.00
Ice	4	2.0	0.10	1.00

The model of the wheels and these values are taken from the Powertrain Blockset of MATLAB Simulink [MathWorks, 2025]. The resulting Simulink model is:



Initially, we had to reduce the simulation step to 0.001 due to numerical issues in the simulation. However, utilizing the ode14x solver enabled stable simulation execution with a time step of 10 ms (0.01 s).

The four Magic Formula coefficients (B, C, D, E) were implemented as configurable parameters within the model, with dry tarmac values established as defaults. This parameterization facilitates user-defined surface conditions during co-simulation when the model is exported as an FMU. These modifications were consistently applied across both follower and leader vehicle models.

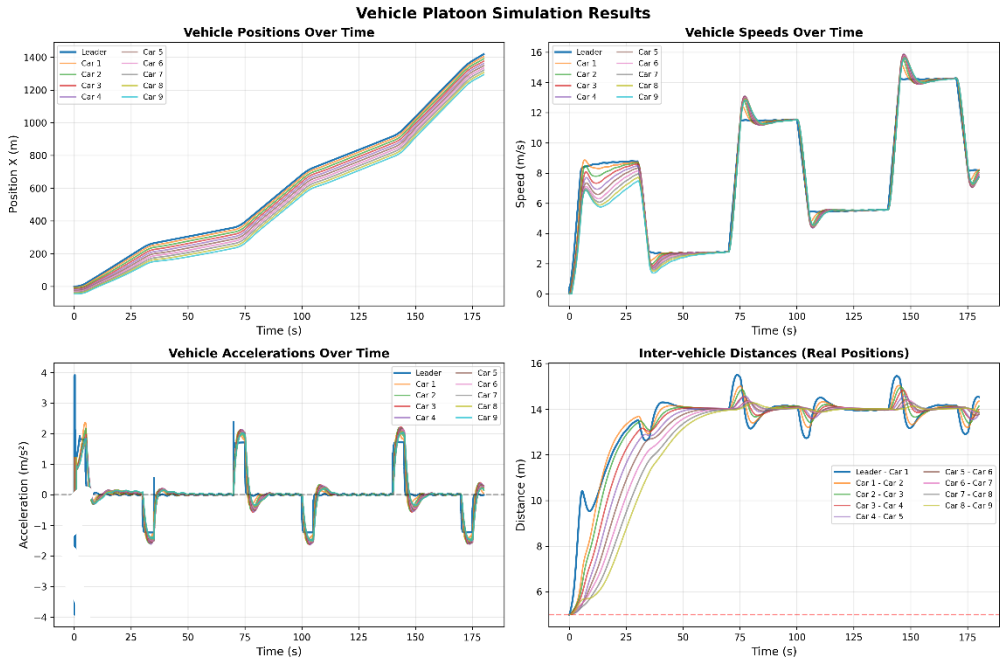
6.2. Co-Simulation Integration

Integration of the enhanced FMU into the platoon co-simulation environment revealed numerical instability in vehicles positioned at the fourth position and beyond in the platoon formation. We determined that a co-simulation time step of 10 ms (0.01 s) was insufficient to guarantee numerical stability for this class of simulation.

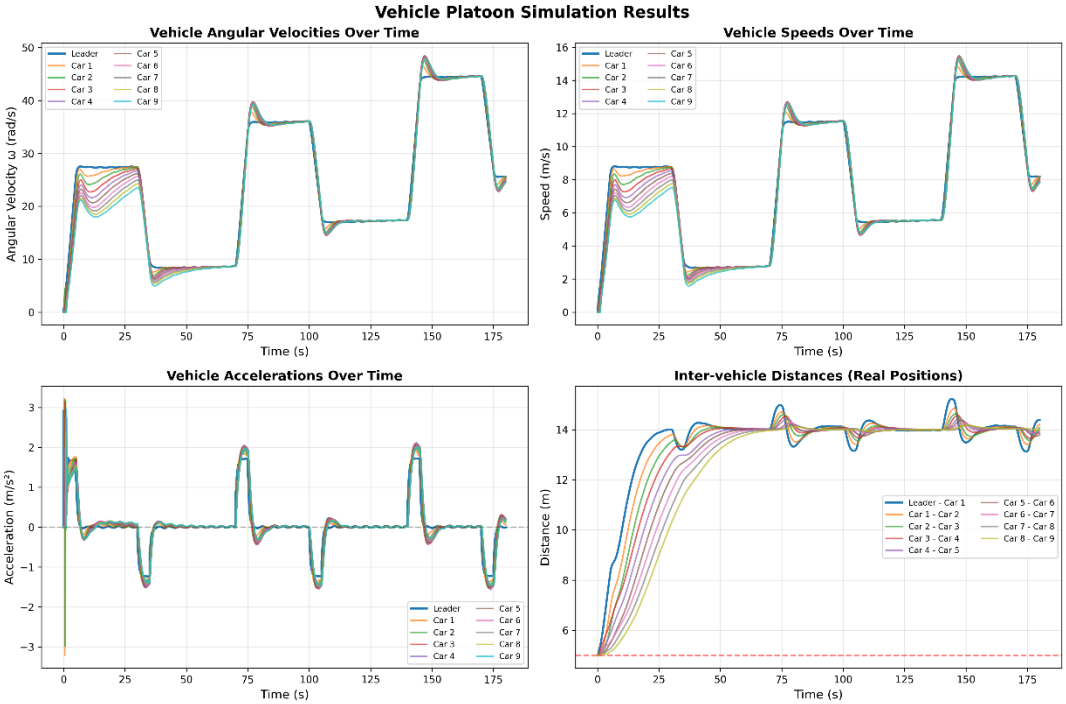
To address the stability issues, the Simulink internal simulation time step was reduced to 1 ms (0.001 s), representing a tenfold increase in temporal resolution. Notably, the co-simulation orchestration time step was maintained at 10 ms (0.01 s), thereby limiting computational overhead. This hierarchical time-stepping approach results in the Simulink solver executing ten internal steps for each co-simulation orchestration step, providing the necessary precision while managing computational cost.

6.3. Results

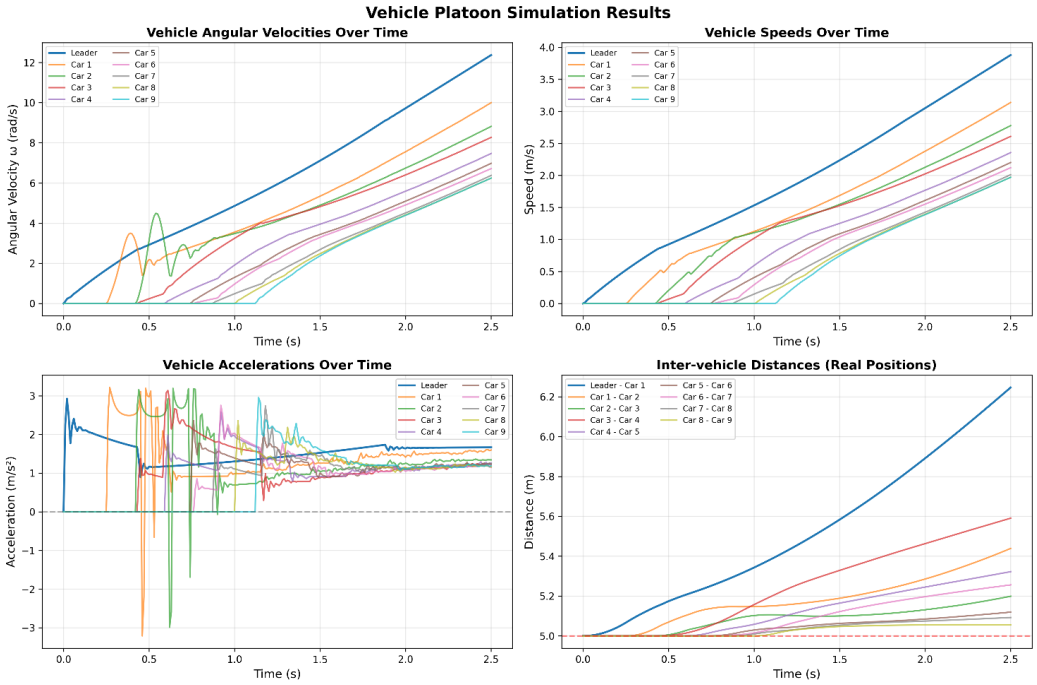
Driving on dry tarmac:



Driving on wet tarmac:

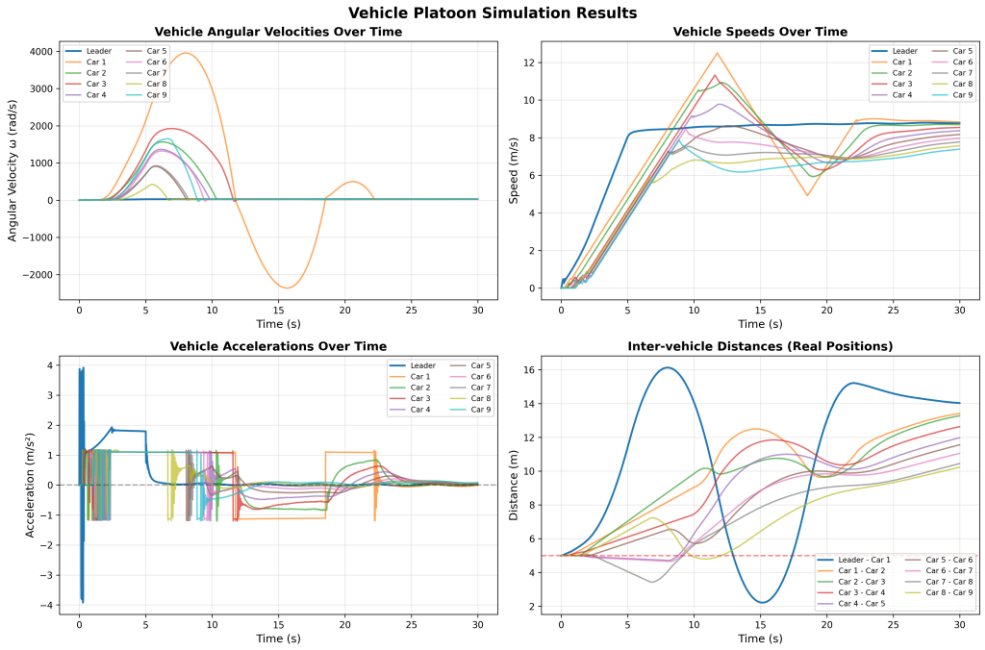


In particular, we can observe how there is some wheel slip of the wheels when pulling away from a standstill as the angular velocity of wheels – denoted "vehicle angular velocities over time" in the plot – is not proportional to the vehicle speed.



However, for the rest of the simulation, the behavior stays identical to the nominal case on dry tarmac.

Driving on snow:



Note that the simulation continues running after a collision, thus resulting in cars driving into each other and creating unrealistic inter-vehicular distances.

It is evident that the loss of traction associated with snow-covered surfaces introduces significant complications regarding platoon safety. Consistent with the findings reported in [Viadero-Monasterio, 2025][Pedone, 2020][Kotilainen, 2021], operating automated vehicle strings at short following distances under these conditions is inherently unsafe. Therefore, our methodology is applicable in all conditions in which the platoon works well in nominal cases (i.e. dry and wet pavement conditions). In our analysis, we focus on simulation data under dry pavement conditions. However, our methodology also works well on wet pavement conditions since the platoon is stable.

More specifically, this would require regenerating the traces from the wet-surface models and then deriving the discrete threshold values needed for the abstract model-checking analysis.

6.4. The effects of the wind

It is possible, in a similar manner, to model the effects of the wind and draft on the platoon. For instance, it may be useful to model persistent or rapidly changing winds, such as those occurring when exiting from a tunnel, driving on viaducts and overpass; or to model the so-called *slipstream effect*, that is, the reduction in aerodynamic drag when a vehicle follows another one closely, like in the CACC platoon. Such a thing is possible to model as shown, for instance, in [Pedone, 2020] at least in a simulated environment like ours by adding an addendum related to the wind speed to the resulting longitudinal force acting on the vehicle.

Such a model might be useful to study the efficiency of the platoon, as slipstreaming can reduce the fuel consumption, but it does not affect either the stability of the platoon or its susceptibility to attacks.

As far as it concerns weather conditions, an analogous reasoning as the ones presented in the case of road surfaces can be done by retuning the relevant thresholds based on simulation traces obtained with the presence of wind. This is ensured by the fact that the wind presence does not change the stability of the platoon.

7. Conclusions

This deliverable presents preliminary results on the applicability of the proposed methodology for detecting attacks in a platoon of vehicles. In particular, a strategy for identifying properties by comparing normal and under-attack traces is presented, as well as some properties identified based on inter-vehicular distances and the acceleration–distance relationship for a vehicle. The developed tools enable the entire discretization and automata-generation process to be automated. Finally, models of the platoon under different environmental conditions are investigated (e.g. various road surfaces). In this regard, the platooning application exhibits unstable behavior on snowy and icy surfaces, even in the absence of attacks. Therefore, within the scope of this project, we have decided to analyze only the operating conditions under which the platoon can be safely deployed: dry and wet surfaces. Furthermore, to enable a more thorough analysis, our focus was specifically on dry surfaces.

Bibliography

[Cleaveland, 2000] Cleaveland, R., Li, T., & Sims, S. (2000). The Concurrency Workbench of the New Century: User's manual (Version 1.2). Dept. of Computer Science, SUNY at Stony Brook.

[Andersen, J.R., 2015] Andersen, J.R. *et al.* (2015). CAAL: Concurrency Workbench, Aalborg Edition. In: Leucker, M., Rueda, C., Valencia, F. (eds) Theoretical Aspects of Computing - ICTAC 2015. ICTAC 2015. Lecture Notes in Computer Science(), vol 9399. Springer, Cham. https://doi.org/10.1007/978-3-319-25150-9_33

[Bakker et al, 1987] Bakker, E., Nyborg, L., & Pacejka, H. B. (1987). Tyre modelling for use in vehicle dynamics studies. *SAE Transactions*, 96, 190–204.

[MathWorks, 2025] MathWorks. (2025). *Longitudinal Wheel*. MATLAB & Simulink Help Center. https://it.mathworks.com/help/releases/R2025b/autoblks/ref/longitudinalwheel.html

[Viadero-Monasterio, 2025] Viadero-Monasterio, F., Meléndez-Useros, M., Jiménez-Salas, M. et al. Key influencing factors in vehicle platoons: a systematic study and review. Evolving Systems 16, 116 (2025). https://doi.org/10.1007/s12530-025-09746-1

[Pedone, 2020] S. Pedone and A. Fagiolini, "Racecar Longitudinal Control in Unknown and Highly-Varying Driving Conditions," in IEEE Transactions on Vehicular Technology, vol. 69, no. 11, pp. 12521-12535, Nov. 2020, doi: 10.1109/TVT.2020.3023059.

[Kotilainen, 2021] Kotilainen, I., Händel, C., Hamid, U., Nykänen, L., et al., "Connected and Automated Driving in Snowy and Icy Conditions—Results of Four Field-Testing Activities Carried Out in Finland," SAE Int. J. CAV 4(1):109-118, 2021, https://doi.org/10.4271/12-04-01-0009.