# Analyzing Many Simulations of Hybrid Programs in Lince

Reydel Arrieta
CISTER, Polytechnic Institute of Porto
Portugal
arrie@isep.ipp.pt

José Proença
CISTER & University of Porto
Portugal
jose.proenca@fc.up.pt

Patrick Meumeu Yomsi
CISTER, Polytechnic Institute of Porto
Portugal
pmy@isep.ipp.pt

**Abstract.** Hybrid systems are increasingly used in critical applications such as medical devices, infrastructure systems, and autonomous vehicles. Lince is an academic tool for specifying and simulating such systems using a C-like language with differential equations. This paper presents recent experiments that enhance Lince with mechanisms for executing multiple simulation variants and generating histograms that quantify the frequency with which a given property holds. We illustrate our extended Lince using variations of an adaptive cruise control system.

## 1 Contextualization and Motivation

Hybrid systems are complex systems that combine discrete with continuous behavior, where it is essential to meet timing, safety, functional and non-functional requirements under different operating conditions [6]. These systems are commonplace in applications such as cyber-physical systems, robotics, automotive and industrial automation. Several well-known tools like Uppaal [3] and KeYmaera [11] have been proposed in the literature to analyze their behavior and ensure their correctness. Although powerful, these tools struggle with non-linear systems and computational complexity. Lince [5, 7, 9] is a newer (academic) tool to simulate hybrid systems using a simple web-based interface, targeting programs with variables that can evolve based on systems of differential equations.

Currently, Lince relies primarily on trajectory plots for analysis. However, assessing the impact of changes in configuration parameters or scenarios requires manually generating and checking each variation individually. This work investigates how to analyze multiple simulations, either by overlapping many variations in a single plot, or by counting how many times a given property holds over time.

**Motivating example.** Consider a simple cruise control system. The interface of Lince for this example is depicted in Fig. 1: the program is specified in the top-left box, other parameters are configured in the bottom-left boxes, and the resulting plot is produced on the right. This hybrid program models the temporal evolution of a car's position $\mathbf{x}$ and velocity $\mathbf{v}$ by solving a system of first-order ordinary differential equations (ODEs).

The system is initialized with position $\mathbf{x} := \mathbf{0}$ and velocity $\mathbf{v} := \mathbf{2}$; while acceleration values are selected from the array $\mathbf{a} := [\mathbf{2}, \mathbf{4}, \mathbf{6}, \mathbf{8}, \mathbf{10}]$. A separate simulation is carried out for each acceleration value. In each case, the car evolves with constant acceleration $\mathbf{a}$, while its velocity fulfils $\mathbf{v} \leq \mathbf{10}$. Every time unit, the system checks if the velocity exceeds the threshold $\mathbf{10}$. If so, the system brakes with a deceleration of $-\mathbf{2}$.

Overlapping trajectories can make it difficult to verify velocity constraints or detect abnormal accelerations, e.g., *ensuring that the vehicle never exceeds a given velocity*. To overcome this limitation, we propose an approach for analyzing multiple simulations simultaneously. This method evaluates whether a given property holds across different time intervals and aggregates the results, which are then visualized using histogram charts to provide an intuitive representation of how frequently the property is satisfied.

Submitted to:
FMAS 2025

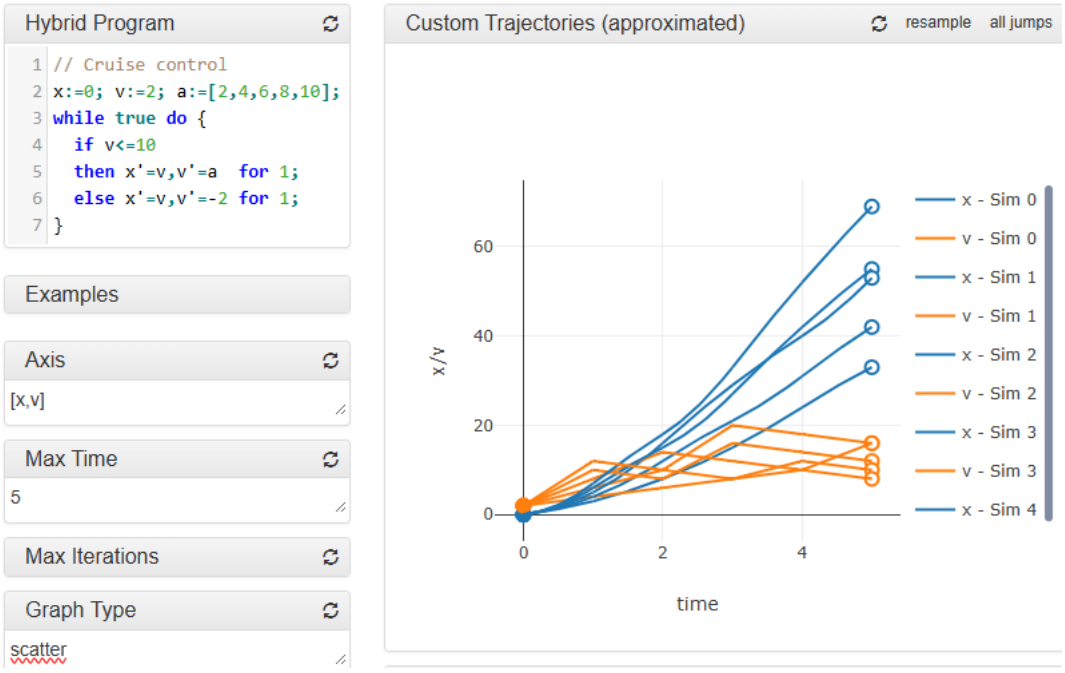


Figure 1: Modelling of a cruise control system in Lince

**Contributions.** The core contribution of this paper is twofold: (1) a step-by-step guide to using Lince through a use case involving a variant of an adaptive cruise control (ACC) system footnoteACC systems aim to maintain a safe distance between vehicles by dynamically adapting the acceleration of a following vehicle to the behavior of the vehicle in front. and (2) an extension of Lince that facilitates the analysis of multiple simulations of a single system with different configurations. The goal of this extension is not to provide a set of reusable libraries for multiple simulations and visualisation of complex systems, such as Numpy libraries for Python,* but to enrich Lince with the capability of running, analysing, and visualising many simulations. In particular, we extend Lince by supporting histograms [4] that count the number of times a given goal is reached at different points in time, and enrich the language to describe ranges of possible configurations. Using this extension, Lince can now run a system multiple times and produce a graph that marks, for each point in time from a set of sampling points, how often a given (desirable or undesirable) state is reached.

**Paper structure.** The paper is structured as follows. Section 2 details the design and implementation of Lince, covering the tool's architecture and the new histogram functionality for temporal analysis. Section 3 shows the application of Lince using an ACC case study, highlighting standard simulation and histogram-based analysis. Section 4 concludes the paper and describes future work.

## 2 Design and Implementation

This section describes the internal architecture of Lince and highlights recent extensions that aim to improve its analytical capabilities. It then describes our extension with histogram-based analysis, providing more intuitive insights into temporal patterns of system behavior.

*https://numpy.org/

## 2.1 Overview of Lince

Lince is implemented in Scala, compiles to JavaScript, features a web-based front-end, and includes a server for symbolic computations (though this server is not exploited in this work, which uses the SageMath algebraic solver) to avoid approximations. Lince is a lightweight and easily extensible tool, but not yet as mature or scalable as existing alternatives such as Simulink [13] or Uppaal [2, 3]. Recent enhancements improved usability, methods, and visualization [7], making it more practical. These improvements have enabled researchers to model more complex interactions between computational and physical components effectively, thus advancing the study and application of hybrid systems.

## 2.2 Lince Extension for Histogram Generation

Histograms depict how often a particular goal is achieved over time, allowing users to observe temporal patterns in system behavior. This functionality in Lince was inspired by similar features in tools such as Uppaal [3]. In Lince, it has been adapted to support the specific needs of hybrid systems analysis, providing a clear visualization of requirements' compliance over time. To generate a histogram in Lince, users modify the *Graph type* field by replacing the **scatter** command with **histogram** command (c.f. Fig. 1. The format of a histogram command is `histogram: <requirement> @ <sampling>`, where `requirement` describes the condition of interest to be checked, e.g., `v>=10`, and `sampling` is an optional argument describing the points used to evaluate the requirement, e.g., the sampling `step 5` divides the run time into 5 evenly distributed sampling points, and `every 0.2` sets sampling points every 0.2 time units). Fig. 2 illustrates two histograms obtained from the example in Fig. 1 using 5 and 50 different acceleration values, respectively.

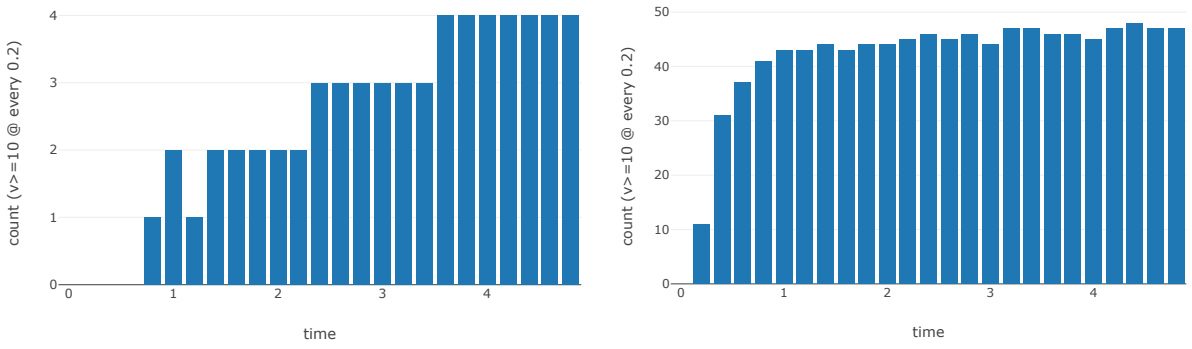


Figure 2: Histogram plotting of Cruise Control with 5 and 50 simulations, respectively

The implementation of this functionality was carried out in two main phases: (1) the adaptation of Lince to collect relevant data during the analysis, and (2) the generation and graphical visualization of histograms from the collected data. This enhancement not only improves the user experience, but also extends Lince 's analytical capabilities, enabling a more in-depth study of temporal patterns in the fulfillment of requirements.

## 3 Analyzing an Adaptive Cruise Control

This section presents a formal model of an ACC system that uses a variant of a hybrid program in Lince,[†] and describes how a safe distance to the vehicle ahead is maintained (and a target speed is ignored). This use case can be run online at **http://arcatools.org/lince**.

**ACC modeling.** We consider a Lince hybrid program that simulates two vehicles on a straight road: a *leader* and a *follower*. The core of the ACC program is the predicate Safe($p_f$, $v_f$, $p_\ell$, $v_\ell$, $a_\ell$), which determines whether it is safe for the *follower* to accelerate. In particular, it checks whether, given the position $p_f$ and velocity $v_f$, of the *follower* together with the position $p_\ell$, velocity $v_\ell$, and acceleration $a_\ell$ of the *leader*, a collision would occur if the *follower* accelerates during the next sample time (**st**). The program includes an infinite while-loop where, at each iteration, the dynamics of both vehicles evolve according to their differential equations. Inside the loop, the Safe function is invoked to guide decisions about acceleration or braking.

```
while true do {
  if Safe(pf,vf,pl,vl,al)
  then af:=bwd;
  else af:=fwd;
pf'=vf,vf'=af,af'=0,
pl'=vl,vl'=al,al'=0 for st;}
```

To compute the Safe predicate, we assume that the *follower* starts by accelerating for a period of **st** (*first phase*), and then brakes (*second phase*). We then manually compute whether the vehicles collide in the *first* or *second* phase. If any collision is detected, **Safe** returns false and the *follower* sets its acceleration to **bwd** $\in \mathbb{R}^-$ (since it is not safe to accelerate); otherwise it returns true and the *follower* sets its acceleration to **fwd** $\in \mathbb{R}^+_*$. Hereafter, we explain the dynamics of each phase.

**First phase.** Assuming the *leader* maintains a constant acceleration $a_\ell$, the estimated positions of the *follower* $p_f^{st}$ and the *leader* $p_\ell^{st}$ up to time **st** are given by $p_f^{st} = \frac{\mathbf{fwd}}{2}\cdot\mathbf{st}^2 + v_f\cdot\mathbf{st} + p_f$ and $p_\ell^{st} = \frac{a_\ell}{2}\cdot\mathbf{st}^2 + v_\ell\cdot\mathbf{st} + p_\ell$, respectively. Furthermore, the *follower*'s velocity is $v_f^{st} = \mathbf{fwd}\cdot\mathbf{st} + v_f$. Therefore, the system reduces to a numeric identity, since there are no variables. As a result, the positions of the vehicles after a time step **st** can be determined exactly. If $p_f^{st} < p_\ell^{st}$, then we assume that the vehicles did not collide in the first phase.

**Second phase.** After **st** time units, we check if the trajectory $p_f(t)$ of the *follower* can intersect the trajectory $p_\ell(t)$ of the *leader* at any time $t \geq 0$. More specifically, we estimate these trajectories as follows, where **fwd** and **bwd** are the acceleration and braking values of the *follower*, respectively, and $p_f^{st}$, $p_\ell^{st}$ and $v_f^{st}$ are defined above.

$$p_f(t) = \frac{\mathbf{bwd}}{2}\cdot t^2 + v_f^{st}\cdot t + p_f^{st} \quad \text{and} \quad p_\ell(t) = \frac{a_\ell}{2}\cdot t^2 + v_\ell\cdot t + p_\ell^{st}$$

A collision occurs if there exists a time $t$ such that $p_\ell(t) - p_f(t) = 0$, which evaluates to the following equation.

$$\underbrace{\frac{(a_\ell - \mathbf{bwd})}{2}\cdot t^2}_{at} + \underbrace{[(a_\ell - \mathbf{fwd})\cdot(\mathbf{st}) + (v_\ell - v_f)]}_{bt}\cdot t + \underbrace{\left[\frac{(a_\ell - \mathbf{fwd})}{2}\cdot(\mathbf{st})^2 + (v_\ell - v_f)\cdot(\mathbf{st}) + (p_\ell - p_f)\right]}_{ct} = 0$$

[†]The source code of Lince is available online at https://github.com/arcalab/lince.

We then search for solutions, which exist if $\Delta = bt^2 - 4 \cdot at \cdot ct \geq 0$ or $at == 0$ and $t = -ct/bt > 0$.

**Combining both phases.**

Combining the results above, the final definition of Safe is defined as follows.

$$\mathsf{Safe}(\mathsf{p_f}, \mathsf{v_f}, \mathsf{p_\ell}, \mathsf{v_\ell}, \mathsf{a_\ell}) = \mathsf{p_\ell}(\mathbf{st}) \leq \mathsf{p_f}(\mathbf{st}) \vee [\mathsf{a_\ell} == \mathbf{bwd} \wedge -ct/bt > 0] \vee \left[\Delta \geq 0 \wedge \mathsf{a_\ell} \neq \mathbf{bwd} \wedge \left(\frac{-bt + \sqrt{\Delta}}{2 \cdot at} > 0 \vee \frac{-bt - \sqrt{\Delta}}{2 \cdot at} > 0\right)\right]$$

**Simulation setups and analysis of the results.** In our simulations, we use **fwd** $= 3$, **bwd** $= -3$, and **st** $= 2$. Note that **st** $= 2$ reflects a value commonly used in the automotive industry [10, 12], where the prediction horizons for forward position and collision prediction are typically between 2 and 4 seconds. The *leader* is assumed to be stationary ($\mathsf{v_\ell} = 0$) at $\mathsf{p_\ell} = 50$ units from the origin. A set of integer values for acceleration $\mathsf{a_\ell}$ is defined, where each value corresponds to a separate run. This set is denoted as $\mathsf{a_\ell} := [-3..3]$, which represents a discrete sampling of possible acceleration profiles and constitutes a basic contribution to the development of Lince. The *follower*, in contrast, is positioned at the origin, also at rest, and begins with a constant acceleration $\mathsf{a_f} = 3$. The system dynamics are modeled by an infinite loop (`while true`), in which the states of the two vehicles evolve according to differential equations that are integrated over the sampling interval **st:=2**. Specifically, for the *follower*: $\mathsf{p_f}$'=$\mathsf{v_f}$, $\mathsf{v_f}$'=$\mathsf{a_f}$ and for the *leader*: $\mathsf{p_\ell}$'=$\mathsf{v_\ell}$, $\mathsf{v_\ell}$'=$\mathsf{a_\ell}$.

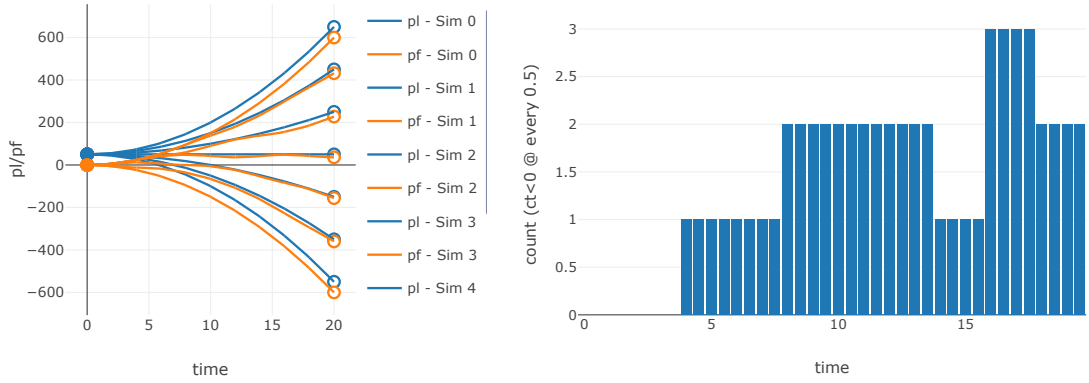


Figure 3: Simulations of the ACC in Lince, displaying the trajectories of the vehicles (left) and the histogram counting when $ct \leq 0$.

.

After modeling the scenario in Lince, a complex graph is obtained consisting of seven simulation traces, each corresponding to one of the previously defined acceleration values (see Figure 3, left). Overlapping trajectories make braking events difficult to identify. To address this, our approach introduces the use of the *Graph type* field with the option **histogram:** $ct \leq 0$ **@ every 0.5**, for example. Here we determine how many collisions occur during the first phase in particular, i.e., $\mathsf{p_\ell}(\mathbf{st}) \leq \mathsf{p_f}(\mathbf{st})$. This way yields a more interpretable visualization (see Figure 3, right). It is important to note that this constitutes a discretized analysis; therefore, reducing the evaluation interval increases the accuracy of the results but also introduces a higher computational complexity. The histogram is interpreted as follows: each increment represents the occurrence of a new event, while each decrement indicates the absence of an event.

Periods without changes indicate steady states that correspond to previously established conditions and therefore do not convey additional information.

**The ACC use case analyzed by other tools.** In **Simulink** [13], our AAC use case could be modeled by creating multiple interconnected blocks for sensors, controllers, and vehicle dynamics, with parameters manually configured for each simulation run. This approach requires some familiarisation with these tools and how to properly configure all parameters. In **UPPAAL** [3], the system would be represented as a network of timed automata, where each component (*leader*, *follower*, and controller) would be described through discrete states and clock constraints. Although UPPAAL targets timed automata (without ODEs) [1], it also supports simple ODEs such as the ones used in this ACC case, but not more complex ODEs (unlike Lince). UPPAAL further supports running multiple random simulations, but there is no explicit mechanism to build specific scenarios, and it supports a variety of queries to infer statistical analysis or produce plots and histograms. Unlike Simulink and UPPAAL, Lince aims at being an easy to run tool, which does not require any installation (a web-browser is enough), and easy to extend and experiment, with a compact dedicated input language that is precise and rich enough to describe and analyse non-trivial examples. Lince does not aim at being efficient, scalable, nor providing an extensive set of features. Table 1 summarizes the qualitative differences between Simulink, UPPAAL, and Lince.

Using dedicated libraries, such as **Numpy** for Python, one can produce most of these analysis by manually describing the system of equations and encoding all the control logics into a Python script. We believe that this is harder to write for non-experts than a Lince program, and harder to maintain and experiment, given that the declarative hybrid program (read by Lince) would be manually implemented in a Python script. Numpy (or an alternative) could be used as a possible back-end for Lince, although we currently focus on Scala libraries or JavaScript (JS) libraries (e.g., we use the Plotly JS libraries[‡] to draw plots and histograms), given that Lince is implemented in Scala and compiled to JS.

| **Feature** | **Simulink** | **UPPAAL** | **Lince** |
|---|---|---|---|
| Model representation | Graphical block diagrams | Networks of timed automata | Textual hybrid programs (guarded commands + ODEs) |
| Installation | Requires MATLAB environment | Requires desktop installation and modeling tool | Runs directly in browser |
| Focus | Large-scale numeric simulation and control design | Rigorous verification of real-time systems | Lightweight exploration of hybrid system logic |
| Model representation | Not supported natively | Symbolic reasoning over guards and invariants of timed automata | Symbolic reasoning over guards, invariants, and continuous dynamics (ODEs) |
| User interaction | Graphical modeling through block connections; parameter tuning via GUI | Graphical editor for automata construction and query interface for verification | Direct code editing and instant visualization of trajectories and histograms in the browser |

Table 1: Comparison of modeling and analysis features across tools.

[‡]`https://plotly.com/javascript/`

**Lessons learned.** Guided by our ACC example, we identified difficulties in the modelling and analysis of non-trivial CPS, namely regarding the impact of experimenting with variations of some parameters. In this example, it was not clear how often a vehicle would reach the target proximity of a leading vehicle in different scenarios with different leading vehicles. Other experiments not reported here, involving injected errors in the periodicity delay, further supported the usefulness of quickly experimenting and simulating these kinds of critical scenarios. We also concluded that further analysis or extensions could be useful, such as: calculating expected values, supporting modular definitions of each vehicle in separate, or supporting fine-tune configuration of some hard-coded parameters (e.g., the precision of the ODE solver). This is left for future work.

## 4 Conclusion and Future Work

Analyzing hybrid systems poses significant challenges due to the combination of discrete and continuous behaviors, especially in critical applications where temporal requirements are essential. To address this issue, Lince has evolved by incorporating new functionalities that facilitate the interpretation of massive simulations and the verification of temporal properties. The main contribution of this improvement is the integration of histograms, which provide a clear visualization of the frequency with which certain events occur over time. This resolves the previous problem of graph overlapping in multiple simulations, offering a direct way to detect compliance patterns and anomalies. Furthermore, adopting a simplified notation for defining lists with unit-step discretization has optimized the simulation setup process, making the tool more agile.

These results open an interesting avenue for future work: extending the current support for random value generation towards a principled treatment of stochastic hybrid systems. Our recent paper [8] introduces a while language that integrates probabilistic constructs with differential dynamics, and shows how Lince can be extended to reason about stochastic hybrid programs. Experimental histograms can already be found in that paper, which are formally presented here.

We are also exploring how Lince could be used to validate formal monitors or to generate tests and monitors from logical specifications, e.g., inspired on the work by Yamaguchi et al. [14]. On a different front, we plan to investigate modularity with existing toolchains and mathematical modelling tools, and modularity of the specifications, supporting the composition of independent components.

## 5 Acknowledgments

This work was supported by national funds through FCT/MECI (Portuguese Foundation for Science and Technology), within the CISTER Research Unit (UIDP/UIDB/04234/2020) and under the project Intelligent Systems Associate Laboratory - LASI (LA/P/0104/2020) and Project Route 25 (ref. TRB/2022/00061 - C645463824-00000063), funded by the EU/Next Generation, within call n.º 02/C05-i01/2022 of the Recovery and Resilience Plan (RRP).

## References


[1] G. Behrmann, A. David & K. Larsen (2004): *A Tutorial on Uppaal*, doi:10.1007/978-3-540-30080-9_7. Updated version available online.

[2] Peter E. Bulychev, Alexandre David, Kim Guldstrand Larsen, Marius Mikucionis, Danny Bøgsted Poulsen, Axel Legay & Zheng Wang (2012): *UPPAAL-SMC: Statistical Model Checking for Priced Timed Automata*. In Herbert Wiklicky & Mieke Massink, editors: *Proceedings 10th Workshop on Quantitative Aspects of Programming Languages and Systems, QAPL 2012, Tallinn, Estonia, 31 March and 1 April 2012*, *EPTCS* 85, pp. 1–16, doi:10.4204/EPTCS.85.1.

[3] Alexandre David, Kim G Larsen, Axel Legay, Marius Mikučionis & Danny Bøgsted Poulsen (2015): *Uppaal SMC tutorial*. *International journal on software tools for technology transfer* 17(4), pp. 397–415, doi:10.1007/s10009-014-0361-y.

[4] DA Gedcke (2001): *How histogramming and counting statistics affect peak position precision*. *ORTEC Application Note AN58*. Available at `https://api.semanticscholar.org/CorpusID:201858890`.

[5] Sergey Goncharov & Renato Neves (2019): *An Adequate While-Language for Hybrid Computation*. In Ekaterina Komendantskaya, editor: *Proceedings of the 21st International Symposium on Principles and Practice of Programming Languages, PPDP 2019, Porto, Portugal, October 7-9, 2019*, ACM, pp. 11:1–11:15, doi:10.1145/3354166.3354176.

[6] Phillip A Laplante & Mohamad Kassab (2022): *Requirements engineering for software and systems*. Auerbach Publications, doi:10.1201/9781003129509.

[7] Pedro Mendes, Ricardo Correia, Renato Neves & José Proença (2024): *Formal Simulation and Visualisation of Hybrid Programs*. In Matt Luckcuck & Mengwei Xu, editors: *Proceedings Sixth International Workshop on Formal Methods for Autonomous Systems, FMAS@iFM 2024, Manchester, UK, 11th and 12th of November 2024*, *EPTCS* 411, pp. 20–37, doi:10.4204/EPTCS.411.2.

[8] Renato Neves, José Proença & Juliana Souza (2025): *An adequate while-language for stochastic hybrid computation*. In Małgorzata Biernacka & Carlos Olarte, editors: *Proceedings of the 27th International Symposium on Principles and Practice of Declarative Programming, PPDP 2025, Rende, Italy, September 10-11, 2025*, ACM. Available at `https://arxiv.org/abs/2507.15913`. To appear.

[9] Renato Jorge Araújo Neves (2018): *Hybrid programs*. Ph.D. thesis, Universidade do Minho (Portugal). Available at `https://hdl.handle.net/1822/56808`.

[10] Jinan Piao & Mike McDonald (2008): *Advanced driver assistance systems from autonomous to cooperative approach*. *Transport reviews* 28(5), pp. 659–684, doi:10.1080/01441640801987825.

[11] André Platzer & Jan-David Quesel (2008): *KeYmaera: A hybrid theorem prover for hybrid systems (system description)*. In: *International Joint Conference on Automated Reasoning*, Springer, pp. 171–178, doi:10.1007/978-3-540-71070-7_15.

[12] Rajesh Rajamani (2011): *Vehicle dynamics and control*. Springer Science & Business Media, doi:10.1007/978-1-4614-1433-9.

[13] Ricardo Sanfelice, David Copp & Pablo Nanez (2013): *A toolbox for simulation of hybrid systems in Matlab/Simulink: Hybrid Equations (HyEQ) Toolbox*. In: *Proceedings of the 16th international conference on Hybrid systems: computation and control*, pp. 101–106, doi:10.1145/2461328.2461346.

[14] Tomoya Yamaguchi, Bardh Hoxha & Dejan Nickovic (2024): *RTAMT - Runtime Robustness Monitors with Application to CPS and Robotics*. *Int. J. Softw. Tools Technol. Transf.* 26(1), pp. 79–99, doi:10.1007/S10009-023-00720-3.