Optimising Highly-Parallel Simulation-Based Verification of Cyber-Physical Systems

Toni Mancini, Igor Melatti, and Enrico Tronci

Abstract—Cyber-Physical Systems (CPSs), *i.e.*, systems comprising both software and physical components, arise in many industry-relevant application domains and often mission- or safety-critical.
System-Level Verification (SLV) of CPSs aims at certifying that given (*e.g.*, safety or liveness) specifications are met, or at estimating the value of some Key Performance Indicators, when the system runs in its operational environment, that is in presence of inputs (from the user or other systems) and/or of additional, uncontrolled disturbances.
In order to enable SLV of complex systems from the early design phases, the currently most adopted approach envisions the *simulation* of a *system model* under the (time bounded) *operational scenarios* deemed of interest.
Unfortunately, simulation-based SLV can be computationally prohibitive (years of sequential simulation), since system model simulation is computationally intensive and the set of scenarios of interest can be extremely large.
In this article, we present a technique that, given a collection of scenarios of interest (extracted from mass-storage databases or from symbolic structures like constraint-based scenario generators), computes *parallel shortest simulation campaigns*, which drive a possibly large number of system model simulators running in parallel in a HPC infrastructure through all (and only) those scenarios in the user-defined (possibly random) order, by wisely avoiding multiple simulations of repeated trajectories, and thus minimising the overall completion time, compatibly with the available simulator memory capacity.
Our experiments on SLV of Modelica/FMU and Simulink case study models with up to almost *200 million scenarios* show that our optimisation yields *speedups as high as* $8\times$. This, together with the enabled *massive parallelisation*, makes practically viable (a few weeks in a HPC infrastructure) verification tasks (both statistical and exhaustive, with respect to the given set of scenarios) which would otherwise take *inconceivably* long time.

This article appears in IEEE Transactions on Software Engineering, 2023. DOI: 10.1109/TSE.2023.3298432

1 Introduction

Cyber-Physical Systems (CPSs) consist of interconnected hardware (the physical part) and software (the cyber part). CPSs are ubiquitous in many industry-relevant application domains, *e.g.*, aerospace, automotive, energy, biology, healthcare, among many others. In many CPSs (*e.g.*, in embedded systems), the software part consists of a (typically microprogrammed) controller which continuously senses the state of the system and sends commands to the hardware actuators in order to achieve an envisioned goal condition while satisfying some requirements.

System-Level Verification (SLV) of a CPS aims at verifying that the whole system (*i.e.*, the software and the hardware working together) meets the given specifications when running in its operational environment, *i.e.*, in presence of inputs and/or additional limited uncontrolled (but possible) events (such as faults, noise signals, or changes in system parameters, collectively referred to as *disturbances*).

Since industry-relevant CPSs are often mission- or safety-critical, their SLV is of paramount importance to build confidence on their robustness and, ultimately, to perform their qualification. To this end, SLV of CPSs is supported from the early design stages by well-known model-based design software tools, *e.g.*, among the others, Simulink, VisSim, Dymola, ESA Satellite Simulation Infrastructure SIMULUS. Such tools allow the user to mathematically model the physical parts of a CPS (the hardware model), by means of, *e.g.*, differential equations and/or algorithmic snippets to manage, *e.g.*, the occurrence of events, and enable their numerical simulation, both open-loop and closed-loop. In particular, during closed-loop simulation, the actual software for the controller continuously reads values from the connected hardware model and decides control actions. During simulation of CPS models, the above model-based design tools also allow the user to inject a time series of inputs and other disturbances stemming from the environment, representing an actual *operational scenario*.

By designing a proper set of scenarios deemed plausible (given the operational environment), SLV of the system is performed either by verifying that the CPS model satisfies the given specifications under *all* of them (aka *exhaustive model checking*, where exhaustiveness is intended with respect to such set of scenarios), or, when they are too many to be simulated exhaustively, the residual probability of errors or expected values of suitable Key Performance Indicators (KPIs) are estimated by simulating the system on a randomly chosen subset of scenarios (*statistical model checking* [3]).

1.1 Background and Motivations

Unfortunately, models of industry-relevant CPSs are often defined as systems of highly non-linear and possibly stiff

• *Authors are with the Computer Science Department, Sapienza University of Rome, Italy.*
E-mail: tmancini@di.uniroma1.it, melatti@di.uniroma1.it, tronci@di.uniroma1.it

Manuscript received MMMMM DD, 2022

differential equations, and their complexity hinders the possibility of any symbolic reasoning (via, *e.g.*, model checkers for hybrid systems). As a result, the main workhorse for SLV of such system models is their black-box simulation on each single scenario, in order to check whether the required system-level specifications are satisfied on all of them or to estimate the values for the KPIs of interest.

Simulating a CPS model on a single scenario can take from seconds to minutes, depending on the requested simulation time horizon and on the complexity of the system model. For example, simulating our case study models on a single scenario takes around 60 (Apollo Lunar Model Autopilot, ALMA, by Simulink), 80 (Buck DC-DC Converter, BDC, by JModelica/FMU) and 40 seconds (Fuel Control System, FCS, by Simulink) on average. This is due to the frequent injections of disturbances and/or changes of parameters, as prescribed by the scenario being simulated. All this makes simulation-based SLV campaigns of such CPSs a *complex and extremely time-consuming activity*.

There are two major sources of complexity to deal with when carrying out simulation-based SLV of CPSs.

The first source of complexity stems right from the definition of the set of scenarios deemed plausible or worth of interest, against which the system model must be verified. Traditionally, such scenarios (which collectively define the CPS *operational environment*) are manually defined by system verification teams together with domain experts, and stored in large databases. When a new version of the CPS model has to be verified, such scenarios are injected during simulation and the resulting model trajectories are evaluated. Beyond being extremely time-consuming (possibly requiring months of work from expert designers), this naïve operational environment definition activity is extremely fragile, as it is hard to assure that the successful verification of the CPS model against such scenarios is sufficient to certify absence of errors. This is because it would be impossible to state whether the defined set of scenarios is representative of *all* the possible situations of interest.

To overcome this obstruction, previous work [28], [38] proposed to lift the hand-crafted definition of operational scenarios into the definition of a *declarative constraint-based specification* of the system operational environment via an automaton encoded in a high-level language. The set of possible scenarios against which to verify the CPS model is then defined as the set of time series of inputs and other uncontrollable events encoded by accepting computation paths on such an automaton. Also, one such form of automaton, named *scenario generator* in [38], allows the efficient extraction of any of its entailed scenarios from their unique indices. The definition of the CPS operational environment by such high-level models greatly eases the task of the verification engineers to capture all scenarios deemed plausible, also allowing them to dynamically focus on those scenarios satisfying additional constraints (see [38] for examples), thus enabling prioritisation of the (typically very long) verification activity.

Also, the availability of an environment model entailing a possibly large, yet finite number of scenarios enables the *exhaustive* (with respect to such an environment model) *verification* of the CPS at hand. Indeed, when the CPS model is exercised on *all* the (finite number of) scenarios entailed by the environment model, a clear *degree of assurance* is attained at the end of the verification process. Furthermore, by properly *randomising* the scenario verification order, suitable information on the probability that a yet-to-be-simulated scenario exists for which the CPS shows an error (*omission probability*) can be returned *any-time* during verification [32]. This allows the user to halt the verification process when the residual probability of an error goes below a given threshold (graceful degradation). Similar advantages can be achieved when the environment model yields a too large or an infinite number of scenarios, which would hinder the possibility to verify the system on all of them. In such cases, statistical model checking can be exploited by randomly sampling a finite number of scenarios from the environment model, and a statistically-sound degree of assurance that the property under verification holds, or a statistically sound estimation of the value of some KPIs can be generated at the end of the (finite) verification process.

The second major source of complexity to deal with when performing simulation-based SLV of CPSs is carrying out the *actual simulation* of the system model on all the selected scenarios (regardless on how they are selected). This is because, to achieve a high-enough level of assurance on its correctness, the system must be typically simulated on a very large number of scenarios (*e.g.*, in our case studies we tackle verification processes on up to almost 200 million scenarios), yielding *prohibitive* simulation times. Tackling this last issue is the main focus of this article.

1.2 Contributions

We present an approach to compute *optimised* simulation campaigns to perform SLV of CPSs in a *highly parallel* environment (*e.g.*, a large High-Performance Computing, HPC, infrastructure), given identical simulators of the CPS model and a (possibly large yet finite) collection of operational scenarios. Our contributions are as follows.

Shortest simulation campaigns for highly-parallel CPS verification. We present an algorithm that, given as input a (typically very large) set of operational scenarios (either generated from a high-level environment model or extracted from a database), computes a set of *optimised simulation campaigns* out of them, which drive multiple simulators of the CPS (running in parallel) through all (and only) such scenarios in the (possibly random) order chosen by the user, while aiming at *minimising* the verification *completion time*.

Since the parallel execution of the computed simulation campaigns requires no inter-process communication, very large HPC infrastructures can be seamlessly exploited to greatly shorten the overall verification activity.

Case studies. We show the applicability of our algorithm on three case studies of industry-scale CPSs, by performing their verification against *very large* sets of scenarios (up to almost 200 million scenarios), using up to (virtually) 65 536 cores of a HPC infrastructure (that is 1024 64-core machines), and evaluate the benefits of our optimised simulation campaign computation algorithm and the scalability of our overall approach.

Thanks to our overall architecture which envisions a *simulator-independent* campaign computation algorithm and *simulator-specific drivers*, we can virtually control any available simulation engine. We have currently developed (and successfully used in our experiments) drivers for the widely popular simulation platforms Simulink and JModelica/Functional Mock-Up Unit (FMU).

2 Formal Framework

We denote with $\mathbb{R}$, $\mathbb{R}_{0+}$ and $\mathbb{R}_{+}$ the sets of, respectively, all, non-negative, and strictly positive real numbers, and with $\mathbb{N}_{+}$ and $\mathbb{N}$ the sets of, respectively, strictly positive and non-negative integer numbers. Also, given two sets A and B, we denote by A^B the set of functions from B to A.

We now briefly describe how we model our System Under Verification (SUV), its operational environment, and the property to be verified. For brevity, formal definitions (including notation recap) and statements as well as their proofs are delayed to Appendix A.

System Under Verification (SUV). We assume our (black-box) SUV $\mathcal{H}$ to be a deterministic, time-invariant, causal, state-input-output dynamical system (*e.g.*, [29], [31], [48]) over a continuous or discrete time set $\mathbb{T}$ (hence $\mathbb{T}$ is either $\mathbb{N}$ or $\mathbb{R}_{0+}$, or an interval thereof), and whose input space $\mathbb{U}$ defines the set of possible values for the user inputs and the other uncontrollable events $\mathcal{H}$ is subject to, *e.g.*, faults in sensors and actuators or changes in system parameters. Thus, $\mathcal{H}$ takes as input a time function $\mathbf{u} \in \mathbb{U}^{\mathbb{T}}$, defining the SUV input values at all time points ($\mathbf{u}$ is called an *operational scenario*, or just *scenario*).

Property to be verified. For maximum generality, we assume that the property to be verified and/or the KPIs to be computed under each scenario are encoded as a monitor within $\mathcal{H}$. The monitor observes the state of the system and checks whether the property under verification is satisfied and/or computes the values of the KPIs of interest. The use of monitors as black boxes gives us maximum flexibility and it allows us to abstract away the actual formalism used to define the property (*e.g.*, it is immediate to define as monitors bounded safety and bounded liveness properties as well as checkers for formulas in any temporal logic; see, *e.g.*, [27], [43] and citations thereof). Since the monitor output is all we need to carry out our verification task, in the sequel we assume that the only outputs of the SUV are those from the monitor.

System-Level Verification (SLV). An *SLV problem* is a pair $\pi = (\mathcal{H}, \mathcal{U})$, where $\mathcal{H}$ is a SUV (with an embedded monitor) and $\mathcal{U}$ is a set of scenarios for it. The *answer to SLV problem* π is the collection of the outputs of the SUV (monitor) produced at the end of each scenario $\mathbf{u} \in \mathcal{U}$, where $\mathbf{u}$ is injected in $\mathcal{H}$ starting from its initial state.

Our definition of (answer to an) SLV problem is very general and, depending on how the SUV monitor is defined, seamlessly accounts for verification activities aimed at either checking whether a scenario in $\mathcal{U}$ exists which raises an error in the SUV (*error scenario*), or at computing *statistics* on (*e.g.*, expected values of) some KPIs. Namely, to find an error scenario, it is enough to define the monitor to return *PASS* or *FAIL* at the end of each of them, depending on whether the scenario satisfies or violates the property under verification. Conversely, to compute any sort of statistics on any KPIs of interest, it is enough to define the monitor to compute and output such KPI values at the end of each scenario.

SUV operational environment. According to our focus on verification tasks where numerical simulation is the only means to get the trajectory of the SUV when fed with an input scenario, we will assume that the set $\mathcal{U}$ is finite and finitely representable, and that each scenario is time bounded. Hence, we assume that the set of values taken by input scenarios in $\mathcal{U}$ (actually, for simplicity, the set $\mathbb{U}$ itself) is finite (and, without loss of generality, ordered) and scenarios in $\mathcal{U}$ are defined via *piecewise constant* input time functions having discontinuities at time points multiple of a given (arbitrarily small) *time quantum* $\tau \in \mathbb{T} \setminus \{0\}$. Such scenarios can be conveniently represented as *input traces* (Definition 1).

Definition 1 (Input trace). *An input trace* $\mathbf{u}$ *with values in* $\mathbb{U}$ *is a finite sequence* $(u_0, \ldots, u_{h-1})$ *where all* u_i *belong to* $\mathbb{U}$. *Value* h *is the trace* horizon.

Given time quantum $\tau \in \mathbb{T} \setminus \{0\}$, an input trace $\mathbf{u} = (u_0, \ldots, u_{h-1})$ is interpreted as the bounded-horizon piecewise constant time function $\mathbf{u} \in \mathbb{U}^{[0,\tau h)}$ defined as $\mathbf{u}(t) = u_{\lfloor \frac{t}{\tau} \rfloor}$ for $t \in [0, \tau h)$. From now on we assume that a time quantum τ is given, and thus interchangeably refer to input traces and to their uniquely associated time-bounded piecewise constant time functions.

Our assumptions above naturally apply to scenarios whose values denote *events* such as user requests or faults. However, scenarios encoding input time functions assuming continuous values can be tackled by means of a suitable discretisation of their domains, whilst smooth continuous-time input functions (*e.g.*, additive noise signals) can be managed as long as they can be cast into (or suitably approximated by) finitely parametrisable functions, in which case the input space actually defines such a (discrete or discretised) parameter space. Examples of finite parameterisations of the SUV input space are those defining limited, quantised Taylor expansions of continuous-time inputs, or those defining quantised values for the first coefficients (those carrying out the most information) of the Fourier series of a finite-bandwidth noise, see, *e.g.*, [1], [34].

As argued in [38], our assumptions are in line with an engineering (rather than purely mathematical) point of view, where man-made CPSs need to satisfy the properties under verification with some degree of *robustness* with respect to the actual input time functions (see, *e.g.*, [1], [16] and references thereof). Our case studies in Section 6 contain uses of several of such features, and show that our setting can be easily met in practice.

3 SUV Simulators

In our simulation-based setting, we aim at performing SLV of our SUV by driving the execution of a *simulator* of the SUV model (in *e.g.*, Simulink, Modelica) via the simulation engine scripting language, which also takes care of injecting piecewise constant input time functions representing scenarios. By extending the formal notion of *SUV simulator* in [31], [34], we provide a general mathematical framework that allows us to link scenarios given as input to a SUV $\mathcal{H}$ (as input traces encoding piecewise-constant input time functions) to inputs for a simulator of $\mathcal{H}$ (*simulation campaigns*).

Formal definitions as well as statements and their proofs in this section are delayed to Appendix B.

A simulator for SUV $\mathcal{H}$ is a tuple $(\mathcal{H}, \mathcal{W})$, where $\mathcal{W}$ denotes the set of simulator *states*. Each $w \in \mathcal{W}$ has the form $w = (x, \mathbf{u}, \mathcal{M})$, where: x is a state of $\mathcal{H}$; $\mathbf{u}$ is an input time function (an input trace in our setting) for $\mathcal{H}$; $\mathcal{M}$ (simulator *memory*) is a finite map whose elements are of the form: $[\lambda \mapsto (x', \mathbf{u}')]$, with $\lambda \in \Lambda$ being an identifier from a countable set (*unique* in $\mathcal{M}$), x' a state of $\mathcal{H}$, and $\mathbf{u}'$ is an input trace.

A simulator $\mathcal{S}$ can take the following *commands*: OUTPUT, which reads the output of $\mathcal{S}$ in the current state; LOAD(λ), which loads from memory the state associated to identifier λ and makes it the current simulator state (and raises an error if such a state is not in memory); STORE(λ), which stores into memory the current simulator state under identifier λ (and raises an error if λ already occurs in memory); FREE(λ), which frees simulator memory entry λ (and raises an error if no such entry exists); RUN(u, t), which injects input u and advances simulation by time $t \in \mathbb{T}$. The *time advancement* due to a command is the time simulated by $\mathcal{S}$ when executing it, and is t for RUN(u, t) and 0 for all the other commands.

A *simulation campaign* χ for $\mathcal{S}$ is a sequence CMD$_0$$(args_0)$ ... CMD$_{c-1}$$(args_{c-1})$ of simulator commands (with their arguments). To χ we can univocally associate: (i) the *sequence of states* traversed by the simulator while executing it; (ii) the *length* $len(\chi)$, which is the sum of the time advancements of its commands; (iii) the *required simulator memory* $mem(\chi)$, which is the maximum number of entries in the simulator memory among the traversed states; (iv) the *output sequence*, which is the sequence of the results of its OUTPUT commands (*i.e.*, the outputs of $\mathcal{S}$ in the states where an OUTPUT command is issued). A simulation campaign is *executable* if it does not raise errors.

Proposition 1 links inputs to a simulator $\mathcal{S}$ for $\mathcal{H}$ (*i.e.*, simulation campaigns) to inputs for $\mathcal{H}$ (input time functions), and lays the foundations to our SLV approach, ensuring that we can carry out a verification activity on $\mathcal{H}$ by properly driving its simulator $\mathcal{S}$. This will be the focus of Section 4.

Proposition 1. *Let $\mathcal{S}$ be a simulator for $\mathcal{H}$, χ an executable simulation campaign for $\mathcal{S}$, and $w_0, \ldots, w_c$ the associated sequence of simulator states. For each $i \in [0, c]$, the input time function $\mathbf{u}_i$ in $w_i = (x_i, \mathbf{u}_i, \mathcal{M}_i)$ is such to drive $\mathcal{H}$ from its initial state to x_i.*

4 Simulation-based SLV

To perform simulation-based SLV of $\mathcal{H}$ over n input traces $\mathcal{U}$ we need a simulator $\mathcal{S} = (\mathcal{H}, \mathcal{W})$ for $\mathcal{H}$ and an executable simulation campaign χ for $\mathcal{S}$ that somewhat drives $\mathcal{S}$ along the scenarios for $\mathcal{H}$ encoded by traces of $\mathcal{U}$ and collects the simulator outputs at the end of each scenario.

To this end, Definition 2 allows us to associate to any executable simulation campaign χ for $\mathcal{S}$ the sequence $\mathcal{U}(\chi)$ of SUV scenarios (as piecewise constant input time functions) for $\mathcal{H}$ actually explored by χ. Full definitions and statements in this section as well as their proofs are delayed to Appendix C.

Definition 2 (Sequence of input time functions associated to a simulation campaign)**.** *The sequence of input time functions associated to simulation campaign χ containing n OUTPUT commands is $\mathcal{U}(\chi) = \mathbf{u}_{j_0}, \ldots, \mathbf{u}_{j_{n-1}}$, where $\mathbf{u}_{j_i}$ is the input time function associated to the state where the simulator executes the i-th OUTPUT command of χ.*

Definition 3 formalises the notion of a simulation campaign aimed at computing the answer to an SLV problem.

Definition 3 (Simulation campaign for an SLV problem)**.** *A simulation campaign χ for SLV problem $\pi = (\mathcal{H}, \mathcal{U})$ is an executable campaign for a simulator $\mathcal{S}$ of $\mathcal{H}$, such that the sequence $\mathcal{U}(\chi) = \mathbf{u}_{j_0}, \ldots, \mathbf{u}_{j_{n-1}}$ of its associated input time functions is a* permutation *of $\mathcal{U}$.*

A simulation campaign χ for $\pi = (\mathcal{H}, \mathcal{U})$ can be used to compute the answer to π by executing χ on a simulator $\mathcal{S}$ for $\mathcal{H}$ and by collecting the simulator outputs during χ. If π aims at finding scenarios witnessing a property violation, the input function associated to any simulator state whose output is *FAIL* constitutes such an error scenario. Conversely, if π amounts to compute statistics on some KPIs of interest, the KPI values returned as the outputs of χ (at the end of each simulated scenario) can be used to build such statistics.

4.1 Shortest simulation campaigns

Among all the simulation campaigns for a given SLV problem, the shortest campaigns whose required simulator memory is bounded by a given constant $m \in \mathbb{N}_+$ (the simulator memory capacity) have a special interest (Definition 4).

Definition 4 (Shortest (m-memory) simulation campaign for a SLV problem)**.** *Let $m \in \mathbb{N}_+ \cup \{\infty\}$. A* shortest *$m$-memory simulation campaign χ for π is a simulation campaign for π such that $mem(\chi) \leq m$ and for which no other simulation campaign χ' for π exists such that $len(\chi') < len(\chi)$ and $mem(\chi') \leq m$. When $m = \infty$ (i.e., we do not put any limitation on the required simulator memory capacity to execute χ), we call χ simply a* shortest simulation campaign *for π.*

By definition, any shortest m-memory simulation campaign for π is not shorter than any shortest $(m+1)$-memory simulation campaign for π. Also, any shortest ∞-memory simulation campaign for π would actually require only a finite simulator memory capacity, which is upper bounded by the number m^* of the distinct longest sequences of disturbances occurring as prefixes of multiple traces of $\mathcal{U}$ (Longest Shared Prefixes, LSPs, see forthcoming Definition 13). Hence, any shortest ∞-memory simulation campaign for π would actually be a m^*-memory simulation campaign.

Computation of shortest simulation campaigns can be pursued by recalling that our SUV $\mathcal{H}$ is deterministic and needs to be simulated, for each scenario (input trace), starting from its initial state x_0. Hence, if two input traces $\mathbf{u}_a, \mathbf{u}_b \in \mathcal{U}$ have a common prefix, the SUV state at the end of such a prefix may be stored during simulation of the first simulated trace (*e.g.*, $\mathbf{u}_b$) and loaded back before simulating the other (*e.g.*, $\mathbf{u}_a$), whose inputs could then be injected from that point on only. This avoids repeated simulation of the common prefix.

This form of compression is particularly effective in practice, as the occurrence of multiple scenarios sharing a common prefix is very frequent when defining SUV operational environments. For example, in our case studies, the shortest simulation campaigns, as computed by our algorithm, are shorter than naïve ones by a factor of 5 to 8. This translates in similar speed-ups of the required overall simulation time (see Section 6).

4.2 Randomised simulation campaigns

Proposition 2 states that, if we put no limitation on the required memory capacity, a shortest simulation campaign exists for *any* ordering of the scenarios of the SLV problem at hand.

Proposition 2. *Let $\pi = (\mathcal{H}, \mathcal{U})$ be a SLV problem ($|\mathcal{U}| = n$) and $\mathcal{S}$ be a simulator for $\mathcal{H}$. For any permutation $\mathbf{u}_{j_0}, \dots, \mathbf{u}_{j_{n-1}}$ of input traces of $\mathcal{U}$, there exists an executable shortest simulation campaign χ for π on $\mathcal{S}$, such that $\mathcal{U}(\chi) = \mathbf{u}_{j_0}, \dots, \mathbf{u}_{j_{n-1}}$.*

However, the choice of the scenario verification order is an important issue. For example, as long as this order is deterministic, no partial conclusion can be drawn, during simulation, about the absence of error scenarios. This is because in a verification setting we need to adopt an adversarial model in which the adversary will place the single error scenario of $\mathcal{U}$ as the last scenario simulated by χ. Previous work [29], [32] shows that the upfront availability of all scenarios to be verified (set $\mathcal{U}$) allows us to adopt a simple yet very effective approach to draw, at *any time* during simulation, mathematically-sound partial conclusions on the probability that a property violation will be witnessed by a yet-to-be-simulated scenario. The idea is to choose our scenario verification order *uniformly at random* among all possible orders. With such a *randomised simulation campaign*, after having verified the absence of errors on the first $j \in [0, n]$ scenarios of $\mathcal{U}$ in the generated random order (where $n = |\mathcal{U}|$), the probability that an error will be found in a yet-to-be-simulated scenario (*omission probability*) is upper-bounded by $1 - \frac{j}{n}$. With this approach, we effectively conjugate *exhaustiveness* with *randomness*.

Randomising the scenario verification order is also required when approximations of statistics (*e.g.*, expected values) of KPIs for each scenario are to be computed with guaranteed accuracy via statistical model checking.

Efficiently computing a shortest, possibly randomised simulation campaign for our SLV problem is the purpose of Section 5.

4.3 Parallel simulation campaigns

As anticipated in Section 1.1, a major efficiency bottleneck for simulation-based SLV of industry-relevant CPSs is simulation time. This is due both to the typically very large number of scenarios to simulate (*e.g.*, up to almost 200 million in our case studies) and to the time needed to numerically simulate the CPS model (our SUV) on each such scenario (up to 80 seconds in our case studies).

The answer to an SLV problem $\pi = (\mathcal{H}, \mathcal{U})$ (*i.e.*, the collection of the simulator outputs at the end of each scenario) can be computed by arbitrarily partitioning $\mathcal{U}$ into $k \in \mathbb{N}_+$ subsets (*slices*) $\mathcal{U}_0, \dots, \mathcal{U}_{k-1}$ (where k is the number of available computational nodes), and by computing and taking the union of the answers to the k smaller SLV problems $\pi_i = (\mathcal{H}, \mathcal{U}_i)$, $i \in [0, k-1]$. In our simulation-based setting, this can be achieved using k simulators for $\mathcal{H}$ running as k independent processes (*e.g.*, in parallel in a HPC infrastructure) and independently driven by k simulation campaigns $\chi_1, \dots, \chi_k$, where, for all i, χ_i is a simulation campaign for π_i. Definition 5 formalises this concept.

Definition 5 (Parallel simulation campaign for an SLV problem)**.** *A k-parallel simulation campaign for SLV problem $\pi = (\mathcal{H}, \mathcal{U})$ is a tuple $\Xi = (\chi_0, \dots, \chi_{k-1})$ such that there exists a partition of $\mathcal{U}$ into sets $\mathcal{U}_0, \dots, \mathcal{U}_{k-1}$ such that, for all i, χ_i is a simulation campaign for $\pi_i = (\mathcal{H}, \mathcal{U}_i)$.*

The length *of χ is $len(\chi) = \max_{i=0}^{k-1} len(\chi_i)$. Given $m \in \mathbb{N}_+ \cup \{\infty\}$, Ξ is a k*-parallel m-memory simulation campaign *if all χ_is are m-memory simulation campaigns.*

The concepts of *shortest* and shortest m-memory simulation campaign are straightforwardly extended to parallel simulation campaigns.

As shown in [32], when the SLV activity seeks to certify absence of error scenarios, if all χ_is of a parallel simulation campaign $\Xi = (\chi_0, \dots, \chi_{k-1})$ are *randomised* (*i.e.*, each χ_i implements a verification order of the scenarios in $\mathcal{U}_i$ chosen independently and uniformly at random among all possible orders), then, at any time during the parallel simulation-based SLV activity, where χ_i has verified the absence of errors on the first $j_i \in [0, n_i]$ scenarios of $\mathcal{U}_i$ in the generated random order (where $n_i = |\mathcal{U}_i|$), the omission probability (*i.e.*, the probability that an error will be found in a yet-to-be-simulated scenario) is upper-bounded by $1 - \min_{i=0}^{k-1} \left(\frac{j_i}{n_i}\right)$.

5 Parallel computation of parallel simulation campaigns

We are now ready to present our algorithm to compute a parallel simulation campaign $\Xi = (\chi_0, \dots, \chi_{k-1})$ for the SLV problem $\pi = (\mathcal{H}, \mathcal{U})$ at hand. The computed Ξ can be executed on k simulators for $\mathcal{H}$ running independently on k nodes of a HPC infrastructure.

Full definitions, additional pseudocode and its description, as well as proofs of statements in this section are delayed to Appendix D.

5.1 Input

Our algorithm takes as input a collection $\mathcal{U}$ of $n \in \mathbb{N}_+$ input traces (encoding the scenarios on which the SUV must be verified) and the memory capacity $m \in \mathbb{N}_+$ of each of the k simulators in terms of the maximum number of states that each simulator can keep simultaneously stored.

Input traces are given either explicitly in the form of a database in mass memory, or symbolically, by means of a *scenario generator*, as designed in [38]. In particular, a scenario generator $\mathcal{G}$ is a symbolic data structure built from a set of requirements (or *constraints*), in turn defined by means of multiple automata (*monitors*). From $\mathcal{G}$, input traces of any horizon satisfying those requirements can be efficiently extracted from their unique indices. Namely, a scenario generator $\mathcal{G}$ offers two main functions: *nb_traces*()

and *trace*(). Function *nb_traces*($\mathcal{G}$) returns the number n of input traces entailed by $\mathcal{G}$, while, for $j \in [0, n-1]$, *trace*($\mathcal{G}, j$) extracts the j-th trace (in *lexicographic order*) from $\mathcal{G}$. When the set of scenarios is given via a scenario generator, the input traces are given as a *set of integers* $\mathcal{I}$ representing unique indices of traces to extracted from $\mathcal{G}$. In other words, when a scenario generator are involved, our set of input traces is defined as $\mathcal{U} = \{trace(\mathcal{G}, j) \mid j \in \mathcal{I}\}$.

5.2 Enabling parallelism

The typically very large number of input traces implies that $\mathcal{U}$ cannot be represented explicitly in central memory, and any form of global optimisation to find a shortest parallel m-memory simulation campaign would be unviable. Hence, our algorithm makes wise use of the available RAM and parallel computational nodes, and exploits suitable heuristics in order to compute an as-short-as-possible randomised m-memory simulation campaign. However, when the available capacity m of each simulator memory is above a certain threshold which depends on $\mathcal{U}$, the algorithm will indeed compute k *provably shortest m-memory simulation campaigns* $\chi_0, \ldots, \chi_{k-1}$ (Proposition 3).

Computing an as-short-as-possible parallel m-memory simulation campaign needs to heavily exploit the presence of multiple traces sharing a prefix. Hence, when splitting $\mathcal{U}$ into slices, it is important to keep as much as possible in the same slice traces sharing long common prefixes.

To this end, our algorithm works best when input traces can be accessed in lexicographic order (according to the total order defined over the SUV input space $\mathbb{U}$), since in this case it can easily keep in the same slice traces that are close together according to the lexicographic order.

Accessing traces in lexicographic order is immediate when they are extracted from a scenario generator, since it would be enough to access them in ascending order of their indices. Hence, in this case slicing is performed by simply partitioning of the set of indices $\mathcal{I}$ of the traces selected for SLV into k evenly-long sequences $\mathcal{I}_0, \ldots, \mathcal{I}_{k-1}$, where each such sequence defines trace indices in ascending order and, for each $i > 0$ the trace indices in the i-th slice are all larger than those in the $(i-1)$-th slice. The i-th slice of traces would then simply be: $\mathcal{U}_i = \{trace(\mathcal{G}, j) \mid j \in \mathcal{I}_i\}$, $i \in [0, k-1]$.

Conversely, when input traces are extracted from a database, standard mass-memory sorting algorithms are exploited to reorder them lexicographically. Even when the number of traces is very large, such mass-memory sorting algorithms offer good scalability and can be effectively used for this purpose. In particular, as shown in Section 6.4, the advantages (in terms of savings in the simulation time) achieved by performing SLV using optimised simulation campaigns heavily outperform the additional cost of ordering them if needed, and this justifies investing computation time in such a preprocessing.

For each slice, a desired, possibly randomised, verification order can be easily defined by the user. For example, a uniformly random verification order can be computed by computing a random permutation of trace indices (when traces are extracted from a scenario generator) or of their keys (when pre-sorted in mass-memory databases, see, *e.g.*, [32]).

5.3 Computing a simulation campaign from each slice

From this point on, computation of the parallel simulation campaign $\Xi = (\chi_0, \ldots, \chi_{k-1})$ proceeds *embarrassingly in parallel*, using up to k *independent* computational nodes, one for each slice. Our algorithm to compute a simulation campaign for a single slice $\mathcal{U}_i$ is sketched as Algorithm 1.

```
1 input $\mathcal{U}_i$, slice of traces in desired (e.g., random) order
    input $\tau \in \mathbb{T}$, time quantum input $m \in \mathbb{N}_+$, the
    simulator memory capacity
2 output $\chi_i$, the output simulation campaign
3 $\chi_i \leftarrow$ an empty sequence of commands;
4 $\mathcal{T} \leftarrow LSPT(\mathcal{U}_i)$ ; /* build Longest Shared Prefix Tree */
5 $j \leftarrow 0$; /* trace counter */
6 foreach $\mathbf{u} \in \mathcal{U}_i$ (in the given order) do
7     append $sim\_cmds(\mathbf{u}, j, \mathcal{T})$ to $\chi_i$; $j$++;
8 return $\chi_i$;
```

Algorithm 1: Simulation campaign computation for a slice.

5.3.1 Longest Shared Prefix Tree

The first step of Algorithm 1 (function *LSPT()*) is to build a data structure called *Longest Shared Prefix Tree (LSPT)*, representing the longest prefixes shared by multiple traces.

In the following, given two (possibly empty) sequences of inputs $\mathbf{u}_a$ and $\mathbf{u}_b$ (*i.e.*, sequences of values of $\mathbb{U}$), we denote by $\mathbf{u}_a \sqsubseteq \mathbf{u}_b$ (respectively, $\mathbf{u}_a \sqsubset \mathbf{u}_b$) the fact that $\mathbf{u}_a$ is a prefix (respectively, proper prefix) of $\mathbf{u}_b$.

A *Longest Shared Prefix (LSP)* for $\mathcal{U}_i$ is a (possibly empty) sequence $\mathbf{u}$ of inputs such that there exist two traces $\mathbf{u}_a$ and $\mathbf{u}_b$ in $\mathcal{U}_i$ such that: $\mathbf{u} \sqsubseteq \mathbf{u}_a$, $\mathbf{u} \sqsubseteq \mathbf{u}_b$, and there exists no $\mathbf{u}'$ in $\mathcal{U}_i$ such that $\mathbf{u} \sqsubset \mathbf{u}'$, $\mathbf{u}' \sqsubseteq \mathbf{u}_a$, and $\mathbf{u}' \sqsubseteq \mathbf{u}_b$. The intelligent storing of the states reached by the simulator after having executed such LSPs (under the available simulator memory capacity constraints) would avoid their recomputation, thus producing shorter simulation campaigns.

A *Longest Shared Prefix Tree (LSPT)* for $\mathcal{U}_i$ (see Appendix D.1.1 for formal statements, details, and pseudocode) is a tree $\mathcal{T} = (V, parent)$. Nodes (set V) denote distinct LSPs of $\mathcal{U}_i$ and the parent node $parent(\mathbf{u})$ of node $\mathbf{u}$ (if one exists) is such that $parent(\mathbf{u}) \sqsubset \mathbf{u}$, and no sequence $\mathbf{u}'$ exists as a node of $\mathcal{T}$ such that $parent(\mathbf{u}) \sqsubset \mathbf{u}' \sqsubset \mathbf{u}$. The latter condition implies that a LSPT is a rooted tree.

The *depth* of LSPT node $\mathbf{u} = (u_0, \ldots, u_{d-1})$ is $depth(\mathbf{u}) = d$, which represents the time point $d\tau$ reached by the simulator (starting from its initial state) after having injected input sequence $\mathbf{u}$. The depth of the node associated to the empty sequence is zero. To each node $(u_0, \ldots, u_{d-1}) \in V$, the number of traces in $\mathcal{U}_i$ having $(u_0, \ldots, u_{d-1})$ as a (proper or non-proper) prefix is stored as $ntraces(u_0, \ldots, u_{d-1})$.

A LSPT $\mathcal{T}$ for $\mathcal{U}_i$ is *complete* if no LSPT for $\mathcal{U}_i$ exists whose nodes are a proper subset of those of $\mathcal{T}$. The *size* of a LSPT is the number of its nodes.

To compute a complete LSPT for $\mathcal{U}_i$ in central memory, function *LSPT()* scans $\mathcal{U}_i$ in lexicographic order, since, under this ordering, deciding which trace prefixes are nodes of the tree is straightforward and memory-efficient.

To keep an as small as possible RAM footprint of the LSPT, the algorithm represents in central memory each of its nodes $(u_0, \ldots, u_{d-1})$ by a unique identifier $\lambda(u_0, \ldots, u_{d-1})$. Unique identifiers for each trace prefix are available for free when traces are extracted from a scenario generator. If traces are taken from a database, any efficiently computable injective function of finite sequences of input values (or even a cryptographic hash function, when the probability of conflicts is small enough) can be used.

5.3.2 Generation of simulation campaign commands

Algorithm 1 proceeds at generating an optimised simulation campaign χ_i which would drive simulator $\mathcal{S}_i$ along all the input traces according to the chosen (possibly random) order, still trying to save as many simulation steps as possible, compatibly with simulator memory capacity constraints.

To this end, the input traces $\mathcal{U}_i$ are considered sequentially in the given order. For each trace $\mathbf{u}$, function *sim_cmds()* is invoked to append to χ_i a sequence of commands to simulate it from the *best* intermediate state available in the simulator memory (see below). During generation of simulator commands, for each LSPT node λ, the algorithm keeps a boolean flag *stored(λ)* (initialised to *false*) whose value reflects, at any point during the computation of χ_i, the fact that state λ would be available or not in the memory of $\mathcal{S}_i$ at that point during the execution of χ_i. Namely, *stored(λ)* is set to *true* (respectively, *false*) when issuing a STORE(λ) (respectively, FREE(λ)) command.

Generating trace simulation commands. Algorithm 2 shows the pseudocode of function *sim_cmds()* which issues the actual commands aimed at simulating trace $\mathbf{u}$, which are appended to χ_i. The function proceeds as follows:

1. Selects λ_{load}, the state corresponding to the longest prefix of $\mathbf{u}$ that, at the current point of the prospective simulation, would be available in the simulator memory and appends command LOAD(λ_{load}) to χ_i, to load it back.
2. Revises the nodes of the LSPT associated to prefixes of $\mathbf{u}$ (proceeding backwards from the full $\mathbf{u}$). For each such LSPT node λ_q, value *ntraces(λ_q)* is decremented (thus memorising the fact that such prefix will occur in one less future trace). If *ntraces(λ_q)* becomes zero, the algorithm knows that the input sequence associated to λ_q will not occur as a prefix in any future trace, and removes λ_q from the LSPT (which, since prefixes of $\mathbf{u}$ are processed backwards from the entire $\mathbf{u}$, is a leaf of the LSPT). Also, if λ_q is known to be stored in the simulator memory at this point of the execution of χ_i (*i.e.*, *stored(λ_q) = true*), it appends to χ_i command FREE(λ_q) to free-up the simulator memory.
3. Appends to χ_i a RUN command for each maximally long constant portion of $\mathbf{u}$ such that no intermediate state traversed by the simulator needs to be stored to shorten simulation of future traces (*i.e.*, function *worth_storing()* returns *false* for it).
4. If the state reached by the simulator after each RUN command is worth to be stored as it can shorten simulation of a later trace (this implies it is a node of the LSPT), the function proceeds at storing it (see below).

Storing intermediate simulation states. Given the limited capacity m of the simulator memory, the decision of which LSPT simulator states will be actually stored must be taken wisely. This is charge of function *worth_storing()*.

```
1  function sim_cmds(u, j, $\mathcal{T}$)
2      input u = $(u_0, \ldots, u_{h-1})$, current (j-th) trace
3      input $\mathcal{T} = (V, parent)$, Longest Shared Prefix Tree
         output sequence of sim. commands for u
4      if $j = 0$ then load ← 0;                          /* first trace */
5      else                                              /* not first trace */
6          load ← max $q \in [0, h]$ s.t.
7              $\lambda_{load} = \lambda(u_0, \ldots, u_{q-1}) \in V \wedge stored(\lambda_{load})$;
8          issue LOAD($\lambda_{load}$);
9      for q from $h - 1$ downto 0 s.t.
         $\lambda_q = \lambda(u_0, \ldots, u_{q-1}) \in V$ do        /* revise $\mathcal{T}$ */
10         ntraces($\lambda_q$)--;
11         if ntraces($\lambda_q$) = 0 then
               /* $\lambda_q$ won't occur in future traces          */
12             if stored($\lambda_q$) then
13                 issue FREE($\lambda_q$); stored($\lambda_q$) ← false;
14             remove $\lambda_q$ from V; /* $\lambda_q$ is leaf in $\mathcal{T}$    */
       /* All nodes still in $\mathcal{T}$ will occur in future traces     */
15     start ← load;
16     while start < h do
17         end ← max $e \in [start, h-1]$ s.t. $\forall q \in [start+1, e]$
18             $u_q = u_{q-1} \wedge \neg worth\_storing(\lambda(u_0, \ldots, u_{q-1}), \mathcal{T})$;

19         issue RUN($u_{start}$, $(end - start + 1)\tau$) ;
20         start ← end + 1;
21         if $start \le h \wedge \lambda_{start} = \lambda(u_0, \ldots, u_{start-1}) \in V \wedge$
             worth_storing($\lambda_{start}, \mathcal{T}$) then
22             do_store($\lambda_{start}, \mathcal{T}, m, \chi_i$) ; /* possibly issues
                 FREE($\lambda'$) for some $\lambda'$ s.t. stored($\lambda'$) and sets
                 stored($\lambda'$) to false, before issuing
                 STORE($\lambda_{start}$)                              */
23             stored($\lambda_{start}$) ← true;
24     issue OUTPUT;
```

Algorithm 2: Function *sim_cmds()*.

Since the LSPT has no information on the *order* with which simulator states represented by LSPT nodes will occur in $\mathcal{U}_i$ (such data would be too large to be kept in RAM), any approach to compute an optimal plan to decide which intermediate state to store and free (and when to do that during the execution of the simulation campaign) is clearly not viable. Hence, the function proceeds *heuristically*.

In particular, *worth_storing(λ, $\mathcal{T}$)* works as follows. If λ is not a LSPT node or is expected to be already stored in memory at that point of the execution of the simulation campaign (*i.e.*, *stored(λ) = true*), then *worth_storing()* returns *false*; otherwise, if the simulator memory is expected to have room to accommodate an additional state (*i.e.*, the number of LSPT nodes λ' such that *stored(λ') = true* is $< m$), then *worth_storing()* returns *true*.

In case the simulator memory is expected to be full at that point of the execution of the simulation campaign, then the function decides whether it is best to make space for λ by freeing up another simulator state λ' already in memory, or to rather ignore the request of storing λ in the first place.

To this end, the function searches for a currently stored state λ' whose associated node in the LSPT is not the root node and has the smallest depth-difference with respect to its parent node, where the depth-difference of λ' is *depth*$(\lambda') -$ *depth*(*parent*$(\lambda')) > 0$. Since the depth-difference of a simulator state defines the additional number of τ-simulation steps needed by the simulator to reach that state when starting from the state represented by its parent node in the LSPT, λ' is a currently stored state which could be used to shorten simulation of a future trace, but whose removal from simulator memory minimises the number of additional τ-simulation steps needed to recompute it (from the state associated to its parent node in the LSPT).

In case the depth-difference of λ' is less than that of λ, then the function decides that it is worth removing λ' from the simulator memory to make room for λ, and returns *true*. Otherwise, the function knows that freeing-up λ' to make room for λ would cost more (in terms of additional τ-long simulation steps to recompute λ' from the state represented by its parent) than simply ignoring the request to store λ, and returns *false*.

When *worth_storing()* returns *true*, function *do_store()* appends STORE(λ) to χ_i, preceded by FREE(λ') in case *worth_storing()* has selected λ' as the state to be freed-up (in which case *stored*(λ') is set to *false* as well).

In order to efficiently find λ', the currently stored LSPT nodes are indexed so as to retrieve efficiently those having minimal depth-difference with respect to their parents.

The following result holds (see Appendix D.2 for the full statement and proof).

Proposition 3 (Correctness of Algorithm 1)**.** *Let $\pi = (\mathcal{H}, \mathcal{U})$ be an SLV problem for SUV $\mathcal{H}$, with input traces $\mathcal{U}$ being associated to time quantum τ, and let m be a positive integer. Given any partition $\{\mathcal{U}_0, \ldots, \mathcal{U}_{k-1}\}$ of $\mathcal{U}$, let $\Xi = (\chi_0, \ldots, \chi_{k-1})$ be the k-parallel simulation campaign such that χ_i is computed by Algorithm 1 on inputs $\mathcal{U}_i$ (under any user-defined order), τ, and m. We have that:*

1. *For all $i \in [0, k-1]$, the sequence $\mathcal{U}(\chi_i)$ is $\mathcal{U}_i$;*
2. *There exists $m^* \in \mathbb{N}_+$ such that, if $m \geq m^*$, all χ_is are* shortest *m-memory simulation campaigns.*

Point 1. implies that Ξ is a k-parallel m-memory simulation campaign for π. Each χ_i drives an independent copy of a simulator of SUV $\mathcal{H}$ along the scenarios in $\mathcal{U}_i$ in the chosen, possibly random, order. In the latter case, an upper bound to the omission probability can be computed at any time during parallel simulation (Section 4.3).

6 Implementation and Experimental Results

In this section we outline our implementation of the parallel algorithm of Section 5 and analyse its performance and scalability on three real case studies.

6.1 Implementation

We implemented our algorithm as a C-language tool which takes as input positive integers k (number of slices) and m (memory capacity of each simulator), and the set of input traces $\mathcal{U}$ for which a parallel campaign is sought. In our experiments we extracted the set of input traces from scenario generators, defined as discussed in [38]. The computed campaign can be executed on k simulators for the SUV $\mathcal{H}$, running independently on k computational nodes. Each simulator is steered by a *driver* which receives the simulation campaign as input. This driver is the only simulator-dependent component of our tool pipeline. We implemented drivers for two popular simulators, namely: Simulink and JModelica/FMU. Additional drivers can be easily written along the same lines.

6.2 Case studies

We selected three industry-relevant SUV models defined in the language of two popular simulators, namely Simulink and Modelica.

6.2.1 Buck DC-DC Converter (BDC)

It is a mixed-mode analog circuit converting the DC input voltage (denoted as V_i) to a desired DC output voltage (V_o), often used off-chip to scale down the typical laptop battery voltage (12–24 V) to the few volts needed by, *e.g.*, a laptop processor (the *load*) as well as on-chip to support dynamic voltage and frequency scaling in multicore processors (see, *e.g.*, [40]). A BDC converter is self-regulating, *i.e.*, it is able to maintain the desired output voltage V_o notwithstanding variations in the input voltage V_i or in the load R. We used a Modelica model of the fuzzy logic–based BDC controller of [47], converted into an FMU 2.0 object via the JModelica extension in [45].

6.2.2 Apollo Lunar Model Autopilot (ALMA)

It is a Simulink/Stateflow model defining the logic that implements the phase-plane control algorithm of the autopilot of the lunar module used in the Apollo 11 mission. The Module is equipped with actuators (16 reaction jets to rotate the Module along the three axes) subject to temporary unavailabilities. The controller takes as input requests to change the Module attitude (*i.e.*, to perform a rotation along the three axes) and computes which reaction jets to fire to obey each request.

6.2.3 Fault Tolerant Fuel Control System (FCS)

It is a Simulink/Stateflow model of a controller for a fault tolerant gasoline engine, which has also been used as a case study in [11], [25], [26], [28], [32], [33], [55]. The FCS has four sensors subject to temporary faults, and the whole control system is expected to tolerate single sensor faults.

6.3 Experimental setting

We defined a scenario generator for each SUV, entailing input traces (time quantum $\tau = 1$ time unit, t.u.) with the properties listed in Table 1. Several constraints have been enforced on the input traces. This allows us to focus the SLV activities on clearly selected portions of the space of inputs and to keep the overall number of traces under control. The enforced constraints are detailed in Appendix E. Here, we just point out that we experimented with the optimisation of parallel simulation campaigns for up to around 50 (BDC), 100 (ALMA) and 200 (FCS) million traces.

To show scalability of our algorithm when computing optimised parallel simulation campaigns as well as the overall savings in simulation time provided by our approach to SLV, we exploited (virtually) up to 1024 identical 64-core machines (CPU: AMD EPYC 7301, RAM: 256GB) of our HPC

SUV	$\lvert\mathbb{U}\rvert$	horizon	n. traces	constraints on traces
BDC	25	60 t.u.	49 971 109	Appendix E.1
ALMA	432	100 t.u.	107 535 209	Appendix E.2
FCS	6	100 t.u.	195 869 671	Appendix E.3

Table 1: Scenario generators for our case studies.

infrastructure, thus our maximum number of slices k has been set to 65 536.

Since actual simulation of the generated campaigns in all the considered settings would be prohibitively long, simulation time of each campaign has been estimated as follows. For each SUV, we generated and actually simulated a random campaign of 100k commands, where each command (LOAD, STORE, FREE, OUTPUT, RUN for all needed durations) was evenly represented. We then computed the average time needed by the simulator to execute each single command, and used such expected values (standard deviation showed to be negligible) to estimate the completion time of each campaign.

6.4 Experimental results

Our experimental results are summarised in Figure 1 (BDC), Figure 2 (ALMA) and Figure 3 (FCS). We computed several *randomised* parallel simulation campaigns for each case study, one of each of several random subsets of all the traces entailed by our scenario generator.

In order to show performance and effectiveness of simulation campaign optimisation in contexts ranging from statistical model checking to random exhaustive verification, we sampled trace subsets by fixing their size from 25% to 100% of the overall number of traces.

Each experiment has been repeated for various amounts of simulator memory available (1 state, meaning no optimisation at all, since only one simulator state –typically the initial state– can be be stored and loaded back, up to m^*, the maximum number of states required in each experiment for maximum optimisation). Given the presence of randomisation, all experiments have been repeated with 5 different random seeds, and all results have been averaged.

6.4.1 Scalability of the campaign computation algorithm

The first (left-most) column of Figures 1 to 3 shows the time (in seconds) needed by our algorithm to compute a parallel simulation campaign for each SUV and each combination of values for the number of traces (row), the number of parallel processes (slices), and the amount of simulator memory (different line shapes).

The plots show that the computation time ranges from a few seconds to a few hours, and this time is *always negligible* when compared to the time savings that such optimisation yields in terms of simulation time (see the corresponding plots on right-most column, where time is expressed in days of parallel computation).

6.4.2 Campaigns efficiency with respect to parallelisation

The second column of Figures 1 to 3 shows how efficiency of the computed campaigns is preserved when a higher number of parallel processes are expected to be used in the verification process (hence, the input traces are split in a higher number of slices).

Namely, for each SUV and each combination of values for the number of traces n (row), the number of parallel processes (slices) k, and the amount of simulator memory m (different line shapes), the charts plot the average value (among our randomised experiments) of the following quantity: $\frac{\textit{sim\_time}(\chi_{n,1024,m})\times 1024}{\textit{sim\_time}(\chi_{n,k,m})\times k}$, which measures, in terms of (estimated) simulation time (*sim_time*), the efficiency of the parallel simulation campaign $\chi_{n,k,m}$ (which verifies n random traces in parallel on k processes assuming that each simulator can keep m states simultaneously stored) with respect to the corresponding parallel simulation campaign $\chi_{n,1024,m}$ (which verifies the same traces under the same assumptions regarding the simulator memory, but running on just 1024 parallel processes, our minimum value).

The plots show how efficiency is *always very high*, and, even when it degrades to a bit less than 90%, the induced overhead in simulation time is *always negligible* when compared to the *very large time savings* yielded by exploiting a higher number of parallel simulators.

6.4.3 Campaigns efficiency with respect to available simulator memory

The third column of Figures 1 to 3 shows how efficiency of the computed campaigns is preserved when reducing the memory available on each simulator.

Namely, for each SUV and each combination of values for the number of traces n (row), the number of parallel processes (slices) k, and for each value for the amount of simulator memory m (different line shapes), the charts plot the average value of the following quantity: $\frac{\textit{sim\_time}(\chi_{n,k,m^*})}{\textit{sim\_time}(\chi_{n,k,m})}$, which measures, in terms of simulation time (*sim_time*), the efficiency of the parallel simulation campaign $\chi_{n,k,m}$ (which verifies n random traces in parallel on k processes assuming that each simulator can keep only m states simultaneously stored) with respect to the corresponding parallel simulation campaign χ_{n,k,m^*} (which verifies the same traces with the same number of parallel processes, but assuming maximum simulator memory, *i.e.*, $m = m^*$).

The plots show how efficiency is *very well preserved* when reducing the value for m to up to $m^* \times 50\%$, unsurprisingly degrading for lower values of m. We also point out that the maximum memory required to each simulator (*i.e.*, when $m = m^*$) is always very limited, and easily met in practice. Namely, since simulator states occupy at most a few dozens of Kilobytes, the memory requirements are always less than (upper limits reached for 1024 parallel processes/slices): 2GB for BDC ($m^* \leq 15\,681$); 4GB for ALMA ($m^* \leq 62\,050$); 8GB for FCS ($m^* \leq 156\,115$).

6.4.4 Simulation speedups and time savings

The fourth column of Figures 1 to 3 shows the speedups in simulation time achieved by our computed optimised campaigns under different settings regarding the memory available on each simulator.

Namely, for each SUV and each combination of values for the number of traces n (row), the number of parallel processes (slices) k, and for each value for the amount of simulator memory m (different line shapes), the charts plot the average value of the following quantity: $\frac{\mathit{sim\_time}(\chi_{n,k,1})}{\mathit{sim\_time}(\chi_{n,k,m})}$, which measures, in terms of simulation time (*sim_time*), the speedup of each parallel simulation campaign $\chi_{n,k,m}$ (which verifies n random traces in parallel on k processes assuming that each simulator can keep only m states simultaneously stored) with respect to the corresponding campaign $\chi_{n,k,1}$ (which verifies the same traces with the same number of parallel processes, but assuming that each simulator can keep simultaneously stored only one state, that is no optimisation at all). The plots show how our simulation campaign optimiser always achieves *very significant speedups*, up to more than $8\times$.

The fifth column of Figures 1 to 3 shows how these speedups translate in *huge reductions in simulation time* (in days). Namely, for each SUV and each combination of values for n, k, and m, the charts plot the average value of the overall simulation time of the parallel simulation campaigns $\chi_{n,k,m}$, which verify the given SUV on n random traces under simulator memory setting m. The plots clearly show that *our simulation campaign optimiser makes practically viable* (in some days or at most weeks of parallel simulation) *verification tasks that would take an inconceivable long time without optimisation* (*i.e.*, when $m = 1$).

6.5 Limitations

Our optimised campaigns heavily rely on storing and loading back intermediate simulator states to avoid simulating common prefixes of different traces multiple times. Hence, for SUV models exhibiting very large states (*e.g.*, those defined via partial differential equations, transport delays, or variable delay blocks), the time to execute STORE and LOAD commands may become substantial, and this raises a question on whether it would be faster to skip optimisation altogether and just run the non-optimised campaigns. Here we briefly discuss this issue.

In the case of SUV models showing larger states than ours, but which are also proportionally slower to advance, the speedups enabled by the campaign optimisation would be somewhat preserved. Thus, the problematic situations for our optimiser occur when dealing with SUV models whose states are larger, but whose simulation is only sub-proportionally slower to advance.

To assess to what extent our optimised campaigns still grant time savings with respect to the non-optimised campaigns, we reconsidered our experiments by *artificially inflating* the duration of STORE and LOAD commands by a factor f ranging from 1 to 100, keeping unchanged the duration of RUN commands. Thus, we placed ourselves in the most hostile setting, *i.e.*, the verification of variations of our SUV models that, although requiring the *same* time to be advanced, have larger states which need f *times* the time need by our original SUV models to be stored and loaded back.

Unsurprisingly, the speedups achieved by optimised campaigns gradually decrease when f increases, but *still typically grant substantial savings in simulation time*. For example, the speedups achieved for our case studies (100% traces) fall to: 2.2–2.4$\times$ (BDC), 4.3–6.0$\times$ (ALMA), 3.7–5.5$\times$ (FCS) for $f = 10$; 1.0–1.3$\times$ (BDC), 2.5–3.2$\times$ (ALMA), 2.5–3.1$\times$ (FCS) for $f = 50$; 0.9–1.0$\times$ (BDC), 1.6–2.4$\times$ (ALMA), 1.7–2.3$\times$ (FCS) for $f = 100$.

7 Related work

Black-box simulation-based SLV of cyber-physical systems has been widely addressed in the literature. For example, simulation-based reachability analysis for large linear continuous-time dynamical systems has been investigated in [6], [14]. A simulation-based data-driven approach to verification of hybrid control systems described by a combination of a black-box simulator for trajectories and a white-box transition graph specifying mode switches has been investigated in [17]. Formal verification of discrete time Simulink models (*e.g.*, Stateflow or models restricted to discrete time operators) with small domain variables has been investigated in, *e.g.*, [9], [41], [49], [52]. However, none of the approaches above supports simulation-based bounded model checking of arbitrary simulation models on a (typically extremely large) set of operational scenarios given as input, and none of them addresses the issue of simulation campaign optimisation.

To the best of our knowledge, the only available literature which deals with simulation campaign optimisation is our previous works [28], [30], where preliminary versions of our algorithm have been presented. With respect to those conference papers, the current article presents a new, more scalable algorithm which guarantees to compute a shortest simulation campaign when enough simulator memory is allowed, and exploits various heuristics to compute an as short campaign as possible even when such memory requirements are not met. Our algorithm computes simulation campaigns that obey to the verification order decided by the user, possibly randomised so as to compute, at any time during simulation, an upper bound to the omission probability, using the results of [32].

Our algorithm takes as input a set of operational scenarios that can be provided in several ways, *e.g.*, from a high-level constraint-based model as discussed in [38], or as a mass-memory database of scenarios. This allows us to seamlessly support both (random) exhaustive verification (when the given scenarios completely define the set of operational scenarios of interest for the verification task) and statistical model checking (when the given scenarios are a random sample of such scenarios).

When exhaustive verification is not a viable option, given the huge number of scenarios of interest, simulation-based statistical model checking is often preferred, in order to compute statistically-sound information about the SUV properties of interest from a random sample of the possible scenarios, see, *e.g.*, [7], [8], [10], [18], [19], [20], [23], [23], [24], [53], [54]. Simulation-based statistical model checking has been successfully applied in several domains, *e.g.*, Simulink CPS models [12], [55], mixed-analog circuits [11]; smart grid control policies [21], [35], [36], [37]; biological models [39], [42], [46], [50]. Finally, simulation-based *falsification* of CPS properties (*e.g.*, for Simulink models) has been extensively investigated. Examples are in [1], [2], [5], [13], [15], [22], [44], [51] and citations thereof. Some of such works also propose suitable data-structures (*e.g.*, tree-like) to represent the set of possible traces, as we do.

Our simulation campaign optimisation algorithm is *independent* of the chosen verification technique, and the computed campaigns would bring *significant speedups* in terms

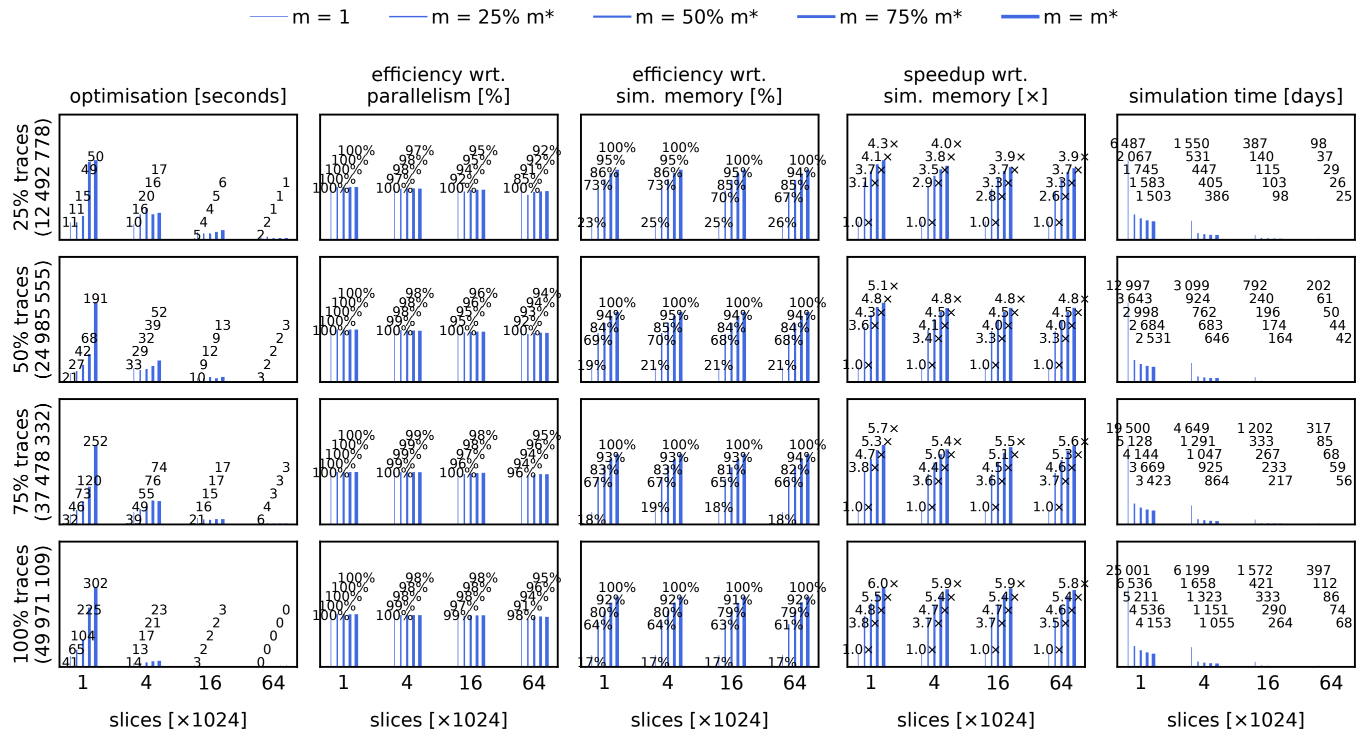

Figure 1: Experimental results: Buck DC-DC Converter (BDC).

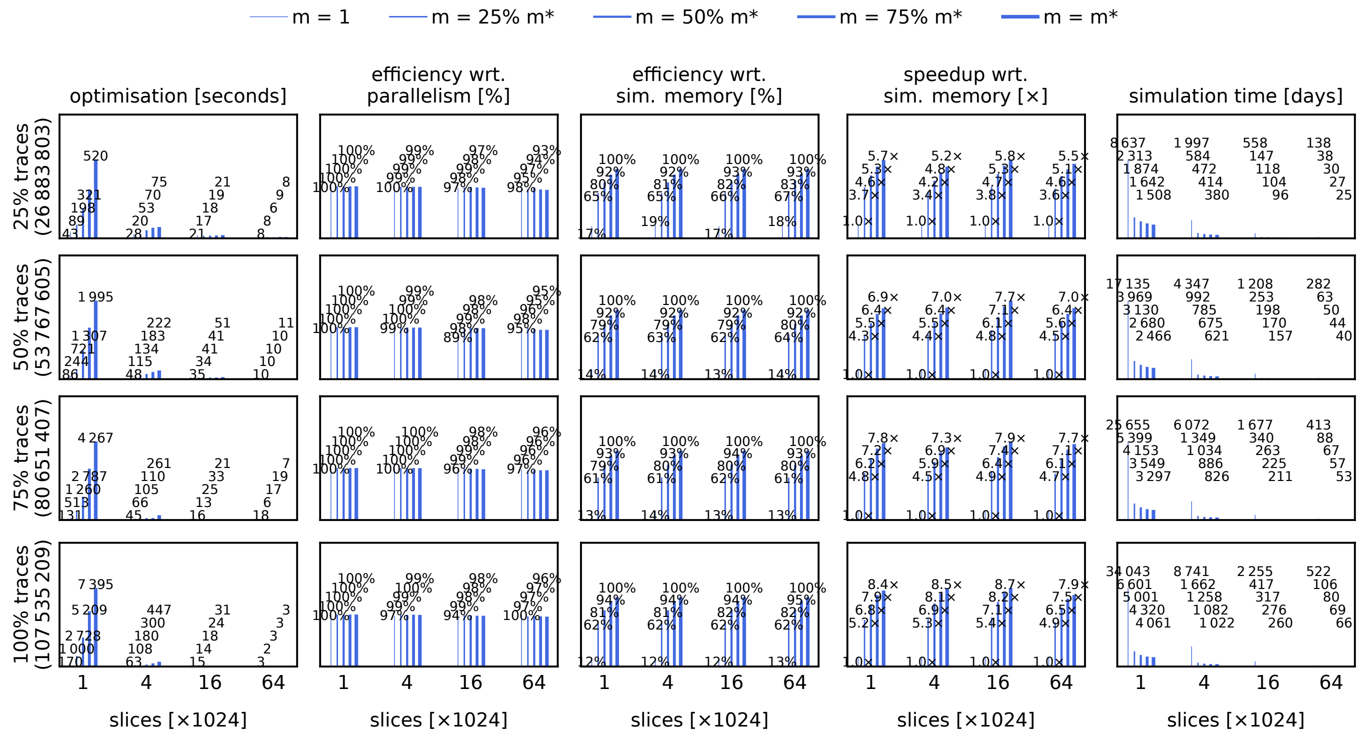

Figure 2: Experimental results: Apollo Lunar Model Autopilot (ALMA).

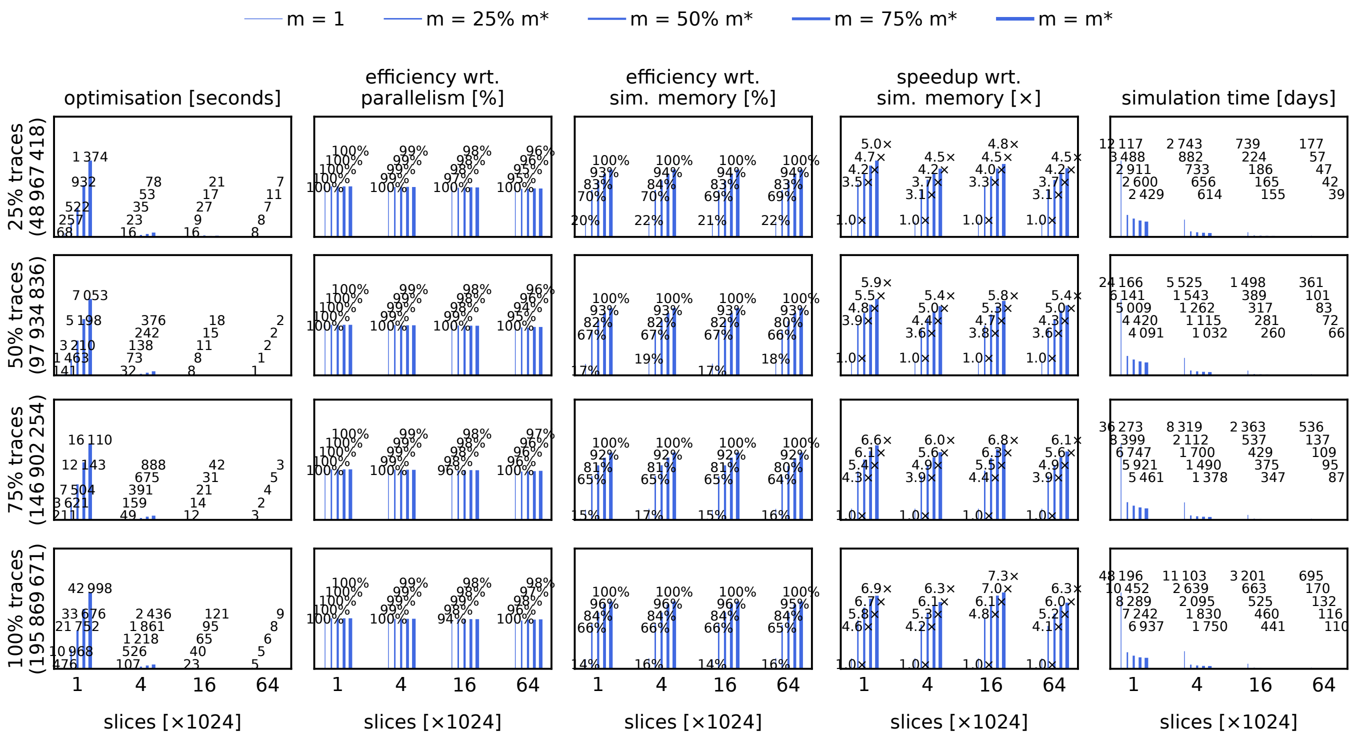

Figure 3: Experimental results: Fuel Control System (FCS).

of simulation time to all of them. For example, the first row of Figures 1 to 3 shows that speedups up to around 6× in simulation time can be achieved even when a small random sample (only 25%) of the entire sets of scenarios is chosen to perform statistical model checking.

The ability to perform *parallel* verification of the SUV is also a key enabler to make simulation-based SLV of industry-scale CPSs practically viable. Parallel approaches have been investigated, see *e.g.*, [4] in the context of probabilistic properties. Our approach seamlessly allows *massive embarrassingly parallel* verification. This is because, once the input set of scenarios has been split into slices, a parallel simulation campaign is computed, which is used to feed *independent* verification processes to be run in parallel.

8 Conclusions

In this article we focused on the generation of *optimised simulation campaigns* to carry out SLV of CPSs using arbitrarily many simulators of the system model running in parallel in a large HPC infrastructure, with the goal of *minimising the overall completion time*.

By taking as input a user-defined collection of (a random sample of) operational scenarios of interest from either a mass-storage database or a symbolic structure such as a constraint-based scenario generator in a (possibly random) user-defined order, our optimiser computes shortest parallel campaigns which exercise the system model on all (and only) the given scenarios. Our campaigns greatly speed-up verification by wisely avoiding the repeated computation of recurrent system trajectories as much as possible, compatibly with simulator memory constraints.

Our experiments on SLV of Modelica/FMU and Simulink case study models with up to *almost 200 million scenarios* show that our optimisation yields *speedups as high as* 8× and scales very well to large HPC infrastructures (efficiency almost always ≥ 90% even when using 65 536 computational nodes, *i.e.*, 1024 64-core parallel machines).

The conjoint exploitation of simulation campaign optimisation and massive parallelism makes practically viable (a few weeks in a HPC infrastructure) verification tasks (both exhaustive and statistical) which would otherwise take *inconceivably* long time.

Acknowledgements

The authors are grateful to the anonymous reviewers for their comments and suggestions.

This work was partially supported by: Italian Ministry of University and Research under grant "Dipartimenti di eccellenza 2018–2022" of the Department of Computer Science of Sapienza University of Rome; EC FP7 project PAEON (Model Driven Computation of Treatments for Infertility Related Endocrinological Diseases, 600773); EC FP7 project SmartHG (Energy Demand Aware Open Services for Smart Grid Intelligent Automation, 317761); INdAM "GNCS Project 2022"; Sapienza University projects RG12117A8B393BDC, RG11916B892E54DB, RG120172B9329D33; Lazio POR FESR projects E84G20000150006, F83G17000830007; NRRP Mission 4, Comp. 2, Inv. 1.5, NextGenEU, MUR CUP B83C22002820006, Rome Technopole.

REFERENCES

[1] H. Abbas, G. Fainekos, S. Sankaranarayanan, F. Ivančić, and A. Gupta. Probabilistic temporal logic falsification of cyber-physical systems. *ACM TECS*, 12(2s), 2013.

[2] A. Adimoolam, T. Dang, A. Donzé, J. Kapinski, and X. Jin. Classification and coverage-based falsification for embedded control systems. In *CAV 2017*, volume 10426 of *LNCS*. Springer, 2017.

[3] G. Agha and K. Palmskog. A survey of statistical model checking. *ACM Trans. Model. Comput. Simul.*, 28(1), 2018.

[4] M. AlTurki and J. Meseguer. PVeStA: A parallel statistical model checking and quantitative analysis tool. In *CALCO 2011*, volume 6859 of *LNCS*. Springer, 2011.

[5] Y.S.R. Annpureddy, C. Liu, G. E. Fainekos, and S. Sankaranarayanan. S-TaLiRo: A tool for temporal logic falsification for hybrid systems. In *TACAS 2011*, volume 6605 of *LNCS*. Springer, 2011.

[6] S. Bak and P.S. Duggirala. Simulation-equivalent reachability of large linear systems with inputs. In *CAV 2017*, volume 10426 of *LNCS*. Springer, 2017.

[7] A. Basu, S. Bensalem, M. Bozga, B. Caillaud, B. Delahaye, and A. Legay. Statistical abstraction and model-checking of large heterogeneous systems. In *Formal Techniques for Distributed Systems*. Springer, 2010.

[8] J. Bogdoll, L.M.F. Fioriti, A. Hartmanns, and H. Hermanns. Partial order methods for statistical model checking and simulation. In *Formal Techniques for Distributed Systems*. Springer, 2011.

[9] P. Boström and J. Wiik. Contract-based verification of discrete-time multi-rate Simulink models. *SoSyM*, 15(4), 2016.

[10] B. Boyer, K. Corre, A. Legay, and S. Sedwards. PLASMA-lab: A flexible, distributable statistical model checking library. In *QEST 2013*. Springer, 2013.

[11] E.M. Clarke, A. Donzé, and A. Legay. On simulation-based probabilistic model checking of mixed-analog circuits. *Form. Meth. Sys. Des.*, 36(2), 2010.

[12] E.M. Clarke and P. Zuliani. Statistical model checking for cyber-physical systems. In *ATVA 2011*, volume 11. Springer, 2011.

[13] J. Deshmukh, X. Jin, J. Kapinski, and O. Maler. Stochastic local search for falsification of hybrid systems. In *ATVA 2015*. Springer, 2015.

[14] A. Donzé. Breach, a toolbox for verification and parameter synthesis of hybrid systems. In *CAV 2010*, volume 6174 of *LNCS*. Springer, 2010.

[15] T. Dreossi, T. Dang, A. Donzé, J. Kapinski, X. Jin, and J.V. Deshmukh. Efficient guiding strategies for testing of temporal properties of hybrid systems. In *NFM 2015*. Springer, 2015.

[16] G.E. Fainekos and G.J. Pappas. Robustness of temporal logic specifications for continuous-time signals. *TCS*, 410(42), 2009.

[17] C. Fan, B. Qi, S. Mitra, and M. Viswanathan. DryVR: Data-driven verification and compositional reasoning for automotive systems. In *CAV 2017*, volume 10426 of *LNCS*. Springer, 2017.

[18] T. Gonschorek, B. Rabeler, F. Ortmeier, and D. Schomburg. On improving rare event simulation for probabilistic safety analysis. In *MEMOCODE 2017*. ACM, 2017.

[19] R. Grosu and S.A. Smolka. Quantitative model checking. In *ISoLA 2004*, 2004.

[20] R. Grosu and S.A. Smolka. Monte Carlo model checking. In *TACAS 2005*, volume 3440 of *LNCS*. Springer, 2005.

[21] B.P. Hayes, I. Melatti, T. Mancini, M. Prodanovic, and E. Tronci. Residential demand management using individualised demand aware price policies. *IEEE Trans. Smart Grid*, 8(3), 2017.

[22] B. Hoxha, A. Dokhanchi, and G. Fainekos. Mining parametric temporal logic properties in model based design for cyber-physical systems. *STTT*, 2017.

[23] C. Jegourel, A. Legay, and S. Sedwards. A platform for high performance statistical model checking–plasma. In *TACAS 2012*, volume 7214 of *LNCS*. Springer, 2012.

[24] C. Jegourel, A. Legay, and S. Sedwards. Importance splitting for statistical model checking rare properties. In *CAV 2013*, volume 8044 of *LNCS*. Springer, 2013.

[25] Y.J. Kim, O. Choi, M. Kim, J. Baik, and T.-H. Kim. Validating software reliability early through statistical model checking. *IEEE Softw.*, 30(3), 2013.

[26] Y.J. Kim and M. Kim. Hybrid statistical model checking technique for reliable safety critical systems. In *ISSRE 2012*. IEEE, 2012.

[27] O. Maler and D. Nickovic. Monitoring temporal properties of continuous signals. In *FORMATS/FTRTFT 2004*, volume 3253 of *LNCS*. Springer, 2004.

[28] T. Mancini, F. Mari, A. Massini, I. Melatti, F. Merli, and E. Tronci. System level formal verification via model checking driven simulation. In *CAV 2013*, volume 8044 of *LNCS*. Springer, 2013.

[29] T. Mancini, F. Mari, A. Massini, I. Melatti, I. Salvo, and E. Tronci. On minimising the maximum expected verification time. *Inf. Proc. Lett.*, 122, 2017.

[30] T. Mancini, F. Mari, A. Massini, I. Melatti, and E. Tronci. Anytime system level verification via random exhaustive hardware in the loop simulation. In *DSD 2014*. IEEE, 2014.

[31] T. Mancini, F. Mari, A. Massini, I. Melatti, and E. Tronci. Simulator semantics for system level formal verification. *EPTCS*, 193, 2015.

[32] T. Mancini, F. Mari, A. Massini, I. Melatti, and E. Tronci. Anytime system level verification via parallel random exhaustive hardware in the loop simulation. *Microprocessors and Microsystems*, 41, 2016.

[33] T. Mancini, F. Mari, A. Massini, I. Melatti, and E. Tronci. SyLVaaS: System level formal verification as a service. *Fundam. Inform.*, 149(1–2), 2016.

[34] T. Mancini, F. Mari, A. Massini, I. Melatti, and E. Tronci. On checking equivalence of simulation scripts. *J. Log. Algebr. Meth. Program.*, 120, 2021.

[35] T. Mancini, F. Mari, I. Melatti, I. Salvo, E. Tronci, J. Gruber, B. Hayes, M. Prodanovic, and L. Elmegaard. Demand-aware price policy synthesis and verification services for smart grids. In *SmartGridComm 2014*. IEEE, 2014.

[36] T. Mancini, F. Mari, I. Melatti, I. Salvo, E. Tronci, J.K. Gruber, B. Hayes, and L. Elmegaard. Parallel statistical model checking for safety verification in smart grids. In *SmartGridComm 2018*. IEEE, 2018.

[37] T. Mancini, F. Mari, I. Melatti, I. Salvo, E. Tronci, J.K. Gruber, B. Hayes, M. Prodanovic, and L. Elmegaard. User flexibility aware price policy synthesis for smart grids. In *DSD 2015*. IEEE, 2015.

[38] T. Mancini, I. Melatti, and E. Tronci. Any-horizon uniform random sampling and enumeration of constrained scenarios for simulation-based formal verification. *IEEE TSE*, 2021.

[39] T. Mancini, E. Tronci, I. Salvo, F. Mari, A. Massini, and I. Melatti. Computing biological model parameters by parallel statistical model checking. In *IWBBIO 2015*, volume 9044 of *LNCS*. Springer, 2015.

[40] F. Mari, I. Melatti, I. Salvo, and E. Tronci. Model based synthesis of control software from system level formal specifications. *ACM TOSEM*, 23(1), 2014.

[41] B. Meenakshi, A. Bhatnagar, and S. Roy. Tool for translating Simulink models into input language of a model checker. In *ICFEM 2006*. Springer, 2006.

[42] N. Miskov-Zivanov, P. Zuliani, E.M. Clarke, and J.R. Faeder. Studies of biological networks with statistical model checking: Application to immune system cells. In *ACM-BCB 2013*. ACM, 2013.

[43] A Rajhans, A. Mavrommati, P.J. Mosterman, and R.G. Valenti. Specification and runtime verification of temporal assessments in simulink. In *RV 2021*. Springer, 2021.

[44] S. Sankaranarayanan, S.A. Kumar, F. Cameron, B.W. Bequette, G. Fainekos, and D.M. Maahs. Model-based falsification of an artificial pancreas control system. *ACM SIGBED Review*, 14(2), 2017.

[45] S. Sinisi, V. Alimguzhin, T. Mancini, and E. Tronci. Reconciling interoperability with efficient verification and validation within open source simulation environments. *Simul. Model. Pract. Theory*, 109, 2021.

[46] S. Sinisi, V. Alimguzhin, T. Mancini, E. Tronci, and B. Leeners. Complete populations of virtual patients for in silico clinical trials. *Bioinformatics*, 36(22–23), 2020.

[47] W.-C. So, C.K. Tse, and Y.-S. Lee. Development of a fuzzy logic controller for DC/DC converters: Design, computer simulation, and experimental evaluation. *IEEE Trans. Pow. Electr.*, 11(1), 1996.

[48] E.D. Sontag. *Mathematical Control Theory: Deterministic Finite Dimensional Systems (2nd Ed.)*. Springer, 1998.

[49] S. Tripakis, C. Sofronis, P. Caspi, and A. Curic. Translating discrete-time Simulink to Lustre. *ACM TECS*, 4(4), 2005.

[50] E. Tronci, T. Mancini, I. Salvo, S. Sinisi, F. Mari, I. Melatti, A. Massini, F. Davi', T. Dierkes, R. Ehrig, S. Röblitz, B. Leeners, T.H.C. Krüger, M. Egli, and F. Ille. Patient-specific models from inter-patient biological models and clinical records. In *FMCAD 2014*. IEEE, 2014.

[51] C.E. Tuncali and G. Fainekos. Rapidly-exploring random trees for testing automated vehicles. In *ITSC 2019*. IEEE, 2019.

[52] M.W. Whalen, D.D. Cofer, S.P. Miller, B.H. Krogh, and W. Storm. Integration of formal analysis into a model-based software development process. In *FMICS 2007*, volume 4916 of *LNCS*. Springer, 2007.

[53] H.L.S. Younes, M.Z. Kwiatkowska, G. Norman, and D. Parker. Numerical vs. statistical probabilistic model checking. *STTT*, 8(3), 2006.

[54] H.L.S. Younes and R.G. Simmons. Probabilistic verification of discrete event systems using acceptance sampling. In *CAV 2002*, volume 2404 of *LNCS*. Springer, 2002.

[55] P. Zuliani, A. Platzer, and E.M. Clarke. Bayesian statistical model checking with application to Stateflow/Simulink verification. *Form. Meth. Sys. Des.*, 43(2), 2013.

Appendix A
Formal framework

A.1 Modelling the System Under Verification

Our SUV is a continuous- or discrete-time Dynamical System whose inputs are operational scenarios defined in terms of time functions (Definition 6) of values in the SUV input space, defining possible values for the user inputs and other *uncontrollable events*, *e.g.*, faults in sensors and actuators or changes in system parameters.

Definition 6 (Time set, time function)**.** *A* time set $\mathbb{T}$ *is* $\mathbb{R}_{0+}$ *(for continuous-time systems) or* $\mathbb{N}$ *(for discrete-time systems) or an interval thereof. Given a time set* $\mathbb{T}$ *and a set of values* $\mathbb{U}$*, a* time function *on* $\mathbb{T}$ *with values in* $\mathbb{U}$ *is a function* $\mathbf{u} : \mathbb{T} \to \mathbb{U}$ *which associates to each time point* $t \in \mathbb{T}$ *a value* $\mathbf{u}(t) \in \mathbb{U}$*.*

Given a time set $\mathbb{T}$ *and a set of values* $\mathbb{U}$*, we denote by* $\mathbb{U}^{\mathbb{T}}$ *the set of time functions on* $\mathbb{T}$ *with values in* $\mathbb{U}$*. When* $\mathbb{T}$ *is the empty interval* $\emptyset$*, we conventionally assume that* $\mathbb{U}^{\emptyset}$ *consists of a single time function* $\mathbf{u}_{\emptyset}$ *(the* empty *time function, having duration zero).*

In the following, we sometimes find convenient to denote the empty time interval $\emptyset$ by $[t, t)$, where t is any value in $\mathbb{T}$. Definition 7 defines two operations on time functions we use in the following.

Definition 7 (Restriction and concatenation of time functions)**.** *Given a time function* $\mathbf{u} \in \mathbb{U}^{\mathbb{T}}$ *and a time set* $\mathbb{T}_1 \subseteq \mathbb{T}$*, the* restriction *of* $\mathbf{u}$ *to* $\mathbb{T}_1$ *is function* $\mathbf{u}_{|\mathbb{T}_1} \in \mathbb{U}^{\mathbb{T}_1}$ *defined in* $\mathbb{T}_1$ *and such that* $\mathbf{u}_{|\mathbb{T}_1}(t) = \mathbf{u}(t)$ *for all* $t \in \mathbb{T}_1$*.*

Given two time sets $\mathbb{T}_1$ *and* $\mathbb{T}_2$ *such* $\mathbb{T}_1 \cap \mathbb{T}_2 = \emptyset$ *and* $\mathbb{T}_1 \cup \mathbb{T}_2$ *is a time set (i.e.,* $\mathbb{R}_{0+}$*,* $\mathbb{N}$*, or an interval thereof), and time functions* $\mathbf{u}_1 \in \mathbb{U}^{\mathbb{T}_1}$ *and* $\mathbf{u}_2 \in \mathbb{U}^{\mathbb{T}_2}$*, the* concatenation *of* $\mathbf{u}_1$ *and* $\mathbf{u}_2$ *is function* $\mathbf{u}_1 \cdot \mathbf{u}_2$ *in* $\mathbb{U}^{\mathbb{T}_1 \cup \mathbb{T}_2}$ *such that, for all* $t \in \mathbb{T}_1 \cup \mathbb{T}_2$*:* $\mathbf{u}_1 \cdot \mathbf{u}_2(t) = \mathbf{u}_1(t)$ *if* $t \in \mathbb{T}_1$ *and* $\mathbf{u}_2(t)$ *otherwise.*

Definition 8 recalls the notion of deterministic, causal Dynamical System from [5] (see also [1], [2]), which take as input time functions with values in its *input space* and output time functions with values in its *output space*.

Definition 8 (Dynamical System)**.** *A* deterministic, causal Dynamical System (DS) *$\mathcal{H}$ is a tuple* $(\mathbb{T}, \mathbb{X}, x_0, \mathbb{U}, \mathbb{Y}, \varphi, \psi)$*, where:*

- $\mathbb{T}$ *is the time set;*
- $\mathbb{X}$*, the* state space *of* $\mathcal{H}$*, is a non-empty set whose elements denote* states*;*
- $x_0 \in \mathbb{X}$ *is the* initial state *of* $\mathcal{H}$*;*
- $\mathbb{U}$*, the* input space *of* $\mathcal{H}$*, is the set of its possible input values;*
- $\mathbb{Y}$*, the* output space *of* $\mathcal{H}$*, is the non-empty set of its possible output values;*
- φ *is the* transition map *of* $\mathcal{H}$*. Given* $t_1, t_2 \in \mathbb{T}$ *such that* $t_1 \le t_2$*,* $x \in \mathbb{X}$*,* $\mathbf{u} \in \mathbb{U}^{\mathbb{T}}$*,* $\varphi(t_2, t_1, x, \mathbf{u})$ *denotes the state reached by* $\mathcal{H}$ *at time* t_2 *when starting from state* x *at time* t_1 *and given input time function* $\mathbf{u}$*.*
  Function φ *must satisfy the following properties:*
  - Causality*: for all* $t_1, t_2, t_3 \in \mathbb{T}$ *such that* $t_1 \le t_2 \le t_3$*,* $\mathbf{u} \in \mathbb{U}^{\mathbb{T}}$*, and* $x \in \mathbb{X}$*,* $\varphi(t_2, t_1, x, \mathbf{u}) = \varphi(t_2, t_1, x, \mathbf{u}_{|[t_1, t_2)})$*.*
  - Consistency*: for all* $t \in \mathbb{T}$*,* $\mathbf{u} \in \mathbb{U}^{\mathbb{T}}$*, and* $x \in \mathbb{X}$*,* $\varphi(t, t, x, \mathbf{u}) = x$*.*
  - Semigroup*: for all* $t_1, t_2, t_3 \in \mathbb{T}$ *such that* $t_1 \le t_2 \le t_3$*,* $\mathbf{u}_{1,2} \in \mathbb{U}^{[t_1, t_2)}$*,* $\mathbf{u}_{2,3} \in \mathbb{U}^{[t_2, t_3)}$*, and* $x \in \mathbb{X}$*,* $\varphi(t_3, t_2, \varphi(t_2, t_1, x, \mathbf{u}_{1,2}), \mathbf{u}_{2,3}) = \varphi(t_3, t_1, x, \mathbf{u}_{1,2} \cdot \mathbf{u}_{2,3})$*.*
- $\psi : \mathbb{X} \to \mathbb{Y}$ *is the* observation function *of* $\mathcal{H}$*.*

A DS *is* time invariant *if its time set* $\mathbb{T}$ *is right-unbounded and, for any* $t_1, t_2, \tau \in \mathbb{T}$ *such that* $t_1 \le t_2$*,* $x \in \mathbb{X}$*, and for any input time function* $\mathbf{u} \in \mathbb{U}^{[t_1, t_2)}$*, we have:* $\varphi(t_1, t_2, x, \mathbf{u}) = \varphi(t_1 + \tau, t_2 + \tau, x, \mathbf{u}')$*, where* $\mathbf{u}' \in \mathbb{U}^{[t_1+\tau, t_2+\tau)}$ *is such that* $\mathbf{u}'(t) = \mathbf{u}(t - \tau)$ *for all* $t \in [t_1 + \tau, t_2 + \tau)$*.*

In the following, we assume that our SUV is a discrete- or continuous-time input-state-output deterministic, causal, time-invariant DS, whose state can undertake continuous as well as discrete changes, and whose output ranges on any combination of discrete and continuous values.

A.2 System-Level Verification

Given a SUV $\mathcal{H}$ and a set of time functions $\mathcal{U}$ on its input space (defining operational scenarios), verifying $\mathcal{H}$ on $\mathcal{U}$ means to collect the outputs of the SUV monitor at the end of each scenario in $\mathcal{U}$ (Definition 9).

Definition 9 (SLV)**.** *A* SLV *problem is a pair* $\pi = (\mathcal{H}, \mathcal{U})$*, where* $\mathcal{H}$ *is a* SUV *(with an embedded monitor) having input space* $\mathbb{U}$*, and* $\mathcal{U}$ *is a set of input time functions in* $\mathbb{U}^{\mathbb{T}}$*.*

The answer to π *is the collection of the outputs of the* SUV *monitor produced at the end of each* $\mathbf{u} \in \mathcal{U}$*, where* $\mathbf{u}$ *is injected in* $\mathcal{H}$ *starting from its initial state. That is:* $\{\psi(\varphi(t, 0, x_0, \mathbf{u})) \mid \mathbf{u} \in \mathcal{U}, \mathbf{u} \in \mathbb{U}^{[0,t)}\}$*.*

A.3 Modelling the SUV operational environment

Given our focus on verification tasks where numerical simulation is the only means to get the trajectory of the SUV when fed with an input scenario, we will assume that the set $\mathcal{U}$ is *finite* and *finitely representable*, and that each scenario is *time-bounded*. Hence, in the following, we assume that the set of values taken by input scenarios in $\mathcal{U}$ (actually, for simplicity, the set $\mathbb{U}$ itself) is *finite* (and, without loss of generality, *ordered*) and scenarios in $\mathcal{U}$ are defined via *piecewise constant* input time functions having discontinuities at time points multiple of a given (arbitrarily small) *time quantum* $\tau \in \mathbb{T} \setminus \{0\}$. Such scenarios can be conveniently represented as *input traces* (Definition 1).

Definition 1 (Input trace). Let $\mathbb{U}$ be a finite set of values (the SUV input space) and $\tau \in \mathbb{T} \setminus \{0\}$ be a *time quantum*.

An input trace $\mathbf{u}$ with values in $\mathbb{U}$ is a finite sequence $(u_0, \ldots, u_{h-1})$ where all, for each $i \in [0, h-1]$, u_i belongs to $\mathbb{U}$. Value $h \in \mathbb{N}_+$ is the trace *horizon*.

An input trace $\mathbf{u} = (u_0, \ldots, u_{h-1})$ is interpreted as the bounded-horizon piecewise constant time function $\mathbf{u} \in \mathbb{U}^{[0, \tau h)}$ defined as $\mathbf{u}(t) = u_{\lfloor \frac{t}{\tau} \rfloor}$ for $t \in [0, \tau h)$. Thus, in the following, we will assume that a time quantum $\tau \in \mathbb{T} \setminus \{0\}$ is given, and interchangeably refer to input traces and to their uniquely associated piecewise constant time functions. We recall (see the main article) that smooth continuous-time functions can be managed as long as they can be cast into (or suitably approximated by) *finitely parametrisable* functions (*e.g.*, via quantised values of a bounded number of coefficients of their Fourier series), in which case the input space actually defines such a (discrete or discretised) parameter space.

Appendix B
SUV simulators

In this appendix, we formalise the notion of SUV simulator (Definition 10, which extends the definition of [2], [3]). This notion offers a general mathematical framework that will allow us to link inputs to a SUV $\mathcal{H}$ (piecewise-constant input time functions representing SUV scenarios and encoded as input traces) to inputs for a simulator of $\mathcal{H}$ (*simulation campaigns*, Definition 12).

Definition 10 (SUV simulator)**.** *Let Λ be a countable set of identifiers (e.g., $\mathbb{N}$). A SUV simulator $\mathcal{S}$ is a tuple $(\mathcal{H}, \mathcal{W})$ where $\mathcal{H} = (\mathbb{T}, \mathbb{X}, x_0, \mathbb{U}, \mathbb{Y}, \varphi, \psi)$ is a time-invariant DS (our SUV), and $\mathcal{W}$ is a set whose elements denote simulator* states. *Each simulator state $w \in \mathcal{W}$ has the form $w = (x, \mathbf{u}, \mathcal{M})$, where:*

- *$x \in \mathbb{X}$ defines a state of $\mathcal{H}$ or a distinguished state $\perp$;*
- *$\mathbf{u} \in \mathbb{U}^{[0,t)}$ denotes an input time function over $\mathbb{U}$ defined over a (possibly empty) bounded time set interval $[0, t)$ (for some $t \in \mathbb{T}$);*
- *$\mathcal{M}$ is a* finite *map* defining the content of the simulator memory. *Each element of $\mathcal{M}$ is of the form: $[\lambda \mapsto (x', \mathbf{u}')]$, where $\lambda \in \Lambda$ is an identifier* (unique *in $\mathcal{M}$), x' a SUV state, and $\mathbf{u}' \in \mathbb{U}^{[0,t')}$ ($t' \in \mathbb{T}$) is an input time function over a bounded (possibly empty) interval of the time set $\mathbb{T}$.*

The simulator initial state *is $w_0 = (x_0, \mathbf{u}_\emptyset, \emptyset)$, where $\mathbf{u}_\emptyset \in \mathbb{U}^{[0,0)}$ is the empty input time function, having zero duration.*

As an extension to [2], [3], each SUV state $x \in \mathbb{X}$ occurring in a simulator state $w = (x, \mathbf{u}, \mathcal{M}) \in \mathcal{W}$ or in the simulator memory $\mathcal{M}$ is always accompanied by the input time function (piecewise constant in our setting) $\mathbf{u}$ which would drive the SUV from its initial state x_0 to x. This is enforced by Definition 11, which gives the semantics of simulator commands and of the simulator transition function, and formally stated in Proposition 1.

Our extension eases the presentation of the forthcoming results. The definitions is [2], [3] can be obtained back by ignoring input time functions in simulator states and in the simulator memory.

Definition 11 (Simulator commands and transition function)**.** *Let $\mathcal{S} = (\mathcal{H}, \mathcal{W})$ be a simulator for SUV $\mathcal{H} = (\mathbb{T}, \mathbb{X}, x_0, \mathbb{U}, \mathbb{Y}, \varphi, \psi)$. The* commands *of $\mathcal{S}$ are:* OUTPUT, LOAD(λ), STORE(λ), FREE(λ), RUN(u, τ), *where λ is an identifier, $u \in \mathbb{U}$ is an input value for $\mathcal{H}$, and $\tau \in \mathbb{T} \setminus \{0\}$ is a non-zero time duration (λ, u, and τ are* command arguments*).*

The transition function of $\mathcal{S}$, $\varphi_{\mathcal{S}}$, defines how the internal simulator state changes upon execution of each command. Namely, $\varphi_{\mathcal{S}}(w, \text{CMD}(args)) = w'$ where $\mathcal{S}$ moves from state w to state w' upon processing command CMD*(args).*

For each $w = (x, \mathbf{u}, \mathcal{M})$, let $\mathbf{u} \in \mathbb{U}^{[0,t)}$ for some $t \in \mathbb{T}$.

Function $\varphi_{\mathcal{S}}$ is defined as follows:

- *$\varphi_{\mathcal{S}}(w, \text{OUTPUT}) = w$, as the* OUTPUT *command only reads the output of $\mathcal{S}$ in the current state, which is the output of SUV $\mathcal{H}$ (Definition 8) when in state x associated to w.*
- *$\varphi_{\mathcal{S}}(w, \text{LOAD}(\lambda)) = (x', \mathbf{u}', \mathcal{M})$ if $x \neq \perp$ and $[\lambda \mapsto (x', \mathbf{u}')] \in \mathcal{M}$.*
- *$\varphi_{\mathcal{S}}(w, \text{STORE}(\lambda)) = (x, \mathbf{u}, \mathcal{M} \cup \{[\lambda \mapsto (x, \mathbf{u})]\})$ if $x \neq \perp$ and $\nexists \mathbf{u}', x'\ [\lambda \mapsto (x', \mathbf{u}')] \in \mathcal{M}$.*
- *$\varphi_{\mathcal{S}}(w, \text{FREE}(\lambda)) = (x, \mathbf{u}, \mathcal{M} \setminus \{[\lambda \mapsto (x, \mathbf{u})]\})$ if $x \neq \perp$ and $[\lambda \mapsto (x, \mathbf{u})] \in \mathcal{M}$.*
- *$\varphi_{\mathcal{S}}(w, \text{RUN}(\hat{u}, \tau)) = (\varphi(\tau, x, \hat{\mathbf{u}}), \mathbf{u} \cdot \hat{\mathbf{u}}, \mathcal{M})$ if $x \neq \perp$; in the formula, $\hat{\mathbf{u}}$ is the input time function in $\mathbb{U}^{[t,t+\tau)}$ having constant value $\hat{u}$ and $\mathbf{u} \cdot \hat{\mathbf{u}} \in \mathbb{U}^{[0,t+\tau)}$ is the concatenation of $\mathbf{u}$ and $\hat{\mathbf{u}}$.*
- *$\varphi_{\mathcal{S}}(w, \text{CMD}(args)) = (\perp, \mathbf{u}, \mathcal{M})$ in all the other cases.*

The time advancement *of command* CMD*(args) is the time simulated by $\mathcal{S}$ when executing* CMD*(args). Namely: time_adv(*CMD*(args)) = τ if* CMD*(args) =* RUN$(\hat{u}, \tau)$ *and is 0 for all the other commands.*

Given a sequence of scenarios (formally represented as piecewise constant input time functions encoded as input traces), we can build a sequence of commands (*simulation campaign*, Definition 12) driving the simulator through those scenarios. Conversely, given a simulation campaign, we can compute the sequence of scenarios (piecewise constant input time functions) simulated by it (Definition 2).

Definition 12 (Simulation campaign, state and output sequences)**.** *Let $\mathcal{S} = (\mathcal{H}, \mathcal{W})$ be a simulator for SUV $\mathcal{H}$ and let $\varphi_{\mathcal{S}}$ be the transition function for $\mathcal{S}$.*

- *A* simulation campaign *χ for $\mathcal{S}$ is a sequence of simulator commands $\chi = \text{CMD}_0(args_0)\ \ldots\ \text{CMD}_{c-1}(args_{c-1})$ along with their arguments, where $c \in \mathbb{N}$.*
- *The* length *$len(\chi)$ of simulation campaign χ is the sum of the time advancements of commands in χ. Namely: $len(\chi) = \sum_{i=0}^{c-1} time\_adv(\text{CMD}_i(args_i))$.*
- *To simulation campaign χ we can univocally associate the* sequence of simulator states *$w_0, \ldots, w_c$ traversed by the simulator while executing it, namely: $w_0 = (x_0, \mathbf{u}_\emptyset, \emptyset)$, i.e., the initial simulator state (Definition 10), and, for each $i \in [1, c]$, $w_i = \varphi_{\mathcal{S}}(w_{i-1}, \text{CMD}_{i-1}(args_{i-1}))$.*
- *Simulation campaign χ is* executable *if and only if $w_c = (x_c, \mathbf{u}_c, \mathcal{M}_c)$ is such that $x_c \neq \perp$.*
- *The* required simulator memory *$mem(\chi)$ of simulation campaign χ is the maximum number of entries in the simulator memory among $w_0, \ldots, w_c$ (i.e., the states traversed by χ, where $w_i = (x_i, \mathbf{u}_i, \mathcal{M}_i)$ for $i \in [0, c]$). Namely: $mem(\chi) = \max_{i=0}^{c} |\mathcal{M}_i|$.*
- *The* output sequence *associated to executable simulation campaign χ containing $n \in \mathbb{N}$* OUTPUT *commands is $\psi(x_{j_0}), \ldots, \psi(x_{j_{n-1}})$, where, for each $i \in [0, n-1]$, $\psi(x_{j_i})$ is the output of SUV $\mathcal{H}$ (Definition 8) when in the state x_{j_i} associated to the simulator state $(x_{j_i}, \mathbf{u}_{j_i}, \mathcal{M}_{j_i})$ corresponding to the i-th* OUTPUT *command.*

Proposition 1 links inputs to a simulator $\mathcal{S}$ for $\mathcal{H}$ (*i.e.*, simulation campaigns) to inputs for $\mathcal{H}$ (input time functions): for each simulation campaign χ, the (piecewise constant) input time function $\mathbf{u}$ of any simulator state $(x, \mathbf{u}, \mathcal{M})$ traversed by $\mathcal{S}$ while executing χ drives $\mathcal{H}$ from its initial state x_0 to x.

Proposition 1. Let $\mathcal{S} = (\mathcal{H}, \mathcal{W})$ be a simulator for $\mathcal{H}$, $\chi = \text{CMD}_0(args_0)\ \ldots\ \text{CMD}_{c-1}(args_{c-1})$ an executable simulation campaign for $\mathcal{S}$, and $w_0, \ldots, w_c$ the sequence of simulator states associated to χ.

For each $i \in [0, c]$, the input time function $\mathbf{u}_i$ in $w_i = (x_i, \mathbf{u}_i, \mathcal{M}_i)$ belongs to $\mathbb{U}^{[0,\tau_i)}$ (for some $\tau_i \in \mathbb{T}$) and is such that $\varphi(\tau_i, 0, x_0, \mathbf{u}_i) = x_i$.

Proof. Let $\mathcal{H} = (\mathbb{T}, \mathbb{X}, x_0, \mathbb{U}, \mathbb{Y}, \varphi, \psi)$. For each $i \in [0, c]$, let the i-th state traversed by $\mathcal{S}$ during execution of χ be $w_i = (x_i, \mathbf{u}_i, \mathcal{M}_i)$.

We prove that, for each i:

1) $\mathbf{u}_i \in \mathbb{U}^{[0,\tau_i)}$ for some $\tau_i \in \mathbb{R}_{0+}$; and
2) $\varphi(\tau_i, 0, x_0, \mathbf{u}_i) = x_i$.

The proof is by induction on i.

Base case ($i = 0$) By Definition 10, $w_0 = (x_0, \mathbf{u}_\emptyset, \emptyset)$, where $\mathbf{u}_\emptyset \in \mathbb{U}^{[0,\tau_0)}$ with $\tau_0 = 0$, *i.e.*, $\mathbf{u}_\emptyset$ is the input time function having zero duration. By Definition 8 (consistency), $\varphi_\mathcal{S}(\tau_0, \tau_0, x_0, \mathbf{u}_\emptyset) = x_0$. The thesis follows.

Inductive case ($i \in [1, c]$) Assume, by inductive hypothesis, that the $(i-1)$-th state traversed by the simulator when executing χ, $w_{i-1} = (x_{i-1}, \mathbf{u}_{i-1}, \mathcal{M}_{i-1})$ is such that $\mathbf{u}_{i-1} \in \mathbb{U}^{[0,\tau_{i-1})}$ for some $\tau_{i-1} \in \mathbb{R}_{0+}$ and $\varphi_\mathcal{S}(\tau_{i-1}, 0, x_0, \mathbf{u}_{i-1}) = x_{i-1}$. We now prove that $\mathbf{u}_i \in \mathbb{U}^{[0,\tau_i)}$ for some $\tau_i \in \mathbb{R}_{0+}$ and that $\varphi_\mathcal{S}(\tau_i, 0, x_0, \mathbf{u}_i) = x_i$. The proof is by cases, depending on the type of command $\textsc{cmd}_i(\mathit{args}_i)$ of χ, which moves $\mathcal{S}$ from state w_{i-1} to state $w_i = (x_i, \mathbf{u}_i, \mathcal{M}_i)$ (see the definition of $\varphi_\mathcal{S}$ in Definition 11).

- If $\textsc{cmd}_i(\mathit{args}_i) = \textsc{store}(\lambda)$, $\textsc{free}(\lambda)$ for some $\lambda \in \Lambda$ or $\textsc{output}$, then the thesis trivially follows, as, from executability of χ and the definition of $\varphi_\mathcal{S}$, $x_i = x_{i-1}$ and $\mathbf{u}_i = \mathbf{u}_{i-1}$.
- If $\textsc{cmd}_i(\mathit{args}_i) = \textsc{load}(\lambda)$ for some $\lambda \in \Lambda$, then executability of χ and the definition of $\varphi_\mathcal{S}$ imply that tuple $[\lambda \mapsto (x_b, \mathbf{u}_b)]$ exists in $\mathcal{M}_{i-1}$, with $x_b = x_i$ and $\mathbf{u}_b = \mathbf{u}_i$. The thesis follows from the inductive hyphotesis.
- If $\textsc{cmd}_i(\mathit{args}_i) = \textsc{run}(u, \tau)$ for some $u \in \mathbb{U}$ and $\tau \in \mathbb{R}_+$, the definition of $\varphi_\mathcal{S}$ implies that $\mathbf{u}_i = \mathbf{u}_{i-1} \cdot \hat{\mathbf{u}}$, *i.e.*, the concatenation of the input time function associated to simulator state w_{i-1}, *i.e.*, $\mathbf{u}_{i-1} \in \mathbb{U}^{[0,\tau_{i-1})}$, and the input time function $\hat{\mathbf{u}} \in \mathbb{U}^{[\tau_{i-1},\tau_{i-1}+\tau)}$ having constant value $\hat{u}$ and defined in time set $[\tau_{i-1}, \tau_{i-1} + \tau)$. Thus, $\mathbf{u}_i \in \mathbb{U}^{[0,\tau_{i-1}+\tau)}$ and from Definition 8 (semigroup), the second point of the thesis follows from the inductive hypothesis.

□

Appendix C Simulation-based SLV

To perform *simulation-based* SLV of $\mathcal{H}$ over n input traces $\mathcal{U}$ we need a simulator $\mathcal{S} = (\mathcal{H}, \mathcal{W})$ for $\mathcal{H}$ and an executable simulation campaign χ for $\mathcal{S}$ that somewhat drives $\mathcal{S}$ along the n scenarios for $\mathcal{H}$ encoded by traces of $\mathcal{U}$ and collects the simulator outputs at the end of each scenario.

To this end, Definition 2 allows us to associate to any executable simulation campaign χ for $\mathcal{S}$ the sequence $\mathcal{U}(\chi)$ of SUV scenarios (as piecewise constant input time functions) for $\mathcal{H}$ actually explored by χ.

Definition 2 (Sequence of input time functions associated to a simulation campaign). Let $\mathcal{S} = (\mathcal{H}, \mathcal{W})$ be a simulator for SUV $\mathcal{H}$, $\varphi_\mathcal{S}$ the transition function of $\mathcal{S}$, $\chi = \textsc{cmd}_0(\mathit{args}_0) \ldots \textsc{cmd}_{c-1}(\mathit{args}_{c-1})$ an executable simulation campaign for $\mathcal{S}$ containing $n \in \mathbb{N}$ $\textsc{output}$ commands, and $w_0, \ldots, w_c$ the sequence of simulator states associated to χ.

The *sequence of input time functions* associated to simulation campaign χ containing n $\textsc{output}$ commands is $\mathcal{U}(\chi) = \mathbf{u}_{j_0}, \ldots, \mathbf{u}_{j_{n-1}}$, where, for all $i \in [0, n-1]$, $\mathbf{u}_{j_i}$ is the input time function associated to the state where the simulator executes the i-th $\textsc{output}$ command of χ.

Definition 3 formalises the notion of a *simulation campaign aimed at computing the answer to a SLV problem*.

Definition 3 (simulation campaign for an SLV problem). A simulation campaign χ *for* SLV problem $\pi = (\mathcal{H}, \mathcal{U})$ is an executable campaign for a simulator $\mathcal{S}$ of $\mathcal{H}$, such that the sequence $\mathcal{U}(\chi) = \mathbf{u}_{j_0}, \ldots, \mathbf{u}_{j_{n-1}}$ of its associated input time functions is a *permutation* of $\mathcal{U}$.

C.1 Randomised simulation campaigns

Proposition 2 states that, if we put no limitation on the required memory capacity, a shortest simulation campaign exists for *any* ordering of the scenarios of the SLV problem at hand.

Proposition 2. Let $\pi = (\mathcal{H}, \mathcal{U})$ be a SLV problem ($|\mathcal{U}| = n$) and $\mathcal{S}$ be a simulator for $\mathcal{H}$. For any permutation $\mathbf{u}_{j_0}, \ldots, \mathbf{u}_{j_{n-1}}$ of input traces of $\mathcal{U}$, there exists an executable shortest simulation campaign χ for π on $\mathcal{S}$, such that $\mathcal{U}(\chi) = \mathbf{u}_{j_0}, \ldots, \mathbf{u}_{j_{n-1}}$.

Proof. Proof sketch. Arrange input traces of $\mathcal{U}$ (with time quantum $\tau \in \mathbb{T} \setminus \{0\}$) as a rooted tree whose nodes are trace *prefixes* (including the empty prefix, which is the root of the tree) and whose edges connect nodes $(u_0, \ldots, u_d)$ ($d \geq 0$) with their parents $(u_0, \ldots, u_{d-1})$ and are labelled with u_d (when $d = 0$, $(u_0, \ldots, u_{d-1})$ is conventionally assumed to be the empty prefix). Every leaf node (or, equivalently, the sequence of edge labels along the unique path from root to it) is uniquely associated to a complete trace of $\mathcal{U}$.

Clearly, a shortest simulation campaign for $\mathcal{U}$ must be long at least τl, where $\tau \in \mathbb{T}$ is the time quantum associated to traces in $\mathcal{U}$ and l is the number of edges of the tree.

A simulation campaign long exactly τl can be easily generated for any ordering $\mathbf{u}_{j_0}, \ldots, \mathbf{u}_{j_{n-1}}$ of $\mathcal{U}$, by considering paths of the tree connecting the root to the leaves (corresponding to the traces) in the required order. The campaign is very simple, given that for this proof we can rely on unlimited simulator memory.

Namely, we want to traverse each edge of the tree from its parent to its child node exactly once, with the aim to reach all the leaves (representing all the traces in $\mathcal{U}$) in the required order. When traversing edge from $(u_0, \ldots, u_{d-1})$ (parent, $d \geq 0$) to $(u_0, \ldots, u_d)$ (child node), we issue commands: $\textsc{store}(\lambda)$ (for a fresh identifier λ), $\textsc{run}(u_d, \tau)$. When reaching a leaf node, we issue command $\textsc{output}$. If the trace just considered is not the last one in the given order, we start the new trace by issuing command $\textsc{load}(\lambda)$, where λ is the *deepest* state of the tree already saved by a previous $\textsc{store}$ command along the root-to-leaf path identifying the new trace, and continue from there.

□

C.2 Parallel simulation campaigns

The answer to a SLV problem $\pi = (\mathcal{H}, \mathcal{U})$ (*i.e.*, the collection of the simulator outputs at the end of each scenario) can be computed by arbitrarily partitioning $\mathcal{U}$ into $k \in \mathbb{N}_+$ subsets (*slices*) $\mathcal{U}_0, \ldots, \mathcal{U}_{k-1}$ (where k is the number of available computational nodes), and by computing and taking the union of the answers to the k smaller SLV problems $\pi_i = (\mathcal{H}, \mathcal{U}_i)$, $i \in [0, k-1]$. In our simulation-based setting, this can be achieved using k simulators for $\mathcal{H}$ running as k *independent* processes (*e.g.*, in parallel in a HPC infrastructure) and *independently* driven by k simulation campaigns $\chi_1, \ldots, \chi_k$, where, for all i, χ_i is a simulation campaign for π_i. Definition 5 formalises this concept.

Definition 5 (Parallel simulation campaign for a SLV problem). A *k-parallel simulation campaign* for SLV problem $\pi = (\mathcal{H}, \mathcal{U})$ is a tuple $\Xi = (\chi_0, \ldots, \chi_{k-1})$ such that there exists a partition of $\mathcal{U}$ into sets $\mathcal{U}_0, \ldots, \mathcal{U}_{k-1}$ such that, for all i, χ_i is a simulation campaign for $\pi_i = (\mathcal{H}, \mathcal{U}_i)$.

The *length* of χ is $len(\chi) = \max_{i=0}^{k-1} len(\chi_i)$. Given $m \in \mathbb{N}_+ \cup \{\infty\}$, Ξ is a *k-parallel m-memory simulation campaign* if all χ_is are m-memory simulation campaigns.

Appendix D
Parallel computation of parallel simulation campaigns

D.1 Computing a simulation campaign from each slice

D.1.1 Computing the Longest Shared Prefix Tree

In the following, given two input traces $\mathbf{u}_a$ and $\mathbf{u}_b$, we denote by $\mathbf{u}_a \sqsubseteq \mathbf{u}_b$ (respectively $\mathbf{u}_a \sqsubset \mathbf{u}_b$) the fact that $\mathbf{u}_a$ (possibly the empty sequence) is a *prefix* (respectively, a *proper prefix*) of $\mathbf{u}_b$. Also, by exploiting the fact that set $\mathbb{U}$ is ordered, we denote by $\mathbf{u}_a \prec \mathbf{u}_b$ the fact that $\mathbf{u}_a$ is *lexicographically less* than $\mathbf{u}_b$.

Definition 13 defines the notions of Longest Shared Prefix and that of Longest Shared Prefix Tree.

Definition 13 (Longest Shared Prefix, LSP; Longest Shared Prefix Tree, LSPT)**.** *Let $\mathcal{U}_i$ be a finite collection of input traces (e.g., a slice of $\mathcal{U}$) with values in $\mathbb{U}$.*

A Longest Shared Prefix (LSP) for $\mathcal{U}_i$ is a (possibly empty) sequence $\mathbf{u}$ of inputs (i.e., sequences of values of $\mathbb{U}$) such that there exist two traces $\mathbf{u}_a$ and $\mathbf{u}_b$ in $\mathcal{U}_i$ such that: $\mathbf{u} \sqsubseteq \mathbf{u}_a$, $\mathbf{u} \sqsubseteq \mathbf{u}_b$, and there exists no $\mathbf{u}'$ in $\mathcal{U}_i$ such that $\mathbf{u} \sqsubset \mathbf{u}'$, $\mathbf{u}' \sqsubseteq \mathbf{u}_a$, and $\mathbf{u}' \sqsubseteq \mathbf{u}_b$.

A Longest Shared Prefix Tree (LSPT) for $\mathcal{U}_i$ is a tree $\mathcal{T} = (V, parent)$ such that:

1. *Nodes (set V) denote* distinct *LSPs of $\mathcal{U}_i$.*
2. *The parent node of $\mathbf{u}$ (if one exists) is $parent(\mathbf{u}) = \mathbf{u}_p \in V$ such that $\mathbf{u}_p \sqsubset \mathbf{u}$ and there exists no $\mathbf{u}' \in V$ such that $\mathbf{u}_p \sqsubset \mathbf{u}' \sqsubset \mathbf{u}$.*

Condition 2. implies that a LSPT is a rooted tree.

The following functions are defined over nodes of $\mathcal{T}$ (set V):

a) *Function depth : $V \to \mathbb{N}$ associates to each node $\mathbf{u} = (u_0, \ldots, u_{d-1})$ of $\mathcal{T}$ its length d, which represents the time point $d\tau$ reached by the simulator (starting from its initial state) after having injected input sequence $\mathbf{u}$. The depth of the node associated to the empty sequence is zero.*
b) *ntraces : $V \to \mathbb{N}_+$, which associates to each node $\mathbf{u}$ the number of traces in $\mathcal{U}_i$ having $\mathbf{u}$ as a (proper or non-proper) prefix, i.e., ntraces($\mathbf{u}$) $= |\{\mathbf{u}' \in \mathcal{U}_i \mid \mathbf{u} \sqsubseteq \mathbf{u}'\}|$.*

A LSPT $\mathcal{T} = (V, parent)$ for $\mathcal{U}_i$ is complete *if no LSPT $\mathcal{T}' = (V', parent')$ exists for $\mathcal{U}_i$ such that $V \subset V'$.*

The size *of LSPT $\mathcal{T} = (V, parent)$ is size($\mathcal{T}$) $= |V|$, i.e., the number of its nodes.*

The goal of function *LSPT()* is to build a complete LSPT for $\mathcal{U}_i$ in central memory. To this end, the algorithm scans $\mathcal{U}_i$ in lexicographic order, since, under this ordering, deciding which trace prefixes are nodes of the tree is straightforward and memory-efficient.

To keep an as small as possible RAM footprint of the LSPT, the algorithm represents in central memory each of its nodes $(u_0, \ldots, u_{d-1})$ by a *unique identifier* $\lambda(u_0, \ldots, u_{d-1})$. Unique identifiers for each trace prefix are available for free when traces are extracted from a scenario generator. In case traces are taken from an input database, any efficiently computable *injective* function of finite sequences of input values (or even a cryptographic hash function, when the probability of conflicts is small enough) can be used.

Pseudocode of function *LSPT()* is reported in Algorithm 3.

```
1  function LSPT($\mathcal{U}_i$)
2      input $\mathcal{U}_i$, slice of traces output $\mathcal{T} = (V, parent)$,
        Longest Shared Prefix Tree of simulator states
3      $\mathcal{T} \leftarrow$ empty tree;
4      $\mathbf{u}_{\mathrm{prv}} \leftarrow$ empty (will keep last trace);
5      foreach $\mathbf{u} \in \mathcal{U}_i$ in lex order do
6          if $\mathbf{u}$ is not the first trace in $\mathcal{U}_i$ then
7              lsp $\leftarrow$ longest (possibly empty) prefix
                shared by $\mathbf{u}$ and $\mathbf{u}_{\mathrm{prv}}$;
8              par $\leftarrow$ longest node in $V$ s.t. par $\sqsubseteq$ lsp
                (possibly none) ;
9              if lsp $\notin V$ then
10                 add lsp to $V$;
11                 $parent(lsp) \leftarrow par$ ;
12                 child $\leftarrow$ shorter prefix of $\mathbf{u}_{\mathrm{prv}}$ s.t.
                    lsp $\sqsubset$ child $\in V$ and parent(child) = par
                    (par can be none; at most one such
                    node exists) ;
13                 if child exists then
14                     $parent(child) \leftarrow lsp$;
15                     ntraces(lsp) $\leftarrow$ ntraces(child);
16                 else ntraces(lsp) $\leftarrow 1$;
17             prefix $\leftarrow$ lsp ;
18             while prefix is not none do
19                 ntraces(prefix)++ ;
20                 prefix $\leftarrow parent(prefix)$;
21         $\mathbf{u}_{\mathrm{prv}} \leftarrow \mathbf{u}$ ;
22     return $\mathcal{T}$;
```

Algorithm 3: Function *LSPT()*.

Algorithm 3 aims at creating a new node of the LSPT for any simulator state associated to a prefix of the current trace $\mathbf{u}$ that satisfies condition 1. of Definition 13.

To recognise the trace prefixes to add as nodes, the function again exploits the fact that traces in $\mathcal{U}_i$ can be accessed in lexicographic order. This implies that (unique identifiers of) prefixes of $\mathbf{u}$ that can be added as nodes of the LSPT must belong to the last-processed trace $\mathbf{u}_{\text{prv}}$. Also, at most one prefix of each $\mathbf{u}$ can be added as a node of the LSPT.

Thus, Algorithm 3 proceeds as follows.

1. It selects the longest (possibly empty) prefix *lsp* shared by $\mathbf{u}$ and $\mathbf{u}_{\text{prv}}$ (line 7) and the longest prefix $par \in V$ such that $par \sqsubseteq lsp$ (possibly none, line 8).

2. It infers that the current trace $\mathbf{u}$ and the previously processed trace $\mathbf{u}_{\text{prv}}$ are identical up to *lsp* and differ at the next time point.

3. If $lsp \in V$, then *lsp* is already a node of the LSPT. Otherwise, *lsp* satisfies condition 1. of Definition 13 and is added as a new node of the LSPT. In particular:

3.1. Node *lsp* is added to the LSPT as a *child* node of *par* (which can possibly be the empty prefix or none; in the latter case, *lsp* becomes the root of the LSPT).

3.2. As node *lsp* could already have children in the LSPT, the tree may need to be rearranged to accommodate the new label *lsp*. Tree rearrangement is again very efficient thanks to the lexicographic order of the input traces. In fact, *par* can have at most one child that needs to be moved and needs to become a child of *lsp*.

This child, if exists, must be the shortest prefix *child* of the previous trace $\mathbf{u}_{\text{prv}}$ already in the LSPT, and such that $lsp \sqsubset child$ and $parent(child) = par$ (where *par* can be none, line 12).

If such a child node exists, then it is moved as to become a child of *lsp* (line 13) and value for *ntraces(lsp)* is temporarily set to *ntraces(child)*.

Otherwise (node *child* does not exist), value for *ntraces(lsp)* is temporarily set to 1 (to account that it occurs in $\mathbf{u}_{\text{prv}}$).

4. In both cases, the values of *ntraces (prefix)* for each LSPT node $prefix \sqsubseteq lsp$ (including both *lsp* and the empty prefix, if the latter is in the LSPT) are incremented by 1, in order to take into account their occurrence in trace $\mathbf{u}$ (lines 17–20).

Lemma 1 (Function *LSPT()*)**.** *Data structure $\mathcal{T}$ computed by function LSPT() (Algorithm 3) is a complete LSPT of $\mathcal{U}_i$ according to Definition 13.*

Proof. Let $\mathcal{U}_i = \mathbf{u}_0, \ldots, \mathbf{u}_{n-1}$. We prove the lemma by induction, by showing that, for each $j \in [1, n]$, the tree $\mathcal{T}$ computed by function *LSPT()* after having processed the set $\mathcal{U}_i^j = \mathbf{u}_0, \ldots, \mathbf{u}_{j-1}$ of the first j traces of $\mathcal{U}_i$ in lexicographic order is a complete LSPT for $\mathcal{U}_i^j$.

Base case When $j = 1$, $\mathcal{U}_i^j$ consists of a single disturbance trace $\mathbf{u}_0 = (u_0, \ldots, u_{h-1})$. In this case, function *LSPT()* would just store $\mathbf{u}_0$ as $\mathbf{u}_{\text{prv}}$ and $\mathcal{T}$ would be the empty tree, which is a complete LSPT when the set of traces is a singleton. The thesis trivially follows.

Inductive case Assume that, after having processed the first $j - 1 \geq 1$ traces of $\mathcal{U}_i$ (*i.e.*, set $\mathcal{U}_i^{j-1}$), data structure $\mathcal{T}$ computed by Algorithm 3 is a complete LSPT for $\mathcal{U}_i^{j-1}$ according to Definition 13. We now show that, after processing trace $\mathbf{u}_{j-1}$, the revised $\mathcal{T}$ is a complete LSPT for $\mathcal{U}_i^j$.

As $\mathbf{u}_{j-1}$ is not the first processed trace, we have that: $\mathbf{u}_{\text{prv}} = \mathbf{u}_{j-2}$ (as set in the previous iteration, see line 21 of Algorithm 3), *lsp* is set (line 7) to the longest (possibly empty) prefix shared by $\mathbf{u}_{j-1}$, and $\mathbf{u}_{\text{prv}} = \mathbf{u}_{j-2}$, and *par* is set (line 8) to the longest prefix of *lsp* (hence, occurring in both $\mathbf{u}_{j-1}$ and $\mathbf{u}_{j-2}$) denoting a node already in $\mathcal{T}$, and to 'none' if none exists.

Clearly, *lsp* is a LSP of $\mathcal{U}_i^j$ and, since traces in $\mathcal{U}_i$ are lexicographically ordered, no additional LSPs of $\mathcal{U}_i^j$ can exist.

If *lsp* is already a node in $\mathcal{T}$, the algorithm ignores it and increments by 1 the value of *ntraces* for all the $\mathcal{T}$ nodes occurring in the current trace $\mathbf{u}_{j-1}$ (lines 17–20), thus making such values satisfy again Definition 13.

On the other hand, in case $lsp \notin V$, the algorithm adds it to the tree (line 11) making the tree complete with respect to $\mathcal{U}_i^j$. The new node is added to $\mathcal{T}$ as a child of *par*, which, by construction, is either the longest *proper* prefix of *lsp* already in V or 'none' (if no such prefix exists, in which case, the newly added *lsp* becomes the new root of the tree), and thus satisfies condition 2. of Definition 13.

However, the introduction of a new node in $\mathcal{T}$ (*lsp*) could make condition 2. false for some of the pre-existing children nodes of *par* in $\mathcal{T}$.

For a child node *child* of *par* to violate condition 2. in the current iteration, it must be that $par \sqsubset \mathbf{u}_x \sqsubset child$ for some $\mathbf{u}_x \in V$. Since the only newly added node is *lsp*, it must be $\mathbf{u}_x = lsp$. Also, given that the input traces are in lexicographic order, at most one such child node exists. Algorithm 3, by reassigning the parent of *child* (if it exists) to *lsp*, makes condition 2. of Definition 13 true again.

The only thing that remains to show is that the value of *ntraces* for all nodes of $\mathcal{T}$ is correctly revised. We show this in two steps:

1. The value of *ntraces* for nodes of the tree as computed before line 17, is correct if we consider only traces processed up to iteration $j - 1$.

2. The current trace $\mathbf{u}_{j-1}$ is correctly taken into account in the revision of *ntraces* values, in lines 17–20.

As for step 1., only the value of *ntraces* for the newly inserted node *lsp* must be set (in case such node is added to $\mathcal{T}$). Node *lsp* might have zero or one children (as seen above).

If *lsp* has a child node (*child*), then the set of traces within $\mathcal{U}_i^{j-1}$ (hence, excluding the current trace $\mathbf{u}_{j-1}$) having prefix *lsp* are exactly those having (the longer) prefix *child* (line 15).

Otherwise (*lsp* has no children), trace $\mathbf{u}_{j-1}$ shares *lsp* as a prefix only with (the previous) trace $\mathbf{u}_{j-2}$ among those seen so far. To see why, assume, for the sake of contradiction, that two traces $\mathbf{u}_a$ and $\mathbf{u}_b$ exist in $\mathcal{U}_i^{j-1}$ which both have *lsp* as a prefix. Since $\mathcal{U}_i^{j-1}$ is lex-ordered, there must exists $\mathbf{u}_p$ such that $lsp \sqsubseteq \mathbf{u}_p \sqsubseteq \mathbf{u}_a$ and $lsp \sqsubseteq \mathbf{u}_p \sqsubseteq \mathbf{u}_b$, and such that $\mathbf{u}_a$ and $\mathbf{u}_b$ differ immediately after $\mathbf{u}_p$. But this would mean that $\mathbf{u}_p$ would have been recognised as a LSP and added as a node of $\mathcal{T}$ in a previous iteration of the algorithm, when the last trace between $\mathbf{u}_a$ and $\mathbf{u}_b$ was processed (contradiction). This proves the correctness of setting *ntraces(lsp)* to 1 in line 16 (when the current trace $\mathbf{u}_{j-1}$ has not yet been considered in this computation).

As for step 2., lines 17–20 increment by 1 the value of *ntraces* for all the nodes of $\mathcal{T}$ occurring as prefixes of the currently processed trace $\mathbf{u}_{j-1}$ (including *lsp*), thus taking into correct account the existence of the $\mathbf{u}_{j-1}$.

As a result of the above, $\mathcal{T}$ is a complete LSPT for $\mathcal{U}_i^j$ and the thesis follows. □

D.2 Algorithm correctness

Proposition 3 (Correctness of Algorithm 1). Let $\pi = (\mathcal{H}, \mathcal{U})$ be a SLV problem for SUV $\mathcal{H} = (\mathbb{T}, \mathbb{X}, x_0, \mathbb{U}, \mathbb{Y}, \varphi, \psi)$, with $\mathcal{U}$ being defined as input traces associated to time quantum $\tau \in \mathbb{T} \setminus \{0\}$, and let $m \in \mathbb{N}_+$.

Given any partition $\{\mathcal{U}_0, \ldots, \mathcal{U}_{k-1}\}$ ($k \in \mathbb{N}_+$) of $\mathcal{U}$, let $\Xi = (\chi_0, \ldots, \chi_{k-1})$ be the k-parallel simulation campaign such that χ_i ($i \in [0, k-1]$) is computed by Algorithm 1 on inputs $\mathcal{U}_i$ (under any user-defined order), τ, and m.

We have that:

1. For all $i \in [0, k-1]$, the sequence $\mathcal{U}(\chi_i)$ is $\mathcal{U}_i$;
2. There exists $m^* \in \mathbb{N}_+$ such that, if $m \geq m^*$, all χ_is ($i \in [0, k-1]$) are *shortest* m-memory simulation campaigns.

Proof. We first make the following observations:

a) At any time during execution of Algorithm 1, when sequence of simulator commands χ (a prefix of the overall simulation campaign) has been computed, the set of simulator states λ such that *stored(λ)* $=$ *true* are *exactly* those that would be available in the simulator memory after the actual execution of χ. In particular, *stored(λ)* is set to *true* (respectively *false*) immediately after appending to χ command STORE(λ) (respectively FREE(λ)).

b) Each of the computed simulation campaigns is *executable*. This is immediate, as LOAD(λ)/FREE(λ) (respectively, STORE(λ)) commands are issued only for simulator state identifiers λ available (respectively, not available) in the simulator memory, as requested by Definition 11. Also, when $j > 0$, function *sim_cmds()* always finds state λ_{load} to load in line 2. This is because, if not, then the j-th trace would have no prefix in common with any of the previous traces, not even the *empty* prefix, which is impossible.

The simulation campaign χ generated by Algorithm 1 has the form:

$$\chi_i \;=\; \chi_{i,0} \;\cdots\; \chi_{i,n-1}$$

where $n = |\mathcal{U}_i|$ and, for each $j \in [0, n-1]$, $\chi_{i,j}$ is the sequence of simulator commands generated by function *sim_cmds()* (Algorithm 2) on $\mathbf{u}_j$, the j-th trace of $\mathcal{U}_i$.

From Algorithm 2, we know that each $\chi_{i,j}$ starts with a LOAD command if $j > 0$ (omitted for $j = 0$), and then continues with a sequence of RUN commands (possibly interleaved with STORE and/or FREE commands). Each RUN command covers a distinct constant portion of $\mathbf{u}_j$, and, together, they cover the entire postfix of $\mathbf{u}_j$ after the prefix loaded with LOAD, if any. Finally, $\chi_{i,j}$ terminates with an OUTPUT command.

Executability of χ_i guarantees that all LOAD, STORE and FREE commands succeed, and Definition 11 and Proposition 1 together ensure that the input time function associated to the *final* state reached by the simulator when executing each $\chi_{i,j}$ (for all j) is exactly $\mathbf{u}_j$. Thus, point 1. immediately follows from Definition 2.

As for the proof of point 2., if m is at least the number of nodes of the LSPT, then each χ_i is a shortest campaign. This can be shown along the lines of the proof of Proposition 2, observing that, by Lemma 1, $\mathcal{T}$ is a a complete LSPT for $\mathcal{U}_i$ (Lemma 1).

However, we also observe that the number of nodes of $\mathcal{T}$ is most often a *very loose* upper bound for m^*. In most cases, the simulation memory required to produce shortest campaigns is *much* smaller than the number of nodes of the LSPT. This is because function *sim_cmds()* (Algorithm 2) greedily frees simulator memory (by injecting FREE commands) *has soon as* it discover that a state will not be needed to shorten the simulation of future traces (*i.e.*, as soon as its associated counter *ntraces* becomes zero). The actual value for m^* of course depends on the (possibly random) simulation order chosen by the user. □

Appendix E
Case studies

Our scenario generators for our three case studies below are inspired from those in [4].

In order to focus the SLV activity on clearly selected portions of the space of inputs and to keep the overall number of traces under control, our scenario generators enforce various constraints on the entailed scenarios.

We chose such constraints from (slight variations of) those in [4] with the final aim to have a number of entailed traces of around 50, 100, and 200 million for BDC, ALMA, and FCS, respectively. Multiple simulation campaigns, then, have been computed (see Section 6) for various random portions of such sets of traces (from 25% to 100%), different random seeds, and different optimisation settings.

E.1 Buck DC-DC Converter (BDC)

The BDC SUV model takes two inputs: the input voltage V_i and the load R, which vary during time.

Our scenario generator enforces the following constraints on the time course of its inputs:

1) Both V_i and R may vary during time up to at most $\pm 30\%$ of their nominal values, in steps of $\pm 5\%$ and $\pm 10\%$ of their initial values.
2) Values for V_i and R are stable for 5 and 6 t.u., respectively.
3) To have a proper set-up, V_i and R are assumed stable to their nominal values for the first 2 t.u.
4) V_i and R do not change simultaneously.
5) Whenever V_i changes, R will change after 7 t.u.

By enforcing the constraints above and a time horizon of 60 t.u., our scenario generator entails 49 971 109 input traces, each one defining a piecewise constant function (time quantum $\tau = 1$ t.u.) over an input space of 25 different values.

E.2 Apollo Lunar Model Autopilot (ALMA)

The ALMA SUV model takes as inputs attitude change requests along each of the three axes ("Yaw", "Pitch", "Roll") as well as events signalling temporary failures of actuators (reaction jets).

Our scenario generator enforces the following constraints on the time course of its inputs:

1) Attitude requests arrive at each t.u., and each request may ask for a unitary positive or negative change of the attitude along *at most* one axis.
2) Attitude requests do not ask the autopilot to immediately undo the rotation requested along any axis in the preceding t.u.
3) Two consecutive requests for attitude changes along the same axis are 10 or 11 t.u. apart.
4) Only two given reaction jets (number 14 and 15) can be subject to temporary unavailability, which are always recovered within 6 to 7 t.u.. Jet unavailability events occur every 12 to 13 t.u.
5) To obey each received attitude request, the autopilot decides which reaction jets must be used at each t.u.. We further constrain attitude requests so that no jet is used consecutively for more than 3 t.u.. Also, when a jet (among number 14 and 15) is used for 2 t.u. in a row, it will certainly become unavailable within 3 to 4 t.u.

The following additional constraints were enforced to further limit the focus of the verification activity and the overall number of traces.

6) Jet number 14 always becomes unavailable (respectively, becomes available) immediately after the reception of a request for a negative (respectively, positive) change in the Yaw attitude.
7) Jet number 15 always becomes unavailable (respectively, becomes available) immediately after the reception of a request for a negative (respectively, positive) change in the Roll or Pitch attitude.
8) Whenever a positive (respectively, negative) attitude Roll request is received, the current attitudes along Yaw and Pitch are at least (respectively, less then) a given threshold value.
9) Whenever a positive (respectively, negative or null) attitude Pitch request is received, the current attitudes along Yaw and Roll are at least (respectively, less then) a given threshold value.

By enforcing the constraints above and a time horizon of 100 t.u., our scenario generator entails 107 535 209 input traces, each one defining a piecewise constant function (time quantum $\tau = 1$ t.u.) over an input space of 432 different values.

E.3 Fault Tolerant Fuel Control System (FCS)

The FCS SUV model takes as inputs failure events on its four sensors ("throttle", "speed", "ECO", "MAP").

Our scenario generator enforces the following constraints on the time course of its inputs:

1) At time zero, all sensors are functioning properly.
2) Each faulty sensor recovers within the following time bounds (in t.u.): 3–5 (throttle), 5–7 (speed), 10–15 (EGO), 13–17 (MAP).
3) At most one sensor is faulty at any given time.
4) A sensor fault occurs every 15–20 t.u.
5) Whenever a fault on the throttle sensor occurs, a fault on the speed sensor will occur within 18 or 19 t.u.
6) Whenever a fault on the EGO sensor occurs, a fault on the MAP sensor will occur within 20 or 21 t.u.

By enforcing the constraints above and a time horizon of 100 t.u., our scenario generator entails 195 869 671 input traces, each one defining a piecewise constant function (time quantum $\tau = 1$ t.u.) over an input space of 6 different values.

References

[1] T. Mancini, F. Mari, A. Massini, I. Melatti, I. Salvo, and E. Tronci. On minimising the maximum expected verification time. *Inf. Proc. Lett.*, 122, 2017.

[2] T. Mancini, F. Mari, A. Massini, I. Melatti, and E. Tronci. Simulator semantics for system level formal verification. *EPTCS*, 193, 2015.

[3] T. Mancini, F. Mari, A. Massini, I. Melatti, and E. Tronci. On checking equivalence of simulation scripts. *J. Log. Algebr. Meth. Program.*, 120, 2021.

[4] T. Mancini, I. Melatti, and E. Tronci. Any-horizon uniform random sampling and enumeration of constrained scenarios for simulation-based formal verification. *IEEE TSE*, 2021.

[5] E.D. Sontag. *Mathematical Control Theory: Deterministic Finite Dimensional Systems (2nd Ed.)*. Springer, 1998.