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

Toni Mancini, Igor Melatti, and Enrico Tronci

***Abstract*—Cyber-Physical Systems (CPSs), comprising both software and physical components, arise in many industry-relevant domains and are 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 and/or of additional, uncontrolled disturbances. 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 databases or from symbolic structures), 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, thus minimising completion time. Our experiments on SLV of Modelica/FMU and Simulink models with up to almost *200 million scenarios* show that our optimisation yields *speedups as high as* 8×. This, together with the enabled *massive parallelisation*, makes practically viable (a few weeks in a HPC infrastructure) verification tasks (both statistical and exhaustive) which would otherwise take *inconceivably* long time.**

***Index Terms*—System-level verification, simulation, cyber-physical systems, systems engineering**

I. 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 micro-programmed) 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]).

A. Background and Motivations

Unfortunately, models of industry-relevant CPSs are often defined as systems of highly non-linear and possibly stiff differential equations, and their complexity hinders the possibility of

Manuscript received 1 June 2022; revised 7 February 2023; accepted 16 July 2023. Date of publication 26 July 2023; date of current version 19 September 2023. This work was supported in part by NRRP Mission 4, Comp. 2, Inv. 1.5, NextGenEU, MUR CUP B83C22002820006, Rome Technopole. Recommended for acceptance by C. Wang. *(Corresponding author: Toni Mancini.)*

Toni Mancini and Enrico Tronci are with the Computer Science Department, Sapienza University of Rome, 00198 Rome, Italy (e-mails: tmancini@di.uniroma1.it; tronci@di.uniroma1.it).

Igor Melatti is with the Department of Information Engineering, Computer Science and Mathematics, University of L'Aquila, 67100 L'Aquila, Italy (e-mail: igor.melatti@univaq.it).

This article has supplementary downloadable material available at https://doi.org/10.1109/TSE.2023.3298432, provided by the authors.

Digital Object Identifier 10.1109/TSE.2023.3298432

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.

B. 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).

II. 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 (see the supplementary material).

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 VI contain uses of several of such features, and show that our setting can be easily met in practice.

III. 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 (available online).

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* $\mathcal{X}$ for $\mathcal{S}$ is a sequence $\mathrm{CMD}_0(args_0) \ldots \mathrm{CMD}_{c-1}(args_{c-1})$ of simulator commands (with their arguments). To $\mathcal{X}$ we can univocally associate: (i) the *sequence of states* traversed by the simulator while executing it; (ii) the *length* $len(\mathcal{X})$, which is the sum of the time advancements of its commands; (iii) the *required simulator memory* $mem(\mathcal{X})$, 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 IV.

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

IV. 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 $\mathcal{X}$ 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 $\mathcal{X}$ for $\mathcal{S}$ the sequence $\mathcal{U}(\mathcal{X})$ of SUV scenarios (as piecewise constant input time functions) for $\mathcal{H}$ actually explored by $\mathcal{X}$. Full definitions and statements in this section as well as their proofs are delayed to Appendix C (available online).

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

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

A simulation campaign $\mathcal{X}$ for $\pi = (\mathcal{H}, \mathcal{U})$ can be used to compute the answer to π by executing $\mathcal{X}$ on a simulator $\mathcal{S}$ for $\mathcal{H}$ and by collecting the simulator outputs during $\mathcal{X}$. 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 $\mathcal{X}$ (at the end of each simulated scenario) can be used to build such statistics.

A. 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 $\mathcal{X}$ for π is a simulation campaign for π such that $mem(\mathcal{X}) \leq m$ and for which no other simulation campaign $\mathcal{X}'$ for π exists such that $len(\mathcal{X}') < len(\mathcal{X})$ and $mem(\mathcal{X}') \leq m$. When $m = \infty$ (i.e., we do not put any limitation on the required simulator memory capacity to execute $\mathcal{X}$), we call $\mathcal{X}$ 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 Definition 13 in Appendix D.1, available online). 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 VI).

B. 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 an 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 $\mathcal{X}$ for π on $\mathcal{S}$, such that $\mathcal{U}(\mathcal{X}) = \mathbf{u}_{j_0}, \ldots, \mathbf{u}_{j_{n-1}}$.

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 $\mathcal{X}$. 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 V.

C. Parallel Simulation Campaigns

As anticipated in Section I-A, 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*) $\mathcal{U}_0, \ldots, \mathcal{U}_{k-1}$ (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 $\mathcal{X}_1, \ldots, \mathcal{X}_k$, where, for all i, $\mathcal{X}_i$ 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 $\Xi = (\mathcal{X}_0, \ldots, \mathcal{X}_{k-1})$ such that there exists a partition of $\mathcal{U}$ into sets $\mathcal{U}_0, \ldots, \mathcal{U}_{k-1}$ such that, for all i, $\mathcal{X}_i$ is a simulation campaign for $\pi_i = (\mathcal{H}, \mathcal{U}_i)$.

The length of $\mathcal{X}$ is $len(\mathcal{X}) = \max_{i=0}^{k-1} len(\mathcal{X}_i)$. Given $m \in \mathbb{N}_+ \cup \{\infty\}$, Ξ is a k-parallel m-memory simulation campaign if all $\mathcal{X}_i$s 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 $\mathcal{X}_i$s of a parallel simulation campaign $\Xi = (\mathcal{X}_0, \ldots, \mathcal{X}_{k-1})$ are *randomised* (*i.e.*, each $\mathcal{X}_i$ 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 $\mathcal{X}_i$ 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 $1 - \min_{i=0}^{k-1} \left(\frac{j_i}{n_i}\right)$.

V. Parallel Computation of Parallel Simulation Campaigns

We are now ready to present our algorithm to compute a parallel simulation campaign $\Xi = (\mathcal{X}_0, \ldots, \mathcal{X}_{k-1})$ 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 (available online).

A. 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 $\mathcal{U} = \{trace(\mathcal{G}, j) \mid j \in \mathcal{I}\}$.

B. 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* $\mathcal{X}_0, \ldots, \mathcal{X}_{k-1}$ (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 VI-D, 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.

Algorithm 1: Simulation campaign computation for a slice

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*
output $\mathcal{X}_i$, *the output simulation campaign*

$\mathcal{X}_i \leftarrow$ an empty sequence of commands;
$\mathcal{T} \leftarrow$ *LSPT*($\mathcal{U}_i$); /* *build Longest Shared Prefix Tree* */
$j \leftarrow 0$; /* *trace counter* */
foreach $\mathbf{u} \in \mathcal{U}_i$ *(in the given order)* **do**
    append *sim_cmds*($\mathbf{u}$, j, $\mathcal{T}$) to $\mathcal{X}_i$; $j++$;
return $\mathcal{X}_i$;

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 presorted in mass-memory databases, see, *e.g.*, [32]).

C. Computing a Simulation Campaign From Each Slice

From this point on, computation of the parallel simulation campaign $\Xi = (\mathcal{X}_0, \ldots, \mathcal{X}_{k-1})$ 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) 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 *LSPT* for $\mathcal{U}_i$ (see Appendix D.1.1, available online, 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.

2) Generation of Simulation Campaign Commands: Algorithm 1 proceeds at generating an optimised simulation campaign $\mathcal{X}_i$ 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 $\mathcal{X}_i$ 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 $\mathcal{X}_i$, the fact that state λ would be available or not in the memory of $\mathcal{S}_i$ at that point during the execution of $\mathcal{X}_i$. 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 $\mathcal{X}_i$. 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 $\mathcal{X}_i$, 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 $\mathcal{X}_i$ (*i.e.*, *stored*(λ_q) = *true*), it appends to $\mathcal{X}_i$ command FREE(λ_q) to free-up the simulator memory.
3. Appends to $\mathcal{X}_i$ 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).

Algorithm 2: Function *sim_cmds()*

```
1  function sim_cmds(u, j, T)
2      input u = (u_0, ..., u_{h-1}), current (j-th) trace
3      input T = (V, parent), Longest Shared Prefix Tree
4      output sequence of sim. commands for u
5      if j = 0 then load ← 0;        /* first trace */
6      else                           /* not first trace */
7       load ← max q ∈ [0, h] s.t.
8            λ_load = λ(u_0, ..., u_{q-1}) ∈ V ∧ stored(λ_load);
9       issue LOAD(λ_load);
10     for q from h − 1 downto 0 s.t.
        λ_q = λ(u_0, ..., u_{q-1}) ∈ V do /* revise T */
11        ntraces(λ_q)--;
12        if ntraces(λ_q) = 0 then
              /* λ_q won't occur in future
                traces                          */
13            if stored(λ_q) then
14                issue FREE(λ_q); stored(λ_q) ← false;
15            remove λ_q from V; /* λ_q is leaf
              in T                              */
      /* All nodes still in T will occur
       in future traces                         */
16    start ← load;
17    while start < h do
18        end ← max e ∈ [start, h−1] s.t. ∀q ∈ [start+1, e]
19              u_q = u_{q-1} ∧¬worth_storing(λ(u_0, ..., u_{q-1}), T);
20        issue RUN(u_start, (end − start + 1)τ);
21        start ← end + 1;
22        if start ≤ h ∧ λ_start = λ(u_0, ..., u_{start-1}) ∈ V ∧
           worth_storing(λ_start, T) then
23            do_store(λ_start, T, m, X_i); /* possibly
               issues FREE(λ') for some λ' s.t.
               stored(λ') and sets stored(λ')
               to false, before issuing
               STORE(λ_start)                   */
24            stored(λ_start)←true;
25    issue OUTPUT;
```

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()*.

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(λ')* − *depth(parent(λ'))* > 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 $\text{STORE}(\lambda)$ to $\mathcal{X}_i$, preceded by $\text{FREE}(\lambda')$ 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, available online, 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 $\Xi = (\mathcal{X}_0, \ldots, \mathcal{X}_{k-1})$ be the k-parallel simulation campaign such that $\mathcal{X}_i$ 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 $\mathcal{U}(\mathcal{X}_i)$ is $\mathcal{U}_i$;
2. There exists $m^* \in \mathbb{N}_+$ such that, if $m \geq m^*$, all $\mathcal{X}_i$s are shortest m-memory simulation campaigns.

Point 1. implies that Ξ is a k-parallel m-memory simulation campaign for π. Each $\mathcal{X}_i$ 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 IV-C).

VI. Implementation and Experimental Results

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

A. 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.

B. Case Studies

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

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].

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.

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.

TABLE I
SCENARIO GENERATORS FOR OUR CASE STUDIES

SUV	$\|\mathbb{U}\|$	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

C. 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 I. 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 (available online). 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 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.

D. Experimental Results

Our experimental results are summarised in Fig. 1 (BDC), Fig. 2 (ALMA) and Fig. 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 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.

1) Scalability of the Campaign Computation Algorithm: The first (left-most) column of Figs. 1–3 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).

2) Campaigns Efficiency With Respect to Parallelisation: The second column of Figs. 1–3 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: $\frac{\textit{sim\_time}(\mathcal{X}_{n,1024,m}) \times 1024}{\textit{sim\_time}(\mathcal{X}_{n,k,m}) \times k}$, which measures, in terms of (estimated) simulation time (*sim_time*), the efficiency of the parallel simulation campaign $\mathcal{X}_{n,k,m}$ (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 $\mathcal{X}_{n,1024,m}$ (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.

3) Campaigns Efficiency With Respect to Available Simulator Memory: The third column of Figs. 1–3 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: $\frac{\textit{sim\_time}(\mathcal{X}_{n,k,m^*})}{\textit{sim\_time}(\mathcal{X}_{n,k,m})}$, which measures, in terms of simulation time (*sim_time*), the efficiency of the parallel simulation campaign $\mathcal{X}_{n,k,m}$ (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 $\mathcal{X}_{n,k,m^*}$ (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

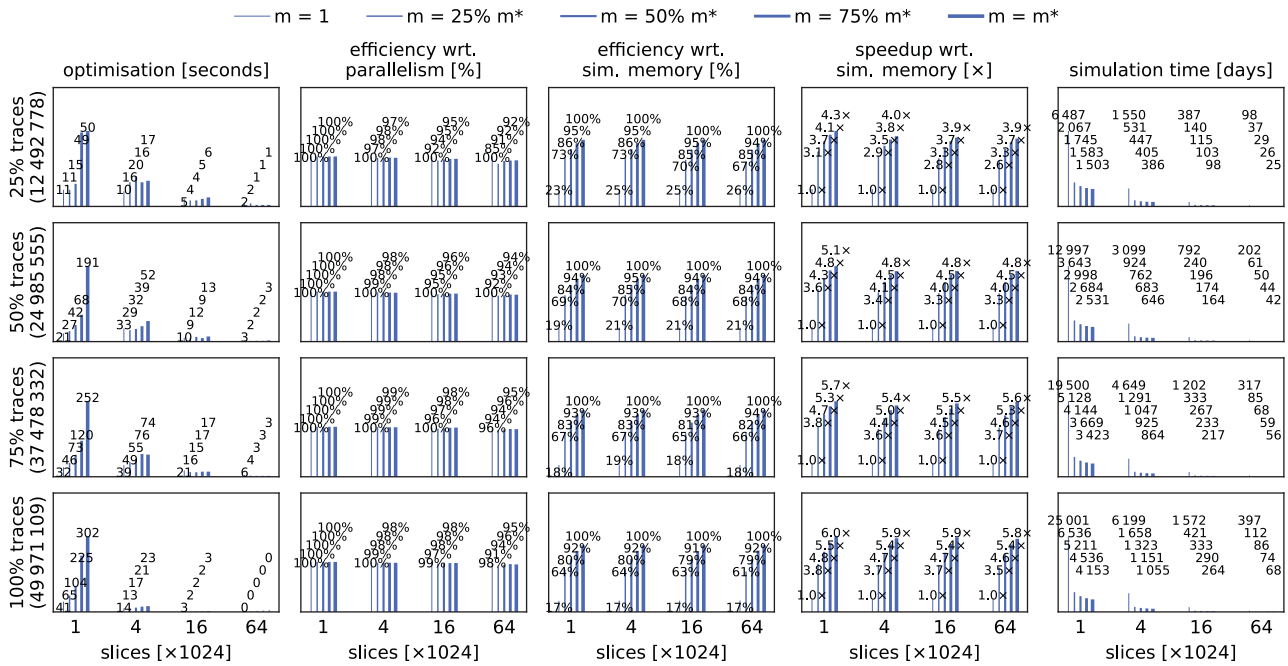

Fig. 1. Experimental results: Buck DC–DC Converter (BDC).

(*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 ($m^* \leq 156\,115$).

4) Simulation Speedups and Time Savings: The fourth column of Figs. 1–3 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: $\frac{sim\_time(\mathcal{X}_{n,k,1})}{sim\_time(\mathcal{X}_{n,k,m})}$, which measures, in terms of simulation time (*sim_time*), the speedup of each parallel simulation campaign $\mathcal{X}_{n,k,m}$ (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 $\mathcal{X}_{n,k,1}$ (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 Figs. 1–3 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 $\mathcal{X}_{n,k,m}$, 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 $m = 1$).

E. 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 needed 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

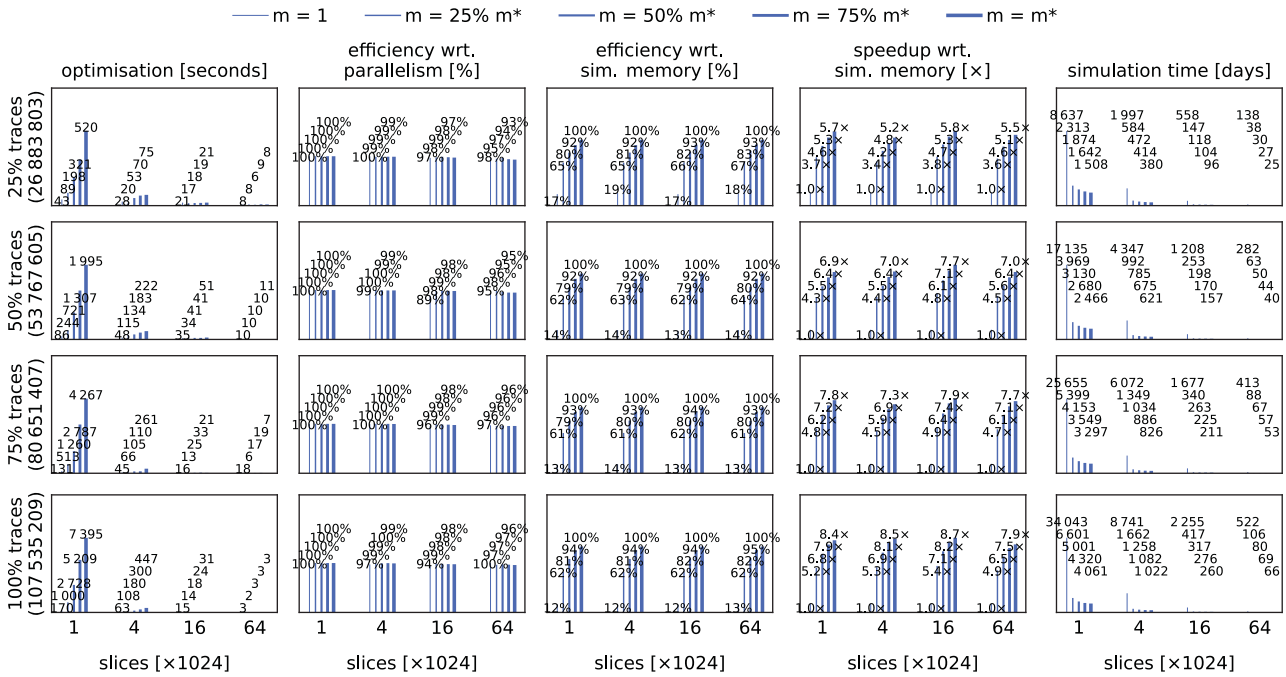

Fig. 2. Experimental results: Apollo Lunar Model Autopilot (ALMA).

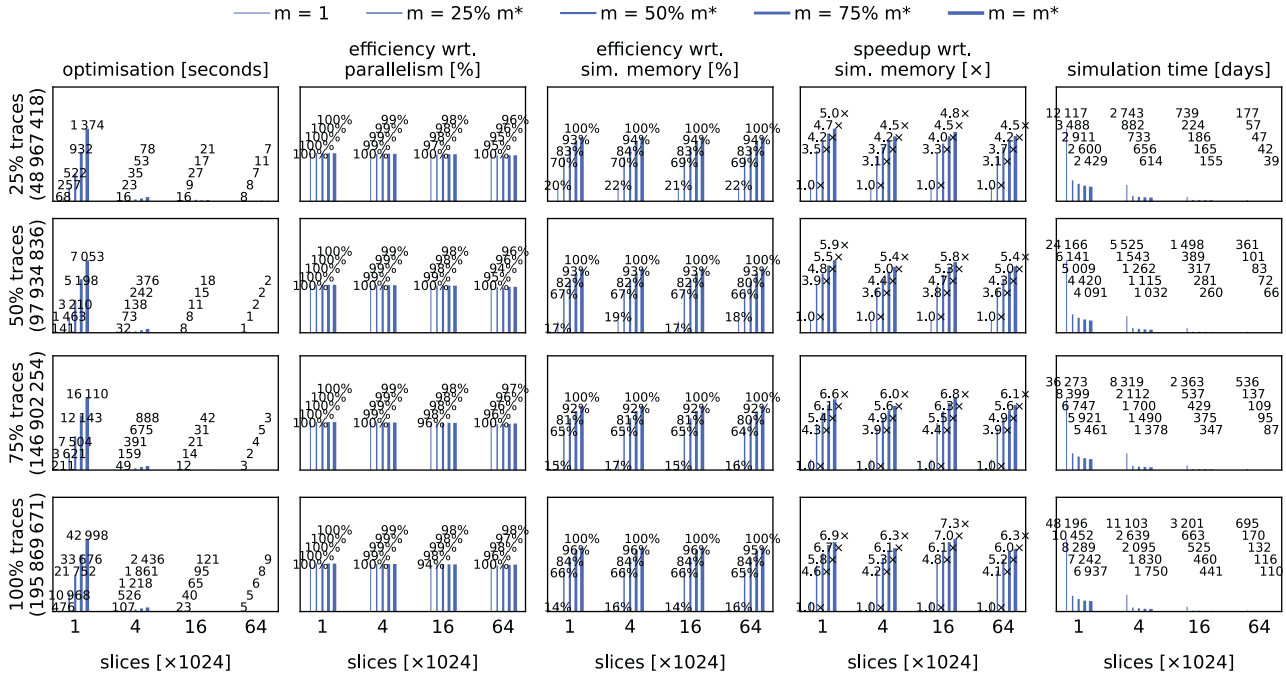

Fig. 3. Experimental results: Fuel Control System (FCS).

to: 2.2–2.4× (BDC), 4.3–6.0× (ALMA), 3.7–5.5× (FCS) for $f = 10$; 1.0–1.3× (BDC), 2.5–3.2× (ALMA), 2.5–3.1× (FCS) for $f = 50$; 0.9–1.0× (BDC), 1.6–2.4× (ALMA), 1.7–2.3× (FCS) for $f = 100$.

VII. 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], [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 of simulation time to all of them. For example, the first row of Figs. 1–3 shows that speedups up to around $6\times$ 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.

VIII. 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* $8\times$ and scales very well to large HPC infrastructures (efficiency almost always $\geq 90\%$ 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.

Acknowledgment

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

References

[1] H. Abbas, G. Fainekos, S. Sankaranarayanan, F. Ivančić, and A. Gupta, "Probabilistic temporal logic falsification of cyber-physical systems," *ACM TECS*, vol. 12, no. 2s, pp. 95:1–95:30, 2013.

[2] A. Adimoolam, T. Dang, A. Donzé, J. Kapinski, and X. Jin, "Classification and coverage-based falsification for embedded control systems," in *Proc. CAV* in Lecture Notes in Computer Science, vol. 10426. Berlin, Germany: Springer, 2017, pp. 483–503.

[3] G. Agha and K. Palmskog, "A survey of statistical model checking," *ACM Trans. Model. Comput. Simul.*, vol. 28, no. 1, pp. 6:1–6:39, 2018.

[4] M. AlTurki and J. Meseguer, "PVESTA: A parallel statistical model checking and quantitative analysis tool," in *Proc. CALCO* in Lecture Notes in Computer Science, vol. 6859. Berlin, Germany: Springer, 2011, pp. 386–392.

[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 *Proc. TACAS* in Lecture Notes in Computer Science, vol. 6605. Berlin, Germany: Springer, 2011.

[6] S. Bak and P. S. Duggirala, "Simulation-equivalent reachability of large linear systems with inputs," in *Proc. CAV* in Lecture Notes in Computer Science, vol. 10426. Berlin, Germany: 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 *Proc. Formal Techn. Distrib. Syst.*, Berlin, Germany: Springer, 2010, pp. 32–46.

[8] J. Bogdoll, L. M. F. Fioriti, A. Hartmanns, and H. Hermanns, "Partial order methods for statistical model checking and simulation," in *Proc. Formal Techn. Distrib. Syst.*, Berlin, Germany: Springer, 2011, pp. 59–74.

[9] P. Boström and J. Wiik, "Contract-based verification of discrete-time multi-rate Simulink models," *Softw. Syst. Model.*, vol. 15, no. 4, pp. 1141–1161, 2016.

[10] B. Boyer, K. Corre, A. Legay, and S. Sedwards, "PLASMA-lab: A flexible, distributable statistical model checking library," in *Proc. QEST*, Berlin, Germany: Springer, 2013, pp. 160–164.
[11] E. M. Clarke, A. Donzé, and A. Legay, "On simulation-based probabilistic model checking of mixed-analog circuits," *Form. Methods Syst. Des.*, vol. 36, no. 2, pp. 97–113, 2010.
[12] E. M. Clarke and P. Zuliani, "Statistical model checking for cyber-physical systems," in *Proc. ATVA*, vol. 11. Berlin, Germany: Springer, 2011, pp. 1–12.
[13] J. Deshmukh, X. Jin, J. Kapinski, and O. Maler, "Stochastic local search for falsification of hybrid systems," in *Proc. ATVA*, Berlin, Germany: Springer, 2015, pp. 500–517.
[14] A. Donzé, "Breach, a toolbox for verification and parameter synthesis of hybrid systems," in *Proc. CAV* in Lecture Notes in Computer Science, vol. 6174. Berlin, Germany: Springer, 2010, pp. 167–170.
[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 *Proc. NFM*, Berlin, Germany: Springer, 2015, pp. 127–142.
[16] G. E. Fainekos and G. J. Pappas, "Robustness of temporal logic specifications for continuous-time signals," *Theor. Comput. Sci.*, vol. 410, no. 42, pp. 4262–4291, 2009.
[17] C. Fan, B. Qi, S. Mitra, and M. Viswanathan, "DRYVR: Data-driven verification and compositional reasoning for automotive systems," in *Proc. CAV* in Lecture Notes in Computer Science, vol. 10426. Berlin, Germany: Springer, 2017, pp. 441–461.
[18] T. Gonschorek, B. Rabeler, F. Ortmeier, and D. Schomburg, "On improving rare event simulation for probabilistic safety analysis," in *Proc. MEMOCODE*, New York, NY, USA: ACM, 2017, pp. 15–24.
[19] R. Grosu and S. A. Smolka, "Quantitative model checking," in *Proc. ISoLA*, 2004, pp. 165–174.
[20] R. Grosu and S. A. Smolka, "Monte Carlo model checking," in *Proc. TACAS* in Lecture Notes in Computer Science, vol. 3440. Berlin, Germany: Springer, 2005, pp. 271–286.
[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*, vol. 8, no. 3, pp. 1284–1294, May 2017.
[22] B. Hoxha, A. Dokhanchi, and G. Fainekos, "Mining parametric temporal logic properties in model based design for cyber-physical systems," *Int. J. Software Tools Technol. Trans.*, vol. 20, pp. 79–93, 2017.
[23] C. Jegourel, A. Legay, and S. Sedwards, "A platform for high performance statistical model checking–plasma," in *Proc. TACAS* in Lecture Notes in Computer Science, vol. 7214. Berlin, Germany: Springer, 2012, pp. 498–503.
[24] C. Jegourel, A. Legay, and S. Sedwards, "Importance splitting for statistical model checking rare properties," in *Proc. CAV* in Lecture Notes in Computer Science, vol. 8044. Berlin, Germany: Springer, 2013, pp. 576–591.
[25] Y. J. Kim, O. Choi, M. Kim, J. Baik, and T.-H. Kim, "Validating software reliability early through statistical model checking," *IEEE Softw.*, vol. 30, no. 3, pp. 35–41, May/Jun. 2013.
[26] Y. J. Kim and M. Kim, "Hybrid statistical model checking technique for reliable safety critical systems," in *Proc. IEEE 23rd Int. Symp. Soft. Rel. Eng.*, Piscataway, NJ, USA: IEEE, 2012, pp. 51–60.
[27] O. Maler and D. Nickovic, "Monitoring temporal properties of continuous signals," in *Proc. FORMATS/FTRTFT* in Lecture Notes in Computer Science, vol. 3253. Berlin, Germany: Springer, 2004, pp. 152–166.
[28] T. Mancini, F. Mari, A. Massini, I. Melatti, F. Merli, and E. Tronci, "System level formal verification via model checking driven simulation," in *Proc. CAV* in Lecture Notes in Computer Science, vol. 8044. Berlin, Germany: Springer, 2013, pp. 296–312.
[29] T. Mancini, F. Mari, A. Massini, I. Melatti, I. Salvo, and E. Tronci, "On minimising the maximum expected verification time," *Inf. Proc. Lett.*, vol. 122, pp. 8–16, 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 *Proc. 17th Euromicro Conf. Digit. Syst. Des. (DSD)*. Piscataway, NJ, USA: IEEE, 2014, pp. 236–245.
[31] T. Mancini, F. Mari, A. Massini, I. Melatti, and E. Tronci, "Simulator semantics for system level formal verification," *Electron. Proc. Theor. Comput. Sci.*, vol. 193, pp. 86–99, 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," *Microprocess. Microsyst.*, vol. 41, pp. 12–28, 2016.
[33] T. Mancini, F. Mari, A. Massini, I. Melatti, and E. Tronci, "SyLVaaS: System level formal verification as a service," *Fundam. Inf.*, vol. 149, no. 1–2, pp. 101–132, 2016.
[34] T. Mancini, F. Mari, A. Massini, I. Melatti, and E. Tronci, "On checking equivalence of simulation scripts," *J. Log. Algebr. Methods Program.*, vol. 120, 2021, Art. no. 100640.
[35] T. Mancini et al., "Demand-aware price policy synthesis and verification services for smart grids," in *Proc. IEEE Int. Conf. Smart Grid Commun. (SmartGridComm)*, Piscataway, NJ, USA: IEEE, 2014, pp. 794–799.
[36] T. Mancini et al., "Parallel statistical model checking for safety verification in smart grids," in *Proc. IEEE Int. Conf. Smart Grid Commun. (SmartGridComm)*, Piscataway, NJ, USA: IEEE, 2018, pp. 1–6.
[37] T. Mancini et al., "User flexibility aware price policy synthesis for smart grids," in *Proc. Euromicro Conf. Digit. Syst. Des.*, Piscataway, NJ, USA: IEEE, 2015, pp. 478–485.
[38] T. Mancini, I. Melatti, and E. Tronci, "Any-horizon uniform random sampling and enumeration of constrained scenarios for simulation-based formal verification," *IEEE Trans. Softw. Eng.*, vol. 48, no. 10, pp. 4002–4013, Oct. 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 *Proc. IWBBIO* in Lecture Notes in Computer Science, vol. 9044. Berlin, Germany: Springer, 2015, pp. 542–554.
[40] F. Mari, I. Melatti, I. Salvo, and E. Tronci, "Model based synthesis of control software from system level formal specifications," *ACM Trans. Softw. Eng. Method.*, vol. 23, no. 1, pp. 1–42, 2014.
[41] B. Meenakshi, A. Bhatnagar, and S. Roy, "Tool for translating Simulink models into input language of a model checker," in *Proc. ICFEM*, Berlin, Germany: Springer, 2006, pp. 606–620.
[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 *Proc. ACM-BCB*, New York, NY, USA: ACM, 2013, p. 728.
[43] A. Rajhans, A. Mavrommati, P. J. Mosterman, and R. G. Valenti, "Specification and runtime verification of temporal assessments in simulink," in *Proc. RV*, Berlin, Germany: Springer, 2021, pp. 288–296.
[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 Rev.*, vol. 14, no. 2, pp. 24–33, 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*, vol. 109, 2021, Art. no. 102277.
[46] S. Sinisi, V. Alimguzhin, T. Mancini, E. Tronci, and B. Leeners, "Complete populations of virtual patients for in silico clinical trials," *Bioinformatics*, vol. 36, no. 22–23, pp. 5465–5472, 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. Power Electron.*, vol. 11, no. 1, pp. 24–32, Jan. 1996.
[48] E. D. Sontag, *Mathematical Control Theory: Deterministic Finite Dimensional Systems,* 2nd ed., Berlin, Germany: Springer, 1998.
[49] S. Tripakis, C. Sofronis, P. Caspi, and A. Curic, "Translating discrete-time Simulink to Lustre," *ACM Trans. Embedded Comput. Syst.*, vol. 4, no. 4, pp. 779–818, 2005.
[50] E. Tronci et al., "Patient-specific models from inter-patient biological models and clinical records," in *Proc. Formal Methods Comput.-Aided Des. (FMCAD)*, Piscataway, NJ, USA: IEEE, 2014, pp. 207–214.
[51] C. E. Tuncali and G. Fainekos, "Rapidly-exploring random trees for testing automated vehicles," in *Proc. IEEE Intell. Transp. Syst. Conf. (ITSC)*, Piscataway, NJ, USA: IEEE, 2019, pp. 661–666.
[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 *Proc. FMICS* in Lecture Notes in Computer Science, vol. 4916. Berlin, Germany: Springer, 2007, pp. 68–84.
[53] H. L. S. Younes, M. Z. Kwiatkowska, G. Norman, and D. Parker, "Numerical vs. statistical probabilistic model checking," *Int. J. Softw. Tools Technol. Transf.*, vol. 8, no. 3, pp. 216–228, 2006.
[54] H. L. S. Younes and R. G. Simmons, "Probabilistic verification of discrete event systems using acceptance sampling," in *Proc. CAV* in Lecture Notes in Computer Science, vol. 2404. Berlin, Germany: Springer, 2002.
[55] P. Zuliani, A. Platzer, and E. M. Clarke, "Bayesian statistical model checking with application to Stateflow/Simulink verification," *Form. Methods Syst. Des.*, vol. 43, no. 2, pp. 338–367, 2013.

Open Access funding provided by 'Università degli Studi di Roma "La Sapienza" 2' within the CRUI CARE Agreement