The Journal of Systems and Software 219 (2025) 112238

Contents lists available at ScienceDirect

The Journal of Systems & Software

journal homepage: www.elsevier.com/locate/jss

# Scaling up statistical model checking of cyber-physical systems via algorithm ensemble and parallel simulations over HPC infrastructures ☆

Leonardo Picchiami [a], Maxime Parmentier [b], Axel Legay [b], Toni Mancini [a,*], Enrico Tronci [a]

[a] Computer Science Department, Sapienza University of Rome, Via Salaria 113, Rome, 00198, Italy
[b] Computer Science Deparment, Universite Catholique de Louvain, Place Sainte Barbe 2, Louvain-la-Neuve, 1348, Brabant Wallon, Belgium

ARTICLE INFO

Dataset link: https://doi.org/10.5281/zenodo.13969757

*Keywords:*
Formal verification
Cyber-Physical Systems
Statistical Model Checking
Adaptive Stopping Algorithms
Monte Carlo estimation
Simulation
High-Performance Computing

ABSTRACT

Model-based formal verification of industry-relevant Cyber-Physical Systems (CPSs) is often a computationally prohibitive task. In most cases, the complexity of the models precludes any prospect of symbolic analysis, leaving numerical simulation as the only viable option. Unfortunately, exhaustive simulation of a CPS model over the entire set of plausible operational scenarios is rarely possible in practice, and alternative strategies such as Statistical Model Checking (SMC) must be used instead.

In this article, we show that the number of model simulations (samples) required by SMC techniques to converge can be significantly reduced by considering multiple (an *ensemble* of) Adaptive Stopping Algorithms (SAs) at once, and that the simulations themselves (by far the most expensive step of the entire workload) can be efficiently sped up by exploiting massively parallel platforms.

With three industry-scale CPS models, we experimentally show that the use of an ensemble of two state-of-the-art SAs ($\mathcal{AA}$ and EBGStop) may require dozens of millions fewer samples when compared to running a single algorithm, with reductions in sample size of up to 78%. Furthermore, we show that our implementation, by massively parallelizing system model simulations on a HPC infrastructure, yields speed-ups for the completion time of the verification tasks which are practically linear with respect to the number of computational nodes, thus achieving an efficiency of virtually 100%, even on very large platforms. This makes it possible to complete tasks of model-based SMC verification for complex CPSs in a matter of hours or days, whereas a naïve sequential execution would require from months to many years.

## 1. Introduction

The growing need for tools designed for the formal verification of complex systems faces a fundamental problem of scalability. In particular, for Cyber-Physical Systems (CPSs) (Clarke and Zuliani, 2011), *i.e.*, systems with physical components which are monitored and controlled by integrated computer entities to execute a specific task safely and effectively (most often in a dynamic fashion and in a nondeterministic environment), exhaustive methods of model checking are basically always impossible to apply. For instance, even the simplest model of a prototype for a self-driving vehicle will have too many states and possible paths of executions for any comprehensive algorithm to be run within a realistic time frame. This is due to the state-explosion problem (Clarke et al., 2016): the size of a state-based model scales exponentially with the number of variables used to represent the system.

Therefore, stochastic algorithms must be used instead. Statistical Model Checking (SMC) (Larsen and Legay, 2016) offers a wide range of frameworks and probabilistic methods to analyze and verify large models of complex systems. While these tools generally cannot provide definite answers to verification problems, they can actually be applied to real-world use-cases and generate approximations of Key Performance Indicators (KPIs) of the system which come with user-defined probabilistic guarantees. SMC, sometimes in conjunction with machine learning (Larsen et al., 2022), has for instance been successfully exploited for the validation and verification or privacy- and security-sensitive structures, such as railway signaling systems or information

☆ Editor: Dr. Jacopo Soldani.

* Corresponding author.
*E-mail addresses:* picchiami@di.uniroma1.it (L. Picchiami), maxime.parmentier@uclouvain.be (M. Parmentier), axel.legay@uclouvain.be (A. Legay), tmancini@di.uniroma1.it (T. Mancini), tronci@di.uniroma1.it (E. Tronci).

https://doi.org/10.1016/j.jss.2024.112238
Received 26 January 2024; Received in revised form 24 July 2024; Accepted 2 October 2024
Available online 17 October 2024

sharing platforms for healthcare (Basile et al., 2022; Baranov et al., 2022).

### 1.1. Motivation

Designing experiments and protocols which minimize the size of a required sample is a very old problem in statistics (Singh and Masuku, 2014). Hence, many techniques have already been developed to improve the sampling strategies in the context of SMC in order to reduce the number of required samples (see Barbot et al., 2012; Jegourel et al., 2012; D'Argenio et al., 2015; Parmentier et al., 2024 for some examples).

One such techniques is the use of *Adaptive Stopping Algorithms* (*SAs*), *i.e.*, approximation algorithms that do not fix the number of required samples in advance, but continuously monitor their own progress and gradually ask for new samples until enough knowledge has been accumulated to compute an approximation of the quantity of interest with the sought statistical properties. Some SAs have been proven to be *optimal*, in that they are probabilistically guaranteed to stop after a number of samples which is always within a constant factor from the minimum number of samples theoretically needed to compute the requested approximation.

SAs can be applied in an extremely large range of situations since they only modify the sampling process. For example, they have been used to scale up machine learning algorithms (Bradley and Schapire, 2007) or to offer an efficient solution to the multi-armed bandit problem (Audibert et al., 2007a; Even-Dar et al., 2002). They have been extensively studied for their applications for SMC as well (Domingo and Watanabe, 2000; Domingo et al., 2002; Mnih et al., 2008; Dagum et al., 2000).

Introducing parallelization to SMC algorithms can seem to be an obvious solution to enhance their performance and scale of use. While in some cases (AlTurki and Meseguer, 2011; Pappagallo et al., 2020), this has been done with great results, the complexity of most of the more advanced techniques of SMC makes it often very difficult to implement at best, or fundamentally impossible at worst (Bulychev et al., 2011). However, it turns out that since SAs only focus on and determine the sampling process itself, without even having any restriction as to how the samples are produced, SAs are not just efficient at minimizing the sample size, but are also highly modular and well-suited to be parallelized.

The various SAs available (some of them are described in Section 2.2) mainly differ with respect to the statistical result exploited to derive their stopping criterion. As a consequence, they may require a fewer or a higher number of samples to converge, depending on various aspects of the problem at hand.

In the case of model-based SMC-driven verification of CPSs, generating each sample is particularly expensive, as this typically requires to numerically simulate the system model under an operational scenario randomly chosen from some distribution, in order to compute the value of a specific KPI. Thus, overshooting the size of the sample by a large margin can have a critical impact on the running time of the approximation algorithm. If the system deals with online processes, this can even completely invalidate the verification task.

To compute approximations of a system KPI as fast and as efficiently as possible, one needs to select the SA which would stop after the smallest possible number of samples for the problem at hand. Unfortunately, the number of samples needed by any given SA to converge depends not only on the requested properties of the approximation, but also on statistical properties of the KPI of interest, which are typically unknown *a priori*.

### 1.2. Contribution

In this article, we address the limitations of model-based verification of CPSs via SMC in two ways:

1. We propose Ensemble of Approximation Algorithms (EAA), a composite algorithm that concurrently runs a set (*ensemble*) of different SAs, by feeding them all with the same sequence of independent and identically distributed (iid) samples, and terminating as soon as the first SA of the ensemble converges with an approximation of the quantity of interest satisfying the requested statistical guarantees.
2. We show a very effective architecture that exploits High-Performance Computing (HPC) infrastructures to speed-up the generation of samples via the use of many identical simulators running in parallel, still guaranteeing that the generated sequence of samples is *identical* to the sequence that would be generated by a sequential sampler (if launched with the same random seed). This avoids any bias in the (pseudo-)randomness of the samples and so in the final result of the approximation.

We present our implementation combining two of the most well-known SAs as an ensemble, EBGStop (Mnih et al., 2008) and $\mathcal{AA}$ (Dagum et al., 2000), and evaluate its performance by conducting simulation-based SMC-driven verification of Simulink/Stateflow models of three CPSs of industrial size: Automatic Transmission (AT), Fuel Control System (FCS) and Apollo Lunar Module Autopilot (ALMA).

We emphasize that EAA can be seamlessly applied to virtually any SMC task, beyond the verification of CPSs. However, in this article, we explicitly focus on model-based CPS verification, since in this setting sample size minimization is crucial, given the high cost of generating each sample (which entails the numerical simulation of the system model).

In this context, we show that exploiting EAA can significantly reduce the number of necessary samples (requiring, in our case studies, up to 31 300 400 and up to 78.73% fewer samples) when compared to running a single SA, and that our implementation for parallel samples generation shows an efficiency very close to 100%, even when using large platforms. Overall, the combined effect of using EAA and of massively parallelizing the samples generation process allows us to carry out in a matter of hours or days SMC-based verification tasks for our case studies that would have taken months or even many years to complete if run sequentially.

### 1.3. Outline

This article is organized as follows. Section 2 presents preliminaries and discusses state-of-the-art SAs, with a special emphasis on EBStop and $\mathcal{AA}$. Section 3 defines the formal framework used throughout the article. Section 4 and Section 5 describe EAA and our massively parallel implementation, respectively. Section 6 is devoted to the experimental analysis of our algorithm and parallel tool. Finally, Section 7 draws conclusions and perspectives.

## 2. Preliminaries and related work

In this section we introduce some key mathematical concepts and discuss the notion of $(\varepsilon, \delta)$-approximation algorithm. We then describe two optimal SAs available from the literature.

We denote by $\mathbb{R}$, $\mathbb{R}_{0+}$, $\mathbb{Z}$, $\mathbb{N}$ the sets of, respectively, real, non-negative real, integer, and non-negative integer numbers. Given sets $A$ an $B$, $A^B$ denotes the set of functions from $B$ to $A$. Finally, given a random variable $X$, we denote by $\mathbb{E}(X)$ the expectation of $X$.

### 2.1. Unbiased estimators

Given a real-valued random variable $X$, an estimator $\hat{p}$ for one of its parameters $p$ is *unbiased* if $\mathbb{E}(\hat{p}) = p$. An estimator is an $(\varepsilon, \delta)$-estimator if for any precision $\varepsilon \in (0, 1)$ and confidence margin $\delta \in (0, 1)$, there exists a sample size $N_{(\varepsilon,\delta)}$ such that the produced estimation is within $[p(1-\varepsilon), p(1+\varepsilon)]$ with a probability greater than $1-\delta$. Such an approximation is called a $(\varepsilon, \delta)$-approximation.

In the case of the expected value of a random variable $X$, the classic sample mean estimator is the go-to estimator. If sampling is performed with respect to the distribution of $X$, it is an unbiased estimator. In that case, it follows directly from the central limit theorem that it is an $(\varepsilon, \delta)$-estimator as well.

### 2.2. Adaptive Stopping Algorithms

Classical estimation algorithms for the expected value of a random variable, such as Monte-Carlo algorithms, fully determine their sample size *a priori*, for instance by exploiting the Chernoff bound (Clarke and Zuliani, 2011; Boyer et al., 2013; Agha and Palmskog, 2018). This approach suffers from three key weaknesses: (a) it is completely agnostic with respect to the random variable under study, (b) it can overshoot the size of the required sample by a large margin, and (c) it is not updated and improved during the execution of the algorithm.

*Adaptive Stopping Algorithms (SAs)* are approximation algorithms that have none of those issues. They are designed so to require an as small as possible sample size for the computation of the sought approximation. They manage to do so by not only exploiting more advanced results of statistics, such as the Hoeffding's inequality (Domingo and Watanabe, 2000), and ideas inspired from the theory of sequential testing, but also by continuously updating their stopping criterion with the information that they have already accumulated at any step to decide whether they have reached a sample size which will guarantee a $(\varepsilon, \delta)$-approximation.

Algorithm 1 is an example of a generic and simplified SA which outputs an $(\varepsilon, \delta)$-approximation of the expected value of random variable $X$. The defining characteristic of an SA is its *stopping criterion*: the algorithm loops and progressively acquires a larger and larger sample, updating the estimation $\hat{\mu}$ and the stopping criterion itself at each step, until the test of the stopping criterion is finally satisfied, at which point the sample size ensures that $\hat{\mu}$ is an $(\varepsilon, \delta)$-approximation of $\mathbb{E}(X)$.

**Algorithm 1:** Generic Adaptive Stopping Algorithm (SA)

```
input: random variable X;
input: ε, δ ∈ (0, 1);
t ← 0;
samples ← ∅;
repeat
    t++;
    samples ← samples ∪ {x_t};
    μ̂ ← mean(samples);
    update stopping criterion;
until stopping criterion satisfied;
return μ̂;
```

The fact that the stopping criterion of a SA depends on and takes into account (the realizations $x_t$ of) the random variable and that it is continuously updated throughout the progress of the algorithm is necessary but is not enough in itself to guarantee that the final sample size will not be larger than necessary. This must be proven from the statistical result from which the stopping criterion has been derived, hence the difficulty to come up with new and useful SAs.

Concentration inequalities are however a good source of theorems that can lead to meaningful stopping criteria. For instance, the NAS algorithm (Domingo and Watanabe, 2000) is a SA developed for bounded random variables with expected value $\mu \neq 0$. It is a straightforward SA whose stopping criterion is of the form $|\hat{\mu}_t| < c_t$, where $c_t = \sqrt{\frac{\log(t(t+1)/\delta)}{2t}}(1 + 1/\varepsilon)$ are parameters of the algorithm. The expressions for these parameters are derived from the Hoeffding's inequality (Hoeffding, 1994).

The EBStop algorithm (Mnih et al., 2008) was introduced in 2008 as an improvement of the NAS algorithm. Similarly to its predecessor, EBStop builds a sequence of parameters $c_t$ as its stopping criterion. However, by exploiting the (empirical) Bernstein's inequality (Audibert et al., 2007a) instead of the Hoeffding's inequality, the variance of the random variable $X$ takes part in the computation of such parameters. Especially when the variance is significantly smaller than the range $R$ of the random variable, these parameters decrease much faster, therefore putting an end to the sampling process much sooner. For any fixed sequence $(d_t)_{t\in\mathbb{N}}$ such that $\sum_{t=1}^{\infty} d_t \leq \delta$, the parameters $c_t$ can be taken as $c_t = \hat{\sigma}_t\sqrt{\frac{2\log(3/d_t)}{t}} + \frac{3R\log(3/d_t)}{t}$, with $\hat{\sigma}_t$ being the sample standard deviation at step $t$. The EBStop algorithm adds another improvement to the most basic version of the NAS algorithm. The stopping criterion is divided into a lower stopping criterion $(1+\varepsilon)\max\left\{0, \max_{1\leq s\leq t}(|\hat{\mu}_s| - c_s)\right\}$ and an upper stopping criterion $(1+\varepsilon)\min_{1\leq s\leq t}(|\hat{\mu}_s| - c_s)$, so that $X$ must not be supposed non-negative. The most upgraded version of the algorithm, called EBGStop, also adopts a geometric sampling strategy, accumulating samples at an exponential rate $\beta$ rather than with a linear rate. The right choice of $\beta$ and additional countermeasures (Mnih et al., 2008; Audibert et al., 2007b) allow for a speed-up of the algorithm without running the risk of taking more samples than needed.

If $R > 0$ is the range of the random variable $X$, Mnih et al. proved that the EBStop algorithm will require a sample size smaller than a multiple of $\max\left\{\frac{\sigma^2}{\varepsilon^2\mu^2}, \frac{R}{\varepsilon|\mu|}\right\} \times \left(\log\frac{1}{\delta} + \log\frac{R}{\varepsilon|\mu|}\right)$, which guarantees its (near-)optimality in a similar way to is main competitor: the $\mathcal{AA}$ algorithm.

The $\mathcal{AA}$ algorithm (Dagum et al., 2000) is a SA in three steps. It was built upon the theory of supermartingales and a generalized version of the zero–one estimator theorem. It can only be applied to any bounded and non-negative random variable $X$. $\mathcal{AA}$ is significantly more complex than EBStop. Actually, $\mathcal{AA}$ can be seen as a relatively basic Monte-Carlo algorithm for its second and third steps, with the size of the required sample being computed indirectly with a SA during the first step. $\mathcal{AA}$ first computes a $\left(\max\left\{\sqrt{\varepsilon}, \frac{1}{2}\right\}, \frac{\delta}{3}\right)$-approximation $\tilde{\mu}$ of $\mu$, which is only used to compute a Monte-Carlo estimation $\tilde{\sigma}^2$ of $\sigma^2$, which is in turn exploited to derive a value $N = \max\left\{\tilde{\sigma}^2, \varepsilon\tilde{\mu}\right\}\frac{\log(\frac{1}{\delta})}{\varepsilon^2\tilde{\mu}^2}$ for the final sample size which guarantees that the final Monte-Carlo estimation $\hat{\mu}$ is an $(\varepsilon, \delta)$-approximation $\hat{\mu}$ of $\mu$.

While the hypothesis that $X$ is non-negative makes $\mathcal{AA}$ a bit more restrictive than EBGStop, it was proven to be fully optimal for that class of random variables. Namely, for any bounded and non-negative random variable $X$, Dagum et al. showed that with probability $1-\delta$, $\mathcal{AA}$ will require a sample size that is at most a multiple of $\max\left\{\sigma^2, \varepsilon\mu\right\} \times \frac{1}{\varepsilon^2\mu^2}\log(\frac{2}{\delta})$, while also proving that any SA designed for that class of random variables will always require with probability $1-\delta$ a sample size of at least a multiple of that exact quantity.

Contrary to the expression of optimality of EBGStop, the expression of optimality of $\mathcal{AA}$ does not involve the range of the random variable, which can be linked to the fact that while both algorithms exploit information about the variance, they do not do it in the same way. This hints at the possibility that even for non-negative bound random variables, for which both algorithm can be applied, EBGStop can outperform $\mathcal{AA}$ in practice.

## 3. Formal framework

CPSs are naturally modeled as dynamical systems (see, *e.g.*, Sontag, 1998). Model-based verification of a CPS amounts to checking whether some system requirements are verified. Such a task thus requires three components: a model of the system, the set of possible

*operational scenarios* (*i.e.*, input time functions), and a formalization of the specification to be verified.

Due to their complexity and diversity, many different formal frameworks have been proposed to define models for CPSs. State-based transition models, defined explicitly or through differential equations (with discrete or continuous time) are the most prevalent.

In this article however, for the sake of generality, we abstract away from any specific formal framework for the definition of the system model and simply assume (along the lines of, *e.g.*, Mancini et al., 2014, 2016a; Esposito and Picchiami, 2022a,b) that our *System Under Verification (SUV)* $\mathcal{S}$ is modeled (Definition 1) as a black-box input–output deterministic strictly causal dynamical system, which takes as input an *operational scenario*, *i.e.*, a time function of both the controllable inputs and the other uncontrollable events $\mathcal{S}$ is subject to (*e.g.*, faults in sensors and actuators or changes in system parameters), and produces a time function of system outputs (called *output function* or *trajectory*).

**Definition 1** (*System Under Verification Model*)**.** A System Under Verification (SUV) model $\mathcal{S}$ is a tuple $(\mathcal{T}, U, O, S)$ where:

- $\mathcal{T}$ is the *time-set* ($\mathbb{R}_{0+}$ or $\mathbb{N}$ for continuous- and discrete-time systems, respectively, or an interval thereof);
- Sets $U$ and $O$ are the *input and output spaces*;
- $S : U^{\mathcal{T}} \to O^{\mathcal{T}}$ is the *I/O* (or *simulation*) *function*. For any *input function* $\mathrm{u} \in U^{\mathcal{T}}$ (*operational scenario*), $S(\mathrm{u})$ denotes the *output function* $\mathrm{o} \in O^{\mathcal{T}}$ of $\mathcal{S}$, defining the output $\mathrm{o}(t)$ of $\mathcal{S}$ for each time point $t \in \mathcal{T}$, when the system is fed with u.

With a small abuse of notation, we denote by $S(t; \mathrm{u})$ the output of $\mathcal{S}$ at time point $t$ when fed with u, *i.e.*, $S(\mathrm{u})(t)$.

A *strictly causal* SUV model $\mathcal{S}$ is such that, for any $t \in \mathcal{T}$ and for any $\mathrm{u}_1, \mathrm{u}_2 \in U^{\mathcal{T}}$ such that $\mathrm{u}_1(t') = \mathrm{u}_2(t')$ for all $t' < t$, it holds $S(t; \mathrm{u}_1) = S(t; \mathrm{u}_2)$.

Models for SUVs must therefore at the very least be executable (black box) models, with respect to discrete or continuous time, such that they output a unique output function (trajectory) for any given operational scenario.

The full set of operational scenarios of interest for a system (*i.e.*, those considered possible or on which the verification activity needs to focus), together with a probability measure of each scenario to materialize is collectively called the *environment* of the SUV (Definition 2). For practical reasons, we focus on *finitely parameterizable* system environments, *i.e.*, environments whose scenarios can be modeled via a finite real- or discrete-valued parameter vector. We remind that a scenario encodes both user (controllable) inputs to the system and other uncontrollable events.

**Definition 2** (*Stochastic System Environment*)**.** Let $\mathcal{S} = (\mathcal{T}, U, O, S)$ be a SUV as in Definition 1. An environment $\mathrm{Env} = (\mathbb{U}, \mathrm{P})$ for $\mathcal{S}$ is defined by a set $\mathbb{U} \subseteq U^{\mathcal{T}}$ of scenarios and an associated probability distribution $\mathrm{P} : \mathbb{U} \to [0, 1]$.

An environment is *finitely parameterizable* if there exist a finite-dimension parameter vector space $\Lambda$ and a bijection between $\Lambda$ and $\mathbb{U}$.

A metric KPI of the SUV is a function associating a real value to each system trajectory. With Definition 3 we limit ourselves to normalized KPIs, that is KPIs assuming values in interval $[0, 1]$.

**Definition 3** (*Normalized Metric KPI*)**.** A normalized metric KPI for SUV $\mathcal{S}$ is a function $\kappa : O^{\mathcal{T}} \to [0, 1]$.

In order to ensure that also $(\varepsilon, \delta)$-approximation algorithms requiring *bounded non-negative* random variables, such as $\mathcal{AA}$ (see Section 2.2), can be exploited by EAA, in the following we focus on model-based SUV verification problems where the property to be assessed is threshold-based (Definition 4):

**Definition 4** (*Threshold-based Verification Problem*)**.** A threshold-based verification problem for a SUV model $\mathcal{S} = (\mathcal{T}, U, O, S)$, an environment $\mathrm{Env} = (\mathbb{U}, \mathrm{P})$, a threshold $\vartheta \in [0, 1]$ and a normalized metric KPI $\kappa : O^{\mathcal{T}} \to [0, 1]$ consists of deciding whether $\mathbb{E}_{\mathrm{u} \in \mathbb{U}}(\kappa(S(\mathrm{u}))) \leq \vartheta$, where the expected value is computed with respect to the probability distribution P of scenarios as defined in Env.

Any approximation algorithm which provides an unbiased $(\varepsilon, \delta)$-estimator for the expected value of a $[0, 1]$-valued random variable can be applied to the binary random variable $X = (S(\mathbb{U}) \leq \vartheta)$ to offer a probabilistic answer to a threshold-based verification problem. Indeed, if an $(\varepsilon, \delta)$-approximation $\hat{\mu}$ of $\mu = \mathbb{E}(\kappa(S(X))$ can be computed, then we know that, with probability at least $1 - \delta$:

$$\mu(1 - \varepsilon) \leq \hat{\mu} \leq \mu(1 + \varepsilon).$$

Therefore:

$$\frac{\hat{\mu}}{1 + \varepsilon} \leq \mu \leq \frac{\hat{\mu}}{1 - \varepsilon}.$$

In conclusion, since $\frac{\hat{\mu}}{1-\varepsilon} \leq \vartheta$ implies that $\mu \leq \vartheta$, it suffices to check whether:

$$\frac{\hat{\mu}}{1 - \varepsilon} - \vartheta \leq 0. \tag{1}$$

If that inequality holds, then the requirement of the system associated with the threshold $\vartheta$ and the KPI $\kappa$ is successfully verified with a precision of $\varepsilon$ and a $1 - \delta$ level of confidence. A threshold-based verification problem with a lower bound threshold can be defined and solved in a very similar fashion.

## 4. Ensemble of Approximation Algorithms (EAA)

As explained in Section 3, any adaptive SA which computes an $(\varepsilon, \delta)$-approximation of the expected value of a $[0, 1]$-valued random variable provides a SMC method to solve threshold-based verification problems. Moreover, as discussed in Section 2, the number of samples *actually* needed by any given SA to converge on a given verification task depends not only on the requested properties of the approximation, but also on statistical properties of the KPI to be approximated, which are typically unknown in advance. In settings, such as SMC model-based verification of CPSs where generating samples of the system KPI of interest is particularly expensive, because it requires numerical simulation of the CPS model, using a single SA to perform SMC might result in an unnecessary large number of samples to be generated. This is true also for those SAs proved to be optimal like $\mathcal{AA}$, since optimality in the number of required samples is guaranteed only probabilistically and up to a multiplicative factor.

In a similar fashion to AI-based techniques such as *ensemble learning* methods (Sagi and Rokach, 2018; Dong et al., 2020), we propose a general procedure, named EAA, that exploits a finite set $\mathcal{A}_1, \ldots, \mathcal{A}_k$ of different $(\varepsilon, \delta)$-approximation algorithms.

The EAA algorithm (see Fig. 1 for a conceptual high-level view and pseudocode) first initializes the states of all the algorithms in the ensemble (line 9). Then, it repeatedly generates samples of random variable $X$. In our case, this means to generate a random scenario u for the SUV (line 10), to simulate the SUV model on it, and to compute the associated value ($x$) of the KPI of interest. Each sample $x$ is then used to feed and advance *all* algorithms of the ensemble (line 12) *by one step only* (*i.e.*, processing only $x$). This is done by functions $advance_{\mathcal{A}_l}()$ ($l \in [1, k]$).

As soon as one of the algorithms (say $\mathcal{A}_l$) reaches its termination condition, EAA stops and returns the approximation $\hat{\mu}_l$ provided by $\mathcal{A}_l$ (line 15), which is a valid $(\varepsilon, \delta)$-approximation of the true mean $\mu$ of the value of the KPI of interest. Value $\hat{\mu}_l$ is then used to answer the threshold-based verification problem, by checking whether Eq. (1) holds.

Since the procedure terminates as soon as the first algorithm in the ensemble, say $\mathcal{A}_l$, reaches *its own* stopping criterion, the number

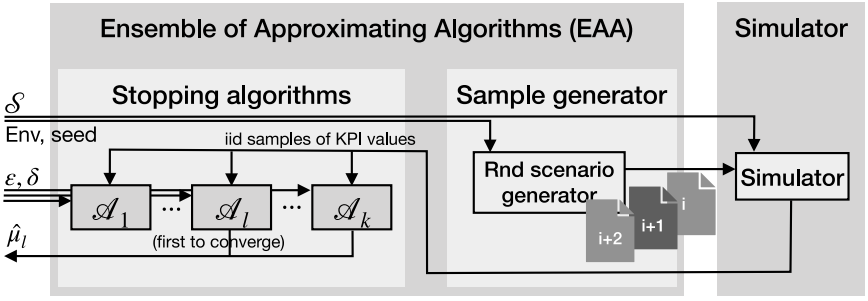


**Fig. 1.** Conceptual view of EAA (sequential reference architecture).

of samples required by EAA is always the *minimum* number of samples that would have been required by each algorithm in the ensemble if it were run alone.

The design of EAA revolves around proper subroutines that advance each algorithm in the ensemble by one step (*i.e.*, processing just one sample) at the time and manipulate the algorithm persistent state appropriately. Namely, each $advance_{\mathcal{A}_l}()$ subroutine: (a) takes as input the current state $s_l$ of algorithm $\mathcal{A}_l$ and a new sample $x$; (b) processes $x$ by advancing the algorithm state into $s'_l$; (c) returns the resulting state $s'_l$. Such subroutines can be easily written by refactoring any approximation algorithm. In our implementation (see Section 6) we selected two of the most widely used algorithms known to be optimal or quasi-optimal (probabilistically and up to a multiplicative factor): EBGStop and $\mathcal{AA}$, already described in Section 2.2. Advancing each algorithm in the ensemble only requires a few algebraic operations. Thus, the computational overhead of advancing multiple algorithms in the ensemble is expected to be *negligible* with respect to the (much longer) time to perform simulations of the SUV model to produce each sample. This expectation will be shown correct in Section 6.4.2.

**Algorithm 2:** EAA

```
1  input: 𝒮, SUV model;
2  input: Env, stochastic environment;
3  input: seed, random seed;
4  input: ε, δ ∈ (0, 1);
5  input: 𝒜_1, ..., 𝒜_k, set of (ε, δ)-appr. algorithms;
6  initialize rnd scen. generator with Env and seed;
7  initialize simulator with 𝒮;
8  initialize 𝒜_1, ..., 𝒜_k with ε and δ;
9  s_1, ..., s_k ← initial states of 𝒜_1, ..., 𝒜_k;
10 while true do
11     x ← next_sample();
12     for l from 1 to k do
13         s_l ← advance_{𝒜_l}(s_l, x) ; // feed 𝒜_l with x
14         if 𝒜_l has reached its stopping cond. then
15             return μ̂_l; // the estimated mean from 𝒜_l
16 function next_sample()
17     u ← generate next rnd scenario;
18     x ← simulate u; // obtain KPI value (sample)
19     return x;
```

## 5. Parallel scenario evaluation over high-performance computing infrastructures

SMC-based verification often requires a huge number of samples, that is, computer simulations of the CPS model. As discussed in Section 1, this is particularly the case for complex systems. For example, verifying the properties defined in Section 6.1 on our case studies required up to more than 100 million samples.

While EAA allows us to stop the verification process as soon as the first algorithm of the ensemble reaches its termination condition, it does not address the second major bottleneck of model-based CPS verification via SMC: the rate at which samples are provided to the approximation algorithm. For our case studies, generating a single sample took, on average, from 0.1 to several seconds, depending on the model. As a consequence, the verification activities we performed (see Section 6) would have required up to many years for EAA to terminate if conducted naïvely.

In this section we outline our approach to exploit HPC platforms to effectively and efficiently speed-up the generation of samples to be provided to EAA.

Our overall architecture is shown in Fig. 2. It is designed to run on a possibly very large HPC infrastructure, and includes: a multi-process or multi-threaded module devoted at advancing the SAs of the ensemble one sample at a time (lines 8–15 of Algorithm 2); $n$ instances (with $n$ which can be very large) of a simulation module, each one running an identical copy of a simulator of the SUV model; one multi-process or multi-threaded module hosting a service, named Asynchronous Parallel Sample Generator (APSG), devoted to the generation of random scenarios and the delegation of the simulation of each of them to one of the $n$ available parallel simulators. Since the latter service runs asynchronously with respect to the SAs, the overall architecture implements an instance of the *producer–consumer paradigm*.

The exact operation of APSG is described in details by Algorithm 3. First, function *initialize*() (line 6) initializes APSG with the given SUV model $\mathcal{S}$, its environment Env, and a random seed to generate scenarios from Env. This function instantiates the $n$ simulators with identical copies of $\mathcal{S}$ and initializes the random scenario generator. Note that, since the environment is finitely parametrizable, scenario generation reduces to generating pseudorandom numerical vectors within some bounded finite-dimensional domain (see Definition 2). Then, function *initialize*() starts process (or thread) *generate_samples()* and returns immediately.

Process *generate_samples()* (line 10) continuously generates (see bullet point 1 in Fig. 2) a random scenario u from the system stochastic environment, retrieves (*dequeues*) a simulator *sim* from the queue of idle simulators (bullet point 2), and delegates to *sim* the two tasks of simulating $\mathcal{S}$ under scenario u and of computing the value of the KPI of interest. The queue of idle simulators is initially filled with identifiers of all the available simulators (line 8). If found empty, the process waits (*blocking dequeue* operation) for a simulator to become idle (line 13). Each scenario is identified by a progressive number (integer $i$ in the pseudocode).

Whenever a simulator terminates its task on scenario $i$, having computed value $x$ for the KPI of interest, it calls function *sample_produced*() of APSG. This function (line 16 and bullet point 3 in the figure) inserts entry $i \mapsto x$ in the map *samples*, thus storing the value of the system KPI on scenario $i$ (this corresponds to the *produce* step in the producer–consumer paradigm), and the simulator is added to the queue of idle simulators, signaling that it is ready to receive new simulation requests. To prevent memory explosion, map *samples* has a maximum capacity. If the map is full, the insertion of the new sample blocks until an entry is removed.

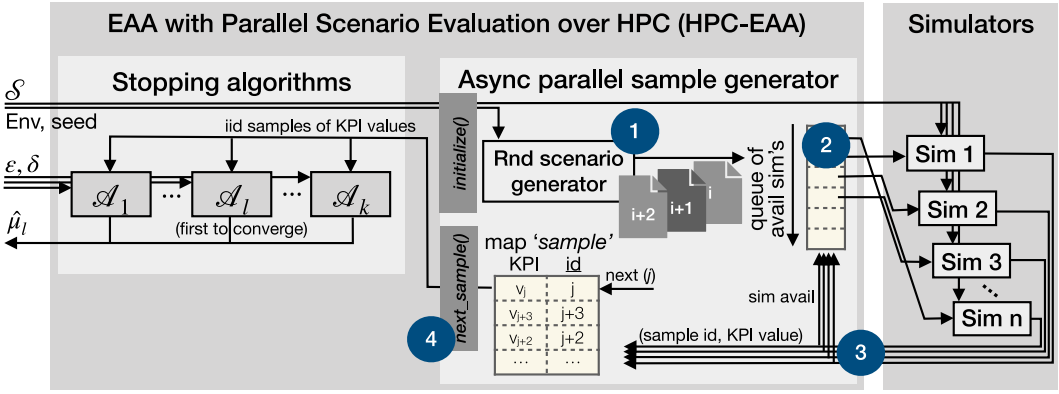


**Fig. 2.** HPC-EAA: high-level parallel architecture.

**Algorithm 3:** Asynchronous Parallel Sample Generator (APSG)

```
1  global i = 0; // id of the next sample to be produced
2  global j = 0; // id of the next sample to be consumed by EAA
3  global avail_sims = ∅; // queue of avail. simulators
4  global samples = ∅; // map: sample id ↦ KPI value
5  param n; // number of available simulators
6  function initialize(S, Env, seed)
7      initialize random generator with Env and seed;
8      initialize the n available simulators with S and enqueue
         them all to avail_sims;
9      start process generate_samples();
10 process generate_samples()
       // continuous process
11     while ¬ halt requested do
12         u ← generate rnd scenario;
13         sim ← dequeue from avail_sims; // wait if queue is empty
14         send (i, u) to sim for simulation and KPI evaluation;
15         i++;
16 function sample_produced(sim, i, x)
       // called by simulator sim when evaluation of scenario i is
          completed (KPI value is x)
17     add (i ↦ x) to samples; // wait if map is full
18     enqueue sim to avail_sims;
19 function next_sample()
       // called by EAA
20     x ← samples[j]; // wait if not yet available
21     free-up samples[j];
22     j++;
23     return x;
```

The module running the SAs continuously (and asynchronously) requests samples from APSG by calling function *next_sample()* (this corresponds to the *consume* step in the producer–consumer paradigm). This function (line 19 and bullet point 4 in the figure) returns the KPI value associated to scenario with id $j$ in the map *samples* and waits in case such an entry does not (yet) exist (because not yet computed). Before returning, this function frees-up memory by removing from map *samples* the entry associated to scenario $j$ and increments $j$, which becomes the id of the next sample to be provided to the SAs.

It is worth noting that APSG provides the SAs with the *very same* sequence of samples that would be produced with the basic (purely sequential) architecture of Fig. 1, if run with the same random seed. In other words, the order of the scenarios generated is *independent* of the number $n$ of simulators. Thus, increasing $n$ will have no other effect than increasing the rate at which the samples are made available to SAs, hence reducing the total completion time. In this way, we avoid any bias in the (pseudo-)randomness of the samples and so in the final result of the approximation.

## 6. Experiments

In this section we present our three industrial case-study models taken from the literature, outline the implementation of our HPC software, and use our tool to perform model-based SMC verification tasks of our three models.

Our results highlight how the conjoint effects of using an ensemble of $(\varepsilon, \delta)$-approximation algorithms and a massively parallel architecture to perform (the expensive) scenario simulations required to provide EAA with iid samples allow us to complete our verification tasks within a reasonable time frame, whereas a naïve implementation would have been taken an *inconceivably* long time.

### 6.1. Case studies

Our case studies were realized with three Simulink/Stateflow models of industry-scale systems, namely: Automatic Transmission (AT) (Mathworks, 2024b), Fuel Control System (FCS) (Mathworks, 2024c) and Apollo Lunar Module Autopilot (ALMA) (Mathworks, 2024a). These models have already been profusely studied in the literature, see, *e.g.*, Mancini et al. (2022, 2021, 2023), Zuliani et al. (2013), Mancini et al. (2014, 2017, 2016b), Hoxha et al. (2014), Abbas et al. (2013), Barbot et al. (2020).

#### 6.1.1. Automatic Transmission (AT)

*System model.* The AT model defines a typical automotive drivetrain. Non-linear ordinary differential equations are used to specify the engine dynamics, the four-speed automatic transmission and the vehicle dynamics. The model has two inputs, the throttle opening (expressed in percentage) and the brake torque (expressed in ft-lb), and two outputs, the engine speed (in RPM) and the vehicle speed (in mph). The throttle opening delineates the accelerating dynamics, whereas the brake torque represents the braking dynamics of the vehicle.

In the literature, the AT system has been extensively used for search-based falsification purposes (see, *e.g.*, Abbas et al., 2013; Hoxha et al., 2018; Dokhanchi et al., 2015) and benchmarks (*e.g.*, Hoxha et al., 2014; Ernst et al., 2022), especially when the design of input trajectories has a strong impact on the verification (see, *e.g.*, Barbot et al., 2020).

*Environment.* We modeled a stochastic environment for the system that generates disturbances as input time functions for the SUV model. In our setting, the brake is always set to 0, whereas the throttle presents a fixed average value $\mu_d$ that is perturbed with a given variance $\sigma_d^2$, *i.e.*, the values are sampled from a normal distribution $N(\mu_d, \sigma_d^2)$ truncated within $[0, 100]$. We set $\mu_d = 40$ and $\sigma_d^2 = 20$ for the truncated normal distribution, thus modeling a varying accelerating dynamics.

*Property to be verified.* In the context of search-based falsification, a typical system-level specification requires that the engine and vehicle speeds do not exceed given safety thresholds for the whole simulation duration. Therefore, we designed a (metric) safety property which quantifies how much time the engine and the vehicle speeds spend above such thresholds.

To define our system-level requirement, we first have to define the maximum time window during which any of the two speeds is above its threshold. We define the *Max Time Speed* (MTS) and the *Max Time Engine* (MTE) at time $t$ as:

$$\mathrm{MTS}(t) = \max\left\{t_2 - t_1 \;\middle|\; 0 \le t_1 \le t_2 \le t \wedge (\forall t' \in [t_1, t_2] : \mathrm{speed}(t') > M_s)\right\}$$
$$\mathrm{MTE}(t) = \max\left\{t_2 - t_1 \;\middle|\; 0 \le t_1 \le t_2 \le t \wedge (\forall t' \in [t_1, t_2] : \mathrm{engineRPM}(t') > M_e)\right\}$$

where $\mathrm{speed}(t)$ and $\mathrm{engineRPM}(t)$ are the values of the vehicle and engine speeds at time $t$, whilst $M_s$ and $M_e$ are the respective maximum safe values. The *Max Time Violation* at time $t$ is then defined as:

$$\mathrm{MTV}(t) = \frac{\max\{\mathrm{MTS}(t), \mathrm{MTE}(t)\}}{h}$$

where $h$ is the simulation time horizon.

The verification activity aims to establish whether $\mathbb{E}(\mathrm{MTV}(h)) \le \vartheta$ for some threshold. We set $M_s = 4000$ RPM and $M_e = 120$ RPM (according to previous work, see, *e.g.*, Fan et al., 2017; Claessen et al., 2018), $h = 1800$ s (*i.e.*, 30 minutes of system simulation time), and $\vartheta = 5\%$, so to assess whether the system violates the desired specification for any time window longer than 5% of the time.

#### 6.1.2. Fuel Control System (FCS)

*System model.* The FCS model represents a control system for a fault-tolerant gasoline engine that is expected to accept up to one sensor fault, and has been widely used as a benchmark of various simulation-based verification approaches (see, *e.g.*, Zuliani et al., 2013; Mancini et al., 2023; Hoxha et al., 2014; Mancini et al., 2022, 2021 and citations therein).

The system has four sensors: the *Throttle Position* (TP), *Engine Speed* (ES), *Exhaust Gas Oxygen* (EGO) and *Manifold Air Pressure* (MAP). The TP and ES sensors provide the current air throttle position and the engine speed to the engine controller. The EGO sensor reads the current oxygen level in the engine's exhaust gas to adapt the fuel supply, whereas the MAP sensor reads the suction pressure of the airflow at the engine's intake manifold.

The model outputs the following two quantities: the *air-fuel ratio*, that is, the ratio between the air mass flow rate computed by the intake manifold and the fuel mass flow rate pumped by the injectors, and the *fuel flow rate* itself.

The system aims at keeping the air-fuel ratio close to the stoichiometric (ideal) value of 14.6 since it is a good compromise between power, fuel economy and emissions. However, this requirement becomes more challenging to meet in the presence of sensor faults. In particular, when a fault on a single sensor is detected, the system modifies its behavior by operating the engine with a higher fuel rate. In case of two or more sensor faults, the system shuts down the engine.

*Environment.* We modeled a stochastic environment for the system where temporary faults may occur on the ES, EGO and MAP sensors. Finite state machines (with Stateflow charts) were used to simulate fault injections and the subsequent recovery operations, similarly to, *e.g.*, Zuliani et al. (2013), Hoxha et al. (2014). Each finite state machine injects faults on a given sensor according to an exponential distribution whose mean corresponds to the Mean Time Between Failures (MTBF). Similarly, fault recovery operations occur after a time window whose length is randomly chosen according to an exponential distribution having a mean of 1 s. We set the MTBF values for the EGO, ES, and MAP to 3 s, 7 s and 8 s respectively, which is the configuration used in Zuliani et al. (2013) for verification purposes.

*Property to be verified.* We experimented with one of the key system-level specifications for this model (see, *e.g.*, Mancini et al., 2021, 2022, 2023), that is establishing whether the fuel flow rate is identically zero for too long time. Therefore, we designed a KPI which outputs the (ratio of the) length of the longest time window during which the fuel flow rate is identically equal to zero.

We define *Max Time Fuel* (MTF) at time $t \ge 5$ s as follows (the first 5 s of each trajectory have been ignored to allow for the initial setup of the system):

$$\mathrm{MTF}(t) = \frac{\max\left\{t_2 - t_1 \;\middle|\; 5 \le t_1 \le t_2 \le t \wedge (\forall t' \in [t_1, t_2] : \mathrm{fuel}(t') = 0)\right\}}{h - 5}$$

The goal of the verification is to establish whether $\mathbb{E}(\mathrm{MTF}(h)) \le \vartheta$ for some threshold. We set $h = 105$ s and $\vartheta = 1\%$, so to assess whether the fuel flow rate is identically zero for more than 1% time on average.

#### 6.1.3. Apollo Lunar Module Autopilot (ALMA)

*System model.* The ALMA model defines the logic of the phase-plane control algorithm for the autopilot program of the lunar module used in the Apollo 11 mission. This model has been extensively used in the literature on simulation-based verification (see, *e.g.*, Mancini et al., 2021, 2022 and citations therein).

The system is equipped with 3 sensors (yaw, roll, pitch) and 16 reaction jets that actuate a rotation over one or more axes. In every space mission, *i.e.*, in every system simulation, the controller takes as input a request to change the module's attitude and computes which reaction jets need to be activated to achieve the desired rotation and stabilize the system over the new current attitude.

*Environment.* We modeled a stochastic environment where temporary faults may occur on yaw, pitch and roll sensors, similarly to what we did for the FCS model. For this model, a fault corresponds to the injection of white noise to one of sensor signals. Analogously to the FCS model, the occurrences of the faults are chosen according to an exponential distribution whose mean corresponds to a MTBF of 4 s. Similarly, fault recovery happen after a duration once again determined by an exponential distribution with a mean of 1 s.

*Property to be verified.* We designed a system-level requirement that quantifies the outcome of a given space mission in terms of the average module's attitude error over the entire simulation. To define such an error, we first have to define the attitude error indicators for each axis. For each axis $s$, if $a_s(t)$ and $R_s$ denote the current and target attitudes at time $t$ along $s$, respectively, we define the attitude error $e_s$ at time $t$ as follows:

$$e_s(t) = |a_s(t) - R_s|$$

that is, the absolute difference between the target and the current attitudes of the lunar module along axis $s$. Since the rotation request is over one or more axes, we define the *Attitude Module Error* (AME) at time $t$ as follows:

$$\mathrm{AME}(t) = \max\left\{e_y(t), e_p(t), e_r(t)\right\}$$

where the $e_y(t)$, $e_p(t)$, and $e_r(t)$ are, respectively, the attitude errors on the yaw, pitch and roll axes at time $t$. Our final KPI is thus the *Normalized Mean Attitude Module Error* at time $t$, defined as:

$$\mathrm{NMAME}(t) = \frac{1}{2\pi t}\int_0^t \mathrm{AME}(\tau)\mathrm{d}(\tau)$$

where $2\pi$ is a normalization factor, since a rotation along a given axis ranges in $[0, 2\pi]$.

Similarly to the AT and the FCS models, the goal of our experiment is to verify whether $\mathbb{E}(\mathrm{NMAME}(h)) \le \vartheta$ for some threshold. We set $h = 60$ s and $\vartheta = 0.5\%$. Note that greater values for $\vartheta$ systematically lead to system mission failures.

### 6.2. Implementation

We implemented our production software using Python and Cython, and exploited the Message Passing Interface (MPI) standard (Gropp et al., 2014) to allow communication between processes.

The overall architecture has been explicitly designed to be run on a HPC infrastructure of networked computational nodes, and reflects that of Fig. 2 with some caveats to boost performance. The SAs and APSG modules of Fig. 2 have been implemented as concurrent threads of a process named HPC-EAA, while the $n$ simulators have been deployed as independent processes on $n$ additional computational nodes. HPC-EAA runs various parallel threads: one devoted to the advancement of each SA, one devoted to the generation of random scenarios and their delegation to an available simulator taken from the queue, and one devoted to process the receiving queue of samples and to populate map *samples*. The use of such a thread-based implementation for the SAs and APSG modules limits the use of the network to the communication with the simulators.

### 6.3. Experimental setting

Here we describe how we conducted a thorough and statistically sound analysis of the performance and scalability of our parallel tool.

For each case study we computed $(\varepsilon, \delta)$-approximations of the KPIs of interest for 9 different values for $\varepsilon$ (ranging from $1\times10^{-3}$ to $1\times10^{-1}$), 3 values of $\delta$ ($1\times10^{-2}$, $5\times10^{-2}$, $1\times10^{-1}$), and 7 values for $n$ (1, 64, 128, 256, 512, 1024, 2048). Each experiment has been repeated 10 times, using 10 sequences of samples, each one produced by a different sequence of iid scenarios.

Conducting such an analysis using our production tool of Section 6.2 would be a *practically unviable* option, as it would require dozens of years (see Fig. 5) and incur in huge costs for the full allocation of a large-enough private HPC infrastructure. Indeed, for a proper analysis to be carried out, the production tool would need to be run 10 times for each case study and for each value of $\varepsilon$, $\delta$ and $n$ (including $n = 1$). Also, each execution of the tool would require the full availability of $n + 4$ computational nodes (CPU cores), including $n + 4 = 2048 + 4 = 2052$.

This obstruction is becoming common when analyzing performance and scalability of modern software explicitly designed to perform intensive computations on large HPC infrastructures, and is being handled in the literature from the methodological point of view. Along these lines, and by following recent approaches such as, *e.g.*, Esposito et al. (2024), we operated in an *indirect* way as follows.

First, we avoided the use of an expensive private HPC infrastructure, by exploiting a *shared* HPC cluster at Sapienza University, which is managed by a centralized scheduler which orchestrates the launch of the computational jobs submitted concurrently by the various research teams. Although this was a mandatory choice, it created some issues, as jobs requesting too many computational nodes to be allocated would have received a very low priority and could have remained in the waiting queue forever.

Hence, as in Esposito et al. (2024), we *disassembled* our tool into its single components, instrumented each component with proper timers within the code, and ran each of them *asynchronously*. More precisely:

1. We generated 10 random seeds for each case study and computed 10 sequences of scenarios long enough to allow all SAs to terminate for all values of $\varepsilon$ and $\delta$. We saved these sequences of scenarios into a centralized database, together with their ids and generation times (which are all on the order of a few microseconds).
2. We submitted to the HPC scheduler the simulation jobs for all scenarios independently, and measured the time taken by each of them to simulate the selected scenario and to compute the corresponding value of the system KPI (we ignored the simulator set-up time, which would not have had to be paid for each sample if we had instead used the production tool). For each scenario, the simulation time and the KPI value were also saved in the centralized database. Note that, as expected, simulation is by far the slowest step in the whole workload, with average times being orders of magnitude higher than those of the others, namely: 0.1377 s, 0.2279 s, 7.1654 s for AT, FCS and ALMA, respectively.
3. For each case study, the average half turnaround network time for each message type from HPC-EAA to a simulator and vice versa was measured independently, by running the real (MPI-based) application over 1 million scenarios. Thanks to a low-latency high-throughput local network, these times are all on the order of a hundred of microseconds (*i.e.*, around $1\times10^{-4}$ s), hence several orders of magnitude lower than simulation times.

We asked the HPC scheduler to run all the computations on nodes of identical machines, each one equipped with 2 AMD EPYC 7301 CPUs with 32 cores and 256 GB RAM, and hyper-threading disabled. This ensured that the measured time durations can be seamlessly combined together.

With the above data, we were able to virtually exercise our tool on 10 random experiments for each case study, assessing convergence of the SAs and overall performance on each setting ($\varepsilon$, $\delta$, and $n$). Indeed, the availability of the generation and simulation times of each scenario as well as of the average half turnaround time of network messages of each type allowed us to perform a reliable estimation of the execution time that our production tool would have shown, under each setting, if run over fully reserved portions of the HPC infrastructure at hand. Namely, for each experiment, the SAs and APSG modules were run normally (as a multi-threaded process on a single node), although properly instrumented, while the set of simulators were replaced by the database containing the pre-computed samples. A discrete event-based emulator module running on the same node (see Esposito et al., 2024) orchestrated the execution of the whole experiment in a faster-than-real-time way, by mimicking the advancement of time via a global counter accumulating the time durations of all steps of Algorithms 2 and 3 (*i.e.*, scenario generation, blocking dequeue operations from the queue of available simulators, network communication to and from the simulators, blocking look-up operations and insertion/deletion of entries to/from the *samples* map and advancement of all SAs). The time durations of the steps performed normally were measured via the timers instrumenting the code, while those of the emulated steps were taken from the database.

We note that, although we did our best to take into account, in our estimation, also the steps which are several orders of magnitude faster than the dominating tasks (system model simulations), it is known that disassembling and instrumenting the production code can introduce unnoticed small divergences between the estimated time and the time that our tool would have shown if run normally. However, such potential small divergences would actually be *unmeasurable and not repeatable* in practice. This is due to how modern large HPC infrastructures (even those fully reserved for a single job) are managed by complex orchestrating and probing services, and by software virtualization and containerization layers, which may introduce a significant amount of noise.

In other words, the estimations below, as all estimations of the completion time of complex massively parallel and computationally intensive software designed for large HPC infrastructures, actually estimate what would happen if the software would run on *bare metal*.

### 6.4. Results

Here we present our experimental results. Namely, in Section 6.4.1 we quantify the benefit of using EAA in terms of reduction in the number of samples required to convergence, whilst in Section 6.4.2

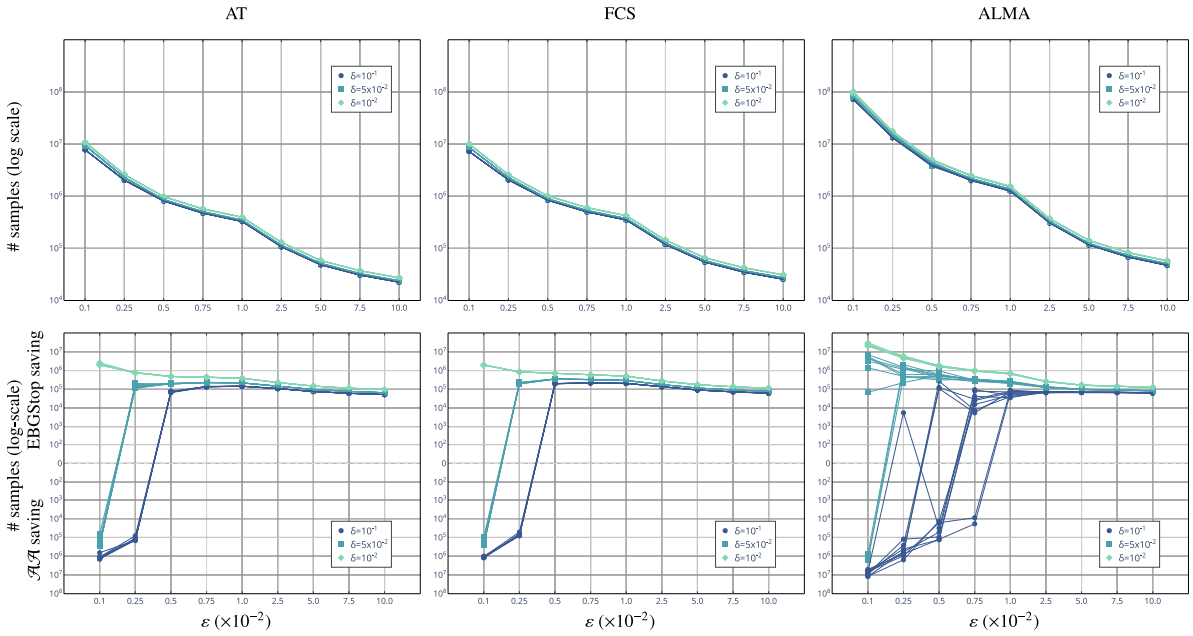


**Fig. 3.** Top: number of samples (vertical log-scale axis) required by EAA to converge in each of the 10 experiments, for various values of $\varepsilon$ and $\delta$. Bottom: savings in the number of samples achieved by EAA in each setting wrt. the algorithm requiring most samples.

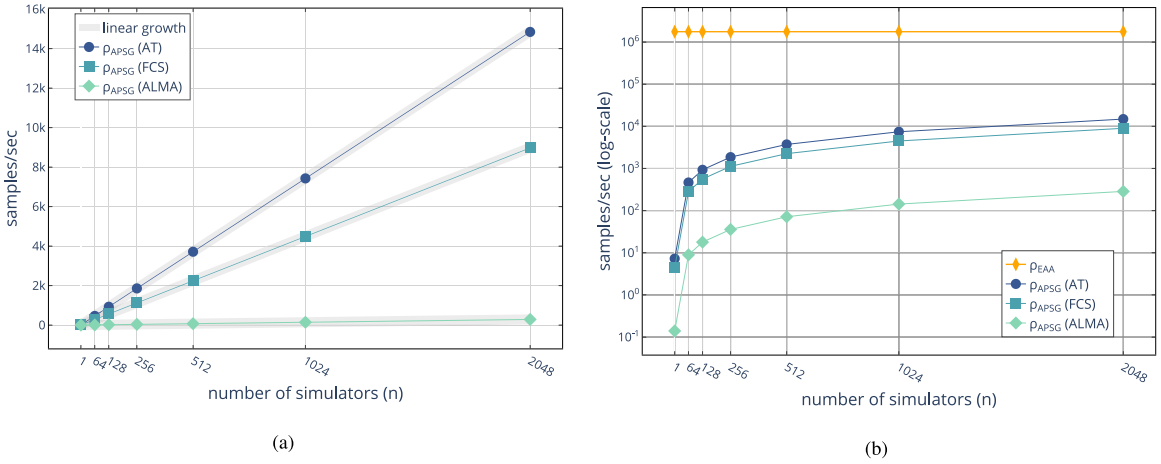


**Fig. 4.** (a) Rates of sample production by APSG ($\rho_{\text{APSG},n}$), for various numbers of available simulators (the grey curves are the *straight* lines passing by the two left-most points of each data series, and show the almost perfect linearity of the sample production rates wrt. $n$).
(b) Comparison (log-scale Y axis) among the sample consumption rate by EAA ($\rho_{\text{EAA}}$) and the sample production rates by APSG from Figure (a).

we exploit the methodology outlined in Section 6.3 to conduct a performance scalability analysis of our parallel tool.

### 6.4.1. Reduction of the number of required samples due to EAA

To evaluate the benefit of using an ensemble of SAs, for each case study we repeated our experiment 10 times. In each of them, we fed the SAs module with one of the 10 random sequences of iid samples computed in advance for that model, as described in Section 6.3, and computed the number of samples required by each of the two SAs of the ensemble to converge. The number of samples required by EAA is then the minimum of the two.

Fig. 3 (top) shows, for each case study and for each of the 10 random sequences of samples, the number of samples required by EAA to converge, for each value of $\varepsilon$ and $\delta$.

Similarly, Fig. 3 (bottom) shows the difference in the number of samples between the two SAs, which is also the saving achieved if using EAA, when compared to using the slowest algorithm only, together with information about which of the two SAs terminated first, and so was responsible for producing the sought $(\varepsilon, \delta)$-approximation. From the figure is clearly emerges how using EAA is a particularly effective way to keep the number of required samples to a minimum. Indeed, our experiments show that EAA may require up to 31 300 400 fewer samples when compared to using a single algorithm, with relative reductions as large as 78.73%.

Predicting which algorithm of the ensemble will terminate first is a difficult task *a priori*, since it depends not only on $\varepsilon$, $\delta$ and on statistical properties of the KPI of interest (with the latter being typically unknown in advance), but may also be influenced by the actual random sequence of iid samples (*i.e.*, on the seed of the pseudorandom number generator). The results for the ALMA case study (see Fig. 3, bottom) are a good illustration of that phenomenon. EAA also provides a stabilizing effect in this regard. Indeed, differently from what emerges in Fig. 3

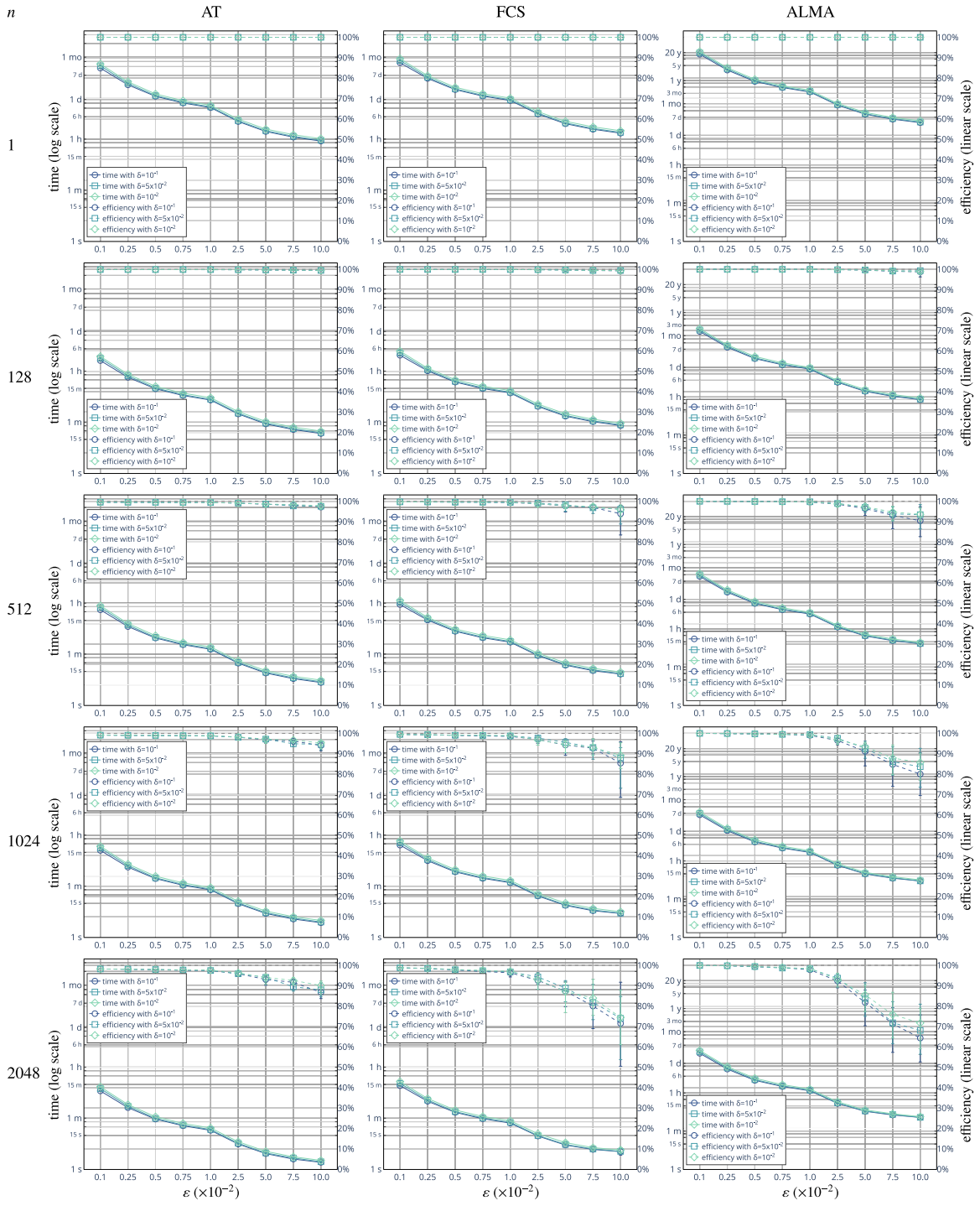


**Fig. 5.** Scalability analysis of our parallel tool: overall completion time (bottom curves, left Y log-scale axis) and efficiency (top curves, right Y axis) for each case study and selected values for the number of simulators $n$.

(bottom), the curves in Fig. 3 (top) regarding the same case study and the same value for $\delta$ (but generated using different random sequences of samples) overlap with each other almost perfectly.

### 6.4.2. Increase of the sample production rate due to the Asynchronous Parallel Sample Generator

Here, we analyze the impact of deploying APSG on a parallel HPC infrastructure. Namely, in Fig. 4(a) we show an estimation (with the method outlined in Section 6.3) of the maximum rate $\rho_{\text{APSG},n}$ of sample production when varying the number $n$ of available simulators. For each value of $n$, $\rho_{\text{APSG},n}$ has been obtained by replacing the SAs module with a simple loop requesting (and just throwing away) 1 million samples from APSG governing $n$ parallel simulators. Hence, these values are the highest sample production rates actually achievable by APSG on our infrastructure when using $n$ simulators of the given model. Fig. 4(a) shows that all $\rho_{\text{APSG},n}$ are practically linear with respect to $n$, with a maximum deviation between the actual and the ideal (linear) rates $< 0.1\%$.

In Fig. 4(b) we show again the three values for $\rho_{\text{APSG},n}$, this time using a logarithmic Y axis and along with the maximum rate $\rho_{\text{EAA}}$

of consumption of samples by the SAs module. Rate $\rho_{\text{EAA}}$ has been obtained by replacing APSG and the controlled simulators by a simple pseudorandom generator feeding the SAs module with numerical values in $[0, 1]$, and measuring how long it took to consume 1 million of such samples. This value is thus the lowest rate of samples that APSG would need to provide to the SAs module (running on our infrastructure) to allow the latter to advance at full speed.

From Fig. 4(b) it clearly emerges that $\rho_{\text{EAA}}$ is always several orders of magnitude higher than all $\rho_{\text{APSG},n}$ (even when the largest number of simulators, $n = 2048$, is deployed, in which case is still more than 100 times larger). This means that the SAs module would spend almost its entire time waiting for APSG to produce the requested samples. Hence, an increase in the number of simulators is expected to result in an essentially proportional reduction in the completion time of the overall parallel tool. In order words, we expect our architecture to show an efficiency very close to 100% even when a large number of simulators are used. This suggests that adding more algorithms to the ensemble of SAs would not introduce any noticeable computational overhead, but could help in reducing the number of required samples for EAA if those additional algorithms happen to benefit from the statistical properties of the system KPI.

#### 6.4.3. Overall performance scalability analysis

To confirm the above, here we describe a performance scalability analysis of our parallel implementation, estimating (again with the method outlined in Section 6.3) its completion time if deployed on fully-reserved HPC infrastructures of various sizes.

Fig. 5 shows the time that our tool would require to terminate on each of the 10 experiments for each case study, as a function of $\varepsilon$, $\delta$, and the number $n$ of parallel simulators (notice the left log-scale Y axes). Each figure shows, for each value of $\delta$, the average completion time of the parallelization, with error bars indicating the minimum and the maximum values among the 10 experiments (error bars are often very tiny and barely visible, and this shows again the stability of the algorithm performance with respect to the random sequence of samples). As expected, all the curves are qualitatively very similar, modulo the time reductions essentially proportional to $n$.

Each plot in Fig. 5 also shows (see the curves at the top, to be read against the linear-scaled right Y axes) the average estimated efficiency of the parallelization (with error bars indicating the minimum and the maximum values among the 10 experiments). The efficiency of the parallel computation involving $n$ simulators is computed, as usual, as $\frac{t_1}{n \times t_n}$, where, in our case, $t_n$ is the (estimated) time of the computation when using $n$ simulators and $t_1$ is the time of *same* computation (*i.e.*, same values for $\varepsilon$, $\delta$ and same random sequence of samples) when using a single simulator.

As expected, the estimated efficiency always sticks to $\sim$ 100%, except when the number of simulators (hence, the size of the envisioned HPC infrastructure) is too large for the problem at hand, in which case a significant number of samples is produced (in parallel) by APSG, but never used by EAA. Although such circumstances are a clear symptom of an over-dimensioning of the hardware infrastructure, they do not pose a usability problem for the tool, since in this case the overall completion time is in the range of a few seconds or minutes.

## 7. Conclusions and perspectives

In this article, we focused on the simulation-based verification of complex systems with Statistical Model Checking (SMC), by highlighting how Adaptive Stopping Algorithms (SAs) form a class of approximation algorithms well-suited for parallelization. Not only their design makes the parallelization of the generation of the system simulations (samples) possible, but it also allows them to be combined without any interference or complication.

We proposed Ensemble of Approximation Algorithms (EAA), a composite algorithm combining the respective strengths of multiple SAs. We successfully implemented and tested EAA exploiting two (quasi) optimal algorithms: EBGStop and $\mathcal{AA}$, together with a massively parallel architecture deployable over large High-Performance Computing (HPC) infrastructures.

We exercised our tool on three industry-scale case studies, verifying specifications of non-trivial properties, and showed that parallelization can indeed make SMC a viable approach for the verification of complex systems, even when the number of required simulations can appear prohibitive from a sequential perspective. We exposed an in-depth experimental analysis of our tool, detailing the benefits of combining multiple SAs, quantifying the effective reductions in sample sizes and execution times, and discussing the efficiency of the parallelization and the impact of the number of parallel simulators. All in all, our work makes it clear that parallelization should be considered as systematically as possible to increase and widen the applicability of SMC.

A direct extension of EAA would obviously be one which integrates other SAs, potentially making it even more versatile and applicable to a larger class of KPIs.

To go further, our tool could be expanded to allow for the verification of formula-based properties, expressed for instance in linear temporal logic. In its simplest form, this would be effortless, since the probability than a chosen property holds for a random possible execution of a system can be seen as a KPI of that system. A more ambitious plan would be to explore whether a strategy such as the one behind the elaboration of EAA could help for the verification of hyper-properties (Arora et al., 2022; Dobe et al., 2023).

Repeating the analysis of the impact of parallelization on the effectiveness and applicability of SMC with a composite algorithm combining multiple optimization algorithms rather than simple estimation algorithms would be interesting. Identifying a class of optimization algorithms for SMC which share some of the key properties of stopping algorithms that make their parallelization relatively straightforward would be a first step.

Our tool could then also be enhanced even further by improving the sampling process itself, similarly to what has been done in Parmentier et al. (2024), especially when taking into account geometric sampling.

## CRediT authorship contribution statement

**Leonardo Picchiami:** Writing – original draft, Writing – review & editing, Visualization, Software, Methodology, Investigation, Formal analysis, Data curation, Conceptualization. **Maxime Parmentier:** Writing – original draft, Writing – review & editing, Methodology, Investigation, Formal analysis, Data curation, Conceptualization. **Axel Legay:** Writing – review & editing, Validation, Supervision, Resources, Project administration, Methodology, Investigation, Funding acquisition, Formal analysis, Conceptualization. **Toni Mancini:** Writing – review & editing, Validation, Supervision, Software, Resources, Project administration, Methodology, Funding acquisition, Investigation, Formal analysis, Conceptualization. **Enrico Tronci:** Validation, Supervision, Project administration, Methodology, Funding acquisition, Formal analysis, Conceptualization.

## Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

## Data availability

Data is available at https://doi.org/10.5281/zenodo.13969757 (DOI:10.5281/z

## Acknowledgments

This work was partially supported by: INdAM "GNCS Project 2023", Italy; Sapienza University projects, Italy RG12117A8B393BDC, RM120172B9F35634, RG123188B482D2D9; Lazio POR FESR projects, Italy E84G20000150006, F83G17000830007; Project funded under the National Recovery and Resilience Plan (NRRP), Mission 4 Component 2 Investment 1.5 - Call for tender No. 3277 of 30 December 2021 of the Italian Ministry of University and Research funded by the European Union – NextGenerationEU, award number: project ECS 0000024 Rome Technopole, Concession Decree No. 1051 of 23 June 2022 adopted by the Italian Ministry of University and Research, CUP B83C22002820006, Rome Technopole.

M. Parmentier is funded by a FNRS PhD Grant, Belgium and by the UCLouvain, Belgium. A. Legay is funded by a FNRS PDR, Belgium - T013721.

## References

Abbas, H., Fainekos, G., Sankaranarayanan, S., Ivančić, F., Gupta, A., 2013. Probabilistic temporal logic falsification of cyber-physical systems. ACM Transactions on Embedded Computing Systems (ISSN: 1539-9087) 12 (2s), 95:1–95:30. http://dx.doi.org/10.1145/2465787.2465797.

Agha, G., Palmskog, K., 2018. A survey of statistical model checking. ACM Transactions on Modeling and Computer Simulation (ISSN: 1049-3301) 28 (1), 6:1–6:39. http://dx.doi.org/10.1145/3158668.

AlTurki, M., Meseguer, J., 2011. PVeStA: a parallel statistical model checking and quantitative analysis tool. In: 4th International Conference on Algebra and Coalgebra in Computer Science (CALCO 2011). In: Lecture Notes in Computer Science, 6859, Springer, pp. 386–392.

Arora, S., Hansen, R.R., Larsen, K.G., Legay, A., Poulsen, D.B., 2022. Statistical model checking for probabilistic hyperproperties of real-valued signals. In: 28th International Symposium on Model Checking Software (SPIN 2022). In: Lecture Notes in Computer Science, 13255, Springer, pp. 61–78.

Audibert, J.-Y., Munos, R., Szepesvári, C., 2007a. Tuning bandit algorithms in stochastic environments. In: 18th International Conference on Algorithmic Learning Theory (ALT 2007). Springer, pp. 150–165.

Audibert, J.-Y., Munos, R., Szepesvári, C., 2007b. Variance estimates and exploration function in multi-armed bandit. In: CERTIS Research Report 07–31. CERTIS.

Baranov, E., Bowles, J., Given-Wilson, T., Legay, A., Webber, T., 2022. A secure user-centred healthcare system: design and verification. In: 10th International Symposium From Data to Models and Back (DataMod 2021). In: Lecture Notes in Computer Science, 13268, Springer, pp. 44–60.

Barbot, B., Basset, N., Dang, T., Donzé, A., Kapinski, J., Yamaguchi, T., 2020. Falsification of cyber-physical systems with constrained signal spaces. In: NASA Formal Methods: 12th International Symposium (NFM 2020). Springer, pp. 420–439.

Barbot, B., Haddad, S., Picaronny, C., 2012. Coupling and importance sampling for statistical model checking. In: 18th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2012). In: Lecture Notes in Computer Science, 7214, Springer, pp. 331–346.

Basile, D., ter Beek, M.H., Ferrari, A., Legay, A., 2022. Exploring the ERTMS/ETCS full moving block specification: an experience with formal methods. International Journal on Software Tools for Technology Transfer 24 (3), 351–370.

Boyer, B., Corre, K., Legay, A., Sedwards, S., 2013. PLASMA-lab: a flexible, distributable statistical model checking library. In: 10th International Conference on Quantitative Evaluation of Systems (QEST 2013). Springer, pp. 160–164.

Bradley, J.K., Schapire, R.E., 2007. FilterBoost: regression and classification on large datasets. In: Advances in Neural Information Processing Systems 20 (NIPS 2007). pp. 185—192.

Bulychev, P., David, A., Larsen, K.G., Mikučionis, M., Legay, A., 2011. Distributed parametric and statistical model checking. In: 10th International Workshop on Parallel and Distributed Methods in Verification (PDMC 2011). In: Electronic Proceedings in Theoretical Computer Science, 72, pp. 30–42.

Claessen, K., Smallbone, N., Eddeland, J., Ramezani, Z., Åkesson, K., 2018. Using valued Booleans to find simpler counterexamples in random testing of cyber-physical systems. IFAC-PapersOnLine 51 (7), 408–415.

Clarke, E.M., Henzinger, T.A., Veith, H., 2016. Handbook of model checking. Springer, ISBN: 9783319105741.

Clarke, E.M., Zuliani, P., 2011. Statistical model checking for cyber-physical systems. In: 9th International Symposium on Automated Technology for Verification and Analysis (ATVA 2011). 11, Springer, pp. 1–12.

Dagum, P., Karp, R.M., Luby, M., Ross, S.M., 2000. An optimal algorithm for Monte Carlo estimation. SIAM Journal on Computing 29 (5), 1484–1496. http://dx.doi.org/10.1137/S0097539797315306.

D'Argenio, P., Legay, A., Sedwards, S., Traonouez, L.-M., 2015. Smart sampling for lightweight verification of Markov decision processes. International Journal on Software Tools for Technology Transfer 17, 469–484.

Dobe, O., Schupp, S., Bartocci, E., Bonakdarpour, B., Legay, A., Pajic, M., Wang, Y., 2023. Lightweight verification of hyperproperties. In: 21st International Symposium on Automated Technology for Verification and Analysis (ATVA 2023). Springer, pp. 3–25.

Dokhanchi, A., Zutshi, A., Sriniva, R.T., Sankaranarayanan, S., Fainekos, G., 2015. Requirements driven falsification with coverage metrics. In: 15th International Conference on Embedded Software (EMSOFT 2015). IEEE, pp. 31–40. http://dx.doi.org/10.1109/EMSOFT.2015.7318257.

Domingo, C., Gavalda, R., Watanabe, O., 2002. Adaptive sampling methods for scaling up knowledge discovery algorithms. Data Mining and Knowledge Discovery 6, 131–152.

Domingo, C., Watanabe, O., 2000. MadaBoost: a modification of AdaBoost. In: 13th Annual Conference on Computational Learning Theory (COLT 2000). Morgan Kaufmann, pp. 180–189.

Dong, X., Yu, Z., Cao, W., Shi, Y., Ma, Q., 2020. A survey on ensemble learning. Frontiers of Computer Science 14, 241–258.

Ernst, G., Arcaini, P., Fainekos, G., Formica, F., Inoue, J., Khandait, T., Mahboob, M.M., Menghi, C., Pedrielli, G., Waga, M., 2022. Arch-comp 2022 category report: falsification with ubounded resources. In: 9th International Workshop on Applied veRification for Continuous and Hybrid Systems (ARCH 2022). In: EasyChair Proceedings in Computing (EPiC), 90, pp. 204–221.

Esposito, M., Mancini, T., Tronci, E., 2024. Optimizing fault-tolerant quality-guaranteed sensor deployments for UAV localization in critical areas via computational geometry. IEEE Transactions on Systems, Man and Cybernetics: Systems 54 (3), 1515–1526. http://dx.doi.org/10.1109/TSMC.2023.3327432.

Esposito, M., Picchiami, L., 2022a. Estimation based verification of cyber-physical systems via statistical model checking. In: 1st International Workshop on HYbrid Models for Coupling Deductive and Inductive ReAsoning (HYDRA 2022) and the 29th RCRA Workshop on Experimental Evaluation of Algorithms for Solving Problems with Combinatorial Explosion (RCRA 2022). In: CEUR Workshop Proceedings, 3281, CEUR-WS.org, pp. 51–63.

Esposito, M., Picchiami, L., 2022b. Formal certification of surrogate models for cyber-physical systems verification. In: 4th Workshop on Artificial Intelligence and Formal Verification, Logic, Automata, and Synthesis (OVERLAY 2022). In: CEUR Workshop Proceedings, 3311, CEUR-WS.org, pp. 63–71.

Even-Dar, E., Mannor, S., Mansour, Y., 2002. PAC bounds for multi-armed bandit and Markov decision processes. In: 15th Annual Conference on Computational Learning Theory (COLT 2002). Springer, pp. 255–270.

Fan, C., Qi, B., Mitra, S., Viswanathan, M., 2017. DryVR: data-driven verification and compositional reasoning for automotive systems. In: 29th International Conference on Computer Aided Verification (CAV 2017). In: Lecture Notes in Computer Science, 10426, Springer, pp. 441–461. http://dx.doi.org/10.1007/978-3-319-63387-9_22.

Gropp, W., Lusk, E., Skjellum, A., 2014. Using MPI, third edition: portable parallel programming with the Message-Passing Interface. Scientific and Engineering Computation, MIT Press, ISBN: 9780262527392.

Hoeffding, W., 1994. Probability inequalities for sums of bounded random variables. In: The Collected Works of Wassily Hoeffding. Springer, pp. 409–426.

Hoxha, B., Abbas, H., Fainekos, G., 2014. Benchmarks for temporal logic requirements for automotive systems. In: 1st International Workshop on Applied veRification for Continuous and Hybrid Systems (ARCH@CPSWeek 2014). In: EasyChair Proceedings in Computing (EPiC), 34, pp. 25–30.

Hoxha, B., Dokhanchi, A., Fainekos, G., 2018. Mining parametric temporal logic properties in model-based design for cyber-physical systems. International Journal on Software Tools for Technology Transfer 20, 79–93.

Jegourel, C., Legay, A., Sedwards, S., 2012. Cross-entropy optimisation of importance sampling parameters for statistical model checking. In: 24th International Conference on Computer Aided Verification (CAV 2012). In: Lecture Notes in Computer Science, 7358, Springer, pp. 327–342.

Larsen, K.G., Legay, A., 2016. Statistical model checking: past, present, and future. In: 7th International Symposium on Leveraging Applications of Formal Methods, Verification and Validation (ISoLA 2016). In: Lecture Notes in Computer Science, 9952, Springer, pp. 3–15.

Larsen, K., Legay, A., Nolte, G., Schlüter, M., Stoelinga, M., Steffen, B., 2022. Formal methods meet machine learning (F3ML). In: 11th International Symposium on Leveraging Applications of Formal Methods, Verification and Validation (ISoLA 2022). In: Lecture Notes in Computer Science, 13701, Springer, pp. 393–405.

Mancini, T., Mari, F., Massini, A., Melatti, I., Salvo, I., Tronci, E., 2017. On minimising the maximum expected verification time. Information Processing Letters (ISSN: 0020-0190) 122, 8–16. http://dx.doi.org/10.1016/j.ipl.2017.02.001.

Mancini, T., Mari, F., Massini, A., Melatti, I., Tronci, E., 2014. Anytime system level verification via random exhaustive hardware in the loop simulation. In: 17th Euromicro Conference on Digital System Design (DSD 2014). IEEE, pp. 236–245. http://dx.doi.org/10.1109/DSD.2014.91.

Mancini, T., Mari, F., Massini, A., Melatti, I., Tronci, E., 2016a. Anytime system level verification via parallel random exhaustive hardware in the loop simulation. Microprocessors and Microsystems (ISSN: 0141-9331) 41, 12–28. http://dx.doi.org/10.1016/j.micpro.2015.10.010.

Mancini, T., Mari, F., Massini, A., Melatti, I., Tronci, E., 2016b. SyLVaaS: system level formal verification as a service. Fundamenta Informaticae 149 (1–2), 101–132. http://dx.doi.org/10.3233/FI-2016-1444.

Mancini, T., Mari, F., Massini, A., Melatti, I., Tronci, E., 2021. On checking equivalence of simulation scripts. Journal of Logical and Algebraic Methods in Programming (ISSN: 2352-2208) 120, 100640. http://dx.doi.org/10.1016/j.jlamp.2021.100640.

Mancini, T., Melatti, I., Tronci, E., 2022. Any-horizon uniform random sampling and enumeration of constrained scenarios for simulation-based formal verification. IEEE Transactions on Software Engineering (ISSN: 0098-5589) 48 (10), 4002–4013. http://dx.doi.org/10.1109/TSE.2021.3109842.

Mancini, T., Melatti, I., Tronci, E., 2023. Optimising highly-parallel simulation-based verification of cyber-physical systems. IEEE Transactions on Software Engineering (ISSN: 0098-5589) http://dx.doi.org/10.1109/TSE.2023.3298432.

Mathworks, 2024a. Developing the Apollo lunar module digital autopilot. https://mathworks.com/help/simulink/slref/developing-the-apollo-lunar-module-digital-autopilot.html. (Accessed 20 October 2024).

Mathworks, 2024b. Model an automatic transmission controller. https://mathworks.com/help/simulink/slref/modeling-an-automatic-transmission-controller.html. (Accessed 20 October 2024).

Mathworks, 2024c. Modeling a fault-tolerant fuel control system. https://mathworks.com/help/simulink/slref/modeling-a-fault-tolerant-fuel-control-system.html. (Accessed 20 October 2024).

Mnih, V., Szepesvári, C., Audibert, J.Y., 2008. Empirical Bernstein stopping. In: 25th International Conference on Machine Learning (ICML 2008). ACM, ISBN: 9781605582054, pp. 672—679. http://dx.doi.org/10.1145/1390156.1390241.

Pappagallo, A., Massini, A., Tronci, E., 2020. Monte Carlo based statistical model checking of cyber-physical systems: a review. Information 11 (12), 588. http://dx.doi.org/10.3390/info11120588.

Parmentier, M., Legay, A., Chenoy, F., 2024. Optimized smart sampling. In: 1st International Conference on Bridging the Gap between AI and Reality (AISoLA 2023). In: Lecture Notes in Computer Science, 14380, Springer, pp. 171–187.

Sagi, O., Rokach, L., 2018. Ensemble learning: a survey. Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery 8 (4), e1249.

Singh, A.S., Masuku, M.B., 2014. Sampling techniques & determination of sample size in applied statistics research: an overview. International Journal of Economics, Commerce and Management 2 (11), 1–22.

Sontag, E.D., 1998. Mathematical control theory: deterministic finite dimensional systems (2nd ed.). Springer, ISBN: 0-387-984895.

Zuliani, P., Platzer, A., Clarke, E.M., 2013. Bayesian statistical model checking with application to Stateflow/Simulink verification. Formal Methods in System Design (ISSN: 1572-8102) 43 (2), 338–367. http://dx.doi.org/10.1007/s10703-013-0195-3.

**Leonardo Picchiami** received his M.Sc. degree in Computer Science from Sapienza University of Rome, Italy, in 2021. He is currently a Ph.D. student in Computer Science at Sapienza University. His research interests include formal verification, cyber–physical systems, artificial intelligence, and systems biology.

**Maxime Parmentier** received his M.Sc. degree in Mathematical Science and his Certificate of Philosophy from UCLouvain, Belgium, respectively in 2016 and 2017. He is currently a Ph.D. student in Computer Science at UCLouvain. His research interests include formal verification, statistical model checking, stopping algorithms, cyber–physical systems, artificial intelligence, and machine learning.

**Axel Legay** received his M.Sc. degree in Computer Science from University of Liège, Belgium, in 2003, and his Ph.D. degree in Computer Science from University of Liège, Belgium, in 2007. He is currently a Full Professor at the Institute of Information and Communication Technologies, Electronics and Applied Mathematics of UCLouvain, Belgium. His research interests include cybersecurity, malware analysis, software vulnerability analysis, threat detection, IoT, smart cities, security tests, formal verification and statistics.

**Toni Mancini** received his Ph.D. degree in Computer Engineering from Sapienza University of Rome, Italy, in 2005. He is currently an Associate Professor at the Department of Computer Science of Sapienza University of Rome, Italy. His research interests include artificial intelligence, formal verification, cyber–physical systems, control software synthesis, systems biology, and autonomous demand and response systems for smart grids.

**Enrico Tronci** received his M.Sc. degree in Electrical Engineering from Sapienza University of Rome, Italy, in 1987, and his Ph.D. degree in Applied Mathematics from Carnegie Mellon University, Pittsburgh, PA, USA, in 1991. He is currently a Full Professor at the Department of Computer Science of Sapienza University of Rome, Italy. His research interests include formal verification, model checking, cyber–physical systems, control software synthesis, smart grids, autonomous demand and response systems for smart grids, and systems biology.