Anytime System Level Verification via Random Exhaustive Hardware In The Loop Simulation

Toni Mancini   Federico Mari   Annalisa Massini   Igor Melatti   Enrico Tronci
Computer Science Department, Sapienza University of Rome, Italy
Email: {tmancini, mari, massini, melatti, tronci}@di.uniroma1.it

***Abstract*—We present a *parallel random exhaustive* Hardware In the Loop Simulation based model checker for hybrid systems that, by simulating *all* operational scenarios *exactly once* in a *uniform random* order, is able to provide, at *any time* during the verification process, an *upper bound* to the probability that the System Under Verification exhibits an error in a yet-to-be-simulated scenario (Omission Probability).**

We show effectiveness of the proposed approach by presenting experimental results on System Level Formal Verification of the Fuel Control System example in the Simulink distribution. To the best of our knowledge, no previously published model checker can *exhaustively* verify hybrid systems of such a size *and* provide at *any time* an upper bound to the Omission Probability.

I. Introduction

Cyber-Physical Systems (CPSs) consists of hardware and software components and can be modelled as hybrid systems (see, e.g., [4] and citations thereof). System Level Verification of CPSs has the goal of verifying that the *whole* (i.e., software + hardware) system meets the given specifications. Model checkers for hybrid systems cannot handle System Level Formal Verification (SLFV) of actual CPSs. Thus, Hardware In the Loop Simulation (HILS) is currently the main workhorse for system level verification and is supported by *Model Based Design* tools like Simulink (http://www.mathworks.com) and VisSim (http://www.vissim.com). In HILS, the *actual software* reads/sends values from/to mathematical models (*simulation*) of the physical systems (e.g., engines, analog circuits, etc.) it will be interacting with.

A. Motivations

SLFV is an exhaustive HILS where *all* relevant simulation scenarios are considered. Using a parallel model checking driven approach, exhaustive HILS enables formal verification of actual systems such as the Fuel Control System (FCS) in the Simulink distribution (e.g., see [1]).

Considering that parallel exhaustive HILS based SLFV may take days of computation (e.g., see [1]), from a practical point of view it would be very useful to have available at *any time* during the verification process, *quantitative* information about the degree of assurance attained. Such an information would enable us to evaluate if it is worth to continue the verification activity, or instead stop it since the degree of assurance attained can be considered adequate for the application at hand (*graceful degradation*).

The above considerations suggest looking for a HILS based model checking approach satisfying the following requirements: (i) it is *parallel*, in order to make exhaustive HILS computationally feasible; (ii) it is *exhaustive*, since our focus is SLFV; (iii) it is *any time*, to support *graceful degradation*.

The work in [2] presents a Propositional Satisfiability (SAT) based model checker for finite state systems which returns, at *any time* during the verification process, the *coverage* (i.e., the fraction of operational scenarios verified so far). Unfortunately, while coverage measures the *amount* of verification work done, it does not provide any information about the *degree of assurance* attained by the verification process. This is because formal verification aims at finding *hard to find* errors, i.e., errors that were not detected verifying operational scenarios designed by experts. As a result, formal verification addresses errors that we are *unlikely* to consider, and we need to adopt an adversarial model where, knowing our verification strategy, the adversary places the error in operational scenarios we are less likely to visit. In such a framework, *any* deterministic ordering of the operational scenarios would not increase the degree of assurance until the end of the verification (as the adversary would place the single error as the *last* scenario picked by the verification procedure).

To provide a formally sound information about the degree of assurance attained by the verification process, approaches have been proposed which verify the operational scenarios in a *random* order. In particular, the work in [3] presents a Monte-Carlo based model checker for finite state systems that provides, at *any time* during the verification process, an upper bound to the probability that the System Under Verification (SUV) exhibits an error in a yet-to-be-simulated scenario (Omission Probability). The Omission Probability (OP) provides indeed the information we are looking for. Unfortunately, while Monte-Carlo based approaches guarantee randomness (thereby enabling OP computation) they are not exhaustive (within a finite time).

To the best of our knowledge, no model checker is available, neither for finite state systems nor for hybrid systems, which, at the same time, is both *random* and *exhaustive*, thereby enabling effective *anytime* SLFV. In this paper we advance the state of the art by presenting a *parallel random exhaustive* HILS based model checker along with experimental results showing its effectiveness.

B. Main Contribution

Our SUV is a *Hybrid System* (e.g., see [4] and citations thereof) whose inputs belong to a finite set of uncontrollable events (*disturbances*) modelling failures in sensors or actuators, variations in the system parameters, etc. We focus on *deterministic systems* (the typical case for control systems) and model nondeterministic behaviours (such as faults) with disturbances. Accordingly, in our framework, a *simulation* *scenario* is just a finite sequence of disturbances and a *simulation campaign* is a finite sequence of simulation instructions (namely: *save* a simulation state, *restore* a saved simulation state, *remove* a saved simulation state, *inject* a disturbance, *advance* the simulation of a given time length).

A system is expected to *withstand* all disturbance sequences that may arise in its operational environment. Correctness of a system is thus defined with respect to such *admissible* disturbance sequences. In our setting, the set of admissible disturbance sequences (*disturbance model*) can be defined as the language accepted by a suitable Finite State Automaton, which in turn can be defined using the modelling language of any finite state model checker.

In such a framework we address *Bounded* SLFV of *safety* properties. That is, given a time step τ (time quantum between disturbances) and a time horizon $T = \tau h$ we return *PASS* if there is no *admissible* disturbance sequence of length h and time step τ that violates the given safety property. We return *FAIL*, along with a counterexample, otherwise. Therefore, SLFV is an *exhaustive* (with respect to admissible disturbance sequences) HILS. In other words, we are aiming at (*black box*) *bounded model checking* where the SUV behaviour is defined by a simulator (Simulink in our examples).

In such a setting, our main contributions can be summarised as follows. We present an *anytime parallel random exhaustive* HILS based model checker that effectively conjugates exhaustiveness with randomness, thereby enabling OP computation. By observing that, in our setting, simulation of operational scenarios takes more than 98% of the verification time (see Fig. 6) we proceed as follows.

First, from the disturbance model we generate all admissible simulation scenarios and evenly split them into disjoint sets (*slices*). We do this along the lines of [1].

Second, for each slice, we compute a highly optimised *simulation campaign* that exploits simulator save/restore/remove commands in order to save on the simulation time while scheduling execution of *all* simulation scenarios *exactly once* and in a *uniform random order*. This guarantees exhaustiveness and allows us to compute, at *any time* during the verification process, an upper bound to the OP. We note that this step is made possible by the fact that we have first generated all admissible simulation scenarios.

Third, we run simulation campaigns in parallel. This guarantees a very efficient parallelism, since no communication among processes is needed. This step is supported by simulation tools (Simulink in our examples).

Devising an effective simulation campaign optimisation algorithm (the second step above) is the main obstacle to overcome in our setting. Note in fact that, when disturbance sequences are simulated in a lexicographic order (as in [1]), sequences with a common prefix tend to be close to each other in the simulation order. This in turn enables saving of the simulation state at the end of the common prefix, and restoring of such a state when a new scenario, with the same prefix, is presented. When simulating disturbance sequences in a random order, sequences with a common prefix may be quite far apart in the simulation order. Thus, unless we store far too many simulation states (impossible, since each simulation state can easily take hundreds of KBytes), it becomes hard to take advantage of common prefixes of disturbance sequences.

C. Experimental Results

We implemented our approach and present (Section V) experimental results on the Fuel Control System (FCS) example in the Simulink distribution. SLFV for this case study entails running more than *4 million simulation scenarios*.

Each processor (actually, a core of a 8-core machine) runs an instance of our optimisation algorithm and takes as input a slice of our set of simulation scenarios. We present experimental results with $16, 32, 64$ machines totalling $128, 256, 512$ parallel processes. Our optimiser takes just a few minutes to generate a simulation campaign for a given slice (see Section V-B). Simulation scenario generation and optimisation takes less than 2% of the total verification time.

Our experimental results show that, by exploiting parallelism, our simulation campaign optimisation algorithm effectively counteracts the simulation time overhead due to the random exhaustive simulation order. For example, with 128 cores our simulation time overhead is about 247% with respect to the deterministic simulation campaign computed by the optimiser in [1], whereas, when using 512 cores, such an overhead becomes less than 10%. The above ensures feasibility of our parallel random exhaustive approach for actual systems, such as the Fuel Control System (FCS) example in the Simulink distribution.

As for the OP, the worst case scenario is when just one error trace (out of 4 million in our case study) is present. Our experimental results (Section V-D) show that even in such a case our upper bound to the OP decreases about *linearly* with respect to the coverage (i.e., the fraction of scenarios simulated). This is the *best* one can hope for in our setting.

Finally, simulation of scenarios in random order allows us to use the coverage as a reliable estimator for the *completion time* of the whole verification task. Our experimental results show that, when the coverage is greater than 30%, the error in the completion time estimation is less than 20% (Section V-E).

D. Paper Outline

Section II gives background notions to make our paper self-contained. Section III presents our formal framework, by formalising the notion of OP and by providing an upper bound for it, computable from the number of the yet-to-be-simulated traces in each slice. Section IV outlines our algorithm to compute, from a sequence (slice) of disturbance traces, a highly optimised simulation campaign which simulates the input traces in uniform random order and exploits the save /restore/remove capabilities of the simulator. Finally, Section V presents experimental results assessing effectiveness of our approach.

II. Background

In this section we give some background notions. Unless otherwise stated, all definitions are based on [1], [5]. Throughout the paper, we use $\mathbb{R}^{\geq 0}$ for the set of non-negative reals, $\mathbb{R}^{+}$ for the set of strictly positive reals, and Bool $= \{0, 1\}$ for the set of Boolean values (0 for *false* and *1* for true). $\mathbb{N}^{+}$ denotes the set of positive natural numbers.

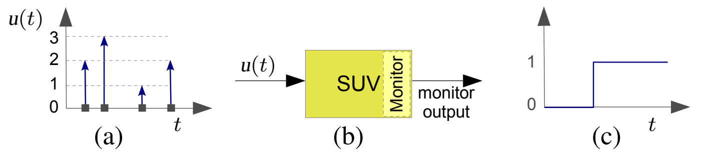


Fig. 1: (a) A discrete event sequence ($d = 3$); (b) Our SUV embedding a monitor; (c) The SUV monitor output.

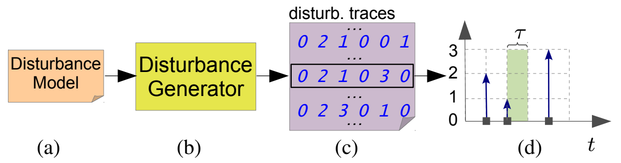


Fig. 2: (a) Disturbance model; (b) CMurphi-based disturbance generator; (c) Generated sequence of disturbance traces ($d = 3, h = 6$); (d) The discrete event sequence associated to the trace in the black rectangle in part (c), given time quantum τ.

A. Modelling the Operational Environment

Our System Under Verification (SUV) is a Discrete Event System (DES), namely a continuous time Input-State-Output deterministic dynamical system [5] whose inputs are *discrete event sequences*. A discrete event sequence is a function $u(t)$ associating to each (continuous) time instant $t \in \mathbb{R}^+$ a *disturbance event* (or, simply, *disturbance*). Disturbances, encoded by integers in the interval $[0, d]$ (for a given $d \in \mathbb{N}^+$), represent uncontrollable events (e.g., faults). We use event 0 to represent the event carrying no disturbance. As no system can withstand an infinite number of disturbances within a finite time, we require that, in any time interval of finite length, a discrete event sequence $u(t)$ differs from 0 only in a finite number of time points (Fig. 1a).

System level verification follows an *Assume-Guarantee* approach aimed at showing that the SUV meets its specification (*Guarantee*) as long as the SUV operational environment behaves as expected (*Assume*). As we focus on *bounded* system level verification, we model (Definition 1) the SUV operational environment as the sequence of disturbances our SUV is expected to withstand within a *finite* time horizon. We also bound the *time quantum* between two consecutive disturbances.

Definition 1 (Disturbance trace): Let $h, d \in \mathbb{N}^+$. An (h, d) *disturbance trace* δ is a finite sequence $\delta : [0, h-1] \to [0, d]$. Given $\tau \in \mathbb{R}^+$ (*time quantum*), to an (h, d) disturbance trace δ we can univocally associate a discrete event sequence u^τ_δ, defined as follows: for all $t \in \mathbb{R}^{\geq 0}$, if there exists $j \in [0, h-1]$ such that $t = \tau j$ then $u^\tau_\delta(t) = \delta(j)$, else $u^\tau_\delta(t) = 0$ (no disturbance).

Thus a disturbance trace δ defines an operational scenario (namely, u^τ_δ) for our SUV. Fig. 2d shows the discrete event sequence associated to a disturbance trace. We represent our SUV *operational environment* as a finite *set* of (h, d) disturbance traces $\Delta = \{\delta_0, \ldots, \delta_{n-1}\}$, since $U^\tau_\Delta = \{u^\tau_{\delta_0}, \ldots, u^\tau_{\delta_{n-1}}\}$ (for a given $\tau \in \mathbb{R}^+$) defines the operational scenarios our SUV should withstand. Note that, by taking h *large enough* (as in Bounded Model Checking (BMC)) and τ *small enough* (to faithfully model our SUV operational scenarios), we can achieve any desired precision. On such considerations rests the effectiveness of the approach.

As it is typically infeasible to define a SUV operational environment by explicitly listing all its disturbance traces, we define an operational environment with a *disturbance model* which is in turn defined as the language accepted by a suitable Finite State Automaton. The following example clarifies this point.

Example 1: Consider a disturbance model consisting of one disturbance (namely, a fault) which is always recovered within 4 seconds. Between two consecutive disturbances

000000✓	010000✓
000001✓	010001⊗
000010✓	01001⊗
000011⊗	0101⊗
000100✓	011⊗
000101⊗	100000✓
00011⊗	100001✓
001000✓	10001⊗
001001⊗	1001⊗
00101⊗	101⊗
0011⊗	11⊗

(a)

```
function disturbanceModel(h)
  c ← 0; /* counter */
  t ← 0; /* time */
  while $t \leq h$ do
    d ← read();   t ← t + 1;
    if $c > 0$ then   c ← c − 1;
    if $d = 1$ then
      if $c > 0$ then   return ⊗;
      else   c ← 4;
  return ✓;
end
```

(b)

Fig. 3: Example 1: (a) Admissible disturbance traces (✓) and shortest disturbance sequences that cannot be extended to an admissible disturbance trace (⊗); (b) Finite state automaton recognising the language of admissible disturbance traces (disturbance model).

(faults) there must be at least 5 seconds. We assume that disturbances can arise only at time steps multiple of $\tau = 1$ second (*time quantum*). We also set the verification time horizon to 6 seconds. In Fig. 3a we show disturbance traces represented as strings of zeros (no disturbance) and ones (disturbance), with time flowing from left to right. Strings terminated by ✓ denote all the disturbance traces accepted by the disturbance model (*admissible disturbance traces*). Strings terminated by ⊗ are the shortest sequences of disturbances that *cannot* be extended to an admissible disturbance trace. Fig. 3b shows pseudo-code for a finite state automaton recognising such a language.

We define a finite state automaton for a disturbance model using the modelling language of a finite state model checker (namely, CMurphi [6]), along the lines of [1].

B. Modelling the Property to be Verified

Along the lines of [7], we model the property to be verified with a continuous-time *monitor* which observes the state of the system to be verified and checks whether the property under verification is satisfied (Fig. 1b). The output of the monitor is 0 as long as the property under verification is satisfied and becomes and stays 1 (*sustain*) as soon as the property fails, thus ensuring that we never miss a property failure report, even when sampling the monitor output only at discrete time points (Fig. 1c). The use of monitors gives us a flexible approach to model the property to be verified. In particular, it is easy to model bounded safety and bounded liveness properties as monitors.

C. Modelling the SUV

Since the monitor output is all we need to carry out our verification task, we can model our SUV *along with* the property to be verified as a DES with an *embedded* monitor (Fig. 1b). We call *Monitored Discrete Event System (MDES)* such a DES.

According to our *black-box* approach to SUV modelling, given a time quantum $\tau \in \mathbb{R}^+$, Definition 2 formalises an (h, d) MDES as a function $\mathcal{H}$ associating, to each (h, d) disturbance trace δ, a Boolean value $\mathcal{H}(\delta)$ representing the output of the SUV monitor at time $T = \tau h$ (the time horizon), when the system (starting from its initial state) is given as input the discrete event sequence $u^\tau_\delta(t)$ associated to δ. For any disturbance trace δ, $\mathcal{H}(\delta)$ is 1 (*error*) if and only if $u^\tau_\delta(t)$ *violates* the property under verification within time horizon $T = \tau h$ (with the SUV starting from its initial state).

Definition 2 ((h, d) MDES): Let $h, d \in \mathbb{N}^+$. A (h, d) Monitored Discrete Event System (MDES) is a function $\mathcal{H} : ([0, h-1] \to [0, d]) \to$ Bool mapping all (h, d) disturbance traces to Boolean values.

D. System Level Formal Verification

Definition 3 formalises our bounded System Level Formal Verification problem.

Definition 3: A *System Level Formal Verification (SLFV) problem* is a tuple $P = (h, d, \Delta, \mathcal{H})$ where: $h, d \in \mathbb{N}^+$, $\Delta = \{\delta_0, \ldots, \delta_{n-1}\}$ is an (h, d) set of disturbance traces, and $\mathcal{H}$ is a (h, d) MDES.

The *answer* to SLFV problem P is *FAIL* if there exists a disturbance trace δ in Δ such that $\mathcal{H}(\delta) = 1$ (in such a case also the *counterexample* δ is returned), *PASS* otherwise.

Note that, notwithstanding the fact that the number of states of our SUV is infinite and we are in a continuous time setting, to answer a SLFV problem we only need to check a *finite* number of disturbance traces. This is because we are bounding: (a) our time horizon to $T = \tau h$, and (b) the set of time points at which disturbances can take place, by taking τ as the time quantum among disturbance events.

E. Parallel HILS based SLFV

In our black-box parallel approach, the MDES $\mathcal{H}$ defining our SUV (plus the property to be verified) is defined using the modelling language of a suitable *simulator* (e.g., MatLab and Stateflow for Simulink). We compute the answer to a SLFV problem $(h, d, \Delta, \mathcal{H})$ by simulating *each* operational scenario δ in the operational environment Δ, thus by performing an exhaustive (with respect to Δ) Hardware In the Loop Simulation (HILS).

In order to enable parallel computation, we evenly partition the sequence of disturbance traces Δ into $k \in \mathbb{N}^+$ sequences of disturbance traces $\Delta_0, \ldots, \Delta_{k-1}$. We then use such k slices to compute, in parallel, k highly optimised simulation campaigns (Fig. 4b), which can be simulated in parallel using k *simulators*, each one running (on a different core of our multiple machines) a model for $\mathcal{H}$ (Fig. 4c–d). The *answer* to the SLFV problem is *FAIL* if one of the simulation campaigns raises the simulator output function to 1 (in this case the disturbance trace δ which raised the error is returned as a *counterexample*, see Fig. 4e). The answer is *PASS* otherwise.

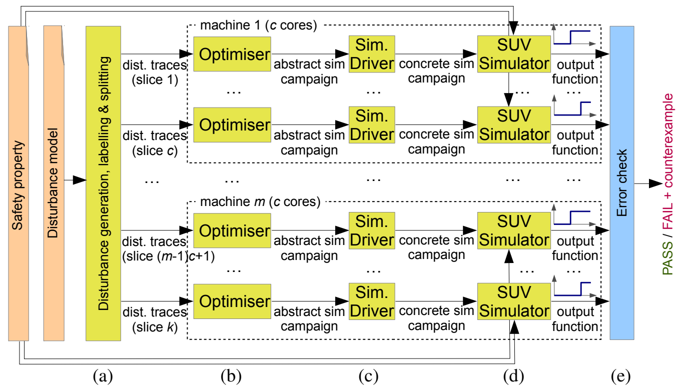


Fig. 4: Our approach to parallel SLFV: k parallel processes are run on m multi-core machines (we show a possible deployment with machines having c cores each, i.e., $k = mc$).

Each simulator accepts four basic commands: *store*, *load*, *free*, *run*. Command *store*(l) stores in memory the current state of the simulator and labels with l such a state. Command *load*(l) loads into the simulator the stored state labelled with l. Command *free*(l) removes from the memory the state labelled with l. Command *run*(e, t) (with $e \in [0, d]$ and $t \in \mathbb{R}^+$) injects disturbance e and then advances the simulation of time t. A *simulation campaign* is thus a sequence of simulator commands.

Using commands *store* and *load* we can avoid revisiting simulation states (much as in explicit model checking). Using command *free* we can remove from the memory states that will never be needed in the remaining part of the simulation campaign. This is needed since each state may require many KB of memory (150–300 KB in the case study presented in this paper).

III. Omission Probability

This section formally defines the notion of Omission Probability (OP) (Definitions 4 and 5) and provides an upper bound for it, which can be computed anytime *during* the parallel verification process from the number of the yet-to-be-simulated traces in each slice (Theorem 1).

Notation 1 (Set of permutations of a set): Let $\Delta = \{\delta_0, \ldots, \delta_{n-1}\}$ be a finite non-empty set. We denote with $\text{Perm}(\Delta)$ the *set of permutations* of elements of Δ:

$$\text{Perm}(\Delta) = \left\{ (\theta_0, \ldots, \theta_{n-1}) \;\middle|\; \begin{array}{l} (\forall i \in [0, n-1]\ \theta_i \in \Delta) \wedge \\ (\forall i, j \in [0, n-1] \\ \qquad\qquad i \neq j \to \theta_i \neq \theta_j) \end{array} \right\}$$

If $\hat{\Delta} = (\delta_0, \ldots, \delta_{n-1}) \in \text{Perm}(\Delta)$ we write also $\hat{\Delta}(i)$ for δ_i.

A Random Sequence Generator (RSG) models the *extraction* of a random permutation from any given finite non-empty set (which, in our case, will be the set of admissible disturbance traces Δ). This is formalised in Definition 4.

Definition 4 (Random Sequence Generator): Let Δ be a finite non-empty set. A Random Sequence Generator (RSG) for Δ is a probability space $(\Omega, \mathcal{F}, P)$, where:

- Ω (the space of *outcomes*) is the set of permutations of Δ, that is $\Omega = \mathrm{Perm}(\Delta)$.
- $\mathcal{F}$ (the space of *events*) is the set of subsets of Ω, that is: $\mathcal{F} = 2^{\Omega} = \{E \mid E \subseteq \Omega\}$.
- $P : \mathcal{F} \to [0, 1]$ is a probability measure such that, for all $\omega \in \Omega$, $P(\omega) = \frac{1}{|\Delta|!}$. That is, permutations of Δ are extracted with uniform probability. Since Ω is countable (actually finite), the probability of any event $E = \{\omega_1, \ldots, \omega_w\}$ is defined as $P(E) = \sum_{\omega \in E} P(\omega)$.

Let $(\Delta_0, \ldots, \Delta_{k-1})$ be a partition of Δ into $k \in \mathbb{N}^+$ disjoint non-empty sets. For any $0 \leq i < k$, let $(\Omega_i, \mathcal{F}_i, P_i)$ be an RSG for Δ_i. A Random Sequence Generator for $(\Delta_0, \ldots, \Delta_{k-1})$ is a probability space $(\Omega, \mathcal{F}, P)$, where: $\Omega = \times_{i=0}^{k-1} \Omega_i$, $\mathcal{F} = \times_{i=0}^{k-1} \mathcal{F}_i$ and, for each event $E_0 \times \cdots \times E_{k-1} \in \mathcal{F}$ ($E_i \in \mathcal{F}_i$ for each $0 \leq i < k$), $P(E_0 \times \cdots \times E_{k-1}) = \prod_{i=0}^{k-1} P_i(E_i)$.

Note that, by Definition 4, a RSG for a partition $(\Delta_0, \ldots, \Delta_{k-1})$ of Δ models the extraction of k permutations of, respectively, $\Delta_0, \ldots, \Delta_{k-1}$. For all $0 \leq i < k$, the extracted permutation of Δ_i is chosen *uniformly* among all possible permutations of Δ_i. Also, the k permutations are extracted *independently* from each other.

Let $(q_0, \ldots, q_{k-1})$ be a tuple of integers, with $q_i \in \{0, \ldots, |\Delta_i|\}$ for each $0 \leq i < k$. Definition 5 defines the probability of omitting the simulation of a trace $\bar{\delta} \in \Delta$ containing an error (i.e., $\mathcal{H}(\bar{\delta}) = 1$) when the verification process has *already* examined, for all $0 \leq i < k$, q_i disturbance traces from a random permutation of slice Δ_i. This is called Omission Probability (OP).

Definition 5 (Omission Probability): Let (h, d, Δ, $\mathcal{H}$) be a System Level Formal Verification (SLFV) problem and $(\Delta_0, \ldots, \Delta_{k-1})$ be a partition of Δ into $k \in \mathbb{N}^+$ disjoint non-empty sets. Let $(\Omega, \mathcal{F}, P)$ be an RSG for $(\Delta_0, \ldots, \Delta_{k-1})$, and $(q_0, \ldots, q_{k-1})$ a tuple such that $q_i \in \{0, \ldots, |\Delta_i|\}$ for each $0 \leq i < k$.

The *Omission Probability (OP)* for $(\Delta_0, \ldots, \Delta_{k-1})$ at stage $(q_0, \ldots, q_{k-1})$, denoted as $\mathrm{OP}_{\mathcal{H}}(|\Delta_0|, \ldots, |\Delta_{k-1}|, q_0, \ldots, q_{k-1})$ is defined as:

$$P\left(\left\{(\omega_0,\ldots,\omega_{k-1}) \middle| \begin{array}{l} \forall i \in [0, k-1]\ \omega_i \in \Omega_i \ \wedge \\ A((\omega_0,\ldots,\omega_{k-1}),(q_0,\ldots,q_{k-1})) \\ \wedge \\ B((\omega_0,\ldots,\omega_{k-1}),(q_0,\ldots,q_{k-1})) \end{array}\right\}\right)$$

where A (*After*) and B (*Before*) are defined as follows:

$$\begin{aligned} &A((\omega_0, \ldots, \omega_{k-1}), (q_0, \ldots, q_{k-1})) \ = \\ &\qquad \exists i \in [0, k-1]\ \exists j \in [q_i, |\Delta_i|] \ \ \mathcal{H}(\omega_i(j)) = 1; \\ &B((\omega_0, \ldots, \omega_{k-1}), (q_0, \ldots, q_{k-1})) \ = \\ &\qquad \forall i \in [0, k-1]\ \forall j \in [0, q_i - 1] \ \ \mathcal{H}(\omega_i(j)) = 0. \end{aligned}$$

In Definition 5, formula A (After) states that there exists a yet-to-be-simulated trace $\bar{\delta}$ (some trace $j \geq q_i$ of some slice i) containing an error, i.e., such that $\mathcal{H}(\bar{\delta})$ evaluates to 1. Formula B (Before) states that none of the already simulated traces contains an error, i.e., function $\mathcal{H}$ evaluates to 0 for all of them.

The following Theorem 1 (proof omitted for lack of space) gives an upper bound to the OP, after having simulated q_i randomly extracted traces from slice Δ_i (for each $0 \leq i < k$). Importantly, the bound provided does *not* depend on $\mathcal{H}$, i.e., it is *independent* of the system model.

Theorem 1: Let (h, d, Δ, $\mathcal{H}$) be a SLFV problem and $(\Delta_0, \ldots, \Delta_{k-1})$ be a partition of Δ into $k \in \mathbb{N}^+$ disjoint non-empty sets. Let $(\Omega, \mathcal{F}, P)$ be a Random Sequence Generator (RSG) for $(\Delta_0, \ldots, \Delta_{k-1})$ and $(q_0, \ldots, q_{k-1})$ a tuple such that $q_i \in \{0, \ldots, |\Delta_i|\}$ for each $0 \leq i < k$. We have:

$$\mathrm{OP}_{\mathcal{H}}(|\Delta_0|,\ldots,|\Delta_{k-1}|,q_0,\ldots,q_{k-1}) \leq 1 - \min\left\{\frac{q_i}{|\Delta_i|} \middle| 0 \leq i < k\right\}.$$

Note that the construction of the slices $\Delta_0, \ldots, \Delta_{k-1}$ from Δ is *non-deterministic* (i.e., *any* partitioning of Δ would work), whereas, for each slice, the selection of a permutation is a *probabilistic* process, modelled as a RSG. Accordingly, Theorem 1 bounds the OP using the *worst case* distribution, i.e., the distribution yielding the greatest OP. From this stems the $\min$ function in the expression of Theorem 1.

Finally, we observe that, from Theorem 1, it follows that $\mathrm{OP}_{\mathcal{H}}(|\Delta_0|, \ldots, |\Delta_{k-1}|, |\Delta_0|, \ldots, |\Delta_{k-1}|) = 0$, that is, our verification task terminates after $\max\{|\Delta_i| \mid 0 \leq i < k\}$ parallel steps, having simulated all traces in Δ.

IV. Random Exhaustive Campaigns

In this section we give more details on our CMurphi-based disturbance trace generator (Section IV-A) and splitter (Section IV-B), and present our *simulation campaign optimiser* (Section IV-C) which enables random exhaustive parallel System Level Formal Verification (SLFV).

A. Disturbance Trace Generation

Our CMurphi-based trace generator (see Section II-A and Fig. 4a) works in Depth-First Search (DFS) mode, and hence produces a sequence Δ of n disturbance traces in lexicographic order. Furthermore, each generated trace δ in Δ is annotated with *labels* and is of the form $\delta = (l_0, d_0, l_1, d_1, \ldots, l_{h-1}, d_{h-1}, l_h)$, where $\delta = (d_0, \ldots, d_{h-1})$ is a sequence of disturbances satisfying the disturbance model and $l_0, \ldots, l_h$ belong to a countable set of *labels* L (e.g., $\mathbb{N}^+$). Labels are defined by an *injective map* λ from finite sequences of disturbances (including the empty sequence) to L. As a consequence, prefixes of disturbance sequences $(\hat{d}_0, \ldots, \hat{d}_{p-1})$ common to multiple disturbance traces in Δ are followed by the *same* label $\hat{l}_p = \lambda(\hat{d}_0, \ldots, \hat{d}_{p-1})$ (see Fig. 5). Labels identifying common disturbance prefixes are essential in the efficient computation of highly optimised simulation campaigns, as they allow to decide which states of the simulators should be stored, as they may be needed later (see the LBT construction phase of our optimiser). Note that, given that our CMurphi-based generator runs in DFS mode, disturbance traces (which are lexicographically ordered) can be labelled at *no additional computational cost* during generation, as shown in [1]. Trace labelling during generation greatly increases the efficiency of the optimiser.

In the following, we will use Δ^{λ} instead of Δ when we want to emphasise that traces in Δ are annotated with labels, or when we need such labels.

Algorithm 1: Optimiser pseudo-code

```
Input: $\Delta^\lambda$, a file holding a labelled lex-ordered sequence of
       disturbance traces
Output: $\chi$, the computed simulation campaign
1  $\chi \leftarrow$ an empty sequence of commands;
2  $LBT \leftarrow buildLBT(\Delta^\lambda)$;
3  $\Delta^\lambda_{rnd} \leftarrow rsg(\Delta^\lambda)$;
4  $lastTraces \leftarrow$ a map associating to each label $l \in LBT$ the index of
   the last trace in $\Delta^\lambda_{\mathrm{rnd}}$ where $l$ occurs;
5  $stored \leftarrow$ empty set of labels ; /* invariant: $stored \subseteq LBT$ */
6  $l_0 \leftarrow$ first label common to all traces;
7  append $store(l_0)$ to $\chi$;
8  $stored \leftarrow stored \cup \{l_0\}$;
9  foreach $\delta^\lambda$ in $\Delta^\lambda_{rnd}$ do
10     $l_{load} \leftarrow$ right-most label of $\delta^\lambda$ in $stored$;
11     append $load(l_{load})$ to $\chi$;
12     append $free(l)$ to $\chi$ for each label $l \in stored$ which will never
       occur in later traces (according to $lastTraces$);
13     append to $\chi$ commands to simulate $\delta^\lambda$ (from $l_{load}$) and to store any
       intermediate states needed to speed-up simulation of later traces;
14 return $\chi$;
```

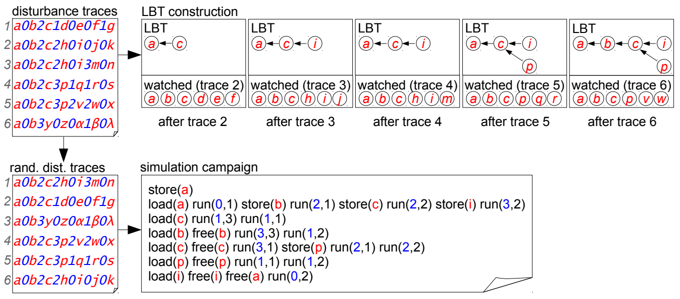
B. Disturbance Trace Splitting

The computed lexicographically ordered sequence of labelled disturbance traces Δ^λ is split (see Section II-E) into k disjoint sequences (*slices*) in order to enable parallel verification via k parallel processes. The splitting process produces slices containing $s = \lfloor n/k \rfloor$ traces each (except the last slice, which may contain up to $k-1$ more traces if n is not a multiple of k), placing trace $0 \leq i < n$ into slice $\min\{\lfloor i/s \rfloor, k-1\}$. Hence each slice contains traces which are *close* in the lexicographic ordering. This is important as the optimisation stage is concerned: lexicographically close traces often have *long* common prefixes, and this allows the optimiser to compute a more efficient simulation campaign within the restrictions on the number of states that each simulator can keep simultaneously stored (when compared to the campaign that could be computed if each slice contained $\lfloor n/k \rfloor$ traces arbitrarily chosen within the whole set).

C. Simulation Campaign Optimiser

Given a sequence (in particular, a *slice*) of (h, d) disturbance traces Δ^λ in lexicographic order, our optimiser (see Fig. 4b) computes a simulation campaign which executes them in a random order uniformly chosen among all possible orders (thus, implementing a Random Sequence Generator (RSG)). The computed campaign is *abstract* in that, for all commands of the form $run(e, t)$, t is a natural number and not an actual time duration. By providing a time step $\tau \in \mathbb{R}^+$, χ can be instantiated into a *concrete* simulation campaign χ_τ, by replacing all $run(e, t)$ commands by $run(e, t\tau)$.

Our simulation campaign optimiser is sketched as Algorithm 1. As the input sequence Δ^λ of disturbance traces can be too big to be kept in main memory, our optimiser is a *disk-based* algorithm which makes a careful use of the available RAM and reads the input file sequentially multiple times. In the first scan of Δ^λ, the optimiser builds a data structure called Labels Branching Tree (LBT) as completely as possible within the available RAM. Afterwards, it randomises Δ^λ producing a random permutation $\Delta^\lambda_{\mathrm{rnd}}$ uniformly chosen among all possible permutations. Finally, it reads $\Delta^\lambda_{\mathrm{rnd}}$ to produce the abstract



```
store(a)
load(a) run(0,1) store(b) run(2,1) store(c) run(2,2) store(i) run(3,2)
load(c) run(1,3) run(1,1)
load(b) free(b) run(3,3) run(1,2)
load(c) free(c) run(3,1) store(p) run(2,1) run(2,2)
load(p) free(p) run(1,1) run(1,2)
load(i) free(i) free(a) run(0,2)
```

Fig. 5: Simulation campaign optimiser: construction of an LBT from 6 labelled traces in lex order, random sequence generation, and generation of the optimised campaign. Labels are shown as *red letters* and disturbances as *blue numbers*.

simulation campaign from the LBT according to the chosen random order.

1) LBT Construction: The LBT is a tree of labels rooted at l_0, the first label of all traces (e.g., $l_0 = a$ in Fig. 5). The LBT collects *branching labels*, i.e., labels l_i for which there exist at least two labelled disturbance traces $\delta^\lambda = (l_0, d_0, \ldots, l_i, d_i, \ldots, l_h)$ and $\delta^{\lambda'} = (l_0, d_0, \ldots, l_i, d'_i, \ldots, l'_h)$ in Δ^λ which have the *same prefix* up to l_i and such that $d_i \neq d'_i$. Branching labels represent simulator states whose storing may save simulation time (by loading them back later). Label l_j is a child of l_i in the LBT iff, for all $\delta^\lambda = (l_0, d_0, \ldots, l_i, \ldots, l_j, \ldots, l_h) \in \Delta^\lambda$, no l_k in δ^λ with $i < k < j$ is in the LBT (note: all such δ^λ are identical at least up to l_j).

The construction of the LBT is shown as function *buildLBT*() in Algorithm 2. The function scans the input slice in order to recognise branching labels, keeping in array *watched* the labels of the last processed trace. In fact, as the traces in Δ^λ are lexicographically ordered, these are the *only* labels that may become branching when processing a new trace. To see why, assume that the optimiser is processing, e.g., trace 2 in Fig. 5 (top left). As this trace starts to be different with respect to the previous trace (trace 1) from the disturbance at step 2 (i.e., disturbance 2 right after label c), the optimiser infers that labels d, e, f, g of trace 1 will never occur in later traces of Δ^λ, and will never become branching.

As for the actual recognition of a new branching label and its addition to the LBT, assume that function *buildLBT*() is processing disturbance trace δ^λ (line 7). Variable l_{lbt} is the right-most label in δ^λ already in the LBT, and l_w is the right-most label in δ^λ which belongs also to array *watched*. As l_0 is put both in the LBT and in *watched[0]* at the beginning, both values are always defined. The algorithm infers that the current trace is identical to the previously processed trace up to l_w, but differs from it after that point. If $l_w = l_{lbt}$, label l_w is already branching, and nothing has to be done. Otherwise, the new label l_w is recognised as branching, and is added to the LBT as a child of l_{lbt} (as, given that the input traces are in lexicographic order, $l_w \neq l_{lbt}$ implies that l_w is on the right of l_{lbt} in δ^λ). As l_{lbt} could already have children in the LBT, the tree may need to be rearranged to accommodate the new label l_w. Given that the input traces are in lexicographic order, the last task is very simple, as at most one child of l_{lbt} must be moved. This child, if exists, must be a label that occurred in the previous trace, i.e., it belongs to the *watched* array (line 11).

Algorithm 2: Function *buildLBT*()

```
function buildLBT($\Delta^\lambda$)
    LBT ← empty tree of labels;
    watched ← empty array [0..$h-1$] of labels;
    let $l_0$ be the first label common to all traces in $\Delta^\lambda$;
    set $l_0$ as the root of LBT;
    watched[0] ← $l_0$;
    foreach $\delta^\lambda$ in $\Delta^\lambda$ do
        $l_{lbt}$ ← right-most label in $\delta^\lambda$ in LBT;
        $l_w$ ← right-most label in $\delta^\lambda$ in watched;
        if $l_{lbt} \neq l_w$ then
            move any child of $l_{lbt}$ in LBT also belonging to watched as
              to be child of $l_w$;
            add $l_w$ to LBT as child of $l_{lbt}$;
        watched ← $[l_0, \ldots, l_{h-1}]$;
    return LBT;
end
```

Fig. 5 (top) shows an example of LBT construction starting from a sequence of 6 labelled disturbance traces Δ^λ. Out of 25 labels in Δ^λ, only 5 of them belong to the LBT.

2) Random Sequence Generation: Once the LBT is built, the optimiser calls function *rsg*() (pseudo-code omitted for lack of space), which reads the input slice Δ^λ again to compute a random permutation $\Delta^\lambda_{\text{rnd}}$ uniformly chosen among all possible permutations, thus implementing a RSG. The function implements a disk-based multi-round algorithm which takes efficiency into account by using, in each round, as much main memory as possible and by reading/writing the input/output trace files sequentially.

Let $n = |\Delta^\lambda|$ be the number of input disturbance traces. Given parameter z for the maximum number of disturbance traces which can be simultaneously stored in main memory, the algorithm, at each round $r \geq 1$, selects the z traces which will have output positions in the interval $[(r-1)z, \min(n, rz-1)]$. Such a selection is performed by computing the first z elements of a random permutation of the traces not yet in the output file $\Delta^\lambda_{\text{rnd}}$, chosen uniformly among all possible permutations. The z selected traces are then appended to $\Delta^\lambda_{\text{rnd}}$ (according to their output positions), all the others are dumped to a temporary file, which becomes the input of the next round. Function *rsg*() terminates in $\lceil n/z \rceil$ rounds.

3) Simulation Campaign Computation: Once the random order $\Delta^\lambda_{\text{rnd}}$ with which the input disturbance traces Δ^λ must be simulated has been computed, the optimiser reads the randomised input slice from disk two more times to compute the abstract simulation campaign.

In the first scan of $\Delta^\lambda_{\text{rnd}}$, the optimiser computes (line 4 of Algorithm 1), for each branching label $l \in LBT$, the position of the last trace in $\Delta^\lambda_{\text{rnd}}$ where it occurs. This information will be important, during the second scan, to free-up the simulator disk space as soon as possible.

In the second scan of $\Delta^\lambda_{\text{rnd}}$, the optimiser actually computes the abstract simulation campaign χ, also keeping track of which LBT labels are stored in simulator disk space at any moment (set *stored*, line 5 of Algorithm 1).

For each δ^λ in $\Delta^\lambda_{\text{rnd}}$ (line 9 of Algorithm 1), suitable commands are appended to the output simulation campaign χ to simulate δ^λ. Let l_{load} be the right-most label of δ^λ currently stored by the simulator. The optimiser appends to the output campaign the following commands: (i) *load*(l_{load}); (ii) *free*(l) for each label $l \in$ LBT which represents a currently stored state that will never occur after trace δ^λ; (iii) a command of the form *run*($\hat{d}$, *steps*) for each maximal sub-sequence of length *steps* in δ^λ (starting from l_{load}) of the form $\hat{d}, l_{i_1}, 0, l_{i_2}, \ldots, 0, l_{i_{\mathit{steps}}} \tilde{d}, \tilde{l}$ where either $\tilde{d} \neq 0$ or label $\tilde{l}$ needs to be stored. In the latter case, command *store*($\tilde{l}$) is appended as well. Label $\tilde{l}$ needs to be stored if it is in the LBT but not yet stored and it will occur again in a later trace. If the simulator disk space is full, the algorithm frees it up by appending *free* commands to the output simulation campaign χ, by selecting labels to free as to minimise the simulation cost (number of steps) to drive the simulator to the state they represent.

Fig. 5 (bottom) shows the simulation campaign computed by the optimiser on the slice in Fig. 5 (top). Except for the first command which stores a (the label common to all traces and representing the simulator initial state), each line represents the portion of the simulation campaign stemming from each trace. Note that only the first trace is simulated entirely, while all the others are simulated starting from intermediate, previously stored, states.

D. Optimiser Soundness and Completeness

Given a SLFV problem $P = (h, d, \Delta, \mathcal{H})$, it can be shown that function *rsg*() (see line 3 of Algorithm 1) computes a permutation $\Delta^\lambda_{\text{rnd}}$ of the input traces, extracted with uniform probability among all possible permutations, and thus it effectively implements a RSG.

Furthermore, it can be shown that Algorithm 1 computes a simulation campaign χ which is *sound* and *complete* with respect to $\Delta^\lambda_{\text{rnd}}$ (or, equivalently, Δ^λ). That is: if the answer to P is *PASS*, then the output of the simulator at the end of the execution of the simulation campaign χ will be 0 (*soundness*). On the other hand, if the answer to P is *FAIL* and δ is the first counterexample in $\Delta^\lambda_{\text{rnd}}$, then the output of the simulator will raise from 0 to 1 during the simulation of a command of χ stemming from δ (*completeness*). The result above can be proved by formalising the notion of *simulator for $\mathcal{H}$* along the lines of [1].

V. Experimental Results

In this section we evaluate the effectiveness of our *random exhaustive parallel approach* to System Level Formal Verification (in short rSLFV) as follows. First, we evaluate the *overhead* due to the randomisation of disturbance traces needed to enable computation of Omission Probability (OP), by comparing our rSLFV approach with the *deterministic parallel approach* (in short dSLFV) of [1]. Second, we evaluate the behaviour of the coverage and the OP bound with respect to simulation time. Third, we evaluate speed-up and efficiency of our parallel approach.

We use the same case study of [1], [8], i.e., the Fuel Control System (FCS) model included in the Simulink distribution. The FCS has three sensors subject to faults (disturbances). We verify one of the system level specifications for such a model, namely: the *fuel_air* model variable is never 0 for more than one second. Accordingly, our System Under Verification (SUV) consists of the Simulink FCS model along with a monitor for the property under verification. In our setting, the complexity of the computation of an optimised simulation campaign primarily depends on the number of disturbance traces to be simulated. Thus, the worst case for our approach is when all disturbance traces have to be simulated, i.e., when the answer to the System Level Formal Verification (SLFV) problem is *PASS*. We know that this is the case when no more than one fault occurs within a second. Thus, this will be our disturbance model. We set the disturbance traces horizon h to 100 and τ (quantum between disturbances) to 1 second.

We ran experiments on multiple Linux PCs, each one equipped with 2 Intel Xeon 3.0 GHz CPUs with 4 cores each and 8 GB RAM. We executed 8 processes (optimisation and simulation) in parallel (one per available core) on each machine. As, in a multi-core setting, the local disk may quickly become a performance bottleneck if heavily used by multiple processes, we have replaced it with 8 RAM disks of 500 MB each per machine, in order to store simulation states. Accordingly, we have used the multi-core version of the dSLFV optimiser of [1] as presented in [9]. Given that, in our case study, the size of the simulation state files is of about 150–300 KB, this experimental setting allowed our optimiser to count on the possibility, for each simulator, to keep at most 1800 states simultaneously stored.

A. Disturbance Trace Generation and Splitting

As [1], we use CMurphi to generate a lexicographically ordered sequence Δ of (labelled) admissible disturbance traces. The generator produces 4,023,955 traces in about 28 minutes and saves them in a 3.5GB file. We then split such a Δ into k slices, with $k = 128, 256, 512$ to enable parallel computation on, respectively, 16, 32, 64 (8-core) machines. Splitting takes a few seconds, regardless of the value of k.

B. Computation of Simulation Campaigns

Table in Fig. 6a compares the performance of our rSLFV optimiser against the dSLFV optimiser. Column *#slices* gives the number of slices in which the sequence of admissible disturbance traces has been partitioned. Column *#traces per slice* shows the number of traces in any single slice (except the last slice, which may have up to *#slices*-1 more traces, as the overall number of traces is not a multiple of *#slices*). Columns *dSLFV optimiser* and *rSLFV optimiser* show the maximum time needed by, respectively, the deterministic and the random optimisers to compute the simulation campaign from a slice.

The random sequence generation phase makes the rSLFV optimisation process longer than that for dSLFV. The difference is, however, *negligible* with respect to the whole verification time (many hours, as described below).

C. Execution of the Simulation Campaigns

Table in Fig. 6b shows the execution time of the simulation campaigns generated by dSLFV and rSLFV optimisers.

The *price to pay* (in terms of simulation time) to enable computation of the OP *during* the simulation activity is quite significant (+247.83%) as for $k =$ *#slices* $= 128$, but can be *drastically mitigated*, if not neutralised, by using more parallel processes (higher values for $k =$ *#slices*). This behaviour is due to the fact that the rSLFV optimiser needs to compute a simulation campaign under the restriction that the number of states that the simulator can keep simultaneously stored is at most 1800 (totalling about 500 MB). For high values of k=*#slices* (e.g., $k = 512$), this is not a big obstacle. On the other hand, for lower values of k, the number of traces in each slice is higher and they share shorter common prefixes on average. Hence, a fully-optimised random order execution of them would need a too high number of simulation states to be simultaneously kept stored. As a consequence, the optimiser is forced to post *free* commands for many simulation states which would be needed again in yet-to-be-simulated traces. Such traces will then be simulated from the simulator initial state, thus yielding performance degradation.

Column *speedup* shows the ratios t_1/t_k, typically used in the evaluation of parallel algorithms, for both dSLFV and rSLFV. For each row ($k =$ *#slices*) of the table in Fig. 6b, time t_k is the *overall* time needed to carry out the SLFV task with k parallel processes, i.e., the sum of the disturbance trace generation and splitting time (about 28 minutes), optimisation time (from the table in Fig. 6a), and the max simulation time (column *max*) over all the $k =$ *#slices* slices. Time t_1 (serial time) is the overall time needed to carry out the SLFV task when only one parallel process is used. Let t_k^{avg} be the average time to simulate a slice where $k =$ *#slices* parallel processes are used (row *#slices* $= k$, column *avg*). For any value of k, the serial time can be estimated as $k \times t_k^{\text{avg}}$. As this value changes a little bit for different values of k, we estimated serial time t_1 as $\min\{128t_{128}^{\text{avg}}, 256t_{256}^{\text{avg}}, 512t_{512}^{\text{avg}}\}$. This leads to $t_1 \approx 470$ days as for dSLFV and $t_1 \approx 570$ days as for rSLFV (such huge values make clear that estimation is the only viable way to compute t_1). Note that in our computation we are slightly overestimating the serial time, since we are assuming that some traces of each slice must be simulated from the initial state. In an actual 1-process execution of a simulation campaign, the optimiser may exploit stored simulator states to avoid simulation of such traces from the initial state. As the time to simulate a single trace is of a few seconds and the simulator can keep only 1800 stored states, this is negligible with respect to the value of t_1.

Column *efficiency* in the table in Fig. 6b is computed, as typically done in the evaluation of parallel algorithms, by dividing the speedup by the number of parallel processes $k =$ *#slices*. Analogously to the speed-up, also here, the higher values of k the lower the overhead.

The overhead due to randomisation in terms of speedup and efficiency reduction are also shown in Fig. 6f–g.

D. Omission Probability Computation

Fig. 6c shows how our upper bound to the OP decreases as a function of the coverage (i.e., the ratio of admissible traces simulated) for $k = 128, 256, 512$. It can be observed that our OP bound is always *very close* to the ratio of yet-to-be-simulated traces, which is the best one can do (i.e., using only one parallel process) without any assumption on the number of error traces.

E. Completion Time Estimation

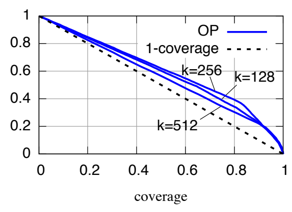
Fig. 6d shows that the OP bound decreases nearly *linearly* in time. The same happens with the coverage, which can thus be used as a reliable *estimator* for the completion time of the whole verification process. Fig. 6e shows the error percentage (on the true completion time) made by a completion time estimation based on the coverage. For each value x of the

#slices	#traces per slice	dSLFV optimiser	rSLFV optimiser
1	4,023,955	0:7:16	0:35:35
2	2,011,977	0:9:43	0:16:33
4	1,005,988	0:9:0	0:8:37
8	502,994	0:5:27	0:3:42
16	251,497	0:2:8	0:2:51
32	125,748	0:0:57	0:2:36
64	62,874	0:0:29	0:1:21
128	31,437	0:0:17	0:1:44
256	15,718	0:0:8	0:0:42
512	7,859	0:0:4	0:0:13

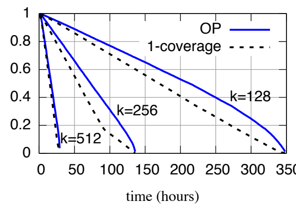
(a) Computation of simulation campaigns (time in h:m:s)

#mach.	#slices	min	max	avg	speedup	efficiency	approach
16	128	70:6:4	100:17:53	87:49:56	111.56×	87.15%	dSLFV
		216:42:13	348:51:47	308:46:18	39.17×	30.60%	rSLFV
		+209.13%	**+247.83%**	**+251.55%**	**+64.89%**	**+56.55%**	**overhead**
32	256	44:0:27	57:57:27	48:34:6	192.38×	75.15%	dSLFV
		63:53:54	136:18:14	108:14:19	100.03×	39.08%	rSLFV
		+45.20%	**+135.18%**	**+122.86%**	**+48.00%**	**+36.07%**	**overhead**
64	512	18:32:36	26:49:4	23:2:19	411.83×	80.43%	dSLFV
		22:9:19	29:23:33	26:43:31	458.01×	89.46%	rSLFV
		+19.48%	**+9.60%**	**+16.00%**	**−11.21%**	**−9.03%**	**overhead**

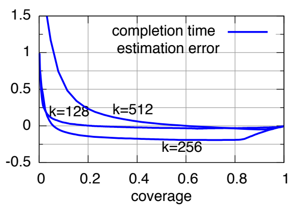
(b) Parallel execution of simulation campaigns by dSLFV and rSLFV (time in h:m:s)



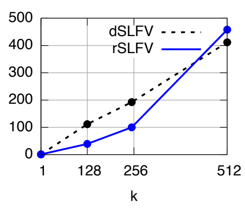
(c) OP against coverage



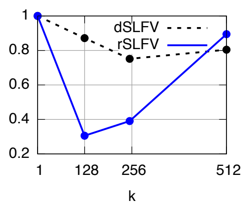
(d) OP & cov. against time



(e) Completion time



(f) Speedup



(g) Efficiency

Fig. 6: Experimental results

coverage, the error is computed as $((t_x/x) - t_c)/t_c$ where t_x is the time elapsed to reach coverage x and t_c is the true completion time. It can be observed that such a completion time estimation becomes accurate very quickly (e.g., when the coverage is $\geq 30\%$, the error is below 20%).

VI. Related Work

The work closest to ours is [1] where a parallel exhaustive Hardware In the Loop Simulation (HILS) based hybrid system model checking algorithm is presented. The main differences with respect to [1] are the following. (i) Our simulation campaign optimiser and the one in [1] both take as input the admissible disturbance sequences (simulation scenarios) in a lexicographic (Depth-First Search) order. However, the optimiser in [1] returns a simulation campaign scheduling scenarios in the very same order they had before optimisation, whereas our optimiser schedules simulation of *all* scenarios *exactly once* in a uniform random order. (ii) During the verification process, [1] only outputs the attained coverage, whereas, resting on the randomisation of the order with which scenarios are scheduled, we also output the attained Omission Probability (OP).

In a finite state (digital hardware verification) setting, the work in [2] presents an algorithm to estimate the coverage achieved during SAT based bounded model checking. Since computation paths are not selected uniformly at random, [2] does not provide any information about the OP.

Random model checking is a formal verification approach closely related to our setting. A random model checker provides, at *any time* during the verification process, an upper bound to the OP. Upon detection of an error, a random model checker stops returning a counterexample. Random model checking algorithms have been investigated, e.g., in [3], [10], [11]. The main differences with respect to our approach are the following. (i) All random model checkers generate simulation scenarios using a sort of Monte-Carlo based random walk. As a result, unlike our algorithm, none of them is exhaustive (within a finite time horizon). (ii) Random model checkers (e.g., see [3]) assume availability of a lower bound to the probability of selecting (with a random-walk) an error trace. Of course, being exhaustive, we do not have any such assumption.

The coverage yielded by random sampling a set of test cases has been studied by mapping it to the Coupon Collector's Problem (CCP) (see, e.g., [12]). In CCP we randomly extract elements (uniformly and with replacement) from a finite set of n test cases (disturbance traces in out context). Known results (see, e.g., [13]) tell us that the probability distribution of the number of test cases to be extracted in order to collect *all* n elements has *expected* value $\Theta(n \log n)$, and a small variance with known bounds. This would allow us to bound the OP during the verification. Differently from such CCP-based approaches, here we not only bound the OP, but also grant the completion of our verification task within *just* n trials. This is made possible by the fact that we first generate all disturbance traces.

Monte-Carlo based robustness analysis of Cyber-Physical Systems (CPSs) has been investigated in [14]. We note that, within a finite time bound, we are exhaustive whereas the approach in [14] is not. On the other hand, unlike out approach, [14] also evaluates how *robustly* the given property holds.

Probabilistic (e.g., see [15], [16]) and, more specifically, *simulation-based statistical model checking* approaches (e.g., see [8], [17]–[23]) are closely related to our work. In particular, [8] addresses statistical model checking of Simulink models and presents experimental results on the very same Simulink case study we use here. The main differences between such approaches and ours are the following. (i) Probabilistic model checking is a *white-box* approach (a model is available), whereas we are in a *black-box* setting (only a simulator is available). Thus, only simulation-based statistical model checking approaches can be used in our context. (ii) Statistical model checking is not exhaustive (within a finite time horizon), whereas we are. (iii) Both probabilistic and statistical model checking use a stochastic model for the System Under Verification (SUV), whereas in our setting the SUV is deterministic and disturbances are nondeterministic. The probability measure in our context, as in random model checking, stems from the randomisation of the verification process itself. (iv) None of the available simulation-based statistical model checking approaches addresses the problem of the optimisation of the simulation campaign, which is an essential step to make our *parallel random exhaustive* HILS based model checking viable.

Formal verification of Simulink models has been widely investigated, examples are in [24]–[26]. Such methods however focus on discrete time models (e.g., Stateflow or Simulink restricted to discrete time operators) with small domain variables. Therefore they are well suited to analyse critical subsystems, but cannot handle complex system level verification tasks (e.g., as our case study). This is indeed the motivation for the development of statistical model checking methods as those in [8], [22], for the exhaustive HILS based approach in [1], and for our present parallel random exhaustive HILS based approach.

VII. Conclusions

We presented a *parallel random exhaustive* Hardware In the Loop Simulation (HILS) based model checker for hybrid systems that, while being *exhaustive*, provides at *any time* during the verification process an *upper bound* to the probability that the System Under Verification (SUV) exhibits an error in a yet-to-be-simulated scenario (Omission Probability, OP). Our experimental results on the Fuel Control System (FCS) case study in the Simulink distribution show that, by exploiting parallelism, our simulation campaign optimiser effectively counteracts the simulation time overhead stemming from randomisation. Finally, we showed that our bound to the OP decreases *linearly* with the coverage, and thus is as good as it can be even in the worst case scenario (just one error trace). Furthermore, resting on randomisation, we can use the coverage as a reliable estimator for the time needed to complete the verification process.

Acknowledgments

The research leading to these results has received funding from the European Union's 7th Framework Programme (FP7 2007-2013) under grant agreements n. 317761 and 600773. The authors would like to thank the anonymous reviewers, whose comments helped in improving this paper.

References

[1] 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 2013*, ser. LNCS, vol. 8044. Springer, 2013, pp. 296–312.

[2] F. A. Aloul, B. D. Sierawski, and K. A. Sakallah, "Satometer: how much have we searched?" in *Proc. DAC 2002*. ACM, 2002, pp. 737–742.

[3] R. Grosu and S. Smolka, "Monte carlo model checking," in *Proc. TACAS 2005*, ser. LNCS, vol. 3440. Springer, 2005, pp. 271–286.

[4] R. Alur, "Formal verification of hybrid systems," in *Proc. EMSOFT 2011*. ACM, 2011, pp. 273–278.

[5] E. Sontag, *Mathematical Control Theory: Deterministic Finite Dimensional Systems*, ser. Texts in Applied Mathematics. Springer, 1998.

[6] G. Della Penna, B. Intrigila, I. Melatti, E. Tronci, and M. Venturini Zilli, "Exploiting transition locality in automatic verification of finite state concurrent systems," *STTT*, vol. 6, no. 4, pp. 320–341, 2004.

[7] O. Maler and D. Nickovic, "Monitoring temporal properties of continuous signals," in *Proc. FORMATS 2004 and FTRTFT 2004*, ser. LNCS, vol. 3253, 2004, pp. 152–166.

[8] P. Zuliani, A. Platzer, and E. Clarke, "Bayesian statistical model checking with application to simulink/stateflow verification," in *Proc. HSCC 2010*, 2010, pp. 243–252.

[9] T. Mancini, F. Mari, A. Massini, I. Melatti, and E. Tronci, "System level formal verification via distributed multi-core hardware in the loop simulation," in *Proc. PDP 2014*. IEEE, 2014, pp. 734–742.

[10] E. Tronci, G. Della Penna, B. Intrigila, and M. Venturini Zilli, "A probabilistic approach to automatic verification of concurrent systems," in *Proc. APSEC 2001*. IEEE, 2001, pp. 317–324.

[11] H. Sivaraj and G. Gopalakrishnan, "Random walk based heuristic algorithms for distributed memory model checking." *Electr. Notes Theor. Comput. Sci.*, vol. 89, no. 1, pp. 51–67, 2003.

[12] A. Arcuri, M. Iqbal, and L. Briand, "Random testing: Theoretical results and practical implications," *IEEE Trans. on Soft. Eng.*, vol. 38, no. 2, pp. 258–277, 2012.

[13] R. Motwani and P. Raghavan, *Randomized Algorithms*. New York, NY, USA: Cambridge University Press, 1995.

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

[15] G. D. Penna, B. Intrigila, I. Melatti, E. Tronci, and M. V. Zilli, "Finite horizon analysis of Markov chains with the Murphi verifier." *STTT*, vol. 8, no. 4-5, pp. 397–409, 2006.

[16] D. Jansen, J. Katoen, M. Oldenkamp, M. Stoelinga, and I. Zapreev, "How fast and fat is your probabilistic model checker? An experimental performance comparison," in *Proc. HVC 2007*, ser. LNCS, vol. 4899. Springer 2008, pp. 69–85.

[17] H. L. S. Younes and R. G. Simmons, "Probabilistic verification of discrete event systems using acceptance sampling." in *Proc. CAV 2002*, ser. LNCS, vol. 2404. Springer, 2002, pp. 223–235.

[18] H. L. S. Younes, "Ymer: A statistical model checker." in *Proc. CAV 2005*, ser. LNCS, vol. 3576. Springer, 2005, pp. 429–433.

[19] Younes, H. L. S., "Probabilistic verification for "black-box" systems." in *Proc. CAV 2005*, ser. LNCS, vol. 3576. Springer, 2005, pp. 253–265.

[20] K. Sen, M. Viswanathan, and G. Agha, "On statistical model checking of stochastic systems." in *Proc. CAV 2005*, ser. LNCS, vol. 3576. Springer, 2005, pp. 266–280.

[21] H. L. S. Younes, M. Z. Kwiatkowska, G. Norman, and D. Parker, "Numerical vs. statistical probabilistic model checking." *STTT*, vol. 8, no. 3, pp. 216–228, 2006.

[22] E. M. Clarke, A. Donzé, and A. Legay, "On simulation-based probabilistic model checking of mixed-analog circuits." *Formal Methods in System Design*, vol. 36, no. 2, pp. 97–113, 2010.

[23] A. David, K. G. Larsen, A. Legay, M. Mikučionis, and Z. Wang, "Time for statistical model checking of real-time systems," in *Proc. CAV 2011*, ser. LNCS, vol. 6806. Springer, 2011, pp. 349–355.

[24] S. Tripakis, C. Sofronis, P. Caspi, and A. Curic, "Translating discrete-time Simulink to Lustre," *ACM Trans. Emb. Comp. Syst.*, vol. 4, no. 4, pp. 779–818, 2005.

[25] B. Meenakshi, A. Bhatnagar, and S. Roy, "Tool for translating Simulink models into input language of a model checker," in *Proc. ICFEM 2006*, 2006, pp. 606–620.

[26] M. Whalen, D. Cofer, S. Miller, B. Krogh, and W. Storm, "Integration of formal analysis into a model-based software development process," in *Proc. FMICS 2007*, 2007, pp. 68–84.