Anytime system level verification via parallel 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

ARTICLE INFO

Keywords:
Model Checking of Hybrid Systems
Model checking driven simulation
Hardware in the loop simulation

ABSTRACT

System level verification of cyber-physical systems 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 verification of actual systems. Thus, Hardware In the Loop Simulation (HILS) is currently the main workhorse for system level verification. By using model checking driven exhaustive HILS, System Level Formal Verification (SLFV) can be effectively carried out for actual systems.

We present a parallel random exhaustive HILS 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 SLFV of the Inverted Pendulum on a Cart and the Fuel Control System examples 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.

© 2015 Elsevier B.V. All rights reserved.

1. Introduction

The cost for fixing a design error in a system becomes larger and larger as the design proceeds from the requirement analysis to the implementation (see, e.g., [2, Chapter 1]) since the later in the design phase an error is detected the more reworking it may trigger. The above observation has motivated the development of methods and tools to verify correctness of a system already in the early phases of its design, namely during the requirement analysis or during the functional specification phases. The goal of all such approaches is to catch design errors well before the system implementation begins.

Of course, all such approaches are model based, that is they work on a model describing the system behaviour since no system implementation exists in the early design phases. Accordingly, System Verification is carried out by simulating a system model and analysing its behaviour under a suitable set of simulation scenarios.

For example, in a digital hardware setting, model based approaches have been used since a long time. In fact, even before considering going to silicon, a heavy simulation activity is performed, aimed at verifying that the system model (defined, e.g., using Verilog, VHDL or SystemC [1]) meets the system requirements for most (possibly all) uncontrollable inputs (that is, primary inputs and faults the system is expected to withstand).

Along the same line of reasoning, in a purely software setting, before generating low level code, model based approaches are used to verify that the software model (defined, e.g., using AADL [3, 4]) meets the given requirements.

If all possible simulation scenarios are considered, then we can prove correctness of the system (i.e., absence of simulation scenarios violating the system requirements), otherwise we can only show that the system design is faulty (by exhibiting a simulation scenario violating the system requirements). In other words, a simulation campaign that does not consider all possible simulation scenarios can only show that the system design has a bug. To show correctness of the system design we need an exhaustive simulation campaign, that is one considering all possible simulation scenarios. A verification approach able to show system correctness is usually referred to as formal verification. One of the most successful techniques to carry out formal verification is Model Checking (see, e.g., [5]).

∗ This paper is an extended and revised version of [1].
∗ Corresponding author.

E-mail addresses: tmancini@di.uniroma1.it (T. Mancini), mari@di.uniroma1.it (F. Mari), massini@di.uniroma1.it (A. Massini), melatti@di.uniroma1.it (I. Melatti), tronci@di.uniroma1.it (E. Tronci).

http://dx.doi.org/10.1016/j.micpro.2015.10.010
0141-9331/© 2015 Elsevier B.V. All rights reserved.
The need for model checking stems from the high cost that a bug may have for certain systems. This is the case for mission critical systems, that is, systems for which a system malfunctioning may entail loss of money, as well as for safety critical systems, that is, systems for which a system malfunctioning may entail loss of human lives. Examples of mission critical systems are: decision support systems, satellites, processors (e.g., the infamous P5 FDI/bug costed about $475 million to INTEL). Examples of safety critical systems are: railway interlocking, avionics control software.

Many Cyber-Physical Systems (CPSs) are indeed mission or safety critical systems. Accordingly, in this paper we focus on formal verification techniques for CPSs.

A CPS consists of hardware (e.g., motors, electrical circuits, etc.) and software components. Thus, in order to verify a CPS design, we need methods and tools that can model and effectively support simulation of hardware as well as software components.

From a formal point of view, CPSs can be modelled as hybrid systems (see, e.g., [6] and citations thereof). Many Model Based Design software tools offer support for modelling and simulation of CPSs. Well known examples are Simulink, VisSim and Modelica. All such tools take as input a (mathematical) model of the behaviour of the CPS along with a simulation scenario and provide as output the time evolution (trace or simulation run) of the system at hand.

System Level Verification of CPSs aims at verifying that the whole (i.e., software + hardware) system meets the given specifications. System Level Formal Verification (SLFV) has the goal of exhaustively verifying that the above holds for all possible operational scenarios.

For digital circuits, formal verification is usually carried out using model checking techniques (e.g., see [7]). Unfortunately, model checkers for hybrid systems cannot handle SLFV of real world CPSs. Thus, HILS is currently the main workhorse for system level verification of CPSs, and is supported by model based design tools (e.g., the previously mentioned Simulink, VisSim and Modelica).

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. Notwithstanding the word hardware, in HILS the only hardware present is the one devoted to support the system simulation, that is: computational and communication devices. This is because HILS is used in a model based design setting to validate the system design before any hardware is built (the whole goal of model based design). For example, Simulink, VisSim, Modelica, ESA Satellite Simulation Infrastructure SIMULUS all provide simulation software supporting HILS, where the only hardware involved is just the computer on which the simulator is actually running.

Simulation can be very time consuming. Accordingly, in order to reduce system design time, Opal-RT and dSpace among others provide modelling and simulation software along with FPGA-based hardware to support real-time simulation. We note that in all cases the only hardware present in HILS is the one supporting the simulation itself.

1.1. 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. Examples of such systems are the Inverted Pendulum on a Cart (IPC) and the Fuel Control System (FCS) in the Simulink distribution (see Section 6.1.1).

Considering that parallel exhaustive HILS based SLFV may take days of computation (e.g., see [8]), 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 anytime, to support graceful degradation.

The work in [9] 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 while verifying operational scenarios designed by experts. As a result, formal verification addresses the search of errors that we are unlikely to consider. For this reason, we can model the problem as an adversary system, that is a system 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. In fact, the adversary would choose to place the single error in 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 [10] 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.

1.2. Main contribution

Our System Under Verification (SUV) is a Hybrid System (e.g., see [6] 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 \( \tau \) (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 step \( \tau \) 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 contribution can be summarised as follows. We present an anytime parallel random exhaustive HILS based model checker that effectively conjugates exhaustiveness with randomness, thereby enabling the computation of the Omission Probability.

While a simulation run for digital hardware takes order of milliseconds on a normal desktop computer, in our setting a simulation run takes order of seconds since it entails heavy numerical computations aimed at solving a system of many Ordinary Differential Equations (modelling the hardware components of the CPS). Indeed (see Section 6) simulation of operational scenarios takes almost 100% of the overall verification time. Resting on such observation we build on the SLFV approach discussed in [8]. In particular:

1. From the disturbance model we generate all admissible simulation scenarios and evenly split them into disjoint sets (slices).
2. 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.
3. 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 case study).

We also show that, thanks to the fact that we have first generated all admissible simulation scenarios, attaining point 2 above (i.e., anytime OP computation) can be done in a not-too-complicated (from both technical and computational points of view) and non-invasive way, by simply introducing a new module into the workflow of [8] that seamlessly interoperates with the others.

1.3. Summary of experimental results

We implemented our approach and present (Section 6) experimental results on two case studies, namely the Inverted Pendulum on a Cart (IPC) and the Fuel Control System (FCS) examples in the Simulink distribution. Overall, we compute optimised simulation campaigns under four operational environments (disturbance models), which entail from 3 208 276 to 35 641 501 simulation scenarios.

Each processor (actually, a core of a 8-core machine) runs an instance of our (random) simulation campaign computation 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 approach takes negligible time to generate an optimised simulation campaign for a given slice, with respect to the time needed to actually execute the simulation campaign (e.g., minutes vs. hours, see Section 6).

Our experimental results show that, by exploiting parallelism, our random exhaustive simulation campaigns effectively counteract the simulation time overhead due to randomisation. The above ensures feasibility of our parallel random exhaustive approach for actual systems, such as the Inverted Pendulum on a Cart (IPC) and Fuel Control System (FCS) examples in the Simulink distribution.

As for the Omission Probability (OP), the worst case scenario is when just one error trace is present. Our experimental results (Section 6.6) show that, even in such a case, our upper bound to the OP decreases about linearly with respect to the coverage, that is the fraction of scenarios simulated. This is the best one can hope to obtain 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 the error in the completion time estimation decreases quickly with respect to coverage.

1.4. Paper outline

Section 2 gives background notions to make our paper self-contained. Section 3 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 4 outlines our algorithm which enables the computation, 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. Section 5 is devoted to the formal proofs of our results. Finally, Section 6 presents experimental results assessing effectiveness of our approach.

2. Background and preliminaries

In this section we give some background notions. Unless otherwise stated, all definitions are based on [8,11,12]. Throughout the paper, we use \( \mathbb{R}^\geq \) for the set of non-negative reals, \( \mathbb{R}^+ \) for the set of strictly positive reals, and \( \text{Bool} = \{0,1\} \) for the set of Boolean values (0 for false and 1 for true). \( \mathbb{N}^+ \) denotes the set of positive natural numbers.

2.1. 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 [11] whose inputs are discrete event sequences. A discrete event sequence (Definition 1) is a function \( u(t) \) associating to each (continuous) time instant \( t \in \mathbb{R}^\geq \) a disturbance event (or, simply, disturbance). Disturbances, encoded by natural numbers 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).

Definition 1 (Discrete event sequence). Let \( d \in \mathbb{N}^+ \). A discrete event sequence over the natural interval \([0,d]\) is a
Assume that \( \omega \) is a finite sequence \( \omega \in U \) such that \( \omega(t) \neq 0 \) has finite cardinality. We denote with \( U_d \) the set of discrete event sequences over \([0, d]\) (following control engineering notation for input functions to dynamical systems, see e.g. [11]).

In Definitions 2 and 3 we specify the concepts of restriction and concatenation, respectively, for functions belonging to \( U_d \).

**Definition 2.** Let \( U_d \) be the set of discrete event sequences over the interval \([0, d]\). Given a function \( u \in U_d \) and two real numbers \( 0 \leq t_1 < t_2 \), we denote with \( u(t_1, t_2) \) the function \( u(t_1, t_2) : [t_1, t_2) \rightarrow [0, d] \) such that \( u(t_1, t_2) = u(t) \) for all \( t \in [t_1, t_2) \). We denote \( U_d(t_1, t_2) \) the restriction of \( U_d \) to the domain \([t_1, t_2)\).

**Definition 3.** Assume that \( t_1, t_2, t_3 \in \mathbb{R}^0 \) such that \( t_1 < t_2 < t_3 \). If \( \omega \in U_d(t_1, t_2) \) and \( \omega' \in U_d(t_2, t_3) \), their concatenation, denoted as \( \omega \omega' \), is the function \( \tilde{\omega} \in U_d(t_1, t_3) \) defined as:

\[
\tilde{\omega}(t) = \begin{cases} 
\omega(t) & \text{if } t \in [t_1, t_2) \\
\omega'(t) & \text{if } t \in [t_2, t_3) 
\end{cases}
\]

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). In this work we focus on bounded system level verification. As a consequence, we model (Definition 4) the SUV operational environment as the sequence of disturbances our SUV is expected to withstand within a finite time horizon, and we bound the time quantum between two consecutive disturbances.

**Definition 4** (Disturbance trace). Let \( h, d \in \mathbb{N}^+ \). An \((h, d)\) disturbance trace \( \delta \) is a finite sequence \( \delta : [0, h - 1] \rightarrow [0, d] \). Given \( \tau \in \mathbb{R}^+ \) (time quantum), an \((h, d)\) disturbance trace \( \delta \) is uniquely associated to a discrete event sequence \( u_h^\delta \), defined as follows: for all \( t \in [0, h - 1] \), if there exists \( j \in [0, h - 1] \) such that \( t = \tau j \), then \( u_h^\delta(t) = \delta(j) \), else \( u_h^\delta(t) = 0 \) (no disturbance).

Thus, a disturbance trace \( \delta \) defines an operational scenario (namely, \( u_h^\delta \)) 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 \( U_d^\delta = \{ u_0^\delta, \ldots, u_{n-1}^\delta \} \) (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 \( \tau \) 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 illustrates this point.

**Example 1.** Consider a disturbance model consisting of a disturbance (namely, a fault) which is always recovered within 4 s (i.e., 4 seconds). Let us assume that between two consecutive disturbances (faults) there must be at least 5 s and that disturbances can arise only at time steps multiple of \( 1s \) (time quantum). We also assume that the verification time horizon is set to 6 s.

In Fig. 3a we show disturbance traces represented as strings of zeros (no disturbance) and ones (disturbance), with time flowing from left to right. The 8 strings terminated by \( \checkmark \) denote all the disturbance traces accepted by the disturbance model (admissible

\[
\begin{array}{cccc}
000000 & 010000 & \checkmark \\
000001 & 010001 & \checkmark \\
000010 & 010101 & \checkmark \\
000011 & 011011 & \checkmark \\
000100 & 011001 & \checkmark \\
000101 & 100000 & \checkmark \\
000110 & 100010 & \checkmark \\
000111 & 101011 & \checkmark \\
001000 & 110011 & \checkmark \\
001001 & 111010 & \checkmark \\
001010 & 111101 & \checkmark \\
001100 & 111110 & \checkmark \\
001101 & 111111 & \checkmark \\
001110 & & \checkmark \\
001111 & & \checkmark \\
\end{array}
\]

(a) Admissible disturbance traces (\( \checkmark \)) and shortest disturbance sequences that cannot be extended to an admissible disturbance trace (\( \otimes \))
disturbance traces). The 14 strings terminated by \( \infty \) are the shortest non-admissible sequences of disturbances, that is disturbance sequences that cannot be extended to admissible disturbance traces.

Fig. 3b shows the pseudo-code for a finite state automaton recognizing 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 [13]), along the lines of [8].

2.2. Modelling the property to be verified

Along the lines of [14], 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. Figs. 8 and 9 show the Simulink/Stateflow representations of our two case studies (Inverted Pendulum on a Cart and Fuel Control System, respectively), along with their property monitors (see Section 6).

2.3. 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 5 formalises an \((h, d)\) MDES as a function \( \mathcal{H} \) associating, to each \((h, d)\) disturbance trace \( \delta \), a Boolean value \( \mathcal{H}(\delta) \) representing the output of the SUV monitor at time \( T = h \tau \) (the time horizon), when the system (starting from its initial state) is given as input the discrete event sequence \( u^T_\delta(t) \) associated to \( \delta \). For any disturbance trace \( \delta \), \( \mathcal{H}(\delta) = 1 \) (error) if and only if \( u^T_\delta(t) \) violates the property under verification within time horizon \( T = h \tau \) (with the SUV starting from its initial state).

Definition 5. \((h, d)\) Monitored DES Let \( h, d \in \mathbb{N}^+ \). A \((h, d)\) Monitored Discrete Event System (MDES) is a function \( \mathcal{H} : (\{0, h - 1\} \rightarrow \{0, d\}) \rightarrow \text{Bool} \) mapping all \((h, d)\) disturbance traces to Boolean values.

2.4. System Level Formal Verification (SLFV)

Definition 6 formalises our bounded System Level Formal Verification problem.

Definition 6. A System Level Formal Verification (SLFV) problem is a tuple \( P = (h, d, \Delta, \mathcal{H}) \) where: \( h, d \in \mathbb{N}^+ \), \( \Delta = \{\delta_0, \ldots, \delta_{m-1}\} \) is an \((h, d)\) set of disturbance traces, and \( \mathcal{H} \) is a \((h, d)\) MDES.

The answer to SLFV problem \( P \) is \( \text{FAIL} \) if there exists a disturbance trace \( \delta \) in \( \Delta \) such that \( \mathcal{H}(\delta) = 1 \) (in such a case also the counterexample \( \delta \) is returned), \( \text{PASS} \) 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 \( T = h \tau \), and (b) the set of time points at which disturbances can take place, by taking \( \tau \) as the time quantum among disturbance events.

2.5. Parallel HILS based deterministic SLFV

In the black-box parallel approach shown in [8], 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). The answer to a SLFV problem \( P = (h, d, \Delta, \mathcal{H}) \) is computed by simulating each operational scenario \( \delta \) in the operational environment \( \Delta \), thus by performing an exhaustive (with respect to \( \Delta \)) Hardware In the Loop Simulation (HILS). The overall workflow is shown in Fig. 4 and described in the remainder of this section. We will refer to this approach as Deterministic SLFV (dSLFV), where the word “deterministic” stems from the fact that the disturbance traces are verified in a deterministic order.
2.5.1. Disturbance trace generation and splitting

Our CMurphi-based trace generator (see Section 2.1 and Fig. 4a) works in Depth-First Search (DFS) mode, and, given the set of disturbances, produces a sequence $\Delta$ of $n$ disturbance traces. Each generated trace $\delta$ in $\Delta$ is annotated with labels and is of the form $\delta = (l_0, d_0, l_1, d_1, ..., l_{n-1}, d_{n-1})$, where $\delta = (d_0, ..., d_k)$ is a sequence of disturbances satisfying the disturbance model and $l_0, ..., l_k$ belong to a countable set $L$. Labels are defined by an injective map $\lambda$ from finite disturbances (including the empty sequence) to $L$. Label $l_0$ is common to all traces and it is associated to the simulator initial state. Prefixes of disturbance sequences $(d_0, ..., d_{k-1})$ common to multiple disturbance traces in $\Delta$ are followed by the same label $l_k$. Fig. 5a shows a sequence of labelled disturbance traces.

Labels identifying common disturbance prefixes are essential in the efficient computation of optimised simulation campaigns. Note that, given our CMurphi-based generator runs in DFS mode, disturbance traces can be labelled at no additional computational cost during generation. Trace labelling during generation greatly increases the efficiency of the simulation campaign optimiser, as shown in [8].

In the following, we will use $δ^k$ instead of $δ$ (respectively, $Δ^k$ instead of $Δ$) when we want to emphasise that a trace $δ$ is annotated (respectively, all traces in set $Δ$ are annotated) with labels, or when we need such labels. In order to enable parallel verification via $k \in \mathbb{N}^+$ processes, we evenly partition the sequence of labelled disturbance traces $Δ^k$ into $k \in \mathbb{N}^+$ sequences of disturbance traces $Δ^k_0, ..., Δ^k_{k-1}$ (called slices).

The splitting process produces slices containing $|n/k|$ traces each, except the last slice, which may contain fewer traces if $n$ is not a multiple of $k$.

2.5.2. Computation of optimised simulation campaigns

In the next phase of the workflow in Fig. 4, we use the $k$ slices $Δ^k_0, ..., Δ^k_{k-1}$ generated so far to compute, independently and in parallel, $k$ highly optimised simulation campaigns (Fig. 4b). Such simulation campaigns can be simulated, again independently and in parallel, using $k$ simulators, each one running (e.g., on a different core of a bunch of multi-core machines) a model for $H$ (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.

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.

The simulation campaign $χ_i$ ($0 \leq i < k$) computed from input slice $Δ^k_i$ steers the simulator as to execute all disturbance traces in $Δ^k_i$ according to their order in the slice (this is the reason why we refer to this approach as Deterministic SLFV).

Each disturbance trace is executed as if starting from the simulator initial state. However, by using commands store and load, the optimiser avoids revisiting simulation states as much as possible (as in explicit model checking). Using command free the optimiser removes from the simulator 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 studies presented in this paper).

Fig. 5b shows the optimised simulation campaign computed from the sequence of labelled disturbance traces in Fig. 5a, in the simple case where we ignore any limit on number of states that can be kept simultaneously stored in the simulator memory.

3. Omission Probability

This section formally defines the notion of Omission Probability (OP) (Definitions 7 and 8) 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).

Definition 7 (Random Sequence Generator). Let $Δ = [δ_0, ..., δ_{n-1}]$ be a finite non-empty set. We denote with $\text{Perm}(Δ)$ the set of permutations of elements of $Δ$:

$$\text{Perm}(Δ) = \{ (θ_0, ..., θ_{n-1}) | (\forall i \in [0, n-1] \ θ_i \in Δ)$$

$$\land (\forall i, j \in [0, n-1] i \neq j \implies θ_i \neq θ_j) \}$$

If $Δ = (δ_0, ..., δ_{n-1}) \in \text{Perm}(Δ)$ we write also $Δ(i)$ for $δ_i$.

A Random Sequence Generator (RSG) models the extraction of a random permutation from a given finite non-empty set (which, in our case, will be the set of admissible disturbance traces $Δ$). This is formalised in Definition 7.

Definition 7 (Random Sequence Generator). Let $Δ$ be a finite non-empty set. A Random Sequence Generator (RSG) for $Δ$ is a probability space $(Ω, F, Pr)$, where:

- $Ω$ (the space of outcomes) is the set of permutations of $Δ$, that is $Ω = \text{Perm}(Δ)$.
- $F$ (the space of events) is the set of subsets of $Ω$, that is: $F = \{ E | E \subseteq Ω \}$.
- $Pr : F \rightarrow [0, 1]$ is a probability measure such that, for all $ω \in Ω$, $Pr(ω) = \frac{1}{|Ω|}$. That is, permutations of $Δ$ are extracted with uniform probability. Since $Ω$ is countable (actually finite), the probability of any event $E \in F$ is defined as $Pr(E) = \sum_{ω \in E} Pr(ω)$.

Let $(Δ_0, ..., Δ_{k-1})$ be a partition of $Δ$ into $k \in \mathbb{N}^+$ disjoint non-empty sets. For any $0 \leq i < k$, let $(Ω_i, F_i, Pr_i)$ be a RSG for $Δ_i$. A Random Sequence Generator for $(Δ_0, ..., Δ_{k-1})$ is a...
using the P in Section 5.

\[ \text{Definition 8 (Omission Probability)} \]

This probability as Omission Probability (OP).

\[ \text{Definition 8} \]

Let \((h, d, \Delta, \mathcal{H})\) be a System Level Formal Verification (SLVF) problem and \((\Delta_0, \ldots, \Delta_{k-1})\) be a partition of \(\Delta\) into \(k \in \mathbb{N}^+\) disjoint non-empty sets. Let \((\Omega, \mathcal{F}, \mathcal{P})\) be a 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 \((q_0, \ldots, q_{k-1})\) is defined as:

\[ \text{OP}_\mathcal{H}(|\Delta_0|, \ldots, |\Delta_{k-1}|, q_0, \ldots, q_{k-1}) = \Pr \left( \left\{ w_i \in [0, k-1] \mid \omega_i \in \Omega_i \wedge A \left( (\omega_0, \ldots, \omega_{k-1}), (q_0, \ldots, q_{k-1}) \right) \right\} \right) \]

where:

- \(A\) is defined as \(A((\omega_0, \ldots, \omega_{k-1}), (q_0, \ldots, q_{k-1})) = B((\omega_0, \ldots, \omega_{k-1}), (q_0, \ldots, q_{k-1})) \wedge B_1((\omega_0, \ldots, \omega_{k-1}), (q_0, \ldots, q_{k-1})) \)
- \(B_1\) (Before) is:
  \[ B_1((\omega_0, \ldots, \omega_{k-1}), (q_0, \ldots, q_{k-1})) = \forall j \in [0, k-1] \mid \forall j \in [0, q_i - 1] \mid \mathcal{H}(\omega_i(j)) = 0 \]
- \(B\) (After) is:
  \[ B((\omega_0, \ldots, \omega_{k-1}), (q_0, \ldots, q_{k-1})) = \exists j \in [0, k-1] \mid \exists j \in [q_i, |\Delta_i|] \mid \mathcal{H}(\omega_i(j)) = 1 \]

In Definition 8, formula \(A\) (After) states that there exists a yet-to-be-simulated trace \(\hat{\sigma}\) (some trace \(j \geq q_i\) of some slice \(i\) containing an error, i.e., such that \(\mathcal{H}(\hat{\sigma})\) 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 gives an upper bound to the OP, after having simulated \(q_i\) randomly extracted traces from slice \(\Delta_i\) (for each \(0 \leq i < k\)). In particular, Theorem 1 provides a bound to the probability of omitting the simulation of a trace \(\hat{\sigma}\) in \(\Delta\) containing an error when the verification process has already examined \(q_i\) disturbance traces from the (the permutation of) slice \(\Delta_i\) (for all \(0 \leq i < k\)). Importantly, the bound provided does not depend on \(\mathcal{H}\), i.e., it is independent of the system model. The proof of the theorem is in Section 5.

\[ \text{Theorem 1} \]

Let \((h, d, \Delta, \mathcal{H})\) be a SLVF problem and \((\Delta_0, \ldots, \Delta_{k-1})\) be a partition of \(\Delta\) into \(k \in \mathbb{N}^+\) disjoint non-empty sets. Let \((\Omega, \mathcal{F}, \mathcal{P})\) be a Random Sequence Generator 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:

\[ \text{OP}_\mathcal{H}(|\Delta_0|, \ldots, |\Delta_{k-1}|, q_0, \ldots, q_{k-1}) \leq 1 - \min \left\{ \frac{q_i}{|\Delta|} \mid 0 \leq i < k \right\} \]

(1)

Note that the construction of the slices \(\Delta_0, \ldots, \Delta_{k-1}\) from \(\Delta\) is non-deterministic (i.e., any partitioning of \(\Delta\) 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 right member of the inequality in Theorem 1.

Finally, we observe that, from Theorem 1, it follows that \(\text{OP}_\mathcal{H}(|\Delta_1|, \ldots, |\Delta_{k-1}|, |\Delta_0|, \ldots, |\Delta_{k-1}|) = 0\), that is, our verification task terminates after max \(|\Delta_i| | 0 \leq i < k\) parallel steps, having simulated all traces in \(\Delta\).

4. Enabling Omission Probability computation: random exhaustive SLVF

Our disturbance trace generator (see Section 2.1) produces a sequence of disturbance traces \(\Delta^k\), whose order is deterministically chosen by the employed algorithm (in our case, Depth-First Search (DFS)). As a consequence, no information about the Omission Probability (OP) can be inferred during the verification process if the simulation campaign computed from each slice verifies the sequence of disturbance traces therein according to their order (as done by the Deterministic SLFV (dSLFV) approach of Section 2.5), or according to any deterministic order, as argued in Section 1.1.

From Section 3 it follows that it is possible to enable OP computation (and hence, graceful degradation during the computationally expensive simulation phase) by simulating the disturbance traces within each slice in an order uniformly chosen at random.

Here we show how we can add support to OP computation in the workflow of Fig. 4 in a not-too-complicated and non-invasive way, by introducing an additional step, in the parallel portion of the workflow, which implements a Random Sequence Generator (RSG).

The new workflow, which we refer to as random exhaustive SLFV (rSLFV), is shown in Fig. 6. All slices are given as input, in parallel, to instances of the new Random Sequence Generator module, shown as Algorithm 1. The Random Sequence Generator module reads a sequence of disturbance traces and computes a random permutation of it, uniformly chosen among all possible permutations, thus implementing a RSG (Definition 7). We argue that the introduction of the new Random Sequence Generator module is not-too-complicated for what concerns both its implementation (Algorithm 1) and its computational viability (see Section 6.3). It is also non-invasive, as it seamlessly interoperates with the dSLFV workflow of Section 2.5.

As the input sequence can be too large to be kept in main memory, the Random Sequence Generator module implements a disk-based multi-round algorithm (shown as Algorithm 1) 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 \(n = |\Delta^k|\) be the number of disturbance traces in the input sequence. Given parameter \(z \in \mathbb{N}^+\) for the maximum number of disturbance traces which can be simultaneously stored in main memory, the algorithm, at each round \(r \geq 1\), selects \(z\) traces which will have output positions in the interval \([r-1]z, \min(n, rz - 1)\).

The above selection is performed by computing the first \(z\) elements of a random permutation of the traces not yet in the output file \(\Delta^k_{\text{rd}}\) chosen uniformly among all possible permutations. Such a permutation prefix is computed by function \(\text{rdPermPrefix}()\) (Algorithm 1, from line 20). Function \(\text{rdPermPrefix}()\) performs a swap-based computation of a permutation of integers \([0, |\Delta^k_{\text{rd}}| - 1]\) (uniformly chosen at random) and interrupts the computation as soon as the first \(z\) elements (see variable \(\text{result}\)) have been determined. During its operation, function \(\text{rdPermPrefix}()\) represents the partially computed permutation as an associative array \(\text{perm}\), which, at any step of the function execution, represents...
permutation $\pi$ of $[0, n - 1]$ such that, for each $j \in [0, n - 1]$: 

$$\pi(j) = \begin{cases} v & \text{if } (j, v) \in \text{perm} \\ j & \text{otherwise, i.e., } \exists v \text{ such that } (j, v) \in \text{perm} \end{cases}$$

The main algorithm then appends the $z$ randomly selected traces (as chosen by function $\text{rndPermPrefix()}$) to $\Delta_{\text{ind}}^z$ (according to their output positions), and dumps all the others to a temporary file, which becomes the input of the next round. Algorithm 1 terminates in $\lceil n/z \rceil$ rounds.

The following theorem asserts that Algorithm 1, when applied to a sequence of disturbance traces $\Delta^z$, produces a uniformly chosen permutation over the set of all permutations. The proof of the theorem is in Section 5.

**Theorem 2.** Let $\Delta^z$ be a file containing $n$ disturbance traces. For any $z \geq 1$, Algorithm 1 stores in file $\Delta_{\text{ind}}^z$ a permutation of the traces in $\Delta^z$ extracted with uniform probability among all possible permutations.

The $k$ output randomised slices (computed in parallel by $k$ instances of the Random Sequence Generator algorithm from $k$ input slices) are then given as input to $k$ parallel instances of the simulation campaign computation module described in Section 2.5.2 which compute $k$ highly optimised simulation campaigns (Fig. 6). As each simulation campaign verifies the traces in its input slice according to the their order, the introduction of the Random Sequence Generator module effectively enforces a random order in the disturbance trace verification within each slice, satisfying the requirements stated in Theorem 1 in order to compute, during simulation, an upper bound to the OP.

$\textbf{5. Proof of results}$

**Theorem 1.** Let $(h, d, \Delta, \mathcal{H})$ be a System Level Formal Verification (SLFV) problem and $(\Delta_0, \ldots, \Delta_{k-1})$ a partition of $\Delta$ into $k \in \mathbb{N}^+$ disjoint non-empty sets. Let $(\Omega, \mathcal{F}, \text{Pr})$ be a Random Sequence Generator 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:

$$OP_h(|\Delta_0|, \ldots, |\Delta_{k-1}|, q_0, \ldots, q_{k-1}) \leq 1 - \min \left\{ \frac{q_i}{|\Delta_i|} \mid 0 \leq i < k \right\}$$

(1)

**Proof.** If, for all $\delta \in \Delta$, $\mathcal{H}(\delta) = 0$, then the left member of (1) is $\text{Pr}(\delta) = 0$ and the thesis follows.

Otherwise, let $E$ be a nonempty set containing error traces, that is $E = \{ \delta \mid \delta \in \Delta, \mathcal{H}(\delta) = 1 \} \neq \emptyset$. $OP_h(|\Delta_0|, \ldots, |\Delta_{k-1}|, q_0, \ldots, q_{k-1})$ can be rewritten as (see Definition 8):

$$\Pr\left(\{ (\omega_0, \ldots, \omega_{k-1}) \mid \forall i \in [0, k-1] \omega_i \in \Omega_i \wedge \forall i \in [0, k-1] \forall j \in [0, q_i - 1] \omega_i(j) \notin E \} \right)$$

(2)

because $(\Delta_0, \ldots, \Delta_{k-1})$ is a partition of $\Delta$.

Consider any $\delta \in \Delta$. Probability (2) is less than or equal to

$$\Pr\left(\{ (\omega_0, \ldots, \omega_{k-1}) \mid \forall i \in [0, k-1] \omega_i \in \Omega_i \wedge \forall i \in [0, k-1] \forall j \in [0, q_i - 1] \omega_i(j) \neq \delta \} \right)$$

(3)

Given that $\delta$ belongs to exactly one of $\Delta_0, \ldots, \Delta_{k-1}$, say $\Delta_i$, and given the definition of $\text{Pr}(\omega_0, \ldots, \omega_{k-1})$ in Definition 7, expression (3) is equal to:

$$\prod_{i=0}^{k-1} \Pr\left(\{ \omega_i \in \Omega_i \mid \forall j \in [0, q_i - 1] \omega_i(j) \neq \delta \} \right)$$

which is equal to

$$\Pr\left(\{ \omega_i \in \Omega_i \mid \forall j \in [0, q_i - 1] \omega_i(j) \neq \delta \} \right)$$

as, for each $i \neq j$ ($0 \leq i < k$), $\text{Pr}(\{ \omega_i \in \Omega_i \}) = 1$.

Probability (4) is the probability of picking a permutation $\omega_i$ of $\Delta_i$ which does not have $\delta$ in the first $q_i$ positions, and evaluates to $1 - \frac{q_i(|\Delta_i| - 1)}{|\Delta_i|} = 1 - \frac{q_i}{|\Delta_i|}$, which is less than or equal to $1 - \min \left\{ \frac{q_i}{|\Delta_i|} \mid 0 \leq i < k \right\}$. The thesis follows. □

**Theorem 2.** Let $\Delta^z$ be a file containing $n$ disturbance traces. For any $z \geq 1$, Algorithm 1 stores in file $\Delta_{\text{ind}}^z$ a permutation of the traces in $\Delta^z$ extracted with uniform probability among all possible permutations.

**Proof.** We prove the following: (i) all traces in the input sequence (file $\Delta^z$) will occur in the output sequence (file $\Delta_{\text{ind}}^z$) exactly once (i.e., the algorithm computes a permutation); (ii) for each $\delta$ occurring in $\Delta^z$, the probability that $\delta$ occurs in $\Delta_{\text{ind}}^z$ at any position is $\frac{1}{n}$ (i.e., the computed permutation is uniformly extracted among all possible permutations).
Algorithm 1 Random Sequence Generator.

Input: $\Delta^\lambda$, a file holding a labelled sequence of disturbance traces
Output: $\Delta^\lambda_{\text{rand}}$, a file holding a random permutation of disturbance traces, uniformly chosen among all possible permutations

1 param $z$, max # of traces that can be kept in RAM
2 $\Delta^\lambda_{\text{rd}} \leftarrow$ a new empty file;
3 $\Delta^\lambda_{\text{tmp}} \leftarrow \Delta^\lambda_{\text{in}}$ /* input of 1st round */
4 $r \leftarrow 1$; /* round counter */
5 while $\Delta^\lambda_{\text{tmp}}$ is not empty do
6 $\Delta^\lambda_{\text{in}} \leftarrow \Delta^\lambda_{\text{tmp}}$; /* tmp file of prev. round */
7 $\Delta^\lambda_{\text{tmp}} \leftarrow$ a new temp file; /* next round input */
8 if $z > |\Delta^\lambda_{\text{in}}|$ then $z \leftarrow |\Delta^\lambda_{\text{in}}|$;
9 selected_pos $\leftarrow$ rndPermPrefix($|\Delta^\lambda_{\text{in}}|$, $z$); /* selected_pos is a mapping from $[0, z-1]$ to $[0, |\Delta^\lambda_{\text{in}}| - 1]$ */
10 selected_traces $\leftarrow$ empty array of $z$ elements;
11 $p \leftarrow 0$;
12 foreach $\delta \in \Delta^\lambda_{\text{in}}$ do
13 if $\exists$ selected_pos[$i$] = $p$ then selected_traces[$i$] $\leftarrow \delta$;
14 else append trace $\delta$ to $\Delta^\lambda_{\text{tmp}}$;
15 $p \leftarrow p + 1$;
16 for $i$ $\leftarrow 0$ . . . $z-1$ do
17 append trace selected_traces[$i$] to $\Delta^\lambda_{\text{rd}}$;
18 $r \leftarrow r + 1$;
19 return $\Delta^\lambda_{\text{rd}}$;

20 function rndPermPrefix($n$, $z$)
21 Input: $n$, positive integer
22 Input: $z$, positive integer ($z \leq n$)
23 Output: result, the first $z$ elements of a permutation of $[0, n-1]$ uniformly chosen at random
24 result $\leftarrow$ array of $z$ elements;
25 perm $\leftarrow$ empty associative array;
26 for $i$ $\leftarrow 0$ . . . $z-1$ do
27 $j \leftarrow$ random index between $i$ and $n-1$;
28 if exists $v$ s.t. $(i, v) \in$ perm then value.$i$ $\leftarrow$ $v$;
29 else value.$i$ $\leftarrow$ $i$;
30 if exists $v$ s.t. $(j, v) \in$ perm then value.$j$ $\leftarrow$ $v$;
31 else value.$j$ $\leftarrow$ $j$;
32 put $(i, \text{value.$i$})$ into perm;
33 put $(j, \text{value.$j$})$ into perm;
34 result[$i$] $\leftarrow$ value.$i$;
35 return result;
36 end

Point (i) is immediate: at each round, $z$ traces are appended to $\Delta^\lambda_{\text{rd}}$, all the others are appended to $\Delta^\lambda_{\text{tmp}}$, which becomes the input of the next round. Also, the algorithm terminates only if the $\Delta^\lambda_{\text{tmp}}$ produced at the previous round is empty.

To prove point (ii), for any $p, q$ in $[0, n-1]$, we compute the probability that trace $\delta$ having position $p$ in the input sequence (file $\Delta^\lambda_{\text{in}}$) will have position $q$ in the output sequence (file $\Delta^\lambda_{\text{rd}}$).

We omit to prove that function $\text{rndPermPrefix}(n,z)$ actually computes the first $z$ elements of a permutation of the integer interval $[0, n-1]$ uniformly selected at random, as the function interrupts a well-known swap-based permutation algorithm as soon as the first $z$ elements have been determined.

Given that, at each round $r \geq 1$, the main algorithm selects the $z$ input traces which will have output positions $(r-1)z$ to min$(n, rz-1)$. $\delta$ is selected only at round $r_\delta = \lceil (q + 1)/z \rceil$. Thus, the probability that $\delta$, having input position $p$, will have output position $q$ is:

$$\Pr \left( \bigcap_{r=1}^{r_\delta-1} \neg E_r \cap O^{q'}_{r_\delta} \right) \quad (5)$$

where, for all $r$, $\neg E_r$ denotes the event “$\delta$ is not selected at round $r$” and $O^{q'}_{r_\delta}$ denotes the event “$\delta$ is the $q'$th trace selected in round $r_\delta$,” where $q' = (q - z(r_\delta - 1))$.

By the chain rule, (5) becomes:
Pr (−E₁) × · · · × Pr (−Eₙ₋₁|−Eₙ₋₂, · · · −E₁) (6)

× Pr (−Eₙ₋₁|−Eₙ₋₂, · · · −E₁).

For all 1 ≤ r ≤ rₙ − 1:

Pr (−E_r|−E_r₋₁, · · · −E₁) = \frac{n − z(r − 1) − 1}{n − z(r − 1)} × · · · × \frac{n − z(r − 1) − z}{n − z(r − 1) − (z − 1)}

(7)

where, for all 0 ≤ i < z, the ith factor is the number of ways we can choose a trace different from δ out of n − z(r − 1) − i (where n − z(r − 1) is the number of traces still not selected at the beginning of round r). The expression simplifies to:

Pr (−E_r|−E_r₋₁, · · · −E₁) = \frac{n − zr}{n − z(r − 1)}

and the product Pr (−E₁) × · · · × Pr (−Eₙ₋₁|−Eₙ₋₂, · · · −E₁) is:

Pr (−E₁) × · · · × Pr (−Eₙ₋₁|−Eₙ₋₂, · · · −E₁) = \prod_{r=1}^{rₙ} \frac{n − zr}{n − z(r − 1)} = \frac{n − z(rₙ − 1)}{n}.

As for Pr (Qᵢ|−Eₙ₋₁, · · · −E₁), i.e., the probability that δ is the qᵢth selected trace in round rₙ provided that it has not been selected in previous rounds, it can be computed as:

Pr (−E₁) × · · · × Pr (−Eₙ₋₁|−Eₙ₋₂, · · · −E₁) = \frac{n − z(rₙ − 1)}{n − z(rₙ − 1) − (qᵢ − 1)} × \frac{n − z(rₙ − 1) − (qᵢ − 1)}{n − z(rₙ − 1) − (qᵢ − 2)} × \frac{1}{n − z(rₙ − 1)} = \frac{1}{n − z(rₙ − 1)} (8)

where, for all 0 ≤ i < qᵢ − 1, the ith factor is the number of ways we can choose a trace different from δ out of n − z(rₙ − 1) − i (where n − z(rₙ − 1) is the number of traces still not selected at the beginning of round rₙ). The last factor is the probability of selecting δ (still unselected) out of n − z(rₙ − 1) − (qᵢ − 1) traces.

By (7) and (8), probability (6) evaluates to \frac{1}{4}, which is independent of p and q. □

6. Experimental results

In this section we evaluate the effectiveness of our random exhaustive SLFV (rSLFV) approach 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 SLFV (dSLFV) approach of [8]. 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.

6.1. Experimental setting

In this section we describe the case studies and the computational infrastructure we used in our experiments.

6.1.1. Case studies

We experiment with two case studies, using two models included in the Simulink distribution, namely the Inverted Pendulum on a Cart (IPC) and the Fuel Control System (FCS).

6.1.1.1. Inverted Pendulum on a Cart (IPC). The IPC is a control loop system where the controlled system is an inverted pendulum installed on a cart (see Fig. 7). The controller (actually a control software) senses the angular position θ of the pendulum, and computes the force F to be applied to the cart to move it left or right along the x axis. The goal is to keep the pendulum in its upright (vertical) unstable position. The physical constraint between the cart and the pendulum gives that both the cart and the pendulum have one degree of freedom each (x and θ, respectively).

The controlled system consists of the cart and the pendulum, whereas the controller consists of the control software computing F from the plant outputs (x and θ). Accordingly, our overall System Under Verification (SUV) model consists of the controlled system and the controller, whose Simulink block diagram is shown in the upper box of Fig. 8.

The system level property that we verify is that after 2 s the pendulum is in upright position, i.e., angle θ is always between [−0.1, 0.1]. The monitor checking for this property is shown in the lower box of Fig. 8.

We introduce disturbances by injecting irregularities in the cart rail. We model such irregularities with a modification on the cart weight m with respect to its nominal value of 0.455 kg. For this, we define three disturbances representing normal rail operation (m = 0.455 kg), abnormal rail operation (m = 1.455 kg), and stressed rail operation (m = 2.455 kg).

We consider two disturbance models for the IPC, DIPC and DIPC. Model DIPC has a horizon of h = 90 and defines 3 208 276 disturbance traces. Model DIPC is defined extending DIPC with more complex operational scenarios and defines 35 641 501 disturbance traces over a horizon of h = 200. For both models we set τ (quantum between disturbances) to 0.1 s. A detailed description of DIPC and DIPC is not relevant for the evaluation of our experiments below. We only observe that, in our setting, the complexity of answering a System Level Formal Verification (SLFV) problem 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 SLFV problem is PASS. Indeed, both DIPC and DIPC satisfy this requirement.

6.1.1.2. Fuel Control System (FCS). The FCS is a controller for a fault tolerant gasoline engine, which has also been used as a case study in [8,12,15–18] in each section.

The FCS has four sensors: throttle angle, speed, EGO (measuring the residual oxygen present in the exhaust gas) and MAP (manifold absolute pressure). The goal of the control system is to maintain the air-fuel ratio (the ratio between the air mass flow rate pumped from the intake manifold and the fuel mass flow rate injected at the valves) close to the stoichiometric ratio of 14.6, which represents a good compromise between power, fuel economy, and emissions.

From the sensor measurements, the FCS estimates the mixture ratio and provides feedback to the closed-loop control, yielding an increase or a decrease of the fuel rate.

The FCS sensors are subject to faults (disturbances), and the whole control system can tolerate single sensor faults. In particular, if a sensor fault is detected, the FCS changes its control law
by operating the engine with a higher fuel rate to compensate. In case two or more sensors fail, the FCS shuts down the engine, as the air-fuel ratio cannot be controlled.

The control logic of the FCS is implemented by six automata, each one with a number of states ranging from two to five. The signal flow is further subdivided into three subsystems, which exhibit several different Simulink block types, involving arithmetic, lookup tables, integrators, filters and interpolation [19] (see [20] for more details).

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 SUV consists of the Simulink FCS model along with a monitor for the property under verification (such a model is shown as Fig. 9).

We consider two disturbance models for the FCS, $D^1_{\text{FCS}}$ and $D^2_{\text{FCS}}$. Model $D^1_{\text{FCS}}$ has a horizon of $h = 100$ and defines 4,023,955 disturbance traces. Model $D^2_{\text{FCS}}$ is defined extending $D^1_{\text{FCS}}$ with more complex operational scenarios and defines 12,948,712 disturbance traces over a horizon of $h = 200$. For both models we set $\tau$ (quantum between disturbances) to 1 s. A detailed description of $D^1_{\text{FCS}}$ and $D^2_{\text{FCS}}$ is not relevant for the evaluation of our experiments below, and can be found in [8,21]. We only observe that, as it happens with our disturbance models for the IPC, for all disturbance traces entailed by $D^1_{\text{FCS}}$ and $D^2_{\text{FCS}}$, the property to be verified is satisfied. This yields the worst scenario to answer our SLFV problem, as all traces need to be simulated.

### Table 1: Disturbance trace generation.

<table>
<thead>
<tr>
<th>SUV</th>
<th>Dist. model</th>
<th>#Traces</th>
<th>Gen. time</th>
<th>File size</th>
</tr>
</thead>
<tbody>
<tr>
<td>IPC</td>
<td>$D^1_{\text{IPC}}$</td>
<td>3208276</td>
<td>0:9:58</td>
<td>4.6 GB</td>
</tr>
<tr>
<td>IPC</td>
<td>$D^2_{\text{IPC}}$</td>
<td>35641901</td>
<td>7:28:24</td>
<td>107 GB</td>
</tr>
<tr>
<td>FCS</td>
<td>$D^1_{\text{FCS}}$</td>
<td>4023955</td>
<td>0:28:39</td>
<td>3.5 GB</td>
</tr>
<tr>
<td>FCS</td>
<td>$D^2_{\text{FCS}}$</td>
<td>12948712</td>
<td>4:45:47</td>
<td>39 GB</td>
</tr>
</tbody>
</table>

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 [8] as presented in [12]. Given that, in our case studies, 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.

### 6.1.2. Computational infrastructure

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.

### 6.2. Disturbance trace generation and splitting

Along the lines of [8], we use CMurphi to generate the set of labelled admissible disturbance traces entailed by the disturbance model given as input.

Table 1 shows, for each of the four disturbance models we consider (two for IPC and two for FCS) the number of entailed disturbance traces (column #Traces), the time needed by CMurphi to generate them (column Gen. time) and the size of the file computed by our generator to store them (column File size).

We then split the generated sequences of disturbance traces into $k$ slices, with $k = 128, 256, 512$ to enable parallel computation on, respectively, 16, 32, 64 (8-core) machines. Splitting always takes negligible time.
6.3. Computation of optimised simulation campaigns

Table 2 shows, for each of the disturbance models we consider, the time needed by our Random Sequence Generator module to enable anytime OP computation during the verification process (rSLFV) and the time needed to compute optimised simulation campaigns with (rSLFV) and without (dSLFV) trace order randomisation.

Column #Mach gives the number of machines we used in parallel. Column #Slices gives the overall number of slices in which the sequence of admissible disturbance traces has been partitioned (one per available core, i.e., 8 slices per machine). Column #Traces per slice shows the number of traces in any single slice (except the last slice, which has fewer traces when the overall number of traces is not a multiple of #Slices). Column group rSLFV shows the maximum overall time to compute the simulation campaign starting from a slice. This time is split into two parts: the time needed to execute the Random Sequence Generator module (column RSG) and the time needed to compute the simulation campaign starting from the randomised slice (column sim. camp. comp.). Column dSLFV shows the maximum time to compute the simulation campaign starting from a slice when no RSG is performed, thus no anytime OP computation is enabled. Column rSLFV overhead shows the difference between the rSLFV and dSLFV times.

It can be observed that our parallel approach to computation of optimised randomised simulation campaigns is able to effectively exploit parallelism in order to handle disturbance models entailing tens of millions of operational scenarios.

Also, Table 2 shows that the random sequence generation phase makes the rSLFV simulation campaign computation process longer than that for dSLFV (i.e., overhead is positive). The difference is, however, negligible with respect to the whole verification time, which takes many hours, even for the massively parallel simulation of operational scenarios entailed by our smallest disturbance models, i.e., $D_{1}^{\text{IPC}}$ for the IPC and $D_{1}^{\text{FCS}}$ for the FCS (see Section 6.4).

6.4. Execution of the simulation campaigns

Table 3 shows the execution time of the simulation campaigns generated by dSLFV and rSLFV.

As the exhaustive simulation of the traces generated by disturbance models $D_{2}^{\text{IPC}}$, $D_{2}^{\text{FCS}}$ (entailing, respectively, 35 641 501 and 12 948 712 disturbance traces) would require a prohibitively long time, from now on we restrict ourselves to the simulation of the traces generated by $D_{1}^{\text{IPC}}$ and $D_{1}^{\text{FCS}}$ (which entail, respectively, 3 208 276 and 4 023 955 disturbance traces).

By enabling the computation of the OP during the simulation activity we have a quite significant increase of simulation time. However, such an overhead can be drastically reduced, or even neutralised, by using more parallel processes (higher values for $k = #\text{Slices}$). 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 limited (to fit within the 500 MB of disk space available to each simulator instance). For high values of $k = #\text{Slices}$ (e.g., $k = 512$), this is not a big obstacle. On the other hand, for low 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
Table 2
Overhead of enabling OP (rSLFV) with respect to dSLFV in the computation of simulation campaigns (time in hours:minutes:seconds).

<table>
<thead>
<tr>
<th>#Mach.</th>
<th>#Slices</th>
<th>#Traces per slice</th>
<th>rSLFV RSG</th>
<th>rSLFV sim.camp.comp.</th>
<th>dSLFV RSG</th>
<th>dSLFV sim.camp.comp.</th>
<th>Total rSLFV sim.camp.comp. overhead</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>8</td>
<td>401 035</td>
<td>0:0:21</td>
<td>1:8:52</td>
<td>1:9:13</td>
<td>0:8:53</td>
<td>1:0:20</td>
</tr>
<tr>
<td>2</td>
<td>16</td>
<td>200 518</td>
<td>0:0:11</td>
<td>0:11:57</td>
<td>0:12:9</td>
<td>0:4:36</td>
<td>0:7:32</td>
</tr>
<tr>
<td>4</td>
<td>32</td>
<td>100 719</td>
<td>0:0:5</td>
<td>0:6:38</td>
<td>0:6:33</td>
<td>0:2:23</td>
<td>0:4:14</td>
</tr>
<tr>
<td>8</td>
<td>64</td>
<td>50 130</td>
<td>0:0:3</td>
<td>0:2:50</td>
<td>0:2:53</td>
<td>0:1:14</td>
<td>0:1:39</td>
</tr>
<tr>
<td>16</td>
<td>128</td>
<td>25 065</td>
<td>0:0:2</td>
<td>0:1:23</td>
<td>0:1:26</td>
<td>0:0:38</td>
<td>0:0:47</td>
</tr>
<tr>
<td>32</td>
<td>256</td>
<td>12 533</td>
<td>0:0:1</td>
<td>0:1:19</td>
<td>0:2:21</td>
<td>0:0:20</td>
<td>0:0:12</td>
</tr>
<tr>
<td>64</td>
<td>512</td>
<td>6 267</td>
<td>0:0:1</td>
<td>0:2:21</td>
<td>0:2:23</td>
<td>0:0:11</td>
<td>0:0:12</td>
</tr>
</tbody>
</table>

(a) Inverted Pendulum on a Cart (IPC), disturbance model \( D_1 \)

IPC: 3 208 276 traces with horizon \( h = 90 \)

<table>
<thead>
<tr>
<th>#Mach.</th>
<th>#Slices</th>
<th>#Traces per slice</th>
<th>rSLFV RSG</th>
<th>rSLFV sim.camp.comp.</th>
<th>dSLFV RSG</th>
<th>dSLFV sim.camp.comp.</th>
<th>Total rSLFV sim.camp.comp. overhead</th>
</tr>
</thead>
<tbody>
<tr>
<td>16</td>
<td>128</td>
<td>278 450</td>
<td>0:1:40</td>
<td>2:41:28</td>
<td>2:43:8</td>
<td>0:35:0</td>
<td>2:8:8</td>
</tr>
<tr>
<td>32</td>
<td>256</td>
<td>139 225</td>
<td>0:1:6</td>
<td>1:9:12</td>
<td>1:10:18</td>
<td>0:17:40</td>
<td>0:52:38</td>
</tr>
<tr>
<td>64</td>
<td>512</td>
<td>69 613</td>
<td>0:0:34</td>
<td>0:21:26</td>
<td>0:22:0</td>
<td>0:8:57</td>
<td>0:13:3</td>
</tr>
</tbody>
</table>

(b) Inverted Pendulum on a Cart (IPC), disturbance model \( D_2 \)

IPC: 35 641 501 traces with horizon \( h = 200 \)

<table>
<thead>
<tr>
<th>#Mach.</th>
<th>#Slices</th>
<th>#Traces per slice</th>
<th>rSLFV RSG</th>
<th>rSLFV sim.camp.comp.</th>
<th>dSLFV RSG</th>
<th>dSLFV sim.camp.comp.</th>
<th>Total rSLFV sim.camp.comp. overhead</th>
</tr>
</thead>
<tbody>
<tr>
<td>16</td>
<td>128</td>
<td>101 162</td>
<td>0:0:45</td>
<td>0:20:36</td>
<td>0:21:21</td>
<td>0:8:18</td>
<td>0:13:3</td>
</tr>
<tr>
<td>32</td>
<td>256</td>
<td>50 581</td>
<td>0:0:24</td>
<td>0:15:35</td>
<td>0:15:59</td>
<td>0:4:31</td>
<td>0:11:28</td>
</tr>
<tr>
<td>64</td>
<td>512</td>
<td>25 291</td>
<td>0:0:14</td>
<td>0:6:43</td>
<td>0:6:57</td>
<td>0:2:4</td>
<td>0:4:53</td>
</tr>
</tbody>
</table>

(c) Fuel Control System (FCS), disturbance model \( D_1 \)

FCS: 4 023 955 traces with horizon \( h = 100 \)

<table>
<thead>
<tr>
<th>#Mach.</th>
<th>#Slices</th>
<th>#Traces per slice</th>
<th>rSLFV RSG</th>
<th>rSLFV sim.camp.comp.</th>
<th>dSLFV RSG</th>
<th>dSLFV sim.camp.comp.</th>
<th>Total rSLFV sim.camp.comp. overhead</th>
</tr>
</thead>
<tbody>
<tr>
<td>16</td>
<td>128</td>
<td>101 162</td>
<td>0:0:45</td>
<td>0:20:36</td>
<td>0:21:21</td>
<td>0:8:18</td>
<td>0:13:3</td>
</tr>
<tr>
<td>32</td>
<td>256</td>
<td>50 581</td>
<td>0:0:24</td>
<td>0:15:35</td>
<td>0:15:59</td>
<td>0:4:31</td>
<td>0:11:28</td>
</tr>
<tr>
<td>64</td>
<td>512</td>
<td>25 291</td>
<td>0:0:14</td>
<td>0:6:43</td>
<td>0:6:57</td>
<td>0:2:4</td>
<td>0:4:53</td>
</tr>
</tbody>
</table>

(d) Fuel Control System (FCS), disturbance model \( D_2 \)

FCS: 12 948 712 traces with horizon \( h = 200 \)

<table>
<thead>
<tr>
<th>#Mach.</th>
<th>#Slices</th>
<th>#Traces per slice</th>
<th>rSLFV RSG</th>
<th>rSLFV sim.camp.comp.</th>
<th>dSLFV RSG</th>
<th>dSLFV sim.camp.comp.</th>
<th>Total rSLFV sim.camp.comp. overhead</th>
</tr>
</thead>
<tbody>
<tr>
<td>16</td>
<td>128</td>
<td>101 162</td>
<td>0:0:45</td>
<td>0:20:36</td>
<td>0:21:21</td>
<td>0:8:18</td>
<td>0:13:3</td>
</tr>
<tr>
<td>32</td>
<td>256</td>
<td>50 581</td>
<td>0:0:24</td>
<td>0:15:35</td>
<td>0:15:59</td>
<td>0:4:31</td>
<td>0:11:28</td>
</tr>
<tr>
<td>64</td>
<td>512</td>
<td>25 291</td>
<td>0:0:14</td>
<td>0:6:43</td>
<td>0:6:57</td>
<td>0:2:4</td>
<td>0:4:53</td>
</tr>
</tbody>
</table>

Table 3
Parallel execution of simulation campaigns by dSLFV and rSLFV (time in hours:minutes:seconds).

<table>
<thead>
<tr>
<th>#Mach.</th>
<th>#Slices</th>
<th>Min</th>
<th>Max</th>
<th>Avg</th>
<th>Approach</th>
</tr>
</thead>
<tbody>
<tr>
<td>16</td>
<td>128</td>
<td>2:14:2</td>
<td>4:10:52</td>
<td>3:44:52</td>
<td>dSLFV</td>
</tr>
<tr>
<td>25:14:28</td>
<td></td>
<td>23:0:26</td>
<td></td>
<td></td>
<td>rSLFV</td>
</tr>
<tr>
<td>32</td>
<td>256</td>
<td>1:6:49</td>
<td>2:5:40</td>
<td>1:50:28</td>
<td>dSLFV</td>
</tr>
<tr>
<td>4:26:16</td>
<td></td>
<td>3:19:27</td>
<td></td>
<td></td>
<td>rSLFV</td>
</tr>
<tr>
<td>64</td>
<td>512</td>
<td>0:30:30</td>
<td>1:3:10</td>
<td>0:53:45</td>
<td>dSLFV</td>
</tr>
<tr>
<td>0:59:26</td>
<td></td>
<td>0:28:56</td>
<td></td>
<td></td>
<td>rSLFV</td>
</tr>
<tr>
<td>0:48:00</td>
<td>0:49:27</td>
<td>0:49:27</td>
<td>0:49:27</td>
<td>0:49:27</td>
<td>overhead</td>
</tr>
</tbody>
</table>

(a) Inverted Pendulum on a Cart (IPC), disturbance model \( D_1 \)

 IPC: 3 208 276 traces with horizon \( h = 90 \)

(b) Inverted Pendulum on a Cart (IPC), disturbance model \( D_2 \)

 IPC: 35 641 501 traces with horizon \( h = 200 \)

(c) Fuel Control System (FCS), disturbance model \( D_1 \)

 FCS: 4 023 955 traces with horizon \( h = 100 \)

(d) Fuel Control System (FCS), disturbance model \( D_2 \)

 FCS: 12 948 712 traces with horizon \( h = 200 \)

yet-to-be-simulated traces. Such traces will then be simulated from the simulator initial state, thus yielding performance degradation.

6.5. Overall verification time and scalability

In this section we evaluate the overall impact of enabling anytime OP computation and the scalability of our parallel approach to SLFV.

Table 4 shows the overall time needed to carry out our SLFV tasks (IPC with disturbance model \( D_1 \) and FCS with disturbance model \( D_1 \)) with \( k \) parallel processes, for the values of \( k \) already used in the previous tables.

In particular, column Gen. & comp. sim. camp. reports the sum of the disturbance trace generation and splitting time, parallel randomisation (only for rSLFV) and parallel simulation campaign computation time (from Table 2). Column Simulation reports the parallel simulation time when using \( k \) parallel processes (i.e., the maximum simulation time over all the \( k = \#Slices \) slices as in column Max of Table 3). Column Overall reports the overall time to carry out each dSLFV and each rSLFV task, as the sum of the previous two columns.

We observe that our approach takes negligible time to generate optimised simulation campaigns with respect to the time needed to actually simulate them (e.g., minutes vs. hours).

6.5.1. Estimation of sequential verification time

In order to evaluate the scalability of our parallel approach to SLFV, Table 4 also reports (in the first two rows of each sub-table),
the overall dSLFV and rSLFV times when using only one parallel process (sequential time). Unfortunately, as for our case studies a sequential simulation would be prohibitively long, we have estimated the sequential simulation time to carry out both dSLFV and rSLFV as follows.

Let $t_{seq}^k$ be the average time to simulate a slice where $k = \#Slices$ parallel processes are used (row $\#slices = k$, column avg, for either dSLFV or rSLFV). For any value of $k$, the sequential simulation time could be estimated as $k \times t_{seq}^k$. As this value changes a little bit for different values of $k$, Table 4 estimates sequential simulation time as \[ \min\{128t_{seq}^k, 256t_{seq}^k, 512t_{seq}^k\}. \] Such huge values (weeks of computation) make clear that estimation is the only viable way to compute the simulation sequential times. Note that in our computation we are slightly overestimating the sequential 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 a limited number of stored states, this is negligible with respect to the value of the sequential simulation time.

### 6.5.2. Speedup and efficiency

Sequential simulation time for both dSLFV and rSLFV is used in Table 4 to compute the speedup and the efficiency of our parallel approach to SLFV, as typically done in the evaluation of parallel algorithms. In particular, for each $k = \#Slices$, column Speedup shows the ratio $t_1/t_k$, where $t_1$ is the estimated overall sequential verification time and $t_k$ is the overall verification time when $k$ parallel processes are used. Column Efficiency is computed by dividing the speedup by the number of parallel processes $k = \#Slices$.

Table 4 also shows the overhead (see bold values) due to randomisation of the verification task (which is the price to pay in order to enable anytime computation of OP), both in terms of overall verification time increase and in terms of reduction of speedup and efficiency. We observe that such an overhead is significant, but it can be drastically reduced by increasing the number $k$ of parallel processes.

#### 6.6. Omission probability

Figs. 10a and c show how our upper bound to the OP decreases as a function of the coverage (i.e., the ratio of admissible traces simulated) during the parallel execution of the $k$ simulation campaigns (IPC with disturbance model $\mathcal{D}_{IPC}$ and FCS with disturbance model $\mathcal{D}_{FCS}$), 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 (curves named “100%-coverage”, i.e., 100% minus coverage), which is the best one can do (using only one parallel process) without any assumption on the number of error traces.

#### 6.7. Completion time estimation

Figs. 10b and d show that OP bound, computed during the parallel execution of the simulation campaigns (IPC with disturbance model $\mathcal{D}_{IPC}$ and FCS with disturbance model $\mathcal{D}_{FCS}$), 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. 11 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 coverage, the error is computed as

---

**Table 4**

Overall performance overhead (including disturbance trace generation and splitting, trace randomisation, computation of simulation campaigns and Simulink simulations) of rSLFV with respect to dSLFV (time in h:m:s).

<table>
<thead>
<tr>
<th>#Mach.</th>
<th>#Slices</th>
<th>Gen. &amp; comp. sim. camp.</th>
<th>Simulation</th>
<th>Overall</th>
<th>Speedup</th>
<th>Efficiency</th>
<th>Approach</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>0:56:11</td>
<td>458:40:0*</td>
<td>459:36:11*</td>
<td>1.00 × 100.00%</td>
<td>dSLFV</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>880:38:24*</td>
<td>907:38:21*</td>
<td>1.00 × 100.00%</td>
<td>rSLFV</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>+97.48%</td>
<td>+0.00%</td>
<td>+0.00%</td>
<td>overhead</td>
</tr>
<tr>
<td>16</td>
<td>128</td>
<td>0:10:36</td>
<td>4:10:52</td>
<td>4:21:28</td>
<td>105.46 × 82.39%</td>
<td>dSLFV</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>85:10:36</td>
<td>85:22:0</td>
<td>10.63 × 8.30%</td>
<td>rSLFV</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>+1858.83%</td>
<td>+89.92%</td>
<td>+74.09%</td>
<td>overhead</td>
</tr>
<tr>
<td>32</td>
<td>256</td>
<td>0:10:18</td>
<td>2:5:40</td>
<td>2:15:58</td>
<td>202.80 × 79.22%</td>
<td>dSLFV</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>23:59:0</td>
<td>24:10:19</td>
<td>37.55 × 14.67%</td>
<td>rSLFV</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>+966.57%</td>
<td>+81.48%</td>
<td>+64.55%</td>
<td>overhead</td>
</tr>
<tr>
<td>64</td>
<td>512</td>
<td>0:10:9</td>
<td>1:3:10</td>
<td>1:13:19</td>
<td>376.04 × 73.45%</td>
<td>dSLFV</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>1:51:10</td>
<td>2:1:31</td>
<td>448.13 × 87.53%</td>
<td>rSLFV</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>+65.71%</td>
<td>−19.17%</td>
<td>−14.08%</td>
<td>overhead</td>
</tr>
</tbody>
</table>

(a) Inverted Pendulum on a Cart (IPC), disturbance model $\mathcal{D}_{IPC}$

<table>
<thead>
<tr>
<th>#Mach.</th>
<th>#Slices</th>
<th>Gen. &amp; comp. sim. camp.</th>
<th>Simulation</th>
<th>Overall</th>
<th>Speedup</th>
<th>Efficiency</th>
<th>Approach</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>0:35:55</td>
<td>11242:31:28*</td>
<td>11243:7:23*</td>
<td>1.00 × 100.00%</td>
<td>dSLFV</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>13683:20:32*</td>
<td>13686:25:28*</td>
<td>1.00 × 100.00%</td>
<td>rSLFV</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>+217.73%</td>
<td>+0.00%</td>
<td>+0.00%</td>
<td>overhead</td>
</tr>
<tr>
<td>16</td>
<td>128</td>
<td>0:28:56</td>
<td>100:17:53</td>
<td>100:46:49</td>
<td>111.56 × 87.16%</td>
<td>dSLFV</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>348:51:47</td>
<td>349:21:52</td>
<td>39.18 × 30.61%</td>
<td>rSLFV</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>+246.66%</td>
<td>+64.88%</td>
<td>+56.55%</td>
<td>overhead</td>
</tr>
<tr>
<td>32</td>
<td>256</td>
<td>0:28:47</td>
<td>57:57:27</td>
<td>58:26:14</td>
<td>192.40 × 75.16%</td>
<td>dSLFV</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>136:18:14</td>
<td>136:47:32</td>
<td>100.05 × 39.08%</td>
<td>rSLFV</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>+134.08%</td>
<td>+48.00%</td>
<td>+36.08%</td>
<td>overhead</td>
</tr>
<tr>
<td>64</td>
<td>512</td>
<td>0:28:43</td>
<td>26:49:4</td>
<td>27:17:47</td>
<td>411.89 × 80.45%</td>
<td>rSLFV</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>28:23:33</td>
<td>28:52:23</td>
<td>458.15 × 89.48%</td>
<td>rSLFV</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>+9.44%</td>
<td>−11.23%</td>
<td>−9.03%</td>
<td>overhead</td>
</tr>
</tbody>
</table>

(b) Fuel Control System (FCS), disturbance model $\mathcal{D}_{FCS}$

* Estimated value.
Fig. 10. Omission Probability (OP) computation during the parallel execution of the simulation campaigns.

\[
\frac{(t_x - t_c)}{t_c}
\]

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 quickly (e.g., when the coverage is \(\geq 30\%), the error is within 30\%).

7. Related work

A parallel exhaustive Hardware In the Loop Simulation based hybrid system model checking similar to the one described in this work is presented in [8]. The main differences of the present work with respect to [8] are the following. (i) Our simulation campaign optimiser and the one in [8] both take as input the admissible disturbance traces (simulation scenarios). However, the simulation campaigns computed in [8] schedule scenarios according to their order, whereas in this work we introduce an intermediate step which enables simulation of all scenarios, exactly once, in a uniform random order. (ii) During the verification process, the approach in [8] only outputs the attained coverage, whereas, in this work also the attained Omission Probability (OP) is computed, by exploiting the randomisation of the order with which scenarios are scheduled.

The work in [9] considers a finite state (digital hardware verification) setting and presents an algorithm to estimate the coverage achieved during SAT based bounded model checking. Since computation paths are not selected uniformly at random, [9] 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 [10,22,23]. 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 [10]) 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., [24]). In CCP elements are randomly extracted (uniformly and with replacement) from a finite set of n test cases (disturbance traces in out context). Known results (see, e.g., [25]) 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 allows 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 CPSs has been investigated in [26]. We note that, within a finite time bound, we are exhaustive whereas the approach in [26] is not. On the other hand, unlike our approach, [26] also evaluates how robustly the given property holds.

Probabilistic (e.g., see [27,28]) and, more specifically, simulation-based statistical model checking approaches (e.g., see [15,16,29–34]) are closely related to our work. In particular, [16] addresses statistical model checking of Simulink models and presents experimental results on one of the Simulink case studies 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 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 [35–37]. 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., our case studies). This is indeed the motivation for the development of statistical model checking methods as those in [15,16], for the exhaustive HILS based approach in [8], and for our present parallel random exhaustive HILS based approach.

8. Conclusions

We presented a parallel random exhaustive (HILS) based model checker for hybrid systems that, while being exhaustive with respect to the disturbance model given as input, provides at any time during the verification process an upper bound to the probability that the SUV exhibits an error in a yet-to-be-simulated scenario (Omission Probability, OP).

Our experimental results on real world case studies from the Simulink distribution (namely: Inverted Pendulum on a Cart and Fuel Control System) show that, by exploiting parallelism, our approach to the computation of optimised simulation campaigns is feasible even for disturbance models entailing tens of millions of operational scenarios.

Also, simulation results show that, by exploiting parallelism, our simulation campaign optimiser effectively counteracts the simulation time overhead stemming from randomisation.

Finally, we have shown that our bound to the OP decreases about linearly with the coverage, which 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.

Acknowledgements

The research leading to these results has received funding from the European Union’s 7th Framework Programme (FP7 2007–2013) under grant agreements no. 317761 and 600773. The authors would like to thank the anonymous reviewers, whose comments helped in improving this paper.

References


Toni Mancini has a Ph.D. in Computer Science Engineering and is assistant professor at the Computer Science Department of Sapienza University of Rome (Italy). His research interests comprise: artificial intelligence, formal verification, cyber-physical systems, control software synthesis, systems biology, smart grids.

Federico Mari is an assistant professor at the Department of Computer Science of Sapienza University of Rome (Italy). He received his Ph.D. degree in Computer Science in 2010 from Sapienza for his dissertation on “Automatic Verification and Control Software Synthesis for Discrete Time Linear Hybrid Systems.” Federico is interested in formal methods applied to embedded systems, namely formal verification and control software synthesis. Other research interests comprise model checking, smart grids, and systems biology.

Annalisa Massini graduated in Mathematics and got her Ph.D. in Computer Science in 1993, at Sapienza University of Rome (Italy). Since 2001 she is associate professor at the Department of Computer Science of Sapienza University of Rome. Her research interests include model checking, hybrid systems, sensor networks, networks topologies.

Igor Melatti is an assistant professor at the Computer Science Department of Sapienza University of Rome (Italy). His current research interests comprise: formal methods, automatic verification algorithms, model checking, software verification, hybrid systems, automatic synthesis of reactive programs from formal specifications.

Enrico Tronci is an associate professor at the Computer Science Department of Sapienza University of Rome (Italy). He received a master’s degree in Electrical Engineering from Sapienza University of Rome and a Ph.D. degree in Applied Mathematics from Carnegie Mellon University. His research interests comprise: formal verification, model checking, system level formal verification, hybrid systems, embedded systems, cyber-physical systems, control software synthesis, smart grids, autonomous demand and response systems for smart grids, systems biology.