*Fundamenta Informaticae XX (2016) 1–37*
*DOI 10.3233/FI-2015-0000*
*IOS Press*



# SyLVaaS: System Level Formal Verification as a Service*

**Toni Mancini** [C], **Federico Mari, Annalisa Massini, Igor Melatti, Enrico Tronci**
*Computer Science Department, Sapienza University of Rome, Italy*

**Abstract.** The goal of System Level Formal Verification is to show system correctness notwithstanding uncontrollable events (disturbances), as for example faults, variations in system parameters, external inputs, etc. This may be achieved with an exhaustive Hardware In the Loop Simulation based approach, by considering all relevant scenarios in the System Under Verification (SUV) operational environment.

In this paper, we present SyLVaaS, a Web-based tool enabling Verification as a Service (VaaS). SyLVaaS implements an *assume-guarantee* approach to (Hardware In the Loop Simulation based) System Level Formal Verification.

SyLVaaS takes as input a finite state automaton defining the SUV operational environment and computes, using parallel algorithms deployed in a cluster infrastructure, a set of highly optimised *simulation campaigns*, which can be executed in an *embarrassingly parallel* fashion (i.e., with no communication among the parallel processes) on a set of Simulink instances, using a *platform independent* Simulink driver downloadable from the SyLVaaS Web site.

As the actual simulation is carried out at the user premises (e.g., on a private cluster), SyLVaaS allows full Intellectual Property protection of the SUV model as well as of the SyLVaaS user verification flow.

The simulation campaigns computed by SyLVaaS randomise the verification order of operational scenarios and this enables, at anytime during the parallel simulation activity, the estimation of the completion time and the computation of an upper bound to the Omission Probability, i.e., the probability that there is a yet-to-be-simulated operational scenario which violates the property under verification. This information supports graceful degradation in the verification activity.

We show effectiveness of the SyLVaaS algorithms and infrastructure by evaluating the system on case studies consisting of input operational environments entailing up to $35\,641\,501$ scenarios related to the system level verification of models from the Simulink distribution (namely, Inverted Pendulum on a Cart and Fuel Control System).

*This is an extended and revised version of [1].
[C]Corresponding author

**Keywords:** Verification as a Service; Model Checking; Hybrid Systems; System Level Formal Verification; Distributed Multi-Core Hardware in the Loop Simulation.

# Contents

**1 Introduction** **4**
1.1 Motivations . . . 4
1.2 Main Contributions . . . 5

**2 Background** **7**
2.1 Modelling the SUV . . . 7
2.2 Modelling the Property to be Verified . . . 7
2.3 Modelling the SUV Operational Environment . . . 8
2.4 System Level Formal Verification . . . 9
2.5 Parallel HILS Based Anytime Random Exhaustive SLFV . . . 10

**3 System Level Formal Verification as a Service** **10**
3.1 Input . . . 10
3.2 Output . . . 11
3.3 Web Interface . . . 11
3.4 How to Use SyLVaaS Output . . . 13

**4 Parallel Generation of Disturbance Traces** **14**
4.1 Algorithm Overview . . . 14
4.2 Distributed Trace Labelling . . . 15
4.3 Orchestrator . . . 16
4.4 Slaves . . . 16
4.5 Algorithm Correctness . . . 17

**5 Experiments** **21**
5.1 SyLVaaS Experimental Deployment . . . 21
5.2 Case Studies . . . 21
5.2.1 Inverted Pendulum on a Cart (IPC) . . . 21
5.2.2 Fuel Control System (FCS) . . . 24
5.3 Experimental Results . . . 25
5.3.1 Parallel Disturbance Trace Generation . . . 25
5.3.2 SyLVaaS Complete Workflow . . . 27
5.3.3 Download of Simulation Campaigns . . . 27

**6 Related Work** **30**

**7 Conclusions** **32**

# 1. Introduction

A Cyber-Physical System (CPS) consists of interconnected hardware and software subsystems. As a result, the *state* of a CPS consists of continuous (e.g., stemming from analog devices) as well as discrete (e.g., stemming from software or digital devices) components. Accordingly, CPSs are typically modelled as *Hybrid Systems* (see, e.g., [2] 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. Hardware In the Loop Simulation (HILS) is the main workhorse for system level verification and is supported by *Model Based Design* tools like Simulink (http://www.mathworks.com), Modelica (https://www.modelica.org) and VisSim (http://www.vissim.com). In HILS, the CPS software components read/send values from/to mathematical models (*simulation*) of the CPS physical subsystems (e.g., engines, analog circuits, etc.) they interact with. This allows designers to simulate the whole CPS behaviour on a given *simulation scenario* (i.e., a sequence of *exogenous* stimuli, such as faults, to the system).

Of course, in order to rule out the presence of design errors, one would like to consider *all* possible simulation scenarios, thereby aiming for System Level Formal Verification (SLFV). Since CPSs can be modelled as hybrid systems, one may think of using model checkers for hybrid systems in order to address SLFV for CPSs. Unfortunately, no model checker for hybrid systems can handle SLFV of actual CPSs. For this reason, currently HILS is basically the only approach used to carry out system level verification of CPSs.

## 1.1. Motivations

System Level Formal Verification (SLFV) is an *exhaustive* HILS, where *all* relevant simulation scenarios are considered. In [3, 4, 5, 6] a methodology has been presented which allows exhaustive HILS. Such a methodology works as follows.

First, we note that the CPS to be verified (the System Under Verification (SUV)) can be regarded as a hybrid system whose inputs belong to a finite set of uncontrollable events (*disturbances*), which model failures in sensors or actuators, variations in the system parameters, etc.

Second, the SUV is a *deterministic system* (the typical case for control systems). Nondeterministic behaviours (such as faults) are modelled with disturbances.

Third, sequences of inputs to the SUV are of *bounded* length, thus the problem addressed is indeed *bounded* SLFV. Accordingly, a *simulation scenario* is a finite sequence of disturbances.

From the above, it follows that a system (namely, our SUV) is expected to *withstand* all disturbance sequences that may arise in its operational environment. Correctness of a system (defined in terms of *safety* properties) is thus defined with respect to such *admissible* disturbance sequences.

Given a high-level model defining the admissible disturbance sequences (*disturbance model*), the approach in [3, 4, 5, 6]:

1. Generates the entire set of admissible disturbance sequences from the disturbance model;
2. Evenly splits such a set into $k > 0$ *slices* in order to enable *parallel* verification;
3. Computes (in parallel) an optimised *simulation campaign* from each slice;

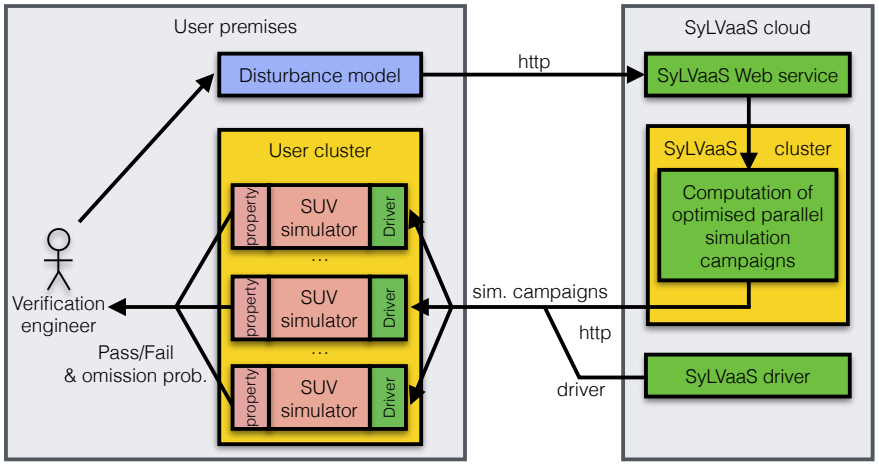



Figure 1: SyLVaaS VaaS architecture.

4. Executes (in parallel) the generated simulation campaigns on a set of $k$ independent *simulators* (e.g., Simulink or Modelica instances).

A *simulation campaign* is a sequence of simulation *instructions*, which exploit the capabilities of modern simulators to save and restore previously stored simulation states (much as in explicit model checking). In particular, a simulation campaign consists of the following commands: *save* a simulation state, *restore* a saved simulation state, *inject* a disturbance, *advance* the simulation for a given time length.

As soon as one of the simulators (running the simulation campaign corresponding to a given slice) finds an error, the whole parallel simulation activity stops, and the disturbance trace which triggered the error is returned as a *counterexample*. Also, as the generated optimised simulation campaigns (one per slice) randomise the verification order of the traces in the input slice, at *anytime* during the parallel simulation activity it is possible to compute an *upper bound* to the Omission Probability (OP), i.e., the probability that an error exists, but no error has been found so far, and to give a quite accurate estimation of the *completion time*.

Algorithms for all the activities described above have been presented in the cited papers. However, an off-the-shelf tool to *effectively* support companies working in the CPS business in their everyday SUV verification activities is not available. To provide such a tool is exactly the purpose of this paper.

## 1.2. Main Contributions

We present SyLVaaS (see Figure 1), a Web-based service taking as input a disturbance model and effectively computing the set of simulation campaigns to be used for a *parallel* HILS based SLFV. The main features of SyLVaaS can be summarised as follows.

### Protection of the SUV Intellectual Property (IP)

HILS based SLFV takes as input three main artefacts: the SUV model, the definition of the property to be verified and the definition of the SUV operational environment (in terms of a disturbance model). In an industry setting, both the SUV model and the property to be verified are subject to IP protection, as they represent the main assets of the company designing the CPS (hence the SUV model). On the other hand, the definition of the operational environment does require IP protection, as it encodes the uncontrollable inputs (exogenous stimuli) to the SUV.

*SyLVaaS introduces the new Verification as a Service (VaaS) paradigm*, allowing verification engineers (SyLVaaS *users*) to use an external (Web) service (SyLVaaS) to compute the simulation campaigns needed for their HILS based SLFV activities, by fully protecting IP of their models. In particular, SyLVaaS does not require the SUV model nor the property to be verified, and takes as input only a *disturbance model*, defined as a CMurphi [7] model. Also, disturbances in the disturbance model are defined in a way fully decoupled from the SUV model, e.g., by means of integers. The actual verification activity is performed in parallel at the user premises (e.g., on a private cluster) running an arbitrarily large set of Simulink simulators, using the optimised simulation campaigns computed by SyLVaaS and plugging-in a Simulink driver downloadable from the SyLVaaS Web site. Before using such a driver, the user must define a suitable correspondence between the opaque values defining disturbances in the disturbance model (e.g., integers) and actual assignments to parameters of the SUV. This further contributes to protect IP of the SUV model, as also such correspondence is kept private.

### Protection of the Verification Flow IP

SyLVaaS also protects the user verification flow IP to outsiders. In fact, in case an error is found during the verification activity (at the user premises), a counterexample is generated. Such a counterexample can then be used to revise the SUV, thereby producing a new *SUV* model. At this point, a new SLFV activity can start. Given that the set of admissible operational scenarios (hence: the disturbance model) has not changed, there is no need to interact with SyLVaaS again, as the previously computed simulation campaigns can be reused.

### Fast Response Time via Parallel Computation

The operational scenario generation algorithm in [3] is a sequential algorithm, taking about half an hour (on a desktop PC) to generate a few millions of simulation scenarios. Although this time is negligible with respect to the whole HILS based SLFV activity (which can take weeks of computation), it becomes a major bottleneck in a VaaS context, as the one provided by SyLVaaS, since scenario generation is the most intensive part of the computation carried out on the SyLVaaS side (i.e., generation of optimised simulation campaigns for parallel HILS, see Figure 1, right).

In order to achieve a fast response time in SyLVaaS, in this paper we present a *new parallel algorithm for the generation of operational scenarios* from a disturbance model, and discuss its *distributed multi-core implementation* explicitly designed to operate efficiently on a cluster of possibly heterogeneous machines.

Our new operational scenario generation algorithm consists of an *Orchestrator* process which governs the exploration of the (state space of the finite state automaton defined by the) disturbance model provided by the user, splitting and delegating the work to a battery of available *Slaves*, whose work load

is dynamically balanced. Slave processes are independent from each other and communicate only with the Orchestrator. This minimises coordination overhead.

**Experimental Evaluation**

We present experimental results on using our parallel algorithm on case studies consisting of disturbance models for two SUVs (namely, the Inverted Pendulum on a Cart (IPC) and the Fuel Control System (FCS) in the Simulink distribution) entailing a number of operational scenarios up to $35\,641\,501$.

Our results show that our new parallel algorithm for operational scenario generation scales well with the number of Slaves. As the operational scenario generation is the most computationally intensive task in the SyLVaaS workflow, and given that the other steps performed by SyLVaaS (computation of optimised simulation campaigns) already exploit an embarrassingly parallel algorithm (i.e., an algorithm with no communication among the processes) from [4], with our new parallel disturbance trace generator the entire SyLVaaS workflow can benefit of a cluster of machines at the SyLVaaS cloud infrastructure.

## 2. Background

In this section we give some background notions. Unless otherwise stated, all definitions are based on [8] and [3, 4, 5, 6] to which we refer the reader for more in-depth details.

In the following, we denote with $\mathbb{R}$, $\mathbb{R}^{\geq 0}$, $\mathbb{R}^{+}$ and $\mathbb{N}^{+}$ the sets of, respectively, all real, non-negative real, strictly positive real, and strictly positive natural numbers, and with $\mathrm{Bool} = \{0, 1\}$ the set of Boolean values (where $0$ means 'false' and $1$ means 'true').

### 2.1. Modelling the SUV

A System Under Verification (SUV) is modelled as a Discrete Event System (DES), namely a continuous time Input-State-Output deterministic dynamical system [8] 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 (Figure 2a).

### 2.2. Modelling the Property to be Verified

The property to be verified is modelled as a continuous time *monitor* embedded in the SUV (see Figure 2b), which observes the state of the system and checks whether the property under verification is satisfied. The output of the monitor (see Figure 2c) 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. 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.

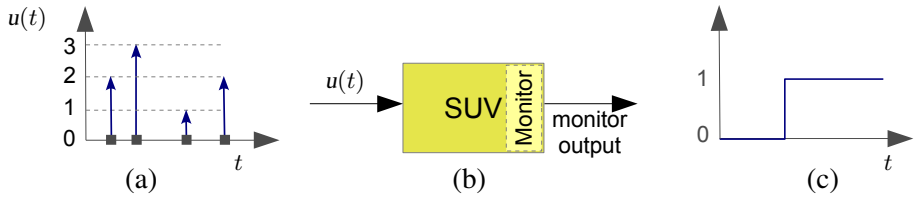



Figure 2: (a) A discrete event sequence ($d = 3$); (b) Our SUV embedding a monitor; (c) The SUV monitor output.

## 2.3. Modelling the SUV Operational Environment

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 2.2) 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.

As it is typically infeasible to define the SUV operational environment by explicitly listing all the admissible disturbance traces, we define it by means of a *disturbance model*, which is in turn defined as the language accepted by a suitable automaton, called Disturbance Generator (DG) (see Definition 2.1 and Figure 3a–c).

**Definition 2.1. (Disturbance Generator)**
A Disturbance Generator (DG) is a tuple $\mathcal{D} = (Z, d, \mathit{dist}, \mathit{adm}, Z_I, Z_F)$ where:

- $Z$ is a finite set of states;
- $Z_I \subseteq Z$ and $Z_F \subseteq Z$ are the set of, respectively, initial and final states;
- $d \in \mathbb{N}^+$ defines the set of disturbance events represented (without loss of generality) with integers in $[0, d]$, where value 0 represents the event carrying no disturbance;
- $\mathit{dist} : Z \times [0, d] \to Z$ is a (*deterministic transition*) function mapping each state/disturbance pair $(z, e)$ to a *next* state $\mathit{dist}(z, e)$;
- $\mathit{adm} : Z \times [0, d] \to$ Bool is a (*guard*) function defining (*the characteristic function of*) the set of disturbances admissible (i.e., that may occur) in a given state. □

Note that we model simultaneous disturbances as one single event (i.e., one disturbance).

Definition 2.2 defines disturbance traces (simulation scenarios) as paths from initial to final states in a DG.

**Definition 2.2. (Disturbance Trace)**
Let $\mathcal{D} = (Z, d, \mathit{dist}, \mathit{adm}, Z_I, Z_F)$ be a DG.

(a) A *disturbance path* of length $h$ for $\mathcal{D}$ is a computation path in $\mathcal{D}$ with $h$ disturbances (transitions). Formally, it is a sequence $z_0, d_0,\ z_1, d_1, \ldots, z_{h-1}, d_{h-1},\ z_h$, where $z_0 \in Z_I$, $z_h \in Z_F$ and, for all $0 \leq i < h$, $z_i \in Z$, $d_i \in [0, d]$, $\mathit{adm}(z_i, d_i) = 1$, and $z_{i+1} = \mathit{dist}(z_i, d_i)$.

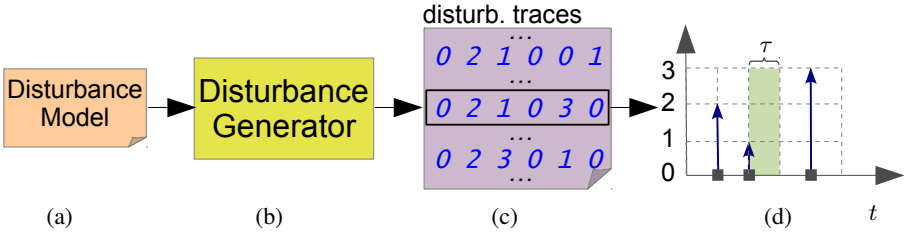



Figure 3: (a) Disturbance model; (b) Disturbance generator; (c) Generated sequence of disturbance traces ($d = 3, h = 6$); (d) The discrete event sequence associated to the trace in the black rectangle in part (c), given time quantum $\tau$.

(b) A *disturbance trace* of length $h$ for $\mathcal{D}$ is a sequence $\delta = d_0, \ldots, d_{h-1}$ of $h$ disturbances such that there exists a disturbance path $z_0, d_0,\ z_1, d_1, \ldots, z_{h-1}, d_{h-1},\ z_h$ for $\mathcal{D}$. We denote with $\delta(j)$ the $j$-th disturbance occurring in trace $\delta$ ($0 \leq j < h$). □

Given $\tau \in \mathbb{R}^+$ (*time quantum*), to a disturbance trace $\delta$ for $\mathcal{D}$ we can univocally associate a discrete event sequence $u^\tau_\delta$, 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 $\delta$ defines an operational scenario (namely, $u^\tau_\delta$) for our SUV. Figure 3d shows the discrete event sequence associated to a disturbance trace. We represent our SUV *operational environment* as a finite *set* of disturbance traces $\Delta = \{\delta_0, \ldots, \delta_{n-1}\}$ for $\mathcal{D}$, 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 $\tau$ *small enough* (to faithfully model our SUV operational scenarios), we can achieve any desired precision. On such considerations rests the effectiveness of the approach.

## 2.4. System Level Formal Verification

Definition 2.3 formalises our bounded System Level Formal Verification problem.

**Definition 2.3.** A *System Level Formal Verification (SLFV) problem* is a tuple $(\mathcal{H}, \mathcal{D}, \tau, h)$ where: $\mathcal{H}$ is a DES with an embedded monitor modelling our SUV, $\mathcal{D}$ is a DG modelling a set of disturbance traces $\Delta$ over horizon $h \in \mathbb{N}^+$, and $\tau \in \mathbb{R}^+$ is a time quantum.

The *answer* to SLFV problem is *FAIL* if there exists a disturbance trace $\delta$ in $\Delta$ such that the SUV monitor output at time $\tau h$ is 1, when $\mathcal{H}$ is given $u^\tau_\delta$ (the discrete event sequence associated to $\delta$ given time quantum $\tau$) as input, and *PASS* otherwise. In case of *FAIL*, the disturbance trace raising the error is returned as a *counterexample*. □

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 $\tau$ as the time quantum among disturbance events.

### 2.5. Parallel HILS Based Anytime Random Exhaustive SLFV

We follow a black-box parallel approach to SLFV, where the DES $\mathcal{H}$ defining our SUV (plus the property to be verified) is defined using the modelling language of a suitable *simulator* (namely, MatLab and Stateflow for Simulink). We compute the answer to a SLFV problem $(\mathcal{H}, \mathcal{D}, \tau, h)$ by simulating *each* disturbance trace $\delta$ in the operational environment $\Delta$, thus performing an *exhaustive* (with respect to $\Delta$) Hardware In the Loop Simulation (HILS).

In order to enable parallel simulation over $k \in \mathbb{N}^+$ machines available in the (private) user cluster, we evenly partition the sequence of disturbance traces $\Delta$ 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 on the SyLVaaS cluster, $k$ highly optimised simulation campaigns, which can be executed in parallel using $k$ independent *simulators*, each one running (on a different core of the user cluster) a model for $\mathcal{H}$. 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 $\delta$ which raised the error is returned as a *counterexample*). 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.

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 important, since each state may require many KB of memory (150–300 KB in the case studies presented in this paper).

Also, as each computed simulation campaign verifies the disturbance traces in the input slice in a *random order*, it is possible to compute at *anytime* during the simulation process (along the lines of [6]), an estimation of the simulation *completion time* and an upper bound to the *Omission Probability (OP)*, i.e., the probability that there is a yet-to-be-simulated disturbance trace which violates the property under verification. This information enables the verification engineer to evaluate if it is worth to continue the simulation activity, or instead stop it since the degree of assurance attained can be considered adequate for the application at hand (*graceful degradation*).

## 3. System Level Formal Verification as a Service

In this section we describe SyLVaaS in terms of input and output, and describe how to use the system output.

### 3.1. Input

SyLVaaS requires two inputs:

1. An integer $k > 0$ describing the number of computational cores available on the user side for parallel execution of simulation campaigns (hence, for parallel verification);

2. A disturbance model defining the operational environment, i.e., the set of disturbance traces the System Under Verification (SUV) should withstand, along with a bounded horizon $h$.

As it is typically infeasible for a verification engineer to define a SUV operational environment by explicitly listing all its disturbance traces, SyLVaaS takes as input a *disturbance model* defining a Disturbance Generator (DG) written in the high-level language accepted by the CMurphi [7] model checker. The following example clarifies this point.

**Example 3.1.** Assume that a SyLVaaS user wants to verify a SUV with two sensors, $A$ and $B$, which may fail (without repair) at times multiple of 1 second. Fault of any sensor might occur only if the other one did not fail, or failed more than 2 seconds before. The CMurphi description for the DG modelling such operational environment is shown in Figure 4, where the verification time horizon is 7 seconds. □

## 3.2. Output

From the value of $k$ and the input disturbance model, SyLVaaS produces $k$ *simulation campaigns*, which can be executed in parallel on the user premises over $k$ independent simulators, in an *embarrassing parallel* fashion (i.e., with no communication among processes).

Each simulation campaign verifies, in a highly optimised way, a disjoint and equally-sized portion of the disturbance traces entailed by the input disturbance model. Conversely, all disturbance traces entailed by the disturbance model are covered by exactly one simulation campaign. This guarantees that the System Level Formal Verification (SLFV) process is both *exhaustive* (with respect to the set of disturbance traces entailed by the disturbance model) and *non-redundant*. Also, the verification order of the disturbance traces covered by each simulation campaign is *randomised*. This, according to [6], enables the computation of an upper bound to the Omission Probability (OP) at *anytime* during the parallel simulation.

The $k$ simulation campaigns are returned to the user via the Web interface, together with an *abstract Simulink driver*. Such a driver is a MatLab script that reads and executes a SyLVaaS-generated simulation campaign, by sending simulation commands to Simulink. It is "abstract" as it must be plugged into the SUV Simulink model and configured at the user premises (see Figure 1 and Section 3.4).

## 3.3. Web Interface

The Web interface of SyLVaaS is hosted at http://mclab.di.uniroma1.it/sylvaas. It consists of four main pages:

1. A standard login page.

2. A user console page (accessible after login, see Figure 5) showing all current, pending, running and completed user jobs. For each job, the console shows the job unique id, the corresponding input and its the status (pending, running, completed or deleted). By selecting a job id, it is possible to see and download the corresponding input. If the job is completed, it is also possible to download the final $k$ simulation campaigns.

3. A page to create a new job request, where the user must fill a form with the required input: the disturbance model, the horizon for the disturbance model and the number of computational cores available on the user side for parallel execution of simulation campaigns (see Section 3.1).

```
const h : 7; -- horizon
const A : 1; -- id of sensor A
const B : 2; -- id of sensor B

-- Global variables
var
  t : 1 .. h + 1; -- time
  d : array [A .. B] of 0 .. h; -- disturbance times

-- Initial state (only one in this example)
startstate
  begin
    t := 1;
    d[A] := 0;
    d[B] := 0;
  end;

-- Rules: if the rule guard is true in the current state, then modify the current state as specified
-- in the rule body. Rules are fired nondeterministically.

rule "ok" -- rule for no disturbance
  t <= h ==> -- it is always possible to have no disturbances
    t := t + 1; -- time is incremented

rule "A fails" -- rule for failing of A
  -- this encodes the conditions in Example 3.1
  t <= h & d[A] = 0 & (d[B] = 0 | (t-d[B] > 2)) ==>
        begin
          d[A] := t; -- failing of A has occurred at time t
          t := t + 1;
        end;

rule "B fails"
  ... -- omitted, as it is similar to the rule above

-- The last state of each trace must fulfil this condition
finalstate (t = h + 1);
```

Figure 4: CMurphi code for the DG described in Example 3.1.


Figure 5: SyLVaaS user console page.

4. A tools page, where the generic driver can be downloaded, together with examples showing how to customise "abstract" Simulink drivers.

When a job is completed, the user is warned by an email. He can then proceed to the download of the simulation campaigns.

## 3.4. How to Use SyLVaaS Output

Given the output downloaded by SyLVaaS, the verification engineer, in order to actually verify the SUV via exhaustive Hardware In the Loop Simulation (HILS), customises and plugs the abstract Simulink driver into the SUV Simulink model. This task is very easy and consists in properly filling the *template* files received by SyLVaaS as part of the abstract driver. Such files define: the SUV model, the SUV property to be verified (as a *monitor* module), the interface between the driver and the SUV, and the mapping between each disturbance (in the CMurphi disturbance model) and its counterpart in the SUV model.

At this point, the $k$ downloaded simulation campaigns can be executed in parallel on $k$ independent simulators. Given the randomisation of the verification order of the disturbance traces within each simulation campaign, at anytime during the simulation process, when ratios $done_1, done_2, \ldots, done_k$ (with $done_i \in [0, 1]$ for all $i$) of the traces covered by each simulation campaign have been verified successfully (i.e., no error has been raised so far), the Omission Probability (OP), i.e., the probability that a future simulation command raises an error, is upper bounded by: $1 - \min_{i \in [1,k]} (done_i)$ (see [6]).

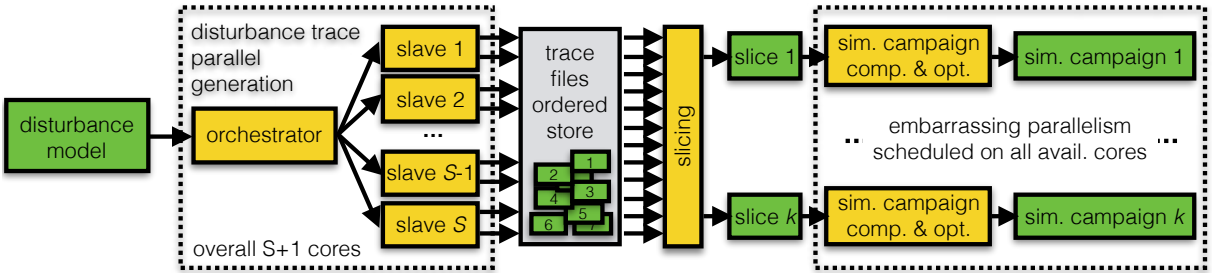



Figure 6: SyLVaaS workflow and deployment.

## 4. Parallel Generation of Disturbance Traces

As reported in [3], the most computationally intensive step of the workflow for the computation of simulation campaigns is disturbance trace generation starting from the user disturbance model. This task is performed in [3] using a modified version of the CMurphi model checker. As reported there, on a disturbance model entailing about 4 million traces (the same referred to as $\mathcal{D}^1_{\text{FCS}}$ in Section 5), trace generation takes about 30 minutes, while the subsequent step (i.e., computation of simulation campaigns) takes about 1 minute, as it can be massively parallelised [5]. The time to generate disturbance traces is anyway negligible if we consider also the time to carry out (in parallel) the actual simulation, which may take days.

However, in a Verification as a Service (VaaS) context as that of SyLVaaS, the simulation campaigns are actually executed at the user premises, and disturbance trace generation from the user disturbance model would become the *most time-dominant step* in the SyLVaaS workflow.

To this end, to achieve fast response time in SyLVaaS, here we present a *new parallel algorithm* for distributed trace generation. As a result, with this new algorithm the *whole* SyLVaaS workflow (i.e., generation of disturbance traces and computation of optimised simulation campaigns) can now take benefit from the availability of a cluster in the SyLVaaS cloud infrastructure (see Figure 6).

### 4.1. Algorithm Overview

Our new parallel algorithm for trace generation has been explicitly designed to operate efficiently on a cluster of possibly heterogeneous machines, and consists on a single *Orchestrator* process and a number $S \in \mathbb{N}^+$ of *Slaves*. The Orchestrator governs the exploration of the state space of the Disturbance Generator (DG) defined by the disturbance model provided by the user, splitting and delegating the work to the Slaves. To avoid communication as well as data structures shared among the Slaves, the DG state space is regarded as a set of trees, one for each DG initial state. This does not pose any termination problem, as we are looking for disturbance traces of bounded length $h$.

The Orchestrator performs a Depth-First Search (DFS) up to bounded level (depth) $L < h$ and delegates the exploration of the subtrees rooted at each node at depth $L$ to an idle slave, see Figure 7. The exploration of each subtree by a Slave $s \in [1, S]$ is again carried out by DFS, and is called a *computation bunch*. Each computation bunch $b$ executed from Slave $s$ gives as output a sequence of traces which is appended to the sequence of traces $\Delta_s$ generated by $s$. Sequence $\Delta_s$ contains a subset

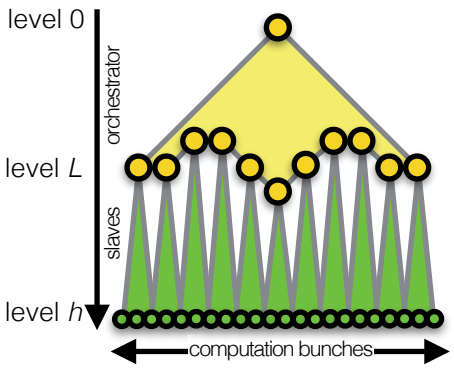



Figure 7: Parallel trace generation.

of the disturbance traces entailed by the model. The sets $(\Delta_1, \ldots, \Delta_S)$ of traces produced by all Slaves form a *partition* of the entire set of admissible disturbance traces $\Delta$.

The simplicity of the algorithm minimises network communication and coordination among processes. In particular, Slave processes are *independent* from each other and communicate only with the Orchestrator.

## 4.2. Distributed Trace Labelling

Both the Orchestrator and the Slaves work in DFS mode, and hence each computation bunch produces a sequence of disturbance traces in *lexicographic order*. Each disturbance trace *prefix* identifies a simulator state, and we associate a unique label to all prefixes of disturbance traces (Definition 4.1).

**Definition 4.1. (Labelling of Disturbance Traces)**
Let $\mathcal{D}$ be a DG defining $d \in \mathbb{N}^+$ disturbances, and $\Lambda$ be a countably infinite set of labels. A labelling function over $[0, d]$ is an injective map $\lambda$ from finite sequences of values in $[0, d]$ (including the empty sequence) to labels in $\Lambda$.

Let $\delta = d_0, \ldots, d_{h-1}$ be a disturbance trace for $\mathcal{D}$. The labelling of $\delta$ (according to $\lambda$) is $\delta^\lambda = l_0, d_0, \ldots, l_{h-1}, d_{h-1}, l_h$ where, for all $0 \leq i \leq h$, $l_i = \lambda(d_0, \ldots, d_{i-1})$. □

As a consequence of Definition 4.1, prefixes of disturbance sequences $(\hat{d}_0, \ldots, \hat{d}_{p-1})$ common to multiple disturbance traces are followed by the *same* label $\hat{l}_p = \lambda(\hat{d}_0, \ldots, \hat{d}_{p-1})$. Labels identifying prefixes common to multiple disturbance traces are essential in the efficient computation of highly optimised simulation campaigns, as they represent the only simulator states which might be worth storing, as they may be needed later (see, for more details, the optimiser in [4]). Note that, given that both the Orchestrator and the Slaves run in DFS mode, disturbance traces can be labelled at *no additional computational cost* during generation. In particular, the Orchestrator labels trace prefixes up to level $L$, while Slaves label trace prefixes longer than $L$.

Our parallel algorithm uses the following labelling schema, which results in an overall injective map $\lambda$ for disturbance prefix labels while avoiding communication among the processes. Let $S \in \mathbb{N}^+$ be the number of available Slaves. We set $\Lambda = \mathbb{N}^+$. The Orchestrator associates, to each new disturbance

prefix, a label extracted from the set $\{l \mid l \in \mathbb{N}^+, l = j(S+1)+1, j \geq 0\}$, according to their natural order. Analogously, each Slave $s \in [1, S]$ associates, to each new disturbance prefix, a label extracted from the set $\{l \mid l \in \mathbb{N}^+, l = j(S+1)+s+1, j \geq 0\}$, So, for example, if $S = 2$, the Orchestrator uses labels from set $\{1, 4, 7, \ldots\}$, Slave 1 uses labels from $\{2, 5, 8, \ldots\}$, and Slave 2 uses labels from $\{3, 6, 9, \ldots\}$. Note that, as these sets of labels are *disjoint*, the resulting overall map is injective.

## 4.3. Orchestrator

The Orchestrator process, whose pseudocode is shown as Algorithm 1, governs the exploration of the DG state space, by performing a DFS up to a bounded depth (level) $1 \leq L \leq h-1$ (whose initial value is given as a parameter), also assigning unique labels (see variable $\lambda$) to disturbance trace prefixes. When level $L$ is reached, the Orchestrator delegates the exploration of the subtree rooted at the current state to an idle Slave, forwarding to it the (labelled) prefix (containing exactly $L$ disturbances) of the disturbance trace computed so far. Each such delegated task (*computation bunch*) is assigned a sequential id (see variable $b$). As the exploration is done by the Orchestrator using DFS, the disturbance sequences passed to the Slaves are generated in *lexicographic order*.

In order to keep a high efficiency of the whole parallel process, the value of $L$ is dynamically and adaptively adjusted by the Orchestrator during exploration, depending on how much the Orchestrator is waiting to find an idle slave. Let $w$ be the time the Orchestrator had to wait, in the last iteration, before finding an idle Slave, and let $t$ be the overall time spent by the Orchestrator in the last iteration. If $\frac{w}{t} > maxW$ (value of $maxW$ is given as a parameter), the Orchestrator increases value of $L$ by one. This means that, from now on, the Orchestrator will perform DFS one level deeper and will delegate to the Slaves smaller computation bunches (i.e., the exploration of smaller subtrees), as it had evidence that Slaves are overloaded. Conversely, if $\frac{w}{t} < minW$ (value of $minW$ is given as a parameter), the Orchestrator decreases value of $L$ by one, hence starts delegating to the Slaves larger computation bunches (i.e., the exploration of larger subtrees), as it has evidence that Slaves are, on average, underloaded.

Together with the fact that the faster Slaves will, on average, execute a higher number of computation bunches than slower Slaves, the above described dynamic and adaptive adjustment of value $L$ provides a simple yet very effective load balancing mechanism among the Orchestrator and the Slaves, which avoids any communication overhead: the communication among the processes is *minimal* and consists only of the set of one-way messages that the Orchestrator sends to the Slaves to delegate computation bunches to them.

## 4.4. Slaves

Slave processes follow Algorithm 2. Each Slave waits for an Orchestrator request to perform a computation bunch. Each such request consists in tuple $(z_0, b, \delta^{\lambda}|_{l_L})$, where $z_0 \in Z_I$ is one of the initial states of $\mathcal{D}$, $b$ is the computation bunch id, and $\delta^{\lambda}|_{l_L}$ is a labelled prefix of disturbance traces (containing $L$ disturbances), as computed by the Orchestrator.

Upon reception of $(z_0, b, \delta^{\lambda}|_{l_L})$, a Slave $s \in [1, S]$: (i) reaches the root of the subtree which is in charge to explore by following $\delta^{\lambda}|_{l_L}$, (ii) starts its own DFS from there, hence limiting its attention to that subtree.

Admissible (complete) disturbance traces found (which have $\delta^{\lambda}|_{l_L}$ as a prefix) are appended to the output file $\Delta_s$ of Slave $s$ and annotated with the id $b$ of the current computation bunch. During ex-

ploration, each Slave also carries out trace labelling using its own (disjoint) set of labels (see variable $\lambda$).

## 4.5. Algorithm Correctness

Theorem 4.2 shows correctness of our parallel algorithm for generation of disturbance traces.

**Theorem 4.2. (Algorithm Correctness)**
Let $\mathcal{D}$ be a DG, $h, S \in \mathbb{N}^+$, and $\lambda$ be a labelling function according to Definition 4.1. Let $\Delta$ be the entire set of disturbance traces for $\mathcal{D}$ with horizon $h$, and let $\Delta_s$ be the (ordered) sequence of disturbance traces generated by Slave process $s \in [1, S]$ of our algorithm. The following holds:

(a) $(\Delta_1, \ldots, \Delta_S)$ form a partition of $\Delta$ (when ignoring the trace order within each $\Delta_s$ and the annotations regarding the computation bunch ids);

(b) for all $s \in [1, S]$, disturbance traces in $\Delta_s$ are lexicographically ordered (when ignoring their associated computation bunch ids);

(c) for all $s, s' \in [1, S]$ and for all $b, b' \in \mathbb{N}^+$ such that $b < b'$, each trace in $\Delta_s$ generated during (hence annotated with) computation bunch $b$ is lexicographically less than all traces in $\Delta_{s'}$ generated during (hence annotated with) $b'$. □

**Proof:**
To prove (a), let us temporarily ignore the dynamic and adaptive adjustment of value $L$ (lines 28–30 of Algorithm 1). In this case, the Orchestrator expands the computation path tree of DG $\mathcal{D}$ using a standard Depth-First Search (DFS) approach up to (fixed) depth level $L$ (storing in *stack* the search *frontier*). From level $L$ the DFS expansion of each subtree (and the relevant frontier) is delegated to a Slave. As $L < h$, every disturbance trace in $\Delta$ (which is an admissible sequence of $h$ disturbances) is generated by exactly one Slave. Thus, $(\Delta_1, \ldots, \Delta_S)$ form a partition of $\Delta$.

This fact is preserved under the dynamic and adaptive adjustment of value $L$ (between 1 and $h-1$). To see why, assume that at some iteration, the Orchestrator pops-out from *stack* record $(z, \hat{d}, j)$. If, at line 28, $adjL \neq 0$, we must have that $adm(z, \hat{d})$ is true and $j + 1 = L$, i.e., the Orchestrator has just delegated the expansion of the subtree rooted at $\hat{z} = dist(z, \hat{d})$ to a Slave. Also, note that if $adjL \neq 0$, then $adjL = \pm 1$.

When $adjL > 0$ (and, hence, 1), $L$ is incremented by one. This does not have any impact on the completeness of the algorithm: at the next iteration of the algorithm, when another record is popped out from the stack, the Orchestrator will simply go one more level deeper in the tree before delegating subtrees to the slaves.

On the other hand, decrementing $L$ could in principle make a disturbance trace being generated in two different computation bunches. However, when $adjL < 0$ (and, hence, $-1$), the Orchestrator decrements $L$ by one (i.e., sets it to value $j$) *only* if $\hat{d} = d$ (see line 28 of Algorithm 1), i.e., only if the Orchestrator has processed the *last* disturbance possibly applicable to state $z$. This implies that all records in *stack* will be of the form $(z', d', j')$, such that if $adm(z', d')$, then $dist(z', d')$ (the state reached by applying disturbance $d'$ to $z'$) is different from and not an ancestor nor a descendant of $z$. This impedes that two subtrees whose exploration is delegated to the Slaves have a common disturbance sequence.

Proof of (b) follows directly from the observation that both the Orchestrator and the Slaves apply, to each state, disturbances in lexicographic order, as both algorithms push them into the stack in *reverse* lexicographic order.

Proof of (c) follows from the previous point and from the observation that sequence of values of the Orchestrator variable $b$ (holding the computation bunch ids) is monotonically increasing. □

Theorem 4.2 shows that, from $\Delta_1, \ldots, \Delta_S$, we can easily produce $k \in \mathbb{N}^+$ lexicographically ordered slices $(\mathit{slice}_1, \ldots, \mathit{slice}_k)$ of the same length (where $k \in \mathbb{N}^+$ is the number of parallel cores available at the user side for parallel simulation), as required by [5].

Once the $k$ slices have been produced, they are independently given to $k$ instances of the optimiser of [3], which are responsible to generate $k$ output simulation campaigns for them, also randomising the trace verification order, along the lines of [6]. This enables Omission Probability (OP) computation at anytime during the simulation activity at the user premises (see Section 2) as well as completion time estimation. As already shown in [5], the generation of the $k$ simulation campaigns can be scheduled on all the cores available to SyLVaaS in an *embarrassingly parallel* fashion.

**function** *Orchestrator*($\mathcal{D}, h, L, S, minW, maxW$)
  **Input**: $\mathcal{D} = (Z, d, \mathit{dist}, \mathit{adm}, Z_I, Z_F)$, a DG
  **Input**: $h$, bounded length for disturbance traces
  **Input**: $L$, level of the search tree below which exploration is delegated to slaves
  **Input**: $S$, number of available slaves
  **Input**: $minW$, minimum percentage of wall-clock time to be spent in waiting for a slave
  **Input**: $maxW$, maximum percentage of wall-clock time to be spent in waiting for a slave
  $b \leftarrow 1$; *// id of the next comp. bunch*
  **let** $\delta^\lambda$ be an array of variables $l_0, d_0, l_1, d_1, \ldots, l_h$;
  $adjL \leftarrow 0$;
  $\lambda \leftarrow 1$; *// next label to be used*
  **foreach** $z_0 \in Z_I$ **do**
    *stack* $\leftarrow$ empty stack;
    **for** $d'$ *from* $d$ *downto* $0$ **do**
      *push*(*stack*, $(z_0, d', 0)$); *// (state, dist. to try, depth)*
    $l_0 \leftarrow \lambda$; $\lambda \leftarrow \lambda + S + 1$;
    **while** *stack is not empty* **do**
      $(z, \hat{d}, j) \leftarrow$ *pop*(*stack*);
      **if** *adm*$(z, \hat{d})$ **then**
        $\hat{z} \leftarrow$ *dist*$(z, \hat{d})$; *// $\hat{z}$ is at depth $j+1$*
        $d_j \leftarrow \hat{d}$;
        $l_{j+1} \leftarrow \lambda$; $\lambda \leftarrow \lambda + S + 1$;
        **if** $j + 1 < L$ **then**
          **for** $d'$ *from* $d$ *downto* $0$ **do** *push*(*stack*, $(\hat{z}, d', j+1)$) ;
        **else**
          *// delegate computation bunch*
          **wait** for an idle slave $s$;
          $w \leftarrow$ time elapsed while waiting;
          $t \leftarrow$ time elapsed in the last iteration;
          **if** $\frac{w}{t} > maxW$ *and* $adjL \leq 0$ **then** $adjL$++ ;
          **else if** $\frac{w}{t} < minW$ *and* $adjL = 0$ **then** $adjL$-- ;
          **let** $\delta^\lambda|_{l_L}$ be the prefix of $\delta^\lambda$ up to label $l_L$;
          **send** $(z_0, b, \delta^\lambda|_{l_L})$ to slave $s$;
          $b$++;
      *// Possibly adjust value of $L$*
      **if** $adjL > 0$ *or* ($adjL < 0$ *and* $\hat{d} = d$) **then**
        $L \leftarrow \max(1, \min(h-1, L + adjL))$;
        $adjL \leftarrow 0$;
  **wait** until all slaves become idle

**Algorithm 1:** Orchestrator.

**function** *Slave*($\mathcal{D}, h, L, S, s$)
  **Input**: $\mathcal{D} = (Z, d, \mathit{dist}, \mathit{adm}, Z_I, Z_F)$, a DG
  **Input**: $h$, bounded length for disturbance traces
  **Input**: $L$, level of the search tree below which exploration is delegated to slaves
  **Input**: $S$, number of available slaves
  **Input**: $s \in [1, S]$, id of this slave
  $\lambda \leftarrow s + 1$; *// next label to be used*
  $\Delta_s \leftarrow$ empty sequence; *// output disturbance traces*
  **while** *true* **do**
    *// slave is idle*
    **wait** for a message $(z_0, b, \delta^\lambda|_{l_L})$ from Orchestrator;
    *//* $\delta^\lambda|_{l_L} = l_0, d_0, l_1, d_1, \ldots, l_L$
    **let** $\tilde{\delta}^\lambda$ be an array of variables $\tilde{l}_0, \tilde{d}_0, \tilde{l}_1, \tilde{d}_1, \ldots, \tilde{l}_L, \tilde{d}_L, \ldots, \tilde{l}_h$;
    *// start computation bunch* $b$
    *stack* $\leftarrow$ empty stack;
    $z \leftarrow z_0$;
    *// follow* $\delta^\lambda|_{l_L}$ *to reach root of requested subtree & copy it into* $\tilde{\delta}^\lambda$
    **for** $j \leftarrow 0$ *to* $L - 1$ **do**
      $\tilde{l}_j \leftarrow l_j$; $\tilde{d}_j \leftarrow d_j$; $z \leftarrow \mathit{dist}(z, \tilde{d}_j)$;
    $\tilde{l}_L \leftarrow l_L$;
    *// start DFS from there*
    **for** $d'$ *from* $d$ *downto* $0$ **do**
      *push*(*stack*, $(z, d'$,L$)$)
    **while** *stack is not empty* **do**
      $(z, \hat{d}, j) \leftarrow$ *pop*(*stack*);
      **if** $\mathit{adm}(z, \hat{d})$ **then**
        $\hat{z} \leftarrow \mathit{dist}(z, \hat{d})$; *//* $\hat{z}$ *is at depth* $j + 1$
        $\tilde{d}_j \leftarrow \hat{d}$;
        $\tilde{l}_{j+1} \leftarrow \lambda$; $\lambda \leftarrow \lambda + S + 1$;
        **if** $j < h$ **then**
          **for** $d'$ *from* $d$ *downto* $0$ **do** *push*(*stack*, $(\hat{z}, d', j + 1)$) ;
        **else if** $z \in Z_F$ **then** append $(b, \tilde{\delta}^\lambda)$ to $\Delta_s$ ;

**Algorithm 2:** Slave.

# 5. Experiments

In this section we experimentally evaluate SyLVaaS, and in particular our new parallel disturbance generation algorithm of Section 4 and the cloud deployment of the overall Verification as a Service (VaaS) infrastructure.

## 5.1. SyLVaaS Experimental Deployment

We deployed SyLVaaS on a cluster of Linux heterogeneous machines, whose configurations are shown in Table 1. We used a maximum number of 89 CPU cores (7 out of the 8 available cores for machines of categories A and B, 15 out of the 16 available cores for machines of category C, and 1 out of the 2 available cores for the machine of category Z). The single Orchestrator process was always run on the single used core of the single machine of category Z.

The SyLVaaS web interface application resides on a yet another host (a tiny virtual machine), external to the cluster and directly connected to the Internet.

## 5.2. Case Studies

We experiment with case studies consisting of disturbance models related to the System Level Formal Verification (SLFV) of two system models included in the Simulink distribution, namely the Inverted Pendulum on a Cart (IPC) and the Fuel Control System (FCS). For each system model, we define two disturbance models, whose properties are summarised in Table 2.

### 5.2.1. 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 Figure 8)

The IPC controller (actually a control software) senses the angular position $\theta$ 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 $\theta$, 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 $\theta$). 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 Figure 9. Overall, the Simulink block diagram consists of 52 blocks.

The system level property that we verify is that after 2 seconds the pendulum is in upright position, i.e., angle $\theta$ is always between $[-0.1, 0.1]$. The monitor checking for this property is shown in the lower box of Figure 9.

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).

| machine id | Type and num. of CPUs | CPUs frequency | total num. of cores | overall RAM | machine category |
|---|---|---|---|---|---|
| 0 | 1 × Intel(R) Celeron(R) | 2.27 GHz | 2 | 4 GB | Z |
| 1 | 2 × Intel(R) Xeon(R) | 2.83 GHz | 8 | 8 GB | A |
| 2 | 2 × Intel(R) Xeon(R) | 2.83 GHz | 8 | 8 GB | A |
| 3 | 2 × Intel(R) Xeon(R) | 2.66 GHz | 8 | 32 GB | B |
| 4 | 2 × Intel(R) Xeon(R) | 2.66 GHz | 8 | 32 GB | B |
| 5 | 2 × Intel(R) Xeon(R) | 2.27 GHz | 16 | 24 GB | C |
| 6 | 2 × Intel(R) Xeon(R) | 2.27 GHz | 16 | 24 GB | C |
| 7 | 2 × Intel(R) Xeon(R) | 2.27 GHz | 16 | 24 GB | C |
| 8 | 2 × Intel(R) Xeon(R) | 2.27 GHz | 16 | 24 GB | C |

Table 1: Configuration of our cluster machines.

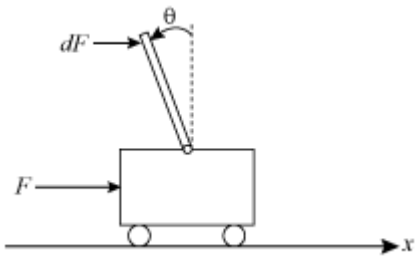



Figure 8: Inverted Pendulum on a Cart (IPC) (from mathworks.com).

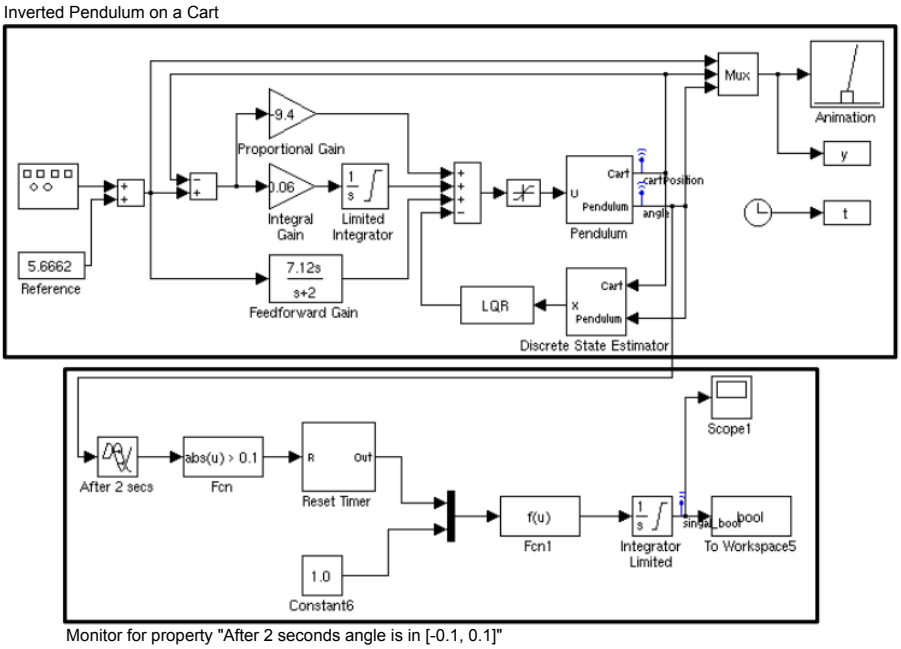



Figure 9: Simulink block diagram for Inverted Pendulum on a Cart (from mathworks.com) with an embedded property monitor.

We consider two disturbance models for the IPC, $\mathcal{D}^1_{\text{IPC}}$ and $\mathcal{D}^2_{\text{IPC}}$. Model $\mathcal{D}^1_{\text{IPC}}$ has a horizon of $h = 90$ and defines $3\,208\,276$ disturbance traces. Model $\mathcal{D}^2_{\text{IPC}}$ is defined extending $\mathcal{D}^1_{\text{IPC}}$ with more complex operational scenarios and defines $35\,641\,501$ disturbance traces over a horizon of $h = 200$. For both models we set $\tau$ (quantum between disturbances) to $0.1$ sec.

Relevant properties of disturbance models $\mathcal{D}^1_{\text{IPC}}$ and $\mathcal{D}^2_{\text{IPC}}$ are shown in Table 2. A more detailed description of such models is not relevant for the evaluation of our experiments below. We only point out that defining such disturbance models and encoding them in the language offered by CMurphi (and taken as input by SyLVaaS) would take about 1 or 2 days of an average verification engineer with some knowledge in formal methods.

### 5.2.2. 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 [9, 10, 11, 12, 3, 5]).

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 [13]. Overall, the Simulink block diagram consists of 246 blocks.

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 Figure 10).

We consider two disturbance models for the FCS, $\mathcal{D}^1_{\text{FCS}}$ and $\mathcal{D}^2_{\text{FCS}}$. Model $\mathcal{D}^1_{\text{FCS}}$ has a horizon of $h = 100$ and defines $4\,023\,955$ disturbance traces. Model $\mathcal{D}^2_{\text{FCS}}$ is defined extending $\mathcal{D}^1_{\text{FCS}}$ 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 $1$ sec.

Relevant properties of disturbance models $\mathcal{D}^1_{\text{FCS}}$ and $\mathcal{D}^2_{\text{FCS}}$ are shown in Table 2. A more detailed description of such models is not relevant for the evaluation of our experiments below and can be found in [3]. Again, we point out that defining such disturbance models and encoding them in the language offered by CMurphi (and taken as input by SyLVaaS) would take about 1 or 2 days of an average verification engineer with some knowledge in formal methods.

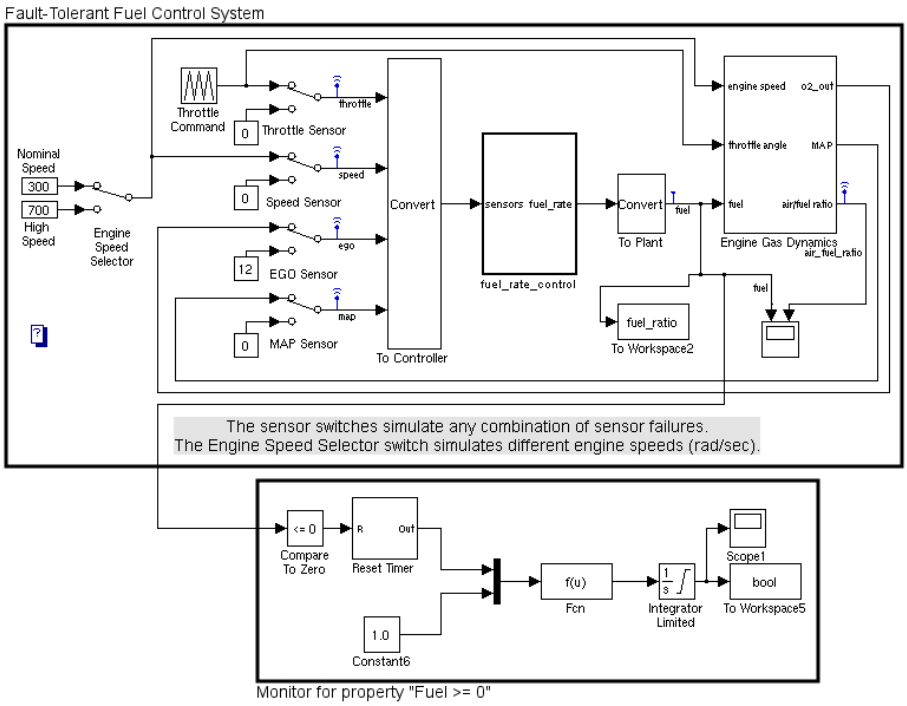



Figure 10: Simulink/Stateflow representation of the Fuel Control System (from mathworks.com) with an embedded property monitor.

## 5.3. Experimental Results

In this section we outline our experimental results on the four disturbance models presented in Section 5.2.

### 5.3.1. Parallel Disturbance Trace Generation

Table 4 shows the time needed by SyLVaaS to generate the disturbance traces entailed by our four disturbance models, when using a *varying* number $S$ of parallel Slaves.

As specified in Section 4.3, the Orchestrator algorithm requires in input the following items:

- The disturbance model $\mathcal{D}$ and the horizon $h$. We use all four disturbance models listed in Table 2, with the corresponding horizons.
- The starting value for the level (depth) $L$ to which the Orchestrator bounds its search and triggers a Slave. We set this value to $\frac{h}{2}$ after preliminary experiments.
- The number of Slaves $S$. We set this value so that our cluster is used to the 33%, 66% and 100% of its total available number of cores. To neutralise biases due to the heterogeneity of our cluster machines, we kept fixed the ratios between the different types of cores listed in Table 1. The

| SUV | dist. model | time quantum ($\tau$) | horizon ($h$) | #traces |
|---|---|---|---|---|
| IPC | $\mathcal{D}^1_{\text{IPC}}$ | 0.1 sec | 90 | 3 208 276 |
| IPC | $\mathcal{D}^2_{\text{IPC}}$ | 0.05 sec | 200 | 35 641 501 |
| FCS | $\mathcal{D}^1_{\text{FCS}}$ | 1 sec | 100 | 4 023 955 |
| FCS | $\mathcal{D}^2_{\text{FCS}}$ | 1 sec | 200 | 12 948 712 |

Table 2: Disturbance models.

| # Slaves ($S$) | # cores per category | | |
|---|---|---|---|
| | **A** | **B** | **C** |
| 1 | – weighted average – | | |
| 23 | 4 | 4 | 15 |
| 51 | 8 | 8 | 35 |
| 88 | 14 | 14 | 60 |

Table 3: Allocation of the Slaves among our cluster computational cores.

chosen allocation of the Slaves on the available cores for the various experimental deployments is shown in Table 3.

- $minW$, $maxW$ as the minimum and maximum percentage of wall-clock time to be spent waiting for a Slave. After preliminary experiments, we set these values to $1\%$ and $60\%$ respectively.

In order evaluate the scalability of our parallel disturbance trace generation algorithm with the number $S$ of Slaves, we have also run the algorithm with only one Slave (*sequential algorithm*, $S = 1$). In order to neutralise biases due to the heterogeneity of our cluster machines, we have performed 3 runs of the sequential algorithm, with the single Slave running on a core of a machine of category A, B and C. We then computed the sequential time as the weighted average of these three running times, where the weights are the ratios of the number of cores available for the execution of a Slave on machines of each category. Namely:

$$\sum_{c \in \{A,B,C\}} \text{ratio}(c) \times \text{seq\_time}(c)$$

where ratio$(c)$ is the ratio of the overall cores of category $c \in \{A, B, C\}$ available for Slaves execution on machines of category $c$ (i.e., $14/88$, $14/88$ and $60/88$ for categories A, B and C respectively –remember that we use up to $n - 1$ cores on a machine with $n$ cores), and seq_time$(c)$ is the time of our sequential algorithm in which the single Slave was run on a core of a machine of category $c$ (the Orchestrator always runs on the single available core of category $Z$).

For each disturbance model and each value for $S$, Table 4 reports the overall time for generating the whole set of disturbance traces (columns "*time*"), the number of computation bunches executed by the algorithm as well as *speedup* and *efficiency* with respect to the execution time of the sequential algorithm (the rows in Table 4 referring to $S = 1$).

As usual in the evaluation of parallel algorithms, for each value of $S$, the *speedup* is defined as $t_1/t_S$, where $t_1$ and $t_S$ are, respectively, the execution times of our disturbance trace generation algorithm when using 1 (*sequential algorithm*) and $S$ parallel Slaves. For each value of $S$, the *efficiency* is computed as the ratio between the speedup and $S$.

As a result, efficiency is never below 75%, and it is often above 80%, showing that our parallel disturbance trace generation algorithm scales well with the number of available Slaves. The observed lack of efficiency, mostly due to network delays, is typical in a cluster setting. To this end, we note that high-performance parallel simulation typically has efficiency values in the range 40%–80% (see, e.g., [14]). Accordingly, an efficiency of about 75%–80% is to be considered state-of-the-art.

Finally, Figure 11 shows how value of $L$ (*delegation level*, i.e., the depth of the computation path tree at which the Orchestrator delegates the exploration to an available Slave) evolves for our case studies. Namely, such plots show how $L/h$ (on the y-axis) varies as a function of the completion time percentage (on the x-axis), for each of the possible values of $S$. As a result, we have that $L$, in our case studies, tends to decrease as the completion time increases. This is due to the fact that, on average, our disturbance models are "left-unbalanced", in that admissible disturbance traces lie more frequently in the left part of the computation path tree.

Hence, in our case studies, the average time spent by a Slave in completing a computation bunch decreases during time due to pruning. To this end we remind that the exploration is done in *lexicographic order*, as this simplifies trace labelling and the forthcoming computation of optimised simulation campaigns, see [5].

Of course, the actual time evolution of value $L$ strongly depends on the structure of the disturbance model at hand. What is important here is that the Orchestrator effectively mitigates any bias during exploration by quickly reacting to any observed unbalanced workload among Slaves.

### 5.3.2. SyLVaaS Complete Workflow

Table 5 reports the overall SyLVaaS response time (summing up trace generation, splitting, and simulation campaign optimisation times, column "*overall time*"), for each disturbance model and each value for $k$. Results in Table 5 have been obtained using $S = 88$ Slaves during trace generation and 89 cores to compute the $k$ simulation campaigns (thus, on average, each core computed $k/89$ campaigns).

### 5.3.3. Download of Simulation Campaigns

SyLVaaS stores simulation campaigns computed as above in .zip archives which are then downloaded by the user. In our experiments, the size of such files is up to the order of *a few Gigabytes*. Hence, their download into the user cluster can be done seamlessly over a standard broad-band Internet connection.

| **# Slaves ($S$)** | **time (h:m:s)** | **speedup** | **efficiency** |
|---|---|---|---|
| 1 | 0:10:35 | 1.00 × | 100.00% |
| 23 | 0:0:32 | 19.37 × | 84.00% |
| 51 | 0:0:14 | 45.01 × | 88.00% |
| 88 | 0:0:9 | 70.25 × | 80.00% |

(a) Inverted Pendulum on a Cart (IPC), disturbance model $\mathcal{D}^1_{\text{IPC}}$

| **# Slaves ($S$)** | **time (h:m:s)** | **speedup** | **efficiency** |
|---|---|---|---|
| 1 | 4:25:52 | 1.00 × | 100.00% |
| 23 | 0:14:14 | 18.67 × | 81.00% |
| 51 | 0:6:5 | 43.68 × | 86.00% |
| 88 | 0:4:0 | 66.22 × | 75.00% |

(b) Inverted Pendulum on a Cart (IPC), disturbance model $\mathcal{D}^2_{\text{IPC}}$

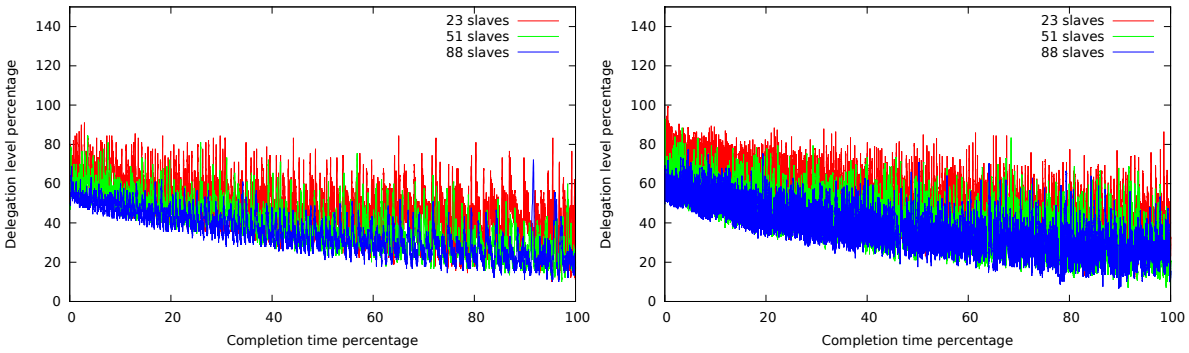
| **# Slaves ($S$)** | **time (h:m:s)** | **speedup** | **efficiency** |
|---|---|---|---|
| 1 | 0:33:14 | 1.00 × | 100.00% |
| 23 | 0:1:34 | 21.06 × | 92.00% |
| 51 | 0:0:41 | 48.64 × | 95.00% |
| 88 | 0:0:26 | 75.39 × | 86.00% |

(c) Fuel Control System (FCS), disturbance model $\mathcal{D}^1_{\text{FCS}}$

| **# Slaves ($S$)** | **time (h:m:s)** | **speedup** | **efficiency** |
|---|---|---|---|
| 1 | 4:33:24 | 1.00 × | 100.00% |
| 23 | 0:13:13 | 20.68 × | 90.00% |
| 51 | 0:5:52 | 46.57 × | 91.00% |
| 88 | 0:4:7 | 66.23 × | 75.00% |

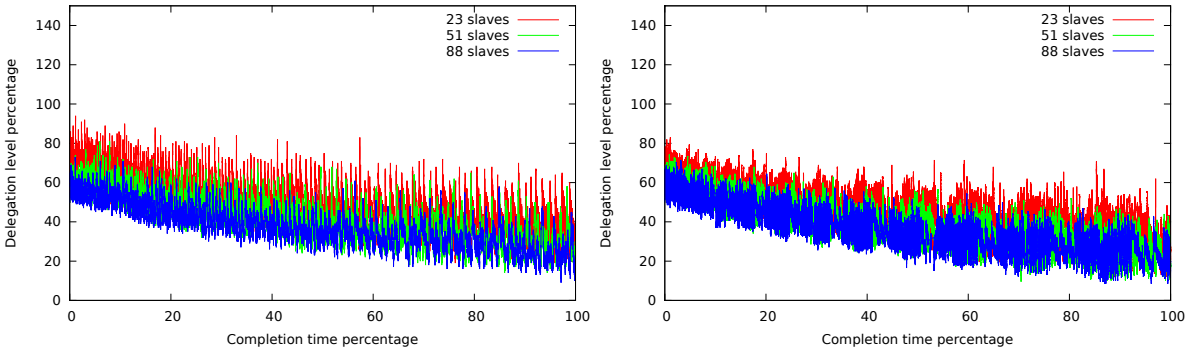
(d) Fuel Control System (FCS), disturbance model $\mathcal{D}^2_{\text{FCS}}$

Table 4: Parallel generation of disturbance traces: completion time, speedup and efficiency.




(a) Disturbance model $\mathcal{D}^1_{\text{IPC}}$ (b) Disturbance model $\mathcal{D}^2_{\text{IPC}}$

Inverted Pendulum on a Cart (IPC)




(c) Disturbance model $\mathcal{D}^1_{\text{FCS}}$ (d) Disturbance model $\mathcal{D}^2_{\text{FCS}}$

Fuel Control System (FCS)

Figure 11: Parallel generation of disturbance traces: time evolution of delegation level $L$.

| # slices ($k$) | overall time (h:m:s) |
|---|---|
| 128 | 0:8:0 |
| 256 | 0:12:53 |
| 512 | 0:21:42 |

(a) IPC, disturbance model $\mathcal{D}^1_{\text{IPC}}$

| # slices ($k$) | overall time (h:m:s) |
|---|---|
| 128 | 7:9:52 |
| 256 | 6:6:18 |
| 512 | 7:25:17 |

(b) IPC, disturbance model $\mathcal{D}^2_{\text{IPC}}$

| # slices ($k$) | overall time (h:m:s) |
|---|---|
| 128 | 0:11:55 |
| 256 | 0:15:37 |
| 512 | 0:24:45 |

(c) FCS, disturbance model $\mathcal{D}^1_{\text{FCS}}$

| # slices ($k$) | overall time (h:m:s) |
|---|---|
| 128 | 1:30:41 |
| 256 | 2:8:2 |
| 512 | 3:59:46 |

(d) FCS, disturbance model $\mathcal{D}^2_{\text{FCS}}$

Table 5: SyLVaaS time of the entire workflow.

## 6. Related Work

The papers closest to ours are [3, 4, 5, 6], where the algorithms underlying SyLVaaS workflow are presented. The work in [3, 4, 5, 6] presents a parallel approach to System Level Formal Verification (SLFV) for Cyber-Physical Systems (CPSs) (i.e., for the class of hybrid systems handled by a simulator like Simulink). This is done by effectively decoupling the computation of the set of system runs (operational scenarios) to be exercised during the Hardware In the Loop Simulation (HILS) based SLFV from their actual simulation. In this paper we complement such results by focusing on parallelising the most intensive computation step within the SyLVaaS workflow, namely the generation of the set of all operational scenarios.

System Verification as a Service (also known as Model Checking in the Cloud) is still in its infancy. In [15], it is argued that ideas may be borrowed from workflow modelling, management and analysis of business processes. In [16] a Map-Reduce algorithm for verification of CTL formulas on a cloud system is proposed. Moreover, panels to discuss on how to set up a reliable Verification as a Service (VaaS) tool are ongoing in major conferences (see, e.g., recent proceedings of the International Conference on Formal Methods in Computer-Aided Design, FMCAD). However, none of such works propose an implemented and available tool, with the features described in Section 1.2. We also point out that "verification as a service" is sometimes used to refer to a *consulting service*, where a company *rents* formal verification experts to another company in order to carry out a certain verification activity. Of course this is not our meaning for VaaS and we note that such a consultant based approach does not provide the same level of Intellectual Property (IP) protection as our SyLVaaS based approach. Furthermore, our proposed approach is fully automatic and does not require formal verification experts. Thus, to the best of our knowledge, SyLVaaS is the first tool providing a genuine Verification as a Service (VaaS) approach.

HILS-based SLFV has been addressed in [17]. However the approach presented in [17] rests on

closely coupling a simulator (SIMSAT) with a model checker (CMurphi, [7]). Accordingly, such an approach cannot be directly used to develop the VaaS approach described here.

Formal verification of Simulink models has been widely investigated, examples are in [18, 19, 20]. 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 [9, 10], as well as for the exhaustive HILS based approach in [3]. Simulation based best-effort *falsification* methods able to handle any Simulink/Stateflow model have been investigated in [21, 22]. *Annotated* Stateflow models comprising both discrete and continuous variables can be analysed with simulation based tools like C2E2 [23]. We differ from C2E2 by providing a black-box approach that, furthermore, does not require model annotations.

*Symbolic approaches* (typically based on polyhedra or SMT solving) to hybrid system verification have also been widely investigated. Although they are not black-box approaches, for sake of completeness we provide a glimpse on some of the available tools in such a context. Timed automata (i.e., hybrid systems whose continuous variables have time derivative equal to 1) can be analysed with UPPAAL [24]. Linear hybrid automata (see, e.g., [25]) can be analysed with HyTech [26]. Piecewise affine hybrid systems can be analysed with symbolic model checkers like PHAVer [27] and SpaceEx [28, 29, 30]. A symbolic model checker capable of handling nonlinear hybrid systems is presented in [31]. Currently, with respect to our proposed approach, the main limitations of symbolic approaches are: (i) they are not black-box, and (ii) they can handle only hybrid systems of moderate size (whereas our approach does not depend on the size of the system to be verified). Finally, within such a symbolic context, we note that, while we use automata to define the set of scenarios to be simulated, temporal logic could also be used to that end. An example is in [32].

*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 Omission Probability (OP). Upon detection of an error, a random model checker stops returning a counterexample. Random model checking algorithms have been investigated, e.g., in [33, 34, 35, 36]. 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 (see, e.g., [34]) 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.

*Probabilistic* (see, e.g., [37, 38]) and, more specifically, *simulation-based statistical model checking* approaches (see, e.g., [39, 40, 41, 33, 42, 10, 9, 43, 44, 45]) are closely related to our work. In particular, [10] addresses statistical model checking of Simulink models and presents experimental results on the Simulink Fuel Control System 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 (i.e., we are looking for the *worst case scenario*). (iv) None of the available simulation-based statistical model checking approaches addresses the problem of the op-

timisation of the simulation campaigns, which is an essential step to make our *parallel exhaustive* HILS based model checking viable.

Synergies between simulation and formal methods have been also widely investigated in digital hardware verification. Examples are in [46, 47, 48, 49] and citations thereof. The main differences between the above approaches and ours are: (i) they focus on finite state systems whereas we focus on infinite state systems (namely, hybrid systems); (ii) they are *white-box* (requiring availability of the system model) whereas we are *black-box*. As for hybrid systems, synergies between explicit and symbolic model checking methods have been investigated in [50, 51, 52, 53, 54, 55, 56, 57] in the context of automatic *synthesis* of controllers for discrete time linear hybrid systems.

Parallel algorithms for explicit state exploration have been widely investigated. Examples are in [58, 59, 60, 61, 62, 63]. The main difference with our approach is that all the above works focus on parallelising the state space exploration engine by devising techniques to minimise locking of the visited state hash table. Conversely, we leave unchanged the state space exploration engine (the simulator in our context), split the set of simulation scenarios into equal size subsets to be simulated on different processors, and stop verification as soon as one of such processors finds an error, thereby enabling an *embarrassing parallel* approach.

## 7. Conclusions

We have presented SyLVaaS, a Web-based software-as-a-service tool for HILS-based System Level Formal Verification (SLFV). Such a tool allows verification engineers to obtain from a Web service the most important part of their HILS campaigns, i.e. a set of simulation campaigns to exercise the System Under Verification (SUV) on *all* the relevant operational scenarios (disturbance traces).

As the simulation campaigns are executed at the user premises, SyLVaaS provides full Intellectual Property (IP) protection for both the SUV model, the property to be verified, and the user verification flow. The simulation may be carried out in parallel on a user cluster whose machines have Simulink installed.

To achieve a short response time and increase the quality of service provided by SyLVaaS, we also proposed a new algorithm to parallelise the most computationally intensive part of the SyLVaaS workflow, i.e., the generation of disturbance traces. As the other step performed by SyLVaaS (computation of optimised simulation campaigns) already exploits an embarrassingly parallel algorithm, with our new parallel disturbance trace generator the entire SyLVaaS workflow can benefit of a cluster of machines at the SyLVaaS cloud infrastructure.

To the best of our knowledge, SyLVaaS is the first Web-based software-as-a-service tool for HILS-based SLFV.

## Acknowledgements

This research has received funding from the EU Seventh Framework Programme (FP7/2007-2013) under grant agreements n. 317761 (SmartHG) and n. 600773 (PAEON).

# References


[1] Mancini T, Mari F, Massini A, Melatti I, Tronci E. SyLVaaS: System Level Formal Verification as a Service. In: Proceedings of 23rd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing (PDP 2015). IEEE; 2015. p. 476–483.

[2] Alur R. Formal Verification of Hybrid Systems. In: Proceedings of 11th International Conference on Embedded Software (EMSOFT 2011). ACM; 2011. p. 273–278. doi:10.1145/2038642.2038685.

[3] Mancini T, Mari F, Massini A, Melatti I, Merli F, Tronci E. System Level Formal Verification via Model Checking Driven Simulation. In: Proceedings of 25th International Conference on Computer Aided Verification (CAV 2013). vol. 8044 of Lecture Notes in Computer Science. Springer; 2013. p. 296–312. doi:10.1007/978-3-642-39799-8_21.

[4] Mancini T, Mari F, Massini A, Melatti I, Tronci E. Anytime System Level Verification via Random Exhaustive Hardware In The Loop Simulation. In: Proceedings of 17th Euromicro Conference on Digital System Design (DSD 2014). IEEE; 2014. p. 236–245.

[5] Mancini T, Mari F, Massini A, Melatti I, Tronci E. System Level Formal Verification via Distributed Multi-Core Hardware in the Loop Simulation. In: Proceedings of 22nd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing (PDP 2014). IEEE; 2014. p. 734–742. doi:10.1109/PDP.2014.32.

[6] Mancini T, Mari F, Massini A, Melatti I, Tronci E. Anytime System Level Verification via Parallel Random Exhaustive Hardware in the Loop Simulation. Microprocessors and Microsystems. 2016;41:12–28. doi:10.1016/j.micpro.2015.10.010.

[7] Della Penna G, Intrigila B, Melatti I, Tronci E, Venturini Zilli M. Exploiting Transition Locality in Automatic Verification of Finite State Concurrent Systems. International Journal on Software Tools for Technology Transfer. 2004;6(4):320–341. doi:10.1007/s10009-004-0149-6.

[8] Sontag ED. Mathematical Control Theory: Deterministic Finite Dimensional Systems (2nd Ed.). Springer; 1998.

[9] Clarke EM, Donzé A, Legay A. On Simulation-Based Probabilistic Model Checking of Mixed-Analog Circuits. Formal Methods in System Design. 2010;36(2):97–113. doi:10.1007/s10703-009-0076-y.

[10] Zuliani P, Platzer A, Clarke EM. Bayesian Statistical Model Checking with Application to Stateflow/Simulink Verification. Formal Methods in System Design. 2013;43(2):338–367. doi:10.1007/s10703-013-0195-3.

[11] Kim YJ, Kim M. Hybrid Statistical Model Checking Technique for Reliable Safety Critical Systems. In: Proceedings of 23rd IEEE International Symposium on Software Reliability Engineering. IEEE; 2012. p. 51–60.

[12] Kim YJ, Choi O, Kim M, Baik J, Kim TH. Validating Software Reliability Early through Statistical Model Checking. IEEE Software. 2013;30(3):35–41. doi:10.1109/MS.2013.24.

[13] Schrammel P, Kroening D, Brain M, Martins R, Teige T, Bienmüller T. Incremental Bounded Model Checking for Embedded Software (Extended Version). CoRR. 2014;abs/1409.5872.

[14] Phillips JC, Sun Y, Jain N, Bohm EJ, Kalé LV. Mapping to Irregular Torus Topologies and Other Techniques for Petascale Biomolecular Simulation. In: Proceedings of International Conference for High Performance Computing, Networking, Storage and Analysis (SC 2014). IEEE; 2014. p. 81–91.

[15] Schaefer I, Sauer T. Towards Verification as a Service. In: Eternal Systems. vol. 255 of Communications in Computer and Information Science. Springer; 2012. p. 16–24. doi:10.1007/978-3-642-28033-7_2.

[16] Bellettini C, Camilli M, Capra L, Monga M. Distributed CTL Model Checking in the Cloud. CoRR. 2013;abs/1310.6670.

[17] Verzino G, Cavaliere F, Mari F, Melatti I, Minei G, Salvo I, et al. Model Checking Driven Simulation of Sat Procedures. In: 12th International Conference on Space Operations (SpaceOps 2012); 2012. doi:10.2514/6.2012-1275611.

[18] Tripakis S, Sofronis C, Caspi P, Curic A. Translating Discrete-Time Simulink to Lustre. ACM Transactions on Embedded Computing Systems. 2005;4(4):779–818. doi:10.1145/1113830.1113834.

[19] Meenakshi B, Bhatnagar A, Roy S. Tool for Translating Simulink Models into Input Language of a Model Checker. In: Proceedings of 8th International Conference on Formal Engineering Methods (ICFEM 2006). Springer; 2006. p. 606–620. doi:10.1007/11901433_33.

[20] Whalen MW, Cofer DD, Miller SP, Krogh BH, Storm W. Integration of Formal Analysis into a Model-Based Software Development Process. In: Proceedings of 12th International Workshop on Formal Methods for Industrial Critical Systems (FMICS 2007). vol. 4916 of Lecture Notes in Computer Science. Springer; 2007. p. 68–84. doi:10.1007/978-3-540-79707-4_7.

[21] Abbas H, Fainekos G, Sankaranarayanan S, Ivančić F, Gupta A. Probabilistic Temporal Logic Falsification of Cyber-Physical Systems. ACM Transactions on Embedded Computing Systems. 2013;12(2s):95:1–95:30. doi:10.1145/2465787.2465797.

[22] Zutshi A, Sankaranarayanan S, Deshmukh JV, Jin X. Symbolic-Numeric Reachability Analysis of Closed-Loop Control Software. In: Proceedings of 19th ACM International Conference on Hybrid Systems: Computation and Control (HSCC 2016). ACM; 2016. p. 135–144.

[23] Duggirala PS, Mitra S, Viswanathan M, Potok M. C2E2: A Verification Tool for Stateflow Models. In: Proceedings of 21st International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2015). vol. 9035 of Lecture Notes in Computer Science. Springer; 2015. p. 68–82. doi:10.1007/978-3-662-46681-0_5.

[24] Bengtsson J, Larsen K, Larsson F, Pettersson P, Yi W. UPPAAL — A Tool Suite for Automatic Verification of Real-Time Systems. In: Proceedings of Hybrid Systems III: Verification and Control. vol. 1066 of Lecture Notes in Computer Science. Springer; 1996. p. 232–243. doi:10.1007/BFb0020949.

[25] Alur R, Courcoubetis C, Halbwachs N, Henzinger TA, Ho PH, Nicollin X, et al. The Algorithmic Analysis of Hybrid Systems. Theoretical Computer Science. 1995;138(1):3–34. doi:10.1016/0304-3975(94)00202-T.

[26] Henzinger TA, Ho PH, Wong-toi H. HyTech: A Model Checker for Hybrid Systems. International Journal on Software Tools for Technology Transfer. 1997;1:460–463.

[27] Frehse G. PHAVer: Algorithmic Verification of Hybrid Systems Past HyTech. International Journal on Software Tools for Technology Transfer. 2008;10(3):263–279. doi:10.1007/s10009-007-0062-x.

[28] Frehse G, Le Guernic C, Donzé A, Cotton S, Ray R, Lebeltel O, et al. SpaceEx: Scalable Verification of Hybrid Systems. In: Proceedings of 23rd International Conference on Computer Aided Verification (CAV 2011). vol. 6806 of Lecture Notes in Computer Science. Springer; 2011. p. 379–395.

[29] Cimatti A, Mover S, Tonetta S. SMT-Based Scenario Verification for Hybrid Systems. Formal Methods in System Design. 2013;42(1):46–66. doi:10.1007/s10703-012-0158-0.

[30] Cimatti A, Griggio A, Mover S, Tonetta S. HyComp: An SMT-Based Model Checker for Hybrid Systems. In: Proceedings of 21st International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2015). vol. 9035 of Lecture Notes in Computer Science. Springer; 2015. p. 52–67. doi:10.1007/978-3-662-46681-0_4.

[31] Bak S, Bogomolov S, Henzinger TA, Johnson TT, Prakash P. Scalable Static Hybridization Methods for Analysis of Nonlinear Systems. In: Proceedings of 19th ACM International Conference on Hybrid Systems: Computation and Control (HSCC 2016). ACM; 2016. p. 155–164. doi:10.1145/2883817.2883837.

[32] Cimatti A, Roveri M, Tonetta S. HRELTL. Information and Computation. 2015;245(C):54–71. doi:10.1016/j.ic.2015.06.006.

[33] Sen K, Viswanathan M, Agha G. On Statistical Model Checking of Stochastic Systems. In: Proceedings of 17th International Conference on Computer Aided Verification (CAV 2005). vol. 3576 of Lecture Notes in Computer Science. Springer; 2005. p. 266–280.

[34] Grosu R, Smolka SA. Monte Carlo Model Checking. In: Proceedings of 11th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2005). vol. 3440 of Lecture Notes in Computer Science. Springer; 2005. p. 271–286. doi:10.1007/978-3-540-31980-1_18.

[35] Tronci E, Della Penna G, Intrigila B, Venturini Zilli M. A Probabilistic Approach to Automatic Verification of Concurrent Systems. In: Proceedings of 8th Asia-Pacific Software Engineering Conference (APSEC 2001). IEEE; 2001. p. 317–324. doi:10.1109/APSEC.2001.991495.

[36] Sivaraj H, Gopalakrishnan G. Random Walk Based Heuristic Algorithms for Distributed Memory Model Checking. Electronic Notes in Theoretical Computer Science. 2003;89(1):51–67. doi:10.1016/S1571-0661(05)80096-9.

[37] Della Penna G, Intrigila B, Melatti I, Tronci E, Venturini Zilli M. Finite Horizon Analysis of Markov Chains with the Murphi Verifier. International Journal on Software Tools for Technology Transfer. 2006;8(4–5):397–409. doi:10.1007/s10009-005-0216-7.

[38] Jansen DN, Katoen JP, Oldenkamp M, Stoelinga MIA, Zapreev IS. How Fast and Fat Is Your Probabilistic Model Checker? An Experimental Performance Comparison. In: Proceedings of Hardware and Software: Verification and Testing, 3rd International Haifa Verification Conference (HVC 2007). vol. 4899 of Lecture Notes in Computer Science. Springer; 2008. p. 69–85.

[39] Younes HLS, Simmons RG. Probabilistic Verification of Discrete Event Systems Using Acceptance Sampling. In: Proceedings of 14th International Conference on Computer Aided Verification (CAV 2002). vol. 2404 of Lecture Notes in Computer Science. Springer; 2002. p. 223–235. doi:10.1007/3-540-45657-0_17.

[40] Younes HLS. Ymer: A Statistical Model Checker. In: Proceedings of 17th International Conference on Computer Aided Verification (CAV 2005). vol. 3576 of Lecture Notes in Computer Science. Springer; 2005. p. 429–433. doi:10.1007/11513988_43.

[41] Younes HLS. Probabilistic Verification for "Black-Box" Systems. In: Proceedings of 17th International Conference on Computer Aided Verification (CAV 2005). vol. 3576 of Lecture Notes in Computer Science. Springer; 2005. p. 253–265. doi:10.1007/11513988_25.

[42] Younes HLS, Kwiatkowska MZ, Norman G, Parker D. Numerical vs. Statistical Probabilistic Model Checking. International Journal on Software Tools for Technology Transfer. 2006;8(3):216–228. doi:10.1007/s10009-005-0187-8.

[43] David A, Larsen KG, Legay A, Mikučionis M, Wang Z. Time for Statistical Model Checking of Real-Time Systems. In: Proceedings of 23rd International Conference on Computer Aided Verification (CAV 2011). vol. 6806 of Lecture Notes in Computer Science. Springer; 2011. p. 349–355.

[44] Tronci E, Mancini T, Salvo I, Sinisi S, Mari F, Melatti I, et al. Patient-Specific Models from Inter-Patient Biological Models and Clinical Records. In: Proceedings of 14th International Conference on Formal Methods in Computer-Aided Design (FMCAD 2014). IEEE; 2014. p. 207–214.

[45] Mancini T, Mari F, Melatti I, Salvo I, Tronci E, Gruber J, et al. Demand-Aware Price Policy Synthesis and Verification Services for Smart Grids. In: Proceedings of 2014 IEEE International Conference on Smart Grid Communications (SmartGridComm 2014). IEEE; 2014. p. 794–799.

[46] Yang CH, Dill DL. Validation with Guided Search of the State Space. In: Proceedings of 35th Conference on Design Automation (DAC 1998). ACM; 1998. p. 599–604. doi:10.1145/277044.277201.

[47] Ho PH, Shiple T, Harer K, Kukula J, Damiano R, Bertacco V, et al. Smart Simulation using Collaborative Formal and Simulation Engines. In: Proceedings of 2000 IEEE/ACM International Conference on Computer-Aided Design (ICCAD 2000). IEEE; 2000. p. 120–126.

[48] Nanshi K, Somenzi F. Guiding Simulation with Increasingly Refined Abstract Traces. In: Proceedings of 43rd Conference on Design Automation (DAC 2006). ACM; 2006. p. 737–742. doi:10.1145/1146909.1147097.

[49] De Paula FM, Hu AJ. An Effective Guidance Strategy for Abstraction-Guided Simulation. In: Proceedings of 44th Conference on Design Automation (DAC 2007). ACM; 2007. p. 63–68. doi:10.1145/1278480.1278498.

[50] Mari F, Melatti I, Salvo I, Tronci E. Model Based Synthesis of Control Software from System Level Formal Specifications. ACM Transactions on Software Engineering and Methodology. 2014;23(1):1–42.

[51] Mari F, Melatti I, Salvo I, Tronci E. Undecidability of Quantized State Feedback Control for Discrete Time Linear Hybrid Systems. In: Proceedings of 9th International Colloquium on Theoretical Aspects of Computing (ICTAC 2012). vol. 7521 of Lecture Notes in Computer Science. Springer; 2012. p. 243–258. doi:10.1007/978-3-642-32943-2_19.

[52] Alimguzhin V, Mari F, Melatti I, Salvo I, Tronci E. On Model Based Synthesis of Embedded Control Software. In: Proceedings of 12th International Conference on Embedded Software (EMSOFT 2012). ACM; 2012. p. 227–236. doi:10.1145/2380356.2380398.

[53] Alimguzhin V, Mari F, Melatti I, Salvo I, Tronci E. Automatic Control Software Synthesis for Quantized Discrete Time Hybrid Systems. In: Proceedings of 51th IEEE Conference on Decision and Control (CDC 2012). IEEE; 2012. p. 6120–6125. doi:10.1109/CDC.2012.6426260.

[54] Alimguzhin V, Mari F, Melatti I, Salvo I, Tronci E. On-the-Fly Control Software Synthesis. In: Proceedings of 20th International SPIN Symposium on Model Checking of Software (SPIN 2013). vol. 7976 of Lecture Notes in Computer Science. Springer; 2013. p. 61–80. doi:10.1007/978-3-642-39176-7_5.

[55] Alimguzhin V, Mari F, Melatti I, Salvo I, Tronci E. A Map-Reduce Parallel Approach to Automatic Synthesis of Control Software. In: Proceedings of 20th International SPIN Symposium on Model Checking of Software (SPIN 2013). vol. 7976 of Lecture Notes in Computer Science. Springer; 2013. p. 43–60. doi:10.1007/978-3-642-39176-7_4.

[56] Mari F, Melatti I, Salvo I, Tronci E. Synthesis of Quantized Feedback Control Software for Discrete Time Linear Hybrid Systems. In: Proceedings of 22nd International Conference on Computer Aided Verification (CAV 2010). vol. 6174 of Lecture Notes in Computer Science. Springer; 2010. p. 180–195. doi:10.1007/978-3-642-14295-6_20.

[57] Della Penna G, Intrigila B, Tronci E, Venturini Zilli M. Synchronized Regular Expressions. Acta Informatica. 2003;39(1):31–70.

[58] Stern U, Dill DL. Parallelizing the Murphi Verifier. Formal Methods in System Design. 2001;18(2):117–129. doi:10.1023/A:1008771324652.

[59] Barnat J, Brim L, Černá I, Moravec P, Ročkai P, Šimeček P. DiVinE: a Tool for Distributed Verification. In: Proceedings of 18th International Conference on Computer Aided Verification (CAV 2006). vol. 4144 of Lecture Notes in Computer Science. Springer; 2006. p. 278–281. doi:10.1007/11817963_26.

[60] Melatti I, Palmer R, Sawaya G, Yang Y, Kirby RM, Gopalakrishnan G. Parallel and Distributed Model Checking in Eddy. International Journal on Software Tools for Technology Transfer. 2009;11(1):13–25. doi:10.1007/s10009-008-0094-x.

[61] Bingham B, Bingham J, De Paula FM, Erickson J, Singh G, Reitblatt M. Industrial Strength Distributed Explicit State Model Checking. In: Proceedings of 9th International Workshop on Parallel and Distributed Methods in Verification and 2nd International Workshop on High Performance Computational Systems Biology (PDMC-HIBI 2010). IEEE; 2010. p. 28–36. doi:10.1109/PDMC-HiBi.2010.13.

[62] Laarman A, van de Pol J, Weber M. Boosting Multi-Core Reachability Performance with Shared Hash Tables. In: Proceedings of 10th International Conference on Formal Methods in Computer-Aided Design (FMCAD 2010). IEEE; 2010. p. 247–255.

[63] Holzmann GJ. Parallelizing the SPIN Model Checker. In: Proceedings of 19th International SPIN Symposium on Model Checking of Software (SPIN 2012). vol. 7385 of Lecture Notes in Computer Science. Springer; 2012. p. 155–171. doi:10.1007/978-3-642-31759-0_12.