This paper appears in the proceedings of the 23rd Euromicro International Conference on Parallel, Distributed and Network-based processing (PDP 2015).

SyLVaaS: System Level Formal Verification as a Service

Toni Mancini, Federico Mari, Annalisa Massini, Igor Melatti, and Enrico Tronci
email:{tmancini, mari, massini, melatti, tronci}@di.uniroma1.it
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, variation 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 the verification problem outlined above.

SyLVaaS takes as input a high-level model 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 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., in a private cluster), SyLVaaS allows full Intellectual Property protection on the SUV model and the 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 industry-scale input related to the verification of the Fuel Control System (FCS) model in the Simulink distribution.

I. Introduction

Cyber-Physical Systems (CPSs) consist of hardware and software components and can be modelled as hybrid systems (see, e.g., [1] and citations thereof). System Level Verification of CPSs has the goal of verifying that the *whole* (i.e., software + hardware) system meets the given specifications.

Model checkers for hybrid systems cannot handle System Level Formal Verification (SLFV) of actual CPSs. Thus, Hardware In the Loop Simulation (HILS) is currently the main workhorse for system level verification and is supported by *Model Based Design* tools like Simulink (http://www.mathworks.com) and VisSim (http://www.vissim.com). In HILS, the *actual software* reads/sends values from/to mathematical models (*simulation*) of the physical systems (e.g., engines, analog circuits, etc.) it will be interacting with.

A. Motivations

System Level Formal Verification (SLFV) is an *exhaustive* HILS, where *all* relevant simulation scenarios are considered. In [2], [3], [4] a methodology has been presented which allows exhaustive HILS. Such methodology works as follows.

The System Under Verification (SUV) is a *Hybrid System* (see, e.g., [1] and citations thereof) whose inputs belong to a finite set of uncontrollable events (*disturbances*), which model failures in sensors or actuators, variations in the system parameters, etc. The SUV is a *deterministic system* (the typical case for control systems). Nondeterministic behaviours (such as faults) are modelled with disturbances. Also, sequences of inputs to the SUV are of *bounded* length, thus the problem addressed is *bounded* SLFV. Accordingly, in [2], [3], [4], a *simulation scenario* is a finite sequence of disturbances. A system 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 [2], [3], [4]: (i) generates the entire set of such sequences, (ii) evenly splits such set into $k \in \mathbb{N}^+$ *slices* in order to allow *parallel* verification, (iii) computes (in parallel) an optimised *simulation campaign* from each slice, (iv) executes (in parallel) the generated simulation campaigns on a set of k independent *simulators* (e.g. Simulink instances).

There, a *simulation campaign* is a sequence of simulation *instructions*, which exploits the capabilities of moderns 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 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 give a quite accurate estimation of the *completion time*.

Algorithms for all the activities above have been presented in [2], [3], [4]. However, an off-the-shelf tool to *effectively* support companies working in the CPS business in their everyday SUV verification activities was not available. To provide such a tool is exactly the purpose of this paper.

B. Main Contributions

We present SyLVaaS (see Fig. 1), a Web-based service computing the set of simulation campaigns to be used for a

This paper appears in the proceedings of the 23rd Euromicro International Conference on Parallel, Distributed and Network-based processing (PDP 2015).

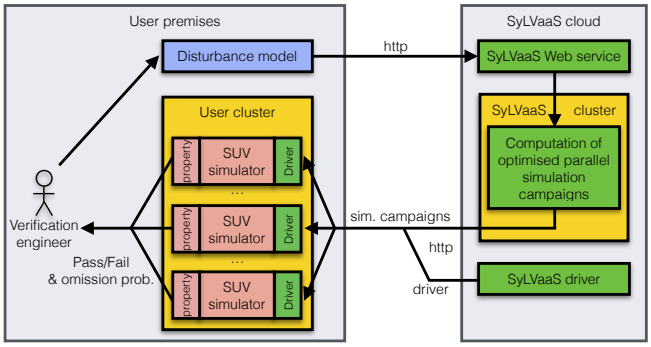


Fig. 1: SyLVaaS VaaS architecture

SLFV task.

SyLVaaS introduces the new Verification as a Service (VaaS) paradigm, allowing verification engineers (SyLVaaS *users*) to compute the simulation campaigns needed to their SLFV activities keeping both the SUV model and the property to be verified *secret*, thus achieving full Intellectual Property (IP) protection. This is mandatory for a VaaS service to be effective and usable, as companies consider the design effort (hence their SUV models) and their verification flow as the core of their IP.

To enable IP protection, SyLVaaS takes as input only a *disturbance model*, in terms of a CMurphi [5] model describing the admissible operational scenarios the SUV must withstand. 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.

In case an error is found during verification, a counterexample is generated. Such a counterexample can then be used to correct the SUV and to produce a new *SUV* model. At this point a new SLFV activity can start. Note that, 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. This property also hides the verification flow of SyLVaaS users.

The operational scenario generation algorithm in [2] is a sequential algorithm, taking about half an hour on their case study. Although this time is negligible with respect to the whole HILS activity (which can take weeks of computation), it becomes a major bottleneck in a VaaS context as the one provided by SyLVaaS, as it 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 Fig. 1, right).

In this paper, to achieve fast response time in SyLVaaS, 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 as 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.

We experiment our new algorithm on two industry-scale case studies (regarding the Fuel Control System (FCS) in the Simulink distribution) consisting of, respectively, 4,023,955 and 12,948,712 operational scenarios. 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 computation within the SyLVaaS workflow, and given that the other step performed by SyLVaaS (computation of optimised simulation campaigns) already exploits an embarrassingly parallel algorithm (from [3]), with our new parallel disturbance trace generator the entire SyLVaaS workflow can benefit of a cluster of machines at the SyLVaaS cloud infrastructure.

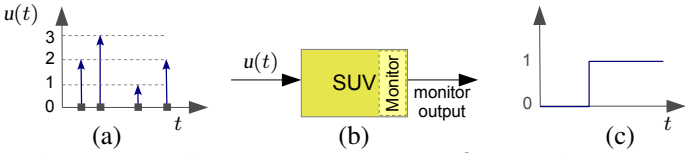
C. Related Work

The papers closest to ours are [2], [3], [4], where the algorithms underlying SyLVaaS workflow are presented. [2], [3], [4] split the computation of the set of system runs to verify from the actual execution (by simulation), presenting a parallel approach to formal verification of the full class of hybrid systems handled by Simulink. 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.

Parallel algorithms for parallel model checking have been widely investigated. For example: OBDD (*Ordered Binary Decision Diagrams*) based (symbolic) model checking algorithms are described in [6], [7] and citations thereof; parallel SAT based algorithms are described in [8] and citations thereof. Explicit algorithms have been investigated in [9], [10]. Unlike our approach, none of the above approaches can handle the full class of hybrid systems handled by Simulink, as we do.

System Verification as a Service (also known as Model Checking in the Cloud) is still in its infancy. In [11], it is argued that ideas may be borrowed from workflow modelling, management and analysis of business process. In [12] 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 VaaS tool are ongoing in major conferences [13]. However, none of such works propose an implemented and available tool, with the features described in Section I-B. We also point out that "verification as a service" is sometimes used with a meaning different from the one we use here, i.e., system verification via Model Checking. Namely, there are companies offering verification as a service, which however is a consulting service. Moreover, verification services are available in many different fields, such as, e.g., identity verification services. Thus, to the best of our knowledge, SyLVaaS is the first verification as a service available.

The methodology and the type of verification problems addressed here are closely related to: HILS-based SLFV (e.g., see [14]), statistical model checking (e.g., see [15], [16], [17]), Monte Carlo model checking methods (e.g., see [18], [19], [20]), formal verification of Simulink models (e.g., see [21], [22]), synergies between simulation and formal methods (e.g., see [23], [24]), parallel algorithms for explicit state exploration (e.g., see [25], [9], [26], [27]), and hybridisation of explicit and symbolic methods for automatic *synthesis* of controllers for discrete time linear hybrid systems [28], [29], [30], [31], [32], [33], [34], [35]. We refer the reader to [2], [3], [4] for a comparison with such works.



(a) (b) (c)

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

II. Background

In this section we give some background notions. Unless otherwise stated, all definitions are based on [36], [2], [3], [4] 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 $\text{Bool} = \{0, 1\}$ the set of Boolean values (where 0 means 'false' and 1 means 'true').

A. 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 [36] whose inputs are *discrete event sequences*. A discrete event sequence is a function $u(t)$ associating to each (continuous) time instant $t \in \mathbb{R}^{+}$ a *disturbance event* (or, simply, *disturbance*). Disturbances, encoded by integers in the interval $[0, d]$ (for a given $d \in \mathbb{N}^{+}$), represent uncontrollable events (e.g., faults). We use event 0 to represent the event carrying no disturbance. As no system can withstand an infinite number of disturbances within a finite time, we require that, in any time interval of finite length, a discrete event sequence $u(t)$ differs from 0 only in a finite number of time points (Fig. 2a).

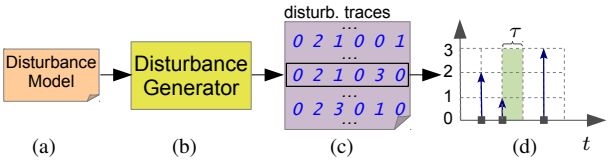
B. Modelling the Property to be Verified

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

C. 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) 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 1 and Fig. 3(a–c)).



(a) (b) (c) (d)

Fig. 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 τ.

Definition 1 (Disturbance generator): A 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;
- $\mathit{dist} : Z \times [0, d] \rightarrow Z$ is a (*deterministic transition*) function mapping each state/disturbance pair (z, e) to a *next* state $\mathit{dist}(z, e)$;
- $\mathit{adm} : Z \times [0, d] \rightarrow \text{Bool}$ 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 defines disturbance traces (simulation scenarios) as paths from initial to final states in a DG.

Definition 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 $z_{i+1} = \mathit{dist}(z_i, d_i)$.

(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 δ ($0 \leq j < h$). □

Given $\tau \in \mathbb{R}^{+}$ (*time quantum*), to a disturbance trace δ for $\mathcal{D}$ we can univocally associate a discrete event sequence u^{τ}_{δ}, defined as follows: for all $t \in \mathbb{R}^{\geq 0}$, if there exists $j \in [0, h-1]$ such that $t = \tau j$ then $u^{\tau}_{\delta}(t) = \delta(j)$, else $u^{\tau}_{\delta}(t) = 0$ (no disturbance).

Thus a disturbance trace δ defines an operational scenario (namely, u^{τ}_{δ}) for our SUV. 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 τ *small enough* (to faithfully model our SUV operational scenarios), we can achieve any desired precision. On such considerations rests the effectiveness of the approach.

D. System Level Formal Verification

Definition 3 formalises our bounded System Level Formal Verification problem.

Definition 3: A *System Level Formal Verification (SLFV) problem* is a tuple $(\mathcal{H}, \mathcal{D}, \tau, h)$ where: $\mathcal{H}$ is a DES with an embedded monitor modelling our SUV, $\mathcal{D}$ is a DG modelling a set of disturbance traces Δ 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 δ in Δ such that the SUV monitor output at time τh is 1, when $\mathcal{H}$ is given u_δ^τ (the discrete event sequence associated to δ given time quantum τ) 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 τ as the time quantum among disturbance events.

E. Parallel HILS based anytime random exhaustive SLFV

Along the lines of [2], [3], [4], 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 δ in the operational environment Δ, thus performing an *exhaustive* (with respect to Δ) 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 (along the lines of [2]) the sequence of disturbance traces Δ into $k \in \mathbb{N}^+$ sequences of disturbance traces Δ_0, $\ldots$, Δ_{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 δ 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 needed since each state may require many KB of memory (150–300 KB in the case study 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 [3]), 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*).

```
const  h : 7;  A : 1;  B : 2;
var -- global variables
  t : 1 .. h + 1; -- time
  d : array [A .. B] of 0 .. h;
      -- disturbance times
startstate begin t := 1;
  d[A] := 0; d[B] := 0; end;
rule "ok" t <= h ==> t := t + 1;
rule "A fails" t <= h & d[A] = 0 &
  (d[B] = 0 | (t-d[B] > 2)) ==>
  begin d[A] := t; t := t + 1; end;
-- dual rule for B omitted
finalstate (t = h + 1);
```

Fig. 4: CMurphi code for the DG described in Example 1

III. 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.

A. 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, along the lines of [2], takes as input a *disturbance model* defining a Disturbance Generator (DG) written in the high-level language accepted by the CMurphi [5] model checker. The following example clarifies this point.

Example 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 scenario is shown in Fig. 4, where the verification time horizon is 7 seconds. □

B. 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.

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 [3] 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 which 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 Fig. 1 and Section III-D).

C. Web Interface

The Web interface of SyLVaaS is hosted at http://mclab.di.uniroma1.it/sylvaas. It consists of four main pages: (i) a standard login page, (ii) a user console page (accessible after login) showing all current, pending, running and completed user jobs, (iii) a page to create a new job (providing the required input) (iv) a tools page, where the generic driver can be downloaded. Users can download the simulation campaigns for each completed job from their console page.

D. 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 [3]: $1 - \min_{i \in [1,k]} (done_i)$.

IV. Parallel Generation of Disturbance Traces

As reported in [2], the most computationally intensive step of the workflow for the generation of simulation campaigns is disturbance trace generation starting from the user disturbance model. This task is performed in [2] using a modified version of the CMurphi model checker. As reported there, on a disturbance model entailing 4 million traces, 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 [4]. 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 by far the *most time-dominant step* in the SyLVaaS workflow.

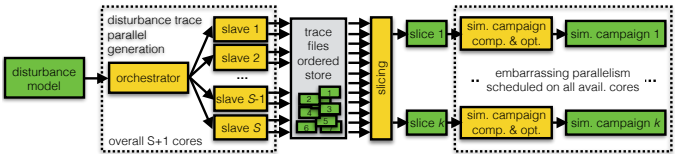


Fig. 5: SyLVaaS workflow and deployment

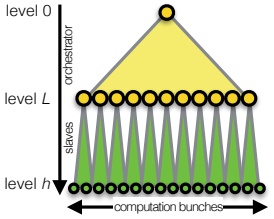


Fig. 6: Parallel trace generation

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., trace generation and optimised simulation campaigns computation) can now take benefit from the availability of a cluster in the SyLVaaS cloud infrastructure (see Fig. 5).

A. Algorithm overview

Our new parallel algorithm for trace generation has been explicitly designed as 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 Fig. 6. 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 gives as output a sequence of traces Δ_b (stored in a file), containing a subset of the disturbance traces entailed by the model. The sets Δ_b of traces produced by all computation bunches b form a *partition* of the entire set of admissible disturbance traces Δ.

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. Also, computation bunches delegated to the Slaves are short, hence *load balancing* can be performed by the Orchestrator simply by assigning more computation bunches to the faster Slaves.

B. Distributed trace labelling

Both the Orchestrator and the Slaves work in DFS mode, and hence each computation bunch b produces a sequence Δ_b of disturbance traces in *lexicographic order*. Each disturbance trace *prefix* identifies a simulator state. To allow generation and optimisation of simulation campaigns, we associate a unique label to each such prefixes (Definition 4).

Definition 4 (Labelling of disturbance traces): Let $\mathcal{D}$ be a DG defining $d \in \mathbb{N}^+$ disturbances, and Λ be a countably infinite set of labels. A labelling function over $[0, d]$ is an injective map λ from finite sequences of values in $[0, d]$ (including the empty sequence) to labels in Λ.

Let $\delta = d_0, \ldots, d_{h-1}$ be a disturbance trace for $\mathcal{D}$. The labelling of δ (according to λ) 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, 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 common disturbance prefixes 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 the optimiser in [3]). 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 λ 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 the 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 these sets of labels are *disjoint*, hence resulting in a overall injective map.

C. 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) $L < h$, also assigning unique labels (see variable λ) 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 δ^λ (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 (δ) are generated in *lexicographic order*.

D. 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, δ^λ), where $z_0 \in Z_I$ is one of the initial states of $\mathcal{D}$, b is the computation bunch id, and δ^λ is a labelled prefix of disturbance traces (containing L disturbances), as computed by the Orchestrator.

Upon reception of (z_0, b, δ^λ), a slave $s \in S$: (i) reaches the root of the subtree which is in charge to explore by following δ^λ, (ii) starts its own DFS from there, hence limiting its attention to that subtree.

Admissible (complete) disturbance traces found (which have δ^λ as a prefix) are appended to the output file Δ_s of slave s and annotated with the id b of the current computation bunch. During exploration, each slave also carries out trace labelling using its own (disjoint) set of labels (see variable λ).

```
1  function Orchestrator(D, h, L, S)
       Input: D = (Z, d, dist, 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
2      λ ← 1; // next label to be used
3      b ← 1; // id of the next comp. bunch
4      let δ^λ be an array of variables l_0, d_0, l_1, d_1, ..., l_L;
5      foreach z_0 ∈ Z_I do
6          stack ← {(z_0, 0)}; l_0 ← λ; λ ← λ + S + 1;
7          while stack is not empty do
8              (z, d̂) ← top(stack);
9              if d̂ ≤ d then
10                 top(stack) ← (z, d̂ + 1); j ← |stack| − 1;
11                 if adm(z, d̂) then
12                     d_j ← d̂; l_{j+1} ← λ; λ ← λ + S + 1;
13                     if j < L then push(stack, (dist(z, d̂), 0));
14                     else
                           // delegate computation bunch
15                         wait for an idle slave s;
16                         send (z_0, b, δ^λ) to slave s;
17                         b++;
18             else pop(stack);
19     wait until all slaves become idle
```

Algorithm 1: Orchestrator

E. Algorithm Correctness

Theorem 1 shows correctness of our algorithm (proof omitted for lack of space):

Theorem 1: Let $\mathcal{D}$ be a DG, $h, S \in \mathbb{N}^+$, and λ be a labelling function according to Definition 4. Let Δ be the entire set of disturbance traces for $\mathcal{D}$ with horizon h, and let Δ_s be the (ordered) sequence of disturbance traces generated by slave process $s \in S$ of our algorithm. The following holds:

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

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

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

Theorem 1 shows that, from $\Delta_1, \ldots, \Delta_S$, we can easily produce $k \in \mathbb{N}^+$ lexicographically ordered slices $(slice_1, \ldots, slice_k)$ 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 [3].

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. This enables Omission Probability (OP) computation at anytime during the simulation activity at the user premises (see Section II) as well as completion time estimation. As already shown in [3], the generation of the k simulation campaigns can be scheduled on all the cores available to SyLVaaS in an *embarrassingly parallel* fashion.

```
1  function Slave(D, h, L, S, s)
     Input: D = (Z, d, dist, 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 ∈ [1, S], id of this slave
2    λ ← s + 1; // next label to be used
3    while true do
       // slave is idle
4      wait for a message (z_0, b, δ^λ) from Orchestrator;
       // δ^λ = l_0, d_0, l_1, d_1, ..., l_L
5      let δ̃^λ be an array of variables l̃_0, d̃_0, l̃_1, d̃_1, ..., l̃_h;
       // start computation bunch b
6      stack ← empty stack; Δ_b ← empty sequence;
7      z ← z_0;
       // follow δ^λ to reach root of req.
          subtree & copy it into δ̃^λ
8      for j ← 0 to L − 1 do
9        l̃_j ← l_j; d̃_j ← d_j; z ← dist(z, d̃_j);
10     l̃_L ← l_L;
       // start DFS from there
11     push(stack, (z, 0));
12     while stack is not empty do
13       (z, d̂) ← top(stack);
14       if d̂ ≤ d then
15         top(stack) ← (z, d̂ + 1); j ← L − 1 + |stack|;
16         if adm(z, d̂) then
17           d̃_j ← d̂; l̃_{j+1} ← λ; λ ← λ + S + 1;
18           if j < h then push(stack, (dist(z, d̂), 0));
19           else if z ∈ Z_F then append (b, δ̃^λ) to Δ_s;
20       else pop(stack);
```

Algorithm 2: Slave

V. Experiments

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

A. SyLVaaS Experimental Deployment

We deployed SyLVaaS on overall 17 computational cores allocated to 5 different machines. One core (on a machine equipped with 2 Intel Xeon 2.83GHz CPUs and 8GB RAM) is dedicated to Orchestrator processes, while 16 cores evenly distributed in 4 identical machines (each one equipped with 2 Intel Xeon 2.27GHz CPUs and 24GB RAM) are dedicated to Slave processes. 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.

B. Case Study

We use the same case study of [15], [2], [3], [4], i.e., the Fuel Control System (FCS) model included in the Simulink distribution. The FCS has three sensors subject to faults (disturbances).

We used two disturbance models for the FCS, $\mathcal{D}_1$ and $\mathcal{D}_2$. Model $\mathcal{D}_1$ (described in more detail in [2]) has a horizon of $h = 100$ and defines 4,023,955 disturbance traces. Model $\mathcal{D}_2$ is defined extending $\mathcal{D}_1$ with more complex operational scenarios and defines 12,948,712 disturbance traces over a horizon of $h = 200$. A detailed description of $\mathcal{D}_1$ and $\mathcal{D}_2$ (not relevant for the evaluation of our experiments below) can be downloaded from the SyLVaaS Web site.

#slaves (S)	disturbance model $\mathcal{D}_1$			disturbance model $\mathcal{D}_2$		
	time (h:m:s)	speedup	efficiency	time (h:m:s)	speedup	efficiency
1	0:32:32	1.00×	100.00%	4:45:47	1.00×	100.00%
8	0:5:32	5.88×	73.50%	0:43:2	6.64×	83.00%
16	0:3:11	10.22×	63.88%	0:26:16	10.88×	68.00%

TABLE I: Results on parallel generation of disturbance traces

#slices (k)	$\mathcal{D}_1$ (h:m:s)	$\mathcal{D}_2$ (h:m:s)
128	0:4:1	0:8:7
256	0:4:32	0:11:25
512	0:4:52	0:13:17

TABLE II: Results on slicing of disturbance traces

C. Experimental Results

1) Parallel Disturbance Trace Generation: Table I shows the time needed by SyLVaaS to generate the disturbance traces entailed by $\mathcal{D}_1$ (4,023,955 traces) and $\mathcal{D}_2$ (12,948,712 traces), when using a *varying* number S of parallel slaves. The level (depth) L to which the Orchestrator bounds its search and triggers a Slave has been fixed at $h/2$ after preliminary experiments. This value seems to be quite stable among various models. Intuitively, a lower value for L typically makes computation bunches (performed by Slaves) too long, hindering load-balancing. A greater value for L typically makes the (single) Orchestrator carry out too much work. In both cases the overall generation time is longer. The number of computation bunches executed by the algorithm is 477,727 for disturbance model $\mathcal{D}_1$ and 1,681,594 for $\mathcal{D}_2$.

For each value for S, Table I reports the overall time for generating the whole set of disturbance traces for both disturbance models (columns "*time*"), as well as *speedup* and *efficiency* with respect to the execution time of the sequential algorithm (the first row in Table I 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.

2) Disturbance Trace Slicing: Table II shows the time needed by SyLVaaS to compute k slices from the disturbance traces generated using $S = 16$ slaves from disturbance models $\mathcal{D}_1$ and $\mathcal{D}_2$, for various values of k, which denotes the number of computational cores available at the user side for parallel simulation. To ease comparison of our results with those in [3], we used the same values of k as those used in that paper.

3) SyLVaaS Complete Workflow: Table III reports the time needed to compute (in parallel) the k simulation campaigns (column "*sim. campaign comp. time*") and the overall SyLVaaS response time (summing up trace generation, splitting, and simulation campaign optimisation times, column "*overall time*"), for each disturbance model ($\mathcal{D}_1$ or $\mathcal{D}_2$) and each value for k. Results in Table III have been obtained using $S = 16$ slaves during trace generation and 16 cores to compute the k simulation campaigns (thus, on average, each core computed $k/16$ campaigns).

4) 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 in the order of *a few hundreds of MB*. Hence, their download into the user cluster can be done seamlessly over a standard broad-band Internet connection.

	disturbance model $\mathcal{D}_1$		disturbance model $\mathcal{D}_2$	
#slices (k)	**sim. campaign comp. time (h:m:s)**	**overall time (h:m:s)**	**sim. campaign comp. time (h:m:s)**	**overall time (h:m:s)**
128	0:1:44	0:8:56	0:4:1	0:38:24
256	0:0:42	0:8:25	0:2:27	0:40:8
512	0:0:13	0:8:16	0:0:24	0:39:57

TABLE III: Results on the entire SyLVaaS workflow

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

[2] T. Mancini, F. Mari, A. Massini, I. Melatti, F. Merli, and E. Tronci, "System level formal verification via model checking driven simulation," in *Proc. CAV 2013*, ser. LNCS, vol. 8044. Springer, 2013, pp. 296–312.

[3] T. Mancini, F. Mari, A. Massini, I. Melatti, and E. Tronci, "Anytime system level verification via random exhaustive hardware in the loop simulation," in *Proc. DSD 2014*. IEEE, 2014.

[4] ——, "System level formal verification via distributed multi-core hardware in the loop simulation," in *Proc. PDP 2014*. IEEE, 2014.

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

[6] S. Ben-David, O. Grumberg, T. Heyman, and A. Schuster, "Scalable distributed on-the-fly symbolic model checking." *STTT*, vol. 4, no. 4, pp. 496–504, 2003.

[7] D. Kunkle, V. Slavici, and G. Cooperman, "Parallel disk-based computation for large, monolithic binary decision diagrams," in *Proc. PASCO 2010*. ACM, 2010, pp. 63–72.

[8] E. Ábrahám, T. Schubert, B. Becker, M. Fränzle, and C. Herde, "Parallel sat solving in bounded model checking," *Journal of Language and Computation*, vol. 21, no. 1, pp. 5–21, 2011.

[9] I. Melatti, R. Palmer, G. Sawaya, Y. Yang, R. M. Kirby, and G. Gopalakrishnan, "Parallel and distributed model checking in Eddy." *STTT*, vol. 11, no. 1, pp. 13–25, 2009.

[10] J. Barnat, L. Brim, M. Ceska, and P. Rockai, "Divine: Parallel distributed model checker," in *Proc. 9th Intl. Workshop on Parallel and Distributed Methods in Verification and 2nd Intl. Workshop on High Performance Computational Systems Biology*, 2010, pp. 4–7.

[11] I. Schaefer and T. Sauer, "Towards verification as a service," in *Eternal Systems*, ser. Communications in Computer and Information Science. Springer, 2012, vol. 255, pp. 16–24.

[12] C. Bellettini, M. Camilli, L. Capra, and M. Monga, "Distributed CTL model checking in the cloud," *CoRR*, vol. abs/1310.6670, 2013.

[13] G. Cabodi and S. Singh, Eds., *Proc. FMCAD 2012*. IEEE, 2012.

[14] G. Verzino, F. Cavaliere, F. Mari, I. Melatti, G. Minei, I. Salvo, Y. Yushtein, and E. Tronci, "Model checking driven simulation of sat procedures," in *Proc. SpaceOps 2012*, 2012.

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

[16] E. Tronci, T. Mancini, I. Salvo, S. Sinisi, F. Mari, I. Melatti, A. Massini, F. Davì, T. Dierkes, R. Ehrig, S. Röblitz, B. Leeners, T. H. C. Krüger, M. Egli, and F. Ille, "Patient-specific models from inter-patient biological models and clinical records," in *Proc. FMCAD 2014*. FMCAD Inc., 2014, pp. 207–214.

[17] T. Mancini, F. Mari, I. Melatti, I. Salvo, E. Tronci, J. Gruber, B. Hayes, M. Prodanovic, and L. Elmegaard, "Demand-aware price policy synthesis and verification services for smart grids," in *Proc. SmartGridComm 2014*. IEEE, 2014.

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

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

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

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

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

[23] K. Nanshi and F. Somenzi, "Guiding simulation with increasingly refined abstract traces," in *Proc. DAC 2006*. ACM, 2006, pp. 737–742.

[24] F. De Paula and A. Hu, "An effective guidance strategy for abstraction-guided simulation," in *Proc. DAC 2007*. ACM, 2007, pp. 63–68.

[25] J. Barnat, L. Brim, I. Černá, P. Moravec, P. Ročkai, and P. Šimeček, "Divine: a tool for distributed verification," in *Proc. CAV 2006*. Springer, 2006, pp. 278–281.

[26] G. Holzmann, "Parallelizing the SPIN model checker," in *Proc. SPIN 2012*. Springer, 2012, pp. 155–171.

[27] A. Laarman, J. van de Pol, and M. Weber, "Boosting multi-core reachability performance with shared hash tables," in *Proc. FMCAD 2010*, IEEE, 2010, pp. 247–255.

[28] F. Mari, I. Melatti, I. Salvo, and E. Tronci, "Model based synthesis of control software from system level formal specifications," *ACM Trans. Softw. Eng. and Method.*, vol. 23, no. 1, 2014.

[29] ——, *Theoretical Aspects of Computing – ICTAC 2012*, ser. LNCS. vol. 7521. Springer, 2012, pp. 243–258.

[30] V. Alimguzhin, F. Mari, I. Melatti, I. Salvo, and E. Tronci, "On model based synthesis of embedded control software," in *Proc. EMSOFT 2012*, ACM, 2012, pp. 227–236.

[31] ——, "Automatic control software synthesis for quantized discrete time hybrid systems," in *Proc. CDC 2012*. IEEE, 2012, pp. 6120–6125.

[32] ——, "On-the-fly control software synthesis," in *Proc. SPIN 2013*, ser. LNCS, vol. 7976. Springer, 2013, pp. 61–80.

[33] ——, "A map-reduce parallel approach to automatic synthesis of control software," in *Proc. SPIN 2013*, ser. LNCS, vol. 7976. Springer, 2013, pp. 43–60.

[34] F. Mari, I. Melatti, I. Salvo, and E. Tronci, "Synthesis of quantized feedback control software for discrete time linear hybrid systems," in *Proc. CAV 2010*, ser. LNCS, vol. 6174. Springer, 2010, pp. 180–195.

[35] G. Della Penna, B. Intrigila, E. Tronci, and M. Venturini Zilli, "Synchronized regular expressions," *Acta Inf.*, vol. 39, no. 1, pp. 31–70, 2003.

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