System Level Formal Verification via Distributed Multi-Core Hardware in the Loop Simulation

Toni Mancini, Federico Mari, Annalisa Massini, Igor Melatti, Enrico Tronci
Computer Science Department, Sapienza University of Rome
Via Salaria 113, I-00198 Roma, Italy
Email: {tmancini,mari,massini,melatti,tronci}@di.uniroma1.it

***Abstract*—The goal of System Level Formal Verification (SLFV) is to show system correctness notwithstanding *uncontrollable events* (such as: faults, variation in system parameters, external inputs, etc). Hardware In the Loop Simulation (HILS) based SLFV attains such a goal by considering exhaustively *all* relevant simulation scenarios.**

We present a distributed multi-core algorithm for HILS-based SLFV. Our experimental results on the Fuel Control System example in the Simulink distribution show that by using 64 machines with an 8 core processor each we can complete the SLFV activity in about 27 hours whereas a sequential approach would require more than 200 days.

To the best of our knowledge this is the first time that a distributed multi-core algorithm for HILS-based SLFV is presented.

***Keywords*-Model Checking; Hybrid Systems; System Level Formal Verification; Distributed Multi-Core Hardware in the Loops Simulation;**

I. Introduction

System Level Verification 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 systems. 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.

SLFV basically is an exhaustive HILS where *all* relevant simulation scenarios are considered. Unless the number of such scenarios is small, exhaustive HILS is infeasible.

The situation can be considerably improved by evenly splitting the simulation scenarios into disjoint *slices* and then effectively distributing the simulation of such slices on different machines. This has been done in [1].

Unfortunately the approach in [1] cannot exploit the availability of multi-core processors. In this paper we advance the state of the art by presenting a distributed multi-core approach to HILS-based SLFV.

A. Main Contribution

Our System Under Verification (SUV) is a *Hybrid System* (see, e.g., [2] and citations thereof) whose inputs belong to a finite set of uncontrollable events (*disturbances*) modelling failures in sensors or actuators, variations in the system parameters, etc. We focus on *deterministic systems* (the typical case for control systems), and model nondeterministic behaviours (such as faults) with disturbances. Accordingly, in our framework, a *simulation scenario* is just a finite sequence of disturbances and a *simulation campaign* is a sequence of simulation instructions (namely: *save* a simulation state, *restore* a saved simulation state, *inject* a disturbance, *advance* the simulation of a given time length).

A system is expected to *withstand* all disturbance sequences that may arise in its operational environment. Correctness of a system is thus defined with respect to such *admissible* disturbance sequences.

In such a framework (as in [1]) we address *Bounded* SLFV of *safety* properties. That is, given a time step τ (time quantum between disturbances) and a time horizon $h\tau$ (i.e., h multiplied by τ) we return *PASS* if there is no *admissible* disturbance sequence of length h and time step τ that violates the given safety property. We return *FAIL*, along with a counterexample, otherwise. Therefore, SLFV is an *exhaustive* (with respect to admissible simulation scenarios) HILS. In other words, we are aiming at (*black box*) *bounded model checking* where the SUV behaviour is defined by a simulator (Simulink in our examples).

To enable an effective distributed approach to SLFV, we split the verification process into two main phases. First, an *off-line* phase, where we: (a) generate the simulation scenarios to be considered and evenly split them into disjoint *slices*, by using the approach in [1]; (b) generate, for each slice, a highly optimised simulation campaign for the given simulator. Second, an *on-line* distributed phase where each simulator runs its simulation campaign independently and stops as soon as an error is found. The rationale is that the simulation phase is the heavier one from a computational point of view, thus our approach aims at parallelising such a phase.

The *on-line* phase is supported by simulation tools (Simulink in our examples). Here we provide methods and tools to effectively carry out step b of the above *off-line* phase. Our main contributions can be summarised as follows.

Distributed multi-core SLFV: We present an *optimisation* algorithm that transforms a sequence of simulation scenarios into a very efficient *simulation campaign* that avoids revisiting already visited states by using simulator save/restore commands. Our optimisation algorithm generates a very efficient simulation campaign that during the simulation stores at most h states (if $h\tau$ is the time horizon). This allows us to store such states using a small amount of RAM (about 15MB in our case study where $h = 100$ and each simulator state takes about 150KB). Such a small RAM footprint allows running in parallel a simulation campaign for each of the available cores. Note that an efficient simulation campaign generated using the optimiser in [1] may need to store many states. For this reason simulation campaigns in [1] store states on the local disk. As a result, the algorithm in [1] cannot exploit availability of multi-core processors, since the the local disk becomes the main bottleneck if more than one simulation campaign tries to use it. Devising a strategy for storing/restoring simulation states that uses a moderate amount of memory and can thus be implemented in RAM rather then on disk is indeed the main obstacle we had to overcome to achieve multi-core parallelism.

Experimental Results: We implemented our approach and present experimental results on its usage in the Fuel Control System (FCS) example in the Simulink distribution. In our experiments we set our *time step* τ to 1 second and our *time horizon* $h\tau$ to 100 seconds (i.e., $h = 100$). SLFV for this case study entails running more than 4 million simulation scenarios.

Each core runs an instance of our optimisation algorithm taking as input a different slice of the simulation scenarios. Our optimiser takes just a few minutes to generate a simulation campaign from a given slice. In our setting each machine has a single processor with $c = 8$ cores. We present experimental results with $k = 8, 16, 32, 64$ machines totalling $kc = 64, 128, 256, 512$ cores.

Our experimental results show that, with respect to the (distributed single-core) approach in [1], our distributed multi-core approach saves more than 65% of the computation time when the same hardware is used. For example, when using 64 machines we can complete *SLFV* for our case study in about 27 hours whereas using the approach in [1] requires about 81 hours. Note that a purely sequential approach would require more than 200 days.

Summing up: We present an effective distributed multi-core approach to HILS-based SLFV. To the best of our knowledge this is the first time that such an approach is presented.

B. Related Work

The paper closest to ours is [1], where a HILS-based distributed algorithm for SLFV has been presented. We note however that the algorithm of [1] cannot exploit availability of multi-core processors since the simulation campaigns generated by such an optimiser heavily relies on the local disk that becomes the main bottleneck if more than one simulation campaign tries to use it. The present paper advances the state of the art by presenting a novel optimisation algorithm that generates simulation campaigns requiring a small amount of RAM. This, in turn, enables use of multi-core parallelism.

HILS-based SLFV has also been investigated in [3] where the CMurphi [4] capability to call external C functions in a *black box* fashion has been used to drive the ESA satellite simulator SIMSAT in order to verify satellite operational procedures.

Statistical model checking, being basically *black box*, is also closely related to our approach. In such a setting, [5] is closely related to our paper since it addresses system level verification of Simulink models and presents experimental results on the very same Simulink case study we are using. Monte Carlo model checking methods (see, e.g., [6], [7], [8]) are also related to our approach. The main differences between the above statistical approaches and ours are the following: (i) statistical methods *sample* the space of admissible simulation scenarios, whereas we address *exhaustive* HILS; (ii) statistical methods do not address optimisation of the simulation campaign which is our main concern here, since this is what makes exhaustive HILS viable.

Formal verification of Simulink models has been widely investigated, examples are in [9], [10], [11]. Such methods however focus on discrete time models (e.g., Stateflow or Simulink restricted to discrete time operators) with small domain variables. Therefore they are well suited to analyse critical subsystems, but cannot handle complex system level verification tasks (e.g., as our case study). This is indeed the motivation for the development of statistical model checking methods as the one in [5] and for our exhaustive HILS-based approach.

Synergies between simulation and formal methods have been widely investigated in digital hardware verification. Examples are in [12], [13], [14], [15] 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*. We note that the idea of speeding up the simulation process by saving and restoring suitably selected visited states is also present in [15].

Parallel algorithms for explicit state exploration have been widely investigated. Examples are in [16], [17], [18], [19], [20], [21]. The main difference with our approach is that all

the above ones focus on parallelising the state space exploration engine by devising techniques to minimise locking of the visited state hash table whereas we leave unchanged the state space exploration engine (the simulator in our context) and use a *Map-Reduce like* strategy that splits (*Map* step) the set of simulation scenarios into equal sized subsets to be simulated on different cores and stops verification as soon as one of such cores finds an error (*Reduce* step). Note that we propose an *embarrassingly parallel* algorithm for (black box) formal verification of hybrid systems. Embarrassingly parallel verification algorithms have also been investigated in [22], as for finite state system verification, and in [23], as for symbolic testing of programs. Such approaches are close in spirit to ours, although they differ from ours as for the class of systems considered (we focus on hybrid systems whereas the above papers focus on discrete systems) as well as for the modelling approach (our black box algorithm rests on the disturbance model whereas the above papers both present white box algorithms resting on the system model).

II. Background

In this section we give some background notions. Unless otherwise stated, all definitions are based on [24], [1]. Throughout the paper, we use $\mathbb{R}^{\geq 0}$ for the set of non-negative reals, $\mathbb{R}^{+}$ for the set of strictly positive reals, and $\text{Bool} = \{0, 1\}$ for the set of Boolean values (0 for *false* and *1* for true). $\mathbb{N}^{+}$ denotes the set of positive natural numbers.

A. Modelling uncontrollable events

A *discrete event sequence* (Definition 1 and Fig. 1a) is a function associating to each (continuous) time instant a *disturbance event* (or, simply, *disturbance*). Disturbances, encoded by integers in the interval $[0, d]$ (for a given $d \in \mathbb{N}^{+}$), represent exogenous 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 differs from 0 only in a finite number of time points.

Definition 1 (Discrete event sequence): Let $d \in \mathbb{N}^{+}$. A *discrete event sequence* over integer interval $[0, d]$ is a function $u : \mathbb{R}^{\geq 0} \rightarrow [0, d]$ such that, for all $t \in \mathbb{R}^{\geq 0}$, the set $\left\{\tilde{t} \mid 0 \leq \tilde{t} \leq t \text{ and } u(\tilde{t}) \neq 0\right\}$ has finite cardinality. We denote with $\mathfrak{U}_d$ the set of discrete event sequences over $[0, d]$.

B. Modelling the System Under Verification

We model (Definition 2) our System Under Verification (SUV) as a continuous time Input-State-Output deterministic dynamical system whose inputs are discrete event sequences.

Definition 2 (Discrete Event System): A Discrete Event System (DES) is a tuple $\mathcal{H} = (S, s_0, d, O, \textit{flow}, \textit{jump}, \textit{output})$ where:

- S is a set of *states* (finite, countable, continuous, or any combination thereof).
- $s_0 \in S$ is the *initial state*.
- $d \in \mathbb{N}^{+}$ defines the *input space* as $\mathfrak{U}_d$ (the set of discrete event sequences over $[0, d]$).
- O is the set of *output* values (finite, countable, continuous, or any combination thereof).
- $\textit{flow} : S \times \mathbb{R}^{\geq 0} \rightarrow S$. For all $s \in S$, $t \in \mathbb{R}^{\geq 0}$, $\textit{flow}(s, t)$ defines the state reached by $\mathcal{H}$ from state s after time t when no event occurs. Accordingly, we stipulate that for all $s \in S$, $\textit{flow}(s, 0) = s$.
- $\textit{jump} : S \times [0, d] \rightarrow S$. For all $s \in S$, $e \in [0, d]$ $\textit{jump}(s, e)$ defines the state reached by $\mathcal{H}$ from state s upon occurrence of event e (no time flows). Accordingly, we stipulate that for all $s \in S$, $\textit{jump}(s, 0) = s$.
- $\textit{output} : S \rightarrow O$. The value $\textit{output}(s)$ defines the output of $\mathcal{H}$ in state s.

The state, respectively output, reached after time t by a DES with a given input can be computed with the DES *state*, respectively *output*, function (Definition 3).

Definition 3 (DES state and output functions): The *state function* of DES $\mathcal{H}$ is a function $\phi : \mathfrak{U}_d \times \mathbb{R}^{\geq 0} \rightarrow S$, where $\phi(u, t)$ is the state reached at time t by $\mathcal{H}$ with input the discrete event sequence u. Function ϕ is defined inductively as follows:

- $\phi(u, 0) = \textit{jump}(s_0, u(0))$, where s_0 is the initial state of $\mathcal{H}$;
- For each $t > 0$, $\phi(u, t) = \textit{jump}(\textit{flow}(\phi(u, t^*), t - t^*), u(t))$, where: $t^* < t$ is the greatest value such that $u(t^*) \neq 0$ and we let $t^* = 0$ if such a value does not exist (i.e., when u is always 0 before t).

The *output function* of $\mathcal{H}$ is the function $\psi : \mathfrak{U}_d \times \mathbb{R}^{\geq 0} \rightarrow O$ defined as $\psi(u, t) = \textit{output}(\phi(u, t))$. In other words, ψ computes the output (as a function of time) of $\mathcal{H}$ when the input to $\mathcal{H}$ is the discrete event sequence u. In general, $\psi(u, t)$ is not a discrete event sequence (e.g., it may take a non-zero value an infinite number of times).

C. Modelling the property to be verified

We model the property to be verified with a continuous-time *monitor* which observes the state of the system to be verified and checks whether the property under verification is satisfied (Fig. 1b). A temporal logic specification can be transformed into a continuous-time *monitor* as in [25]. The output of our monitor is 0 as long as the property under verification is satisfied and becomes and stays 1 (*sustain*) as soon as the property fails. This non-decreasing property of the monitor output ensures that we never miss a property failure report, even when sampling the monitor output only at discrete time points (Fig. 1c). The use of monitors gives us a flexible approach to model the property to be verified. In particular, it is easy to model bounded safety and bounded liveness properties as monitors.

Since the monitor output is all we need to carry out our verification task, we can model our SUV along with the property to be verified as a DES with an embedded monitor

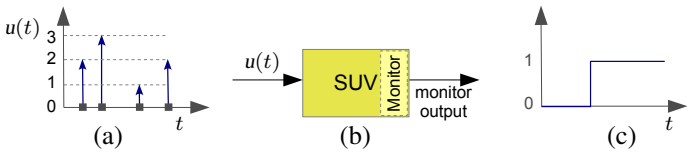


Figure 1: (a) a discrete event sequence ($d = 3$); (b) our SUV with an embedded monitor; (c) the SUV monitor output.

whose set of output values is Bool. We call such a DES a Monitored Discrete Event System (Definition 4 and Fig. 1b).

Definition 4 (Monitored Discrete Event System): A Monitored Discrete Event System (MDES) is a tuple $\mathcal{H} = (S,\ s_0,\ d, \textit{flow},\ \textit{jump},\ \textit{output})$ such that $(S,\ s_0,\ d,$ Bool, *flow*, *jump*, *output*) is a DES whose output function $\psi(u, t)$ is non-decreasing with respect to t. That is, for any input sequence $u \in \mathfrak{U}_d$, for all $t, t' \in \mathbb{R}^{\geq 0}$, if $t \leq t'$ then $\psi(u, t) \leq \psi(u, t')$. In other words, an MDES is a DES with non-decreasing boolean outputs.

D. Modelling SUV operational scenarios

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 in Bounded Model Checking (BMC), we model (Definition 5) scenarios in the SUV operational environment as sequences of disturbances (*disturbance traces*) our SUV is expected to withstand. Each disturbance is an integer in $[0, d]$ and disturbance traces are of finite length h. Given a time quantum $\tau \in \mathbb{R}^+$, a disturbance trace can be associated to a discrete event sequence where all disturbances occur at time points multiple of τ.

Definition 5 (Disturbance trace): Let $h, d \in \mathbb{N}^+$. An (h, d) *disturbance trace* δ is a finite sequence $\delta : [0, h-1] \rightarrow [0, d]$. Given $\tau \in \mathbb{R}^+$ (*time quantum*), to an (h, d) disturbance trace δ we can univocally associate a discrete event sequence u^τ_δ, defined as follows (see also Fig. 2d, ignoring the letters in the disturbance traces). For all $t \in \mathbb{R}^{\geq 0}$, if there exists $k \in [0, h-1]$ such that $t = \tau k$ then $u^\tau_\delta(t) = \delta(k)$, else $u^\tau_\delta(t) = 0$ (no disturbance).

Thus a disturbance trace δ defines an operational scenario (namely, u^τ_δ) for our SUV.

An (h, d) *sequence of disturbance traces* is a finite sequence $\Delta = \delta_0, \ldots, \delta_{n-1}$ of (h, d) disturbance traces. Given $\tau \in \mathbb{R}^+$, to each sequence of disturbance traces $\Delta = \delta_0, \ldots, \delta_{n-1}$ is associated a sequence of discrete event sequences $U^\tau_\Delta = u^\tau_{\delta_0}, \ldots, u^\tau_{\delta_{n-1}}$. Accordingly, we model our SUV *operational environment* as a sequence of disturbance traces Δ since U^τ_Δ defines the operational scenarios our SUV should withstand.

E. The System Level Formal Verification problem

A *System Level Formal Verification (SLFV) problem* is a tuple $P = (h,\ d,\ \tau,\ \Delta,\ \mathcal{H})$ where: $h, d \in \mathbb{N}^+$, $\tau \in \mathbb{R}^+$, $\Delta = \delta_0, \ldots, \delta_{n-1}$ is an (h, d) sequence of disturbance traces, and $\mathcal{H} = (S,\ s_0,\ d, \textit{flow},\ \textit{jump},\ \textit{output})$ is an MDES.

The *answer* to SLFV problem P is *FAIL* if there exists a disturbance trace δ in Δ such that $\psi(u^\tau_\delta, \tau h) = 1$ (in such a case also the *counterexample* δ is returned), *PASS* otherwise.

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

Thus, by taking h *large enough* (as in 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.

F. HILS-based System Level Formal Verification

We use a black-box approach where the MDES $\mathcal{H}$ defining our SUV and property to be verified is defined using the modelling language of a suitable simulator (e.g., MatLab and Stateflow for Simulink).

We compute the answer to an SLFV problem (h, d, τ, Δ, $\mathcal{H}$) by simulating each operational scenario δ in the operational environment Δ. In other words, we are performing an exhaustive (with respect to Δ) HILS.

We drive a *simulator for* $\mathcal{H}$ (that is, a simulator running a model for $\mathcal{H}$) using 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 a sequence of simulator commands.

Using the 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 KB in the case study presented in this paper). We will show how optimised simulation campaigns enable HILS-based distributed multi-core SLFV.

III. Overall approach

Our overall approach is shown in Fig. 3. To define an SLFV problem (h, d, τ, Δ, $\mathcal{H}$), we need to build the sequence of admissible disturbance traces Δ (operational environment).

Of course, it is typically infeasible to define operational environments by listing all their disturbance traces. In [1]

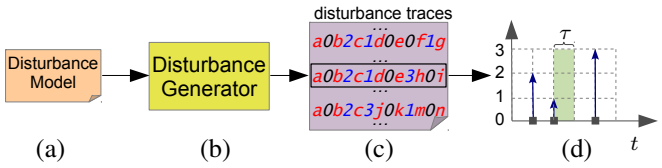


Figure 2: (a) disturbance model; (b) CMurphi-based disturbance generator; (c) generated labelled admissible sequence of disturbance traces ($d = 3, h = 6$, *integers* from 0 to d denote disturbances, *letters* denote labels); (d) the discrete event sequence associated to the trace in the black rectangle in part (c), given time quantum τ.

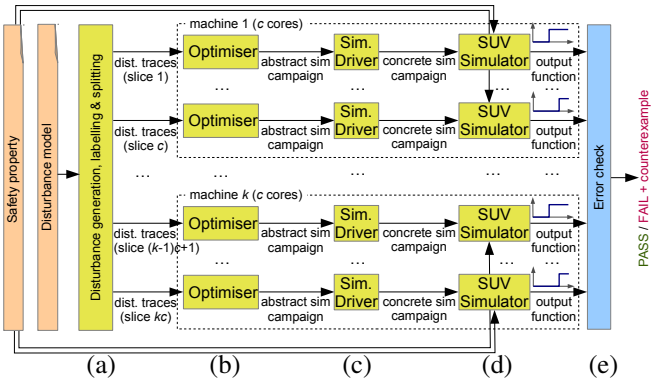


Figure 3: Our approach to distributed multi-core SLFV

it is shown how operational environments can be easily defined using the modelling language of a finite state model checker (CMurphi [4]). We follow the approach in [1] and run CMurphi in Depth-First Search (DFS) mode to generate the operational environment Δ. DFS guarantees that the disturbance traces in Δ are generated in *lexicographic order* (see Fig. 2a–b and Fig. 3a).

Starting from Δ, we aim at generating highly optimised simulation campaigns, which exploit as much as possible the capabilities of modern simulators to store and restore simulation states. Given a disturbance trace $\delta = d_0, \ldots, d_{h-1} \in \Delta$, any prefix of δ univocally identifies a simulation state, given that, to answer our SLFV problem, the simulator is intended to be run under input δ starting from its initial state.

When simulating all traces in Δ, often multiple traces, e.g., $\delta = \hat{d}_0, \ldots, \hat{d}_p, d_{p+1}, \ldots, d_{h-1}$ and $\delta' = \hat{d}_0, \ldots, \hat{d}_p, d'_{p+1}, \ldots, d'_{h-1}$, have a *common prefix*, e.g., $\hat{d}_0, \ldots, \hat{d}_p$. In order to properly exploit the load/store capabilities of the simulator, we would like to proceed as follows. When verifying the first of such traces, e.g., δ, we: (i) run the simulator with input being the common prefix $\hat{d}_0, \ldots, \hat{d}_p$; (ii) *store* under a given label, e.g., l, the state reached by the simulator so far; (iii) continue the simulation of δ by injecting the remaining disturbances of δ, i.e., $d_{p+1}, \ldots, d_{h-1}$. When verifying δ', we: (i) *avoid the recomputation* of the state that the simulator would reach when run on the common prefix of disturbances $\hat{d}_0, \ldots, \hat{d}_p$ by *loading* back the state previously stored under label l; (ii) continue the simulation of δ' by injecting the remaining disturbances of δ', i.e., $d'_{p+1}, \ldots, d'_{h-1}$.

Unfortunately, a naive identification of common prefixes of traces in Δ would be computationally very expensive, given that Δ may contain a huge number of traces (about 4 million in our examples, which need about 3.5GB of memory to be stored). Following [1], we delegate the CMurphi-based disturbance trace generator to produce a *labelled* lexicographically ordered sequence of disturbance traces Δ^λ. Each δ^λ in Δ^λ is of the form $\delta^\lambda = l_0, d_0, l_1, d_1, \ldots, l_{h-1}, d_{h-1}, l_h$, where $\delta = d_0, \ldots, d_{h-1}$ is a disturbance trace in Δ and $l_0, \ldots, l_h$ belong to a countably set of *labels* L (e.g., $\mathbb{N}^+$). Labels are defined by an *injective map* λ from finite sequences of disturbances (including the empty sequence) to L. As a consequence, prefixes $\hat{d}_0, \ldots, \hat{d}_{p-1}$ common to multiple disturbance traces in Δ are followed, in Δ^λ, by the same label $\hat{l}_p = \lambda(\hat{d}_0, \ldots, \hat{d}_{p-1})$ (see Fig. 2c). Given that our CMurphi-based generator runs in DFS mode, traces are produced in lexicographic order and labelled at *no additional computational cost* during generation, as shown in [1]. This allows us to assume that Δ^λ is available as an input. Hence, in our setting, the SLFV can be regarded to be $(h, d, \tau, \Delta^\lambda, \mathcal{H})$ rather than $(h, d, \tau, \Delta, \mathcal{H})$. This greatly simplifies the design of our simulation campaign optimiser (Section IV).

In order to enable parallel computation on $k \in \mathbb{N}^+$ machines with $c \in \mathbb{N}^+$ cores each, we evenly partition the labelled lexicographically ordered sequence of disturbance traces Δ^λ into kc (labelled) lexicographically ordered sequences of disturbance traces $\Delta^\lambda_0, \ldots, \Delta^\lambda_{kc-1}$, by assigning the i-th trace ($0 \leq i < |\Delta^\lambda|$) to the $\lfloor ikc/|\Delta^\lambda| \rfloor$-th slice (Fig. 3a). We use such kc slices to compute in parallel kc highly optimised simulation campaigns (Fig. 3b), which can be simulated in parallel using kc simulators each one running on a different core of our k machines (Fig. 3c–d).

The *answer* to the SLFV problem is *FAIL* if one of the simulation campaigns raises the simulator output function to 1. The answer is *PASS* otherwise. In case the answer is *FAIL*, the driver of the simulator which raised the error can compute a disturbance trace δ (called *counterexample*) in the input slice such that the discrete event sequence associated to δ under time quantum τ (see, e.g., Fig. 2d) would lead the SUV to an error state (Fig. 3e).

IV. Computation of Simulation Campaigns

In this section we describe our RAM-based simulation campaign optimiser which enables multi-core SLFV.

Given a *labelled* (h, d) *lexicographically ordered sequence of disturbance traces* $\Delta^\lambda = \delta^\lambda_0, \ldots, \delta^\lambda_{n-1}$, our optimiser computes a simulation campaign for *any* simulator of *any* DES $\mathcal{H}$ whose set of inputs is $[0, d]$. The computed

Input: Δ^λ, a labelled lex-ordered sequence of disturbance traces
Output: χ , the computed simulation campaign, initially empty

1 *LBT* $\leftarrow$ *buildLBT*(Δ^λ);
2 let l_0 be the first label common to all traces in Δ^λ;
3 *stored* $\leftarrow$ empty set of labels; /* inv: *stored*$\subseteq$*LBT* and |*stored*|$\leq h$ */
4 append *store*(l_0) to χ and add l_0 to *stored*;
5 $i \leftarrow 0$;
6 **foreach** $\delta^\lambda = l_0, d_0, \ldots, l_{h-1}, d_{h-1}, l_h$ *in* Δ^λ **do**
7 *i*++; /* δ^λ is the *i*-the trace in Δ^λ */
8 *t_load* $\leftarrow$ max t s.t. $l_t \in$ *stored*;
9 append *load*(l_{t_load}) to χ;
10 **foreach** *label* $l \in$ *stored s.t. LBT[l].lastTrace* $\leq i$ **do**
11 append *free*(l) to χ;
12 remove l from *stored*;
13 $\hat{d} \leftarrow d_{t\_load}$; *steps* $\leftarrow 1$;
14 **for** $t \leftarrow$ *t_load* $+ 1$ *to* $h - 1$ **do**
15 *toBeStored* $\leftarrow$ ($l_t \in$ *LBT* $-$ *stored* and *LBT*[l_t].*lastTrace* $> i$);
16 **if** *toBeStored or* $d_t \neq 0$ **then**
17 append *run*($\hat{d}$, *steps*) to χ; $\hat{d} \leftarrow d_t$; *steps* $\leftarrow 1$;
18 **if** *toBeStored* **then**
19 append *store*(l_t) to χ and add l_t to *stored*;
20 **else** *steps*++;
21 **return** χ;

---

22 **function** *buildLBT*(Δ^λ)
23 *LBT* $\leftarrow$ empty tree of labels;
  /* for each $l \in$ *LBT*, *LBT[l].lastTrace* stores the index of last trace where it is known to occur */
24 *watched* $\leftarrow$ empty array $[0..h-1]$ of labels;
25 let l_0 be the first label common to all traces in Δ^λ;
26 set l_0 as the root of *LBT* with *LBT[l_0].lastTrace* $\leftarrow |\Delta^\lambda|$;
27 *watched[0]* $\leftarrow l_0$;
28 $i \leftarrow 0$;
29 **foreach** $\delta^\lambda = l_0, d_0, \ldots, l_{h-1}, d_{h-1}, l_h$ *in* Δ^λ **do**
30 *i*++; /* δ^λ is the *i*-th trace in Δ^λ */
31 **for** $t \leftarrow$ *0 to* $h - 1$ *s.t.* $l_t \in$ *LBT* **do** *LBT[l_t].lastTrace* $\leftarrow i$;
32 *t_lbt* $\leftarrow$ max t s.t. $l_t \in$ *LBT*;
33 *t_w* $\leftarrow$ max t s.t. $l_t \in$ *watched*;
34 **if** *t_lbt* $\neq$ *t_w* **then**
  /* label $l_{t\_w} \notin$ *LBT*: add it */
35 *t_child* $\leftarrow$ min $t >$ *t_w* s.t. *watched*[t_*child*] $\in$ *LBT* (if any);
36 add l_{t_w} to *LBT* as child of l_{t_lbt} with *LBT[l_{t_w}].lastTrace* = *i*;
37 move l_{t_child} (if any) as to be child of l_{t_w} in *LBT*;
38 **foreach** $t \leftarrow$ *t_w* $+ 1$ *to* $h - 1$ **do** *watched[t]* $\leftarrow l_t$;
  /* *watched* now contains labels of the last trace */
39 **return** *LBT*;

Algorithm 1: DFS-Optimiser pseudo-code

campaign is *abstract* in that, for all commands of the form *run*(e, t), t is a natural number and not an actual time duration. By providing a time step $\tau \in \mathbb{R}^+$, χ can be instantiated into a *concrete* simulation campaign χ_τ, by replacing all *run*(e, t) commands by *run*$(e, t\tau)$.

The algorithm of our optimiser is shown as Algorithm 1. As the input sequence Δ^λ of labelled disturbance traces can be too big to be kept in main memory, the optimiser reads the input file sequentially twice. In the first scan of Δ^λ, the optimiser builds a data structure called Labels Branching Tree (LBT) as completely as possible within the available RAM. Afterwards, it reads Δ^λ again to produce the abstract simulation campaign from the LBT.

A. LBT construction

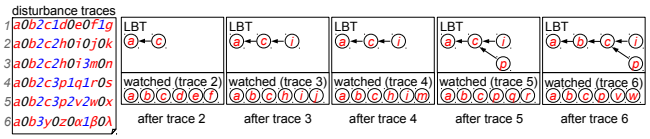
The LBT is a tree of labels rooted at l_0, the first label of all traces (e.g., $l_0 = a$ in Fig. 2c and Fig. 4). The LBT collects *branching labels*, i.e., labels l_i for which there exist at least two labelled disturbance traces $\delta^\lambda = l_0, d_0, \ldots, l_i, d_i, \ldots, l_h$ and $\delta^{\lambda'} = l_0, d_0, \ldots, l_i, d'_i, \ldots, l'_h$ in Δ^λ which are identical up to l_i and such that $d_i \neq d'_i$. Branching labels represent simulator states whose storing may save simulation time (by loading them back later).

Label l_j is a child of l_i in the LBT iff, for all $\delta^\lambda = l_0, d_0, \ldots, l_i, \ldots, l_j, \ldots, l_h \in \Delta^\lambda$, no l_k in δ^λ with $i < k < j$ is in the LBT (note: all such δ^λ are identical at least up to l_j). For each label l in the LBT, the number of the last trace in Δ^λ where it occurs is kept.

The construction of the LBT is shown as function *buildLBT()* in Algorithm 1 (from line 22). The function scans the input slice in order to recognise branching labels, keeping in array *watched* the labels of the last processed trace. In fact, as the traces in Δ^λ are lexicographically ordered, these are the *only* labels that may become branching when processing a new trace. To see why, assume that the optimiser is processing, e.g., trace 2 in Fig. 4a (left). As this trace starts to be different with respect to the previous trace (trace 1) from the disturbance at step 2 (i.e., disturbance 2 right after label c), the optimiser infers that labels d, e, f, g of trace 1 will never occur in later traces of Δ^λ, and will never become branching.

As for the actual recognition of a new branching label and its addition to the LBT, assume that function *buildLBT()* is processing the i-th disturbance trace $\delta = l_0, d_0, \ldots, l_{t\_lbt}, d_{t\_lbt}, \ldots, l_{t\_w}, d_{t\_w}W, \ldots, l_h$ (line 29). Variable *t_lbt* is set to the max index of a label in δ^λ already in the LBT, and *t_w* is the max index of a label in δ which belongs also to array *watched*. As l_0 is put both in the LBT and in *watched[0]* at the beginning, both values are always defined. The algorithm infers that the current trace is identical to the previously processed one up to *t_w*, but differs from it starting from the disturbance to be injected at step *t_w* $+ 1$. If *t_w* $=$ *t_lbt*, label l_{t_w} is already branching, and nothing has to be done. Otherwise, the new label l_{t_w} is recognised as branching, and is added to the LBT as a child of l_{t_lbt} (as, given that the input traces are in lexicographic order, *t_w* $\neq$ *t_lbt* implies *t_w* $>$ *t_lbt*). As l_{t_lbt} could already have children in the LBT, the tree may need to be rearranged to accommodate the new label l_{t_w}. Given that the input traces are in lexicographic order, the last task is very simple, as at most one child of l_{t_lbt} must be moved. This child, if exists, must be a label that occurred in the previous trace, i.e., it belongs to the *watched* array (line 35).

Fig. 4a shows an example of LBT construction starting from a labelled lexicographically ordered sequence of disturbance traces Δ^λ consisting of 6 traces. Note that, out of 25 labels in Δ^λ, only 5 of them belong to the LBT. When



(a)

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

(b)

Figure 4: Simulation campaign optimiser. (a) Construction of an LBT from 6 traces (labels are shown as *letters*). (b) The computed optimised simulation campaign.

processing, e.g., trace 6, $t_{lbt} = 0$ points to label a having index 0 in trace 6 and $t_w = 1$ points to label b having index 1 in trace 6 and occurring in array *watched*. Label b becomes branching as there are previous traces (e.g., trace 5) identical to trace 6 up to b and different from that point on. The new branching label b becomes a child of a in the LBT. The previous child of a, i.e., c, becomes a child of b.

B. Computation of the abstract simulation campaign

Once the LBT is built, the optimiser reads the input slice a second time to compute the abstract simulation campaign, keeping track of which LBT labels are stored in simulator memory at any moment (set *stored*, line 3 of Algorithm 1).

For each $\delta^\lambda = l_0, d_0, \ldots, l_{\text{load}}, \ldots, l_h$ in Δ^λ, let l_{load} be the right-most label of δ^λ currently stored by the simulator (line 8). The optimiser appends to the output campaign the following commands: (i) *load*(l_{load}) (line 9); (ii) *free*(l) for each label $l \in$ LBT which represents a currently stored state that will never occur in future traces (line 12); (iii) a command of the form *run*($\hat{d}$, *steps*) for each maximal subsequence of length *steps* in δ^λ (starting from l_{load}) of the form $\hat{d}, l_{i_1}, 0, l_{i_2}, \ldots, 0, l_{i_{steps}} \tilde{d}, \tilde{l}$ where either $\tilde{d} \neq 0$ or label $\tilde{l}$ needs to be stored (line 17). In the latter case, command *store*($\tilde{l}$) is appended as well (line 19). Label $\tilde{l}$ needs to be stored (see line 15) if it is in the LBT but not yet stored and it will occur again in a later trace.

The maximum number of states that the simulator must keep stored at any moment is bounded by h (the horizon). This is because, when starting the simulation of the portion of χ_τ stemming from any trace $\delta^\lambda \in \Delta^\lambda$, the simulator executes command *load*(l_{load}) with l_{load} being a label in trace δ^λ having index i. Given that the disturbance traces in Δ^λ are in lexicographic order, all labels occurred with indices $> i$ in traces of Δ^λ *before* δ^λ never occur in traces of Δ^λ *after* δ^λ. Hence, currently stored states identified by those labels will not need to be loaded back in the future and can be safely freed (line 12).

Fig. 4(b) shows the simulation campaign computed by the optimiser on the slice in Fig. 4(a). Except for the first command which stores a (the label common to all traces and representing the simulator initial state), each line represents the portion of the simulation campaign stemming from each trace. Note that only the first trace is simulated entirely, while all the others are simulated starting from intermediate, previously stored, states.

C. Optimiser soundness and completeness

Given an SLFV problem $P = (h, d, \tau, \Delta^\lambda, \mathcal{H})$, Algorithm 1 computes a simulation campaign χ_τ which is *sound* and *complete* with respect to Δ^λ. That is: if the answer to P is *PASS*, then the output of the simulator at the end of the execution of the simulation campaign χ_τ will be 0 (*soundness*). On the other hand, if the answer to P is *FAIL* and δ^λ is the first counterexample in Δ^λ, then the output of the simulator will raise from 0 to 1 during the simulation of a command of χ_τ stemming from δ^λ (*completeness*).

The result above can be proved by formalising the notion of *simulator for* $\mathcal{H}$ along the lines of [1].

V. Experimental Results

In this section we evaluate the effectiveness of our *distributed multi-core approach* to SLFV (in short mcSLFV) and compare it with the *distributed single-core approach* (in short scSLFV) of [1]. For this reason we: (i) use the very same case study of [1], i.e., the Fuel Control System (FCS) model included in the Simulink distribution; (ii) run experiments on very similar machines, i.e., multiple 3.0 GHz, 8GB RAM Intel hyperthreaded Quad Core Linux PCs.

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

A. Generation and splitting of simulation scenarios

As [1], we use CMurphi to generate a labelled lexicographically ordered sequence Δ^λ of 4,023,955 disturbance traces. This takes about 28 minutes and produces a 3.5GB file. We then split such a Δ^λ into kc slices, with $k = 8, 16, 32, 64$ and $c = 8$. Splitting takes a few seconds, regardless of the value of kc.

#slices	#traces per slice	scSLFV optimiser	mcSLFV optimiser	time saving %
1	4,023,955	20:27:26	0:7:16	99.41%
2	2,011,977	3:47:57	0:9:43	95.74%
4	1,005,988	1:45:4	0:9:0	91.43%
8	502,994	0:44:27	0:5:27	87.74%
16	251,497	0:16:24	0:2:8	86.99%
32	125,748	0:4:50	0:0:57	80.34%
64	62,874	0:0:51	0:0:29	43.14%
128	31,437	0:0:35	0:0:17	51.43%
256	15,718	0:0:10	0:0:8	20.00%
512	7,859	0:0:5	0:0:4	20.00%

Table I: Comparison between scSLFV optimiser of [1] and our mcSLFV optimiser (time in h:m:s).

#mach	#slices	min	max	avg	$\frac{\text{stddev}}{\text{avg}}$%	speedup	efficiency
8	64	180:3:0	205:19:57	194:17:52	4.979%	54.63×	85.35%
16	128	70:6:4	100:17:53	87:49:56	13.772%	111.56×	87.15%
32	256	44:0:27	57:57:27	48:34:6	10.323%	192.38×	75.15
64	512	18:32:36	26:49:4	23:2:19	11.110%	411.83×	80.43%

Table II: Statistics on the distributed (k = *#mach(ines)*) multi-core ($c = 8$) execution of simulation campaigns (time in h:m:s).

B. Computation of optimised simulation campaigns

Table I compares the performance of our mcSLFV optimiser with those of the scSLFV optimiser of [1]. Column *#slices* gives the number of slices in which the sequence of disturbance traces has been partitioned. Column *#traces per slice* shows the number of traces in any single slice (except the last slice, which may have up to *#slices*-1 more traces, as the overall number of traces is not a multiple of *#slices*). Columns *scSLFV optimiser* and *mcSLFV optimiser* show the maximum time needed by, respectively, the [1] and our optimisers to compute the simulation campaign from a slice. For each row in Table I, the entry in column *time saving %* is defined as $(t_{sc} - t_{mc})/t_{sc}$, where t_{sc} and t_{mc} are, respectively, the entries in columns *scSLFV optimiser* and *mcSLFV optimiser*.

Note that, by exploiting the lexicographical order among traces in the input sequence, our mcSLFV optimiser is always much faster than the scSLFV optimiser of [1].

C. Execution of the simulation campaigns

Table II shows some statistics on the execution time of the simulation campaigns generated by our mcSLFV optimiser. Note that the standard deviation of the simulation time is always very small, always less than 15% with respect to the average time (see column *stddev/avg%*). This shows that the computational load among cores is well balanced.

Column *speedup* shows the ratios t_1/t_{kc}, typically used in the evaluation of parallel algorithms. For each row (k = *#mach(ines)*) of Table II, time t_{kc} is the *overall* time needed to carry out the SLFV task with k c-core machines, i.e., the sum of the disturbance trace generation and splitting time (about 28 minutes), optimisation time (from Table I), and the max simulation time (column *max*) over all the kc = *#slices* slices. Time t_1 (serial time) is the overall time needed to carry out the SLFV task when only one core is used. Let t_{kc}^{avg} be the average time to simulate a slice where kc = *#slices* cores are used (row *#mach(ines)* $= k$, column *avg*). When using kc cores, the serial time can be estimated as $kc \times t_{kc}^{\text{avg}}$. As this value changes a little bit for different values of k, we estimated serial time t_1 as $\min\{64t_{64}^{\text{avg}}, 128t_{128}^{\text{avg}}, 256t_{256}^{\text{avg}}, 512t_{512}^{\text{avg}}\}$. This leads to $t_1 = 491.5$ days. From such a huge value it follows that estimation is the only viable way to compute t_1. Note that in our computation we are slightly overestimating the serial time, since we are assuming that the first trace of each slice must be simulated from the initial state. In an actual 1-core execution of a simulation campaign, the optimiser may exploit stored simulator states to avoid simulation of such traces from the initial state. As the time to simulate a single trace is of a few seconds, this is negligible with respect to the value of t_1.

Column *efficiency* in Table II is computed, as typically done in the evaluation of parallel algorithms, by dividing the speedup by the number of parallel processes kc = *#slices*.

In the same fashion, we can estimate the serial time and efficiency of the scSLFV approach of [1]. Serial time is about 200 days, and efficiency is higher than ours, being almost 1. Our loss of efficiency stems from the fact that processes running on different cores of the same machine share the RAM.

	scSLFV		mcSLFV		
#machines	#slices	time	#slices	time	time saving %
8	8	711:3:33	64	205:49:20	71.05%
16	16	343:24:27	128	100:47:4	70.65%
32	32	167:6:9	256	58:26:29	65.03%
64	64	81:49:3	512	27:18:2	66.63%

Table III: Completion time of the parallel simulation (i.e., completion time of the *longest* campaign) with respect to the approach of [1] (time in h:m:s).

Such an efficiency measure does not take into account the cost of the hardware. In fact, to enable kc-parallel processes, the scSLFV approach needs kc machines, whereas our mcSLFV approach needs only k machines. Table III investigates such an issue, by showing the time saving realised by our mcSLFV approach. In particular, for each value of k (*#machines*), columns *scSLFV* and *mcSLFV* show the number of processes (*#slices*) that can be run on the given machines and the time needed (*time*) to complete the verification task with, respectively, the approach of [1] and ours. For each row, the entry in column *time sav-*

ing % is defined as $(t_{sc} - t_{mc})/t_{sc}$, where t_{sc} and t_{mc} are, respectively, the entries in columns *scSLFV time* and *mcSLFV time*. Table III shows that, when using the same hardware, our distributed multi-core approach saves at least 65% of verification time. For example, using 64 machines, the verification task using the single-core approach of [1] would need 81 hours, while ours needs less than 27 hours. Note that a serial approach to verification would need more than 200 days.

VI. Conclusions

We have presented a distributed multi-core approach to HILS-based SLFV. We have implemented our algorithms and run experiments on a large control system case study in the Simulink distribution, whose operational environment consists of more than 4 million simulation scenarios. Our distributed multi-core approach allows us to complete the verification of such a system in about 27 hours using 64 8-core machines, whereas a sequential computation would require more than 200 days.

To the best of our knowledge, this is the first time that a distributed multi-core algorithm for HILS-based SLFV is presented.

Acknowledgements: Work partially supported by FP7 projects SmartHG (317761) and PAEON (600773). We thank our reviewers for their valuable comments.

References

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

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

[3] F. Cavaliere, F. Mari, I. Melatti, G. Minei, I. Salvo, E. Tronci, G. Verzino, and Y. Yushtein, "Model checking satellite operational procedures," in *Proc. DASIA 2011*, 2011.

[4] 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.

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

[6] 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.

[7] 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.

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

[9] 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.

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

[11] 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.

[12] C. Yang and D. Dill, "Validation with guided search of the state space," in *Proc. DAC 1998*. ACM, 1998, pp. 599–604.

[13] P. Ho, T. Shiple, K. Harer, J. Kukula, R. Damiano, V. Bertacco, J. Taylor, and J. Long, "Smart simulation using collaborative formal and simulation engines," in *Proc. ICCAD 2000*. IEEE, 2000, pp. 120–126.

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

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

[16] U. Stern and D. Dill, "Parallelizing the Murphi Verifier," *Form. Methods Syst. Des.*, vol. 18, no. 2, pp. 117–129, 2001.

[17] 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.

[18] I. Melatti, R. Palmer, G. Sawaya, Y. Yang, R. Kirby, and G. Gopalakrishnan, "Parallel and distributed model checking in eddy," *Int. J. Softw. Tools Technol. Transf.*, vol. 11, no. 1, pp. 13–25, 2009.

[19] B. Bingham, J. Bingham, F. De Paula, J. Erickson, G. Singh, and M. Reitblatt, "Industrial strength distributed explicit state model checking," in *Proc. PDMC-HIBI 2010*. IEEE, 2010, pp. 28–36.

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

[21] 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.

[22] A. Wijs, "Towards informed swarm verification," in *Proc. NFM 2011*, ser. LNCS, vol. 6617. Springer, 2011.

[23] M. Staats and C. S. Pasareanu, "Parallel symbolic execution for structural test generation," in *Proc. ISSTA 2010*. ACM, 2010, pp. 183–194.

[24] E. Sontag, *Mathematical Control Theory: Deterministic Finite Dimensional Systems*. Springer, 1998.

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