System Level Formal Verification via Model Checking Driven Simulation

Toni Mancini, Federico Mari, Annalisa Massini, Igor Melatti, Fabio Merli, and Enrico Tronci

Computer Science Department - Sapienza University of Rome
Via Salaria 113, I-00198 Roma, Italy
{tmancini,mari,massini,melatti,merli,tronci}@di.uniroma1.it

Abstract. We show how by combining *Explicit Model Checking* techniques and simulation it is possible to effectively carry out (bounded) *System Level Formal Verification* of large *Hybrid Systems* such as those defined using *model-based* tools like *Simulink*.

We use an *explicit model checker* (namely, *CMurphi*) to generate all possible (*finite horizon*) simulation scenarios and then optimise the simulation of such scenarios by exploiting the ability of simulators to *save* and *restore* visited states. We show feasibility of our approach by presenting experimental results on the verification of the fuel control system example in the Simulink distribution. To the best of our knowledge this is the first time that (exhaustive) verification has been carried out for hybrid systems of such a size.

1 Introduction

System Level Verification of Embedded Systems has the goal of verifying that the *whole* (i.e., software + hardware) system meets the given specifications. 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.

The main concerns in a HILS campaign are: the effort needed to define the simulation scenarios (may require months of work from expert designers), the time needed to carry out the campaign itself (may require weeks or even months of simulation activity), the degree of *assurance* achieved at the end of the HILS campaign. In this paper we show how using Explicit Model Checking (EMC) techniques it is possible to advance the state of the art on all points above.

1.1 Main Contributions

Our System Under Verification (SUV) is a *Hybrid System* (e.g., see [1] 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

N. Sharygina and H. Veith (Eds.): CAV 2013, LNCS 8044, pp. 296–312, 2013.
© Springer-Verlag Berlin Heidelberg 2013

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.

A system is expected to *withstand* all disturbance sequences that may arise in its operational scenario. Correctness of a system is thus defined with respect to such *admissible* disturbance sequences. The set of admissible disturbance sequences typically satisfies constraints like the following: 1) the number of failures occurring within a certain period of time is below a given threshold; 2) the time interval between two consecutive failures is greater than a given threshold; 3) a failure is repaired within a certain time, etc. Thus, in our setting, the set of admissible disturbance sequences (*disturbance model*) can be defined using a Finite State Automaton, which in turn can be defined using the modelling language of any finite state model checker. To this end we use CMurphi [10,9] since its rule based modelling language turns out to be quite handy to define admissible disturbance sequences.

In such a framework we address *Bounded* System Level Formal Verification (SLFV) of *safety* properties. That is, given a time horizon T and a time step τ (time quantum between disturbances) we return *PASS* if there is no *admissible* disturbance sequence of length T 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 parallel approach to SLFV, we split the verification process into two main phases. First, an *off-line* phase, where Explicit Model Checking techniques are used to compute, from the disturbance model, say k, highly optimised simulation campaigns for a set of k simulators. Second, an *on-line* parallel 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.

Note that if an error is found, only the *on-line* phase above has to be repeated since the set of admissible simulation scenarios computed in the *off-line* phase does not change. The *on-line* phase is supported by simulation tools (Simulink in our examples). Here we provide methods and tools to effectively carry out the *off-line* phase computing *optimised* simulation campaigns for the available simulator.

While most *Model Based Testing* approaches focus on modelling the SUV, in this work we model the set of disturbances the SUV is supposed to withstand. Accordingly, the performance of our *off-line* phase does not depend on the SUV model and only depends on the disturbance model. On the other hand, simulation times in the *on-line* phase depend on the size of SUV and disturbance models.

We implemented our approach and present experimental results on its usage on the fuel control system example in the Simulink distribution. In our experiments we set our *time horizon* to 100 seconds and our *time step* to 1 second. Our main contributions can be summarised as follows.

Automatic Exhaustive Simulation Scenario Generation. We show how a suitable search on the transition graph of (the automaton defining) the disturbance model can be

used to generate *all and only* the admissible disturbance sequences (simulation scenarios) split into k disjoint *slices*. Such an initial partitioning of the simulation scenarios allows us to distribute our later steps among k parallel processes. We implemented such an *exhaustive simulation scenario generator* within the CMurphi model checker. Our case study disturbance model yields about 4 million disturbance traces (simulation scenarios) stored into a 3.5GB file. To generate such traces our algorithm takes about 30 minutes and within 15 seconds splits them into $k = 64$ *slices* of equal size.

Simulation Campaign Optimisation. We present a disk based *optimisation* algorithm that transforms a sequence of simulation scenarios into a very efficient *simulation campaign*, that 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). Our algorithm will be run in parallel on k processors, each taking as input a different slice of the simulation scenarios. For example, when using $k = 8$ [$k = 64$] parallel processors our algorithm can compute k disjoint optimised simulation campaigns for our case study in about 44 minutes [one minute]. Simulation of the optimised campaign for $k = 64$ takes about 3 days, whereas simulation of the *unoptimised* one, that is a simulation campaign that does not exploit the save/restore features of simulators, thereby always restarting scenario simulations from the initial state, takes about 12 days (i.e., the speed-up is about $3.8\times$). Similarly to Explicit Model Checking (e.g., CMurphi [9], SPIN [14]), our simulation campaign optimiser counteracts *scenario explosion* by avoiding as much as possible revisiting already visited simulator states. Since the size of a simulation state can easily take many MB, it is not possible to store too many states, even resorting to the disk. Thus, a clever strategy is needed to decide when to save a visited state or just to recompute it. This is what our simulation campaign optimiser does, thereby transforming a simulator into a sort of *explicit model checker*.

Summing Up. We show how using explicit model checking techniques it is possible to generate optimised simulation campaigns for a set of simulators. This enables parallel HILS based SLFV. We show the effectiveness of the proposed approach on an industrial case study in the Simulink distribution. To the best of our knowledge this is the first time that SLFV is carried out for a *real world* hybrid system of such a size.

1.2 Related Work

The paper closest to ours is [6] where CMurphi 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. Along the same line of thinking, in [16] the analogous SPIN capability has been used to verify actual C code. Such approaches differ from ours since optimisation of the simulation campaign is not considered. Safety checking has been widely investigated in a finite state setting (e.g., see [22] and citations thereof). In our setting, *black box* verification of continuous time hybrid systems, we check specifications using *monitors*, similarly to [18].

Statistical model checking, being basically *black box*, is also closely related to our approach. In such a setting, [31] 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., [23,27,12]) 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. It is worth noticing that in trading off *precision* of the answer to the SLFV problem and *size* of the set of admissible scenarios, statistical model checking and our proposed approach are somehow complementary. In fact, the former returns a *statistical* answer but can consider (potentially) infinite sets of admissible scenarios whereas the latter, being exhaustive, returns a *precise* answer, but can only address finite sets of admissible scenarios.

Formal verification of Simulink models has been widely investigated, examples are in [26,19,29]. 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 [31] and for our exhaustive HILS based approach.

Of course *Model Based Testing* (e.g., see [5]) has widely considered automatic generation of test cases from models. In our HILS setting, automatic generation of simulation scenarios (for Simulink) has been investigated in [11,17,4,28]. The main differences with our approach are the following. First, such approaches cannot be used in our *black box* setting since they generate simulation scenarios from the Stateflow/Simulink model of the SUV (whereas we generate scenarios from the disturbance model). Second, the above approaches are not exhaustive, whereas ours is.

Synergies between simulation and formal methods have been widely investigated in digital hardware verification. Examples are in [30,13,21,7] 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 [7].

Parallel algorithms for explicit state exploration have been widely investigated. Examples are in [25,2,20,3,15]. 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 an embarrassing parallel (*map and reduce like* [8]) strategy that splits (*map* step) the set of simulation scenarios into equal size subsets to be simulated on different processors and stops verification as soon as one of such processors finds an error (*reduce* step).

1.3 Outline of the Paper

Section 2 defines how we model disturbances, SUV, and our SLFV problem. Section 3 formalises the notion of simulator. Section 4 outlines how disturbance traces are generated from a CMurphi model. Section 5 outlines our simulation campaign optimisation

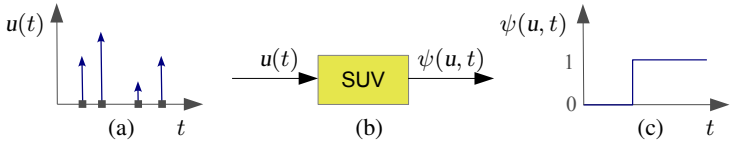

Fig. 1. (a) a discrete event sequence $u \in \mathfrak{U}_d$; (b) our SUV; (c) the SUV output $\psi(u, t)$

algorithm and proves its correctness. Section 6 outlines how we execute simulation campaigns on Simulink and presents experimental results.

2 Bounded System Level Formal Verification

In this section we define the system level verification problem we address (Definition 7). To this end we model disturbances (Definition 1), our SUV (Definitions 2, 3, 4) as well as the class of disturbances (Definitions 5, 6) our SUV is supposed to withstand in its operational scenario.

Throughout the paper, we use $\mathbb{R}^{\geq 0}$, the set of non-negative reals, to represent time, $\mathbb{R}^{+}$, the set of strictly positive reals, to represents non-zero time durations, and Bool $= \{0, 1\}$ to represent Booleans. $\mathbb{N}^{+}$ is the set of positive natural numbers.

A *discrete event sequence* (Definition 1(a) and Fig. 1a), is a function associating to each (continuous) time instant a disturbance event (such as a fault, a variation in system a parameter, etc). As no system can withstand an infinite number of disturbances within a finite time, we require that, in any time interval of finite length, only a finite number of disturbances can take place. We represent with the integer 0 the event carrying no disturbance and with positive integers actual disturbances. Thus we have that, in any finite time interval, a discrete event sequence differs from 0 only in a finite number of time points. An *event list* (Definition 1.b) provides an explicit representation for a discrete event sequence by listing event/time distance pairs for disturbance events.

Definition 1 (Discrete event sequence and event list). *Let* $d \in \mathbb{N}^{+}$.

(a) A discrete event sequence *over integer interval* $[0, d]$ *is a function* $\mathsf{u} : \mathbb{R}^{\geq 0} \to [0, d]$ *such that, for all* $t \in \mathbb{R}^{\geq 0}$*, the set* $\{\tilde{t} \mid 0 \leq \tilde{t} \leq t \text{ and } \mathsf{u}(\tilde{t}) \neq 0\}$ *has finite cardinality. Following control engineering notation for input functions to dynamical systems (e.g., see [24]), we denote with* $\mathfrak{U}_d$ *the set of discrete event sequences over* $[0, d]$.

(b) An event list *on* $[0, d]$ *is a (finite or infinite) sequence of pairs:* $(u_0, \tau_0), (u_1, \tau_1), \ldots$ *such that for all* $i \geq 0$*,* $u_i \in [0, d]$ *and* $\tau_i \in \mathbb{R}^{+}$*. Each event list denotes a unique discrete event sequence* $\mathsf{u}(t)$ *defined as follows:* $\mathsf{u}(0) = u_0$ *and, for each* $t > 0$*, if there exist an integer* $h \geq 0$ *such that* $t = \sum_{i=0}^{h} \tau_i$ *and* (u_{h+1}, τ_{h+1}) *is in the event list, then* $\mathsf{u}(t) = u_{h+1}$*, else* $\mathsf{u}(t) = 0$.

In our setting the system to be verified can be modelled as a Discrete Event System (Definition 2 and Fig. 1b), that is, a continuous time *Input-State-Output* deterministic dynamical system [24] whose input functions are discrete event sequences, whose state can undertake continuous as well as discrete changes, and whose output ranges on any combination of discrete and continuous values.

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).*

– $\textit{flow} : S \times \mathbb{R}^{\geq 0} \to S$. *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* $s \in S$, $\textit{flow}(s, 0) = s$.

– $\textit{jump} : S \times [0, d] \to S$. *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* $s \in S$, $\textit{jump}(s, 0) = s$.

– $\textit{output} : S \to O$. *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, Fig. 1c).

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

– $\phi(\mathrm{u}, 0) = \textit{jump}(s_0, \mathrm{u}(0))$, *where* s_0 *is the initial state of* $\mathcal{H}$*;*

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

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

We model the property to be verified with a continuous-time *monitor* that observes the state of the system to be verified and checks whether the property under verification is satisfied. Thus we can handle any property for which a monitor exists. In particular bounded safety and bounded liveness properties can easily be modelled using monitors. Checking properties with Simulink monitors can be done as outlined in [18].

Since we observe our monitor output only at discrete time points, we may miss a property failure report. To avoid this, we ask our monitor output to be 0 as long as the property under verification is satisfied and to become and stay 1 (*sustain*) as soon as the property fails. Since the monitor output is all we need to carry out our verification task, we model our System Under Verification as a DES with an embedded monitor whose set of output values is Bool. We call Monitored DES such a DES (Definition 4, summarised in Fig. 1).

Definition 4 (Monitored DES). *A Monitored DES (MDES) is a tuple* $\mathcal{H} = (S, s_0, d, \textit{flow}, \textit{jump}, \textit{output})$ *such that* $(S, s_0, d, \textit{Bool}, \textit{flow}, \textit{jump}, \textit{output})$ *is a DES whose output function* $\psi(\mathrm{u}, t)$ *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.

Admissible disturbance sequences (or *traces*) formally model the set of operational scenarios our SUV is supposed to *withstand*. It is typically infeasible to explicitly list all such scenarios manually. Therefore, in our *Model Based* approach we define (Definition 5) them with a suitable finite state automaton with *guarded* transitions and initial as well as final (accepting) states. An (admissible) disturbance trace is then a sequence of transitions from an initial to a final state (Definition 6). In our setting, guards (*adm* in Definition 5) are used to define the set of disturbances that may occur in a given state, whereas final states are used to model constraints on whole disturbance traces. For example, if our set of disturbance traces consists of traces where a certain disturbance event (say, d_1) only occurs at most three times but never immediately after disturbance d_2, then we can use guard *adm* to disable occurrence of d_1 immediately after d_2 and take as final states those where d_1 has occurred at most three times.

Definition 5 (Disturbance generator). *A Disturbance Generator (DG) is a tuple $\mathcal{D} = (Z, d, \mathsf{dist}, \mathsf{adm}, Z_I, Z_F)$ 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.*
- *$\mathsf{dist} : Z \times [0, d] \rightarrow Z$ is a (*deterministic transition*) function mapping each state/disturbance pair (z, e) to a* next *state $\mathsf{dist}(z, e)$.*
- *$\mathsf{adm} : Z \times [0, d] \rightarrow$ Bool is a (*guard*) function defining (*the characteristic function of*) the set of disturbances admissible (i.e., that may occur) in a given state.*

A disturbance generator, being a finite state automaton, can be defined using the input language of any finite state model checker. Note that we model simultaneous disturbances as one single event (i.e., one disturbance). Definition 6 defines disturbance traces (simulation scenarios) as paths from initial to final states in a DG. Since we are in a *Bounded Model Checking* setting, we focus on disturbance traces of finite length.

Definition 6 (Disturbance trace and associated event list). *Let $\mathcal{D} = (Z, d, \mathsf{dist}, \mathsf{adm}, Z_I, Z_F)$ 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, a* disturbance path *of length h for $\mathcal{D}$ 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 $0 \leq i < h$, $z_i \in Z$, $d_i \in [0, d]$, $\mathsf{adm}(z_i, d_i) = 1$, and $z_{i+1} = \mathsf{dist}(z_i, d_i)$.*

(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 $\mathcal{D}$.*

(c) Given a time step $\tau \in \mathbb{R}^+$, the event list associated to a disturbance trace *$\delta = d_0, \ldots, d_{h-1}$ with respect to τ evenly maps the events in δ on the time axis at time points multiple of τ. Formally, the* event list associated *to δ with respect to a time step $\tau \in \mathbb{R}^+$ is $u_\tau(\delta) = (d_0, \tau), \ldots, (d_{h-1}, \tau)$.*

(d) We denote with $\Delta^h_{\mathcal{D}}$ the set of all disturbance traces of length h for $\mathcal{D}$.

System Level Formal Verification (SLFV) (Definition 7) aims at verifying that our SUV (modelled as an MDES) can *withstand* all disturbance traces (defined with a DG) that may occur in the SUV operational environment.

Definition 7 (System Level Formal Verification problem). *A System Level Formal Verification (SLFV) problem is a tuple* $(\mathcal{H}, \mathcal{D}, \tau, h)$*, where:* $\mathcal{H} = (S, s_0, d, \mathsf{flow}, \mathsf{jump}, \mathsf{output})$ *is an MDES (modelling our SUV),* $\mathcal{D} = (Z, d, \mathsf{dist}, \mathsf{adm}, Z_I, Z_F)$ *is a DG (modelling our SUV operational scenario and whose set of outputs* $[0, d]$ *is equal to the set of input values of* $\mathcal{H}$*),* $\tau \in \mathbb{R}^+$ *is a* time step *(for disturbance occurrences), and* $h \in \mathbb{N}^+$ *is a* horizon *(for our error search).*

Let ψ *be the output function (Definition 3) of* $\mathcal{H}$*. The* answer *to a SLFV problem* $(\mathcal{H}, \mathcal{D}, \tau, h)$*, denoted by* $\mathcal{H}(\Delta^h_{\mathcal{D}})$*, is:*

– *FAIL if there exists* $\delta \in \Delta^h_{\mathcal{D}}$ *(*counterexample*) such that* $\psi(\mathsf{u}_\tau(\delta), \tau h) = 1$ *(i.e., the MDES modelling our SUV signals an error by outputting 1 when given as input the discrete event sequence associated to* δ*);*

– *PASS otherwise (i.e., for all* $\delta \in \Delta^h_{\mathcal{D}}$*,* $\psi(\mathsf{u}_\tau(\delta), \tau h) = 0$*, as* ψ *is non-decreasing by Definition 4).*

In case the answer is FAIL, an error (witnessed by δ*) exists in the SUV (namely, in its software, in its hardware mathematical model or in their interaction).*

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 System Level Formal Verification (SLFV) problem we only need to check a finite number of disturbance traces (Definition 7). This is because we are bounding: (i) our time horizon to $T = \tau h$, (ii) the set of time points at which disturbances can take place, by taking τ as the time quantum among disturbance events. Thus, we should make h *large enough* (as in bounded model checking) and τ *small enough* in order to faithfully model our SUV operational scenario. Indeed, as no physical system can withstand arbitrarily (time) close disturbances, any operational scenario can be modelled with the desired precision by suitably choosing τ and h. On such considerations rests the effectiveness of our approach.

3 Simulators and Simulation Campaigns

In HILS based verification the SUV model (for example, a DES defined using MatLab and Stateflow) runs on a simulator (e.g., Simulink) taking as inputs simulation scenarios (disturbance traces in our formal setting). Because of the huge number (about 4 millions in our case study) of simulation scenarios to be considered for exhaustive HILS, the overall number of simulation steps may be prohibitively large if we simulate each scenario from the initial state of the (SUV) simulator. We counteract such a *scenario explosion* by avoiding as much as possible revisiting already visited simulator states, similarly to Explicit Model Checking algorithms (e.g., CMurphi [9] or SPIN [14]). Unfortunately, in our setting each simulator state can be a quite large file (e.g., about 150 KB in our case study). Thus, a clever strategy is needed to decide when to save a visited state or when to just recompute it. Section 5 shows such a strategy. Here, we formalise the notion of DES simulator (Definition 8) to support design and analysis of such strategies.

Definition 8 (DES simulator). *A* DES simulator *is a tuple* $\mathcal{S} = (\mathcal{H}, L, W, m)$ *where:* $\mathcal{H} = (S, s_0, d, O, \mathit{flow}, \mathit{jump}, \mathit{output})$ *is a DES;* L *is a set of* labels *(denoting simulator states);* W *is the set of simulator* states*;* $m \in \mathbb{N}^+$ *denotes the maximum number of states the simulator can store.*

Each $w \in W$ *is a tuple* (s, u, M) *where:* $s \in S$ *is a state of* $\mathcal{H}$ *or a distinguished sink state* $\perp$*;* $\mathrm{u} \in \mathfrak{U}_d$ *is an event list;* M *(*simulator memory*) is a set of at most* m *triples* $(l, s, \mathrm{u}_s) \in L \times S \times \mathfrak{U}_d$*, such that, for each* $l \in L$*, there exists at most one triple* $(l, s, \mathrm{u}_s) \in M$ *where* l *occurs. The simulator initial state is* $w_0 = (s_0, \emptyset, \emptyset)$*.*

Each triple in the simulator memory M binds a label $l \in L$ to a state $s \in S$ of $\mathcal{H}$ and to an event list u_s. Definition 9 gives the semantics of simulator commands.

Definition 9 (Simulator commands and transition function). *Let* $\mathcal{S} = (\mathcal{H}, L, W, m)$ *be a DES simulator, with* $\mathcal{H} = (S, s_0, d, O, \mathit{flow}, \mathit{jump}, \mathit{output})$*.*
The commands for $\mathcal{S}$ *are:* $\mathrm{load}(l)$*,* $\mathrm{store}(l)$*,* $\mathrm{free}(l)$*,* $\mathrm{run}(e, t)$*, where* $l \in L$ *is a label,* $t \in \mathbb{R}^+$ *is a time duration, and* $e \in [0, d]$ *is an event (*l, t, e *are command arguments).*
The transition function *of* $\mathcal{S}$*,* $\mathrm{sim}_{\mathcal{S}}$*, defines how the internal state of* $\mathcal{S}$ *changes upon execution of a command. Namely:* $\mathrm{sim}_{\mathcal{S}}(s, \mathrm{u}, M, \mathrm{cmd}(\mathrm{args})) = (s', \mathrm{u}', M')$ *when* $\mathcal{S}$ *moves from state* (s, u, M) *to* (s', u', M') *upon processing* cmd *with arguments* args*.*

The function is defined as follows:

- $\mathrm{sim}_{\mathcal{S}}(s, \mathrm{u}, M, \mathrm{load}(l)) = (s', \mathrm{u}', M)$*, if* $s \neq \perp$ *and* $(l, s', \mathrm{u}') \in M$*.*
- $\mathrm{sim}_{\mathcal{S}}(s, \mathrm{u}, M, \mathrm{free}(l)) = (s, \mathrm{u}, M \setminus \{(l, s', \mathrm{u}')\})$*, if* $s \neq \perp$ *and* $(l, s', \mathrm{u}') \in M$*.*
- $\mathrm{sim}_{\mathcal{S}}(s, \mathrm{u}, M, \mathrm{store}(l)) = (s, \mathrm{u}, M \cup \{(l, s, \mathrm{u})\})$*, if* $s \neq \perp$*,* $|M| < m$ *and* $\neg\exists s', \mathrm{u}'[(l, s', \mathrm{u}') \in M]$*.*
- $\mathrm{sim}_{\mathcal{S}}(s, \mathrm{u}, M, \mathrm{run}(e, t)) = (s', \mathrm{u}', M)$*, where* $s \neq \perp$*,* $s' = \mathrm{flow}(\mathrm{jump}(s, e), t)$*, and* u' *is* (e, t) *concatenated to* u*.*
- $\mathrm{sim}_{\mathcal{S}}(s, \mathrm{u}, M, \mathrm{cmd}(\mathrm{args})) = (\perp, \mathrm{u}, M)$*, in all the other cases.*

Given a sequence of simulation scenarios (formally represented as disturbance traces), we can build a sequence of commands (*simulation campaign*) driving the simulator through such scenarios. We define the simulator *output sequence* as the sequence of the SUV outputs associated with the simulator states traversed by a simulation campaign. Conversely, given a simulation campaign, we can compute the sequence of scenarios (*event lists*) simulated by it. These concepts are formalised in Definition 10.

Definition 10 (Simulation campaign and output sequence). *Let* $\mathcal{S} = (\mathcal{H}, L, W, m)$ *be a simulator and* $\mathrm{sim}_{\mathcal{S}}$ *be its transition function.*

(a) A simulation campaign *of length* c *for* $\mathcal{S}$ *is a sequence* $\chi = \mathrm{cmd}_0(\mathrm{args}_0), \ldots, \mathrm{cmd}_{c-1}(\mathrm{args}_{c-1})$ *of commands along with their arguments.*

(b) Each simulation campaign univocally defines a sequence of simulator states *traversed by the simulator while executing the simulation campaign. Formally, the* sequence of simulator states *of* $\mathcal{S}$ *with respect to a simulation campaign* χ *(as above) is* $(s_0, \mathrm{u}_0, M_0), \ldots, (s_c, \mathrm{u}_c, M_c)$ *where* s_0 *is the initial state of* $\mathcal{H}$*,* $\mathrm{u}_0 = \emptyset$*,* $M_0 = \emptyset$*, and for all* $0 \leq j < c$*,* $\mathrm{sim}_{\mathcal{S}}(s_j, \mathrm{u}_j, M_j, \mathrm{cmd}_j(\mathrm{args}_j)) = (s_{j+1}, \mathrm{u}_{j+1}, s_{j+1})$*.*

(c) A simulation campaign is admissible *if it is actually executable, i.e., iff* $s_c \neq \perp$*.*

(d) The output sequence *associated to an admissible simulation campaign* χ *is* $\mathit{output}(s_0), \mathit{output}(s_1), \ldots, \mathit{output}(s_c)$*.*

Note that, when $\mathcal{H}$ is a MDES (Definition 4), the output sequence of any simulation campaign χ on $\mathcal{S}$ is non-decreasing, as $s_0, \ldots, s_c$ are the S-components of the sequence of simulator states $(s_0, \mathsf{u}_0, M_0), \ldots, (s_c, \mathsf{u}_c, M_c)$ traversed in that order. Therefore, the output sequence will be 0 as long as the property under verification is satisfied, and goes to (and stays at) 1 as soon as a property violation is detected by the monitor.

The event list sequence associated to a simulation campaign (Definition 11) is the sequence of the event lists associated to the simulator states where the simulator executes a *load* command, plus the event list associated to the simulator final state s_c. Forthcoming Definition 11 and Theorem 1 will be used to state (and prove) the correctness of our simulation campaign optimisation algorithm outlined in Section 5.

Definition 11 (Event list sequence associated to a simulation campaign). *Let $\mathcal{S} = (\mathcal{H}, L, W, m)$ be a simulator, $\mathsf{sim}_{\mathcal{S}}$ be its transition function, $\chi = \mathsf{cmd}_0(\mathsf{args}_0), \ldots, \mathsf{cmd}_{c-1}(\mathsf{args}_{c-1})$ a simulation campaign for $\mathcal{S}$ and $(s_0, \mathsf{u}_0, M_0), \ldots, (s_c, \mathsf{u}_c, M_c)$ be the sequence of simulator states associated to χ.*

The event list sequence *associated to χ is $U(\chi) = \mathsf{u}_{j_0}, \ldots, \mathsf{u}_{j_{v-1}}, \mathsf{u}_c$, where, for all $0 \leq r < v$, u_{j_r} is the event list associated to the state where the simulator executes the r-th load command in χ (i.e., $\mathsf{cmd}_{j_r} = \mathsf{load}$ and there are exactly r load commands in χ before cmd_{j_r}), and u_c is the event list associated to the final simulator state.*

Theorem 1 links inputs (simulation campaigns) for a simulator $\mathcal{S}$ for $\mathcal{H}$ to inputs (event lists) for $\mathcal{H}$: for each simulation campaign χ, the event list u of any simulator state (s, u, M) traversed by $\mathcal{S}$ while executing χ drives $\mathcal{H}$ from its initial state s_0 to s.

Theorem 1. *Let $\mathcal{S} = (\mathcal{H}, L, W, m)$ be a simulator for $\mathcal{H}$, χ be an admissible simulation campaign for $\mathcal{S}$ of length c, and $(s_0, \mathsf{u}_0, M_0), \ldots, (s_c, \mathsf{u}_c, M_c)$ be the sequence of simulator states of $\mathcal{S}$ with respect to χ.*

For each $0 \leq j \leq c$, event list u_j in state (s_j, u_j, M_j) has form $(v_0, \tau_0), \ldots, (v_{q_j-1}, \tau_{q_j-1})$ and defines a discrete event sequence that drives $\mathcal{H}$ from its initial state s_0 to state s_j in time $T_j = \sum_{r=0}^{q_j-1} \tau_r$ (where $T_j = 0$ if $q_j = 0$).

4 Automatic Generation of Exhaustive Simulation Scenarios

In this section we outline our approach to disturbance modelling and disturbance trace generation. Each disturbance trace prefix identifies a simulator state. To allow generation and optimisation of simulation campaigns (Section 5), we associate a unique label to each of such prefixes (Definition 12).

Definition 12 (Labelling of disturbance traces). *Let $d \in \mathbb{N}^+$ and L be a countably infinite set of labels (e.g., $\mathbb{N}^+$). 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 L.*

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

We now outline our disturbance trace generation algorithm. We model the finite state automaton DG $\mathcal{D} = (Z, d, \mathit{dist}, \mathit{adm}, Z_I, Z_F)$ using the CMurphi [10,9] finite state

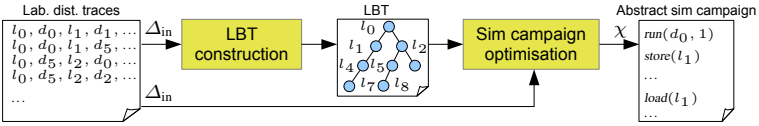

Fig. 2. High-level view of our simulation campaign optimiser

model checker, modified so as to generate all paths of length h from a DG initial state to a final state. Of course, any other finite state model checker, e.g., SPIN [14], could be used for this purpose. CMurphi has a rule-based modelling language: we define a disturbance with a rule whose *guard* and *body* define, respectively, functions *adm* and *dist* in $\mathcal{D}$. The labelling function is realised with a counter incremented each time a rule is fired (i.e., a disturbance is injected).

In order to enable a parallel approach to simulation, we partition the generated sequence of disturbance traces into $k \in \mathbb{N}^+$ subsequences (slices) of equal size. We will see in Section 5 that keeping together disturbance traces having a long common prefix enables better optimisation of simulation campaigns. This suggests to keep together traces generated consecutively by DFS. Thus, we first generate all n labelled disturbance traces into a single file and then we split such a file into k *slices* $\Delta^0_{\text{slice}}, \ldots, \Delta^{k-1}_{\text{slice}}$, by assigning the i-th trace ($0 \leq i < n$) to the $\lfloor \frac{ik}{n} \rfloor$-th slice.

5 Computation of Optimised Simulation Campaigns

Given a DG $\mathcal{D}$ and a sequence $\Delta_{\text{in}} = \delta_0, \ldots, \delta_{n-1}$ of labelled disturbance traces for $\mathcal{D}$, our simulation campaign optimiser (Fig. 2) computes a simulation campaign χ for *any* simulator $\mathcal{S}$ of *any* DES $\mathcal{H} = (S, s_0, d, O, \mathit{flow}, \mathit{jump}, \mathit{output})$ whose set of inputs $[0, d]$ is equal to the set of outputs of $\mathcal{D}$. The computed χ is *abstract* in that, for all commands of the form $\mathit{run}(e, t)$, 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 $\mathit{run}(e, t)$ commands by $\mathit{run}(e, t\tau)$.

The sequence of event lists $U(\chi_\tau)$ of χ_τ is *equal* (Theorem 2) to the sequence of event lists $u_\tau(\delta_0), \ldots, u_\tau(\delta_{n-1})$ associated to $\delta_0, \ldots, \delta_{n-1}$ with respect to time step τ. This implies that if the disturbance traces for a SLFV problem $(\mathcal{H}, \mathcal{D}, \tau, h)$ are split into $k \in \mathbb{N}^+$ slices $\Delta^0_{\text{slice}}, \ldots, \Delta^{k-1}_{\text{slice}}$, k instances of our optimiser can be used to *independently* compute k simulation campaigns, one for each slice. These can then be *independently* executed on k simulators. As the SUV $\mathcal{H}$ is an MDES (Definition 4), the answer to $(\mathcal{H}, \mathcal{D}, \tau, h)$ is *FAIL* iff the output of at least one simulator becomes 1 (Theorem 3). In that case, a *counterexample* can be derived from that simulator (Fig. 3).

We now outline our simulation campaign optimiser (Fig. 2). As the input sequence Δ_{in} 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_{in}, the optimiser builds a data structure called Labels Branching Tree (LBT) as completely as possible within the available RAM. Afterwards, it reads Δ_{in} again to produce the abstract simulation campaign from the LBT, ensuring that the number of states stored on simulator side (by means of *store* and *free* commands) is always within the simulator capabilities. RAM and simulator memory management follows precise *policies* discussed next.

LBT Construction. The LBT is a tree of labels rooted at l_0, the first label of all traces. The LBT collects *branching labels*, i.e., labels l_i for which there exist at least two disturbance traces $\delta = l_0, d_0, \ldots, l_i, d_i, \ldots, l_h$ and $\delta' = l_0, d_0, \ldots, l_i, d'_i, \ldots, l'_h$ in Δ_{in} 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 $\delta = l_0, d_0, \ldots, l_i, \ldots, l_j, \ldots, l_h \in \Delta_{\text{in}}$, no l_k in δ with $i < k < j$ is in the LBT (note: all such δ are identical at least up to l_j). For each label in the LBT, the number of the first and last trace in Δ_{in} where it occurs are kept. To recognise branching labels, the optimiser needs to maintain in RAM already seen labels not yet proven to be branching. Whenever the optimiser runs short of memory, it forgets some of these labels. As this would prevent or delay the recognition of further branching labels (leading to a smaller LBT and causing the computation of a less optimised simulation campaign), the optimiser first forgets the deepest labels (less likely to become branching later).

Computation of the Abstract Simulation Campaign. Once the LBT is built, the optimiser reads Δ_{in} a second time to compute the abstract simulation campaign χ, keeping track of which LBT labels are stored in simulator memory at any moment.

For each $\delta = l_0, d_0, \ldots, l_{\text{load}}, \ldots, l_h$ in Δ_{in}, let l_{load} be the right-most label in the LBT currently stored by the simulator. The optimiser appends to χ the following commands: (i) *load*(l_{load}); (ii) a command of the form *run*$(\hat{d}, t)$ for each maximal subsequence of length t in δ (starting from l_{load}) of the form $\hat{d}, l_{i_1}, 0, l_{i_2}, \ldots, 0, l_{i_t}\tilde{d}, \tilde{l}$ where either $\tilde{d} \neq 0$ or label $\tilde{l}$ needs to be stored. In the latter case, command *store*$(\tilde{l})$ is appended as well. Label $\tilde{l}$ needs to be stored if it is in the LBT, it will occur again in a later trace, and simulator memory is not full. If the latter requirement fails, the optimiser first needs to free-up simulator memory: it selects a label l_{free} to free among all those already stored and $\tilde{l}$ itself, and appends command *free*(l_{free}) to χ. Label l_{free} is chosen among those that will not occur in later traces. If none exists, then l_{free} is chosen as to minimise the simulation cost (number of steps) to drive the simulator to the state represented by l_{free}, starting from the state represented by its parent label in the LBT.

Theorem 2 shows that commands in the abstract simulation campaign χ computed by the optimiser on input Δ_{in} drive $\mathcal{S}$ as to correctly simulate the effects of Δ_{in} on $\mathcal{H}$.

Theorem 2. *Let $(\mathcal{H}, \mathcal{D}, \tau, h)$ be a SLFV problem, $\mathcal{S} = (\mathcal{H}, L, W, m)$ a simulator for $\mathcal{H}$, and $\Delta_{in} = \delta_0, \ldots, \delta_{n-1}$ be an ordered sequence of* some *labelled disturbance traces for $\mathcal{D}$, each of which being of the form $\delta_i = l_0, d_0, \ldots, l_{h-1}, d_{h_i-1}, l_{h_i}$ $(0 \leq i < n)$. Let $U_\tau(\Delta_{in})$ be the sequence $\mathsf{u}_\tau(\delta_0), \ldots, \mathsf{u}_\tau(\delta_{n-1})$ of event lists associated to the disturbance traces in Δ_{in} with respect to time step τ. The simulation campaign χ produced by the optimiser on input Δ_{in} is such that $U(\chi_\tau) = U_\tau(\Delta_{in})$ where χ_τ is the instantiation of χ with time step τ.*

Theorem 3 shows that if there exist disturbance traces in $\Delta^h_{\mathcal{D}}$ falsifying the property under verification, at least one of the generated campaigns returns a counterexample.

Theorem 3. *Let $(\mathcal{H}, \mathcal{D}, \tau, h)$ be a SLFV problem, $k \in \mathbb{N}^+$, $\mathcal{S}^0, \ldots, \mathcal{S}^{k-1}$ be k simulators for $\mathcal{H}$, and $\Delta^h_{\mathcal{D}}$ be a labelling of all disturbance traces for $\mathcal{D}$ of length h.*

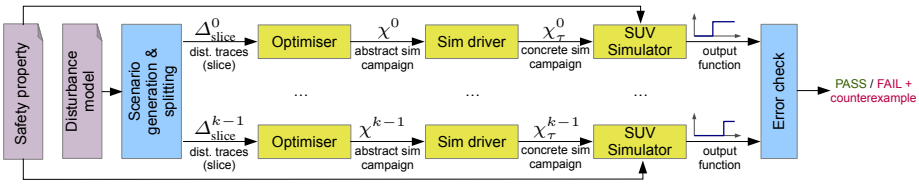

Fig. 3. Our overall approach

Let $\Delta^0_{slice}, \ldots, \Delta^{k-1}_{slice}$ be a partition of $\Delta^h_{\mathcal{D}}$ into k sequences. Let $\chi^0_\tau, \ldots, \chi^{k-1}_\tau$ be the instantiations with time step τ of the abstract simulation campaigns $\chi^0, \ldots, \chi^{k-1}$ computed by the optimiser on inputs $\Delta^0_{slice}, \ldots, \Delta^{k-1}_{slice}$.

The answer to $(\mathcal{H}, \mathcal{D}, \tau, h)$ is FAIL iff there exists $0 \leq j < k$ such that the state sequence of simulator $\mathcal{S}^j$ on χ^j_τ contains a state (s^*, u^*, M^*) such that output$(s^*) = 1$.

6 Experimental Results

In this section we present experimental results in order to evaluate the effectiveness of our SLFV approach summarised in Fig. 3.

From Fig. 3 we see that the disturbance model and thus disturbance generation and simulation campaign optimisation do not depend on the SUV model, which only affects the simulation time. For this reason we experiment with just one *large* SUV model. As our optimiser (Section 5) takes as input a set of disturbance traces (Fig. 3), disturbance models generating the same set of disturbance traces will yield the same results. For this reason we only focus on one disturbance model and change the number of the disturbance traces given as input to our optimiser in order to evaluate its performance.

Our optimiser computes an *abstract* simulation campaign χ. In order to execute χ, we need a *driver* that instantiates χ with a time step τ into χ_τ and translates χ_τ into commands for the target simulator. We implemented such a driver (*Sim driver* in Fig. 3) for the Simulink simulator and performed SLFV of the fuel control system in the Simulink distribution. This example has been studied in [31] using statistical model checking techniques. The fuel control system 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 model for the system along with a monitor for the property under verification. In our disturbance model, system sensors are subject to temporary faults, which are repaired after one second. In our setting, the complexity of the computation of an optimised simulation campaign does not depend on the SUV, it 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.

Experiments are performed on multiple 3.0 GHz, 8GB RAM Intel hyperthreaded Quad Core Linux PCs. Our time step τ (quantum between disturbances) is 1 second.

Table 1. Experimental results

h	time (h:m:s)	#traces	file size (MB)
50	0:1:35	448,105	195.725
60	0:3:29	805,075	420.743
70	0:6:35	1,314,145	799.584
80	0:11:41	2,002,315	1,390.157
90	0:21:34	2,896,585	2,259.642
100	0:28:39	4,023,955	3,484.489

(a) Disturbance trace generation

k	time (h:m:s)	slice size (MB)
2	0:0:14	1,742.244
4	0:0:14	871.122
8	0:0:15	435.561
16	0:0:14	217.78
32	0:0:14	108.89
64	0:0:13	54.445

(b) Instance $h = 100$ splitting

		LBT	$m = 1$		$m =$ 100,000		
k	#traces	size	time	#cmds	time	#cmds	%opt
2	2,011,977	670,661	0:3:14	16,040,520	3:47:57	8,047,912	**79.42%**
4	1,005,988	335,331	0:2:28	8,012,662	1:45:04	4,023,955	**83.32%**
8	502,994	167,666	0:0:35	4,001,378	0:44:27	2,011,978	**86.49%**
16	251,497	83,834	0:0:18	1,997,486	0:16:24	1,005,991	**88.97%**
32	125,748	41,918	0:0:07	996,660	0:4:50	502,996	**90.87%**
64	62,874	20,959	0:0:03	496,906	0:0:51	251,497	**92.47%**

(c) Simulation campaign optimisation ($h = 100$, time in h:m:s)

k	$m = 1$ time	$m =$ 100,000 time	speedup
8	n/a	29, 13:50:12	$>$ **1.7**×
16	n/a	14, 6:39:09	$>$ **3.5**×
32	25, 23:07:43	6, 22:32:25	**3.8**×
64	12, 22:58:16	3, 9:19:18	**3.8**×

(d) Simulation (time in days, h:m:s)
'n/a' Simulation aborted after 50 days

	offline				online		
k	gener.	split.	optimis.	total	simulation	%offline	%online
8	0:28:39	0:0:15	0:44:27	1:13:21	29, 13:50:12	**0.17%**	99.83%
16	0:28:39	0:0:14	0:16:24	0:45:17	14, 6:39:09	**0.22%**	99.78%
32	0:28:39	0:0:14	0:4:50	0:33:43	6, 22:32:25	**0.34%**	99.66%
64	0:28:39	0:0:13	0:0:51	0:29:43	3, 9:19:18	**0.31%**	99.69%

(e) Offline vs. online phase (time in days, h:m:s)

Automatic Generation of Exhaustive Simulation Scenarios. Table 1a shows the time needed by our disturbance generator (Section 4) to generate disturbance traces with different horizons (column h). Our experiments show that the generation of even millions of traces is done in a matter of minutes. In the following, we focus on experiment $h = 100$ in Table 1a, since it has the largest sequence of disturbance traces (4,023,955).

Table 1b shows the time needed to split (Section 4) the sequence of disturbance traces of experiment $h = 100$ in Table 1a to enable parallel computation of simulation campaigns, with different degrees of parallelism (column k). Our experiments show that such tasks take just a few seconds.

Computation of Optimised Simulation Campaigns. Table 1c shows the performance of our optimiser (Section 5) when computing a simulation campaign from a slice of disturbance trace sequence (one k-th of the disturbance traces of instance $h = 100$). The table shows the number of traces of each slice, the size of the LBT, the time to compute the simulation campaign as well as the number of commands it consists of in two scenarios: columns $m = 1$ refer to computations of *unoptimised* simulation campaigns (as only the initial state can be stored in the simulator), while those for $m =$100,000 refer to *optimised* campaigns for a simulator with about 15GB of disk

space available (as, in our case study, each stored state takes about 150KB). Column *%opt* shows how much the simulation campaigns for m =100,000 are optimised with respect to the case with $m = 1$. Namely, *%opt* is the average value of L_l/h, where L_l is the number (at most h) of simulation time steps we save thanks to command *load*(l).

Execution of the Simulation Campaigns. Table 1d shows the time needed to execute our simulation campaigns on Simulink. For each row (degree of parallelism k) we report the maximum of the time needed by Simulink to execute the k simulation campaigns forming the verification task for $m = 1$ and $m = 100{,}000$.

The parallelism enabled by our approach is essential to handle large simulation campaigns as those considered here. In fact, for $k < 8$, we could not complete the simulation in 50 days (while we can easily compute the simulation campaign, Table 1c).

Summing Up. Table 1e sums up our results by showing the total time spent offline computing the optimised simulation campaign (column *total*), the time spent online by executing the simulation campaign (column *online simulation*) and the (percentage of the) time spent in the offline [online] computation (column *%offline*) [(column *%online*)]. We can see that our offline computations account for less than 0.5% of the total simulation time and, most importantly, enable exhaustive parallel HILS almost 4 times faster than without optimisation (Table 1d).

7 Conclusions

We have presented a HILS based approach to SLFV. We use explicit model checking techniques to model, generate and optimise exhaustive simulation scenarios for parallel HILS. This enables *black box* SLFV of *actual* systems. We have shown the effectiveness of our approach by applying it to a large control system case study in the Simulink distribution. To the best of our knowledge, this is the first time that exhaustive HILS has been carried out on a set of simulation scenarios (disturbance traces) of the size considered here (about 4 millions). Our experimental results show that we spend more than 99% of the SLFV time in the simulation activity. Thus, investigation of guided search techniques, for example as in [7], is a promising future work in our setting.

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

References

1. Alur, R.: Formal verification of hybrid systems. In: Proc. EMSOFT 2011. ACM (2011)
2. Barnat, J., Brim, L., Černá, I., Moravec, P., Ročkai, P., Šimeček, P.: DiVINE – A tool for distributed verification. In: Ball, T., Jones, R.B. (eds.) CAV 2006. LNCS, vol. 4144, pp. 278–281. Springer, Heidelberg (2006)
3. Bingham, B., Bingham, J., De Paula, F.M., Erickson, J., Singh, G., Reitblatt, M.: Industrial strength distributed explicit state model checking. In: Proc. PDMC-HIBI 2010. IEEE (2010)

4. Brillout, A., He, N., Mazzucchi, M., Kroening, D., Purandare, M., Rümmer, P., Weissenbacher, G.: Mutation-based test case generation for simulink models. In: de Boer, F.S., Bonsangue, M.M., Hallerstede, S., Leuschel, M. (eds.) FMCO 2009. LNCS, vol. 6286, pp. 208–227. Springer, Heidelberg (2010)
5. Broy, M., Jonsson, B., Katoen, J.-P., Leucker, M., Pretschner, A. (eds.): Model-Based Testing of Reactive Systems. LNCS, vol. 3472. Springer, Heidelberg (2005)
6. Cavaliere, F., Mari, F., Melatti, I., Minei, G., Salvo, I., Tronci, E., Verzino, G., Yushtein, Y.: Model checking satellite operational procedures. In: Proc. DASIA 2011 (2011)
7. De Paula, F.M., Hu, A.J.: An effective guidance strategy for abstraction-guided simulation. In: Proc. DAC 2007, pp. 63–68. ACM (2007)
8. Dean, J., Ghemawat, S.: Mapreduce: simplified data processing on large clusters. In: Proc. OSDI 2004. USENIX Association (2004)
9. Della Penna, G., Intrigila, B., Melatti, I., Tronci, E., Zilli, M.: Exploiting transition locality in automatic verification of finite state concurrent systems. STTT 6(4), 320–341 (2004)
10. Dill, D.L., Drexler, A.J., Hu, A.J., Han Yang, C.: Protocol verification as a hardware design aid. In: Proc. IEEE Int. Conf. Comp. Design on VLSI in Comp. & Proc., 1991. IEEE (1992)
11. Gadkari, A.A., Yeolekar, A., Suresh, J., Ramesh, S., Mohalik, S., Shashidhar, K.C.: AutoMOTGen: Automatic model oriented test generator for embedded control systems. In: Gupta, A., Malik, S. (eds.) CAV 2008. LNCS, vol. 5123, pp. 204–208. Springer, Heidelberg (2008)
12. Grosu, R., Smolka, S.A.: Monte carlo model checking. In: Halbwachs, N., Zuck, L.D. (eds.) TACAS 2005. LNCS, vol. 3440, pp. 271–286. Springer, Heidelberg (2005)
13. Ho, P.H., Shiple, T., Harer, K., Kukula, J., Damiano, R., Bertacco, V., Taylor, J., Long, J.: Smart simulation using collaborative formal and simulation engines. In: Proc. ICCAD 2000 (2000)
14. Holzmann, G.J.: The SPIN model checker. Addison-Wesley (2003)
15. Holzmann, G.J.: Parallelizing the spin model checker. In: Donaldson, A., Parker, D. (eds.) SPIN 2012. LNCS, vol. 7385, pp. 155–171. Springer, Heidelberg (2012)
16. Holzmann, G.J., Joshi, R., Groce, A.: Model driven code checking. Autom. Softw. Eng. 15(3-4), 283–297 (2008)
17. Kanade, A., Alur, R., Ivančić, F., Ramesh, S., Sankaranarayanan, S., Shashidhar, K.C.: Generating and analyzing symbolic traces of simulink/Stateflow models. In: Bouajjani, A., Maler, O. (eds.) CAV 2009. LNCS, vol. 5643, pp. 430–445. Springer, Heidelberg (2009)
18. Maler, O., Nickovic, D.: Monitoring temporal properties of continuous signals. In: Lakhnech, Y., Yovine, S. (eds.) FORMATS 2004 and FTRTFT 2004. LNCS, vol. 3253, pp. 152–166. Springer, Heidelberg (2004)
19. Meenakshi, B., Bhatnagar, A., Roy, S.: Tool for translating simulink models into input language of a model checker. In: Liu, Z., Kleinberg, R.D. (eds.) ICFEM 2006. LNCS, vol. 4260, pp. 606–620. Springer, Heidelberg (2006)
20. Melatti, I., Palmer, R., Sawaya, G., Yang, Y., Kirby, R.M., Gopalakrishnan, G.: Parallel and distributed model checking in eddy. Int. J. Softw. Tools Technol. Transf. 11(1), 13–25 (2009)
21. Nanshi, K., Somenzi, F.: Guiding simulation with increasingly refined abstract traces. In: Proc. DAC 2006, pp. 737–742. ACM (2006)
22. Rozier, K.Y., Vardi, M.Y.: Deterministic compilation of temporal safety properties in explicit state model checking. In: Proc. HVC 2012. Springer (2012)
23. Sen, K., Viswanathan, M., Agha, G.: On statistical model checking of stochastic systems. In: Etessami, K., Rajamani, S.K. (eds.) CAV 2005. LNCS, vol. 3576, pp. 266–280. Springer, Heidelberg (2005)
24. Sontag, E.D.: Mathematical Control Theory: Deterministic Finite Dimensional Systems. Texts in Applied Mathematics. Springer (1998)
25. Stern, U., Dill, D.L.: Parallelizing the Murphi Verifier. Form. Methods Syst. Des. 18(2), 117–129 (2001)

26. Tripakis, S., Sofronis, C., Caspi, P., Curic, A.: Translating discrete-time Simulink to Lustre. ACM Trans. Emb. Comp. Syst. 4(4), 779–818 (2005)
27. Tronci, E., Della Penna, G., Intrigila, B., Zilli, M.: A probabilistic approach to automatic verification of concurrent systems. In: Proc. APSEC 2001, pp. 317–324. IEEE (2001)
28. Venkatesh, R., Shrotri, U., Darke, P., Bokil, P.: Test generation for large automotive models. In: Proc. ICIT 2012, pp. 662–667. IEEE (2012)
29. Whalen, M., Cofer, D., Miller, S., Krogh, B.H., Storm, W.: Integration of formal analysis into a model-based software development process. In: Leue, S., Merino, P. (eds.) FMICS 2007. LNCS, vol. 4916, pp. 68–84. Springer, Heidelberg (2008)
30. Yang, C.H., Dill, D.L.: Validation with guided search of the state space. In: Proc. DAC 1998, pp. 599–604. ACM (1998)
31. Zuliani, P., Platzer, A., Clarke, E.M.: Bayesian statistical model checking with application to simulink/stateflow verification. In: Proc. HSCC 2010, pp. 243–252 (2010)