Automated Verification of Cyber-Physical Systems

A.A. 2022/2023

Corso di Laurea Magistrale in Informatica

System Level Formal Verification

Igor Melatti

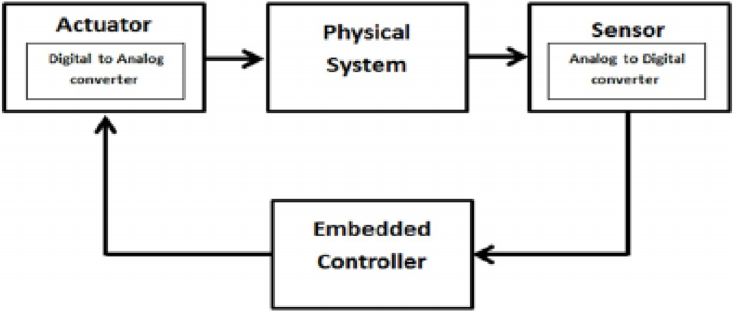
Università degli Studi dell'Aquila

Dipartimento di Ingegneria e Scienze dell'Informazione e Matematica

Embedded Systems

- One of the task given to computers from the very start: monitoring and/or controlling some external system
  - where the "system" is anything without computational capabilities
  - 60s: guidance of missiles and Apollo Guidance System
- In the following, we will restrict our attention to control
- Thus, an embedded system is mainly composed by two parts: a controller and a plant
  - the plant must accept inputs able to modify its behaviour
  - the plant must also expose some output
- Nowadays, embedded systems are everywhere
  - may control something very little, like an electrical circuit (e.g., buck DC/DC converter)
  - or something very big, like an automobile or an aircraft

Embedded Systems: Closed-Loop System



System Level Formal Verification

- System level verification has the aim to discover errors to some (embedded) system considered as a whole
  - all components are considered together
  - we assume they have been separately tested before
- Typically done by testing
  - plant is nearly always replaced by a *simulator*
  - often built in Simulink or Modelica
  - HILS: Hardware-in-the-loop simulation
- System level formal verification: we want to apply Model Checking techniques

System Level Formal Verification

- In "standard" Model Checking, we are given
  - a non-deterministic Kripke Structure (KS)
  - an LTL or CTL property to be verified
- We get a PASS/FAIL response
  - possibly with a counterexample
- When we deal with complex embedded systems, having a KS is difficult
  - moreover: most plants are described by *real* variables, thus they have an infinite number of states
  - approximation may be ok for early verification, but here we want system level verification
  - with actual software involved

System Level Formal Verification

- Thus, we want to apply Model Checking to the closed-loop system (SUV, System Under Verification) as:
  - a black-box controller
  - a simulator for the plant
- We are still interested in some property to be verified
  - let us suppose we have a safety property for starting
- How to accomplish such a task?
- The idea is: kind of Statistical Model Checking, but *exhaustive*
  - that is: perform simulations of the whole system (like in HILS) considering all possible scenarios

System Level Formal Verification

- This should be impractible, how can we do this?
- The idea is: if we see the system as a black-box, verification is about
  - (incontrollable) interactions with the external environment
  - (incontrollable) "hardware" (i.e., parts of the plant) failures
  - (incontrollable) changes in the plant simulation parameters
- Interactions between the plant and the controller are inside the system
  - as a consequence of the variations listed above
- We can see all of this as *inputs* to our closed-loop system
- A system is *not* expected to withstand *any* combination of the preceding
  - e.g., if we put an airplane inside a violent windshire, we cannot expect its controller to safely land it

System Level Formal Verification: Requirements

- Requirement 1: we can write a model for the meaningful interactions between the system and the environment
  - “meaningful”: those we want to verify
- In the following, we will call such interactions as *disturbances*
  - because they are deviations from the current behaviour
  - e.g., if we move an inverted pendulum while it is upright and still, we are disturbing it
  - causing its controller to react and return it upright and still
- As in Statistical Model Checking, we consider a bounded verification
  - thus, we are interested in *finite sequences* of possible disturbances
  - e.g., move the inverted pendulum, then move it again before it is returned upright

System Level Formal Verification: Requirements

- Requirement 2: the simulator for the plant accepts the following commands
  - **I** *d*: inject disturbance *d*
    - will modify the plant behaviour
    - that is, the following **R** commands
  - **R** *t*: compute the evolution of the plant within *t* units of time
    - this is the main function for all simulators...
  - **S** *l* save the current simulator state with id *l*
  - **F** *l* free the simulator state with id *l*
  - **L** *l* load (i.e., restore) the simulator state with id *l*
    - simulator states are saved in some permanent memory, e.g., files on disk
    - S_1, **S** *l*, S_2, **L** *l*, S_3, where S_i are command sequences, is equivalent to the command sequences S_1, S_2 (restart) S_1, S_3

System Level Formal Verification: Requirements

- A sequence **R** t_1, **S** l, **R** t_2, **L** l, **R** t_3 is equivalent to the following *two* simulations: **R** $t_1 + t_2$ and **R** $t_1 + t_3$
  - in the middle, the system simulation is restarted from time 0
- A sequence **I** d, **R** t is equivalent to:
  - modify the simulator by changing some plant parameters
    - each disturbance corresponds to a modification of a selection of plant parameters
    - "modification": change the value
  - run a simulation for t units of time with the new plant model
- A sequence **R** t_1, **I** d, **R** t_2 is equivalent to:
  - modify the simulator so that the d parameters changing happens after t_1 units of time
    - e.g., in Modelica, this could be done with an `if` inside the main `whensample`, if any
  - run a simulation for $t_1 + t_2$ units of time

System Level Formal Verification: Requirements

- A sequence **R** t_1, **I** d, **S** l, **R** t_2, **L** l, **R** t_3 is equivalent to:
  - modify the simulator for d after t_1 units of time
  - perform simulations **R** $t_1 + t_2$ and **R** $t_1 + t_3$
- A sequence **R** t_1, **S** l, **R** t_2, **I** d_1, **R** t_3, **L** l, **I** d_2, **R** t_4 is equivalent to
  - modify the simulator for d_1 after $t_1 + t_2$ units of time
  - modify the simulator for d_2 after t_1 units of time
  - perform simulations **R** $t_1 + t_2 + t_3$ and **R** $t_1 + t_4$
  - is this correct????

System Level Formal Verification: Requirements

- *Simulation campaign*: any *finite* sequence of simulator commands
  - finite because we are performing bounded verification
- We assume that we can write some software which takes as input a simulation campaign and executes it on the simulator
  - we call it *driver*
  - either within the simulator or with some external script
  - e.g.: in Simulink, we may use Simulink scripts
  - e.g.: in Modelica, we have to use something external
  - we can write model-independent Simulink and Modelica drivers

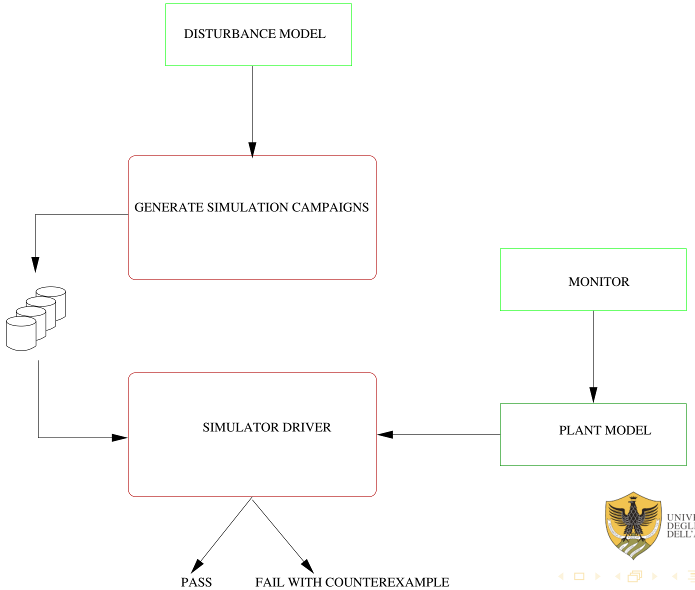
System Level Formal Verification: Modeling

- Thus, we need two models:
  - disturbance model
  - plant model
- Plus the actual software for the controller
  - which directly interacts with the plant model
  - e.g., using external functions, available both in Modelica an Simulink
  - in the following, we will consider it embedded in the plant model

System Level Formal Verification: Modeling

- In embedded systems design a simulation model for the plant is always built
- Thus, the only modeling required is that of the disturbance model
  - we are performing a kind of exhaustive functional testing
  - exhaustive w.r.t. the given disturbance model
- We also need to enlarge the existing plant model with a *monitor*
  - when an error is found, a boolean variable will become one
  - equivalent to specify a bounded safety property

System Level Formal Verification: Architecture



UNIVERSITÀ DEGLI STUDI DELL'AQUILA

DISIM Dipartimento di Ingegneria e Scienze dell'Informazione e Matematica

- Let $d \in \mathbb{N}^+$ be a positive integer
  - total number of disturbances is $d + 1$
  - 0 is a special value for "no disturbance"
- A *discrete event sequence* is a function $u : \mathbb{R}^{\geq 0} \to [0, d] \cap \mathbb{N}$ s.t., for all $t \in \mathbb{R}^{\geq 0}$, $\mathrm{card}(\{\tilde{t} \mid 0 \leq \tilde{t} \leq t \wedge d(\tilde{t}) \neq 0\}) < \infty$
  - that is: given a time t, $u(t)$ returns the disturbance at time t
  - thus, we are requiring that it is almost always without disturbances
  - i.e., some disturbance happens only in a finite number of times
- Let $\mathcal{U}_d = \{u \mid u \text{ is a discrete event sequence for } d\}$

System Level Formal Verification: Definitions

- An *event list* is a sequence $(u_0, \tau_0), (u_1, \tau_1), \ldots$ s.t., for all $i \geq 0$, $u_i \in [0, d] \cap \mathbb{N}, \tau_i \in \mathbb{R}^{\geq 0}$
  - not only disturbances, but also their durations
- For each event list there is a unique discrete event sequence u defined as:
  - $u(0) = u_0$
  - $u(t) = u_h$ if $t = \sum_{i=0}^{h-1} \tau_i$ for some $h \geq 1$
  - $u(t) = 0$ otherwise
- The viceversa also holds (derive the formula by yourself)

System Level Formal Verification: Definitions

- A *Discrete Event System* (DES) is a tuple $\mathcal{H} = \langle S, s_0, d, O, \text{flow}, \text{jump}, \text{output} \rangle$ where:
  - S is a (possibly infinite) set of states; $s_0 \in S$ is the initial state
    - Cartesian product of the domains of the state variables
  - d is the number of disturbances (defines the *input space* $\mathcal{U}_d$)
  - O is a (possibly infinite) set of output values
    - useful to define the monitor
  - $\text{output} : S \to O$, i.e., each state defines an output
  - $\text{flow} : S \times \mathbb{R}^{\geq 0} \to S$
    - dynamics without disturbances: $\text{flow}(s, t)$ is the state reached after t units of time, starting from state s
    - w.r.t. hybrid systems, this may also result in location changes!
    - $\text{flow}(s, 0) = s$
  - $\text{jump} : S \times [0, d] \to S$
    - dynamics with disturbances: $\text{jump}(s, d)$ is the state reached when disturbance d is applied in state s
    - $\text{jump}(s, 0) = s$

System Level Formal Verification: Definitions

- The *state function* of a DES tells us in which state we go after some simulation time
  - starting from s_0 and considering intervening disturbances in a discrete even sequence
  - our DES are *deterministic*, thus there is only one such state
- Given a DES $\mathcal{H} = \langle S, s_0, d, O, \mathrm{flow}, \mathrm{jump}, \mathrm{output} \rangle$, the *state function* of $\mathcal{H}$ is $\phi : \mathcal{U}_d \to S$ s.t.:
  - $\phi(u, 0) = \mathrm{jump}(s_0, u(0))$
    - i.e., if there is some disturbance at time 0, let us begin from the resulting state
    - otherwise, we begin from s_0
  - for each $t > 0$, $\phi(u, t) = \mathrm{jump}(\mathrm{flow}(\phi(u, t^*), t - t^*), u(t))$
    - $t^* = \max\{\tilde{t} \mid \tilde{t} < t \wedge u(\tilde{t}) \neq 0\}$
    - with $\max \varnothing = 0$

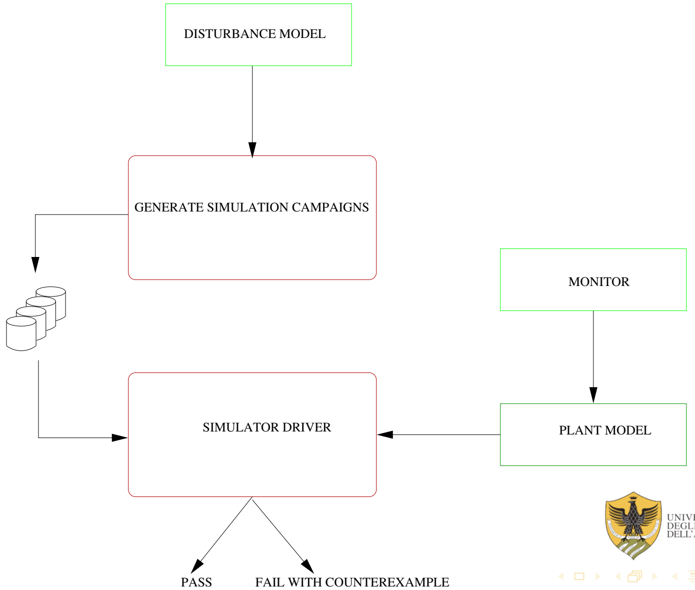
System Level Formal Verification: Definitions

- We may view the *state function* in a more computation-like way
- Given a DES $\mathcal{H} = \langle S, s_0, d, O, \mathrm{flow}, \mathrm{jump}, \mathrm{output} \rangle$, a discrete event sequence u and a time t:
  1. compute the (minimal) event list $(u_0, \tau_0), (u_1, \tau_1), \ldots, (u_n, \tau_n)$ corresponding to u
     - must be finite by definition of discrete event sequence
  2. with $s = s_0$ as initialization, for $i = 0, \ldots, n$:
     1. let s be $\mathrm{jump}(s, u_i)$
     2. let s be $\mathrm{flow}(s, \tau_i)$
  3. output s

System Level Formal Verification: Definitions

- We also need the *output function* of a DES $\mathcal{H} = \langle S, s_0, d, O, \text{flow}, \text{jump}, \text{output} \rangle$
  - easy when we have the state function
- Namely, $\psi : \mathcal{U}_d \times \mathbb{R}^{\geq 0} \to O$ is defined as $\psi(u, t) = \text{output}(\phi(u, t))$
- Monitor: when the safety property becomes false, the output is false
  - this is the only output we need
  - once is false, it must stay false, otherwise we may not realize it
- A *monitored DES* is a tuple $\mathcal{H} = \langle S, s_0, d, \text{flow}, \text{jump}, \text{output} \rangle$ s.t.
  - $\langle S, s_0, d, \{0, 1\}, \text{flow}, \text{jump}, \text{output} \rangle$ is a DES
  - for all $u \in \mathcal{U}_d$, $\psi(u, t)$ is non-increasing w.r.t. t

System Level Formal Verification: Architecture



Modeling the Disturbances

- The system part is now ok: a Monitored DES encompasses the closed-loop system and the property monitor
  - let us go with the disturbance model
- The "Generate simulation campaign" part is divided in two parts
  - from a model of disturbances, generate all possible sequences of disturbances (*disturbance traces*) of length T
  - from sequences of disturbances, generate the optimized simulation campaigns
- Thus, we need some model able to define complex disturbance traces
  - e.g.: in a given trace, d_1 only occurs at most three times but never immediately after d_2

Modeling the Disturbances

- One possible way is using a standard Model Checker
- Here, we will use CMurphi: each rule corresponds to a disturbance
  - by suitably using rule guards, we may implement any wanted logic behind disturbance traces
  - see attached example
- By suitably modifying the CMurphi source code, we may generate disturbance traces as required
- Also a slight modification to the input language is required to introduce *final states*

Modeling the Disturbances: Definitions

- A *disturbance generator* (DG) is a tuple $\mathcal{D} = \langle Z, d, \mathrm{dist}, \mathrm{adm}, Z_I, Z_F \rangle$ where:
  - Z is a finite set of states
  - $Z_I, Z_F \subset Z$ are the subsets of initial and final states
  - $d \in \mathbb{N}^+$ is again the number of disturbances
  - $\mathrm{adm} : Z \times [0, d] \cap \mathbb{N} \to \{0, 1\}$ defines the disturbances admitted at a given state
  - $\mathrm{dist} : Z \times [0, d] \cap \mathbb{N} \to Z$ defines the deterministic transition relation
    - but CMurphi was nondeterministic!
    - yes, but here we are adding the disturbance, i.e., the rule getting fired...
- Easy to show that this is equivalent to a Kripke Structure

Modeling the Disturbances: Definitions

- Let $\mathcal{D} = \langle Z, d, \mathrm{dist}, \mathrm{adm}, Z_I, Z_F \rangle$ be a DG
- A *disturbance path* of length h for $\mathcal{D}$ is a sequence $z_0 d_0 \ldots z_{h-1} d_{h-1} z_h$ where:
  - $z_0 \in Z_I, z_h \in Z_F$: we start from an initial and end in a final state
  - $\forall i = 0, \ldots, h-1.\ \mathrm{adm}(z_i, d_i) = 1$
  - $\forall i = 0, \ldots, h-1.\ \mathrm{dist}(z_i, d_i) = z_{i+1}$
    - the DG semantics is preserved
- A *disturbance trace* is a sequence $\delta = d_0 \ldots d_{h-1}$ s.t. there exists a disturbance path $z_0 d_0 \ldots z_{h-1} d_{h-1} z_h$ for $\mathcal{D}$
- We define $\Delta^h_{\mathcal{D}} = \{\delta \mid \delta \text{ is a disturbance trace for } \mathcal{D} \wedge |\delta| = h\}$

UNIVERSITÀ DEGLI STUDI DELL'AQUILA

DISIM Dipartimento di Ingegneria e Scienze dell'Informazione e Matematica

System Level Formal Verification Problem

- We can now formally define the overall problem we want to verify
  - for standard model checking it was: you have a Kripke Structure and a property, tell me if the property holds
  - with suitably defined semantics for the property holding on a Kripke Structure
- Here things are slightly more complicated: we also need a *time step* τ
  - not very strange: also simulators use some simulator step to perform simulations
- τ allows us to go from disturbance traces to event lists (and discrete event sequences)
  - from $\delta = d_0, \ldots d_{h-1}$ to $(d_0, \tau) \ldots (d_{h-1}, \tau)$
  - we denote with $u(\delta)$ the discrete event sequence of δ

UNIVERSITÀ DEGLI STUDI DELL'AQUILA
DISIM Dipartimento di Ingegneria e Scienze dell'Informazione e Matematica

System Level Formal Verification Problem

- Given an MDES $\mathcal{H}$ and a DG $\mathcal{D}$, a *System Level Formal Verification Problem* (SLFVP) is a tuple $\mathcal{P} = \langle \mathcal{H}, \mathcal{D}, \tau, h \rangle$ where
  - $\tau \in \mathbb{R}^+, h \in \mathbb{N}^+$
  - d is the same both in $\mathcal{H}$ and in $\mathcal{D}$
- Let ψ be the output function for $\mathcal{H}$, then the *answer* to $\mathcal{P}$ is
  - $\langle \mathrm{FAIL}, \delta \rangle$ if $\delta \in \Delta_{\mathcal{D}}^{h}$ is s.t. $\psi(u_\tau(\delta), \tau h) = 0$
  - PASS if such a $\delta \in \Delta_{\mathcal{D}}^{h}$ does not exist

System Level Formal Verification Problem

- Two main assumptions:
  - disturbances cannot happen at any time, but only at multiple times of τ
  - disturbances traces are of length h
    - which implies that the total simulation time is $T = h\tau$
- The larger h and smaller τ, the closest we are to reality
  - as for h, it is the same of Bounded Model Checking and Statistical Model Checking
- 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

System Level Formal Verification: Algorithms

- To simulate a MDES, we rely on existing simulators
  - Simulink, Modelica, NGSpice...
- As for the "Generate simulation campaign", is divided in two parts
  - from a model of disturbances, generate all *disturbance traces* of length h
  - from sequences of disturbances, generate the optimized simulation campaigns
- Let us see how this is implemented

Generating all Disturbance Traces: Algorithm

function $generateByDFS(\mathcal{D}, T)$:

$S_Z \leftarrow \emptyset$, $S_D \leftarrow \emptyset$, $DistTraces \leftarrow \emptyset$, $c \leftarrow 1$
$Push(S_Z, z_0)$, $Push(S_D, 1)$, $\delta_0 \leftarrow c$, $c \leftarrow c + 1$
while $StackIsNotEmpty(S_Z)$ **do**
  $z \leftarrow Top(S_Z)$, $\tilde{d} \leftarrow Top(S_D)$
  if $\tilde{d} \leq d$ **then**
    $Top(S_D) \leftarrow \tilde{d} + 1$
    if $adm(z, \tilde{d})$ **then**
      $\delta_{|S_Z|} \leftarrow (\tilde{d}, c)$, $c \leftarrow c + 1$
      if $|S_Z| \leq T$ **then**
        $Push(S_Z, dist(z, \tilde{d}))$, $Push(S_D, 1)$
      else
        if $z \in Z_F$ **then** $DistTraces \leftarrow DistTraces \cup \delta$
  else
    $Pop(S_Z)$, $Pop(S_D)$
return $DistTraces$

Generating all Disturbance Traces: Algorithm

- This is for one initial state only, easy to generalize
- Standard non-recursive DFS
  - two stacks, one for states, one for rules
- Main difference 1: no check for already visited states
  - we are interested in transitions, so states may and must be visited multiple times
  - the bound T guarantees termination
- Main difference 2: the disturbance traces also encompass *labels*
  - simply a growing integer c
- Will be used by the simulation campaign generator

Generating Simulation Campaigns: Definitions

- A *DES Simulator* is a tuple $\mathcal{S} = \langle \mathcal{H}, L, W, m \rangle$ where:
  - $\mathcal{H} = \langle S, s_0, d, O, \mathrm{flow}, \mathrm{jump}, \mathrm{output} \rangle$ is a DES
  - L is a set of labels
  - $m \in \mathbb{N}^+$ is the maximum number of states the simulator can store
  - W is a set of simulator states s.t., for all $w \in W$, $w = (s, u, M)$ and:
    - $s \in S \cup \perp$ (a DES state or a sink state)
    - $u \in \mathcal{U}_d$ (an event list)
    - $M \subseteq L \times S \times \mathcal{U}_d$ s.t., for each $l \in L$, there exist at most one triple $(l, s, u) \in M$
    - $|M| \leq m$
  - the DES simulator initial state is $(s_0, \varnothing, \varnothing)$

Generating Simulation Campaigns: Definitions

- The dynamics of a DES Simulator is simulator is defined on the basis of simulation campaign commands
- That is, we need to define $\mathrm{sim}_{\mathcal{S}} : W \times C \to W$
- Where C is the set of the following commands:
  - $\mathrm{load}(l)$ for $l \in L$
  - $\mathrm{store}(l)$ for $l \in L$
  - $\mathrm{free}(l)$ for $l \in L$
  - $\mathrm{run}(t)$ for $t \in \mathbb{N}^+$
  - $\mathrm{inject}(\tilde{d})$ for $\tilde{d} \in [0, d] \cap \mathbb{N}$
- Thus, we define $\mathrm{sim}_{\mathcal{S}}$ by cases

UNIVERSITÀ DEGLI STUDI DELL'AQUILA

DISIM Dipartimento di Ingegneria e Scienze dell'Informazione e Matematica

Generating Simulation Campaigns: Definitions

- $\mathrm{sim}_{\mathcal{S}}(s, u, M, \mathrm{load}(l)) = (s', u', M)$, being $(l, s', u') \in M$
- $\mathrm{sim}_{\mathcal{S}}(s, u, M, \mathrm{free}(l)) = (s, u, M \setminus \{(l, s', u')\})$
- $\mathrm{sim}_{\mathcal{S}}(s, u, M, \mathrm{store}(l)) = (s, u, M \cup \{(l, s, u)\})$ if $|M| < m$
- $\mathrm{sim}_{\mathcal{S}}(s, u, M, \mathrm{run}(t)) = (\mathrm{flow}(s, t\tau), u \cdot (0, t), M)$
- $\mathrm{sim}_{\mathcal{S}}(s, u, M, \mathrm{inject}(\tilde{d})) = (\mathrm{jump}(s, \tilde{d}), u \cdot (\tilde{d}, 0), M)$
- Plus error checking, not considered here
  - e.g., trying to free something which was not stored
  - e.g., trying to store when memory is already full
  - e.g., trying to store without freeing first (if already present)

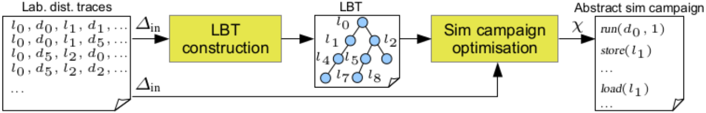
Generating Simulation Campaigns: Definitions

- A simulation campaign is a sequence $\chi = c_0(a_0) \ldots c_k(a_k)$ of commands as above
  - note that k and h are independent
- A χ identifies a sequence $w_0, \ldots, w_k$ s.t., for all $i = 0, \ldots, k-1$, $\mathrm{sim}_{\mathcal{S}}(w_i, c_i(a_i)) = w_{i+1}$ and $w_i = (s_i, u_i, M_i)$
  - by construction, u_i leads from s_0 to s_i
- This also defines the *output sequence* $\mathrm{output}(s_0) \ldots \mathrm{output}(s_k)$
- Less strightforward: the *event list sequence* associated to χ
  - watch out: a sequence of lists...
  - $U(\chi) = u_{j_1}, \ldots, u_{j_\ell}, u_k$ where ℓ is the number of load commands in χ
  - for $r = 1, \ldots, \ell$, j_r is the index of the r-th load command in χ

Generating Simulation Campaigns: Definitions

- Let $d \in \mathbb{N}^+$ and L be countably infinite set of labels. A *labelling* is an injective $\lambda : ([0, d] \cap \mathbb{N})^* \to L$
  - from finite sequence of integers to labels
- The labelling of a disturbance trace $\delta = d_0 \dots d_{h-1}$ is $\lambda(\delta) = l_0 d_0, \dots, h_{h-1} d_{h-1} l_h$
  - for all $i = 0, \dots, h$, $l_i = \lambda(d_0, \dots, d_{i-1})$
- Thus, the algorithm for disturbance traces given above returns labelled disturbance traces
- Let us go with the simulation campaign generation

Generating Simulation Campaigns: Idea



Generating Simulation Campaigns: Algorithm

- A *Labels Branching Tree* (LBT) is a DAG where nodes are labels
- There is an edge (l, l') iff $\exists \delta, \delta' \in \Delta_{\mathrm{in}}$ s.t.
  - $\delta = l_0, d_0, \ldots, d_{h-1} l_h$, $\delta' = l'_0, d'_0, \ldots, d'_{h-1} l'_h$
  - $\exists i = 0, \ldots, h-1 : \; d_i \neq d'_i \wedge \forall j = 0, \ldots, i-1. \; l_j = l'_j \wedge d_j = d'_j$
  - $l = l_i, l' = l'_i$
  - that is, if there are two traces which differs by (l, l') for the first time, l, l' will be siblings in the LBT
- Branching labels represent simulator states whose storing may save simulation time (by loading them back later)
- The LBT generation keeps into account that memory to store states is limited by m
  - thus, the result is optimal only for at most m states stored

Generating Simulation Campaigns: Algorithm

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

Generating Simulation Campaigns: Algorithm

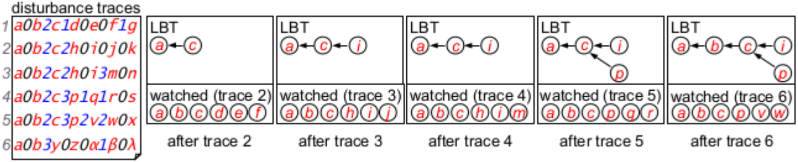
- Given the LBT $\mathcal{L}$, the output simulation campaign χ is computed by scanning again Δ_{in}
- For $\delta = l_0, d_0, \ldots, d_{h-1} l_h \in \Delta\mathrm{in}$, let r be the higher (i.e., rightmost) index s.t. l_r is in some already generated load command and is in the LBT
- Append to χ first $\mathrm{load}(l_r)$ and then one of the following:
  - $\mathrm{inject}(\tilde{d}), \mathrm{run}(t)$ where:
    - in δ there is a subsequence $l_r \tilde{d} l_{r+1} 0 \ldots 0 l_{r+t} \hat{d} \hat{l}$
    - $\hat{d} \neq 0$
  - $\mathrm{inject}(\tilde{d}), \mathrm{run}(t), \mathrm{store}(\hat{l})$ where:
    - in δ there is a subsequence $l_r \tilde{d} l_{r+1} 0 \ldots 0 l_{r+t} \hat{d} \hat{l}$
    - $\hat{l}$ needs to be stored, i.e., $\hat{l}$ is in the LBT and it will occur again in another $\delta' \in \Delta_{in}$
  - $\mathrm{inject}(\tilde{d}), \mathrm{run}(t), \mathrm{free}(\bar{l}), \mathrm{store}(\hat{l})$ where:
    - if memory is already full, for a suitably chosen $\bar{l}$

Generating Simulation Campaigns: Algorithm

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

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

Generating Simulation Campaigns: Example



(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)

System Level Formal Verification: Theorem

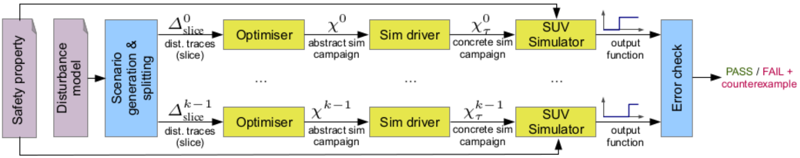
- As a corollary, if an error is present in the specified disturbance traces, our method will find it
- Formally, let $\mathcal{P} = \langle \mathcal{H}, \mathcal{D}, \tau, h \rangle$ be a SLFVP, $\mathcal{S}$ a simulator for $\mathcal{H}$ and $\Delta_{\mathcal{D}}^{h}$ be the set of all labeled disturbance traces of length h. Let χ be the simulation campaign as computed above.
- Then, the answer to $\mathcal{P}$ is FAIL iff the sequence of simulator states contains (s, u, M) s.t. $\mathrm{output}(s) = 0$
- Thus, our approach is *sound* (no false positives) and *complete* (no false negatives)

UNIVERSITÀ DEGLI STUDI DELL'AQUILA

DISIM Dipartimento di Ingegneria e Scienze dell'Informazione e Matematica

System Level Formal Verification

For now, suppose $k = 1$



Experimental Results

SUV: Fuel Control System from Simulink; variable `fuel_air` is never 0 for more than 1s

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)

	$m = 1$	$m = 100{,}000$	
k	time	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%

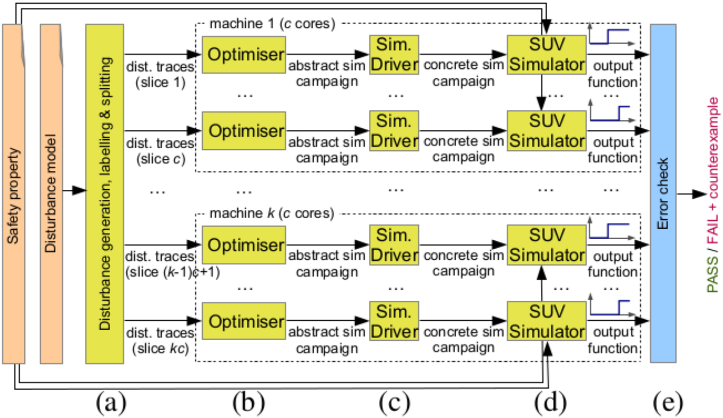
(e) Offline vs. online phase (time in days, h:m:s)

Multicore System Level Formal Verification

- If we have multiple processors, we may easily parallelize our computations
  - both with shared (multicore processors) or distributed memory (clusters)
  - also clusters where k nodes have c cores each
  - we will consider $K = kc$ as the overall number of cores available
- To start with, the generation of disturbance traces may be parallelized
  - an "orchestrator" may expand till horizon fT, for some $0 < f < 1$
  - and then leave the remaining subtree to a "slave" from the other $k - 1$ cores
- It may be shown that labels are ok
- However, this is not the main part to be improved

- Main advantage is in parallelizing the simulation campaign execution
  - simulation phase dominates the overall verification time
- To this aim, starting from the overall disturbances traces set Δ^{h}_{calD}, we must generate k simulation campaigns
- The idea is to perform this is 2 steps:
  1. "slice" Δ^{h}_{calD} in k equal parts
  2. for each slice, compute the corresponding simulation campaign

System Level Formal Verification



UNIVERSITÀ DEGLI STUDI DELL'AQUILA

DISIM Dipartimento di Ingegneria e Scienze dell'Informazione e Matematica

Multicore System Level Formal Verification

- Main advantage is in parallelizing the simulation campaign execution
  - simulation phase dominates the overall verification time
- To this aim, starting from the overall disturbances traces set Δ^h_{calD}, we must generate k simulation campaigns
- The idea is to perform this is 2 steps:
  1. "slice" Δ^h_{calD} in k equal parts
     - all slices have the same length, thus this is easy
  2. for each slice, compute the corresponding simulation campaign as before

Multicore System Level Formal Verification

- First slicing and then optimizing is suboptimal
  - optimal would be to detect all maximal prefix of disturbance traces
  - so that they are stored once and then loaded when needed
- If two slices with a common prefix end up in different slices, no way to do this
- However, reading all disturbance traces file requires too computation time
  - easily a file of hundreds of GBs, or even TBs
- Thus, we are happy with a suboptimal solution

Multicore System Level Formal Verification: Results

#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).

Multicore System Level Formal Verification: Results

	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).

Anytime System Level Formal Verification

- Suppose we have the K simulation campaigns and we are performing the verification phase
- Can we do something better than simply wait for it to finish?
  - as an example: in SAT, there are methodologies computing the coverage achieved so far
  - at "anytime" we can get an estimate of such coverage
- Here we are not interested simply in coverage: we want the Omission Probability (OP)
  - i.e., we want an an upper bound to the probability that there is an error in a yet-to-be-simulated scenario
  - to be provided at any time, during the simulation phase

Anytime System Level Formal Verification

- Main difficulty: optimization comes from lexicographically ordered $\Delta_{\mathcal{D}}^{h}$
- In order to enable some kind of probability on traces, we need random permutations of $\Delta_{\mathcal{D}}^{h}$
- How to obtain this? see in the following

Generating Simulation Campaigns: Standard Algorithm

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

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

Algorithm 1: Optimiser pseudo-code

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

$\chi \leftarrow$ an empty sequence of commands;
$LBT \leftarrow buildLBT(\Delta^\lambda)$;
$\Delta^\lambda_{rnd} \leftarrow rsg(\Delta^\lambda)$;
lastTraces $\leftarrow$ a map associating to each label $l \in LBT$ the index of the last trace in Δ^λ_{rnd} where l occurs;
stored $\leftarrow$ empty set of labels ; /* invariant: $stored \subseteq LBT$ */
$l_0 \leftarrow$ first label common to all traces;
append $store(l_0)$ to χ;
$stored \leftarrow stored \cup \{l_0\}$;
foreach δ^λ *in* Δ^λ_{rnd} **do**
    $l_{load} \leftarrow$ right-most label of δ^λ in *stored*;
    append $load(l_{load})$ to χ;
    append $free(l)$ to χ for each label $l \in stored$ which will never occur in later traces (according to *lastTraces*);
    append to χ commands to simulate δ^λ (from l_{load}) and to store any intermediate states needed to speed-up simulation of later traces;
return χ;

DISIM
Dipartimento di Ingegneria e Scienze dell'Informazione e Matematica

Anytime System Level Formal Verification: Example

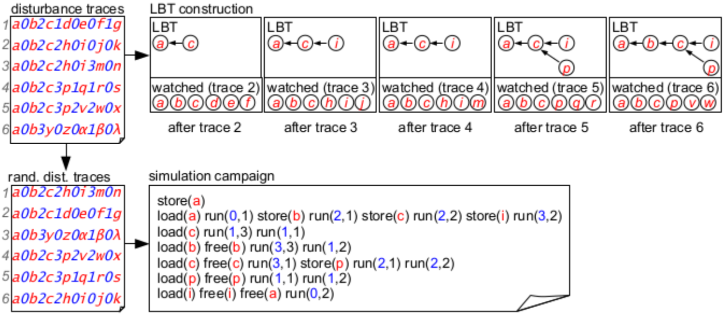


Fig. 5: Simulation campaign optimiser: construction of an LBT from 6 labelled traces in lex order, random sequence generation, and generation of the optimised campaign. Labels are shown as *red letters* and disturbances as *blue numbers*.

Anytime System Level Formal Verification: Definitions

- For a finite set $\Delta = \{\delta_0, \ldots, \delta_{n-1}\}$, if denote $\mathrm{Perm}(\Delta)$ as the set of all permutations of $\delta \in$ *Delta*
  - i.e., $\mathrm{Perm}(\Delta) = \{(\delta_{\pi(0)}, \ldots, \delta_{\pi(n-1)}) \mid \pi : [0, n-1] \cap \mathbb{N} \to [0, n-1] \cap \mathbb{N}$ and π is injective$\}$
  - for a $\hat{\Delta} = (\delta_0, \ldots, \delta_{n-1}) \in \mathrm{Perm}(\Delta)$, we write $\hat{\Delta}(i)$ for δ_i
  - recall that, in our setting, each δ is a disturbance sequence
- A *Random Sequence Generator* (RSG) for Δ is a probability space $(\Omega, \mathcal{F}, \mathbf{P})$ s.t.:
  - $\Omega = \mathrm{Perm}(\Delta)$ is the space of outcomes
  - $\mathcal{F} = 2^\Omega$ is the space of events
  - $\mathbf{P} : \mathcal{F} \to [0, 1]$ is the probability measure
  - in our setting, $\mathbf{P}$ is uniform, thus $\mathbf{P}(\{\omega\}) = \mathbf{P}(\omega) = |\mathrm{Perm}(\Delta)|^{-1} = (|\Delta|!)^{-1}$
  - being $|\Omega| < \infty$, $\forall E \in \mathcal{F}$. $\mathbf{P}(E) = \sum_{\omega \in E} \mathbf{P}(\omega)$

Anytime System Level Formal Verification: Definitions

- Let $\langle \mathcal{H}, \mathcal{D}, h, \tau \rangle$ be a SLFVP, let Δ be a set of disturbance traces and $(\Omega, \mathcal{F}, \mathbf{P})$ be an RSG for Δ.
- Furthermore, let $0 \leq q \leq |\Delta|$ be the current progress with the verification.
  - that is, we already simulated q out of $|\Delta|$ disturbance traces
- Then, the Omission Probability for Δ at stage q, denoted as $\mathrm{OP}_{\mathcal{H}}(\Delta, q)$ is defined as $\mathbf{P}(\{\omega \mid A(\omega, q) \wedge B(\omega, q)\})$
  - $A(\omega, q) \equiv [\exists q < j \leq |\Delta| : \ \psi(\omega(j), h\tau)] = 0$
  - $B(\omega, q) \equiv [\forall 0 \leq j \leq q : \ \psi(\omega(j), h\tau)] = 1$
  - A stands for "after", B stands for "before"

Anytime System Level Formal Verification: Theorem

- Let $\langle \mathcal{H}, \mathcal{D}, h, \tau \rangle$ be a SLFVP, let Δ be a set of disturbance traces and $(\Omega, \mathcal{F}, \mathbf{P})$ be an RSG for Δ. Furthermore, let $0 \leq q \leq |\Delta|$ be the current progress with the verification.
- Then, $\mathrm{OP}_{\mathcal{H}}(\Delta, q) \leq 1 - \frac{q}{|\Delta|}$
  - at the end of the verification, $q = |\Delta|$...
- The previous definitions and this theorem are generalizable to k slices of Δ
- That is, $\mathrm{OP}_{\mathcal{H}}(\Delta_0, \ldots, \Delta_{k-1}, q_0, \ldots, q_{k-1}) \leq 1 - \min_{1l \leq i < k} \frac{q_i}{|\Delta_i|}$
  - being the k parallel verifications independent, all q_i may be different
  - taking the minimum means considering the worst case

Anytime System Level Formal Verification: Results

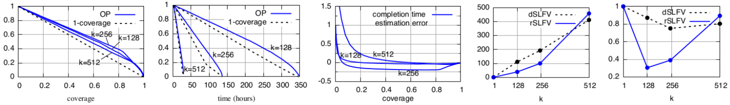
We pay the OP computation in terms of performance degradation

#slices	#traces per slice	dSLFV optimiser	rSLFV optimiser
1	4,023,955	0:7:16	0:35:35
2	2,011,977	0:9:43	0:16:33
4	1,005,988	0:9:0	0:8:37
8	502,994	0:5:27	0:3:42
16	251,497	0:2:8	0:2:51
32	125,748	0:0:57	0:2:36
64	62,874	0:0:29	0:1:21
128	31,437	0:0:17	0:1:44
256	15,718	0:0:8	0:0:42
512	7,859	0:0:4	0:0:13

(a) Computation of simulation campaigns (time in h:m:s)

#mach.	#slices	min	max	avg	speedup	efficiency	approach
16	128	70:6:4	100:17:53	87:49:56	111.56×	87.15%	dSLFV
		216:42:13	348:51:47	308:46:18	39.17×	30.60%	rSLFV
		+209.13%	**+247.83%**	**+251.55%**	**+64.89%**	**+56.55%**	**overhead**
32	256	44:0:27	57:57:27	48:34:6	192.38×	75.15%	dSLFV
		63:53:54	136:18:14	108:14:19	100.03×	39.08%	rSLFV
		+45.20%	**+135.18%**	**+122.86%**	**+48.00%**	**+36.07%**	**overhead**
64	512	18:32:36	26:49:4	23:2:19	411.83×	80.43%	dSLFV
		22:9:19	29:23:33	26:43:31	458.01×	89.46%	rSLFV
		+19.48%	**+9.60%**	**+16.00%**	**−11.21%**	**−9.03%**	**overhead**

(b) Parallel execution of simulation campaigns by dSLFV and rSLFV (time in h:m:s)



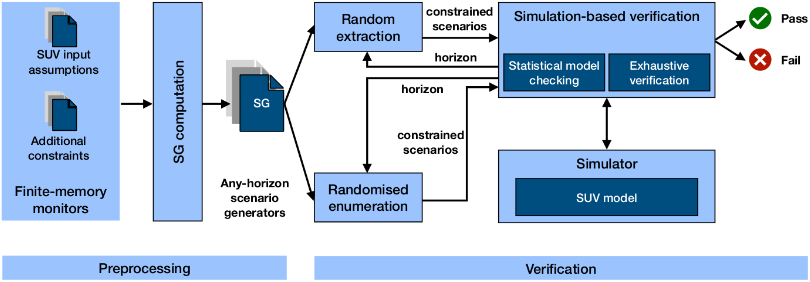
(c) OP against coverage (d) OP & cov. against time (e) Completion time (f) Speedup (g) Efficiency

Fig. 6: Experimental results

System Level Formal Verification: Enhancements

- Main drawbacks for the method seen so far:
  - need of a huge file holding all disturbance traces
    - to be doubled with slicing
  - CMurphi may be not easily used by testing engineers
  - preprocessing is computationally heavy
- Let us see how we can overcome such points

System Level Formal Verification: New Architecture



System Level Formal Verification: Definitions

- System contract: assumptions for inputs, guarantees for outputs
  - if the SUV is fed with inputs satisfying the assumptions...
  - ...then it must provide outputs satisfying the guarantees
- Monitors for assumptions
  - takes an input sequence, and rejects it if violates assumptions
  - assumptions are typically time-unbounded, but a monitor must be an algorithm with finite memory
  - on the other hand, $\mathbb{U}_V$ is finite
  - that is, we have a finite set of disturbances
  - for continuous disturbances, a discretization is required

System Level Formal Verification: Definitions

- We have a finite set $\mathbb{V} = \{v_1, \ldots, v_n\}$
  - each v_i is an *input variable*
  - may have different domains: values (assignments) for v_i are $u \in \mathbb{U}_{v_i}$
  - for $V \subseteq \mathbb{V}$, $\mathbb{U}_V = \times_{v \in V} \mathbb{U}_v$
  - for $u \in \mathbb{U}_V$ and $V' \subseteq$, $w = u_{V'} \in \mathbb{U}_{V'}$ is s.t. $u_v = w_v$ for $v \in V'$ and $w_v = \perp$ otherwise
- At time t, an assignment is provided for all $v \in \mathbb{V}$ (*input time functions*)

System Level Formal Verification: Definitions

- A *monitor* is a Finite State Machine (FSM) $\mathcal{M} = (V, X, x_0, f)$ where:
  - V is the set of input variables as above
  - $\mathbb{U}_V$ is the *monitor input space*
  - X is a finite set of monitor states, $x_0 \in X$ being the initial one
  - $f : X \times \mathbb{U}_V \to X$ is the monitor transition function
    - possibly partial: if it does not result in an infinite path, it is violating the assumptions
- A *trace* is an infinite sequence $(u_0, u_1, \ldots)$ s.t.
  - each u_i is an assignment to variables in V (i.e., $u_i \in \mathbb{U}_V$)
  - there is an infinite path $x_0 u_0 x_1 u_1 \ldots$ in $\mathcal{M}$
- $\mathrm{Traces}(\mathcal{M})$ is the set of all (infinite) traces
- $\mathrm{Traces}|_h(\mathcal{M})$ is the set of all prefixes of length $h \in \mathbb{N}$ of some trace in $\mathrm{Traces}(\mathcal{M})$

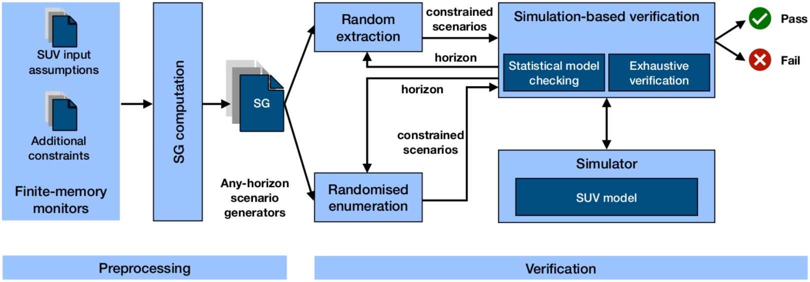
System Level Formal Verification: Definitions

- Systems (and their contracts) may be discrete-time or continuous-time
  - in the former case, we have $\mathbb{T} = \mathbb{N}$, in the latter, $\mathbb{T} = \mathbb{R}$
- Provided that we choose a time-step $\tau \in \mathbb{T}^+$, a monitor may be used for both
  - typically, for discrete-time systems, $\tau >> 1$, whilst for continuous-time systems $\tau << 1$
- In fact, a trace $u_0, u_1, \ldots$ of a monitor $\mathcal{M}$ may be translated in an input time function $u(t) = u_{\lfloor \tau^{-1} \rfloor}$
- For our purposes, monitors may also be black-box: it is sufficient we may repeatedly invoke f
- Note that monitors behave like supervisory controllers

System Level Formal Verification: Definitions

- Suppose we have two monitors $\mathcal{M}_1, \mathcal{M}_2$ with possibly overlapping input variables. The *conjoint monitor* $\mathcal{M} = \mathcal{M}_1 \bowtie \mathcal{M}_2$ is a monitor s.t.
  - $V = V_1 \cup V_2$
  - $X = X_1 \times X_2$, $x_0 = (x_{0,1}, x_{0,2})$
  - $f = f_1 \bowtie f_2$ s.t. $f((x_1, x_2), u) = (f_1(x_1, u|_{V_1}), f_2(x_2, u|_{V_2}))$ if both components are defined
  - the formula holds $\forall x_1 \in X_1, x_2 \in X_2, u \in \mathbb{U}_{V_1 \cup V_2}$
- Note that, for each $(u_0, u_1, \ldots) \in \mathrm{Traces}(\mathcal{M})$, we have that $(u_0|_{V_1}, u_1|_{V_1}, \ldots) \in \mathrm{Traces}(\mathcal{M}_1)$ and $(u_0|_{V_2}, u_1|_{V_2}, \ldots) \in \mathrm{Traces}(\mathcal{M}_2)$
- This allows to define monitors basing on sub-monitors (*compositional modeling*)
  - e.g., assumptions may be implemented conjoining monitors on separate subsets of variables...
  - ... and then monitors for additional constraints on wider variables subsets

System Level Formal Verification: New Architecture



System Level Formal Verification: Algorithms

- Let us go towards the verification phase: as all black-box approaches, it will be with a finite horizon
- We have monitors which considers disturbance traces of infinite length
- For verification purposes, we need to extract prefixes with a given length h
  - the verification may be carried out either exhaustively or by statistical model checking
  - thus, extraction must be possible also in a random way
- As usual, a uniform time step for actual verification is added afterwards
- We want to perform this "online", without storing all traces in a file
  - essentially, monitors are a way to compactly represent disturbance traces

System Level Formal Verification: Algorithms

- It is sufficient to provide two functions:
  - $\mathrm{nb\_traces} : \mathbb{N} \to \mathbb{N}$
    - given h, overall number of disturbance traces of length h accepted by the monitor
  - $\mathrm{trace} : \mathbb{N} \times \mathbb{N} \to \mathbb{U}_V^*$
    - given h and an index $1 \leq i \leq h$, the i-th disturbance trace of length h accepted by the monitor
    - lexicographic order: for a random enumeration, simply extract at random i
- We will show an implementation with time:
  - $O(|\mathbb{U}_V| \cdot |X|^2)$ for initialization
  - $O(1)$ for each subsequent $\mathrm{nb\_traces}$ call
  - $O(h \log \mathbb{U}_V)$ for each subsequent trace call

System Level Formal Verification: Definitions

- The monitor defined by testing engineers may contain finite paths
  - corresponding to non-legal disturbance sequences
  - note that a finite path of length $h+1$ is not to be considered when performing verification with horizon h...
- This is ok for modeling purposes, but we want to get rid of this for the computation
- Thus, we define a new monitor which discards finite paths
  - retaining infinite ones
  - and not introducing other (spurious) paths, of course

System Level Formal Verification: Definitions

- Let $\mathcal{M} = \langle V, X, x_0, f \rangle$ be a monitor. The *safe state function* $\Phi_f : X \to \{0, 1\}$ is defined as the greatest fixed point of

$$\Phi_f(x) \equiv [\exists u, x' : \ x' = f(x, u) \land \Phi_f(x')]$$

  - easier if seen backwards: first, all states such that $\forall u.\ f(x, u) = \perp$ are s.t. $\Phi_f(x) = 0$
    - *deadlock* states
  - then, for all other states x, which *only* goes in x' s.t. $\Phi_f(x') = 0$, we have $\Phi_f(x) = 0$ as well
    - that is, if $\forall u.\ \Phi_f(f(x, u)) = 0$, then $\Phi_f(x) = 0$
  - for all other states x, $\Phi_f(x) = 1$
- A state $x \in X$ is *safe* for $\mathcal{M}$ iff $\Phi_f(x)$ holds
  - all paths starting from x are of infinite length

System Level Formal Verification: Definitions

- Let $\mathcal{M} = \langle V, X, x_0, f \rangle$ be a monitor. The *Scenario Generator* (SG) of $\mathcal{M}$ is a monitor $\mathrm{Gen}(\mathcal{M}) = \langle V, X, x_0, f_{\mathrm{gen}} \rangle$ s.t. $f_{\mathrm{gen}}(x, u) = f(x_u)$ if $\Phi_f(f(x, u)) = 1$ and $f_{\mathrm{gen}}(x, u) = \perp$ otherwise
  - i.e., we remove transitions towards non-safe states
  - by theorems on fixed points, a SG always exists and it is unique
  - may not contain any transition...
  - using controller theory parlance, the scenario generator is the most liberal supervisory controller for $\mathcal{M}$
- Given $\mathcal{M}$, $\mathrm{Gen}(\mathcal{M})$ can be computed in time $O(|\mathbb{U}_V| \cdot |X|^2)$

System Level Formal Verification: Definitions

- Monitors may be accessed as *black-box* code, provided that they:
  - provide functions to get and set the current internal state
    - as some possibly non-interpretable bytes sequence
  - start from some initial internal state
  - provide a function which, given the current internal state, returns the list of admissible actions
  - provide a function which, given the current internal state and an admissible action, changes its internal state
  - provide a function which, given an action, provide a possibly non-interpretable encoding for such action
- As an example, this is easy to do with Python

System Level Formal Verification: Definitions

- Let $\mathcal{M} = \langle V, X, x_0, f \rangle$ be a monitor and $\mathrm{Gen}(\mathcal{M}) = \langle V, X, x_0, f_{\mathrm{gen}} \rangle$ be its SG. Then:
  - each finite path in $\mathrm{Gen}(\mathcal{M})$ may be extensible to an infinite path
    - otherwise phrased: the last state of the path always has at least one successor state
    - *non-blocking* property
  - $\mathrm{traces}(\mathcal{M}) = \mathrm{traces}(\mathrm{Gen}\mathcal{M}))$
    - recall that "traces" mean an infinite sequence...
- Such properties follow directly from the definition

System Level Formal Verification: Definitions

- Let $\mathcal{M}_1, \mathcal{M}_2$ be two monitors. Then:
  - $\mathrm{Gen}(\mathcal{M}_1) = \mathrm{Gen}(\mathrm{Gen}(\mathcal{M}_1))$
    - blocking paths only need to be removed once
  - if $V_1 \cap V_2 = \varnothing$, then $\mathrm{Gen}(\mathcal{M}_1 \bowtie \mathcal{M}_2) = \mathrm{Gen}(\mathcal{M}_1) \bowtie \mathrm{Gen}(\mathcal{M}_2)$
    - i.e., if $\mathcal{M}_1, \mathcal{M}_2$ are *independent* monitors
    - if there is some common variable, then $\mathcal{M}_1$ could restrict something which is allowed in $\mathcal{M}_2$, thus...
  - $\mathrm{Gen}(\mathcal{M}_1 \bowtie \mathcal{M}_2) = \mathrm{Gen}(\mathrm{Gen}(\mathcal{M}_1) \bowtie \mathrm{Gen}(\mathcal{M}_2))$
    - general case
- Such properties allow incremental combination of monitors

System Level Formal Verification: Definitions

- The following is needed to compute $\mathrm{nb\_traces}$ and trace
- Let $\mathcal{M} = \langle V, X, x_0, f \rangle$ be a monitor and $\mathrm{Gen}(\mathcal{M}) = \langle V, X, x_0, f_{\mathrm{gen}} \rangle$ be its SG. Then:
  - $\mathrm{ext} : X \times \mathbb{N} \to \mathbb{N}$ is s.t.
    - $\mathrm{ext}(x, 0) = 1$ for all $x \in X$
    - $\mathrm{ext}(x, k) = \sum_{u \in \mathbb{U}_V} \mathrm{ext}(f_{\mathrm{gen}}(x, u), k - 1)$ for all $x \in X, k \in \mathbb{N}^+$
    - of course, $\mathrm{ext}(\perp, k) = 0$ for all $k \in \mathbb{N}$
    - $\mathrm{ext}(x, k) = \#$all distinct paths of length k starting from x
  - $\xi : X \times \mathbb{U}_V \times \mathbb{N} \to \mathbb{N}$ is s.t., for all $x \in X, u \in \mathbb{U}_V, k \in \mathbb{N}$, $\xi(x, u, k) = \sum_{\hat{u} < u} \mathrm{ext}(f_{\mathrm{gen}}(x, \hat{u}), k)$
    - of course, some ordering is required in each $\mathbb{U}_v$, so we can take the lexicographic one for $\mathbb{U}_V$
    - $\xi(x, u, k) = \#$distinct paths of length k starting from x with some action preceding u

System Level Formal Verification: Algorithms

global

$Gen(\mathcal{M}) = (V, X, x_0, f_{\text{gen}})$;

$h_{\text{max}} \in \mathbb{N} \cup \{$**undef**$\}$, initially **undef**;

ext, a map of the form $X \times \mathbb{N} \to \mathbb{N}$, initially empty;

ξ, a map of the form $X \times \mathbb{U}_V \times \mathbb{N} \to \mathbb{N}$, init. empty;

// Invariant: $ext(x, h)$ & $\xi(x, u, h)$ defined iff $h \leq h_{max}$

function *nb_traces*(h)

Input: $h \in \mathbb{N}$

if $h_{max} =$ **undef** *or* $h > h_{max}$ **then**

incrementally compute *ext* and ξ up to h;

$h_{max} \leftarrow h$;

return $ext(x_0, h)$;

function *trace*(i, h)
  Input: $i \in \mathbb{N}$, $h \in \mathbb{N}$
  Output: $(u_0, u_1, u_2, \ldots u_{h-1})$, i-th trace of len. h
  if $i \geq$ *nb_traces*(h) **then** **error** *index out of bounds*;
  $x \leftarrow x_0$; $k \leftarrow h$; $m \leftarrow i$;
  for j *from* 0 *to* $h - 1$ **do**
    $u_j \leftarrow \max \{u \mid \xi(x, u, k-1) \leq m\}$;
    $m \leftarrow m - \xi(x, u_j, k-1)$;
    $x \leftarrow f_{\text{gen}}(x, u_j)$;
    $k \leftarrow k - 1$;
  return $(u_0, u_1, u_2, \ldots u_{h-1})$;

UNIVERSITÀ DEGLI STUDI DELL'AQUILA
DISIM Dipartimento di Ingegneria e Scienze dell'Informazione e Matematica

System Level Formal Verification: Algorithms

- The above algorithms are correct, that is the following holds
- Let $\mathcal{M} = \langle V, X, x_0, f \rangle$ be a monitor and $\mathrm{Gen}(\mathcal{M}) = \langle V, X, x_0, f_{\mathrm{gen}} \rangle$ be its SG. Then:
  - for all $h \in \mathbb{N}$, $\mathrm{nb\_traces}(h) = \mathrm{card}(\mathrm{traces}(\mathrm{Gen}(\mathcal{M}))|_h)$
  - for all $h \in \mathbb{N}, i \in [0, \mathrm{nb\_traces}(h) - 1] \cap \mathbb{N}$, $\mathrm{trace}(i, h)$ returns the i-th element of $\mathrm{traces}(\mathrm{Gen}(\mathcal{M}))|_h$
    - lexicographic order

- Let $\mathcal{M}_1, \mathcal{M}_2$ be two independent monitors. Then, for all $h \in \mathbb{N}$:
  - $\mathrm{nb\_traces}_{\mathcal{M}_1 \bowtie \mathcal{M}_2}(h) = \mathrm{nb\_traces}_{\mathcal{M}_1}(h)\mathrm{nb\_traces}_{\mathcal{M}_2}(h)$
  - for all $i \in [0, \mathrm{nb\_traces}_{\mathcal{M}_1 \bowtie \mathcal{M}_2}(h) - 1] \cap \mathbb{N}$, $\mathrm{trace}_{\mathcal{M}_1 \bowtie \mathcal{M}_2}(i, h) = \mathrm{trace}_{\mathcal{M}_1}(\mathrm{sel}(i, 1), h) \cdot \mathrm{trace}_{\mathcal{M}_2}(\mathrm{sel}(i, 2), h)$, where:
    - $\mathrm{sel}(i, 1) = \left\lfloor \frac{i}{\mathrm{nb\_traces}_{\mathcal{M}_2}(h)} \right\rfloor$
    - $\mathrm{sel}(i, 2) = i \bmod \mathrm{nb\_traces}_{\mathcal{M}_2}(h)$
    - operator $\cdot$ is the pairing of two traces: $(u_{0,1}, \ldots, u_{h-1,1}) \cdot (u_{0,2}, \ldots, u_{h-1,2}) = ((u_{0,1}, u_{0,2}), \ldots, (u_{h-1,1}, u_{h-1,2}))$
- This means that we may compute $\mathrm{nb\_traces}_{\mathcal{M}_1 \bowtie \mathcal{M}_2}(h)$ and $\mathrm{trace}_{\mathcal{M}_1 \bowtie \mathcal{M}_2}(h)$ without computing $\mathcal{M}_1 \bowtie \mathcal{M}_2$
  - only the (typically much smaller) $\mathcal{M}_1, \mathcal{M}_2$ are required (separately)

System Level Formal Verification: Case Studies

- Fuel control system (FCS): classical example from Simulink distribution
  - also used in papers for Statistical Model Checking
- Controller for a fault tolerant gasoline engine
  - goal: keep the air-fuel ratio close to 14.6
  - that is, a stoichiometric ratio representing a good compromise between power, fuel economy and emissions
  - air-fuel ratio is between the air mass flow rate pumped from the intake manifold and the fuel mass flow rate injected at the valves
- *Experiment scenario*: a full set of disturbance traces to be verified

FCS Experiment Scenarios

- For FCS, we are interested in its 4 sensors:
  - throttle angle, speed, residual oxygen in exhaust gas (EGO) and manifold absolute pressure (MAP)
- All of them may fail
  - fortunately, they are typically repaired (i.e., restarted) within a few seconds
- FCS is expected to withstand one failure at a time
  - by compensating with internal commands
- From the verification point of view, we want to exercise the system with multiple (non-contemporary) failures and repairs

FCS Experiment Scenarios

- Base assumptions, which are valid for all experiment scenarios:
  - each of the four sensor may fail at any time
  - each sensor, once failed, is repaired within a given time: 3–5 (throttle), 5–7 (speed), 10–15 (EGO), 13–17 (MAP)
  - but for each time, only one sensor may be in "failed" state
  - e.g., if in a disturbance trace throttle fails at step 1 and is repaired at time 4, there cannot be any other failure in [1, 4]
- If we have separate monitors for each sensor, many non-valid traces can be generated
  - to be discarded when computing the SG of the conjoint monitor also considering the above assumptions
- However, here it is easier to implement all such constraints within one monitor
- Experiment scenarios are obtained by adding one or more monitors (i.e., constraints) from the following table

FCS Experiment Scenarios

constraint monitor	description
1	Each sensor will fail every 15–20 t.u.
2	Whenever a fault on the throttle sensor occurs, a fault on the speed sensor will occur within 9–11 t.u.
3	Whenever a fault on the throttle sensor occurs, a fault on the speed sensor will occur within 13–15 t.u.
4	Whenever a fault on the throttle sensor occurs, a fault on the speed sensor will occur within 18 or 19 t.u.
5	Whenever a fault on the EGO sensor occurs, a fault on the MAP sensor will occur within 16 or 17 t.u.
6	Whenever a fault on the EGO sensor occurs, a fault on the MAP sensor will occur within 20 or 21 t.u.

System Level Formal Verification: Case Studies

- Buck DC/DC Converter: another classical example used in litarature
  - also used in papers for controllers generation
- Mixed-mode analog circuit converting the DC input voltage V_i to a desired DC output voltage V_o
  - e.g., used inside laptop battery
  - to do this, it is equipped with a microcontroller activating a switch u
  - to react to changes in the input voltage and other parameters (e.g., the load R)

Buck DC/DC Converter Experiment Scenarios

- We are interested in the following two parameters: V_i and R
  - disturbances act by modifying the parameter value
  - in an bounded way: it may be modified so as to take values in a n-steps discretized interval $[m, M]$, i.e., $\{m + is \mid i = 0, \ldots, n-1 \wedge s = \frac{M-n}{n}\}$
  - we have $n = 12$ for V_i and $n = 6$ for R
  - for both V_i and R, $[m, M]$ is the corresponding nominal range: $[70, 130]V$ for V_i and $[70, 130]\Omega$ for R
- Base assumptions: the changes as above and
  - no changes for the first 2 steps
  - once a change is made, do not modify further for the following 6 steps for V_i and 5 steps for R

Buck DC/DC Converter Experiment Scenarios

- Differently from FCS, buck actually has two independent monitors
  - one for V_i and one for R
- As discussed before, they can be computed separately and then conjoined in the "easy" way
- Experiment scenarios are obtained by adding one or more monitors (i.e., constraints) from the following table

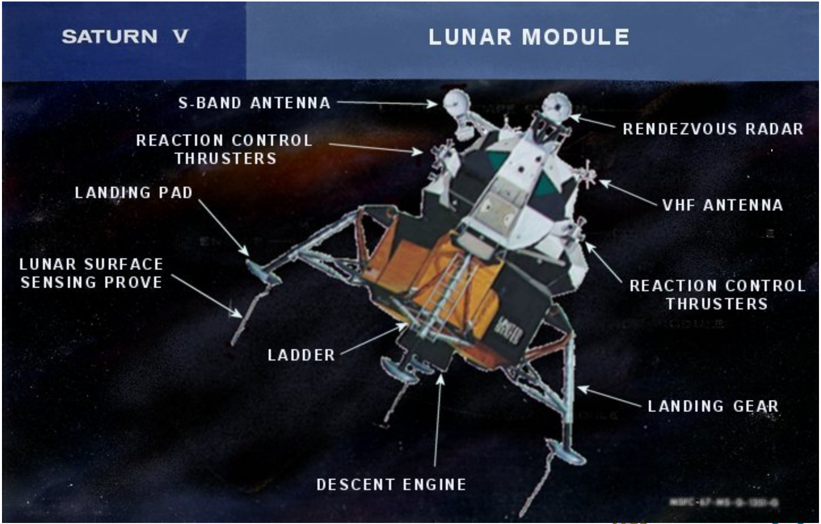
Buck DC/DC Converter Experiment Scenarios

constraint monitor	description
1	V_i changes at least every 6 t.u.
2	V_i changes at least every 7 t.u.
3	R changes at least every 5 t.u.
4	R changes at least every 6 t.u.
5	V_i and R do not change simultaneously
6	Whenever V_i changes, R will change after 8 or 9 t.u.
7	Whenever V_i changes, R will change after 2 t.u.

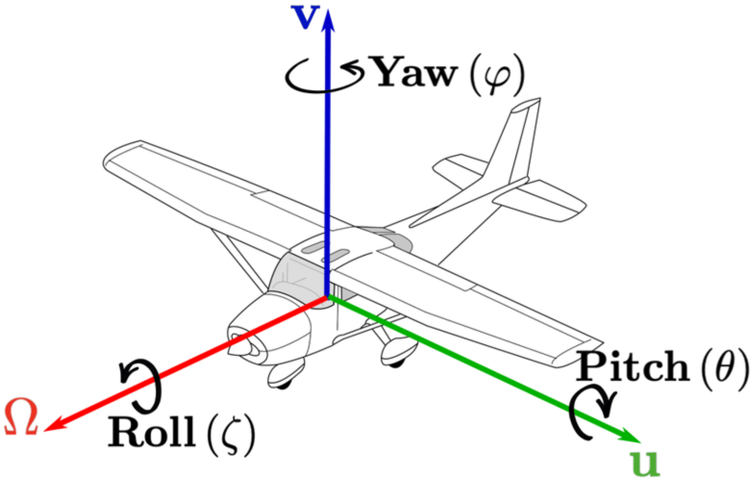
System Level Formal Verification: Case Studies

- Apollo: other classical example from Simulink distribution
- Phase-plane controller for the autopilot of the LEM (Lunar Excursion Module) in the Apollo 11 mission
  - goal: given a request to change attitude, actuate jets so as to achieve it
- 3 sensors and 16 jets
  - sensors detect the attitude of the module: yaw, roll and pitch
  - jets to change the attitude

System Level Formal Verification: Case Studies



System Level Formal Verification: Case Studies



Apollo Experiment Scenarios

- We disturb both the sensors and the jets
- Sensors are disturbed in 6 possible ways
  - for our purposes, a number from 1 to 6
  - such number is then translated at verification time in a one of 6 predefined continuous-time signal noise
- Jets may become unavailable for 2 or 3 time units
  - control will have to compensate...
- External request of change attitude may be any in any of the 3 directions
  - only 3 values: $\{-1, 0, 1\}$
  - no requests undoing the immediately preceding one
- Experiment scenarios are obtained by adding one or more monitors (i.e., constraints) from the following table

Apollo Experiment Scenarios

constraint monitor	description
1	Only jets number 15 and 16 may be temporarily unavailable
2	Whenever a jet is actuated for 2 consecutive t.u., it will certainly become unavailable within 3 or 4 t.u.
3	At most 1 jet is unavailable at any time
4	Rotation requests regard at most 1 axis each
5	Rotation requests regard at most 2 axes each
6	Noise signal changes for at most 1 sensor at any time
7	Noise signal for each sensor remains stable for at least 5 and at most 10 t.u. and changes by ± 1 position in the given order

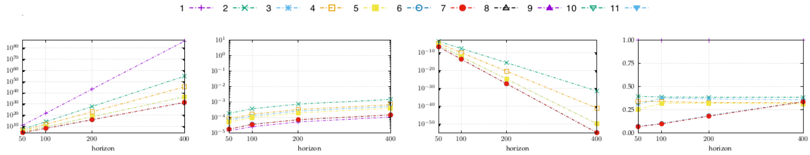
Results for Generating SGs

SUV	SG nb.	$\mathcal{M}$ assumptions monitor	$\mathcal{M}$ constraint monitors	$\mathcal{M}$ size of input space	$\|Gen(\mathcal{M})\|$ time [s]
FCS	1	$\mathcal{A}_{\text{FCS}}$	–	6	0.1
	2	$\mathcal{A}_{\text{FCS}}$	1	6	7.99
	3	$\mathcal{A}_{\text{FCS}}$	1, 3	6	4.92
	4	$\mathcal{A}_{\text{FCS}}$	1, 2	6	4.61
	5	$\mathcal{A}_{\text{FCS}}$	1, 4	6	6.34
	6	$\mathcal{A}_{\text{FCS}}$	1, 4, 5	6	5.92
	7	$\mathcal{A}_{\text{FCS}}$	1, 4, 6	6	6.55
BDC	1	$\mathcal{A}_i$	–	5	0.19
	2	$\mathcal{A}_R$	–	5	0.17
	3	$\mathcal{A}_i \bowtie \mathcal{A}_R$	–	25	0.36
	4	$\mathcal{A}_i$	1	5	0.12
	5	$\mathcal{A}_i$	2	5	0.17
	6	$\mathcal{A}_R$	3	5	0.11
	7	$\mathcal{A}_R$	4	5	0.16
	8	$\mathcal{A}_i \bowtie \mathcal{A}_R$	5	25	37.34
	9	$\mathcal{A}_i \bowtie \mathcal{A}_R$	2, 4, 5	25	29.68
	10	$\mathcal{A}_i \bowtie \mathcal{A}_R$	2, 4, 5, 6	25	1.94
	11	$\mathcal{A}_i \bowtie \mathcal{A}_R$	1, 3, 5, 7	25	2.16
ALMA	1	$\mathcal{A}_{\text{rj}}$	–	1 769 472	0.44
	2	$\mathcal{A}_{\text{rj}}$	1	108	0.44
	3	$\mathcal{A}_{\text{rj}}$	1, 2	108	448.88
	4	$\mathcal{A}_{\text{rj}}$	1, 2, 3	108	247.27
	5	$\mathcal{A}_{\text{rj}}$	1, 2, 3, 4	108	55.19
	6	$\mathcal{A}_{\text{rj}}$	1, 2, 3, 5	108	188.3
	7	$\mathcal{A}_{\text{s}}$	–	27	2.94
	8	$\mathcal{A}_{\text{s}}$	6	27	1.33
	9	$\mathcal{A}_{\text{s}}$	6, 7	27	782.2
	10	$\mathcal{A}_{\text{ALMA}}$	1, 2, 3, 4, 6, 7	2916	837.39

Experimental Results: Presentation

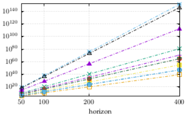
- For each case study, we show, as a function of some meaningful values of the verification horizon h:
  - the number returned by $\mathrm{nb\_traces}(h)$, i.e., the overall number of traces fulfilling the given monitors
  - trace extraction time: computation time, in seconds, to compute $\mathrm{trace}(i, h)$
    - 1000 values for i are chosen in a uniformly random way in $[0, \mathrm{nb\_traces}(h) - 1]$
    - the average value for the computation time is then shown
    - this allows to amortize computation of ext, ξ
  - selectivity of monitors: $\frac{\#\text{traces with all constraints}}{\#\text{traces with base assumptions}}$
    - having tiny values shows SGs selects *important* experiments scenario
    - errors, if any, are discovered first
  - selectivity of SGs: $\frac{\#\text{traces with Gen}(\mathcal{M})}{\#\text{traces with } \mathcal{M}}$
    - at the denominator, we consider possibly blocking (i.e., non-valid) traces

Experimental Results for FCS

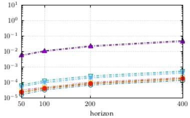


Number of traces | Trace extraction time [s] | Constraint monitors selectivity | SG selectivity

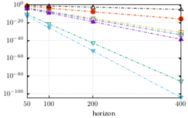
Experimental Results for Buck



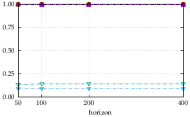
Number of traces



Trace extraction time [s]



Constraint monitors selectivity

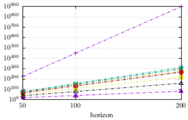


SG selectivity

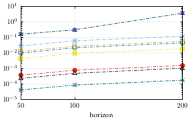
UNIVERSITÀ DEGLI STUDI DELL'AQUILA

DISIM Dipartimento di Ingegneria e Scienze dell'Informazione e Matematica

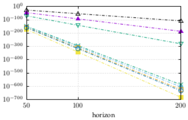
Experimental Results for Apollo



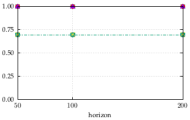
Number of traces



Trace extraction time [s]



Constraint monitors selectivity



SG selectivity