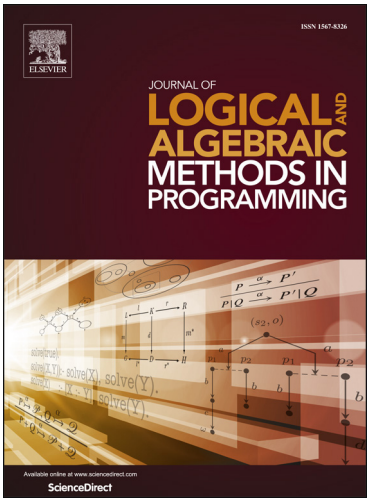
Journal Pre-proof



On Checking Equivalence of Simulation Scripts

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

PII: S2352-2208(21)00003-1
DOI: https://doi.org/10.1016/j.jlamp.2021.100640
Reference: JLAMP 100640

To appear in: *Journal of Logical and Algebraic Methods in Programming*

Received date: 3 March 2020
Revised date: 11 August 2020
Accepted date: 10 January 2021

Please cite this article as: T. Mancini, F. Mari, A. Massini et al., On Checking Equivalence of Simulation Scripts, *Journal of Logical and Algebraic Methods in Programming*, 100640, doi: https://doi.org/10.1016/j.jlamp.2021.100640.

On Checking Equivalence of Simulation Scripts

Toni Mancini[a], Federico Mari[b], Annalisa Massini[a], Igor Melatti[a], Enrico Tronci[a]

[a] *Computer Science Department*
Sapienza University of Rome
Rome - Italy
[b] *Computer Science at the Department of Movement, Human and Health Sciences*
University of Rome Foro Italico
Rome - Italy

Abstract

To support *Model Based Design* of *Cyber-Physical Systems* (CPSs) many *simulation based* approaches to *System Level Formal Verification* (SLFV) have been devised. Basically, these are *Bounded Model Checking* approaches (since simulation horizon is of course bounded) relying on simulators to compute the system dynamics and thereby verify the given system properties. The main obstacle to simulation based SLFV is the large number of *simulation scenarios* to be considered and thus the huge amount of simulation time needed to complete the verification task. To save on computation time, simulation based SLFV approaches exploit the capability of simulators to *save* and *restore* simulation states. Essentially, such a time saving is obtained by *optimising* the simulation script defining the simulation activity needed to carry out the verification task. Although such approaches aim to (bounded) formal verification, as a matter of fact, the proof of correctness of the methods to optimise simulation scripts basically relies on an intuitive semantics for simulation scripting languages. This hampers the possibility of formally showing that the optimisations introduced to speed up the simulation activity do not actually omit checking of relevant behaviours for the system under verification. The aim of this paper is to fill the above gap by presenting an *operational semantics* for simulation scripting languages and by proving *soundness* and *completeness* properties for it. This, in turn, enables formal proofs of *equivalence* between unoptimised and optimised simulation scripts.

Keywords: Formal Verification, Simulation based Formal Verification, Formal Verification

1. Introduction

Safety critical system development standards require the use of *Formal Methods* typically through a model based approach. For example, the DO-178C [1] by which the certification authorities approve all commercial software-based aerospace systems, follows the DO-333 Formal Methods supplement (governing usage in airborne and ground-based aviation software) and the DO-331 Model-Based Development and Verification Supplement (for avionics software development).

Model based Verification and Validation (VV) requires two main steps: 1) on the basis of the system requirements define the system properties to be verified and the set of operational scenarios to be simulated; 2) carry out the simulation activity. Both such steps are very time consuming. Furthermore, step 1 requires highly skilled personnel. Not surprisingly a considerable part of the time and budget needed to complete the design of mission or safety critical systems goes into the VV activity. In fact, VV easily accounts for more than 50% of the system design cost/time. This problem is further exacerbated by the ever increasing complexity and autonomy of many Cyber-Physical Systems (CPSs) such as aerial, terrestrial or maritime (possibly unmanned) vehicles, spacecrafts, robots, biomedical devices, etc.

One of the most challenging, costly and time consuming VV activity is System Level Verification (SLV) whose goal is to verify that the *whole* (i.e., software + hardware) system meets the given specifications. This motivates research on Model Based Design (MBD) methods and tools since they hold the promise to decrease time and cost for SLV of complex CPSs. MBD achieves its goals by first developing a mathematical model of the System Under Verification (SUV) and then by automatically analysing such a system model in order to support, among other things, SLV, design space exploration, operator training. This enables early detection of design errors, well before the system implementation starts. The SUV model must encompass discrete behaviours (stemming, e.g., from software based subsystems) as well as continuous ones (stemming, e.g., from physical subsystems). Accordingly, the SUV

is typically modelled using (see, e.g., [2] and citations thereof) *Hybrid Systems* (e.g., as in model checkers like [3, 4, 5, 6, 7, 8]).

While hybrid system model checkers can be used during the preliminary phases of a system design, because of state explosion, they currently cannot handle SLV of a full fledged system design, our target here. Accordingly, in the following we will focus on simulation based system level (formal) verification.

Simulation is currently the main workhorse for SLV and is supported by MBD tools (such as Simulink [9], VisSim [10]), and Modelica [11] based simulators (such as OpenModelica [12], JModelica [13] and Dymola [14]). During simulation, the software (the *actual* one or a model of it) reads [sends] values from [to] mathematical models (*simulation*) of the physical systems (e.g. engines, analog circuits, etc.) it will be interacting with.

For example, within the space domain system, simulation is used to support system verification within all phases of system design. For example (see [15]), simulation is used in phase 0 to support requirements validation, in phase A to support system design verification and mission verification, in phase B to support functional (subsystem) verification, in phase C to support *On-Board Software Verification* (OBSW) and *Assembly Integration & Verification* (AIV), in phase D to validate ground segment operations procedures, in phase E to support team training and, finally, in phase F to investigate system disposal options. Accordingly, a system model (and its simulator) will have to interact with different subsystems and users, depending on the system life-cycle phase we are focusing on.

As usual within MBD, we model our SUV as a *deterministic system* and model nondeterministic behaviours (such as faults, user inputs, parameter variations) with disturbances (*uncontrollable events*). 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 scenarios. Correctness of a system is thus defined with respect to such a set of *admissible* disturbance sequences and the goal of the simulation based verification activity is exactly that of showing that indeed the considered SUV can withstand all 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 less than 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.

We focus on simulation based *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) our simulation activity returns *PASS* if there is no *admissible* disturbance sequence of length T and time step τ that violates the property under verification and *FAIL*, along with a counterexample, otherwise. In other words, *Bounded* SLFV is an *exhaustive* (with respect to the set of admissible disturbance sequences) simulation. In such a framework, exhaustive simulation works as a *black box bounded model checker* where the SUV behaviour is defined by a simulator.

1.1. Motivations

Typically the set of operational scenarios (*admissible disturbance sequences*) to be simulated in order to complete a SLV activity is defined through a *simulation campaign*, that is a simulation script driving the simulation activity.

In our context the number of admissible disturbance sequences is finite since the number of disturbances is finite and the time horizon as well as the time quantum between disturbances are both bounded. Nevertheless the number of operational scenarios to be considered within a VV activity can be huge (easily many millions). As a result, depending on the system considered and on the degree of assurance sought, a simulation campaign may easily require months of simulation activity since simulating a single scenario can easily require many seconds of simulation even for a relatively small CPS.

Operational scenarios to be simulated are automatically generated from formal specifications of the system environment (*e.g.*, as in [16, 17, 18, 19, 20, 21, 22]) or from a database of operational scenarios (*e.g.*, as in [15, 23, 24]).

A simulation campaign is typically represented using the scripting language of the simulator running the model of the SUV. Beside the syntax (which is not our focus here) all such scripting languages offer four basic commands: load, run, store, free. Command store(x),

stores, say in a file with name x, the current simulation state. Command load(x) loads into the simulator memory the simulation state stored in file x. Command run(λ, τ) sets the values for the model parameters to λ and advance the simulation of τ seconds. Command free(x), deletes (the state stored in) file x.

To decrease the time needed to run a simulation campaign many approaches have been devised. Typically such approaches save simulation time by exploiting the fact that many operational scenarios have common *prefixes* (*i.e.*, they share their initial sequence of disturbances). By saving (through command store) the simulation state attained at the end of such common prefixes we can avoid simulating twice the same disturbance sequence by just loading (through command load) a previously saved simulation state. For example, Listing 1 shows a MATLAB script defining a simulation campaign for the *Inverted Pendulum on Cart* system from Simulink distribution (further details are in Section 4.1). Listing 2 shows an optimised version of the simulation campaign in Listing 1. Such an optimised version is obtained by saving suitably chosen intermediate simulation states in order to avoid recomputing prefixes of operational scenarios. Comparing Listings 1 and 2 we see that, even within this tiny example, we save about 13% of computation time (about 3 seconds for the unoptimised simulation script against 2.6 seconds for the optimised one). This is because for CPSs simulation is computationally expensive even for relatively small systems.

From the above considerations, we see that optimisation methods for simulation scripts defining simulation campaigns are at the very heart of simulation based *Bounded Model Checking* approaches to SLFV. Of course, in order to formally guarantee *correctness* of the optimisation method used, it is essential to show that the unoptimised simulation script and the optimised one entail the same set of operational scenarios. Unfortunately, even though such approaches aim at (bounded) formal verification, as a matter of fact, to the best of our knowledge, the proof of correctness of simulation scripts optimisation methods basically relies on the intuitive semantics for simulation scripting languages. This hampers the possibility of formally showing that the optimisations introduced to speed up the simulation activity do not actually omit checking of relevant behaviours for the SUV (*i.e.*, that we omit to simulate some of the operational scenarios entailed by the original unoptimised simulation

Listing 1: An unoptimised simulation campaign consisting of three operational scenarios. Commands followed by comment *(*)* or *(-)* both lead to state `x3`.

```
s_load('x0');
s_run(0, 0.04);    % (*)
s_run(1, 0.04);    % (*)
s_run(1, 0.04);    % (*)     % x3
s_run(1, 0.04);
s_run(1, 0.04);
s_run(1, 0.04);
s_load('x0');
s_run(0, 0.04);    % (-)
s_run(1, 0.04);    % (-)
s_run(1, 0.04);    % (-)     % x3
s_run(0, 0.04);
s_run(0, 0.04);
s_run(1, 0.04);
s_load('x0');
s_run(1, 0.04);
s_run(0, 0.04);
s_run(2, 0.04);
s_run(1, 0.04);
s_run(0, 0.04);
s_run(2, 0.04);
% Elapsed time is 3.020143 seconds
```

Listing 2: An optimised version of the simulation campaign in Listing 1. By storing state `x3`, we avoid repeating commands with *(-)* in Listing 1.

```
s_load('x0');
s_run(0, 0.04);    % (*)
s_run(1, 0.04);    % (*)
s_run(1, 0.04);    % (*)     % x3
s_store('x3');     % (+)     % x3
s_run(1, 0.04);
s_run(1, 0.04);
s_run(1, 0.04);
s_load('x3');      % (+)     % x3
s_run(0, 0.04);
s_run(0, 0.04);
s_run(1, 0.04);
s_free('x3');
s_load('x0');
s_run(1, 0.04);
s_run(0, 0.04);
s_run(2, 0.04);
s_run(1, 0.04);
s_run(0, 0.04);
s_run(2, 0.04);
% Elapsed time is 2.609006 seconds
```

campaign).

The aim of this paper is to fill the above gap by presenting an *operational semantics* for simulation scripting languages and by proving *soundness* and *completeness* properties for it. This, in turn, enables formal proofs of *equivalence* between unoptimised and optimised simulation scripts. We focus on the four simulation commands mentioned above since they form the core of the scripting languages for all simulators.

1.2. Main Contributions

Our main contributions can be summarised as follows.

We provide a formal *operational semantics* for *simulation scripting languages* focusing on their core instructions: load, store, run and free. Such an operational semantics allows us to formally prove if it holds that two simulation scripts entail the same set of *operational scenarios*.

We show *soundness* of our operational semantics by proving that any *simulation script* defines a set of (*in-silico*) experiments that can be carried out on our SUV. In other words, any simulation script defines a set of *operational scenarios*.

We show *completeness* of our operational semantics by proving that any set of (*in-silico*) experiments to be carried out on our SUV (*i.e.*, any set of *operational scenarios*) can be defined through a *simulation script*.

1.3. Related Work

This paper is a journal version of the conference paper [25]. The present version extends [25] as follows. The definition of *Discrete event sequence* and the related definitions of Section 3 are revised in order to avoid the use of the *no disturbance* event, since it does not belong to the set of simulator commands. Definition 12 of simulator commands and transition function has been simplified resting on the well-established notion of *Labelled Transition System*. Finally, we provide the formal proof of Lemma 1 and Theorems 1 and 2, that in [25] are just outlined through examples.

Semantics for the modelling language of Simulink has been studied in [26, 27, 28] and citations thereof. Also, semantics for Modelica modelling language [11], that is supported

by many open source (*e.g.*, Open Modelica, JModelica) as well as commercial (*e.g.*, Dymola, Wolfram System Modeler) simulators, has been investigated in [29, 30] and citations thereof. We note that all such research work focuses on defining a semantics for the simulator modelling language, that is the language used to define the model to be simulated (our SUV), whereas our paper focuses on defining a semantics for the language (namely, the simulator scripting language) used to control the simulation process itself.

SLFV of cyber-physical systems via simulation based bounded model checking has been studied in many contexts. Here are a few examples.

Formal verification of fully general Simulink models has been investigated in [16, 22, 20, 21, 17, 18, 19]. Formal verification of satellite operational procedures using ESA SIMSAT simulator has been investigated in [31].

Simulation based reachability analysis for large linear continuous time dynamical systems has been investigated in [32].

A simulation based data-driven approach to verification of hybrid control systems described by a combination of a black-box simulator for trajectories and a white-box transition graph specifying mode switches has been investigated in [33].

Formal verification of discrete time Simulink models (e.g., Stateflow or models restricted to discrete time operators) with small domain variables has been investigated in [34, 35, 36].

Simulation based *falsification* of CPS properties has been extensively investigated for Simulink models. Examples are in: [37, 38, 39, 40, 41].

Simulation based approaches to statistical model checking have been also widely investigated. Here are just a few examples. Simulink models for CPS have been studied in [42], mixed-analog circuits have been analyzed in [43]. Smart grid control policies have been considered in [44, 45, 46], biological models have been studied in [47, 48, 49].

Of course *Model Based Testing* (e.g., see [50]) has widely considered automatic generation of test cases from models. In our setting, automatic generation of simulation scenarios (for Simulink) has been investigated, for example, in [51, 52, 53, 54].

Finally, synergies between simulation and formal methods have been widely investigated also in digital hardware verification. Examples are in [55, 56, 57, 58] and citations thereof.

All simulation based verification approaches considered in the literature heavily rely on optimising the simulation scripts defining the simulation campaign to be carried out to complete the planned verification activity. However, to the best of our knowledge, none of them addresses the issue of formally proving the equivalence between the optimised simulation script and the unoptimised one. By providing an *operational semantics* for the core commands of simulator scripting languages we aim at filling this gap.

1.4. Outline of the paper

Section 2 shows the impact that errors may have on the verification activity. Section 3 describes how we model disturbances as uncontrollable inputs (*events*) to our (cyber-physical) SUV that, in turn, is modelled as a *discrete event system*. Section 4 formalises the notion of simulator, simulation campaign and, finally, operational semantics for simulation scripting languages. Sections 5 and 6 provide, respectively, soundness and completeness theorems for our operational semantics for simulation scripting languages.

2. Impact of Errors in Simulation Scripts

In this section, we provide an estimation of the impact errors in simulation scripts may have on the reliability of the whole verification activity.

A simulation campaign consists of a set of operational scenarios to be simulated. As discussed in Section 1.1 a typical approach is to optimise simulation of such scenarios by avoiding simulating twice the same sequence of actions coming from the external environment. An error in the simulation script driving the simulation campaign may have two main consequences. First, we may simulate more scenarios than we actually need to. Second, we may omit simulation of some of the scenarios we are supposed to simulate. The first case is harmless from a verification standpoint, whereas the second one may lead to accept as correct a system that is not, because the simulation of *all* scenarios witnessing the error is omitted.

Hence, the first kind of script error has no impact on the reliability of the verification activity, it only decreases its efficiency. On the other hand, the second kind of script error

invalidates the verification activity, since we may conclude that there is no operational scenario violating our requirements just because we omitted simulation of *all* scenarios witnessing the error.

To quantify the impact of omitting the simulation of scenarios containing an error, let n be the number of simulation scenarios to be simulated and let p be the fraction of operational scenarios witnessing a violation in the specifications. This means that there are pn operational scenarios witnessing an error.

Suppose that our simulation script instead of simulating all n scenarios omits simulation of a fraction α of them. The probability that we accept as correct a system that has an error is then the probability $OP(n, p, \alpha)$ that all pn error scenarios are among the αn omitted scenarios. In our setting we may assume that p is small since typically formal verification starts when the *easy to find* errors have already been fixed. Accordingly we assume $p \leq \alpha$.

Assuming all distributions are uniform, we can compute the probability OP as follows. First, there are $\binom{n}{\alpha n}$ ways of selecting the αn scenarios to be omitted. Second, we note that if a set of omitted scenarios contains the pn error ones, then it will contain $\alpha n - pn = (\alpha - p)n$ scenarios not leading to a violation in the requirements. Thus the number of sets of omitted scenarios each of which contains all error scenarios is the number of ways of selecting $(\alpha - p)n$ scenarios from those not leading to a violation in the requirements, that is $\binom{(1-p)n}{(\alpha-p)n}$. Thus $OP(n, p, \alpha)$ is:

$$OP(n, p, \alpha) = \frac{\binom{(1-p)n}{(\alpha-p)n}}{\binom{n}{\alpha n}} \tag{1}$$

In Figures 1 to 4 we show $OP(n, p, \alpha)$ (y-axis) when $n = 10^6$, $p \in \{10^{-3}, 10^{-4}, 10^{-5}, 5 \cdot 10^{-6}\}$ and α (x-axis) is in $[0, 1]$. From these figures, we see that the harder the verification task (*i.e.*, the smaller is p), the greater the impact of an error in the simulation script. For example, from Figure 4 we see that $OP(10^6, 5 \cdot 10^{-6}, 0.3) \geq 0.001$, typically an unacceptably high value in a setting where formal verification is called for.

The impact of omitting the simulation of scenarios containing an error motivates our proposal of an operational semantics for simulation scripts.

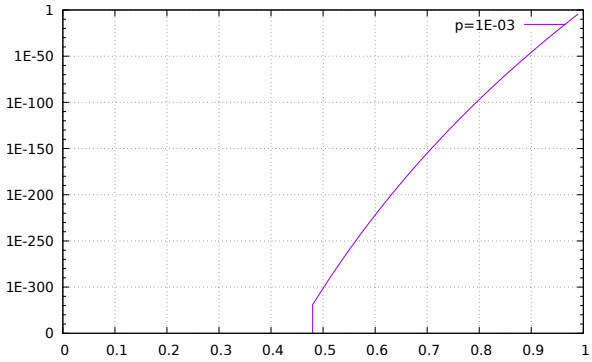


Figure 1: Probability of accepting a wrong system (y-axis) when $p = 10^{-3}$ (α on x-axis).

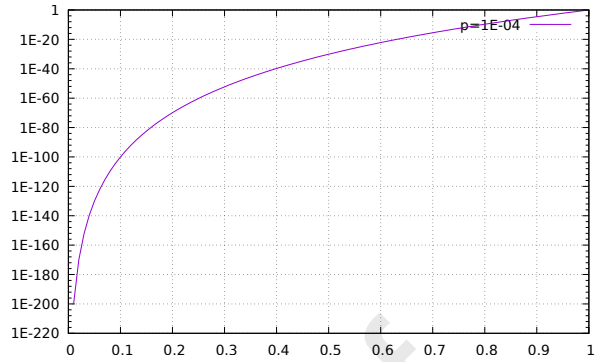


Figure 2: Probability of accepting a wrong system (y-axis) when $p = 10^{-4}$ (α on x-axis).

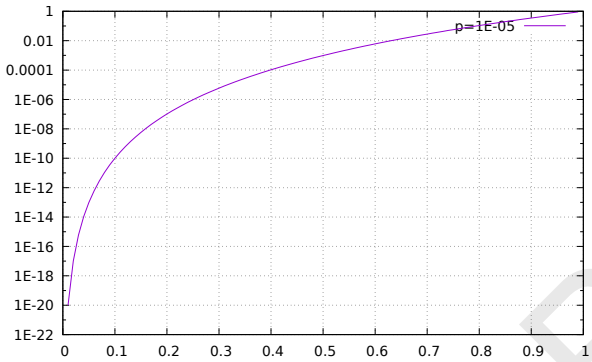


Figure 3: Probability of accepting a wrong system (y-axis) when $p = 10^{-5}$ (α on x-axis).

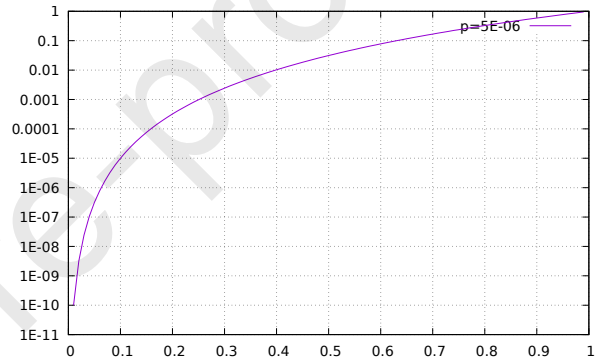


Figure 4: Probability of accepting a wrong system (y-axis) when $p = 5 \cdot 10^{-6}$ (α on x-axis).

3. Dynamical Systems

In this section we give the formal background on which our approach rests. To this end, we model the disturbances (Definition 3) acting on our system, formalised in Definition 7, resting on the notion of dynamical system (see, e.g., [59]). Then, we define the notion of *simulation scenario* (Definition 8), that is the sequence of disturbances occurring from a given system state, and the set of transitions associated to a simulation scenario (Definition 9).

Throughout the paper, we denote with $\mathbf{N}$ the set of natural numbers, $\mathbf{N}^+$ the set of positive natural numbers, $\mathbf{R}_+$, $\mathbf{R}_{0+}$ and $\mathbf{R}$ the sets of positive, non-negative and all real numbers, respectively. We use $\mathbf{R}_{0+}$ to represent time and $\mathbf{R}_+$ to represent non-zero time durations.

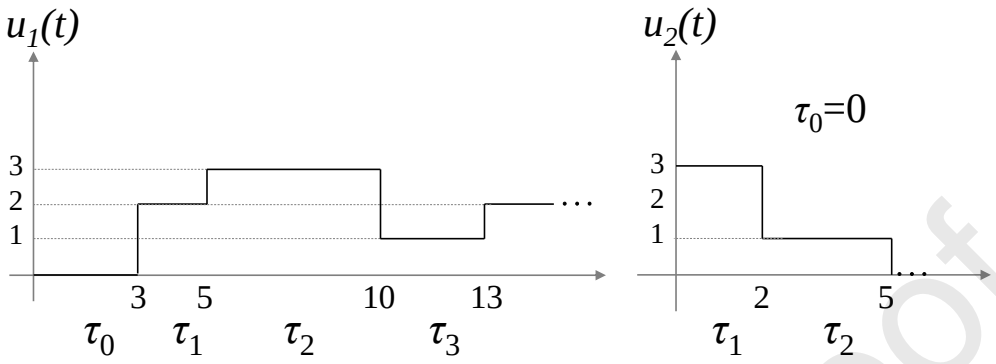


Figure 5: Two discrete event sequences.

Operational scenario for the SUV can be modelled as *uncontrollable inputs* (*disturbances*). Examples of disturbances are: noise on sensors, faults, variations in system parameters, etc. As in [38] we can finitely parametrize the set of continuous time functions defining the operational scenarios our SUV is expected to withstand. For example, a sinusoidal noise with amplitude A and frequency f can be defined by the function $A \sin 2\pi f t$. A set of pairs $(A,\ f)$ defines a set of possible sinusoidal disturbances. Indeed, using Fourier series, any function we may be interested in (finite bandwidth) can be represented, with arbitrary precision, with a finite set of parameters. In other words, the uncontrollable inputs define the function parameters whereas function itself are implemented, as in [38], inside the simulator. This allows us to consider finite the set of values for our disturbances.

A *discrete event sequence* (Definition 3 and Figure 5), is a function associating to each (continuous) time instant a disturbance event (such as a fault, a variation in a system parameter, etc). We consider a bounded time horizon, and accordingly we require that the number of disturbances is finite, since no system can withstand an infinite number of disturbances within a finite time.

Notation 1. *We denote with H the (Heaviside) function $H : \mathbf{R} \to \mathbf{R}$ defined as follows:*

$H(t) = \textbf{\textit{if}}\ \ (t \geq 0)\ \textbf{\textit{then}}\ 1\ \textbf{\textit{else}}\ 0.$

Definition 1 (Step function). *A function $u : \mathbf{R}_{0+} \to \mathbf{R}^n$ is said to be a step function if it can be written as: $u(t) = \sum_{i=0}^{k-1} a_i H(t - \sum_{j=0}^{i} \tau_j)$ for some $k \in \mathbf{N}^+$, $a_i \in \mathbf{R}^n$, $\tau_0 \in \mathbf{R}_{0+}$ and $\tau_i \in \mathbf{R}_+$ $\forall i > 0$. We say that a step function u is in canonical form if: $\forall i > 0$ $[a_i \neq 0]$.*

Remark 1 (Uniqueness). *The request that $\forall i > 0$ $[a_i \neq 0]$ and the condition that $\forall i > 0$ $[\tau_i \in \mathbf{R}_+]$ guarantee that the canonical form of a step function is unique.*

Definition 2 (Time horizon). *We call* time horizon *of u the value $\sum_{j=0}^{k-1} \tau_j$.*

Definition 3 (Discrete event sequence). *Let $\mathcal{U}$ be a finite subset of $\mathbf{R}^n$. A* discrete event sequence *over $\mathcal{U}$ is a step function u such that $\forall t \in \mathbf{R}_{0+}$ $u(t) \in \mathcal{U}$. We denote with $\mathcal{U}^{\mathbf{R}^{\geq 0}}$ the set of discrete event sequences over $\mathcal{U}$.*

Example 1. *Let us consider the two discrete event sequences u_1 and u_2 represented in Figure 5. The function u_1 can be written as: $u_1(t) = 2H(t-3) + H(t-5) - 2H(t-10) + H(t-13)$, whereas the function u_2 can be written as $u_2(t) = 3H(t) - 2H(t-2) - H(t-5)$.*

An explicit representation of a discrete event sequence, denoted as *event list*, can be obtained by listing pairs (τ, e), where e is an event and τ is a time interval. This is formalised in Definition 4.

Definition 4 (Event list). *Let $u(t)$ be as in Definition 1. Then u can be represented with a finite sequence of pairs,* event list*, defined as follows: $[(\tau_0, b_0), (\tau_1, b_1), (\tau_2, b_2), \ldots, (\tau_{k-1}, b_{k-1})]$, where $b_j = \sum_{i=0}^{j} a_i$, and τ_j is the time elapsed since the event immediately preceding event b_j, $j = 0, \ldots, k-1$.*

Definition 4 is well posed since, given the event list for u, we can compute the coefficients of u as $a_i = b_i - b_{i-1}$, with the convention that $b_{-1} = 0$. From now on, we will use the event list to define a discrete event sequence, and write $u = [(\tau_0, b_0), (\tau_1, b_1), (\tau_2, b_2), \ldots, (\tau_{k-1}, b_{k-1})]$ for $u(t) = \sum_{i=0}^{k-1} (b_i - b_{i-1}) H(t - \sum_{j=0}^{i} \tau_j)$.

Example 2. *Let us consider the discrete event sequences u_1 and u_2 in Example 1, and shown in Figure 5. The event list representing u_1 is:* $[(3,2), (2,3), (5,1), (3,2)]$*, whereas the event list representing u_2 is* $[(0,3), (2,1), (3,0)]$.

Example 3 shows that a discrete event sequence can be represented by different event lists.

Example 3. *Let us consider the two functions:* $u = [(0,0), (0.04,1), (0.2,0)]$ *and* $v = [(0,0), (0.04,1), (0.08,1), (0.12,0)]$. *We have that $u = v$ and u is in canonical form.*

The restriction of an event sequence to a finite interval is described in Definition 5.

Definition 5 (Restriction). *Let $\mathcal{U}^{\mathbf{R}^{\geq 0}}$ be the set of discrete event sequences over the set $\mathcal{U}$. Given a discrete event sequence $u \in \mathcal{U}^{\mathbf{R}^{\geq 0}}$ and two positive real numbers $t_1 \leq t_2$, we denote with $u\,|_{[t_1,t_2)}$ the restriction of u to the interval $[t_1, t_2)$, i.e. the function $u\,|_{[t_1,t_2)}\colon [t_1,t_2) \to \mathcal{U}$, such that $u\,|_{[t_1,t_2)}\,(t) = u(t)$ for all $t \in [t_1,t_2)$. We denote $\mathcal{U}^{[t_1,t_2)}$ the* restriction *of $\mathcal{U}^{\mathbf{R}^{\geq 0}}$ to the domain $[t_1,t_2)$.*

The concatenation of event sequences is described in Definition 6.

Definition 6 (Concatenation). *Assume that $t_1, t_2, t_3 \in \mathbf{R}_{0+}$ such that $t_1 < t_2 < t_3$. If $\omega \in \mathcal{U}^{[t_1,t_2)}$ and $\omega' \in \mathcal{U}^{[t_2,t_3)}$, their* concatenation*, denoted as $\omega\omega'$, is the function $\tilde{\omega} \in \mathcal{U}^{[t_1,t_3)}$ defined as:*

$$\tilde{\omega}(t) = \begin{cases} \omega(t) & \text{if } t \in [t_1,t_2) \\ \omega'(t) & \text{if } t \in [t_2,t_3) \end{cases}$$

In our setting the system to be verified can be modelled as a continuous time *Input-State-Output* deterministic dynamical system (see e.g. [59]) 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.

In our setting discrete event systems (Definition 7), model hybrid systems describing cyber-physical system, as shown in Examples 4 to 6. For this reason, here we denote systems with $\mathcal{H}$.

Definition 7 (Discrete Event System). *A* Discrete Event System*, or simply DES, $\mathcal{H}$ is a tuple $(\mathcal{X}, \mathcal{U}, \mathcal{Y}, \varphi, \psi)$, where:*

- $\mathcal{X}$*, the* state space *of $\mathcal{H}$, is a non-empty set whose elements denote* states*;*
- $\mathcal{U}$*, the* input value space *of $\mathcal{H}$, is a finite subset of $\mathbf{R}^n$;*
- $\mathcal{Y}$*, the* output value space *of $\mathcal{H}$, is a non-empty set whose elements denote* outputs*;*
- $\varphi : \mathbf{R}_+ \times \mathbf{R}_+ \times \mathcal{X} \times \mathcal{U}^{\mathbf{R}^{\geq 0}} \to \mathcal{X}$ *is the* transition map *of $\mathcal{H}$. Function φ must satisfy the following properties:*
  - semigroup*: for each $t_1, t_2, t_3 \in \mathbf{R}_{0+}$ such that $t_1 < t_2 < t_3$, $\omega \in \mathcal{U}^{[t_1,t_2)}$, $\omega' \in \mathcal{U}^{[t_2,t_3)}$, $x \in \mathcal{X}$ we have that $\omega\omega' \in \mathcal{U}^{[t_1,t_3)}$ is such that $\varphi(t_3, t_1, x, \omega\omega') = \varphi(t_3, t_2, \varphi(t_2, t_1, x, \omega), \omega')$;*
  - consistency*: for each $u \in \mathcal{U}^{\mathbf{R}^{\geq 0}}$, $x \in \mathcal{X}$, $t \in \mathbf{R}_+$ we have $\varphi(t, t, x, u) = x$;*
- $\psi : \mathbf{R}_{0+} \times \mathcal{X} \to \mathcal{Y}$ *is the* observation function *of $\mathcal{H}$.*

A system is set to be stationary if for all t_1, t_2, τ, x, u, it holds $\varphi(t_1 + \tau, t_1, x, u) = \varphi(t_2 + \tau, t_2, x, u)$. In the following we will focus on stationary systems, and we will also write $\varphi(t, x, u)$ for $\varphi(t, 0, x, u)$.

In the following, unless otherwise specified, $\mathcal{H}$ stands for $\mathcal{H} = (\mathcal{X}, \mathcal{U}, \mathcal{Y}, \varphi, \psi)$.

Note that any simulator driven by its scripting language (consisting of the core functions outlined in Section 1.1) can be seen as a discrete event system. This is why we focus on DES.

Remark 2 (Simulator and DES). *Let $\rho(a, \theta, \tau, t)$ be the constant function returning value a and defined in the interval $[\theta, \theta+\tau)$. Then, the step function $u(t) = \sum_{i=0}^{k-1} a_i H(t - \sum_{j=0}^{i} \tau_j)$ can be written as the concatenation of ρ functions as follows: $u = \rho(c_0, \theta_0, \tau_0)\rho(c_1, \theta_1, \tau_1)$ $\rho(c_2, \theta_2, \tau_2)\rho(c_3, \theta_3, \tau_3) \ldots \rho(c_{k-1}, \theta_{k-1}, \tau_{k-1})\rho(c_k, \theta_k, +\infty) = \prod_{i=0}^{k} \rho(c_i, \theta_i, \tau_i)$, where: $c_0 = 0$, $\theta_0 = 0$, $\tau_k = +\infty$ and, for $j \in \{1, \ldots k\}$, $\theta_j = \theta_{j-1} + \tau_{j-1} = \sum_{i=0}^{j-1} \tau_i$, $c_j = c_{j-1} + a_{j-1} = \sum_{i=0}^{j-1} a_i$.*

Let $\mathcal{H}$ be a DES. Then by the semigroup properties (Definition 7) follows: $\varphi(t, 0, x_0, u) = \varphi(t, 0, x_0, \prod_{i=0}^{k} \rho(c_i, \theta_i, \tau_i)) = x_k$, where: for $j \in \{0, \ldots, k-1\}$, $x_{j+1} = \varphi(\theta_j + \tau_j, \theta_j, x_j, \rho(c_j, \theta_j, \tau_j))$.

Note that $\varphi(\theta_j + \tau_j, \theta_j, x_j, \rho(c_j, \theta_j, \tau_j))$ computes the state reached by $\mathcal{H}$ when starting from state x_j and applying for τ_j time units the constant input c_j. Taking into account that a simulator for $\mathcal{H}$ actually computes φ from the system definition (e.g., through Differential Algebraic Equations (DAEs)) from the above follows that the semigroup property implies that we can simulate the effect of any step function by running consecutive simulations with constant input. This is completely independent from the integration algorithm used by simulator solver. In particular, the simulation length τ_j defining the time between consecutive disturbances does not depend on the simulator solver time step. Actually it is the opposite, the solver time step will have to adapt to the time events *triggered by disturbances (e.g., see [60]).*

Our approach can model both the case in which the input is controllable, for example by control software (Example 4), and the case in which the input is uncontrollable, for example disturbances such as faults are injected (Examples 5 and 6).

Example 4 (Inverted Pendulum). *A simple example of a system is given by the Inverted Pendulum with Stationary Pivot Point, see e.g. [61, 62, 63]. The system is modelled by taking the angle θ and the angular velocity $\dot{\theta}$ as state variables. The input of the system is the torquing force u, that can influence the velocity in both directions. Moreover, the behaviour of the system depends on the pendulum mass m, the length of the pendulum l and the gravitational acceleration g. Given such parameters, the motion of the system is described by the differential equation $\ddot{\theta} = \frac{g}{l} sin\theta + \frac{1}{ml^2} u$.*

Let $\mathcal{U}$ be $\{-1, 0, 1\}$, and $\tau = 10^{-6}$. Our discrete event system $\mathcal{H}$ is the tuple $(\mathcal{X}, \mathcal{U}, \mathcal{Y}, \varphi, \psi)$, where:

- *$\mathcal{X} = \mathbf{R}^2$ and $\mathcal{Y} = \mathbf{R}^2$;*
- *φ is solution to the system of differential equations:*

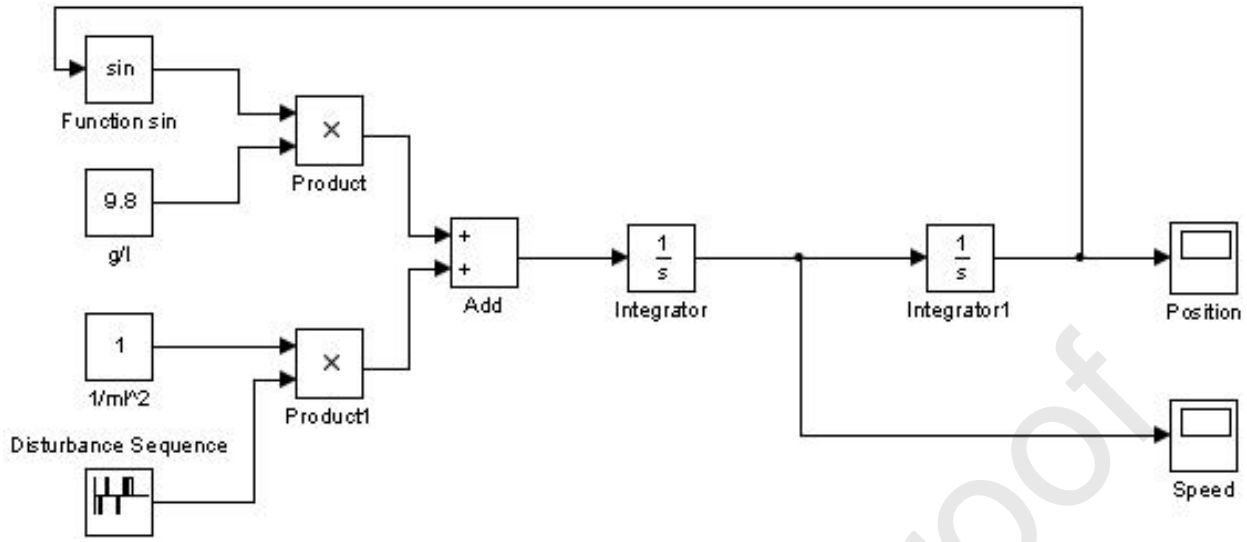


Figure 6: Simulink model of the inverted pendulum (from *mathworks.com*).

$\dot{x_1} = x_2$

$\dot{x_2} = \frac{g}{l} sin x_1 + \frac{1}{ml^2} u$

where x_1 *is the angle* θ *and* x_2 *is the angular velocity* $\dot{\theta}$*;*

- ψ *is given by* $[x_1(t), x_2(t)]$.

In Figure 6 the Simulink model of the inverted pendulum is shown, where we assume the pendulum mass $m = 1$ *and the length of the pendulum* $l = 1$*. Also we assume the function* u *is given to the model as a sequence of values in the set* $\{-1, 0, 1\}$.

Example 5 (Inverted Pendulum on Cart). *Another example of a system is given by the Inverted Pendulum on Cart (IPC). For this system, the control input is the force* F *that moves the cart horizontally and the outputs are the angular position of the pendulum* θ *and the horizontal position of the cart* x*. The physical constraint between the cart and pendulum gives that both the cart and the pendulum have one degree of freedom each (*x *and* θ*, respectively). The controlled system (the* plant*) consists of the cart and the pendulum, whereas the controller consists of the control software computing* F *from the plant outputs (*x *and* θ*). The dynamics of the system is described in the example available in the Simulink*

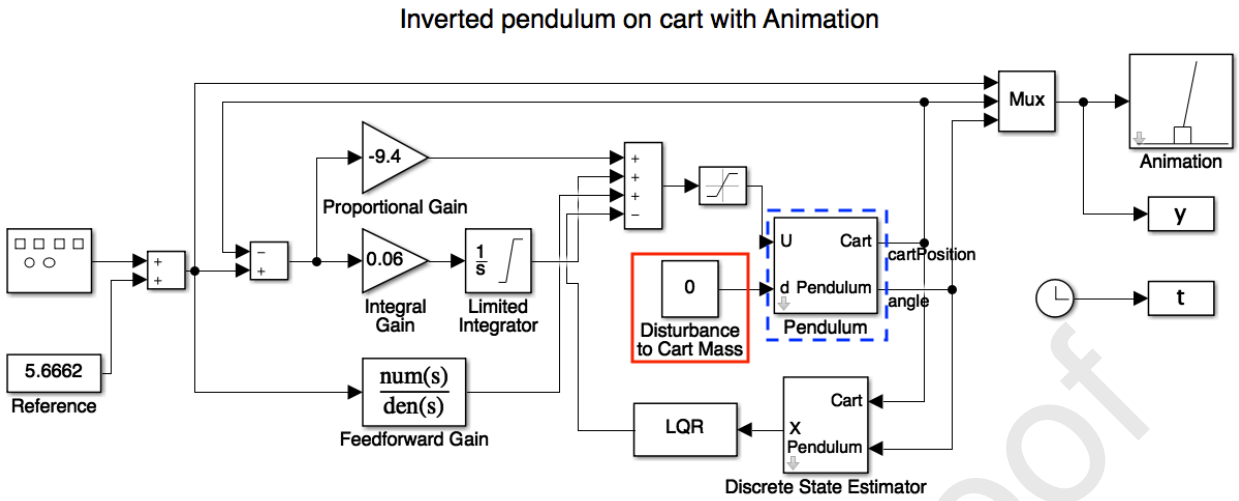


Figure 7: *Main diagram* of Inverted Pendulum on Cart of the Simulink distribution (*mathworks.com*) modified for accepting events. Variations with respect to the original mathworks.com model are highlighted with boxes.

distribution. The Simulink model of the IPC and the pendulum model where disturbances are added, are shown in Figures 7 and 8, respectively.

The system state is a pair (z, w) *where* z *is the state of the control software, and* w *is the plant state. Namely,* $w = [w_1, w_2, w_3, w_4]$*, where:* w_1 *is the cart position,* w_2 *is the cart velocity ,* w_3 *is the pendulum angle,* w_4 *is the pendulum angular velocity.*

We model irregularities in the cart rail injecting disturbances on the cart weight with respect to its nominal value 0.455 kg. We assume that $\mathcal{U} = \{0, 1, 2\}$ *is our set of disturbances (see Definition 3) and model a change in the cart rail giving the cart weight as* $(d + 0.455)$*, with* $d \in \mathcal{U}$*. The Simulink block containing such a disturbance is highlighted in red box of Figure 7. Disturbance on cart mass is then passed as an input to pendulum block, modified with respect to the original version to accept this extra parameter* d*. Modified pendulum block is highlighted in dashed blue box and is shown in Figure 8.*

We will use the inverted pendulum on cart (Example 5) as running example throughout the paper.

Example 6 (Fuel Control System). *The Fuel Control System (FCS) model in the Simulink distribution (see Figure 9) has been studied in [42] using statistical model checking*

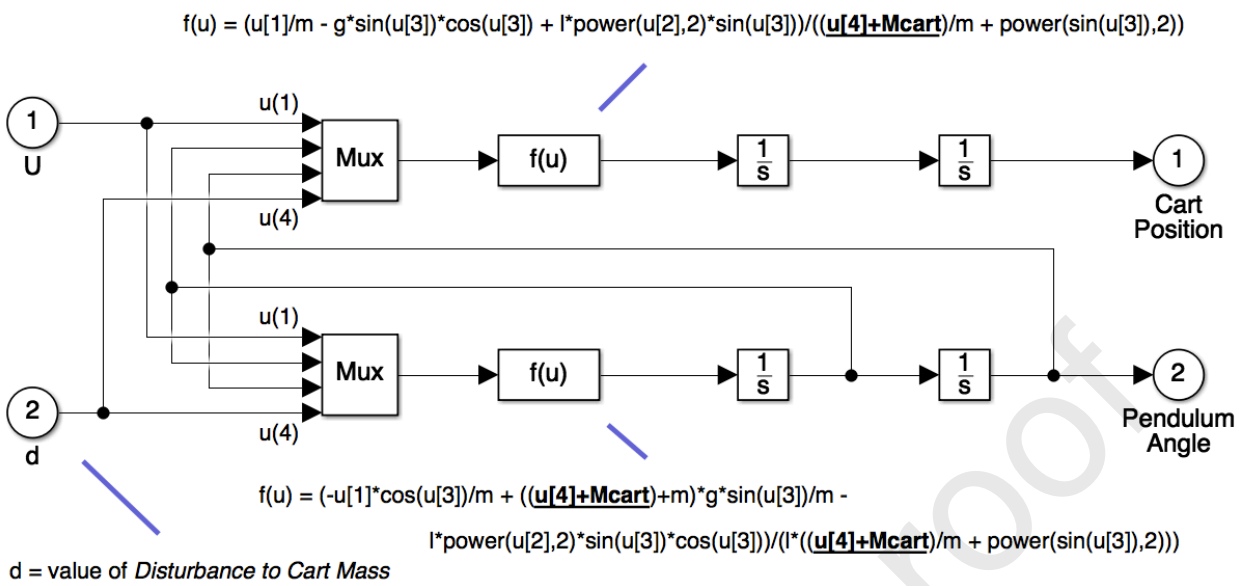


Figure 8: *Pendulum block* of Inverted Pendulum on Cart of the Simulink distribution (*mathworks.com*) modified for accepting events. Differently from original mathworks.com model, this version has one more

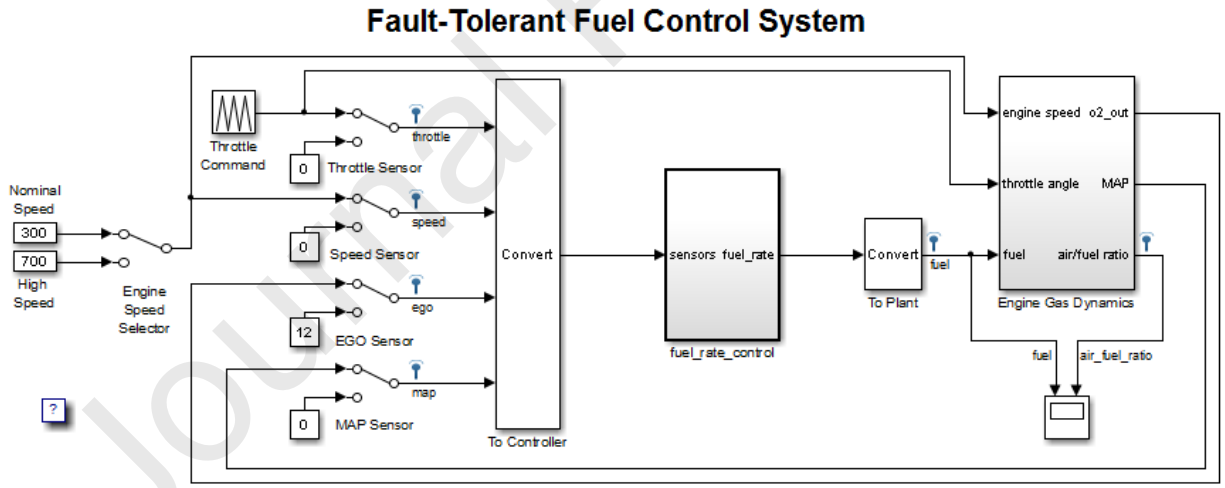


Figure 9: The Simulink Fuel Control System (from *mathworks.com*).

techniques, whereas the formal verification has been discussed in [16, 18]. The model is equipped with four sensors: throttle angle, speed, oxygen in exhaust gas (EGO) and manifold absolute pressure (MAP). For this model, the elements of the tuple $(\mathcal{X}, \mathcal{U}, \mathcal{Y}, \varphi, \psi)$*, representing the discrete event system, are defined as follows:*

- $\mathcal{X}$ *is the set of plant (i.e., the engine) states along with the control software states;*
- $\mathcal{Y}$ *is the set of plant outputs monitored by the control software;*
- $\mathcal{U}$ *is the set of disturbance sequences that may be obtained assuming that only sensors EGO and MAP can fail, giving rise to disturbances* d_1 *and* d_2*, respectively; the minimum time between consecutive faults is one second and all faults are transient, that is disturbance* d_1 *models a fault on sensor EGO, followed by a repair within one second, and disturbance* d_2 *models a fault on sensor MAP, followed by a repair within one second;*
- φ *computes the dynamics of the system states;*
- $\psi(t)$ *computes the system output from the present system state.*

Example 7 (Apollo Digital Autopilot). *An example where disturbances are continuous is the Apollo Digital Autopilot (ADAP) model (Figures 10 and 11). The Simulink ADAP (Figure 10) is equipped with three sensors (i.e., yaw, roll, and pitch jets) and two actuators (i.e., yaw, PR–pitch and roll jets). In order to inject continuous disturbances to a sensor* s*, we modify* s *output through a novel* noise block *(Figure 11) which perturbs the sensor signal as follows. Parameters of a noise block* b_s *(*$s \in \{$*yaw, roll, pitch*$\}$*) are amplitude and frequency of two sinusoidal signals, one representing noise of communication channel (ampl.* $a_{n,s}$*, freq.* $f_{n,s}$*) and another one representing the disturbance itself (ampl.* $a_{d,s}$*, freq.* $f_{d,s}$*). As a consequence, for each sensor* $s \in \{$*yaw, roll, pitch*$\}$*, given the sensor measure at time* t*, denoted* $i_s(t)$*, the sensor measure output disturbed by block* b_s *at time* t*, denoted* $o_s(t)$*, is given by the expression:* $o_s(t) = i_s(t) + a_{n,s} \sin(f_{n,s}t + \pi/2) + a_{d,s} \sin(f_{d,s}t + \pi/2)$.

Similarly to Fuel Control System (FCS), for this model the elements of the tuple $(\mathcal{X}, \mathcal{U}, \mathcal{Y}, \varphi, \psi)$*, representing the discrete event system, are defined as follows:*

- $\mathcal{X}$ *is the set of plant (i.e., yaw, roll and pitch) states along with the control software states;*
- $\mathcal{Y}$ *is the set of plant outputs monitored by the control software;*
- $\mathcal{U}$ *is the set of disturbance sequences that may be obtained assuming that all sensors can fail. For example, disturbances* $d_1, \ldots, d_6$ *may correspond, respectively, to set noise block disturbance amplitude as follows:* $a_{d,\text{yaw}} = 0.01$, $a_{d,\text{yaw}} = 0.02$, $a_{d,\text{yaw}} = 0.03$, $a_{d,\text{roll}} = 0.02$, $a_{d,\text{pitch}} = 0.01$, $a_{d,\text{pitch}} = 0.02$*). The minimum time between consecutive faults is one second and all faults are transient, that is each fault is followed by a repair within one second;*
- φ *computes the dynamics of the system states;*
- $\psi(t)$ *computes the system output from the present system state.*

In the following we define the notion of simulation scenario, that is the sequence of disturbances received by our system starting from a given initial state, and we give an example.

Definition 8 (Simulation scenario). *A* simulation scenario *for* $\mathcal{H}$ *is a pair* (x, u) *where* $x \in \mathcal{X}$ *and* $u \in \mathcal{U}^{\mathbf{R}^{\geq 0}}$.

Example 8 (Simulation scenario). *Let* $\mathcal{H}$ *be the IPC system described in Example 5. Let* u *be the discrete event sequence defined as:* $u = [(0.04, 1), (0.08, 0)]$ *and let the initial state be* $x_0 = (z_0, [0, 0, 0, 0])$*, where* z_0 *is the control software initial state. Then, a simulation scenario for* $\mathcal{H}$ *is* (x_0, u).

Definitions 9 and 10 give the definition of sequence of transitions and set of transitions explored by a SUV under a given simulation scenario, respectively.

Definition 9 (Trace of a simulation scenario). *Let us consider a DES* $\mathcal{H}$*, a state* $x \in \mathcal{X}$*, and a discrete event sequence* $u = [(\tau_0, b_0), (\tau_1, b_1), (\tau_2, b_2), \ldots, (\tau_{k-1}, b_{k-1})] \in \mathcal{U}^{\mathbf{R}^{\geq 0}}$

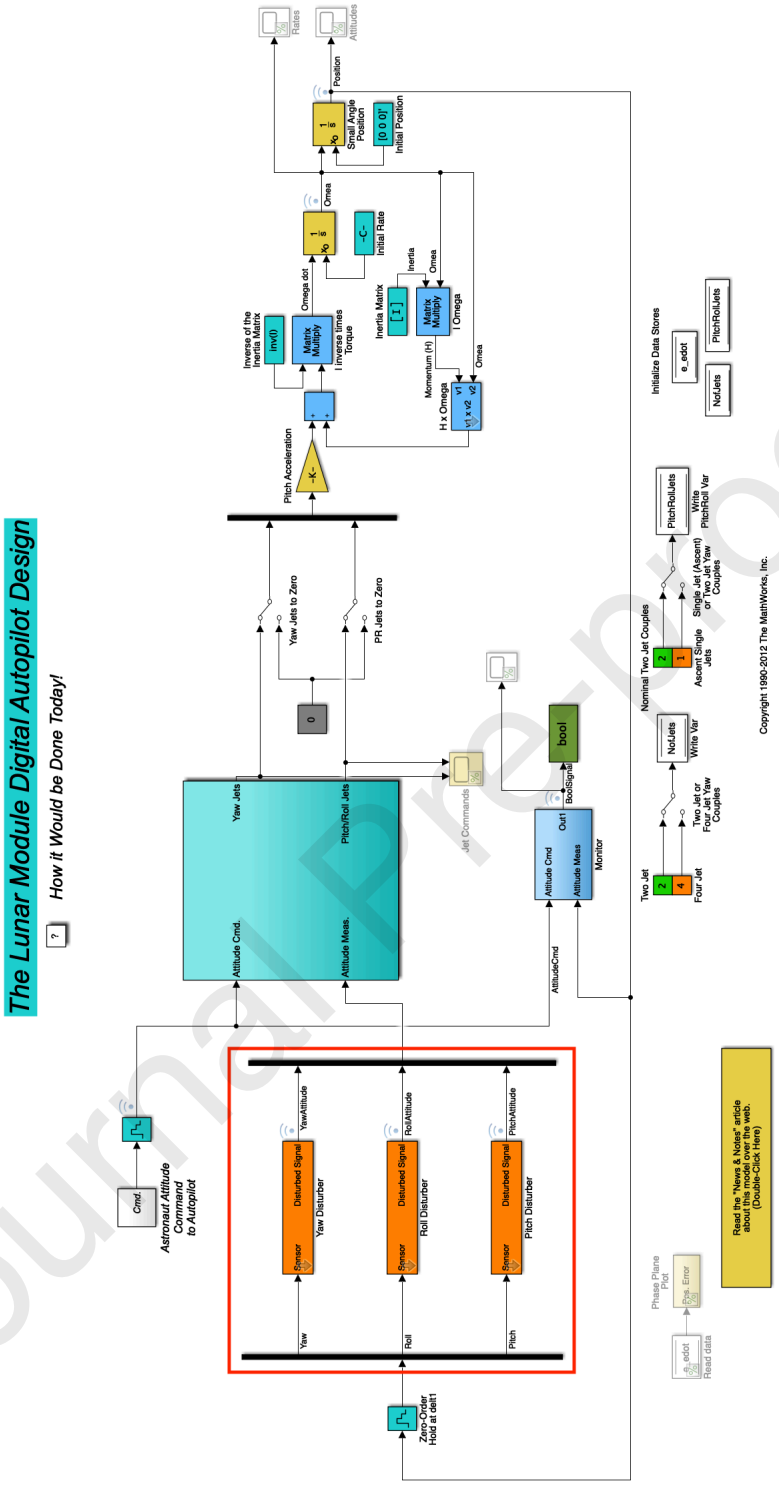


Figure 10: The Simulink Apollo Digital Autopilot (ADAP) (*mathworks.com*) modified to accept continuous noise. Variation with respect to the original model are the novel *noise blocks*, highlighted with the red box on the left side.

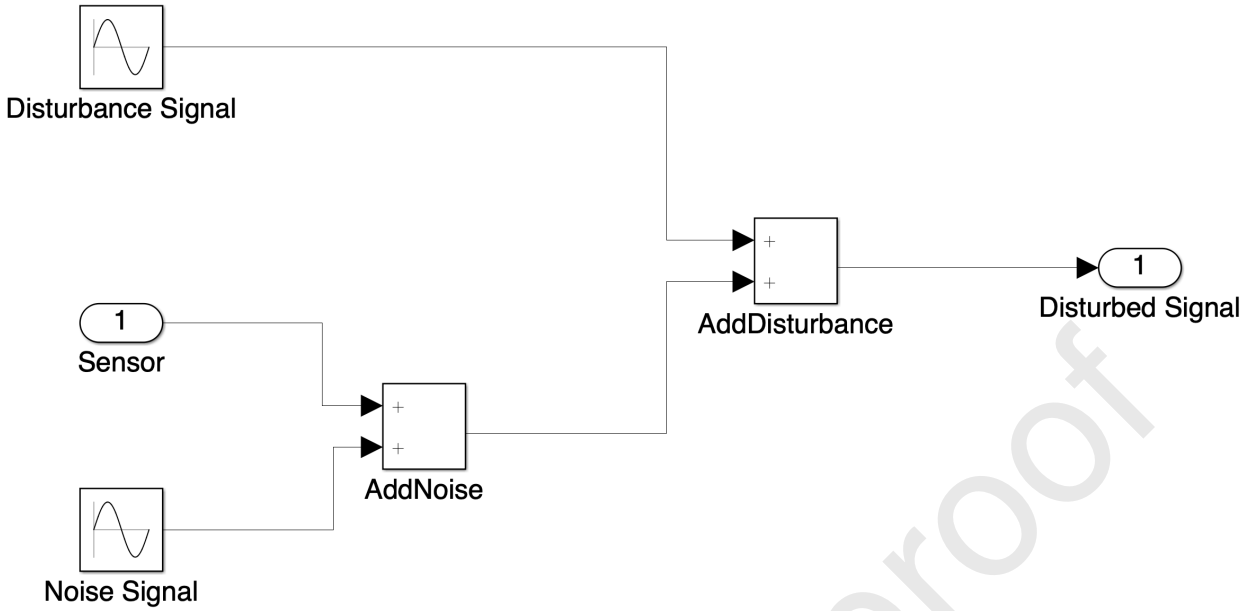


Figure 11: *Noise block* used in the Apollo Digital Autopilot (ADAP) of Figure 10 for disturbing signals related to yaw, roll, and pitch sensors.

giving a simulation scenario (x, u) *for* $\mathcal{H}$*, and let* $[(\tau_0, b_0), (\tau_1, b_1), (\tau_2, b_2), \ldots, (\tau_{k-1}, b_{k-1})]$ *be the event list representing* $u(t)$*. The* trace *of the simulation scenario* (x, u)*, denoted* $Tr(x, u)$*, is the finite sequence of transitions* $Tr(x, u) = [(x_0, 0, \tau_0, x_1), (x_1, b_0, \tau_1, x_2), \ldots, (x_{k-1}, b_{k-2}, \tau_{k-1}, x_k)]$ *such that* $x_0 = x$ *and* $x_{i+1} = \varphi(\tau_i, x_i, b_{i-1}H(t))$*.*

Example 9 (Trace of a simulation scenario). *Let* $\mathcal{H}$ *be the IPC system described in Example 5, and let* (x_0, u) *be the simulation scenario of Example 8.*

The trace of (x_0, u) *is* $Tr(x_0, u) = [(x_0,\ 0,\ 0.4,\ x_1),\ (x_1,\ 1,\ 0.08,\ x_2)]$*, where* $x_0 = (z, [0, 0, 0, 0])$*, and the* x_i *values,* $i = 1, 2$ *are obtained by running the simulation with the Simulink model shown in Example 5, that are:* $x_1 = (z_1, [-0.017, -0.881, 0.057, 2.914])$ *and* $x_2 = (z_2, [-0.072, -0, 431, 0.253, 1.878])$*.*

Definition 10 (Set of transitions of a simulation scenario). *The set of transitions associated to a simulation scenario* (x, u) *is the set:*

$$\mathcal{T}_{(x,u)} = \{(z, b, \tau, z') | (z, b, \tau, z') \in Tr(x, u)\}$$

Example 10 (Set of transitions of a simulation scenario). *Let us consider system $\mathcal{H}$, simulation scenario (x_0, u), and trace $Tr(x_0, u)$ as in Example 8. The set of transitions associated to (x_0, u) is simply the set $\mathcal{T}_{(x,u)} = \{(x_0, 0, 0.04, x_1), (x_1, 1, 0.08, x_2)\}$, where x_1 and x_2 assume the values given in Example 8.*

4. Simulators and Simulation Campaigns

In this section, we formalise the notion of discrete event system simulator (Definition 11 and Definition 12), of simulation campaign (Definition 13) and of set of transitions of a simulation campaign (Definition 15).

In many cases, it is necessary to consider a huge number of simulation scenarios to obtain an exhaustive simulation. The overall number of simulation steps can be prohibitively large if each scenario is simulated from the initial state of the (SUV) simulator. The definition of the set of transitions of a simulation campaign (Definition 15) formalises the above concepts.

Let R be a relation on sets A and B, i.e. $R \subseteq A \times B$. We denote with $\mathrm{Dom}(R)$ (domain of R) the set $\mathrm{Dom}(R) = \{x \in A \mid \exists\, y \in B\ (x, y) \in R\}$.

Definition 11 (Discrete Event System Simulator). *A Discrete Event System (DES) simulator $\mathcal{S}$ is a tuple $(\mathcal{H}, W)$, where $\mathcal{H} = (\mathcal{X}, \mathcal{U}, \mathcal{Y}, \varphi, \psi)$ is a DES and W is a finite set whose elements are called* simulator states*. Each $w \in W$ is a pair (z, M), where $z \in \mathcal{X}$, and M, modelling the content of the* simulator memory*, is a finite set of pairs (id, x) with $x \in \mathcal{X}$ state of $\mathcal{H}$, and id an identifier in $\mathrm{Dom}(M)$, i.e. a state name, such that $[(id, x), (id, x') \in M] \implies x = x'$.*

Note that, at the beginning of the simulation, the simulator memory contains at least the pair (id, x_0), where x_0 is the initial state of $\mathcal{H}$.

Unless otherwise stated, in the following $\mathcal{S}$ is a simulator for the DES $\mathcal{H}$ as in Definition 11.

The semantics of simulator commands we use to execute our simulation scenarios and the transition function ξ are given in Definition 12.

Definition 12 (Simulator commands and transition function). *Let $\mathcal{S}$ be a DES simulator.*

- *The commands for $\mathcal{S}$ are:* load(id), store(id), free(id), run(b, τ), *where id is an identifier of $\mathcal{H}$, $t \in \mathbf{R}^+$ is a time duration, and $b \in \mathcal{U}$ is an event (id, t, b are command arguments). The set of commands is denoted by Λ.*

- *The* transition function ξ *of $\mathcal{S}$ defines how the internal state of the simulator $\mathcal{S}$ changes upon the execution of a command. Namely: $\xi(x, M,$ cmd(args)$) = (x', M')$ when the simulator $\mathcal{S}$ moves from internal state (x, M) to state (x', M') upon processing command* cmd *with arguments* args.

  For each $x \in \mathcal{X}$, function ξ is defined as follows:

  - *if $(id, x') \in M$, then $\xi(x, M, \text{load}(id)) = (x', M)$*
  - *if $(id, x') \in M$, then $\xi(x, M, \text{free}(id)) = (x,\ M \setminus \{(id, x')\})$*
  - *if $id \notin \text{Dom}(M)$, then $\xi(x, M, \text{store}(id)) = (x,\ M \cup \{(id, x)\})$*
  - *$\xi(x, M, \text{run}(b, \tau)) = (x', M)$, where $x' = \varphi(\tau,\ x,\ bH(t))$.*

  Note that ξ is defined only when its preconditions are satisfied.

- *An LTS (Labelled Transition System) for $\mathcal{S}$ is a tuple $(W, \Lambda, \rightarrow)$ such that: W is the set of simulator states, Λ is the set of simulator commands, $\rightarrow$ is a set of labelled transitions (i.e., a subset of $W \times \Lambda \times W$).*

  If $\xi(x, M, \text{cmd(args)}) = (x', M')$ then $\rightarrow$ is defined as $(x, M) \xrightarrow{\text{cmd(args)}})(x', M')$. Namely, for each $x \in \mathcal{X}$, $\rightarrow$ is defined as follows:

  - *if $(id, x') \in M$, then $(x, M) \xrightarrow{\text{load}(id)} (x', M)$*
  - *if $(id, x') \in M$, then $(x, M) \xrightarrow{\text{free}(id)} (x, M \setminus \{(id, x')\})$*
  - *if $id \notin \text{Dom}(M)$, then $(x, M) \xrightarrow{\text{store}(id)} (x, M \cup \{(id, x)\})$*
  - *$(x, M) \xrightarrow{\text{run}(b,\tau)} (x', M)$, where $x' = \varphi(\tau, x, bH(t))$.*

Given a sequence of simulation scenarios, 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 to the simulator states traversed by a simulation campaign. Conversely, given a simulation campaign, we can compute the sequence of scenarios simulated by it. These concepts are formalised in Definition 13.

Definition 13 (Simulation campaign and sequence of simulator states). *Let $\mathcal{S}$ be a simulator and let ξ be its transition function.*

- *A* simulation campaign *for $\mathcal{S}$ is a triple $\Xi = (x, M, \chi)$, where $x \in \mathcal{X}$, $M \subset \mathcal{X}$ and χ is a sequence (possibly empty or infinite) of commands along with their arguments, $\chi = \mathrm{cmd}_0(\mathrm{args}_0), \mathrm{cmd}_1(\mathrm{args}_1), \ldots$. A simulation campaign consisting of a finite sequence of commands is a* finite *simulation campaign or a simulation campaign of finite length.*

- *The* sequence of simulator states *of $\mathcal{S}$ with respect to a simulation campaign $\Xi = (x_0, M_0, \chi)$ is the sequence $(x_0, M_0), (x_1, M_1), \ldots$, where for all j, $(x_j,\ M_j) \xrightarrow{\mathrm{cmd}_j(\mathrm{args}_j)} (x_{j+1}, M_{j+1})$.*

  We denote with $\chi(x_0, M_0, j)$ the j-th element of the simulator state sequence, that is $\chi(x_0, M_0, j) = (x_j, M_j)$. In other words $\chi(x_0, M_0, j)$ is the simulator state after the execution of the j-th command.

- *The set of simulator states with respect to a simulation campaign χ is denoted $\chi(x_0, M_0)$, that is $\chi(x_0, M_0) = \{\chi(x_0, M_0, j) \mid j = 0, 1, \ldots, |\chi| - 1\}$.*

Example 11 (Simulation campaign). *Let $\mathcal{H}$ be the IPC considered in Example 5 and let (x_0, u) be the simulation scenario considered in Example 8, where $u(t) = [(0.04, 1), (0.08, 0)]$.*

The simulation campaign Ξ obtained by using this simulation scenario is the triple $\Xi = (x_0, \{x_0\}, \chi)$, where χ is the sequence of commands $\chi =$ (run(0, 0.04), run(1, 0.08)).

The sequence of simulator states with respect to Ξ is: $(x_0, \{(id, x_0)\}) \xrightarrow{\mathrm{run}(0,0.04)} (x_1, \{(id, x_0)\}) \xrightarrow{\mathrm{run}(1,0.08)} (x_3, \{(id, x_0)\})$.

State values can be obtained by running the simulation.

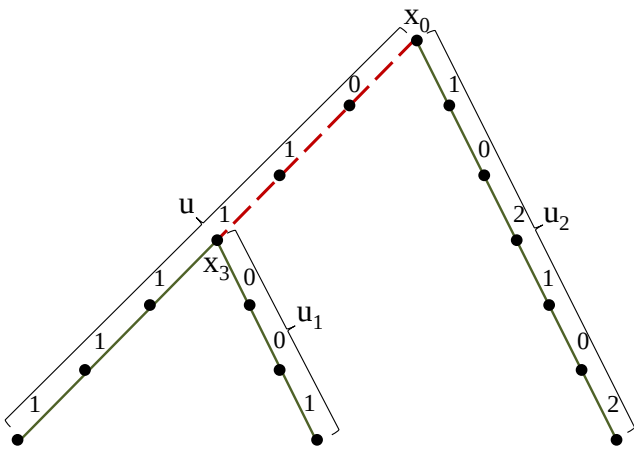


Figure 12: A graphical representation of simulation campaign Ξ_1 (Example 11).

An example of a more complex simulation campaign, Ξ_1*, can be obtained by considering the sequence of simulation scenarios* $((x_0, u), (x_3, u_1), (x_0, u_2))$*, where:*

- $u(t) = [(0.04, 1),\ (0.04, 1),\ (0.04, 1),\ (0.04, 1),\ (0.04,\ 1)]$
- $u_1(t) = [(0.04, 0),\ (0.04, 1)]$
- $u_2(t) = [(0, 1),\ (0.04, 0),\ (0.04, 2),\ (0.04, 1),\ (0.04, 0),\ (0.04, 2)]$.

A graphical representation of simulation campaign Ξ_1 *is shown in Figure 12, where we can see that, according to the considered sequence of simulator states, the discrete event sequences* $u(t)$ *and* $u_2(t)$ *are applied from state* x_0*, and sequence* u_1 *is applied from state* x_3*. Notice that where dashed line represents the event sequence* $u(t)$*.*

The simulation campaign Ξ_1 *obtained by using the sequence of simulation scenarios above is the triple* $\Xi_1 = (x_0, \{x_0\}, \chi_1)$ *, where* χ_1 *is given by the following command sequence:*
$\chi_1 =$ *(*run(0, 0.04), run(1, 0.04), run(1, 0.04), store(x_3), run(1, 0.04), run(1, 0.04), run(1, 0.04), load(x_3), run(0, 0.04), run(0, 0.04), run(1, 0.04), free(x_3), load(x_0), run(1, 0.04), run(0, 0.04), run(2, 0.04), run(1, 0.04), run(0, 0.04), run(2, 0.04)*).*

The sequence of simulator states with respect to Ξ_1 *is:* $(x_0,\ \{(id,\ x_0)\}) \xrightarrow{\text{run(0,0.04)}} (x_1,\ \{(id,\ x_0)\}) \xrightarrow{\text{run(1,0.04)}} (x_2,\ \{(id,\ x_0)\}) \xrightarrow{\text{run(1,0.04)}} (x_3,\ \{(id,\ x_0)\}) \xrightarrow{\text{store(id')}} (x_3,\ \{(id,\ x_0),\ (id',\ x_3)\}) \xrightarrow{\text{run(1,0.04)}} (x_4,\ \{(id,\ x_0),\ (id',\ x_3)\}) \xrightarrow{\text{run(1,0.04)}} (x_5,\ \{(id,\ x_0),\ (id',\ x_3)\}) \xrightarrow{\text{run(1,0.04)}} (x_6,\ \{(id,\ x_0),\ (id',\ x_3)\}) \xrightarrow{\text{load}(id')} (x_3,\ \{(id\ ,x_0),\ (id',\ x_3)\}) \xrightarrow{\text{run(0,0.04)}} (x_7,\ \{(id\ ,x_0),\ (id',\ x_3)\})$

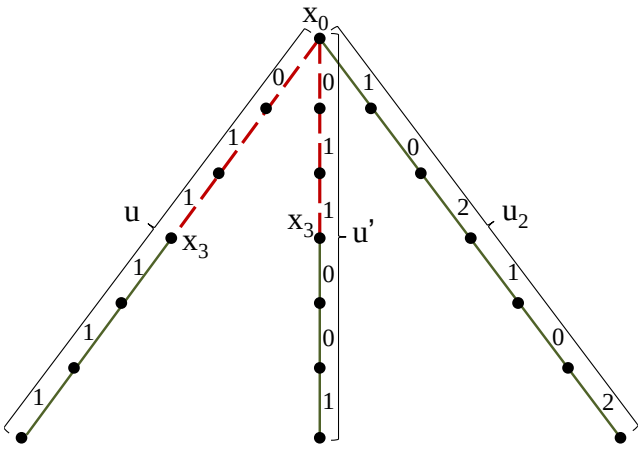


Figure 13: A graphical representation of the normal simulation campaign Ξ_2 (Example 12).

$\xrightarrow{\text{run}(0,0.04)} (x_8, \{(id, x_0), (id', x_3)\}) \xrightarrow{\text{run}(1,0.04)} (x_9, \{(id, x_0), (id', x_3)\}) \xrightarrow{\text{free}(id')} (x_9, \{(id, x_0)\}) \xrightarrow{\text{load}(id)} (x_0, \{(id, x_0)\}) \xrightarrow{\text{run}(1,0.04)} (x_{10}, \{(id, x_0)\}) \xrightarrow{\text{run}(0,0.04)} (x_{11}, \{(id, x_0)\}) \xrightarrow{\text{run}(2,0.04)} (x_{12}, \{(id, x_0)\}) \xrightarrow{\text{run}(1,0.04)} (x_{13}, \{(id, x_0)\}) \xrightarrow{\text{run}(0,0.04)} (x_{14}, \{(id, x_0)\}) \xrightarrow{\text{run}(2,0.04)} (x_{15}, \{(id, x_0)\})$.

Definition 14 gives the notion of normal simulation campaign, that is a simulation campaign for which every simulation scenario starts from an initial state.

Definition 14 (Normal Simulation Campaign). *A simulation campaign is in* normal *form if it consists only of commands* load *and* run.

Example 12 (Normal Simulation Campaign). *An example of simulation campaign in normal form is* $\Xi_2 = (x_0, \{x_0\}, \chi_2)$, *where* χ_2 *is given by the following command sequence:* $\chi_2 = ()\text{run}(0,\ 0.04),\ \text{run}(1,\ 0.04),\ \text{run}(1,\ 0.04),\ \text{run}(1,\ 0.04),\ \text{run}(1,\ 0.04),\ \text{run}(1,\ 0.04),\ \text{load}(x_0),\ \text{run}(0,\ 0.04),\ \text{run}(1, 0.04),\ \text{run}(1,\ 0.04),\ \text{run}(0,,\ 0.04),\ \text{run}(0,\ 0.04),\ \text{run}(1,\ 0.04),\ \text{load}(x_0),\ \text{run}(1,\ 0.04),\ \text{run}(0,\ 0.04),\ \text{run}(2,\ 0.04),\ \text{run}(1,\ 0.04),\ \text{run}(0,\ 0.04),\ \text{run}(2,\ 0.04))$.

This simulation campaign is obtained considering the sequence of simulation scenarios $((x_0, u), (x_0, u'), (x_0, u_2))$, *where* u *and* u_2 *are defined as in Example 11 and* u' *is defined as* $u'(t) = [(0.04, 1), (0.04, 1), (0.04, 0), (0.04, 0), (0.04, 1)]$ *(where the first three steps are the same as* $u(t)$*).*

A graphical representation of Ξ_2 is shown in Figure 13, where the dashed line represents the event sequence $u(t)$.

Note that, since a normal simulation campaign has no store commands, a command load can only load an initial state.

Definition 15, resting on Definition 13, defines the set of transitions of a simulation campaign. Definition 16 gives the notion of equivalent simulation campaigns.

Definition 15 (Set of transitions of a Simulation Campaign). *We denote with $\mathcal{T}_\Xi$ the set of transitions of $\mathcal{S}$ explored by Ξ, that is $\mathcal{T}_\Xi = \{(x, b, \tau, x') \mid \exists\ M, M'\ \ [(x, M),\ (x', M')$ are simulator states of $\mathcal{S}\ \wedge\ \xi(x,\ M,\ \mathrm{run}(b, \tau)) = (x',\ M')]\}$.*

Definition 16 (Equivalent simulation campaigns). *We say that the simulation campaign Ξ is equivalent to Ξ' and we write $\Xi \sim \Xi'$ if $\mathcal{T}_\Xi = \mathcal{T}_{\Xi'}$.*

Example 13 (Equivalent simulation campaigns). *The simulation campaign Ξ_1 in Example 11 and the normal simulation campaign Ξ_2 in Example 12 are equivalent.*

In fact the set of transitions explored by Ξ_1 and Ξ_2 is: $\mathcal{T}_{\Xi_1} = \mathcal{T}_{\Xi_2} = \{\ (x_0,\ 0,\ 0.04,\ x_1),\ (x_1,\ 1,\ 0.04,\ x_2),\ (x_2,\ 1,\ 0.04,\ x_3),\ (x_3, 1,\ 0.04, x_4),\ (x_4, 1,\ 0.04, x_5),\ (x_5, 1,\ 0.04, x_6),\ (x_3, 0,\ 0.04, x_7),\ (x_7, 0,\ 0.04, x_8),\ (x_8, 1,\ 0.04, x_9),\ (x_0, 1,\ 0.04, x_{10}),\ (x_{10}, 0,\ 0.04, x_{11}),\ (x_{11}, 2,\ 0.04, x_{12}),\ (x_{12}, 1,\ 0.04, x_{13}),\ (x_{13}, 0,\ 0.04, x_{14}),\ (x_{14}, 2,\ 0.04, x_{15})\}$.

This can also be seen looking at the set of edges (transitions) in Figures 12 and 13.

Given a simulation campaign, we can obtain an equivalent simulation campaign in which each simulation scenario starts from an initial state. This is formalised in Lemma 1, whose proof idea is given in Example 14.

Lemma 1. *Given a simulation campaign Ξ for a simulator $\mathcal{S}$, there exists a simulation campaign Ξ' such that:*

- *Ξ' is in normal form*
- *$\Xi' \sim \Xi$*

PROOF. We show that, given a simulation campaign Ξ, it is always possible to obtain a simulation campaign Ξ' in normal form equivalent to Ξ. A simulation campaign in normal form consists of load and run commands only. Then, to transform a simulation campaign in normal form we have to eliminate free commands and store commands. free commands are eliminated as follows. First, we refresh all identifiers in Ξ so that each identifier used in a store command does not occur in any of the commands preceding such a store in the simulation campaign. After that we remove all free commands. This does not modify the execution of the simulation campaign, it just uses more memory. So from now on we assume that Ξ does not contain any free command.

To eliminate a store command, we need to substitute every load command (loading the state whose store is eliminated) occurring in the rest of the command sequence, by the sequence of commands providing the stored state. We prove that this is possible by induction on the number of store commands.

Base case: There is no store command in the command sequence of the simulation campaign Ξ.

In this case simulation campaign Ξ is already in normal form, hence $\Xi' = \Xi$.

Inductive step: Assume, by inductive hypothesis, that given the simulation campaign Ξ containing n store commands (and no commands free), there exists an equivalent simulation campaign Ξ' in normal form.

We show that if the simulation campaign Ξ contains $n + 1$ store commands (and no commands free), then there exists an equivalent simulation campaign Ξ' in normal form.

Let store(id) be the first store command occurring in the simulation campaign. Then there exists a unique pair $(id, \tilde{x})$ in the simulator memory M.

In the sequence of commands following the command store(id) there will be zero, one or more commands load(id). Then, to make the elimination of store(id) possible, we must first replace all commands load(id) with a sequence of commands leading to the state $\tilde{x}$ loaded by load(id).

We use the sequence of commands that certainly produces the state $\tilde{x}$, denoted by $seq(\tilde{x})$, that is the sequence of run commands ending immediately before the store(id) command and

starting from the first load command we encounter going back from the store(id) command. Namely, $seq(\tilde{x})$ is the sequence $seq(\tilde{x}) = \text{load}(\hat{id})$, $run(b_0, \tau_1)$, $run(b_1, \tau_2)$, ..., $run(b_{h-1}, \tau_h)$ preceding store(id), where $(\hat{id}, \hat{x}) \in M$. Note that, at the beginning of the simulation the pair $(\hat{id}, \hat{x})$ is stored in the simulator memory M since store(id) is the first store command in the simulation campaign Ξ. We can substitute all commands load(id) by $seq(\tilde{x})$, and then eliminate the store(id) command, thus obtaining the normal form simulation campaign Ξ'.

The obtained simulation campaign Ξ' is equivalent to the original simulation campaign Ξ, that is $Tr(\Xi) = \{(x, b, \tau, x^*) \mid \exists\, M, M^*\ \ [(x, M), (x^*, M^*)$ are simulator states of $\mathcal{S} \wedge \xi(x, M, \text{run}(b, \tau)) = (x^*, M^*)]\} = Tr(\Xi')$.

In fact, the set of run commands in Ξ is the same as the set of run commands in Ξ'. Accordingly, the set $Tr(\Xi')$ of transitions of $\mathcal{S}$ explored by Ξ' is exactly the same as the set $Tr(\Xi)$ of transitions of $\mathcal{S}$ explored by Ξ, since Ξ' is obtained from Ξ by executing the commands of $seq(\tilde{x})$ instead of the load(id) command, thus repeating the execution of run commands (giving the transitions) already executed, hence no new transition is produced.

Remark 3. *Using Lemma 1 we can check equivalence between two simulation campaigns Ξ_1 and Ξ_2. In fact, by Lemma 1, $\Xi_1 \sim \Xi_2$ are equivalent if and only if their normal forms, respectively Ξ_1' and Ξ_2' are equivalent, i.e., $\Xi_1' \sim \Xi_2'$. The latter can be easily verified by checking that Ξ_1' and Ξ_2' define the same set of simulation scenarios, an easy task on normal forms.*

Example 14 clarifies the proof of Lemma 1.

Example 14 (Lemma 1). *Consider the simulation campaign Ξ_1 in Example 11. Ξ_1 is not in normal form. However, by modifying it so that all simulation scenarios (paths on the tree of Figure 12) start from the initial state, we get the normal simulation campaign Ξ_2 illustrated in Example 12. Further, it follows from Example 12 that $\Xi_1 \sim \Xi_2$.*

4.1. Simulation Campaigns and MATLAB Simulink

Tables 1 and 2 show the sequence of simulation campaign commands of Example 11 (resp. Example 12) on the left, and MATLAB results on the right (for the sake of completeness, a complete MATLAB implementation to obtain such results is shown in Appendix A).

Table 1: Execution of Example 12 (file `example11.m`, left column) with output (right column). The corresponding graphical representation is in Figure 13.

```
s_load('x0');
s_run(0, 0.04);
s_run(1, 0.04);
s_run(1, 0.04);
s_run(1, 0.04);
s_run(1, 0.04);
s_run(1, 0.04);
s_load('x0');
s_run(0, 0.04);
s_run(1, 0.04);
s_run(1, 0.04);
s_run(0, 0.04);
s_run(0, 0.04);
s_run(1, 0.04);
s_load('x0');
s_run(1, 0.04);
s_run(0, 0.04);
s_run(2, 0.04);
s_run(1, 0.04);
s_run(0, 0.04);
s_run(2, 0.04);
% Elapsed time
```

```
CurSnapTime: 0.00, State: Sf51dcdc73387f49a04d3197ee7e2922
    % x0
CurSnapTime: 0.04, State: aea17f51af5abed4f0c1eb8672af5095
CurSnapTime: 0.08, State: d101ecd41137346849f016300aad2e47
CurSnapTime: 0.12, State:
    s4026ca9fcd64c22334a565e1185823b % x3
CurSnapTime: 0.16, State: 6373410d983ba0419365944b5a34e2c4
CurSnapTime: 0.20, State: 3c27414a30e81becf83e2369bc782324
CurSnapTime: 0.24, State: 01d6278d0be8e3cfecb7bdab07db4605
CurSnapTime: 0.00, State: Sf51dcdc73387f49a04d3197ee7e2922
    % x0
CurSnapTime: 0.04, State: aea17f51af5abed4f0c1eb8672af5095
CurSnapTime: 0.08, State: d101ecd41137346849f016300aad2e47
CurSnapTime: 0.12, State:
    s4026ca9fcd64c22334a565e1185823b % x3
CurSnapTime: 0.16, State: d4856d2fe92abad92ed46b41cf35aa63
CurSnapTime: 0.20, State: 0bbe28164a01e5618a3530d00aab2991
CurSnapTime: 0.24, State: a0c9d33816228df2541b943f1ce4b8ec
CurSnapTime: 0.00, State: Sf51dcdc73387f49a04d3197ee7e2922
    % x0
CurSnapTime: 0.04, State: 4a0140db9d4664a67bb888cdc046f297
CurSnapTime: 0.08, State: e79acee86fe25505617a2d054305371f
CurSnapTime: 0.12, State: 1b6ca061850702b18342984ad61de140
CurSnapTime: 0.16, State: a24a9314ef723f933c1d9c0892e96703
CurSnapTime: 0.20, State: d4036cd46ce6d858c7f263c89c5f3b8a
CurSnapTime: 0.24, State: 3770506f3a4ba798aa91c532111f6e1c
Elapsed time is 3.020143 seconds.
```

Table 2: Execution of Example 11 (file `example10.m`, left column) with output (right column). The corresponding graphical representation is in Figure 12.

```
s_load('x0');
s_run(0, 0.04);
s_run(1, 0.04);
s_run(1, 0.04);
s_store('x3');
s_run(1, 0.04);
s_run(1, 0.04);
s_run(1, 0.04);
s_load('x3');
s_run(0, 0.04);
s_run(0, 0.04);
s_run(1, 0.04);
s_free('x3');
s_load('x0');
s_run(1, 0.04);
s_run(0, 0.04);
s_run(2, 0.04);
s_run(1, 0.04);
s_run(0, 0.04);
s_run(2, 0.04);
% Elapsed time
```

```
CurSnapTime: 0.00, State: Sf51dcdc73387f49a04d3197ee7e2922
     % x0
CurSnapTime: 0.04, State: aea17f51af5abed4f0c1eb8672af5095
CurSnapTime: 0.08, State: d101ecd41137346849f016300aad2e47
CurSnapTime: 0.12, State:
    s4026ca9fcd64c22334a565e1185823b % x3
CurSnapTime: 0.12, State:
    s4026ca9fcd64c22334a565e1185823b % x3
CurSnapTime: 0.16, State: 6373410d983ba0419365944b5a34e2c4
CurSnapTime: 0.20, State: 3c27414a30e81becf83e2369bc782324
CurSnapTime: 0.24, State: 01d6278d0be8e3cfecb7bdab07db4605
CurSnapTime: 0.12, State:
    s4026ca9fcd64c22334a565e1185823b % x3
CurSnapTime: 0.16, State: d4856d2fe92abad92ed46b41cf35aa63
CurSnapTime: 0.20, State: 0bbe28164a01e5618a3530d00aab2991
CurSnapTime: 0.24, State: a0c9d33816228df2541b943f1ce4b8ec
CurSnapTime: 0.24, State: a0c9d33816228df2541b943f1ce4b8ec
CurSnapTime: 0.00, State: Sf51dcdc73387f49a04d3197ee7e2922
     % x0
CurSnapTime: 0.04, State: 4a0140db9d4664a67bb888cdc046f297
CurSnapTime: 0.08, State: e79acee86fe25505617a2d054305371f
CurSnapTime: 0.12, State: 1b6ca061850702b18342984ad61de140
CurSnapTime: 0.16, State: a24a9314ef723f933c1d9c0892e96703
CurSnapTime: 0.20, State: d4036cd46ce6d858c7f263c89c5f3b8a
CurSnapTime: 0.24, State: 3770506f3a4ba798aa91c532111f6e1c
Elapsed time is 2.609006 seconds.
```

The SUV taken in consideration is the Inverted Pendulum on Cart (Example 5) shown in Figure 7. Disturbances on cart mass (modelling irregularities in the cart rail) are injected simply modifying the value of Simulink block *Disturbance to Cart Mass* to 0, 1 or 2.

Simulation campaign commands *run*, *store*, *load*, and *free* are implemented through MATLAB functions `s_run`, `s_store`, `s_load`, and `s_free` (left columns of Tables 1 and 2).

In order to trace how the simulation proceeds, the right columns of Tables 1 and 2 show, for each function call, the execution time and a hash value of the current simulator state. Note that equal states will yield the same hash values, whereas different states will (very likely) yield different hash values. For the sake of clarity, states are highlighted in right columns of Tables 1 and 2. In particular, state `x0` hash value is highlighted in blue. Occurrences of the common prefix are underlined and highlighted in red in Tables 1 and 2. Storing the final state `x3` reached by such a common prefix, we avoid repeating the computation entailed by the prefix second occurrence.

We see that the unoptimised simulation campaign (Table 1) takes about 3 seconds of computation time whereas the optimised one (Table 2), that saves intermediate states (*i.e.*, the result of simulating common disturbance prefixes), takes about 2.6 seconds, thereby saving about 13% of computation time.

5. Soundness

In this section we show the soundness of our operational semantics for simulation scripting languages. That is, we show that any simulation campaign (simulation script) stems from a set of simulation scenarios. This guarantees that any simulation campaign has indeed a physical (computational) meaning.

Theorem 1 (Soundness). *Given a simulation campaign* Ξ *for* $\mathcal{S}$ *there exists a set* $\mathcal{A} = \{(x_0, u_0),\ (x_1,\ u_1), \ldots,\ (x_{k-1},\ u_{k-1})\}$ *of simulation scenarios such that* $\mathcal{T}_\Xi = \bigcup\{\mathcal{T}_{(x_l,u_l)} | \ l = 0, \ldots, k-1\}$.

Proof. Given a simulation campaign Ξ for a simulator $\mathcal{S}$, there exists a simulation campaign Ξ' in normal form, that is consisting of load and run commands only, equivalent to Ξ

(see Lemma 1).

This means that we consider only commands producing new states x_i.

Then, to our simulation campaign corresponds the sequence of commands χ: load(id_0) run(b_0^0, τ_0^0)... run($b_{h_0}^0$, $\tau_{h_0}^0$) load(id_1) run(b_0^1, τ_0^1)... run($b_{h_1}^1$, $\tau_{h_1}^1$)... load(id_{k-1}) run(b_0^{k-1}, τ_0^{k-1})... run($b_{h_{k-1}}^{k-1}$, $\tau_{h_{k-1}}^{k-1}$).

Let $u_l(t)$, $l = 0, \ldots, k-1$, be defined as in Definition 1, namely $u(t) = \sum_{i=0}^{k-1} a_i H(t - \sum_{j=0}^{i} \tau_j)$.

Note that the set of transitions associated to the sequence of commands load(id_l)run(b_0^l, τ_0^l) ...run($b_{h_l}^l, \tau_{h_l}^l$), representing the simulation scenario (x_l, u_l) is: $\mathcal{T}_{(x_l,u_l)} = \{(x_0^l, b_0^l, \tau_l, x_1^l), \ldots, (x_{h_l-1}^l, b_{h_l-1}^l, \tau_l, x_{h_l}^l)\}$.

Hence, the set of transition associated to the simulation campaign Ξ is:

$$\mathcal{T}_\Xi = \bigcup \{\mathcal{T}_{(x_l,u_l)} |\ l = 0, \ldots, k-1\}.$$

Taking $\mathcal{A} = \{(x_l, u_l) \mid l = 0, \ldots, k-1\}$ completes the proof.

We illustrate the proof by using an example.

Example 15 (Soundness). *Consider the simulation campaign Ξ_1 in Example 11. The set of transitions of Ξ_1, $\mathcal{T}_{\Xi_1}$, is shown in Example 13.*

Now, let us consider the simulation scenarios (x_0, u), (x_3, u_1) and (x_0, u_2), defined in Example 11. Let $\mathcal{A}$ be the set $\mathcal{A} = \{(x_0, u), (x_3, u_1), (x_0, u_2)\}$

The sets of transitions for simulation scenario (x_0, u) is shown in Example 10, and the sets of transitions associated to the other two simulation scenarios are, respectively:

- $\mathcal{T}_1 = \mathcal{T}_{(x_3,u_1)} = \{(x_3, 0, 0.04, x_7), (x_7, 0, 0.04, x_8), (x_8, 1, 0.04, x_9)\}$
- $\mathcal{T}_2 = \mathcal{T}_{(x_0,u_2)} = \{(x_0, 1, 0.04, x_{10}), (x_{10}, 0, 0.04, x_{11}), (x_{11}, 2, 0.04, x_{12}), (x_{12}, 1, 0.04, x_{13}), (x_{13}, 0, 0.04, x_{14}), (x_{14}, 2, 0.04, x_{15})\}$.

It is easy to see that $\mathcal{T}_{\Xi_1} = \mathcal{T}_0 \cup \mathcal{T}_1 \cup \mathcal{T}_2$.

6. Completeness

In this section we show the completeness of our semantics for simulation scripting languages. That is, we show that any set of simulation scenarios can be defined through a

simulation campaign (simulation script). This guarantees that any set of physical experiments can be defined by a suitable simulation campaign.

Theorem 2. *Let $\mathcal{A} = \{(x_0, u_0), (x_1, u_1), \ldots, (x_{k-1}, u_{k-1})\}$ be a set of simulation scenarios of $\mathcal{H}$. Then there exists a simulation campaign Ξ for $\mathcal{S}$ such that $\mathcal{T}_\Xi = \bigcup\{\mathcal{T}_{(x_l,u_l)} |\ \ l = 1, \ldots, k-1\}$.*

Proof. Let us consider a simulation scenario $(x_l, u_l) \in \mathcal{A}$ starting from a state x_l, and let $u_l \in \mathcal{U}^{\mathbf{R}^{\geq 0}}$ be defined as in Definition 1, that is as a step function in the form $u(t) = \sum_{i=0}^{k-1} a_i H(t - \sum_{j=0}^{i} \tau_j)$.

To such a simulation scenario (x_l, u_l) corresponds the sequence of commands: $\chi_l = \text{load}(id_l)\ \text{run}(b_0^l, \tau_0^l) \ldots \text{run}(b_{h_l}^l, \tau_{h_l}^l)$. Also, (x_l, u_l) defines the set of transitions: $\mathcal{T}_{(x_l,u_l)} = \{(x_0^l, b_0^l, \tau_l, x_1^l), \ldots, (x_{h_l-1}^l, b_{h_l-1}^l, \tau_l, x_{h_l}^l)\}$.

By concatenating the sequences of commands χ_l for $l = 0, \ldots, k-1$, corresponding to the simulation scenarios in $\mathcal{A}$, we obtain the whole sequence of commands describing the simulation campaign Ξ.

Hence we obtain that the union of sets of transitions of simulation scenarios in $\mathcal{A}$ is equal to the set of transitions of the simulation campaign Ξ, that is

$$\bigcup\{\mathcal{T}_{(x_l,u_l)} |\ i = 0, \ldots, k-1\} = \mathcal{T}_\Xi.$$

Also for this theorem, we illustrate the proof by using an example.

Example 16 (Completeness). *Let us consider the set $\mathcal{A}$ consisting of simulation scenarios (x_0, u), (x_3, u_1) and (x_0, u_2), defined in Example 11.*

The sets of transitions associated to these simulation scenarios, $\mathcal{T}_0 = \mathcal{T}_{(x_0,u)}$, $\mathcal{T}_1 = \mathcal{T}_{(x_3,u_1)}$ and $\mathcal{T}_2 = \mathcal{T}_{(x_0,u_2)}$, are shown in Example 15. Let $\bar{\mathcal{T}}$ be the set obtained as union of the sets of transitions above, that is $\bar{\mathcal{T}} = \mathcal{T}_0 \cup \mathcal{T}_1 \cup \mathcal{T}_2$.

Now, let us consider the simulation campaign Ξ_1 in Example 11 and the set of transitions of Ξ_1, $\mathcal{T}_{\Xi_1}$, shown in Example 13.

It is easy to see that $\bar{\mathcal{T}} = \mathcal{T}_{\Xi_1}$.

7. Conclusions

We provided a formal notion of simulator, of simulation campaign, and a formal *operational semantics* for simulation scripting languages through which simulation campaigns are defined on actual simulators.

Furthermore, we showed *soundness* and *completeness* of our operational semantics by proving that *any* simulation campaign defines a set of operational scenarios (soundness) and, conversely, that *any* set of operational scenarios can be defined through a simulation campaign (completeness).

Our operational semantics enables construction of formal proofs showing *equivalence* of two simulation scripts. This, in turn, enables formal proofs of correctness of the many simulation scripts optimisations used within simulation based formal verification approaches. Finally, we point out that availability of an operational semantics for simulation scripts enables investigation of more aggressive approaches to the optimisation of the simulation scripts driving simulation based SLFV.

Acknowledgments. This research has been partially supported by the following projects: EC FP7 grant SmartHG: *Energy Demand Aware Open Services for Smart Grid Intelligent Automation* (317761), EC FP7 grant PAEON: *Model Driven Computation of Treatments for Infertility Related Endocrinological Diseases* (600773), and MIUR grant *Excellence Departments 2018-2022* to Sapienza University of Rome Computer Science Department, SCAPR (POR FESR 2014-2020, Aerospazio e sicurezza), INdAM "GNCS Project 2019", Sapienza University 2018 project RG11816436BD4F21.

References

[1] RTCA DO-178C, Software Considerations in Airborne Systems and Equipment Certification (December 2011).

[2] R. Alur, Formal verification of hybrid systems, in: EMSOFT 2011, ACM, 2011. doi:10.1145/2038642.2038685.

[3] J. Bengtsson, K. Larsen, F. Larsson, P. Pettersson, W. Yi, UPPAAL — a tool suite for automatic verification of real-time systems, in: Hybrid Systems III, Vol. 1066 of LNCS, Springer, 1996. doi:10.1007/BFb0020949.

[4] T. Henzinger, P.-H. Ho, H. Wong-toi, HyTech: A model checker for hybrid systems, STTT 1 (1997).

[5] G. Della Penna, B. Intrigila, I. Melatti, E. Tronci, M. Venturini Zilli, Exploiting transition locality in automatic verification of finite state concurrent systems, STTT 6 (4) (2004). doi:10.1007/s10009-004-0149-6.

[6] G. Frehse, PHAVer: Algorithmic verification of hybrid systems past hytech, STTT 10 (3) (2008). doi:10.1007/s10009-007-0062-x.

[7] A. Cimatti, M. Roveri, A. Susi, S. Tonetta, Validation of requirements for hybrid systems: A formal approach, ACM TOSEM 21 (4) (2013). doi:10.1145/2377656.2377659.

[8] S. Kong, S. Gao, W. Chen, E. Clarke, dreach: δ-reachability analysis for hybrid systems, in: TACAS 2015, Vol. 9035 of LNCS, Springer, 2015. doi:10.1007/978-3-662-46681-0_15.

[9] Simulink.
URL http://www.mathworks.com

[10] VisSim.
URL http://www.vissim.com

[11] Modelica.
URL http://www.modelica.org

[12] OpenModelica.
URL http://www.openmodelica.org

[13] JModelica.
URL http://www.jmodelica.org

[14] Dymola.
URL http://www.claytex.com/products/dymola/

[15] E. C. for Space Standardization (ECSS), System modelling and simulation, ESA Requirements and Standards Division, ECSS-E-TM-10-21A (2010).

[16] T. Mancini, F. Mari, A. Massini, I. Melatti, F. Merli, E. Tronci, System level formal verification via model checking driven simulation, in: CAV 2013, Vol. 8044 of LNCS, Springer, 2013. doi:10.1007/978-3-642-39799-8_21.

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

[18] T. Mancini, F. Mari, A. Massini, I. Melatti, E. Tronci, System level formal verification via distributed multi-core hardware in the loop simulation, in: PDP 2014, IEEE, 2014. doi:10.1109/PDP.2014.32.

[19] T. Mancini, F. Mari, A. Massini, I. Melatti, E. Tronci, SyLVaaS: System level formal verification as a service, in: PDP 2015, IEEE, 2015.

[20] T. Mancini, F. Mari, A. Massini, I. Melatti, E. Tronci, Anytime system level verification via parallel random exhaustive hardware in the loop simulation, Microprocessors and Microsystems 41 (2016). doi:10.1016/j.micpro.2015.10.010.

[21] T. Mancini, F. Mari, A. Massini, I. Melatti, E. Tronci, SyLVaaS: System level formal verification as a service, Fundam. Inform. 1–2 (2016). doi:10.3233/FI-2016-1444.

[22] T. Mancini, F. Mari, A. Massini, I. Melatti, I. Salvo, E. Tronci, On minimising the maximum expected verification time, Inf. Proc. Lett. 122 (2017). doi:10.1016/j.ipl.2017.02.001.

[23] Database of relevant traffic scenarios for highly automated vehicles.
URL http://www.pegasusprojekt.de/files/tmpl/pdf/AVT%20Symposium%202017%20Database%20traffic%20scenarios_Folien.pdf

[24] Flames: a powerful scenario database management system.
URL http://www.ternion.com/scenario-database

[25] T. Mancini, F. Mari, A. Massini, I. Melatti, E. Tronci, Simulator semantics for system level formal verification, EPTCS 193 (2015).

[26] G. Hamon, J. Rushby, An operational semantics for stateflow, in: M. Wermelinger, T. Margaria-Steffen (Eds.), Fundamental Approaches to Software Engineering, Springer Berlin Heidelberg, Berlin, Heidelberg, 2004, pp. 229–243.

[27] O. Bouissou, A. Chapoutot, An operational semantics for simulink's simulation engine, in: Proc. 13th ACM SIGPLAN/SIGBED Int. Conf. Languages, Compilers, Tools and Theory for Embedded Systems, LCTES '12, ACM, New York, NY, USA, 2012, pp. 129–138. doi:10.1145/2248418.2248437.
URL http://doi.acm.org/10.1145/2248418.2248437

[28] O. Bouissou, A. Chapoutot, An operational semantics for simulink's simulation engine, SIGPLAN Not. 47 (5) (2012) 129–138. doi:10.1145/2345141.2248437.
URL http://doi.acm.org/10.1145/2345141.2248437

[29] D. Kågedal, P. Fritzson, Generating a modelica compiler from natural semantics specifications, in: In Proceedings of the Summer Computer Simulation Conference, 1998.

[30] S. Foster, B. Thiele, A. Cavalcanti, J. Woodcock, Towards a utp semantics for modelica, in: J. Bowen, H. Zhu (Eds.), Unifying Theories of Programming, Springer International Publishing, Cham, 2017, pp. 44–64.

[31] G. Verzino, F. Cavaliere, F. Mari, I. Melatti, G. Minei, I. Salvo, Y. Yushtein, E. Tronci, Model checking driven simulation of sat procedures, in: SpaceOps 2012, 2012. doi:10.2514/6.2012-1275611.

[32] S. Bak, P. Duggirala, Simulation-equivalent reachability of large linear systems with inputs, in:

CAV 2017, Vol. 10426 of LNCS, Springer, 2017. doi:10.1007/978-3-319-63387-9_20.

[33] C. Fan, B. Qi, S. Mitra, M. Viswanathan, DryVR: Data-driven verification and compositional reasoning for automotive systems, in: CAV 2017, Vol. 10426 of LNCS, Springer, 2017. doi:10.1007/978-3-319-63387-9_22.

[34] S. Tripakis, C. Sofronis, P. Caspi, A. Curic, Translating discrete-time Simulink to Lustre, ACM TECS 4 (4) (2005). doi:10.1145/1113830.1113834.

[35] B. Meenakshi, A. Bhatnagar, S. Roy, Tool for translating Simulink models into input language of a model checker, in: ICFEM 2006, Springer, 2006. doi:10.1007/11901433_33.

[36] M. Whalen, D. Cofer, S. Miller, B. Krogh, W. Storm, Integration of formal analysis into a model-based software development process, in: FMICS 2007, Vol. 4916 of LNCS, Springer, 2007. doi:10.1007/978-3-540-79707-4_7.

[37] Y. Annpureddy, C. Liu, G. E. Fainekos, S. Sankaranarayanan, S-TaLiRo: A tool for temporal logic falsification for hybrid systems, in: TACAS 2011, Vol. 6605 of LNCS, Springer, 2011. doi:10.1007/978-3-642-19835-9_21.

[38] H. Abbas, G. Fainekos, S. Sankaranarayanan, F. Ivančić, A. Gupta, Probabilistic temporal logic falsification of cyber-physical systems, ACM TECS 12 (2s) (2013). doi:10.1145/2465787.2465797.

[39] B. Hoxha, A. Dokhanchi, G. Fainekos, Mining parametric temporal logic properties in model based design for cyber-physical systems, STTT (2017). doi:10.1007/s10009-017-0447-4.

[40] S. Sankaranarayanan, S. Kumar, F. Cameron, B. Bequette, G. Fainekos, D. Maahs, Model-based falsification of an artificial pancreas control system, ACM SIGBED Review 14 (2) (2017). doi:10.1145/3076125.3076128.

[41] A. Adimoolam, T. Dang, A. Donzé, J. Kapinski, X. Jin, Classification and coverage-based falsification for embedded control systems, in: CAV 2017, Vol. 10426 of LNCS, Springer, 2017. doi:10.1007/978-3-319-63387-9_24.

[42] P. Zuliani, A. Platzer, E. Clarke, Bayesian statistical model checking with application to Stateflow/Simulink verification, Form. Meth. Sys. Des. 43 (2) (2013). doi:10.1007/s10703-013-0195-3.

[43] E. Clarke, A. Donzé, A. Legay, On simulation-based probabilistic model checking of mixed-analog circuits, Form. Meth. Sys. Des. 36 (2) (2010). doi:10.1007/s10703-009-0076-y.

[44] T. Mancini, F. Mari, I. Melatti, I. Salvo, E. Tronci, J. Gruber, B. Hayes, M. Prodanovic, L. Elmegaard, Demand-aware price policy synthesis and verification services for smart grids, in: SmartGridComm 2014, IEEE, 2014. doi:10.1109/SmartGridComm.2014.7007745.

[45] B. Hayes, I. Melatti, T. Mancini, M. Prodanovic, E. Tronci, Residential demand management using individualised demand aware price policies, IEEE Trans. Smart Grid 8 (3) (2017). doi:10.1109/TSG.2016.2596790.

[46] T. Mancini, F. Mari, I. Melatti, I. Salvo, E. Tronci, J. Gruber, B. Hayes, M. Prodanovic, L. Elmegaard, User flexibility aware price policy synthesis for smart grids, in: DSD 2015, IEEE, 2015. doi:10.1109/DSD.2015.35.

[47] N. Miskov-Zivanov, P. Zuliani, E. Clarke, J. Faeder, Studies of biological networks with statistical model checking: Application to immune system cells, in: ACM-BCB 2013, ACM, 2013. doi:10.1145/2506583.2512390.

[48] E. Tronci, T. Mancini, I. Salvo, S. Sinisi, F. Mari, I. Melatti, A. Massini, F. Davi', T. Dierkes, R. Ehrig, S. Röblitz, B. Leeners, T. Krüger, M. Egli, F. Ille, Patient-specific models from inter-patient biological models and clinical records, in: FMCAD 2014, IEEE, 2014. doi:10.1109/FMCAD.2014.6987615.

[49] T. Mancini, E. Tronci, I. Salvo, F. Mari, A. Massini, I. Melatti, Computing biological model parameters by parallel statistical model checking, in: IWBBIO 2015, Vol. 9044 of LNCS, Springer, 2015. doi:10.1007/978-3-319-16480-9_52.

[50] M. Broy, B. Jonsson, J. Katoen, M. Leucker, A. Pretschner, Model-Based Testing of Reactive Systems: Advanced Lectures, Springer, 2005. doi:10.1007/b137241.

[51] A. Gadkari, A. Yeolekar, J. Suresh, S. Ramesh, S. Mohalik, K. Shashidhar, Automotgen: Automatic model oriented test generator for embedded control systems, in: CAV 2008, Vol. 5123 of LNCS, Springer, 2008. doi:10.1007/978-3-540-70545-1_19.

[52] A. Kanade, R. Alur, F. Ivancic, S. Ramesh, S. Sankaranarayanan, K. Shashidhar, Generating and analyzing symbolic traces of Simulink/Stateflow models, in: CAV 2009, Vol. 5643 of LNCS, Springer, 2009. doi:10.1007/978-3-642-02658-4_33.

[53] A. Brillout, N. He, M. Mazzucchi, D. Kroening, M. Purandare, P. Rümmer, G. Weissenbacher, Mutation-based test case generation for simulink models, in: FMCO 2009, Springer, 2010. doi:10.1007/978-3-642-17071-3.

[54] R. Venkatesh, U. Shrotri, P. Darke, P. Bokil, Test generation for large automotive models, in: ICIT 2012, Vol. 7521, IEEE, 2012. doi:10.1109/ICIT.2012.6210014.

[55] C. Yang, D. Dill, Validation with guided search of the state space, in: DAC 1998, ACM, 1998. doi:10.1145/277044.277201.

[56] P. Ho, T. Shiple, K. Harer, J. Kukula, R. Damiano, V. Bertacco, J. Taylor, J. Long, Smart simulation using collaborative formal and simulation engines, in: ICCAD 2000, IEEE, 2000.

[57] K. Nanshi, F. Somenzi, Guiding simulation with increasingly refined abstract traces, in: DAC 2006, ACM, 2006. doi:10.1145/1146909.1147097.

[58] F. De Paula, A. Hu, An effective guidance strategy for abstraction-guided simulation, in: DAC 2007, ACM, 2007. doi:10.1145/1278480.1278498.

[59] E. Sontag, Mathematical Control Theory: Deterministic Finite Dimensional Systems (2nd Ed.),

Springer, 1998.

[60] F. Cellier, E. Kofman, Continuous System Simulation, Springer, 2010.

[61] G. Kreisselmeier, T. Birkholzer, Numerical nonlinear regulator design, IEEE Trans. Aut. Contr. 39 (1) (1994).

[62] V. Alimguzhin, F. Mari, I. Melatti, I. Salvo, E. Tronci, Automatic control software synthesis for quantized discrete time hybrid systems, in: CDC 2012, IEEE, 2012. doi:10.1109/CDC.2012.6426260.

[63] V. Alimguzhin, F. Mari, I. Melatti, I. Salvo, E. Tronci, Linearizing discrete-time hybrid systems, IEEE TAC 62 (10) (2017). doi:10.1109/TAC.2017.2694559.

Appendix A. Simulation Campaigns and MATLAB Simulink Program

In this Appendix, we illustrate the MATLAB implementation used to obtain the results shown in Tables 1 and 2 of Section 4.1

Listings 3 to 9 show the MATLAB program for Examples 11 and 12, described at a higher level in Section 4.1. Listing 3 is the MATLAB script for simulating command sequences of Examples 11 and 12 on the Inverted Pendulum on Cart (Example 5), named `penddemoDist` and shown in Figures 7 and 8. Line 11 of Listing 3 invokes script `example10` of left column of Table 2 (resp. `example11` of left column of Table 1) containing the sequence of simulation campaign commands of Example 11 (resp. Example 12).

Simulation campaign commands *run*, *store*, *load*, and *free* are realised in corresponding MATLAB functions `s_run`, `s_store`, `s_load`, and `s_free` of Listings 4 to 7.

In order to follow simulation steps, at the end of such functions current time and hashing of current state vector are printed by invoking function `display_state_hash` (Listing 8). Notice that a numerical vector is identified by its hash value, thus no two different states have the same hashing and two states with the same hashing are indeed the same state.

Programmatic simulations in MATLAB are performed through the `sim` command (line 5 of Listing 4). Besides the name of system to be simulated, `sim` takes as input a set of simulation parameters[1]. In our case, simulation parameters are in global variable `SimParam` defined in lines 5–9 of Listing 3 in order to meet the following behaviour.

[1] https://it.mathworks.com/help/simulink/ug/using-the-sim-command.html

To *stop* and *restore* simulations, we first need to save the complete simulation state `xFinal` (containing values of system variables such as, *e.g.*, integrators) and then to consider `xFinal` as the initial state of a new simulation (`xInitial`). Such an assignment to simulation parameters is used in function `s_run` (line 6 of Listing 4), where global variable `xInitial` is assigned to the current value of `xFinal`. The first state of simulation is an exception, since there is no prior `xFinal` to be read. For this reason, `SimParam.LoadInitialState` is initially set to `off` (Listing 3) and then assigned to `on` each time a simulation ends (Listing 4). Output of `sim` command is kept in global variable `simOut`.

Global variables–among which there are output of `sim` (`simOut`), current simulation time (`CurSnapTime`) and initial state of next simulation (`xInitial`)–are saved by command `s_store` (Listing 5) in a file named after the state name with suffix `.mat`. Command `s_load` restores previously saved global variables, thus allowing the simulation behaviour described above. Command `s_free` physically deletes file created with command `s_store` according to its input (Listing 7).

To programmatically control simulations in Simulink, it is possible to associate callback functions to particular event such as, *e.g.*, starting of a simulation (https://it.mathworks.com/help/simulink/slref/model-parameters.html). This feature can be exploited in order to inject events in a simulation campaign. At line 3 of Listing 4, function `penddemo_inject` (Listing 9) is assigned to starting of simulation (`StartFcn`). Consequently, when a simulation starts with `s_run(e,t)`, Simulink first calls function `penddemo_inject(e)` and then runs simulation for t seconds. When invoked, function `penddemo_inject(e)` changes value of block *Mcart Dist* to the given input e (line 3 of Listing 9), thus triggering an event as required.

Listing 3: MATLAB main script `executetrace.m`

```
global FcnSetEvent System CurSnapTime SimParam xInitial simOut;
FcnSetEvent = 'penddemo_inject';
System = 'penddemoDist';
CurSnapTime = 0.0;
SimParam.SaveFinalState = 'on';
SimParam.SaveCompleteFinalSimState = 'on';
```

```
SimParam.FinalStateName = 'xFinal';
SimParam.LoadInitialState = 'off';
SimParam.InitialState = 'xInitial';
load_system(System);
tic; example11; toc;  % or: tic; example11; toc;
close_system(System, 0);
```

Listing 4: Function *run* (`s_run.m`)

```
function s_run(event, time)
    global FcnSetEvent System CurSnapTime SimParam xInitial simOut;
    set_param(System, 'StartFcn', sprintf('%s(%d)', FcnSetEvent, event));
    SimParam.StopTime = sprintf('%g', CurSnapTime + time);
    simOut = sim(System, SimParam);
    xInitial = simOut.get('xFinal');
    CurSnapTime = simOut.get('xFinal').snapshotTime;
    SimParam.LoadInitialState = 'on';
    display_state_hash();
end
```

Listing 5: Function *store* (`s_store.m`)

```
function s_store(st)
    global FcnSetEvent System CurSnapTime SimParam xInitial simOut;
    save(st);
    display_state_hash();
end
```

Listing 6: Function *load* (`s_load.m`)

```
function s_load(st)
    global FcnSetEvent System CurSnapTime SimParam xInitial simOut;
    load(st);
    display_state_hash();
end
```

Listing 7: Function *free* (`s_free.m`)

```
function s_free(st)
    delete(strcat(st, '.mat'));
    display_state_hash();
end
```

Listing 8: Function to print current time and hash value of current state vector (`display_state_hash.m`)

```
function display_state_hash()
    global xInitial CurSnapTime;
    res = sprintf('CurSnapTime: %.2f', CurSnapTime);
    if (CurSnapTime > 0)
        y = xInitial.loggedStates;
    else
        x0 = Simulink.BlockDiagram.getInitialState('penddemoDist');
        y = x0.signals;
    end
    v = zeros(length(y));
    for i = 1:length(y)
        for j = 1:length(y(i).values)
            v(i) = y(i).values(j);
        end
    end
    fprintf('%s, State Hash: %s\n', res, DataHash(v));
end
```

Listing 9: Function for event injection (`penddemo_inject.m`)

```
function penddemo_inject(mass)
    assert(mass == 0 || mass == 1 || mass == 2);
    set_param('penddemoDist/Mcart Dist', 'value', sprintf('%d', mass));
end
```

Declaration of interests

X The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

☐The authors declare the following financial interests/personal relationships which may be considered as potential competing interests: