Microprocessors and Microsystems 41 (2016) 12–28

Contents lists available at ScienceDirect

Microprocessors and Microsystems

journal homepage: www.elsevier.com/locate/micpro

Anytime system level verification via parallel random exhaustive hardware in the loop simulation☆

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

Computer Science Department, Sapienza University of Rome, Italy

ARTICLE INFO

Keywords:
Model Checking of Hybrid Systems
Model checking driven simulation
Hardware in the loop simulation

ABSTRACT

System level verification of cyber-physical systems has the goal of verifying that the *whole* (i.e., software + hardware) system meets the given specifications. Model checkers for hybrid systems cannot handle system level verification of actual systems. Thus, Hardware In the Loop Simulation (HILS) is currently the main workhorse for system level verification. By using model checking driven exhaustive HILS, System Level Formal Verification (SLFV) can be effectively carried out for actual systems.

We present a *parallel random exhaustive* HILS based model checker for hybrid systems that, by simulating *all* operational scenarios *exactly once* in a *uniform random* order, is able to provide, at *any time* during the verification process, an *upper bound* to the probability that the System Under Verification exhibits an error in a yet-to-be-simulated scenario (Omission Probability).

We show effectiveness of the proposed approach by presenting experimental results on SLFV of the Inverted Pendulum on a Cart and the Fuel Control System examples in the Simulink distribution. To the best of our knowledge, no previously published model checker can *exhaustively* verify hybrid systems of such a size and provide at *any time* an upper bound to the Omission Probability.

1. Introduction

The cost for fixing a design error in a system becomes larger and larger as the design proceeds from the requirement analysis to the implementation (see, e.g., [2, Chapter 1]) since the later in the design phase an error is detected the more reworking it may trigger. The above observation has motivated the development of methods and tools to verify correctness of a system already in the early phases of its design, namely during the requirement analysis or during the functional specification phases. The goal of all such approaches is to catch design errors well *before* the system implementation begins.

Of course, all such approaches are *model based*, that is they work on a model describing the system behaviour since no system implementation exists in the early design phases. Accordingly, *System Verification* is carried out by simulating a system model and analysing its behaviour under a *suitable set* of simulation scenarios.

For example, in a digital hardware setting, model based approaches have been used since a long time. In fact, even before considering going to silicon, a heavy simulation activity is performed, aimed at verifying that the system model (defined, e.g., using Verilog, VHDL or SystemC[1]) meets the system requirements for most (possibly all) *uncontrollable inputs* (that is, *primary inputs* and *faults* the system is expected to withstand).

Along the same line of reasoning, in a purely software setting, before generating low level code, model based approaches are used to verify that the software model (defined, e.g., using AADL [3,4]) meets the given requirements.

If all possible simulation scenarios are considered, then we can prove *correctness* of the system (i.e., absence of simulation scenarios violating the system requirements), otherwise we can only show that the system design is faulty (by exhibiting a simulation scenario violating the system requirements). In other words, a *simulation campaign* that does not consider all possible simulation scenarios can only show that the system design has a bug. To show correctness of the system design we need an *exhaustive* simulation campaign, that is one considering all possible simulation scenarios. A verification approach able to show system correctness is usually referred to as *formal verification*. One of the most successful techniques to carry out formal verification is *Model Checking* (see, e.g., [5]).

☆ This paper is an extended and revised version of [1].

* Corresponding author.
E-mail addresses: tmancini@di.uniroma1.it (T. Mancini), mari@di.uniroma1.it (F. Mari), massini@di.uniroma1.it (A. Massini), melatti@di.uniroma1.it (I. Melatti), tronci@di.uniroma1.it (E. Tronci).

http://dx.doi.org/10.1016/j.micpro.2015.10.010
0141-9331/

[1] http://www.mentor.com/products/fv/hdl_designer/

The need for model checking stems from the high cost that a bug may have for certain systems. This is the case for *mission critical* systems, that is, systems for which a system malfunctioning may entail loss of money, as well as for *safety critical* systems, that is, systems for which a system malfunctioning may entail loss of human lives. Examples of mission critical systems are: decision support systems, satellites, processors (e.g., the infamous P5 FDIV bug costed about $475 million to INTEL). Examples of safety critical systems are: railway interlocking, avionics control software.

Many Cyber-Physical Systems (CPSs) are indeed mission or safety critical systems. Accordingly, in this paper we focus on formal verification techniques for CPSs.

A CPS consists of hardware (e.g., motors, electrical circuits, etc.) and software components. Thus, in order to verify a CPS design, we need methods and tools that can model and effectively support simulation of hardware as well as software components.

From a formal point of view, CPSs can be modelled as hybrid systems (see, e.g., [6] and citations thereof). Many *Model Based Design* software tools offer support for modelling and simulation of CPSs. Well known examples are Simulink[2], VisSim[3] and Modelica[4]. All such tools take as input a (mathematical) model of the behaviour of the CPS along with a simulation scenario and provide as output the time evolution (*trace* or *simulation run*) of the system at hand.

System Level Verification of CPSs aims at verifying that the *whole* (i.e., software + hardware) system meets the given specifications. *System Level Formal Verification (SLFV)* has the goal of *exhaustively* verifying that the above holds for *all* possible operational scenarios.

For digital circuits, formal verification is usually carried out using model checking techniques (e.g., see [7]). Unfortunately, model checkers for hybrid systems cannot handle SLFV of real world CPSs. Thus, HILS is currently the main workhorse for system level verification of CPSs, and is supported by model based design tools (e.g., the previously mentioned Simulink, VisSim and Modelica).

In HILS, the *actual software* reads/sends values from/to mathematical models (*simulation*) of the physical systems (e.g., engines, analog circuits, etc.) it will be interacting with. Notwithstanding the word *hardware*, in HILS the only hardware present is the one devoted to support the system simulation, that is: computational and communication devices. This is because HILS is used in a model based design setting to validate the system design *before* any hardware is built (the whole goal of model based design). For example, Simulink, VisSim, Modelica, ESA Satellite Simulation Infrastructure SIMULUS[5] all provide simulation software supporting HILS, where the only hardware involved is just the computer on which the simulator is actually running.

Simulation can be very time consuming. Accordingly, in order to reduce system design time, Opal-RT[6] and dSpace[7] among others provide modelling and simulation software along with FPGA-based hardware to support real-time simulation. We note that in all cases the only hardware present in HILS is the one supporting the simulation itself.

1.1. Motivations

SLFV is an exhaustive HILS where *all* relevant simulation scenarios are considered. Using a parallel model checking driven approach, exhaustive HILS enables formal verification of actual systems. Examples of such systems are the Inverted Pendulum on a Cart (IPC) and the Fuel Control System (FCS) in the Simulink distribution (see Section 6.1.1).

Considering that parallel exhaustive HILS based SLFV may take days of computation (e.g., see [8]), from a practical point of view it would be very useful to have available at *any time* during the verification process, *quantitative* information about the degree of assurance attained. Such an information would enable us to evaluate if it is worth to continue the verification activity, or instead stop it since the degree of assurance attained can be considered adequate for the application at hand (*graceful degradation*).

The above considerations suggest looking for a HILS based model checking approach satisfying the following requirements: (i) it is *parallel*, in order to make exhaustive HILS computationally feasible; (ii) it is *exhaustive*, since our focus is SLFV; (iii) it is *any time*, to support *graceful degradation*.

The work in [9] presents a Propositional Satisfiability (SAT) based model checker for finite state systems which returns, at *any time* during the verification process, the *coverage* (i.e., the fraction of operational scenarios verified so far). Unfortunately, while coverage measures the *amount* of verification work done, it does not provide any information about the *degree of assurance* attained by the verification process. This is because formal verification aims at finding *hard to find* errors, i.e., errors that were not detected while verifying operational scenarios designed by experts. As a result, formal verification addresses the search of errors that we are *unlikely* to consider. For this reason, we can model the problem as an adversary system, that is a system where, knowing our verification strategy, the adversary places the error in operational scenarios we are less likely to visit. In such a framework, *any* deterministic ordering of the operational scenarios would not increase the degree of assurance until the end of the verification. In fact, the adversary would choose to place the single error in the *last* scenario picked by the verification procedure.

To provide a formally sound information about the degree of assurance attained by the verification process, approaches have been proposed which verify the operational scenarios in a *random* order. In particular, the work in [10] presents a Monte-Carlo based model checker for finite state systems that provides, at *any time* during the verification process, an upper bound to the probability that the System Under Verification (SUV) exhibits an error in a yet-to-be-simulated scenario (Omission Probability). The Omission Probability (OP) provides indeed the information we are looking for. Unfortunately, while Monte-Carlo based approaches guarantee randomness (thereby enabling OP computation) they are not exhaustive (within a finite time).

To the best of our knowledge, no model checker is available, neither for finite state systems nor for hybrid systems, which, at the same time, is both *random* and *exhaustive*, thereby enabling effective *anytime* SLFV. In this paper we advance the state of the art by presenting a *parallel random exhaustive* HILS based model checker along with experimental results showing its effectiveness.

1.2. Main contribution

Our System Under Verification (SUV) is a *Hybrid System* (e.g., see [6] and citations thereof) whose inputs belong to a finite set of uncontrollable events (*disturbances*) modelling failures in sensors or actuators, variations in the system parameters, etc. We focus on *deterministic systems* (the typical case for control systems) and model nondeterministic behaviours (such as faults) with disturbances. Accordingly, in our framework, a *simulation scenario* is just a finite sequence of disturbances and a *simulation campaign* is a finite sequence of simulation instructions (namely: *save* a simulation state, *restore* a saved simulation state, *remove* a saved

[2] http://www.mathworks.com.
[3] http://www.vissim.com.
[4] http://www.modelica.org.
[5] http://www.esa.int/Our_Activities/Operations/gse/ESA_operations_software_licensable_products_-_overview .
[6] http://www.opal-rt.com/about-hardware-loop-simulation .
[7] https://www.dspace.com/en/inc/home.cfm .