BreachFlows: Simulation-Based Design with Formal Requirements for Industrial CPS

Alexandre Donzé
Decyphir SAS, Moirans, France
http://www.decyphir.com
alex@decyphir.com

Abstract

Cyber-Physical Systems (CPS) are computerized systems in interaction with their physical environment. They are notoriously difficult to design because their programming must take into account these interactions which are, by nature, a mix of discrete, continuous and real-time behaviors. As a consequence, formal verification is impossible but for the simplest CPS instances, and testing is used extensively but with little to no guarantee. Falsification is a type of approach that goes beyond testing in the direction of a more formal methodology. It has emerged in the recent years with some success. The idea is to generate input signals for the system, monitor the output for some requirements of interest, and use black-box optimization to guide the generation toward an input that will falsify, i.e., violate, those requirements. Breach is an open source Matlab/Simulink toolbox that implements this approach in a modular and extensible way. It is used in academia as well as for industrial applications, in particular in the automotive domain. Based on experience acquired during close collaborations between academia and industry, Decyphir is developing BreachFlows, and extension/front-end for Breach which implements features that are required or useful in an industrial context.

2012 ACM Subject Classification Software and its engineering; Computer systems organization → Embedded and cyber-physical systems; Theory of computation → Timed and hybrid models; Theory of computation → Streaming models; Mathematics of computing → Solvers; Computing methodologies → Model verification and validation; Computing methodologies → Simulation evaluation; Computing methodologies → Simulation tools; Computing methodologies → Machine learning; Software and its engineering → Software creation and management; Theory of computation → Mathematical optimization

Keywords and phrases Cyber Physical Systems, Verification and Validation, Test, Model-Based Design, Formal Requirements, Falsification

Digital Object Identifier 10.4230/OASIcs.ASD.2020.5

Category Extended Abstract

1 Context: CPS Design, Verification and Validation

Cyber-Physical Systems (CPS) such as cars, planes, robots, medical devices, etc, have been steadily growing in sophistication and complexity. As a consequence, their construction require advanced design tools to ensure that they achieve their functional and safety goals. Model-based design (MBD) has become a standard practice to cope with the complexity and cost of development. In this paradigm, models of the system and its environment of increasing realism are developed and iteratively verified and validated against a suite of requirements until the real or production design is achieved. Stemming from the domains of logic, computation, digital circuits and later software verification, formal methods were developed to automate the process of proving that a given design satisfy a given requirement (Model-Checking) or alternatively, creating a design from a given requirement, which is then proven to be satisfied by construction (synthesis). For decades, various attempts have been made to bring these approaches to MBD for CPS. However, *proving* requirements in

the strictly formal sense, is possible only for the simplest CPS models, e.g., finite state machines, or in some cases timed automata. Most mathematical and computational models for CPS are so-called *hybrid systems*, mixing non-linear continuous and discrete dynamics. For these, existing formal methods do not scale well in general. This is the case for examples of most models created with modeling frameworks such as Mathworks tool Matlab/Simulink [6], which is ubiquitous in the industry. These frameworks are used by engineers mostly for simple testing using simulation. They sometimes implement formal methods but their application is restricted to simple models or small components.

In the last decade or so, an intermediate approach between simple testing and formal methods has emerged. On one side, it is still based on simulations; but as simulator technology has progressed, it has become easier to produce large amount of simulation data, making it possible to perform different types of analysis such as statistical (a la Monte-Carlo), guided search, learning, etc. On the other side, this approach retain some characteristics of formal methods, e.g., the use of formal specifications languages such as temporal logics and their quantitative semantics. One popular example is falsification. Given a system S with inputs u and a formal requirement φ, its goal is to find some input u^* such that the behavior of S using u^* falsifies or violates φ. The most common approach to solve this problem makes use of numerical black-box optimization and quantitative semantics. The satisfaction of φ can be estimated by a function $u \mapsto \rho(\varphi, S(u))$ where $\rho(\varphi, S(u)) < 0$ implies that u falsifies φ. Therefore looking to minimize $J(u) = \rho(\varphi, S(u))$ can lead to finding an input u^* such that $J(u^*) < 0$, meaning that u^* is a falsifying input. Conversely, if $J(u^*) \geq 0$ can be proven to be a global minimum, then we have proven that φ is always satisfied by S.

2 Breach Features Overview

The falsification concept and core ideas can be described in a few words but its application in practice can be much more daunting. Breach [3] is an open source Matlab/Simulink toolbox that implements the required ingredients in such a way that each one can be dealt with in a modular and reusable way, thus applying a separation of concern approach. In the following, we use the automatic transmission system pictured in Figure 1 to describe and briefly illustrate these components. They can be broadly categorized as follows:

- **Interfaces**, which define which signals in the models are inputs and outputs for Breach. In our example, throttle and brake are inputs, RPM, gear and speed are outputs for the model, but Breach can also monitor internal signals such as the OutputTorque or ImprellerTorque. Various parameters can also be part of an interface.
- **Input generators**, which define the search space or variable domain for the inputs. E.g., we might consider steps, pulses, piecewise-linear signals, etc. An input generator is often responsible for converting infinite domains (dense time, real-valued set of signals) into finite sets of variables suitable for an optimization problem. For example, if throttle is chosen to be a step signals going from zero to some value, then only two variables are enough to define the throttle signal at all times t: the time of the step and and its amplitude.
- **Requirements**, which define formally the requirements to be falsified. Breach supports Signal Temporal Logics (STL) [7], a formal specification language adapted for CPS. An example of a requirement easily expressed in STL is the following: φ= "whenever the car is in gear 4, the speed is above 30 miles per hour." Breach implements the efficient quantitative monitoring algorithm of [4], so that computing the quantitative satisfaction of a requirement is generally a negligible overhead compared to computing a simulation of the system.

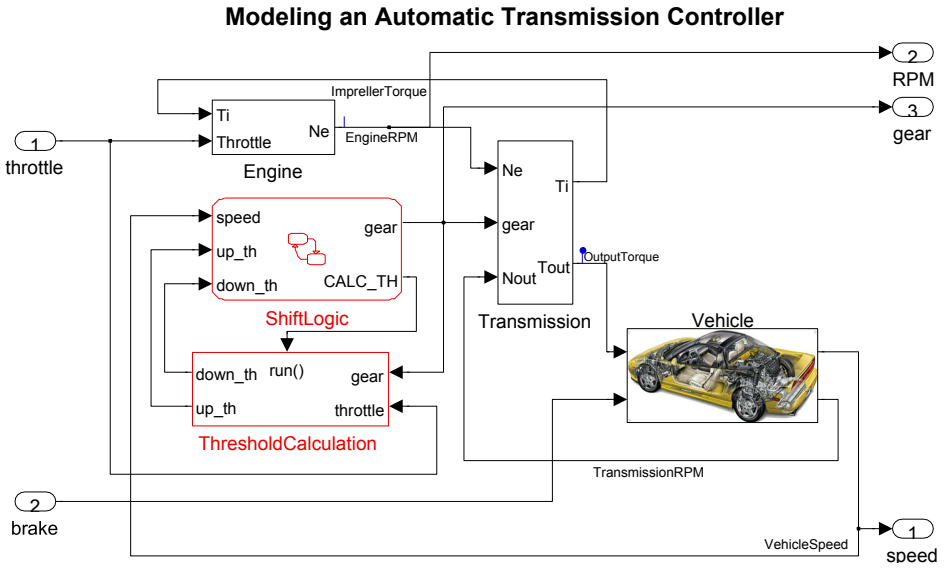

Figure 1 Automatic transmission system. This model simulates the behavior of an automatic gearbox as a function of throttle and braking from the driver.

- **Solvers**, which define the automated strategies that will solve the underlying optimization problem defined to find a falsifying input for the requirement. Solver go from "barbaric" random searches to genetic algorithm, local search with gradient estimation, etc.

Breach provides a collection of classes for each of these components (interfaces for data and Simulink models, basic input generators, STL, and optimization solvers) but one of its design goals was that this collection could be also extensible with reasonable ease. Examples of such extensions can be found in [5], where Breach is interfaced with a driving simulator written with the Unity engine and a neural network controller in Python; in [1], a different quantitative semantics for STL is implemented; in [2] a new stochastic local search solver is implemented; etc.

3 BreachFlows: An Engineer Friendly Interface to Breach

When applying falsification methods (or similar simulation-based approaches with formal requirements) in an industrial context, one is faced with the following challenges:

- **Models complexity, heterogeneity and changeability.** As one may expect, industrial-scale models are typically larger and more complex than theoretical/academic ones. The first consequence is that simulation has a higher computational cost so that performing as few simulations as possible is of paramount importance. Furthermore, a Simulink model can typically embed legacy code, or co-simulate with a different external simulator, import data or post-process outputs. Finally models are often under constant development by different engineers or teams of engineers so that new version with changes can be pushed frequently.
- **Abundance of requirements and lack of *formalized* requirements for inputs (test cases) and outputs.** Requirements for a given system design are about as complex and heterogeneous as models, but they are also typically informal and qualitative.

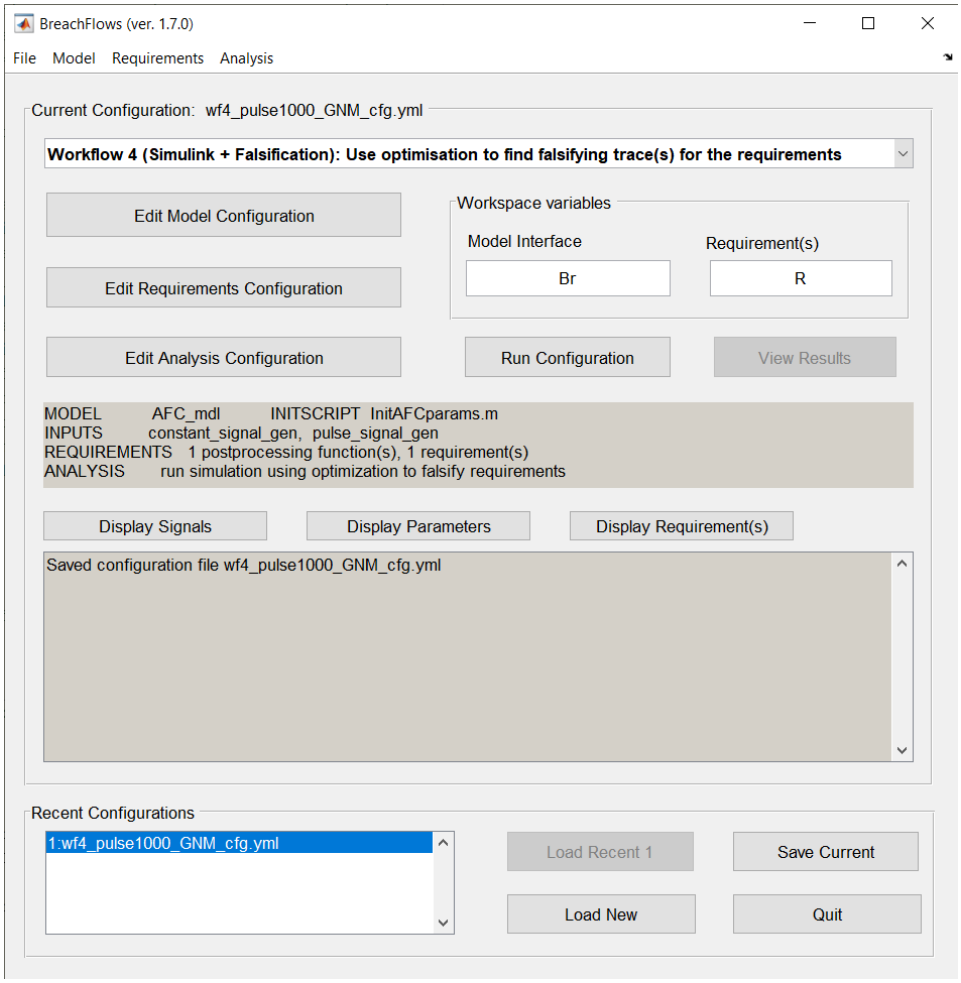

Figure 2 Top level GUI of BreachFlows.

Furthermore, although formalism such as (signal) temporal logic are languages that relatively simple in terms of their syntax and core semantics, they can be tricky to use properly. What is worse, engineers are rarely trained to use them and often reluctant or not able to invest time in learning them. Test cases used for testing those requirements are also typically under-specified and custom-made at the discretion of the developers or test engineers.

BreachFlows was developed on top of Breach to address these specific problems. It is essentially a user friendly front-end for Breach which can be used to build requirements sets and setup and maintain falsification problems and other types of analysis, so that they can be applied iteratively to support a CPS design at all stages of developments in a consistent way. Several features and characteristics and how they try to answer to the various challenges described above are the following:

- **Configuration management:** at the top level, BreachFlows is a GUI creating and managing configuration files for Breach typical workflows. They are clearly divided into three sections: models or data, requirements, and analysis, as illustrated on Figure 2. Sections can be imported from configuration to another, and they are designed to be robust to model or requirement changes, so that, e.g., a small change in the model can be reflected by a small change in a corresponding configuration;

- **Various features helping with high cost of simulation and heterogeneity:**
  - Use of parallel computation when possible;
  - System of efficient disk caching mechanism to reload previously computed simulations with the same parameters;
  - Use of custom scripts and functions (user-defined callbacks) for model initialization, inputs and requirements;
  - A mechanism of pre-conditions on input signals with constraints possibly expressed in STL so that whenever a test case or input is invalid, the corresponding simulation is skipped;
- **Input and requirement builders** consisting of a GUI pre-loaded with a collection of parameterized templates. Requirement templates are expressed in structured plain English which are mapped to STL formulas.

Breach is available as open source at `https://github.com/decyphir/breach` and BreachFlows is available on request at `info@decyphir.com`.

References

1 Koen Claessen, Nicholas Smallbone, Johan Liden Eddeland, Zahra Ramezani, Knut Åkesson, and Sajed Miremadi. Applying valued booleans in testing of cyber-physical systems. In *MT@CPSWeek*, pages 8–9. IEEE, 2018.

2 Jyotirmoy V. Deshmukh, Xiaoqing Jin, James Kapinski, and Oded Maler. Stochastic local search for falsification of hybrid systems. In *ATVA*, volume 9364 of *Lecture Notes in Computer Science*, pages 500–517. Springer, 2015.

3 Alexandre Donzé. Breach, A toolbox for verification and parameter synthesis of hybrid systems. In *Proc. of CAV 2010: the 22nd International Conference on Computer Aided Verification*, volume 6174 of *LNCS*, pages 167–170. Springer, 2010.

4 Alexandre Donzé, Thomas Ferrère, and Oded Maler. Efficient robust monitoring for STL. In *Proc. of CAV 2013: the 25th International Conference on Computer Aided Verification*, volume 8044 of *LNCS*, pages 264–279. Springer, 2013.

5 Tommaso Dreossi, Alexandre Donzé, and Sanjit A. Seshia. Compositional falsification of cyber-physical systems with machine learning components. *J. Autom. Reasoning*, 63(4):1031–1053, 2019. `doi:10.1007/s10817-018-09509-5`.

6 Mathworks Inc. Test generated code with sil and pil simulations. Cf. `https://www.mathworks.com/help/ecoder/examples/software-and-processor-in-the-loop-sil-and-pil-simulation.html`.

7 Oded Maler and Dejan Nickovic. Monitoring temporal properties of continuous signals. In *FORMATS/FTRTFT*, pages 152–166, 2004.