Bridging the Gap Between Requirements and Simulink Model Analysis

Anastasia Mavridou[1,2], Hamza Bourbouh[1,2], Pierre-Loïc Garoche[1,2,3], Dimitra Giannakopoulou[1], Tom Pressburger[1], and Johann Schumann[1.2]

[1]NASA Ames Research Center, [2]SGT, Inc., [3]Onera, The French Aerospace Lab

Abstract. Formal verification and simulation are powerful tools for the verification of requirements against complex systems. Requirements are developed in early stages of the software lifecycle and are typically expressed in natural language. There is a gap between such requirements and their software implementations. We present a framework that bridges this gap by supporting a tight integration and feedback loop between high-level requirements and their analysis against software artifacts. Our framework implements an analysis portal within the FRET requirements elicitation tool, thus forming an end-to-end, open-source environment where requirements are written in an intuitive, structured natural language, and are verified automatically against Simulink models.

1 Introduction

The safety critical industry imposes a strict development process according to which requirements are written in the early phases of the software lifecycle, and are refined into models and/or code, while keeping track of traceability information. Verification and validation (V&V) activities must ensure that the development process properly preserves these requirements (for example, see [8]) Requirements are typically written in natural language, which is well-known to be ambiguous and as such, not amenable to formal analysis. Frameworks like Stimulus [7] or FRET (Formal Requirements Elicitation Tool) [5] address this problem by enabling the capture of requirements in restricted natural languages with formal semantics. FRET additionally supports automated formalization of requirements in temporal logics.

To support V&V activities, it is necessary to associate high-level requirements with software artifacts in terms of architectural information such as components and signals. For formulas generated by FRET for example, the atomic propositions that make up a formula must be connected to variable values or method executions in the target Simulink model. To this end, we have developed an end-to-end, open-source requirements analysis framework that supports a tight integration and feedback loop between high level requirements and the V&V of models or code against these requirements. Our framework is available, open source, within FRET [1]; it currently connects FRET with the COCOSIM model verifier, with plans on extending it to support a variety of analysis tools.

[1] `https://github.com/NASA-SW-VnV/fret`

Our framework provides: 1) automatic extraction of Simulink model information and association of requirements with target model signals and components; 2) translation of FRET temporal logic formulas into synchronous dataflow COCOSPEC [1] specifications as well as Simulink monitors, to be used by verification tools; and 3) interpretation of counterexamples produced by verification back at model and requirement levels.

Similarly to [11,10], our framework checks formal properties against Simulink models, but unlike [11], it does not involve translation by hand, and unlike [10], property propositions do not need to match model variables. Moreover, in our framework, analysis results can be traced back to requirements.

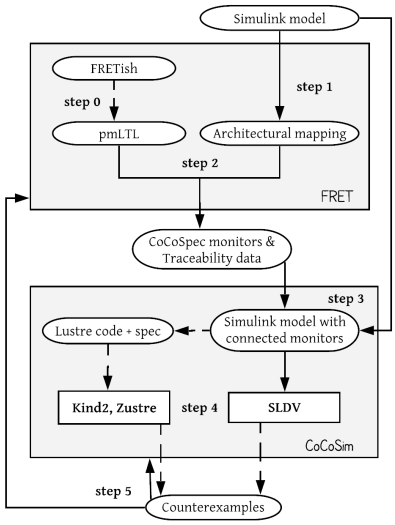


Fig. 1: Requirement analysis framework.

2 Our framework step-by-step

Figure 1 shows the workflow of our requirement analysis framework. The contributions of this paper are represented by continuous arrows. In step 0, requirements written in FRETISH are translated into pure Past Time Metric LTL (pmLTL) formulas. In step 1, data is used from the model under analysis, to produce an architectural mapping between requirement propositions and Simulink signals. In step 2, the pmLTL formulas and the architectural mapping are used to generate monitors in COCOSPEC, which is an extension of the synchronous dataflow language Lustre [6] for the specification of assume-guarantee contracts. In step 3, the generated COCOSPEC monitors and model traceability data are imported into COCOSIM [2] along with the Simulink model under analysis. COCOSIM automatically generates and attaches monitors on the Simulink model. From the complete model (initial model and attached monitors), COCOSIM also generates equivalent Lustre code. As a result, the complete model can be analyzed by both Simulink-based (e.g., Simulink Design Verifier (SLDV)) and Lustre-based (e.g., Kind2, Zustre) verification tools in step 4. Counterexamples generated during the analysis can be traced back to COCOSIM or FRET for simulation in step 5.

The next sections illustrate each workflow step in detail, using a requirement from the Lockheed Martin Cyber Physical Systems (LMCPS) challenge [3]. The LMCPS challenge is representative of flight-critical systems and is publicly available.[2] Requirement `[FSM-001]` (Figure 2) partly describes the required behavior of an advanced autopilot system with an independent sensor platform.

Step 0 : FRETISH to pmLTL A FRETISH requirement contains up to six fields: `scope`, `condition`, `component*`, `shall*`, `timing`, and `response*`. Mandatory fields are indicated by an asterisk. `component` specifies the component that

[2] `https://github.com/hbourbouh/lm_challenges`

NL: "*Exceeding sensor limits shall latch an autopilot pullup when the pilot is not in control (not standby* and the system is supported without failures (not apfail)"

FRETish: FSM shall always satisfy (sensorLimits & autopilot) ⇒ pullup

pmLTL: `H((sensorLimits & autopilot) ⟹ pullup)`

Fig. 2: `FSM-001` in Natural Language (NL), FRETISH, and pmLTL forms.

the requirement refers to. `shall` is used to express that the component's behavior must conform to the requirement. `response` is a Boolean condition that the component's behavior must satisfy. `scope` specifies the period when the requirement holds. The optional `condition` field is a Boolean expression that further constrains when the `response` shall occur. `timing`, e.g., *always*, *after/for N time units*, specifies when the response shall happen, subject to `condition` and `mode`.

The manually written FRETISH version of requirement `[FSM-001]`, shown in Figure 2, uses the `component`, `shall`, `timing`, and `response` fields. Since `scope` and `condition` fields are omitted, the requirement holds universally. The `autopilot` proposition was used by the requirements engineer to simplify the requirement; it equals `(! standby & ! apfail & supported)`. For each requirement, FRET generates a pmLTL formalization, e.g., see Figure 2 for the pmLTL of `[FSM-001]`. `H` refers to the Historically pmLTL operator [9].

Step 1 : Architectural Mapping To generate monitors and automatically attach them at the right hierarchical level of the model, we need architectural data from the model. For instance, for `[FSM-001]`, we need information about the hierarchical level, i.e., the path, of the model component that corresponds to the `FSM` component mentioned in FRETISH. Additionally, we need information about the signals of the component, e.g., name, type (e.g., `input, output`), datatype (e.g., `boolean, double, bus`) that correspond to the propositions mentioned in `[FSM-001]`.Our framework provides a mechanism to automatically extract the required data from a Simulink model.

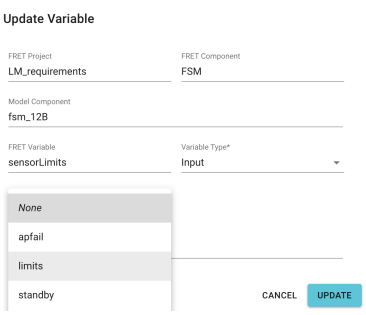


Fig. 3: `sensorLimits` mapping.

Once model data is imported, the architectural mapping procedure starts, which includes mapping every component and proposition mentioned in a requirement to a model component and a signal, respectively. There are two ways to do the architectural mapping: in the fortunate case that the same names are used both in the requirements and in the model, our tool automatically constructs the desired mapping. From our experience however, this is usually not the case. Different engineers work on requirements and on models, and these two parts are hardly ever aligned. For this reason, we provide an easy-to-use user interface, through which the user can pick the path of the corresponding model component or port from a drop-down menu and map it with a requirement component or proposition (see Figure 3 for the mapping of the `sensorLimits`

proposition of `FSM`, to the `limits` signal of the `fsm_12B` model component). Then our tool automatically identifies all the other required information (data types, dimensions, etc) to generate correct-by-construction monitors and corresponding traceability data. Alternatively, a user may provide the required data manually.

Step 2 : COCOSPEC Monitors and Traceability Data To translate pmLTL into COCOSPEC, we created a library of pmLTL operators in COCOSPEC:

```
--Once
node O(X:bool) returns (Y:bool);
let
  Y = X or (false -> pre Y);
tel
--Y since X
node S(X,Y: bool) returns (Z:bool);
let
Z = X or (Y and (false -> pre Z));
tel
```

```
--Historically
node H(X:bool) returns (Y:bool);
let
  Y = X -> (X and (pre Y));
tel
--Y since inclusive X
node SI(X,Y: bool) returns (Z:bool);
let
  Z = Y and (X or (false -> pre Z));
tel
```

The semantics of the unary `pre` and the binary initialization `->` operators is as follows. At time $t = 0$, `pre` p is undefined for an expression p, while for each time step $t > 0$ it returns the value of p at $t-1$. At time $t = 0$, p `->` q returns the value of p at $t = 0$, while for $t > 0$ it returns the value of q at t. The correctness of the COCOSPEC generated code was checked with extensive testing; we omit the details due to space limitation. Here is the `[FSM001]` COCOSPEC monitor:

```
contract FSMSpec(apfail:bool; sensorLimits:bool; standby:bool; supported:bool
    ; ) returns (pullup: bool; );
let
var autopilot:bool=supported and not apfail and not standby;
guarantee "FSM001" H ((sensorLimits and autopilot) => (pullup));
tel
```

The generated traceability data include the mapping of FRETISH propositions to the absolute path of Simulink signals and they are provided in JSON format.

Step 3 : Simulink Monitor Generation COCOSIM attaches COCOSPEC monitors to Simulink subsystems. This process relies heavily on COCOSIM's Lustre-to-Simulink compiler. The first compilation step is performed by LustreC [4], an open-source Lustre compiler, which produces information necessary to extract the model structure. The second step transforms the produced structure into Simulink blocks through the Simulink API. Each COCOSPEC construct (e.g., assume, guarantee) is compiled and translated: their equivalent Simulink blocks are provided by a dedicated COCOSIM library [2]. Mathematical operators are translated into equivalent Simulink blocks. The `pre` operator is implemented as a Simulink Unit delay block. Figure 4 shows the generated Simulink monitor for `[FSM-001]`. Once the monitor is generated, COCOSIM automatically attaches it on the Simulink model based on the traceability data from step 2.

Step 4: Verification of the complete model At this step, verification can be performed either at the Simulink level using e.g., the Simulink Design Verifier or at the Lustre level using e.g., Kind2. Since requirements are initially given to us in natural language, their semantics is often ambiguous. For instance, our interpretation of requirement `[FSM001]`, where all conditions must be satisfied at the same time for `pullup` to be activated, in FRETISH was shown to be invalid. After revisiting the requirement, we thought that potentially there is a time step

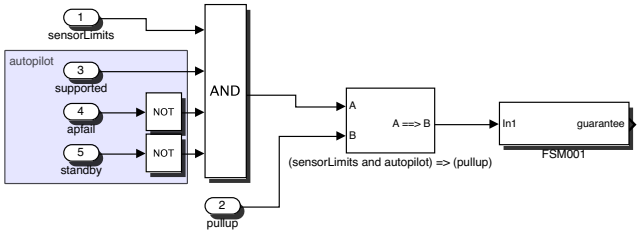
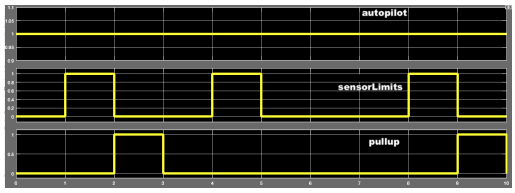


Fig. 4: Generated Simulink monitor for requirement `[FSM001]`.

Inputs	T=0	T=1	T=2	T=3
standby	F	F	F	F
apfail	F	F	F	F
supported	T	T	T	T
sensorLimits	T	F	T	F
Outputs				
pullup	F	T	F	F

(a) Counterexample.



(b) Simulation of `[FSM-001v2]`.

difference between `limits = true` and the activation of `pullup`. Thus we wrote the following second version, which however was also shown to be invalid.

FSM-001v2: if autopilot & pre_autopilot & pre_limits FSM shall immediately satisfy pullup

Step 5: Counterexample simulation Simulation of counterexamples is helpful for identifying weaker properties and producing meaningful reasoning scenarios. For instance, let us consider requirement `[FSM-001v2]`, for which Kind2 returned the counterexample shown in Table 5a. It is clear that even though `pullup` was activated the first time `sensorlLimits` hold, it was not activated at the second occurrence of `sensorLimits`. To better understand the behavior of the model, we performed a simulation based on this counterexample. Figure 5b illustrates a scenario when `sensorLimits` occurs multiple times during the autopilot operation, during which condition `autopilot` must be true. Based on this simulation, we found that `pullup` is latched only when `sensorLimits` holds in the previous step and has not been true for at least three steps before that.

3 Preliminary results

Table 1 summarizes preliminary results from applying our approach to the LM-CPS challenge [3]. Our framework is generic and can use the strengths of several analysis tools. For example, our case study uses Kind2 and SLDV: Kind2 was able to return an answer (decided) by using abstractions of non-linear functions such as trigonometric functions, in comparison with SLDV that mostly returned undecided for these cases. Due to its architectural mapping, our framework allows us to deploy COCOSPEC specifications at different levels of the model behavior. For instance, for the FSM component, we generated three different contracts

[3] The complete case study can be found at `https://tinyurl.com/fretForREFSQ`

that we deployed at three different hierarchical levels of the model. This is important for complex models where verification does not scale for global scopes. We applied modular verification to 20 out of the 69 requirements.

4 Conclusion

We described an end-to-end framework in which requirements written in a restricted natural language can be equivalently transformed into monitors and be analyzed against Simulink models by Simulink-based and Lustre-based verification tools. Our framework ensures that requirements and analysis activities are fully aligned: Simulink monitors are derived directly from requirements (and not handcrafted), and analysis results are traced back to requirements. The features of our framework are generic and can be used to integrate other requirement elicitation and analysis tools. In the future, we plan on providing additional ways of providing feedback from analysis tools to requirement engineers, to support them in correcting requirements. We also plan on extending our framework with additional types of analysis that can be performed at the level of requirements, e.g., realizability checking.

		Kind2	SLDV
Name	N_R	D/UN	D/UN
Triplex Signal Monitor (TSM)	6	6/0	6/0
Finite State Machine (FSM)	13	13/0	13/0
Tustin Integrator (TUI)	3	3/0	3/0
Control Loop Regulators (REG)	10	6/4	1/9
Nonlinear Guidance (NLG)	7	0/7	0/7
Feedforward Neural Network (NN)	4	0/4	0/4
Control Effector Blender (EB)	3	0/3	0/3
6DoF Autopilot (AP)	8	8/0	8/0
System Safety Monitor (SWIM)	3	3/0	1/2
Euler Transformation (EUL)	7	7/0	1/6
Total	64	46/18	33/31

Table 1: LMCPS results. N_R: #analyzed requirements, D: Decided, UN: undecided.

References

1. Champion, A., Gurfinkel, A., Kahsai, T., Tinelli, C.: CoCoSpec: A mode-aware contract language for reactive systems. In: Proc. SEFM 2016, pp. 347–366 (2016).
2. CoCo-team: CoCoSim – Automated Analysis Framework for Simulink. `https://github.com/coco-team/cocoSim2`
3. Elliott, C.: An example set of cyber-physical V&V challenges for S5. S5 (2016). `http://mys5.org/Proceedings/2016/Day_2/2016-S5-Day2_0945_Elliott.pdf`
4. Garoche, P., Kahsai, T., Thirioux, X.: LustreC, `https://github.com/coco-team/lustrec`
5. Giannakopoulou, D., Pressburger, T., Mavridou, A., Schummann, J.: Generation of formal requirements from structured natural language. REFSQ (2020)
6. Halbwachs, N., Caspi, P., Raymond, P., Pilaud, D.: The synchronous data flow programming language Lustre. Proc. IEEE **79**(9), 1305–1320 (1991)
7. Jeannet, B., Gaucher, F.: Debugging Embedded Systems Requirements with STIMULUS: an Automotive Case-Study. In: Proc. ERTS (2016).
8. RTCA: DO-178C: software considerations in airborne systems and equipment certification. (2011)
9. Baier, Christel., Katoen, Joost-Pieter.: Principles of model checking.
10. D. Balasubramanian., G. Pap., H. Nine., G. Karsai., M. Lowry., C. Păsăreanu., T. Pressburger.: Rapid property specification and checking for model-based formalisms. RSP (2011).
11. Nejati, Shiva., Gaaloul, Khouloud., Menghi, Claudio., Briand, Lionel C., Foster, Stephen., Wolfe, David.: Evaluating Model Testing and Model Checking for Finding Requirements Violations in Simulink Models (2019).