Bridging the Gap Between Requirements and Simulink Model Analysis

Anastasia Mavridou
KBR / NASA Ames
anastasia.mavridou@nasa.gov

Hamza Bourbouh
KBR / NASA Ames
hamza.bourbouh@nasa.gov

Pierre Loic Garoche
Onera / KBR / NASA Ames
pierre-loic.garoche@onera.fr

Dimitra Giannakopoulou
NASA Ames
dimitra.giannakopoulou@nasa.gov

Thomas Pressburger
NASA Ames
tom.pressburger@nasa.gov

Johann Schumann
KBR / NASA Ames
johann.m.schumann@nasa.gov

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 implementation. 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 industry imposes a strict development process according to which requirements for safety-critical code 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 the DO-178C [17] document). Requirements are typically written in natural language, which is prone to be ambiguous and, as such, not amenable to formal analysis. Frameworks like STIMULUS [14] or FRET (Formal Requirements Elicitation Tool) [11, 12] 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 or free variables of a formula must be connected to variable values or method invocations

In: M. Sabetzadeh, A. Vogelsang, S. Abualhaija, M. Borg, F. Dalpiaz, M. Daneva, N. Fernández, X. Franch, D. Fucci, V. Gervasi, E. Groen, R. Guizzardi, A. Herrmann, J. Horkoff, L. Mich, A. Perini, A. Susi (eds.): Joint Proceedings of REFSQ-2020 Workshops, Doctoral Symposium, Live Studies Track, and Poster Track, Pisa, Italy, 24-03-2020, published at http://ceur-ws.org

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 and open source; it currently connects FRET[1] with the COCOSIM model verifier [3, 4, 7], with plans to extend it to support a variety of analysis tools.

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 [5] 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 [2, 16], our framework checks formal properties against Simulink models, but unlike [16], it does not involve translation by hand, and unlike [2], property propositions do not need to match model variables. Moreover, in our framework, analysis results can be traced back to requirements.

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 by FRET into pure Past Time Metric Linear Temporal Logic (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 [13] for the specification of assume-guarantee contracts. In Step 3, the generated COCOSPEC monitors and model traceability data are imported into COCOSIM [7] along with the Simulink model under analysis. COCOSIM automatically generates and attaches monitors to 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 [6], Zustre [10]) verification tools in Step 4. Counterexamples generated during the analysis can be traced back to COCOSIM or FRET for simulation in Step 5.

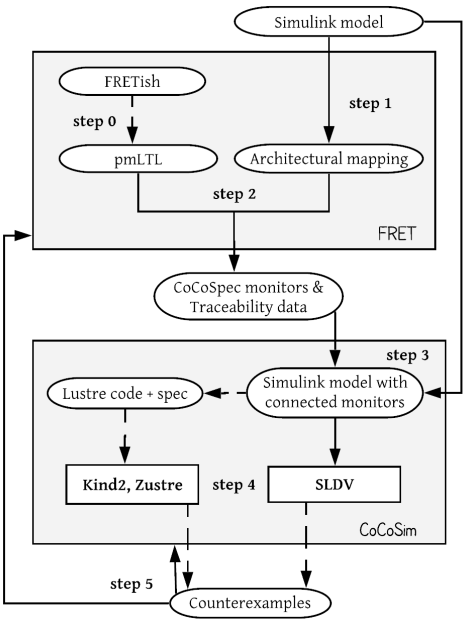

Figure 1: Requirement analysis framework

The next sections illustrate each workflow step in detail, using a requirement from the Lockheed Martin Cyber Physical Systems (LMCPS) challenge [8]. 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.

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)`

Figure 2: `FSM-001` in Natural Language (NL), FRETISH, and pmLTL forms (the Boolean variable autopilot is an abbreviation of `(!standby & !apfail & supported)`)

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

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 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 `scope`.

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 [1].

Step 1 : Architectural Mapping

To generate monitors and automatically attach them at the appropriate 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 architectural data from a Simulink model.

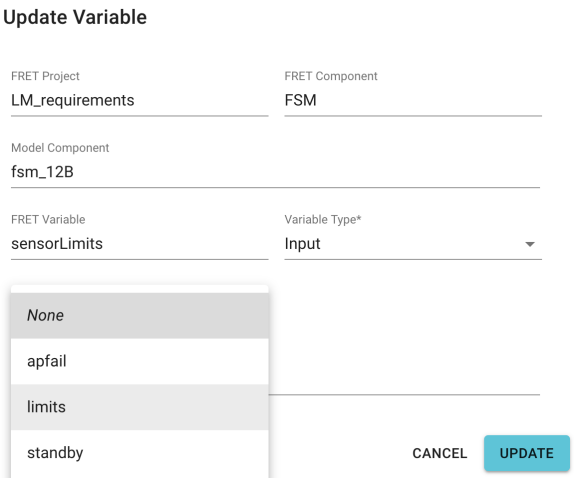

Figure 3: `sensorLimits` mapping

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 ideal case where 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 properly synchronized. 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 to 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 information manually.

Step 2 : COCOSPEC Monitors and Traceability Data

To translate pmLTL into COCOSPEC, we created a library of pmLTL operators in COCOSPEC, a specification language for Lustre:

```
--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 are defined as follows, in the synchronous dataflow language Lustre. At time $t = 0$, `pre` p is undefined for an expression p, while for each later time step $t > 0$, `pre` p 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. Here is the monitor fir `[FSM001]` in the COCOSPEC language:

```
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, which include the mapping of FRETISH propositions to the absolute paths of the Simulink signals, are provided in JSON format.

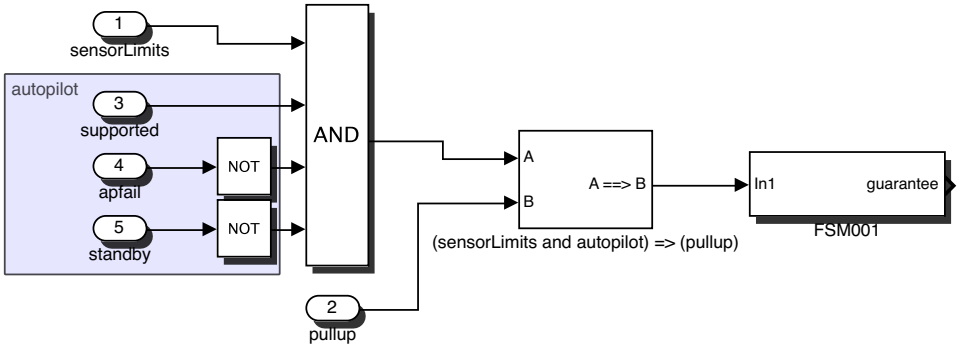

Figure 4: Generated Simulink monitor for requirement `[FSM001]`

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 [9], 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 block library [7]. 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 to the Simulink model based on the traceability data from Step 2. Once generated and attached at the model, the monitors can be used as runtime V&V components.

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

Table 1: Counterexample for `[FSM-001v2]`

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 [6]. Since requirements are initially given to us in natural language, their semantics is often ambiguous. For instance, our interpretation in FRETISH of the requirement `[FSM001]`, where all conditions must be satisfied at the same time for `pullup` to be activated, was shown to be invalid when checked against the model. After revisiting the requirement, we thought that potentially there is a time step 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 1. 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 5 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 [15].

This additional information helped us to tailor the proper requirement by disambiguating and refining the original natural language requirement. This shows on one hand, the ambiguous nature of natural language and, on the other hand, the elicitation capabilities of our framework.

3 Preliminary results

Table 2 summarizes preliminary results from applying our approach to the LMCPS challenge, which is described in detail in [15].Our framework is generic and can use the strengths of several analysis tools. For example, our case study uses Kind2 and SLDV. However, since the MathWorks license prevents the publication of empirical results comparing with SLDV, we only provide the Kind2 results in Table 2.

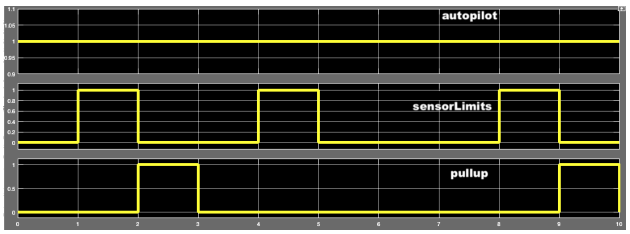

Figure 5: Simulation of `[FSM-001v2]`

In general, the LMCPS models are highly numeric and non-linear, which makes analysis very challenging when using SMT-based model checkers such as Kind2 and SLDV. In the case of Kind2, to handle non-linearities, we used abstractions of non-linear functions such as trigonometric functions and as a result, Kind2 was able to return an answer (decided) in cases that were undecided before adding the abstractions. We found modular verification particularly helpful in order to obtain meaningful results. 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 that we deployed at three different hierarchical levels of the model. This is important for complex models where verification does not scale when applied at the top level. We applied modular verification to 20 out of the 64 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 to provide additional ways of providing feedback from analysis tools to requirement engineers, to support them in correcting requirements. We also plan to extend our framework with additional types of analysis that can be performed at the level of requirements, e.g., realizability checking.

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

Table 2: LMCPS results with Kind2, N_R: #analyzed requirements, D: Decided, UN: Undecided

Acknowledgements. We thank Mohammad Hejase, Cesare Tinelli, and Daniel Larraz for fruitful discussions and feedback. This work was funded by the NASA ARMD System-Wide Safety Project.

References

[1] Baier, C., Katoen, J.P.: Principles of model checking. MIT press (2008)

[2] Balasubramanian, D., Pap, G., Nine, H., Karsai, G., Lowry, M., Păsăreanu, C., Pressburger, T.: Rapid property specification and checking for model-based formalisms. In: 2011 22nd IEEE International Symposium on Rapid System Prototyping. pp. 121–127 (May 2011). https://doi.org/10.1109/RSP.2011.5929985

[3] Bourbouh, H., Garoche, P.L., Garion, C., Gurfinkel, A., Kahsai, T., Thirioux, X.: Automated analysis of Stateflow models. EPiC Series in Computing **46**, 144–161 (2017)

[4] Bourbouh, H., Garoche, P.L., Loquen, T., Noulard, É., Pagetti, C.: CoCoSim, a code generation framework for control/command applications: An overview of CoCoSim for multi-periodic discrete Simulink models. In: 10th European Congress on Embedded Real Time Software and Systems (ERTS 2020) (2020)

[5] Champion, A., Gurfinkel, A., Kahsai, T., Tinelli, C.: CoCoSpec: A mode-aware contract language for reactive systems. In: Software Engineering and Formal Methods - 14th International Conference, SEFM 2016, Held as Part of STAF 2016, Vienna, Austria, July 4-8, 2016, Proceedings. pp. 347–366 (2016). https://doi.org/10.1007/978-3-319-41591-8_24

[6] Champion, A., Mebsout, A., Sticksel, C., Tinelli, C.: The Kind 2 model checker. In: Computer Aided Verification - 28th International Conference, CAV 2016, Toronto, ON, Canada, July 17-23, 2016, Proceedings, Part II. pp. 510–517 (2016). https://doi.org/10.1007/978-3-319-41540-6_29

[7] CoCo-team: CoCoSim – automated analysis framework for Simulink. `https://github.com/NASA-SW-VnV/CoCoSim`

[8] Elliott, C.: An example set of cyber-physical V&V challenges for S5, Lockheed Martin Skunk Works. In: Laboratory, A.F.R. (ed.) Safe & Secure Systems and Software Symposium (S5), 12-14 July 2016, Dayton, Ohio (2016), `http://mys5.org/Proceedings/2016/Day_2/2016-S5-Day2_0945_Elliott.pdf`

[9] Garoche, P., Kahsai, T., Thirioux, X.: LustreC, `https://github.com/coco-team/lustrec`

[10] Garoche, P., Kahsai, T., Thirioux, X.: Zustre, `https://github.com/coco-team/zustre`

[11] Giannakopoulou, D., Mavridou, A., Pressburger, T., Rhein, J., Schumann, J., Shi, N.: Formal requirements elicitation with FRET. In: 26th Intl Working Conference on Requirements Engineering: Foundation for Software Quality (REFSQ-2020, Tool) (2020), `http://ceur-ws.org`

[12] Giannakopoulou, D., Pressburger, T., Mavridou, A., Schumann, J.: Generation of formal requirements from structured natural language. In: 26th Intl Working Conference on Requirements Engineering: Foundation for Software Quality (REFSQ-2020) (2020)

[13] Halbwachs, N., Caspi, P., Raymond, P., Pilaud, D.: The synchronous data flow programming language LUSTRE. Proceedings of the IEEE **79**(9), 1305–1320 (1991)

[14] Jeannet, B., Gaucher, F.: Debugging Embedded Systems Requirements with STIMULUS: an Automotive Case-Study. In: 8th European Congress on Embedded Real Time Software and Systems (ERTS 2016). TOULOUSE, France (Jan 2016), `https://hal.archives-ouvertes.fr/hal-01292286`

[15] Mavridou, A., Bourbouh, H., Garoche, P.L., Hejase, M.: Evaluation of the FRET and CoCoSim tools on the ten Lockheed Martin cyber-physical challenge problems. Tech. Rep. TM-2019-220374, National Aeronautics and Space Administration (February 2020)

[16] Nejati, S., Gaaloul, K., Menghi, C., Briand, L.C., Foster, S., Wolfe, D.: Evaluating model testing and model checking for finding requirements violations in Simulink models. In: Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering. p. 1015–1025. ESEC/FSE 2019, Association for Computing Machinery, New York, NY, USA (2019). https://doi.org/10.1145/3338906.3340444

[17] RTCA, S.C.: DO-178C, Software Considerations in Airborne Systems and Equipment Cfertification (2011)