Bridging the Gap Between Requirements and Model Analysis : Evaluation on Ten Cyber-Physical Challenge Problems

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. [**Context**] Formal verification and simulation are powerful tools to validate requirements against complex systems. [**Problem**] Requirements are developed in early stages of the software lifecycle and are typically written in ambiguous natural language. There is a gap between such requirements and formal notations that can be used by verification tools, and lack of support for proper association of requirements with software artifacts for verification. [**Principal idea**] We propose to write requirements in an intuitive, structured natural language with formal semantics, and to support formalization and model/code verification as a smooth, well-integrated process. [**Contribution**] We have developed an end-to-end, open source requirements analysis framework that checks Simulink models against requirements written in structured natural language. Our framework is built in the Formal Requirements Elicitation Tool (FRET); we use FRET's requirements language named FRETISH, and formalization of FRETISH requirements in temporal logics. Our proposed framework contributes the following features: 1) automatic extraction of Simulink model information and association of FRETISH requirements with target model signals and components; 2) translation of temporal logic formulas into synchronous dataflow COCOSPEC specifications as well as Simulink monitors, to be used by verification tools; we establish correctness of our translation through extensive automated testing; 3) interpretation of counterexamples produced by verification tools back at requirements level. These features support a tight integration and feedback loop between high level requirements and their analysis. We demonstrate our approach on a major case study: the Ten Lockheed Martin Cyber-Physical, aerospace-inspired challenge problems.

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 [19] and its formal method supplement [20]).

Requirements are typically written in natural language, which is well-known to be ambiguous and as such, not amenable to formal analysis. On the other hand, formal mathematical notations can be used by analysis tools but are unintuitive for developers. Frameworks like Stimulus [15] or FRET (Formal Requirements Elicitation Tool) [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. This is also the case when requirements are formal; for example, the atomic propositions that make up a formula must be connected to variable values or method executions in the target code.

This paper presents 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 built on top of the open source tool FRET[4]. It connects FRETISH requirements to Simulink models for verification, and verification results back to requirements.

More specifically, our framework provides the following features: 1) automatic extraction of Simulink model information and association of FRETISH requirements with target model signals and components; 2) translation of FRET temporal logic formulas into synchronous dataflow COCOSPEC [3] specifications as well as Simulink monitors, to be used by Simulink verification tools; we establish correctness of our translation through extensive automated testing; and 3) interpretation of counterexamples produced by verification tools back at requirements level.

Our framework can be connected to any Simulink/Lustre V&V tools, although it currently uses our group's COCOSIM [6], and the Simulink Design Verifier (SLDV). We have applied our framework to a major case study: the Lockheed Martin Cyber Physical Systems (LMCPS) challenge [9], which is a set of aerospace-inspired examples provided as text documents specifying the requirements along with associated Simulink models. Examples range from basic integrators to complex autopilots. We report on our experience from the use of FRET, COCOSIM, and their interconnection, to capture and analyze LMCPS requirements.

The remainder of the paper is structured as follows: Section 2 recalls the underlying semantics of temporal logics and synchronous dataflow languages. Section 3 presents the workflow of our tool set. The generation of synchronous data flow contracts and the injection of these contracts as model elements in Simulink are presented in Sections 4 and 5, respectively. Section 6 describes the LMCPS case study and the lessons learned. Section 7 positions the contribution with respect to the state of the art, and finally, Section 8 concludes the paper.

[4] `https://github.com/nasa/FRET`

2 Background

Requirements Language and Temporal Logics. FRETISH is based on a restricted natural-language grammar and allows the user to conveniently express requirements. Here is an example requirement in FRETISH:

> **AP-002**: In roll_hold mode RollAutopilot shall always satisfy autopilot_engaged & no_other_lateral_mode

A FRET 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. If omitted, the requirement is deemed to hold universally, subject to `condition`. The optional `condition` field is a Boolean expression that further constrains when the `response` shall occur. For instance, `scope` can specify system behavior *before* a mode occurs, or *after* a mode ends, or when the system is *in* a mode. `timing`, e.g., *immediately*, *always*, *after/for/within N time units*, specifies when the response shall happen, subject to `condition` and `mode`.

For each such requirement, FRET generates a pure Future Time Metric LTL (fmLTL) and a pure Past Time Metric LTL (pmLTL) formalization. The syntax of the generated formulas is compatible with the NuSMV model checker [5]. In this paper, we focus on pmLTL, since COCOSPEC assume-guarantee contracts are pairs of past time formula predicates.

We briefly review the main pmLTL operators (`Y, O, H, S, SI`), which stand for Yesterday, Once, Historically, Since, and Since Inclusive, respectively. `Y` refers to the previous time step, i.e., at any non-initial time, $\texttt{Y}\phi$ is true iff ϕ holds at the previous time step. `O` refers to at least one past time step, i.e., $\texttt{O}\phi$ is true iff ϕ is true at some past time step including the present time. $\texttt{H}\phi$ is true iff ϕ is always true in the past. $\phi\,\texttt{S}\,\psi$ is true iff ψ holds somewhere at step t in the past and for all the step t' (such that $t' > t$) ϕ is true. Finally, $\phi\,\texttt{SI}\,\psi \equiv \phi\,\texttt{S}\,(\psi\,\&\,\phi)$. Timed modifiers constrain an operator's scope to specific intervals: $O_p\ [l, r]\ \phi$, where $O_p \in \{\texttt{O, H, S, SI}\}$ and $l, r \in \mathbb{N}^0$. For instance, $\texttt{O}\ [l, r]\ \phi$ is true at time t iff f was true in *at least one* of the previous time steps t' such that $t - r \leq t' \leq t - l$. E.g., $\texttt{O}[0, 3]$ restricts the scope of `O` to the interval including the step where the interval is interpreted and the previous 3 time steps.

Specification Language and Analysis Tools. We use the COCOSPEC language to generate monitors. COCOSPEC [3] is an extension of the synchronous dataflow language Lustre [13]. Lustre code consists of a set of *nodes* that transform infinite streams of *input* flows to streams of *output* flows, with possible local variables denoting *internal* flows. A notion of a symbolic "abstract" universal clock is used to model system progress. COCOSPEC extends Lustre with constructs for the specification of assume-guarantee contracts. Each contract is linked to a node and has access only to the input/output streams of that node.

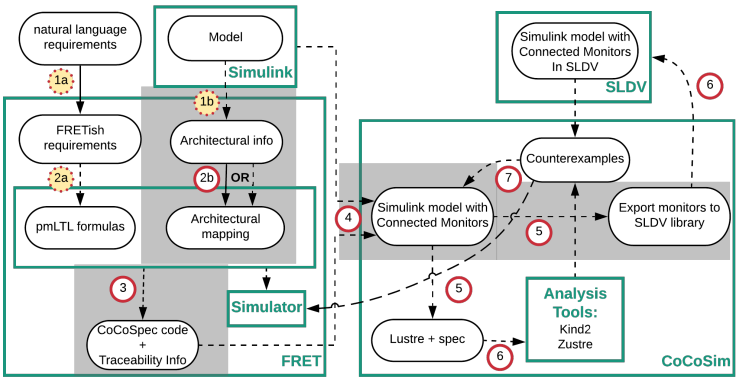

Fig. 1: FRET-COCOSIM Workflow.

The body of a contract contains `assume` (A) and `guarantee` (G) statements as well as `mode` and internal variable declarations. Modes consist of `require` (R) and `ensure` (E) statements. A mode is *active* at time t, if $\bigwedge R = true$ at t. Assumptions and requires are expressions over input streams, while guarantees and ensures are expressions over input/output streams. A node *satisfies* a contract $C = (A, G')$ if it satisfies $\mathtt{H}\ A \Rightarrow G'$, where $G' = G \cup \{R_i \Rightarrow E_i\}$.

The main goal of the open-source COCOSIM framework [6] is to support the analysis of safety-critical Simulink systems. Discrete systems developed in Simulink can be faithfully translated into Lustre [22], which is the intermediate language of COCOSIM. Using the Simulink API, COCOSIM iterates over Simulink blocks and produces equivalent Lustre nodes. Different Lustre-based tools can check the validity of the generated Lustre nodes, by using SMT-based model checking. In this paper, we perform analysis with the Kind2 [4] model-checker that uses k-induction, IC3/PDR [1], and invariant generation [16].

3 FRET-COCOSIM Workflow

Figure 1 shows the steps of the FRET-COCOSIM workflow. The contributions of this paper are highlighted by gray boxes and include steps `1b`, `2b`, `3`, `4`, `5`, and `7`. Continuous arrows represent manually-performed steps, whereas steps shown as dashed arrows are performed automatically.

In Step `1a`, requirements are manually written in FRETISH, which are subsequently formalized by FRET (Step `2a`) as pmLTL formulas. Our COCOSPEC code generator COCOGEN (Steps `2b` and `3`) takes pmLTL formulas and model information and generates a COCOSPEC representation. Since COCOSPEC, as a stand-alone tool does not require FRET, Steps `1a` and `2a` are optional (depicted by dotted circles). We ensure in the workflow that requirements and analysis activities are fully aligned. As a result, 1) Simulink monitors are derived directly from the requirements (and not handcrafted), and 2) analysis results can be traced back not only at the model level but also at the level of requirements so that the requirements engineer can benefit from insights gained through the analysis.

To do that, we need information that will bridge the gap between requirement propositions and model signals/architecture. In Step `1b`, this information can be automatically generated from a Simulink model. In Step `2b`, the mapping between the requirements' propositions and the model signals/architecture can be automatically or manually performed. In Step `3`, COCOGEN generates COCOSPEC contracts and traceability information.

The generated files are imported into COCOSIM along with the Simulink model. Then COCOSIM automatically generates a new Simulink model with monitor components generated from the imported contracts that are connected to the Simlink model using traceability information. At Step `5`, COCOSIM either: 1) generates equivalent Lustre code, which is annotated with the COCOSPEC specification properties that can be subsequently analyzed by the Kind2 and Zustre model checkers (Step `6`); or 2) transforms the COCOSPEC monitors to make the new system analyzable by SLDV. Any counterexample generated during the analysis can be traced back to Simulink or FRET for simulation (Step `7`).

4 COCOSPEC Specification Generation

Requirement Examples Next, we provide examples of LMCPS requirements that were given to us in natural language and show how we wrote them in FRETISH. Let us consider requirement `[FSM-001]` from the Finite State Machine (FSM) challenge problem, which is an abstraction of an advanced autopilot system with an independent sensor platform. The natural language form of `[FSM-001]` is : "*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)*". A FRETISH version of this requirement is:

FSM-001: FSM shall always satisfy (limits & autopilot) ⇒ pullup

where autopilot equals `(! standby & ! apfail & supported)`.

Additionally, let us consider requirement `[AP-004b]`. This challenge problem includes a realistic full six degree of freedom model of the DeHavilland Beaver airplane with an autopilot (AP). The natural language form of `[AP-004b]` is: "*Response to roll step commands shall not exceed 10% overshoot in calm air*". A FRETISH version of this requirement is:

AP-004b: in roll_hold mode AP shall always satisfy overshoot ≤ 0.1

Architectural Mapping To generate the COCOSPEC monitors and automatically connect them at the right hierarchical level of the model, we need architectural information 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 `FSM`. Additionally, we need information about the ports that form the interface of the model component, i.e., name, port type (e.g., `Inport, Outport`), datatype (e.g., `boolean, double, enum, bus`), dimensions of the port and its

width.[5] Our framework provides a mechanism to automatically extract this information from a Simulink model and import it into COCOGEN.

Once imported, the architectural mapping procedure starts (Step 2b), which includes mapping every component and proposition mentioned in a requirement to a model component and a model port path, 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, COCOGEN 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 in COCOGEN, through which the user can pick the path of the corresponding model component or port from a drop-down menu (see Figure 2) and map it with a requirement component or proposition. Then COCOGEN can automatically identify all the other required information (port types, data types, dimensions, etc) to generate correct-by-construction monitors and corresponding traceability information. Alternatively, a user may provide the required information manually.

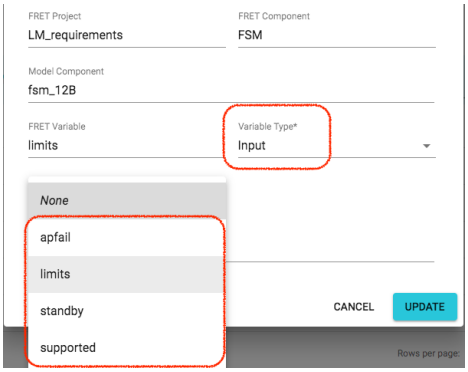

Fig. 2: Limits variable mapping.

Library of pmLTL Operators in Lustre COCOGEN receives as input a pmLTL formula, which it translates into COCOSPEC code. To facilitate this translation, we have created a library of pmLTL operators in Lustre. Before, we go through the library of operators, let us briefly review the two main Lustre operators: 1) the unary right-shift `pre` (for `pre`vious) operator and the binary initialization `->` (for followed-by) operator. 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.

We now present the pmLTL operators `O, H, S, SI`.

```
--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
```

[5] https://www.mathworks.com/help/simulink/slref/common-block-parameters.html

To support timed modifiers that constrain an operator's scope to a specific interval $[l, r]$, we defined additional Lustre nodes. For instance for the timed version of `O`, we added the following nodes to the library:

```
--Timed Once: general case
node OT(const L: int; const R: int; X: bool;) returns (Y: bool);
   var D:bool;
let
   D = delay(X,R);
   Y = OTlore(L-R,D);
tel

--Timed Once: less than or equal to N
node OTlore(const N: int; X: bool; ) returns (Y: bool);
    var C:int;
let
    C = if X then 0
        else (-1 -> pre C + (if pre C <0 then 0 else 1));

    Y = 0 <= C and C <= N;
tel
```

The delay function delays input `X` by `R` time units to define the right bound of the interval in which the valuation of `X` must be checked. Once the input `X` has been delayed by `R` time steps, we can treat the `R` bound as zero and use the `OTlore` (Once Timed less than or equal to) node to check the valuation of `X` in the interval defined by the 0 (current) time step and the left bound `L-R`. `OTlore` is implemented using an integer counter C, which counts the number of time steps that occurred since the last occurrence of property X. If the event has never occurred, the counter keeps its initial value of -1. Below we can see a simple example with $N = 2$ for time steps t=[0..7]:

X	F	F	F	T	F	F	F	F	...
C	-1	-1	-1	0	1	2	3	4	...
Y	F	F	F	T	T	T	F	F	...
t	0	1	2	3	4	5	6	7	...

Other time-constrained operators are defined through `OT` using the usual temporal logic equivalences.

```
-- Timed Historically: general case
node HT(const L: int; const R: int; X: bool;) returns (Y: bool);
let
  Y = not OT(L,R,not X);
tel

-- Timed Since: general case
node ST(const L: int; const R: int; X: bool; Y: bool;)  returns (Z: bool);
let
  Z = S(X, Y) and OT(L,R,X);
tel
```

COCOSPEC Code Generation From the FRETISH version of `[FSM-001]` (Section 4), FRET generates the following pmLTL formula: `H f`, where `f` is a placeholder for `(limits & autopilot)` $\Rightarrow$ `pullup`. The generated COCOSPEC code uses the data types of the input and output variables, provided through the architectural mapping, to generate the contract signature.

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

Below we can see part of the generated code for `roll_hold_mode` from requirement `[AP-004b]`. For the complete code, COCOGEN aggregates all requirements that refer to the same mode to generate all `ensure` and `guarantee` statements.

```
var overshoot : real = (roll - step)/step;
mode roll_hold_mode (
    ensure "AP-004b" overshoot <= 0.1;
    );
```

Verifying COCOSPEC Formalizations We provide assurance that the COCOSPEC code generated by our approach captures the intended semantics. We extend the verification framework provided by FRET to check whether the COCOSPEC code of a requirement conforms to the intended FRET semantics. The FRET semantics [12] is compositionally defined based on different valuations of the FRETISH fields scope, condition, and timing. We call *template key* a combination of values of these fields, e.g., [*in*, *null*, *after*] identifies requirements of the following form: *in mode m, the software shall after 2 seconds satisfy P*. Our framework uses the following FRET components [12]:

- `Trace_Generator`, which uses two approaches to produce traces, i.e., example executions. The first approach uses boundary value analysis and equivalence class to define concrete traces that capture interesting relations of template keys. The second approach is based on random trace generation and produces 60000 different random traces in range [0..12].
- `Oracle`, which takes a trace and a verification pair $\langle t, \phi \rangle$, where t is a template key and ϕ is its corresponding formalization and computes the truth value of t on the trace, based on the FRET semantics of t.

For COCOGEN, we have additionally developed the following components:

- `CoCoSpec_Retriever`, which produces the set of all possible verification pairs $\langle t, \phi \rangle$, that must be checked.
- `CoCoSpec_Evaluator`, which receives a trace, a verification pair $\langle t, \phi \rangle$, and an expected value e from the `Oracle` and checks whether ϕ evaluates to e on the trace. Essentially, it checks whether the generated COCOSPEC code conforms to the template key semantics, for a particular execution trace. For conformance checking, `CoCoSpec_Evaluator` uses the Kind2 model checker.

`Trace_Generator` outputs a trace e. Both ϕ and e are then fed to the Kind2 model checker. We use the *interpreter* Kind2 mode to check conformity, in which Kind2 uses the input valuations from the trace file to evaluate the COCOSPEC formalization at each time step. Since Lustre formalizations are based on past-time formulas, those are evaluated at the end of the trace.

Our verification framework helped us detect discrepancies between the COCOSPEC generated formulas and the intended FRET template key semantics. Despite our deep knowledge of temporal logic operators and their semantics, we

detected a problem regarding our definition of the timed once operator in CoCoSpec. Let us consider the generated formalization ϕ that corresponds to the [*null*,*null*,*within*] template key (no scope, no condition, within timing).

```
node test164_3_null_null_within_CoCoSpec (RES: bool) returns (PROP: bool);
  var FTP: bool = true -> false;
  let
    PROP = H((H(not RES)) => OT(1,0,FTP));
  tel
```

In particular, this means that if `RES` has not occurred yet, we are either within 1 step from `FTP` (First Time Point of trace) or the property is violated. The following discrepancy was reported by our framework over a trace interval [0..12]:

Mode: {[8..11]}; Duration: 2; Response: {[2..4][7..10]}
Discrepancy `null,null,within`: expected: true; Kind2: false.

Since the scope is `null`, ϕ is evaluated throughout the entire trace. Additionally no condition means that `RES` must occur at time steps 0, 1, or 2. Our initial definition of the (`OT`) operator was non-inclusive of the right bound interval, i.e., $[l, r)$ instead of $[l, r]$, which would omit evaluating ϕ at time step 2.

5 Checking requirements against the Simulink model

CoCoSim attaches CoCoSpec contracts to Simulink subsystems. This process relies heavily on CoCoSim's Lustre-to-Simulink compiler. The first compilation step is performed by LustreC [11], an open-source Lustre compiler, produces information necessary to extract the model structure. The second step transforms the produced structure into Simulink blocks, relying on the Simulink API. Each Lustre node is defined as a Simulink subsystem, hence each node call is transformed into an instance of that subsystem in Simulink. Mathematical operators are translated into equivalent Simulink blocks. The `pre` operator is implemented as a Simulink Unit delay block. CoCoSpec constructs (i.e., assume, guarantee, require and ensure) are also compiled and translated: their equivalent Simulink blocks are provided by a dedicated CoCoSim library [6]. Figure 3 illustrates the generated Simulink observer for requirement `[FSM-001]`.

After importing CoCoSpec contracts as Simulink observers, the user may rely on CoCoSim backends to evaluate these contracts; CoCoSim acts as a verification hub, providing easy access to existing solvers. CoCoSim supports checking the validity of requirements against the Simulink model: transforming the model to an equivalent Lustre model with contracts. Kind2 or Zustre model checkers were then used to check the validity of the properties. In case of a counterexample, CoCoSim creates a harness model for the user to simulate the trace of the counterexample, and thus, support model or specification debugging. The contracts can also be transformed into SLDV-compatible verification blocks. SLDV can then be used to prove properties, generate test cases, or evaluate the MC-DC test coverage of a set of tests. The translation technique preserves the structural and modal behavior of the system, allowing us to compositionally analyze it. CoCoSim carefully handles traceability during model analysis, which enables simulating counter-examples on the model.

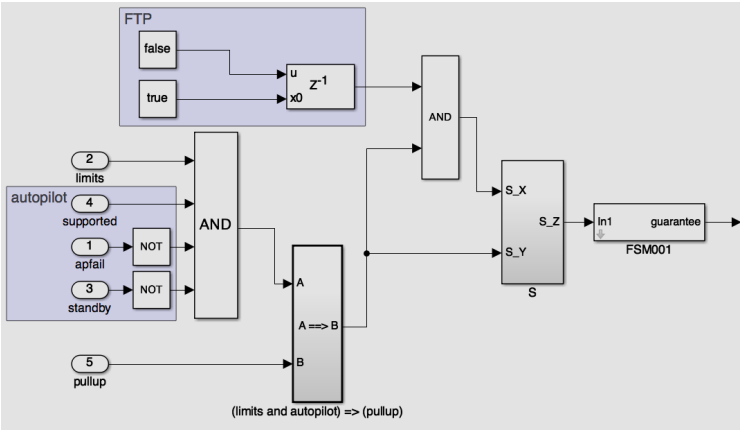

Fig. 3: The generated Simulink version of requirement `[FSM-001]`.

By translating COCOSPEC contracts into executable Simulink blocks, the latter can be used as monitors for simulation, testing, and runtime verification. Additionally, the required condition (activation) of a mode can be used to check whether a mode can be activated or not for a given test suite. Checking the feasibility of a mode activation allows us to perform a form of semantics coverage.

6 LMCPS Challenge Problems and Lessons Learned

The LMCPS case study [9] is representative of flight-critical systems and is publicly available.[6] It includes a set of challenge problems, which range from basic integrators to more complex autopilots. The analysis of these challenge problems is not an easy task due to the use of transcendental functions (e.g., trigonometric functions), nonlinear functions and discontinuous math (e.g. `abs`, inverse of matrices), multi-dimensional signals, and delay blocks. The complete case study and analysis results are presented in our technical report.[7]

For the analysis we used the Kind2 and SLDV tools. The verification results are summarized in Table 1. The analysis was carried out on a MacBook Pro with 3.1 GHz intel Core i7 and 16 GB Memory, with a R2019b Matlab/Simulink, and a v1.1.0 Kind2.

6.1 Lessons Learned

Counterexample interpretation. We found the ability to interpret and trace counterexamples both at the model and requirement levels particularly useful, since for certain cases, we did not need the model to interpret the counterexamples. For instance, in the `FSM` challenge problem we found several pairs of

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

[7] `https://drive.google.com/drive/u/1/folders/1GsKiu_O9_OSK_5XcLZZefi6g9MDAe0CC`

				Kind2		SLDV	
Name	N_R	N_F	N_A	V/IN/UN	t(s)	V/IN/UN	t(s)
Triplex Signal Monitor (TSM)	6	6	6	5/1/0	37.7	5/1/0	26
Finite State Machine (FSM)	13	13	13	7/6/0	141.1	7/6/0	96
Tustin Integrator (TUI)	4	3	3	2/1/0	19.2	2/1/0	19
Control Loop Regulators (REG)	10	10	10	1/5/4	TO	1/0/9	TO
Nonlinear Guidance (NLG)	7	7	7	0/0/7	TO	0/0/7	15
Feedforward Neural Network (NN)	4	4	4	0/0/4	TO	0/0/4	TO
Control Effector Blender (EB)	5	3	3	0/0/3	TO	0/0/3	131
6DoF Autopilot (AP)	14	13	8	5/3/0	40.6	5/3/0	32
System Safety Monitor (SWIM)	3	3	3	2/1/0	25	0/1/2	18
Euler Transformation (EUL)	8	7	7	1/6/0	43	1/0/6	30
Total	74	69	64	23/23/18		21/12/31	

Table 1: LMCPS verification results. N_R: #requirements, N_F: #formalized requirements, N_A: # requirements analyzed by Kind2 and SLDV. Analysis results categorized by **V**alid/**IN**valid/**UN**decided. Timeout (TO) was set to 2 hours.

requirements that were not mutually exclusive, i.e., their preconditions could be simultaneously satisfied leading to unrealizable specifications. This type of analysis, e.g., consistency, realizability checking, can be performed directly at the level of requirements. To perform such analysis, we still need information regarding the type (input, output), and data types of the requirements' propositions, for which we can use COCOGEN to infer[8]/map and automatically generate the specifications.

Requirements elicitation. The requirements were initially written in natural language and as a result their semantics was often ambiguous. For instance, we were not sure how to interpret requirement `[FSM-001]`. Our initial understanding of this requirement was the FRETISH version presented in Section 4, where all conditions must be satisfied at the same time step for `pullup` to be activated. But 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:

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

Table 2: `[FSM-001v2]` Counter-example.

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

Both versions, however, were shown to be invalid.

Reasoning for violated properties. Having a tight integration between requirement and verification activities allowed us to use different approaches to interpret violated properties. In particular, we found useful the combination of

[8] COCOGEN supports automatic type inference directly from FRETISH.

reasoning at the level of requirements and counterexample simulation at the model level. When a property was shown to be invalid, we tried to understand the reason; i.e., is it because of a faulty requirement or a faulty model? Since in most cases, our formalized requirements were invariants of the form `H` $(A \Rightarrow B)$, we used two approaches: 1) check a weaker property, e.g., by strengthening the preconditions, i.e., $A' \subset A$ and check whether the invariant `H` $(A' \Rightarrow B)$ is satisfied, and 2) check feasibility of B with bounded model checking, i.e., `H` $(\neg B)$, in which case the model checker returns counterexamples that could help construct stronger preconditions for B to be satisfied. Our case study showed that using these approaches was helpful for reasoning about violated properties. Furthermore, simulation of counterexamples was 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 2. It is clear that even though `pullup` was activated the first time `limits = true`, it was not activated the second time `limits = true`.

To better understand the behavior of the model, we performed a simulation based on this counterexample. Figure 4 illustrates a scenario when `limits = true` occurs multiple times during the autopilot operation, during which condition `autopilot` must be true. We found that `pullup` is latched only when `limits = true` in the previous step and has not been true for at least three steps before that. Based on this simulation, we were able to refine requirement `FSM-001v2` to form a weaker property that was proven valid.

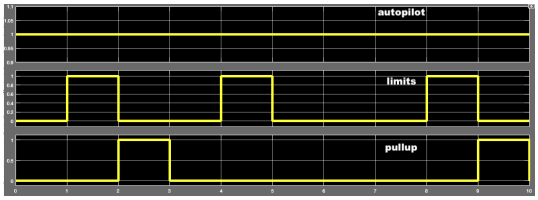

Fig. 4: Simulation of `[FSM-001v2]`

What we gained by using COCOSPEC. COCOSPEC modes introduce structure into the specification. A mode has preconditions that describe the activation of the mode (`R`equires) and actual conditions to be checked (`E`nsures) of the form `H` (`R` $\Rightarrow$ `E`). Our case study showed that it is interesting to check whether the activation of a mode `R` is reachable. If not, the property is trivially true. So, in terms of analysis, showing that `R` is reachable allows us to have a better understanding of whether the property is meaningful for the current model. For instance, we discovered that in the 6DoF Autopilot challenge problem, one of the modes was never reachable.

Scalability of the approach. Our architectural mapping approach allows us to deploy COCOSPEC specifications at different levels of the model behavior. For instance, in the FSM challenge problem, we generated three different contracts that we deployed at three different hierarchical levels of the model. This is especially important for complex models where verification does not scale for global scopes. We applied modular verification to 20 out of the 69 requirements.

Comparison of analysis tools. The features provided by our framework are generic and can be used to integrate a variety of analysis tools, with different

strengths and weaknesses. In our case study we used and compared the Kind2 and SLDV analysis tools, which performed similarly for 7 out of the 10 challenge problems. For the remaining 3 case studies, Kind2 was able to return an answer (valid/invalid) by using abstractions of non-linear functions such as trigonometric functions and the *sqrt* function, in comparison with SLDV that mostly returned undecided. Overall, SLDV was faster, i.e., SLDV returned undecided due to nonlinearities or stubbing in a shorter time than Kind2, which sometimes reached timeout without converging to an answer. One main difference between the two tools is when it comes to verifying requirements in sub-components: SLDV analyzes them against the top level component, whereas Kind2 can be used in both settings, locally on the sub-component level and globally. We experienced an interesting issue in the autopilot case study, which has an algebraic loop in the Simulink model. An algebraic loop occurs when there is a circular dependency of block outputs and inputs in the same time-step. Simulink solves it numerically at each step using the ODE (Ordinary Differential Equation) solver, whereas Lustre forbids such constructs. Strangely, Kind2 was not able to detect the algebraic loop. We contacted a Kind2 developer and confirmed that there is a bug in the algebraic loop detection algorithm.

7 Related work

An example of work aiming to connect assertions to models is the AGREE tool [7]. It provides a specification layer for the AADL architecture language that enables specification of assume/guarantee contracts on components using the Lustre language. The specifications are therefore in a past-time (bounded) temporal logic and are reasoned about using model checkers for Lustre. This enables a compositional validation, at the architecture level, of the components' interactions. In some case, when the underlying component is a Simulink model, the Lustre specification can be evaluated with the SLDV solver thanks to a compilation of the observer as a MATLAB block. The approach, however, does not provide means to support the expression of natural language requirements.

Regarding the expression and formalization of requirements, there are various solutions: from less formal but widely used in the industry to more advanced frameworks. A widespread solution is IBM Rational DOORS, that provides a framework to describe requirements and record their dependencies. This is critical when developing in a certified context where traceability and refinement of requirements play a major role. DOORS typically deals with natural language requirements but does not provide any means to associate a formal semantics nor connect to formal verification.

Without trying to perform formal verification of requirements, the tool Stimulus [15] intends to provide an intermediate solution between pure traceability solutions and formal verification tools, supporting the debugging of requirements, with a dataflow semantics. Using patterns in natural languages, similar to FRET, it provides means to evaluate the consistency of these requirements, as well as the possible lack of specificity. Without providing details on the algorithmic aspects of the approach, a set of test cases, satisfying the requirements is shown to

the user, which may highlight undesired behavior. When associated components are implemented, the formalized requirements can be used as test oracles. The approach is interesting but does not provide insights on the formalization of the properties, the mathematical frameworks used to synthesize the scenarios, nor any connections to model based design tools and formal verification.

The SPIDER tool [18] allows users to incrementally derive, by means of instantiating a grammar, a restricted natural language requirement. The connection to a system is made via a model where its elements can be selected in the built-up requirement. The requirement is translated to a formal logic sentence that can be verified by an analysis tool. The grammar, formalization, connection to software, and formal analysis tool are configurable; for example, in one instance, it supports real-time relevant patterns, metric temporal logic, UML models, and the SPIN [14] model checker and the UML timing analysis tool HYDRA [17]. Our work supports similar real-time patterns, but, metric past-time logic, and the connection to Simulink models is more elaborate.

Last, the SPEAR tool [10] accepts requirements written in a restricted natural language that has a past-time temporal logic semantics. It can then check entailment of properties from assumptions and requirements, and n-step consistency of the assumptions and requirements. It was proposed by the authors of the AGREE framework but no connection is mentioned yet.

8 Conclusion

We described our framework in which requirements are written in a structured natural language and then associated to Simulink model components where they can be analyzed by model-checkers. The proposed approach was applied to substantial challenge problems from Lockheed Martin on which the elicitation and verification tools showed adequate expressivity and verification capabilities. In cases where the verification concluded that the model failed to satisfy a requirement, a counter-example was generated to produce feedback for further model and requirement analysis. The features provided by our framework are generic and can be used to integrate other requirement elicitation and analysis tools. In the future, we plan to automate the reasoning of violated properties approach that we used in LMCPS and also address the analysis of more advanced numerical components, which raised issues due to SMT solver limitations. We also plan on connecting runtime analysis tools to our framework, for cases where model checkers do not scale, or to support analysis during deployment.

References

1. Bradley, A.R.: IC3 and beyond: Incremental, inductive verification. In: Proc. CAV 2012. Springer (2012). https://doi.org/10.1007/978-3-642-31424-7_4
2. Brat, G., Bushnell, D.H., Davies, M., Giannakopoulou, D., Howar, F., Kahsai, T.: Verifying the safety of a flight-critical system. In: Proc. FM 2015 pp. 308–324. Springer (2015). https://doi.org/10.1007/978-3-319-19249-9_20

3. 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). https://doi.org/10.1007/978-3-319-41591-8_24
4. Champion, A., Mebsout, A., Sticksel, C., Tinelli, C.: The Kind 2 model checker. In: Proc. CAV, pp. 510–517. (2016). https://doi.org/10.1007/978-3-319-41540-6_29
5. Cimatti, A., Clarke, E., Giunchiglia, E., Giunchiglia, F., Pistore, M., Roveri, M., Sebastiani, R., Tacchella, A.: NuSMV 2: An opensource tool for symbolic model checking. In: Proc. CAV, pp. 359–364. Springer (2002)
6. CoCo-team: CoCoSim – Automated Analysis Framework for Simulink. `https://github.com/coco-team/cocoSim2`
7. Cofer, D., Gacek, A., Miller, S., Whalen, M.W., LaValley, B., Sha, L.: Compositional verification of architectural models. In: Proc. NfM'12, pp. 126–140. (2012)
8. Dwyer, M.B., Avrunin, G.S., Corbett, J.C.: Patterns in property specifications for finite-state verification. In: Proc. ICSE, pp. 411–420. IEEE (1999)
9. Elliott, C.: An example set of cyber-physical V&V challenges for S5. In: Proc. S5 (2016). `http://mys5.org/Proceedings/2016/Day_2/2016-S5-Day2_0945_Elliott.pdf`
10. Fifarek, A.W., Wagner, L.G., Hoffman, J.A., Rodes, B.D., Aiello, M.A., Davis, J.A.: SpeAR v2.0: formalized past LTL specification and analysis of requirements. In Proc. NfM, pp. 420–426. Springer (2017)
11. Garoche, P., Kahsai, T., Thirioux, X.: LustreC, `https://github.com/coco-team/lustrec`
12. Giannakopoulou, D., Pressburger, T., Mavridou, A., Schummann, J.: Generation of formal requirements from structured natural language (2019), under Submission
13. Halbwachs, N., Caspi, P., Raymond, P., Pilaud, D.: The synchronous data flow programming language Lustre. Proc. IEEE **79**(9), 1305–1320 (1991)
14. Holzmann, G.: Spin Model Checker: Primer and Reference Manual. Addison-Wesley. (2003)
15. Jeannet, B., Gaucher, F.: Debugging Embedded Systems Requirements with STIMULUS: an Automotive Case-Study. In: Proc. ERTS 2016 (2016). `https://hal.archives-ouvertes.fr/hal-01292286`
16. Kahsai, T., Garoche, P., Tinelli, C., Whalen, M.: Incremental verification with mode variable invariants in state machines. In: Proc. NfM'4, pp. 388–402. Springer (2012). https://doi.org/10.1007/978-3-642-28891-3_35
17. Konrad, S., Campbell, L.A., Cheng, B.H.C.: Automated analysis of timing information in UML diagrams. In: Proc. ASE'04, pp. 350–353 (2004)
18. Konrad, S., Cheng, B.H.C.: Facilitating the construction of specification pattern-based properties. In: Proc. RE, pp. 329–338 IEEE (2005). https://doi.org/10.1109/RE.2005.29
19. RTCA: DO-178C: software considerations in airborne systems and equipment certification. (2011)
20. RTCA: DO-333: formal methods supplement to DO-178C and DO-278A. (2011)
21. Souyris, J., Wiels, V., Delmas, D., Delseny, H.: Formal verification of Avionics Software Products. In: Proc. FM, pp. 532–546 (2009). https://doi.org/10.1007/978-3-642-05089-3_34
22. Tripakis, S., Sofronis, C., Caspi, P., Curic, A.: Translating discrete-time Simulink to Lustre. ACM Trans. Emb. Comp. Syst. **4**(4), 779–818 (2005). https://doi.org/10.1145/1113830.1113834