Generation of Formal Requirements from Structured Natural Language

✉ Dimitra Giannakopoulou[1], Thomas Pressburger[1], Anastasia Mavridou[2], and Johann Schumann[2]

[1] NASA Ames Research Center, CA, USA
{dimitra.giannakopoulou,tom.pressburger}@nasa.gov
[2] SGT, NASA Ames Research Center, CA, USA
{anastasia.mavridou, johann.m.schumann}@nasa.gov

Abstract. [**Motivation**] The use of structured natural languages to capture requirements provides a reasonable trade-off between ambiguous natural language and unintuitive formal notations. [**Problem**] There are two major challenges in making structured natural language amenable to formal analysis: 1) associating requirements with formulas that can be processed by analysis tools and 2) ensuring that the formulas conform to the language semantics. [**Results**] FRETISH is a structured natural language that incorporates features from existing research and from NASA applications. Even though FRETISH is quite expressive, its underlying semantics is determined by the types of four fields: *scope*, *condition*, *timing*, and *response*. Each combination of field types defines a template with Real-Time Graphical Interval Logic (RTGIL) semantics. We present an approach that constructs future and past-time metric temporal logic formulas for each template compositionally, from its fields. To establish correctness of our approach we have developed a framework which, for each template: 1) extensively tests the generated formulas against the template semantics and 2) proves equivalence between its past-time and future-time formulas. Our approach has been used to capture and analyze requirements for a Lockheed Martin Cyber-Physical System challenge. [**Contribution**] To the best of our knowledge, this is the first approach to generate pure past-time and pure future-time formalizations to accommodate a variety of analysis tools. The compositional nature of our algorithms facilitates maintenance and extensibility, and our extensive verification framework establishes trust in the produced formalizations. Our approach is available through the open-source tool FRET.

1 Introduction

Requirements engineering is a central step in the development of safety-critical systems. Requirements are typically written in natural language, which is ambiguous and consequently not amenable to formal analysis. On the other hand, a variety of analysis techniques have been developed for requirements written in formal, mathematical notations [4,7,10,11,17], e.g. completeness, consistency,

realizability, model checking, or vacuity checking. Despite the ambiguity of unrestricted natural language, it is unrealistic to expect developers to write high-level requirements in mathematical notations.

FRETISH is a restricted natural language for writing unambiguous requirements, supported by our open source tool FRET[3] (see Figure 1). FRETISH incorporates features from existing approaches (e.g., property patterns [8] and EARS [19]), and from NASA applications. Even though the FRETISH grammar is quite expressive, its underlying semantics is determined by the types of four fields: *scope*, *condition*, *timing*, and *response*. Each combination of field types defines a template with Real-Time Graphical Interval Logic (RTGIL)[21] semantics. There are two challenges in making FRETISH amenable to formal analysis: 1) associating FRETISH requirements with formulas that can be processed by analysis tools, and 2) ensuring that the formulas conform to the FRETISH semantics.

We propose an approach that constructs two metric temporal logic formulas for each FRETISH template: a pure past-time (denoted pmLTL), and a pure future-time (denoted fmLTL) formula, interpreted over finite traces.[4] We support both fmLTL and pmLTL so as to interface with a variety of analysis tools. Formula generation is performed compositionally, based on the types of the template fields. We establish correctness of the produced formalizations through a fully automated framework which, for each template: 1) extensively tests the generated formulas against the template semantics, and 2) proves equivalence between its past-time and future-time formulas. The FRETISH grammar, its RTGIL semantics, the formula generation approach and its verification framework are available through the FRET repository. We report on the application of our approach to the Lockheed Martin Cyber-Physical Systems (LMCPS) challenge [20].

Related Work. Work on gleaning patterns from a body of property specifications resulted in the Specification Pattern System [8], with later extensions for real-time properties [16], composite properties [12], and semantic subtleties [6]. Tools such as Prospec [12], SPIDER [15] and SpeAR [10] were developed to support users in writing requirements according to supported patterns. SALT (Structured Assertion Language for Temporal logic) [3] is a general purpose specification and assertion language designed for readability, which incorporates property pattern features like scope. We use SALT as an intermediate language. The Easy Approach to Requirements Syntax (EARS, [19]) proposed five informal templates that were found to be sufficient to express most high-level requirements; recent work has attempted to formalize the templates in LTL [18]. STIMULUS [14] enables the user to build up a formal requirement by dragging and dropping phrases, and then simulate the system specified by the requirements. ASSERT$^{\text{TM}}$ [7] uses the constrained natural language SADL for formalizing domain ontologies, and a requirements language SRL that can express conditions, including temporal conditions, on monitored variables, and constraints on

[3] Formal Requirements Elicitation Tool: `https://github.com/NASA-SW-VnV/fret`

[4] fmLTL with infinite-trace semantics can be produced with a very simple modification to our generation algorithms.

controlled variables. Tools such as VARED [2] and ARSENAL [13] attempt to formalize more general natural language.

Contributions. Our approach is related to all these works by pursuing similar goals and incorporating experience represented by existing requirement templates and patterns. The main driver for our work is to enable intuitive writing of requirements during early phases of the software lifecycle. We do not require users to define the variables used in requirement sentences; variables can be defined later for analysis, or can be connected to models or code as needed for verification [20]. We are not aware of other work that supports the generation of pure past-time, together with finite- and infinite-trace future-time metric temporal logic formulas. We are thus able to connect to analysis tools that may not support the combination of future and past time operators (e.g., CoCoSim [20]). Our developed algorithms are open source, and their compositional nature facilitates maintenance and extensibility. We currently support 112 templates, which may increase in the future to accommodate the needs of FRET users. Finally, unlike previous work, we provide an extensive, open source, automated verification framework for the correctness of the generated formulas. This is crucial for using FRET in safety-critical contexts.

2 Background

Intermediate Language. SALT [3] serves as an intermediate language in our formula generation approach. In particular, several SALT features facilitate our formalization algorithms: operator qualifiers `inclusive/exclusive` or `required/optional`; scope operators such as `before`, `after`, or `between`; formula simplifications and generation in nuXmv format. Note that we are only able to use scope operators with fmLTL formulas; unfortunately, scope operators in the context of past-time SALT expressions result in formulas with mixed future and past-time operators. Our framework targets pure future-time or pure past-time formulas, i.e., formulas that utilize exclusively future-time or past-time operators, respectively.

We use SALT's propositional operators: `not`, `and`, `or`, `implies`, and the temporal operators: `until`, `always`, `eventually`, `next` for future time, and `since`, `historically`, `once`, `previous` for past-time. A timed modifier: `timed`[$\sim$] where $\sim$ is one of $<$ or $\leq$ turns temporal operators into timed ones (e.g., `once timed`[≤ 3]ϕ). Modifier `timed`[=] is also allowed with `previous` and `next`. It is mandatory to specify whether delimiting events are included (`inclusive`) or not (`exclusive`), and whether their occurrence is strictly `required` or `optional`. For example, ϕ `until inclusive required` ψ means that ϕ needs to hold until and including the point where ψ occurs, and moreover ψ must occur in the execution.

Temporal Logics. fmLTL formulas use exclusively future-time temporal operators (`X, F, G, U`, corresponding to `next`, `eventually`, `always`, `until` in SALT), and look at the portion of an execution that follows the state at which they are interpreted. pmLTL formulas use exclusively past-time temporal operators (`Y,`

`O, H, S`, corresponding to `previous`, `once`, `historically`, `since` in SALT); they look at the portion of the execution that has occurred up to the state where they are interpreted. We interpret formulas over discrete time points. An fmLTL/pmLTL formula is satisfied by an execution if the formula holds at the initial/final state of the execution, respectively.

We review the main future and past time operators for LTL by exploring their dualities. The `X` (resp. `Y`) operator refers to the next (resp. previous) time point, i.e., $\mathtt{X}\phi$ (resp. $\mathtt{Y}\phi$) is true iff ϕ holds at the next (resp. previous) time point.[5] The `F` (resp. `O`) operator refers to at least one future (resp. past) time point, i.e., $\mathtt{F}\phi$ (resp. $\mathtt{O}\phi$) is true iff ϕ is true at some future (resp. past) time point including the present time. $\mathtt{G}\phi$ (resp. $\mathtt{H}\phi$) is true iff ϕ is always true in the future (resp. past). Finally, $\phi\mathtt{U}\psi$ is true iff ψ holds at some point t in the future and for all time points t' (such that $t' < t$) ϕ is true. $\phi\mathtt{S}\psi$ is true iff ψ holds at some point t in the past and for all time points t' (such that $t' > t$) ϕ is true. Our formalizations often use `since inclusive` so, in order to reduce formula complexity, we extend LTL with an operator `SI` where $\phi\,\mathtt{SI}\,\psi \equiv \phi\,\mathtt{S}\,(\psi\,\&\,\phi)$. This feature is only used when the targeted analysis tools support operator `SI`. Timed modifiers restrict the scope of temporal operators to specific intervals. For example, $\mathtt{O}[\leq 3]$ restricts the scope of operator `O` to the interval including the point where a formula is interpreted and 3 points in the past.

3 Requirements Language

The FRETISH language aims at providing a vocabulary natural to the user. As such, the FRETISH grammar offers a variety of ways for expressing semantically equivalent notions; for example, conditions can be introduced using the synonyms while, when, where, and if. While certain aspects of FRETISH requirements are in natural language, Boolean expressions, familiar to most developers, are used to concisely capture conditions on state. Internally, each requirement is mapped to a semantic template, used to construct the requirement's formalization. To illustrate the FRETISH language, we use requirement **[AP-003b]** from the LMCPS challenge (see Section 6):

"In roll_hold mode RollAP shall immediately satisfy
abs(roll_angle) < 6 ⇒ roll_hold_reference = 0."

A FRETISH requirement is automatically parsed into six sequential fields, with the FRET editor dynamically coloring the text corresponding to the fields as the requirement is entered (Figure 1). The fields are *scope*, *condition*, *component*, *shall*, *timing*, and *response*, three of which are optional: *scope*, *condition*, and *timing*. The mandatory *component* field specifies the component that the requirement applies to (e.g., RollAP, the roll autopilot). The *shall* keyword states that the component behavior must conform to the requirement. The *response* field currently is of the form *satisfy* R, where R is a non-temporal Boolean-valued expression. The three optional fields above specify when the response is, or is not, to occur, which we now describe.

[5] $\mathtt{Y}p$ is false at the first time point, for all p.

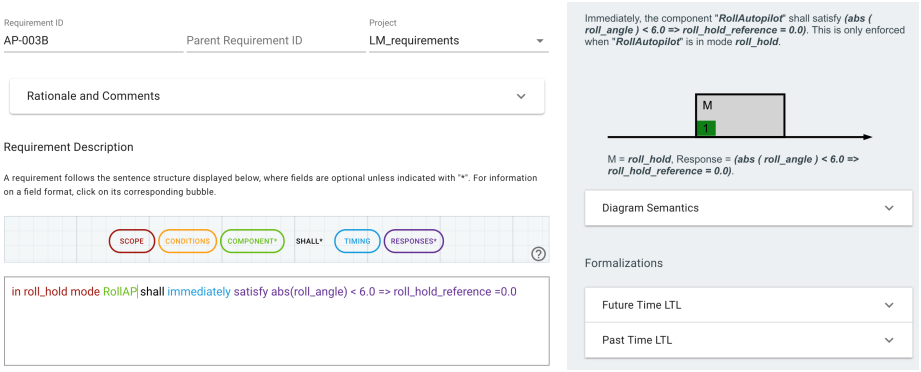

Fig. 1. FRET screenshot: editor (left) and semantics (right) for requirement [**AP-003b**]. Semantics is provided in intuitive textual and diagrammatic forms. The LTL accordions are not expanded to save space but the formulas are displayed in Table 5.

Component behavior is often mode-dependent. Field *scope* specifies the interval(s), relative to when a mode holds, within which the requirement must hold (e.g., "in roll_hold mode"). If scope is omitted, the requirement is enforced on the entire execution, known as *global* scope. For a mode M, FRET provides seven relationships: *before* M (the requirement is enforced strictly before the first point M holds); *after* M (the requirement is enforced strictly after the last point M holds[6]); *in* M (or the synonym *during* M; the requirement is enforced while the component is in mode M); and *not in* M. It is sometimes necessary to specify that a requirement is enforced *only* in some time frame, meaning that it should *not* be satisfied outside of that frame. For this, the scopes *only after*, *only before*, and *only in* are provided.

Field *condition* is a Boolean expression that triggers the need for a response within the specified scope. For example, requirement [**AP-004a**] (Section 6) contains the condition *"when steady_state & calm_air"*. Lastly, field *timing* specifies when the response is expected (e.g., immediately) relative to each trigger (or relative to the beginning of the scope, when condition is omitted). There are seven possibilities for the timing field: *immediately*, *never*, *eventually*, *always*, *within* n time units, *for* n time units[7], and *after* n time units, the latter meaning: not within n time units and at the $n{+}1^{st}$ time unit. When timing is omitted, it is taken to mean *eventually*. To specify that the component shall satisfy R at all times where C holds, one can use Boolean implication in combination with timing *always*, as done in requirement [**AP-001**] (Section 6).

To summarize, we currently support 8 values for field mode (including global scope), 2 values for field condition (condition included or omitted), and 7 values for field timing, for a total of $8 \times 2 \times 7 = 112$ semantic templates. Each template is designated by a *template key*; for example, [*in*, *null*, *always*] identifies requirements of the form *In* M *mode, the software shall always satisfy* R; *null* means

[6] Actually the first occurrence of a last time in the mode; see Section 4.

[7] The *timing* field possibilities correspond to the absence, existence and universality occurrence patterns of [8] and the bounded response and invariance patterns of [16].

the optional condition has been omitted (as opposed to *regular* when a condition is included). The classic response pattern: *always (condition implies eventually response)*, is captured by the key [*null*, *regular*, *eventually*]; *null* means scope is omitted, which corresponds to global scope.

4 Compositional Formalization

Our approach to formalization is compositional: rather than creating a dedicated formula for each semantic template, we build formulas by putting together subformulas corresponding to the types of the template fields. For each semantic template key of FRETISH, we generate an fmLTL and a pmLTL formula; these formulas contain variables that get instantiated for each particular requirement. For example, the template for the key [*in*, *null*, *immediately*] of our example requirement [**AP-003b**] is: `H($Fin_scope_mode$` $\rightarrow$ `$post_condition$)`, and gets instantiated as shown in the last row of Table 5.

Our formalization algorithms produce SALT formulas, and invoke the SALT tool to convert these formulas into nuXmv format. This paper focuses on finite traces so we generate future-time formulas that only check up to the last point of a finite trace, denoted LAST. Due to limited space, we only present our construction of pmLTL formulas; the structure for fmLTL generation is similar but simpler, since it can directly incorporate SALT's support for expressing scope.

Scope. The scope of a requirement characterizes a set of disjoint finite intervals where the requirement must hold, and as such defines a high-level template for the generated formulas. Our approach treats a scope interval as an abstract interval between endpoints LEFT (inclusive) and RIGHT (exclusive), with two semantic options: if LEFT occurs but is never followed by RIGHT, then the interval 1) is not defined (*between* semantics) or 2) is defined and spans to the end of the finite trace (*after-until* semantics). Figure 2(a) illustrates after-until semantics for scope: "in MODE". It characterizes the types of intervals where requirements must hold: 1) intervals defined between *any* point (denoted by the box on the top line of the diagram) where MODE becomes true and the first subsequent point where MODE becomes false; and 2) an interval where MODE is true to the end of the execution. Our pmLTL formulas have the following high-level template:

```
generalform = (g-a) and (g-b)
g-a = historically (RIGHT implies previous BASEFORM_TO_LEFT)
g-b = ((not RIGHT) since inclusive required LEFT)
       implies BASEFORM_TO_LEFT
BASEFORM_TO_LEFT = (BASEFORM [since inclusive required LEFT]*)
```

The template is a conjunction of two formulas. In formula **g-a**, historically imposes the requirement on all intervals of the target scope; previous is needed because intervals are open on the right; BASEFORM_TO_LEFT is the formula BASEFORM that must be checked back to, and including, the left endpoint of each interval. BASEFORM is defined later in the section, and expresses the requirements that must hold within each scope interval; the part within BASEFORM enclosed in $[]^*$

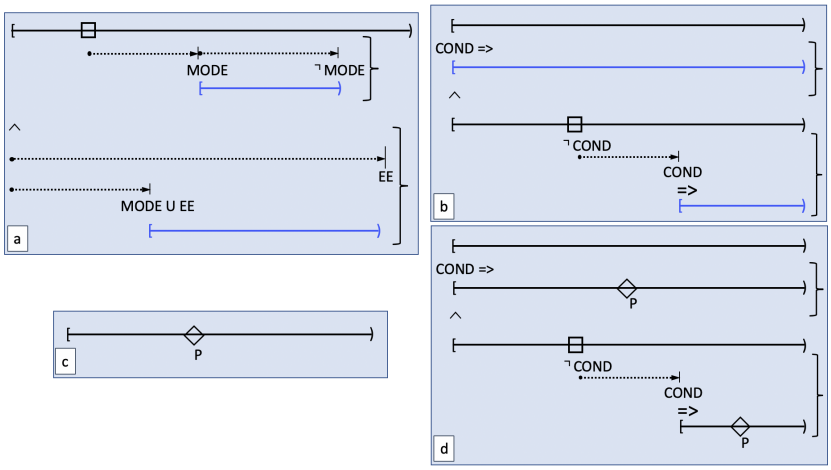

Fig. 2. RTGIL semantics: (a)"in MODE"; (b)"when condition COND"; (c)"eventually P". Our semantics is compositional: the blue interval of a diagram can be replaced by another diagram. For example, (d) illustrates the combined result of (b) and (c), i.e., "when COND, eventually P". EE (end of execution) denotes time point LAST+1.

is omitted in some cases, as discussed later. Formula **g-b** is applicable only with the *after-until* option; it similarly imposes BASEFORM_TO_LEFT on intervals that span to the end of the execution (i.e., RIGHT never occurs).

The endpoints LEFT and RIGHT in our general template get instantiated depending on the type of scope. Table 1 defines scope endpoints in terms of abbreviations (left), each characterized by a logical formula that tracks changes in the values of mode variables M (right). We use abbreviations FiM/LiM: first/last state in mode; FNiM/LNiM: first/last state not in mode; FFiM/FLiM: first occurrence of FiM/LiM in execution; FTP: first time point in execution; LAST: last time point in execution. FiM and LiM are used with scope key *in*; they are characterized by M becoming true (from false) and vice versa, respectively. Endpoint formulas may involve checking whether endpoints occur at state FTP (e.g., when (M `and` FTP) is true, FiM holds).

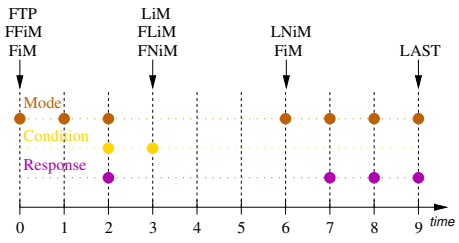

Fig. 3. Example execution including graphical representation of endpoints used in pmLTL scope semantics

Figure 3 provides an example system execution including mode-related information, and depicting the different types of endpoints used in defining scopes. For scope *after*, LEFT is time point 3, and RIGHT is time point 10 (LAST+1). As mentioned, our intervals are open to the right: [LEFT, RIGHT); this is because the RIGHT endpoint of scopes can only be detected one time point later (see Table 1). For example, in Figure 3, the

last point in the first mode interval is 2, but LiM is detected at time point 3, where M is false, but was true at the previous time point. As mentioned in Section 2, qualifier *only* expects the requirement to *not* hold outside of the specified scope. This means two things: 1) scope interval endpoints must be selected accordingly (Table 1), and 2) the base formula must be negated.

Note that for scopes *null*, *after*, and *only before*, the right endpoint of their associated intervals is LAST+1 (see Table 1). Since past-time formulas get evaluated backwards starting at the last point in an execution, we do not need to provide a formula for LAST+1. Rather, for these cases, we simplify the general template to the following: **generalform** = `(once LEFT) implies BASEFORM_TO_LEFT`.

Scope	LEFT	RIGHT
null	FTP	LAST+1
before	FTP	FFiM
after	FLiM	LAST+1
in	FiM	LiM
notin, onlyin	FNiM	LNiM
only before	FFiM	LAST+1
only after	FTP	FLiM

Symbol	Formula
FFiM	FiM `and previous (historically (not` M`))`
FLiM	LiM `and previous (historically (not` LiM`))`
FiM	M `and` (FTP `or (previous not` M`))`
LiM	`not` M `and previous` M
FNiM	`not` M `and` (FTP `or previous` M`)`
LNiM	M `and previous (not` M`)`
FTP	`not previous true`

Table 1: (left) Scope endpoints. (right) pmLTL formulas associated with each endpoint. LAST+1 is not provided because our formulas do not use it.

Timing	BASEFORM	BASEFORM with conditions
immediately	LEFT `implies` RES	TRIGGER `implies` RES
always	[RES `since_ir` LEFT]*	NOCONDITION `or` (RES `since_ir` TRIGGER)
never	[always(`not` RES)]*	always(COND, (`not` RES))
eventually	[`not` ((`not` RES) `since_ir` LEFT)]*	[NOCONDITION `or` `not` ((`not` RES) `since_ir` TRIGGER)]*
for n	(`once timed`[$\leq n$] LEFT) `implies` RES	F_1 `and` F_2 $F_1 \equiv$ (((`not` LEFT) `since_er` TRIGGER) `and` (`once timed`[$\leq n$] TRIGGER)) `implies` RES $F_2 \equiv$ (COND `and` LEFT) `implies` RES
within n	((`not` RES) `since_ir` LEFT) `implies` (`once timed`[$<n$] LEFT)	(`previous timed`[$=n$] (TRIGGER `and not` RES)) `implies` (`once timed`[$<n$] (LEFT `or` RES))
after n	for(n, `not` (RES)) `and` within(n+1, RES)	for(COND, n, `not` (RES)) `and` within(COND, n+1, RES)

Table 2: BASEFORMS without and with conditions. `since_ir`/`since_er` denote `since inclusive/exclusive required`, respectively.

Base formulas. BASEFORM describes the expectations of the requirement within each scope interval. We remind the reader that all BASEFORM formulas appear in the context of **generalform** and are interpreted starting at the RIGHT of each scope. A base formula is determined by whether a condition exists, the timing, and the type of response.

Table 2 illustrates the base formulas that correspond to various timings, without, and with conditions. A base formula f enclosed in $[f]^*$ indicates that the part in BASEFORM_TO_LEFT similarly enclosed in $[]^*$ must be omitted; for example, eventually formulas cannot be checked at each point of the interval. Some timings are expressed in terms of others (e.g., never); we use a function-like notation to denote that. Timed cases have special treatment when the remaining interval in scope is too short to cover their duration. Take the trace of Figure 3, for example. At time point 8: 1) *for* 3 time units imposes RES to the end of the execution; 2) *within* 3 time units is trivially true; 3) *after* 3 time units imposes that RES *not* occur until the end of the execution.

There are several options for interpreting conditions: is a requirement triggered by a condition *being* or *becoming* true? We currently only support the latter option, as illustrated in Figure 2: we check requirements when a condition becomes true (from false) or is true at the first point where the requirement is in scope, as expressed by a trigger formula: `TRIGGER = (COND and previous not(COND)) or (COND and LEFT)`. We can easily add support for different options by providing alternative trigger formulas. When conditions never occur in a scope of interest, then the requirement is true trivially. Base formulas with conditions therefore typically contain a disjunction with `NOCONDITION` (Table 2), where `NOCONDITION = (not COND since inclusive required LEFT)`.

Finally, note that negating base formulas in *only* scopes does not always consist of wrapping the formula in a logical *not*. For this reason, negations of timings are specified explicitly in our approach (not illustrated for lack of space).

5 Verifying Formalizations

We provide assurance that formulas generated by our approach capture the intended semantics through a modular and extensible verification framework. For each template key and its corresponding fmLTL and pmLTL formulas ϕ_{ft} and ϕ_{pt}, our framework 1) checks that ϕ_{ft} and ϕ_{pt} conform to the template key RTGIL semantics, and 2) verifies for a specified trace length that ϕ_{ft} and ϕ_{pt} are equivalent. Our verification framework consists of the following components:

• TRACE_GENERATOR produces traces, i.e., example executions such as the one illustrated on Figure 3: mode M holds in intervals {[0..2],[6..9]}, condition COND holds in the interval {[2..3]}, and response RES holds in intervals {[2..2], [7..9]}.

• FORMULA_RETRIEVER produces the set of all possible verification tuples $\langle t, \phi_{ft}, \phi_{pt} \rangle$, where t is a template key, and ϕ_{ft} and ϕ_{pt} are its corresponding fmLTL and pmLTL formulas, respectively. This component establishes the set of formulas that must be checked by our framework.

• ORACLE takes a trace and a verification tuple $\langle t, \phi_{ft}, \phi_{pt} \rangle$, and computes the truth value of t on the trace, in terms of RTGIL semantics. For example, for template key [*in*, *null*, *always*] and the trace of Figure 3, the expected value is `false`, because when M is active in interval [0..2], RES does not hold on the entire interval.

• SEMANTICS_EVALUATOR receives a trace, a verification tuple $\langle t, \phi_{ft}, \phi_{pt} \rangle$, and an expected value e (provided by ORACLE), and checks whether ϕ_{ft} and ϕ_{pt} evaluate to e on the trace. In other words, it checks if, in the context of the particular trace, the generated formulas conform to the template key semantics.

• EQUIVALENCE_CHECKER receives a verification tuple $\langle t, \phi_{ft}, \phi_{pt} \rangle$, and checks whether ϕ_{ft} and ϕ_{pt} are equivalent formulas, thus ensuring consistency between different formalizations of the same template key.

5.1 Trace Generation

We support two approaches for trace generation: the first targets interesting relationships between mode, condition, response, and duration (for metric timing), while the second uses a random approach. Our framework is designed in a highly modular way, so additional strategies can easily be incorporated.

The first approach uses boundary value analysis and equivalence class strategies similar to [22], with the difference that we generate traces automatically as opposed to manually, and we additionally deal with durations for metric timing. We base trace generation on specifying numerical constraints on endpoints for mode, condition, and response. We then use constraint logic programming[8] to compute all solutions satisfying the constraints. These solutions define concrete traces used by our framework.

A trace spans between time points 0 and *Max*. We first select a point x where a condition trigger is imposed, with $0 \leq x \leq Max$. We optionally add another trigger point a fixed distance away. Condition intervals are currently of length 1 (for example, [5..6]). We then generate a mode interval $[x_1..x_2]$ where $0 \leq x_1 \leq x_2 \leq Max$ around the first trigger point according to boundary value and equivalence class testing strategies. In particular, we generate constraints on x_1 and x_2 where $x_1 = x$, or $x_1 + 1 = x$ (boundary cases), or x is the midpoint of x_1 and x_2 (to represent the equivalence class of interior points between x_1 and x_2), or $x_2 = x$ (another boundary case).

We also generate traces with a second scope interval $[x_3..x_4]$ (where $x_3 > x_2 + 1$) based on a selected duration n. There are several cases for time point $x + n$: it could lie between x_1 and x_2, be x_2, be between x_2 and x_3, be x_3, be between x_3 and x_4, be x_4, or be greater than x_4. Next, we explore response intervals that implement each of the Allen interval relationships [1] to each mode interval, merging pairs of response intervals that are not separated. This process generates, for example: 1908 traces with $Max = 6$ and duration= 2; 12562 traces with $Max = 9$ and duration $= 4$ (the example of Figure 3 is one of those); and 32717 traces with $Max = 12$ and duration $= 4$.

Random trace generation constructs a random number, between 0 and 3, of random, disjoint, non-consecutive intervals between 0 and *Max*, for each of mode, condition, and response. It also generates a random duration for metric timings. We used thus produced 60000 different random traces in the range [0..12].

[8] We use clp(fd) in SWI-Prolog: https://www.swi-prolog.org/

5.2 Test oracles

ORACLE interprets the RTGIL semantics of a template key on a trace generated as above and produces an expected value of `true` or `false`. It performs this in a compositional fashion, which reflects the way in which the corresponding RTGIL semantics is defined. More specifically, fields scope and condition determine the intervals within a trace where the template is relevant, and fields timing and response determine the corresponding `true` or `false` value, as follows.

The first step consists of establishing the scope of the requirement as a set of intervals where the requirement must be evaluated. This is performed based on the trace and the type of field *scope*. Take, for example, the trace illustrated in Figure 3, where M holds in intervals {[0..2],[6..9]}. If the scope field is *after* or *in*, then the scope of the requirement is {[3..9]} or {[0..2],[6..9]}, respectively.

If the condition field is *regular*, then the intervals where the requirement must be evaluated get modified accordingly, based on the trigger point for the condition. The trigger point is computed as the first point where a scope interval intersects some condition interval. This could be the left endpoint of the scope interval, some other point within the interval, or no point, if the condition never holds within that interval. For example, in Figure 3 where COND holds in the interval {[2..3]}, if the scope is {[0..2],[6..9]}, the condition triggers are time point 2 for [0..2], and none for [6..9]. As a consequence, the requirement must only be evaluated in interval [2..2]; this is established by truncating interval [0..2] to start at the condition trigger 2, resulting in interval [2..2].

Timing and response fields determine the `true` or `false` value produced by the oracle through appropriate interval operations for each of the timing operators. Note that the timing constraints are applied to *each* interval in the scope, and the results are combined to establish the returned value. At a high level, our approach is based on interval operations, which we have implemented in a generic interval logic class. We discuss a few examples here to provide the intuition behind this step. For the trace illustrated in Figure 3 and for a template key with scope field *in*, requirements must be evaluated in intervals {[0..2],[6..9]}. Let us focus on interval [0..2], where similar steps are applied to the second interval [6..9].

First, consider the case where the condition is *null*, i.e., the requirement must hold unconditionally. For timing *always*, our algorithm checks whether there exists some interval in the set of response intervals that includes interval [0..2], resulting in `false` (since RES holds in intervals {[2..2], [7..9]}). For *eventually*, it checks whether there exists some interval in the set of response intervals that is not disjoint with interval [0..2], resulting in `true`. For timing field *within* and duration 1, we truncate [0..2] to interval [0..1] that has the specified duration, and within which we expect the response to occur. We then check whether there exists some interval in the set of response intervals that is not disjoint with the truncated interval [0..1], resulting in `false`.

If the condition field is *regular*, then we need to take the condition trigger into consideration. If there exists no condition trigger in the scope interval (e.g., [6..9]), then the result is vacuously `true`. For interval [0..2], the trigger is 2. As

mentioned, the scope interval is then truncated to start at the condition trigger, meaning to [2..2], and timing operators are applied similarly as before, but this time on interval [2..2]. For timing field *always* and *eventually*, our algorithm returns `true`; *within* with duration 1 falls outside the range of the original scope interval, and hence also returns `true` (i.e., the remaining interval in scope is too short to cover duration).

Since requirements are expected to hold in all scope intervals, our oracle computes the expected result as the conjunction of the results obtained for each interval. For example, in the case of template key [*in*, *null*, *always*], the result is `false` for scope interval [6..9], and `false` for [0..2], so the expected value is `false`.[9] Note that *only* scopes involve negating the body of the requirement, which our ORACLE also supports.

5.3 Testing and Verification

Components SEMANTICS_EVALUATOR and EQUIVALENCE_CHECKER use the model checker nuXmv. Given a trace and a verification tuple $\langle t, \phi_{ft}, \phi_{pt} \rangle$, SEMANTICS_EVALUATOR encodes the trace in nuXmv and evaluates the truth value of formulas ϕ_{ft} and ϕ_{pt} on the trace. Our framework subsequently checks if the truth values of ϕ_{ft} and ϕ_{pt} agree with the expected value computed by ORACLE.

The code listing below is the nuXmv code generated for the trace of Figure 3. The intervals for mode, condition and response involved in a trace correspond in nuXmv to definitions of propositions (see lines 7 through 22 in Listing 1.1). Following the define clause are future-time and past-time formalizations of each template key to be checked, represented as $\phi(arguments)$ in Listing 1.1. The future-time formulas are evaluated at the beginning of time ($t = 0$) at line 24, and the past-time formulas are evaluated at the end of time ($t = 9$) at line 26.

Listing 1.1. nuXmv Input for $\phi(\cdot)$

```
1  MODULE main
2  VAR    t : 0 .. 10;
3  ASSIGN init(t):=0;
4         next(t):=(t >= 10)?10:t+1;
5  DEFINE
6         LAST := (t = 9);
7         MODE := case
8            t < 0 : FALSE;
9            t <= 2 : TRUE;
10           t < 6 : FALSE;
11           t <= 9 : TRUE;
12           TRUE : FALSE; esac;
13        COND := case
14           t < 2 : FALSE;
15           t <= 3 : TRUE;
16           TRUE : FALSE; esac;
17        RES := case
18           t < 2 : FALSE;
19           t <= 2 : TRUE;
20           t < 7 : FALSE;
21           t <= 9 : TRUE;
22           TRUE : FALSE; esac;
23 LTLSPEC NAME F0_ft_key :=
24   G((t=0)->φ_ft(LAST,MODE,COND,RES));
25 LTLSPEC NAME F1_pt_key :=
26   G((t=9)->φ_pt(MODE,COND,RES));
```

Given a verification tuple $\langle t, \phi_{ft}, \phi_{pt} \rangle$, EQUIVALENCE_CHECKER uses nuXmv to check $(\mathrm{G}\,(LAST \Rightarrow \phi_{pt})) \Leftrightarrow \phi_{ft}$ over an unconstrained model of specified trace length (for example, length 10 in Listing 1.2. Formulas ϕ_{ft} and ϕ_{pt} are instantiated with the unconstrained nuXmv Boolean variables `mode`, `cond`, and `response`; moreover, a specific duration (say, 3) is chosen for metric timings.

[9] Had the scope interval [6..9] been [7..9] instead, the result would have been `true` for that interval, but still `false` for the result.

Listing 1.2. Equivalence checking

```
MODULE main
VAR     t : 0 .. 10;
        mode, cond, res : boolean;
ASSIGN init(t):=0;
        next(t):=(t >= 10)?10:t+1;
DEFINE LAST := (t = 9);
LTLSPEC NAME FO_key :=
G(LAST -> φ_pt(mode,cond,res))
<-> φ_ft(LAST,mode,cond,res);
```

Despite our high expertise with formal logics, our verification framework was central for detecting errors in our produced formalizations. The compositional nature of our algorithms simplifies formalization repairs: changes target particular fields and automatically affect all templates that include these fields. In the following, we describe a very subtle problem detected by our framework, concerning the formalization for conditions with *within* timing, for which the BASEFORM formula was originally:

(((**not** RES) **and** (**not** LEFT)) **since exclusive required** ((**not** RES) **and** TRIGGER)) **implies** (**once timed**[$<n$] TRIGGER)

In other words, if within the target scope interval, no RES occurs since and including TRIGGER, TRIGGER must occur less than n time points in the past, otherwise *within* is violated. The following discrepancy is reported by our verification framework for the pmLTL formula over a trace interval [0..12]:
Mode: {[0..1][5..10]}; Condition: {[1..2],[4..5]}; Duration: 4; Response: {[0..0],[6..10]}
Discrepancy *null,regular,within*: expected: false; nuXmv: true.

Scope *null* signifies that the requirement is evaluated in the entire trace interval. The condition is triggered at points 1 and 4. The trigger at point 1 requires RES to occur within 4 time points, i.e., by, or at, time point 5. Despite the fact that the response does not occur in that interval, the formula evaluates to true. The reason is that the above formula states that if RES does not hold since TRIGGER, then TRIGGER must occur in less than 4 time units. Unfortunately, since TRIGGER also holds at time point 4, it satisfies the formula. Indeed, it is not possible to identify which TRIGGER the formula refers to in order to avoid this problem. To address it, we used the timed equality operator **previous timed**[$=n$]. The formula of Table 2 removes all discrepancies associated with this error.

6 Lockheed Martin Cyber Physical Systems Challenge

We applied FRET to the publicly-available Lockheed Martin Cyber Physical Systems (LMCPS) challenge [9]. The requirements, given in natural language, were formulated in FRETISH. The Simulink models, included with the challenge, were verified against the formulas generated by FRET. The case study aimed to assess the expressiveness of FRETISH, the quality of produced formalizations, and the capability of FRET to drive analysis tools. Table 3 provides an overview of the detailed study [20]: we found that most requirements could be captured in FRETISH and FRET successfully produced formalizations for analysis tools.

We also studied the conciseness of formulas generated by FRET compared to equivalent[10] formulas produced by hand starting from the original natural language requirements. We observed that, for elaborate semantic templates, writing

[10] Equivalence of formulas was checked with Kind2. [5]

Component	N_R	N_F	N_A
Triplex Signal Monitor (TSM)	6	6	6
Finite State Machine (FSM)	13	13	13
Tustin Integrator (TUI)	4	3	3
Control Loop Regulators (REG)	10	10	10
Nonlinear Guidance (NLG)	7	7	7
Feedforward Neural Network (NN)	4	4	4
Control Allocator Effector Blender (EB)	5	3	3
6DoF Autopilot (AP)	14	13	13
System Safety Monitor (SWIM)	3	3	3
Euler Transformation (EUL)	8	7	7

Table 3: LMCPS summary. N_R : #requirements; N_F : #requirements expressed in FRETISH; N_A : #requirements for which FRET produced verification code.

formulas was hard and error-prone; for simple semantic templates, hand-written formulas could be significantly more concise. Motivated by these findings, we implemented a rewriting engine that applies Boolean algebra and temporal logic simplifications to reduce the complexity and size of produced formulas.

We discuss three requirements of increasing complexity. These are part of the "6DoF Autopilot" challenge, which concerns an aircraft autopilot (AP) system featuring several modes and commands under various conditions. The challenge includes components *Autopilot*, and the *RollAP* unit of the AP. In **[AP-001]** (Table 4), signal `ap_engaged` indicates whether the AP is active (engaged) or not; `roll_act_cmd` denotes the numeric output signal to the aircraft control surfaces for roll. The last row of Table 4 illustrates the significantly more concise formula produced by the rewriting engine as compared to the original formula above it.

When roll AP is not engaged, the command to the roll actuator shall be zero.
RollAP shall always satisfy !ap_engaged ⇒ roll_act_cmd = 0.0
`(!ap_engaged → roll_act_cmd = 0.0) S` `((!ap_engaged → roll_act_cmd = 0.0) & FTP)`
`H(!ap_engaged → roll_act_cmd = 0.0)`

Table 4: **[AP-001]**: natural language, FRETISH, pmLTL, simplified pmLTL

Requirement **[AP-003b]** (Table 5, Fig. 1) describes the conditions that must be satisfied in the *roll hold* mode of operation and belongs to the [*in*, *null*, *immediately*] template key. Here, *Fin_roll_hold*, *Lin_roll_hold* are as described in Table 1 for M = *roll_hold*. The *immediately* timing was used to specify that the response must be satisfied at the time of roll hold mode engagement. For this template key, the complicated pmLTL formula is equivalent to the formula in the last row of the table. We could not devise rewriting rules to achieve this result, so we added a special case in the formula generation algorithms. Finally, **[AP-004a]** (Table 6) talks about conditions that must be satisfied when commands are sent in the *roll_hold* mode. The displayed pmLTL formula using operator `SI` is over 3x shorter than the corresponding formula using operator `S`.

The roll hold reference shall be set to zero if the actual roll angle is less than 6 degrees, in either direction, at the time of roll hold engagement.
In roll_hold mode RollAP shall immediately satisfy abs(roll_angle) $<$ 6.0 $\Rightarrow$ roll_hold_reference = 0.0
`((LAST V ((! (Fin_roll_hold & (! LAST)))	(X (abs_roll_angle < 6.0 → roll_hold_reference = 0.0)))) & (roll_hold → (abs_roll_angle < 6.0 → roll_hold_reference = 0.0)))`
`((H ((Lin_roll_hold & (! FTP)) → (Y ((Fin_roll_hold → (abs_roll_angle < 6.0 → roll_hold_reference = 0.0)) S ((Fin_roll_hold → (abs_roll_angle < 6.0 → roll_hold_reference = 0.0)) & Fin_roll_hold))))) & (((! Lin_roll_hold) S ((! Lin_roll_hold) & Fin_roll_hold)) → ((Fin_roll_hold→ (abs_roll_angle < 6.0 → roll_hold_reference = 0.0)) S ((Fin_roll_hold → (abs_roll_angle < 6.0 → roll_hold_reference = 0.0)) & Fin_roll_hold))))`
`(H (Fin_roll_hold → (abs_roll_angle < 6.0 → roll_hold_reference = 0.0)))`

Table 5: [**AP-003b**]: natural language, FRETISH, fmLTL, pmLTL, equivalent pmLTL

Steady state roll commands shall be tracked within 1 degree in calm air.
When in roll_hold mode when steady_state & calm_air AP shall always satisfy abs(roll_err) $\leq$ 1.0
`(H ((Lin_roll_hold & (!FTP)) → (Y ((((!(steady_state & calm_air)) SI Fin_roll_hold)	((abs(roll_err) <=1.0) SI (((steady_state & calm_air) & (Y (!(steady_state & calm_air))))	((steady_state & calm_air) & Fin_roll_hold)))) SI Fin_roll_hold)))) & (((!Lin_roll_hold) SI Fin_roll_hold) → ((((!(steady_state & calm_air)) SI Fin_roll_hold)	((abs(roll_err) <=1.0) SI (((steady_state & calm_air) & (Y (!(steady_state & calm_air))))	((steady_state & calm_air) & Fin_roll_hold)))) SI Fin_roll_hold))`

Table 6: [**AP-004a**]: natural language, FRETISH, pmLTL

7 Conclusions

We presented a compositional approach for generating and verifying formalizations of structured natural language requirements. Such modularity is key for maintainability and extensibility. We have also developed an automated verification framework for the formulas that we generate. Despite our high degree of expertise in temporal logics, automated verification has been key for detecting subtle errors in our algorithms. Our approach may produce more complex formulas than could be custom-written for individual template keys. We implement several formula simplification steps, which we will further improve in the future; in particular, we will focus on templates that occur most often in practice.

We plan to extend FRETISH with responses that involve ordering of actions, and conditions that persist for some time interval. Moreover, we intend to support customization of FRETISH to fit domain-specific styles and towards including other requirement notations such as tables or finite-state machines. Finally, we are exploring natural-language processing in order to fit existing requirements within the templates supported by FRET. We are also extending FRET towards providing user-support in correcting requirements.

Acknowledgements. We gratefully acknowledge the NASA ARMD System-Wide Safety Project for funding this work.

References

1. Allen, J.F.: Maintaining knowledge about temporal intervals. CACM **26**(11), 832–843 (Nov 1983).
2. Badger, J., Throop, D., Claunch, C.: VARED: Verification and analysis of requirements and early designs. In: RE'14. pp. 325–326 (2014).
3. Bauer, A., Leucker, M.: The theory and practice of SALT. In: Proc. NfM'11. pp. 13–40. Springer (2011)
4. Bloem, R., Cavada, R., Pill, I., Roveri, M., Tchaltsev, A.: RAT: A tool for the formal analysis of requirements. In: Proc. CAV'07, LNCS, vol. 4590, Springer (2007).
5. Champion, A., Mebsout, A., Sticksel, C., Tinelli, C.: The Kind2 model checker. In: Proc. CAV'16, pp. 510–517. Springer (2016)
6. Cobleigh, R.L., Avrunin, G.S., Clarke, L.A.: User guidance for creating precise and accessible property specifications. In: Proc. SIGSOFT'06/FSE-'14. ACM (2006).
7. Crapo, A., Moitra, A., McMillan, C., Russell, D.: Requirements capture and analysis in ASSERT(TM). In: RE'17. pp. 283–291 (2017).
8. Dwyer, M.B., Avrunin, G.S., Corbett, J.C.: Patterns in property specifications for finite-state verification. In: Proc. ICSE'99. pp. 411–420. ACM (1999).
9. Elliott, C.: An example set of cyber-physical V&V challenges for S5. Lockheed Martin Skunk Works. In: Proc. S5'16. AFRL (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: NfM'17. pp. 420–426 (2017).
11. Gacek, A., Katis, A., Whalen, M.W., Backes, J., Cofer, D.D.: Towards realizability checking of contracts using theories. In: Proc. NfM'15. LNCS, vol. 9058, pp. 173–187. Springer (2015).
12. Gallegos, I., Ochoa, O., Gates, A., Roach, S., Salamah, S., Vela, C.: A property specification tool for generating formal specifications: Prospec 2.0. SEKE'08, pp. 273–278 (2008)
13. Ghosh, S., Elenius, D., Li, W., Lincoln, P., Shankar, N., Steiner, W.: ARSENAL: automatic requirements specification extraction from natural language. In: Proc. NfM'16. LNCS, vol. 9690, pp. 41–46. Springer (2016).
14. Jeannet, B., Gaucher, F.: Debugging Embedded Systems Requirements with STIMULUS: an Automotive Case-Study. In: ERTS'16. (2016),
15. Konrad, S., Cheng, B.H.C.: Facilitating the construction of specification pattern-based properties. In: Proc. RE'05. pp. 329–338. IEEE (2005).
16. Konrad, S., Cheng, B.H.C.: Real-time specification patterns. In: Proc. ICSE'05. pp. 372–381. ACM (2005).
17. Kupferman, O., Vardi, M.Y.: Vacuity detection in temporal model checking. International Journal on Software Tools for Technology Transfer **4**(2), 224–233 (2003).
18. Lúcio, L., Iqbal, T.: Formalizing EARS – first impressions. In: 1st International Workshop on Easy Approach to Requirements Syntax (EARS). pp. 11–13 (2018).
19. Mavin, A.: Listen, then use EARS. IEEE Software **29**(2), 17–18 (2012).
20. 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, NASA (2019).
21. Moser, L.E., Melliar-Smith, P.M., Ramakrishna, Y.S., Kutty, G., Dillon, L.K.: The real-time graphical interval logic toolset. In: CAV'96. pp. 446–449. Springer (1996).
22. Salamah, S., Gates, A., Roach, S., Mondragon, O.: Verifying pattern-generated LTL formulas: A case study. In: Proc. SPIN'05. pp. 200–220. Springer (2005).