A Compositional Proof Framework for FRETish Requirements*

Esther Conrad
NASA Langley Research Center
Hampton, VA, USA
esther.d.conrad@nasa.gov

Laura Titolo
National Institute of Aerospace
Hampton, VA, USA
laura.titolo@nianet.org

Dimitra Giannakopoulou
NASA Ames Research Center
Moffett Field, CA, USA
dimitra.giannakopoulou@nasa.gov

Thomas Pressburger
NASA Ames Research Center
Moffett Field, CA, USA
tom.pressburger@nasa.gov

Aaron Dutle
NASA Langley Research Center
Hampton, VA, USA
aaron.m.dutle@nasa.gov

Abstract

Structured natural languages provide a trade space between ambiguous natural languages that make up most written requirements, and mathematical formal specifications such as Linear Temporal Logic. FRETish is a structured natural language for the elicitation of system requirements developed at NASA. The related open-source tool FRET provides support for translating FRETish requirements into temporal logic formulas that can be input to several verification and analysis tools. In the context of safety-critical systems, it is crucial to ensure that a generated formula captures the semantics of the corresponding FRETish requirement precisely. This paper presents a rigorous formalization of the FRETish language including a new denotational semantics and a proof of semantic equivalence between FRETish specifications and their temporal logic counterparts computed by FRET. The complete formalization and the proof have been developed in the Prototype Verification System (PVS) theorem prover.

CCS Concepts: • **Theory of computation** → **Proof theory**; **Modal and temporal logics**; **Logic and verification**.

Keywords: Metric Temporal Logic, Structured Natural Language, Requirements, Formal Proofs, PVS

1 Introduction

Natural language requirements are typically ambiguous and not amenable to be input to formal methods tools. Conversely, formal mathematical notations are unambiguous but they require domain-specific expertise and can be unintuitive and hard to specify. Structured natural languages provide a good trade-off between natural language and formal mathematical notation. FRETish [14] is a restricted structured natural language developed at NASA for writing unambiguous requirements. A FRETish requirement is composed by five fields: scope, condition, component, timing, and response. The Formal Requirements Elicitation Tool (FRET) [13] provides support for writing specifications in FRETish and for generating corresponding metric temporal logic (MTL) formulas that can be input to several formal verification tools. FRET currently outputs formulas in the language of NuSMV [6] and in CoCoSpec [5] syntax. To improve the confidence in the correctness of the generated formula, an extensive automated testing framework has been presented in [15]. Testing increases the confidence in the tool, but cannot guarantee full coverage of all cases. In order to use FRET in a safety-critical context, such as the software architecture for UAV presented in [9], it is ideal to formally guarantee that the semantics of the FRETish requirement is preserved in the generated temporal logic specification.

This paper presents a rigorous formalization of the FRETish language. This formalization includes a denotational semantics for the FRETish language, and a rigorous proof of correctness of the MTL formula generation algorithm implemented in FRET. The denotational semantics maps a FRETish requirement into a set of traces where each state is a set of formulas that holds at a certain point in time. The proof of correctness

ensures that a trace belongs to the semantics of a FRETish requirement if and only if the trace is a model of the MTL formula generated by FRET for that requirement. This formalization[1] was carried out in the Prototype Verification System (PVS) [26]. The correctness proof has been designed to be compositional on the FRETish requirement fields. This is crucial to building a compact and modular set of definitions and theorems that can be easily extended over time as the FRETish language evolves with new constructs and features. Besides providing a robust proof framework for the FRETish language, this research effort gave useful insights on how to simplify the MTL generation algorithms and helped improve the simulation and explanation capabilities of FRET. To the best of the authors' knowledge, this is the first formalization of a structured natural language in a theorem prover.

The paper is organized as follows. In Section 2, MTL and other preliminary notions are presented. The FRETish language is presented in Section 3. Section 4 introduces a new denotational semantics for FRETish. In Section 5, the algorithm used by FRET to generate an MTL formula from a FRETish requirement is presented. Section 6 shows the main results on the correctness of this algorithm. Section 7 illustrates the advantages of the proposed formalization. Related work is discussed in Section 8. Section 9 concludes the paper.

2 Metric Temporal Logic

Metric Temporal Logic (MTL) [19] is an extension of Linear Temporal Logic (LTL) in which the temporal operators are augmented with timing constraints. In this paper, the past-time fragment of MTL is considered. This choice is guided by the ultimate goal of the authors of using this formalization to ensure the correctness of runtime monitors automatically generated for autonomous systems, as described in [9]. In fact, past-time temporal logic is usually preferred to its future-time counterpart in the formalization of runtime monitors.

Past-time formulas look at the portion of the execution that has occurred up to the state where they are interpreted. A past-time formula is satisfied by an execution trace if the formula holds at the final state of the trace.

Let $\mathbb{I}$ denote the set of intervals of natural numbers of the form $[l, u]$ such that $l, u \in \mathbb{N}$, $l \leq u$, and for all $x \in \mathbb{N}$, $x \in [l, u] \iff l \leq x \leq u$. Given a set Π of atomic formulas, the set of past-time MTL formulas is generated by the following grammar.

$$\phi ::= true \mid false \mid p \mid \neg\phi \mid \phi \wedge \phi \mid \phi \vee \phi \mid \phi \rightarrow \phi$$
$$\mid \mathcal{Y}\phi \mid O_I \phi \mid \mathcal{H}_I \phi \mid \phi \mathcal{S}_I \phi$$

where $p \in \Pi$ and $I \in \mathbb{I}$. When I is omitted it is considered to be $[0, +\infty)$. The Boolean constants *true* and *false*, and the Boolean connectives $\wedge$, $\vee$, $\rightarrow$, and $\neg$ have the usual logic

meaning. The past-time temporal operators allowed are previous ($\mathcal{Y}$), once (O), historically ($\mathcal{H}$), and since ($\mathcal{S}$). The set of formulas generated with this grammar is denoted with $\mathbb{MTL}$.

Let $\vdash$ be the entailment relation between two atomic formulas with respect to a given theory. For instance, if the theory of real numbers arithmetics is considered, it holds that $x - 1 > 0 \vdash x > 1$. The semantics of a past-time MTL formula is given in terms of a satisfaction relation $\models$. Let $\mathbb{T}$ be the domain of finite traces of the form $\rho_0 \ldots \rho_n$ where for each $i \in [0, n]$, $\rho_i \subseteq \Pi$. Given $\rho \in \mathbb{T}$ and $t \in \mathbb{N}$, $\rho \models_t \phi$ denotes that ϕ holds in ρ at time t.

$$\rho \models_t p \iff \rho_t \vdash p$$
$$\rho \models_t \neg\phi \iff \rho \not\models_t \phi$$
$$\rho \models_t \phi_1 \wedge \phi_2 \iff \rho \models_t \phi_1, \text{ and } \rho \models_t \phi_2$$
$$\rho \models_t \phi_1 \vee \phi_2 \iff \rho \models_t \phi_1 \text{ or } \rho \models_t \phi_2$$
$$\rho \models_t \phi_1 \rightarrow \phi_2 \iff \rho \not\models_t \phi_1 \text{ or } \rho \models_t \phi_2$$
$$\rho \models_t \mathcal{Y}\phi \iff \rho \models_{t-1} \phi \text{ and } t \neq 0$$
$$\rho \models_t O_I \phi \iff \exists t_0 \in \mathbb{N} : t_0 \leq t, \; t - t_0 \in I \text{ and } \rho \models_{t_0} \phi$$
$$\rho \models_t \mathcal{H}_I \phi \iff \forall t_0 \in \mathbb{N} : t_0 \leq t \wedge t - t_0 \in I \text{ implies } \rho \models_{t_0} \phi$$
$$\rho \models_t \phi_1 \mathcal{S}_I \phi_2 \iff \exists t_0 \in \mathbb{N} : t_0 \leq t, \; t - t_0 \in I, \; \rho \models_{t_0} \phi_2$$
$$\text{and } \forall t_1 \in \mathbb{N} \text{ s.t. } t_0 < t_1 \leq |\rho| : \rho \models_{t_1} \phi_1$$

Additionally, $\rho \models \phi$ if ϕ holds at the end of the trace ρ, i.e., $\rho \models_n \phi$.

Given $m \in \mathbb{N}^{>0}$, the operator $\mathcal{Y}^m$ checks if a formula is satisfied exactly m steps before the current time. It is defined as $\mathcal{Y}^m\phi = O_{[m,m]} \phi$. The operators since-inclusive-required ($\mathcal{S}_{inc}^{req}$) and since-inclusive-optional ($\mathcal{S}_{inc}^{opt}$), introduced in [2], are used as syntactic sugar as follows:

$$\phi_1 \mathcal{S}_{inc}^{req} \phi_2 = \phi_1 \mathcal{S}(\phi_1 \wedge \phi_2)$$
$$\phi_1 \mathcal{S}_{inc}^{opt} \phi_2 = O \phi_2 \rightarrow \phi_1 \mathcal{S}(\phi_1 \wedge \phi_2).$$

Intuitively, $\phi_1 \mathcal{S}_{inc}^{req} \phi_2$ means that ϕ_1 needs to hold since and including the point in which ϕ_2 occurs and moreover ϕ_2 must eventually occur, while in the optional case $\phi_1 \mathcal{S}_{inc}^{opt} \phi_2$ the formula is only evaluated if ϕ_2 once holds.

3 The Fretish Specification Language

This section provides an introduction to the FRETish language. More details can be found in [13–15].

FRETish is a restricted natural language for expressing unambiguous requirements. It aims at providing a natural vocabulary to the user and, at the same time, avoiding ambiguities to ease the translation of the specification into a rigorous mathematical language.

EXAMPLE 3.1 Consider the following detect-and-avoid requirement written in natural language: "While in flight, a warning alert must be raised within 3 seconds of entering

---

[1]The PVS formalization is available at https://lauratitolo.github.io/

the warning hazard zone of an intruder (250 feet horizontal and 50 feet vertical)".

This requirement can be rephrased in FRETish as follows: "In *flight* mode, when *horizontal_distance* ≤ 250 & *vertical_distance* ≤ 50 the aircraft shall *within* 3 seconds satisfy *warning_alert*".   ∎

The FRET [13] tool[2] parses a FRETish requirement, maps it into a template, and translates it into an MTL formula ready to be input to a verification/analysis tool. Figure 1, depicts the FRET's requirements elicitation interface for the requirement of Example 3.1. Once a requirement is entered, the "Semantics" pane shows a text description of the FRETISH requirement, displays a "semantic diagram" showing a visual explanation of the requirement applicability over time, and provides translations from FRETISH to future- and past-time Metric Linear Temporal Logic.

A FRETish requirement is parsed into five different fields: *scope*, *condition*, *component*, *timing*, and *response*, three of which are optional: *scope*, *condition*, and *timing*. In addition, the *shall* keyword must appear and states that the component behavior must conform to the requirement.

The *component* field specifies the component that the requirement applies to (e.g., *"aircraft"* in Example 3.1). The *response* field is of the form *satisfy* ϕ, where ϕ is a non-temporal Boolean-valued expression (e.g., *warning_alert*).

Field *scope* specifies the interval(s) within which the requirement must hold (e.g., when in *flight* mode in Example 3.1). If the scope is omitted, the requirement is enforced on the entire execution, known as *global* scope. Given a mode *mode*, a FRETish *scope* is one of the following relationships: *before*(*mode*), *after*(*mode*), *in*(*mode*), *onlyAfter*(*mode*), *notIn*(*mode*), *onlyBefore*(*mode*), and *onlyIn*(*mode*).

Scope *before*(*mode*) indicates that the requirement is enforced strictly before the first point in which *mode* holds, *after*(*mode*) means it is enforced strictly after the last point in which *mode* holds, and *in*(*mode*) that it is enforced while the component is in mode *mode*. Scope *notIn*(*mode*) is the dual of *in*(*mode*). It is sometimes necessary to specify that a requirement can be satisfied *only* in some time frame, meaning it should *not* be satisfied outside of that frame. For this, the scopes *onlyAfter*, *onlyBefore*, and *onlyIn* are provided; these will be referred to as *only* scopes.

Field *condition* is a Boolean expression that triggers the need for a response within the specified scope. Boolean expressions, familiar to most developers, are used to concisely capture conditions. The set of (non-temporal) Boolean expressions is denoted by $\mathbb{B}$. If the *condition* field is omitted, it is said to be *null* and it is equivalent to *true*. Therefore, the trigger is the beginning of the trace. For instance, the condition in Example 3.1 is *horizontal_distance* ≤ 250 & *vertical_distance* ≤ 50. Figure 2 illustrates how the different scopes are defined for a given mode.

Field *timing* specifies when the response is expected relative to each trigger (e.g., *within* 3 seconds in Example 3.1). Given a duration $d \in \mathbb{N}$, and a stop condition *stop* $\in \mathbb{B}$, there are nine possibilities for the timing field: *immediately*, in the *next* time unit, *eventually*, *always*, *never*, *within d* time units, *for d* time units, *until* the specified stop condition *stop* occurs, *before* the specified stop condition *stop* occurs, and *after d* time units (interpreted as not for *d* time units and at the $d + 1$ time unit). When timing is omitted, it is assumed to be *eventually*.

A FRETish requirement is characterized by the tuple of its fields *scope*, *condition*, *timing*, and *response*. The field *component* is not relevant to the semantics of the requirement, therefore it is omitted in the definition of requirement.

DEFINITION 3.2 (FRETISH REQUIREMENT) *Given $d \in \mathbb{N}$ and mode, stop, res $\in \mathbb{B}$, a FRETish requirement is a tuple* $\langle scope, timing, cond, res \rangle$ *where*

$$scope \in \{null, in(mode), notIn(mode), before(mode),$$
$$after(mode), onlyAfter(mode),$$
$$onlyBefore(mode), onlyIn(mode)\}$$
$$condition \in \mathbb{B} \cup \{null\}$$
$$timing \in \{immediately, next, never, eventually, always,$$
$$within(d), for(d), after(d),$$
$$until(stop), before(stop)\}.$$

4 A Denotational Semantics for Fretish

In [15], the semantics of FRETish is described in terms of a discrete fragment of the Real-Time Graphical Interval Logic (RTGIL) [24]. RTGIL is a graphical language that interprets linear-time temporal formulas over ordered lists of intervals. The RTGIL toolset includes a graphical editor and a formula satisfiability checker implemented using tableaux. However, this checker is no longer maintained.

The RTGIL semantics provides an intuitive graphical representation which is helpful to visualize the meaning of the requirement. In order to obtain a compositional theoretical framework, this section introduces a denotational semantics for FRETish that mimics the RTGIL semantics of [15]. As part of the work presented in this paper, this semantics has been formalized in the PVS specification language, which offers the necessary expressive power and the possibility to rigorously reason about the properties of the semantics via the PVS theorem prover.

The proposed semantics is based on the notion of *ordered list of intervals*.

DEFINITION 4.1 *An ordered list of intervals (OLI) l is a finite list of closed intervals of natural numbers in $\mathbb{I}$ of the form $l = \langle [a_0, b_0], \ldots, [a_k, b_k] \rangle$ where $k \in \mathbb{N}$ such that for all $0 \le i \le k$, $a_i \le b_i, b_i < a_{i+1} - 1, a_0 \ge 0$. The empty OLI is denoted as ϵ.* The i-th interval $[a_i, b_i]$ in l is denoted by l_i. By abuse of notation, given $x \in \mathbb{N}$, $x \in l$ denotes that there exists i such

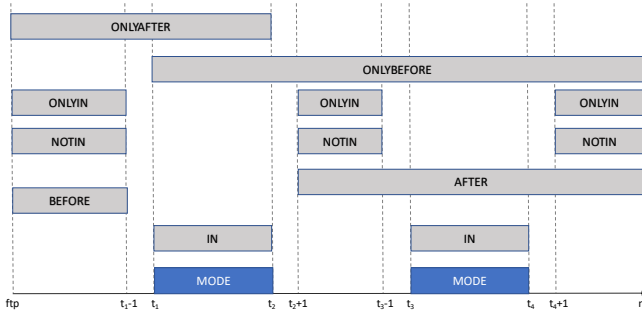**Figure 1.** FRET user interface for the requirement of Example 3.1.

Figure 2. Scope intervals definition.

Figure 3. An example trace with the four abstract properties of FRETish requirements.

that $x \in l_i$. The size of l is denoted as $|l|$. The lower bound l_0 of l is denoted with $lb(l)$, while the upper bound b_k is denoted with $ub(l)$. The same notation is used for denoting the lower and upper bound of a single interval, i.e., $lb([a_i, b_i]) = a_i$ and $ub([a_i, b_i]) = b_i$.

An *OLI* l is said to be bounded by $n \in \mathbb{N}$ if $ub(l) \leq n$. The set of all *OLI* bounded by n is denoted by $\dot{\mathbb{I}}_n$, or $\mathbb{I}$ when n is clear from the context or irrelevant.

An *OLI* encodes when a certain property holds along a trace of states indexed by natural numbers. For FRETish requirements, these properties consist of Boolean expressions
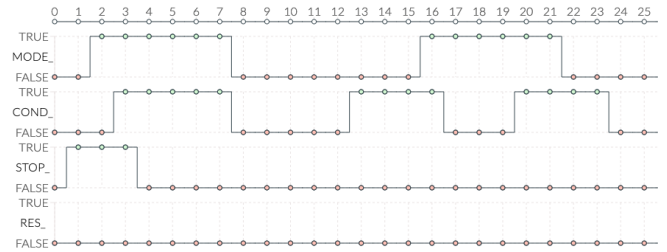
mode, *stop*, *cond*, and *res*. Notice that in FRETish time is discrete. For this reason, the domain of *OLI*s and the traces are indexed by natural numbers, not reals. While continuous time may be a more accurate representation in some situations, it introduces difficulties in operators like "next" and would require a notion of computational step to be included in execution traces.

Given a Boolean expression ψ and a trace ρ, the function $\mathcal{I}$ computes the *OLI* that encodes when ψ holds in ρ. In the example trace ρ of Figure 3, $\mathcal{I}(mode, \rho) = \langle [2, 7], [16, 21] \rangle$, and $\mathcal{I}(res, \rho) = \epsilon$. The function $\mathcal{I}$ is defined formally below.

Definition 4.2 *Given* $\psi \in \mathbb{B}$ *and* $\rho \in \mathbb{T}$, $\mathcal{I}(\psi, \rho) \in \dot{\mathbb{I}}$ *is such that* $i \in \mathcal{I}(\psi, \rho) \iff \rho \models_i \psi$.

The notion of complement models when a Boolean expression does not hold, i.e., its negation holds.

Definition 4.3 *Let* $l \in \dot{\mathbb{I}}$, *the* complement $\bar{l} \in \dot{\mathbb{I}}$ *of* l *is such that, for every* $x \in \mathbb{N}$, $x \leq n$, $x \notin l \iff x \in \bar{l}$.

The following lemma follows directly from Definition 4.2 and Definition 4.3.

Lemma 4.4 *Given* $\psi \in \mathbb{B}$, $\rho \in \mathbb{T}$, *and* $i \in \mathbb{N}$:

$$i \in \mathcal{I}(\psi, \rho) \iff i \in \overline{\mathcal{I}(\neg\psi, \rho)}.$$

In the following, the FRETish semantics is described compositionally, through the values of fields *scope*, *condition*, and *timing*. The *scope semantics* defines the list of intervals within which the temporal requirement specification must hold.

Definition 4.5 (Scope Semantics) *Let* $\rho \in \mathbb{T}$ *such that* $|\rho| > 0$, $n = |\rho| - 1$ *and* $modeInt = \mathcal{I}(mode, \rho)$. *The semantics of a scope* s *is defined as follows.*

$\mathcal{S}[\![null]\!]_\rho = \langle [0, n] \rangle$

$\mathcal{S}[\![in(mode)]\!]_\rho = modeInt$

$\mathcal{S}[\![notIn(mode)]\!]_\rho = \overline{modeInt}$

$\mathcal{S}[\![after(mode)]\!]_\rho = \begin{cases} \epsilon & \text{if } modeInt = \epsilon \\ & \text{or } ub(modeInt_0) = n \\ \langle [ub(modeInt_0) + 1, n] \rangle & \text{otherwise} \end{cases}$

$\mathcal{S}[\![before(mode)]\!]_\rho = \begin{cases} \epsilon & lb(modeInt_0) = 0 \\ \langle [0, lb(modeInt_0) - 1] \rangle & \text{otherwise} \end{cases}$

$\mathcal{S}[\![onlyIn(mode)]\!]_\rho = \overline{modeInt}$

$\mathcal{S}[\![onlyAfter(mode)]\!]_\rho = \begin{cases} \langle [0, n] \rangle & \text{if } modeInt = \epsilon \\ \langle [0, ub(modeInt_0)] \rangle & \text{otherwise} \end{cases}$

$\mathcal{S}[\![onlyBefore(mode)]\!]_\rho = \begin{cases} \epsilon & \text{if } modeInt = \epsilon \\ \langle [lb(modeInt_0), n] \rangle & \text{otherwise} \end{cases}$

As already mentioned, note that scopes of type *only*, mandate that a requirement does *not* hold outside of their corresponding scope. Thus, the semantics of an only scope is defined as the complement of the semantics of its corresponding regular scope. Within an interval I, a requirement is triggered at each index of I, where the conditional expression *cond* becomes true from false in ρ, and at $lb(I)$, if $\rho \models_{lb(I)} cond$. When no condition is specified (*null*), the requirement is only triggered at $lb(I)$. For a trace ρ, an interval I, and a condition *cond*, the function $triggers(cond, \rho, I)$ returns the set of all such indices.

Definition 4.6 (Triggers) *Let* $cond \in \mathbb{B} \cup \{null\}$, $I \in \mathbb{I}$, *and* $\rho \in \mathbb{T}$, $triggers(cond, \rho, I) \subseteq \mathbb{N}$ *is defined as follows.*

$triggers(cond, \rho, I) = \{lb(I) \mid cond = null\} \cup$

$$\bigcup_{0 \leq j < |condInt|} \{max(lb(condInt_j), lb(I)) \mid cond \neq null,$$
$$condInt_j \cap I \neq \emptyset\}$$

where $condInt = \mathcal{I}(cond, \rho)$.

When, within the range of interval I, a condition is always false in ρ, the function *triggers* will return an empty set; the requirement is never triggered and is therefore vacuously true. Note that this is different from the case where the condition field in the FRETish requirement is *null* and, thus, the trigger occurs at $lb(I)$.

Stop conditions, necessary for the timings *until* and *before*, are computed similarly to triggers. Given a stop condition *stop*, the function $stops(stop, \rho, I)$ computes the set of indices where *stop* is satisfied, within a trace ρ, restricted to a particular scope interval I. The function $firststop(stop, t, \rho, I)$ returns the index of the first occurrence of *stop* after a trigger t, or $ub(I)$, if *stop* is never true within interval I.

Definition 4.7 (Stops) *Given* $stop \in \mathbb{B}$, $t \in \mathbb{N}$, $\rho \in \mathbb{T}$, *and* $I \in \mathbb{I}$, *the functions stops and firststop are defined as follows.*

$stops(stop, \rho, I) =$

$$\bigcup_{0 \leq j < |stopInt|} \{max(lb(stopInt_j), lb(I)) \mid stopInt_j \cap I \neq \emptyset\}$$

$firststop(stop, t, \rho, I) = \begin{cases} ub(I) + 1 & \text{if } \nexists s \in stops: \ t < s \\ min(\{s \in stops \mid t < s\}) & \text{otherwise} \end{cases}$

where $stopInt = \mathcal{I}(stop, \rho)$ and $stops = stops(stop, \rho, I)$.

Given a trace ρ, a condition *cond* and a response *res*, the semantics of a timing field is the set of intervals of indices I in ρ such that, for each trigger defined by *cond*, the response *res* is satisfied per the particular timing field in I.

Definition 4.8 (Timing Semantics) *Let* $n \in \mathbb{N}^{>0}$, $\rho \in \mathbb{T}$, *and* $resInt = \mathcal{I}(res, \rho)$, *the semantics of a timing field is defined as in Figure 4.*

The semantics of timing *immediately* is composed of all the intervals I such that the triggers occurring in I are also in the response. Similarly, the semantics of timing *next* contains all the intervals I such that the time index following a trigger is in the response.

For timing *always*, the semantics contains all the intervals I such that all the indices ranging from the first trigger that occurs until the end of the interval are also contained in the response. The semantics of timing *never* is defined as the semantics of timing *always* with the negated response. From Lemma 4.4, this is equivalent to checking that all the indices from the first trigger until the end of the interval are included in the complement of the response.

The semantics of the *eventually* timing includes all the intervals I such that there exists an index which is both a trigger and it is included in the interval from the last trigger to the end of the interval. It is worth noting that, for each trigger t, the response is required to hold at least once in the interval from t until the end of the interval. Therefore, it is

sufficient to check that it holds from the last trigger until the interval upper bound. Similarly, for the *always* timing it is sufficient to check every index from the first trigger until the interval upper bound.

The semantics of the *within(d)* timing contains all the intervals I such that, for all the triggers t, either there are less than d time instants between t and the end of the interval ($t + d \notin I$), or there exists a k such that $t + k$ is in the interval I and the response holds at $t + k$.

In the case of the *for(d)* timing, the semantics includes all the intervals I such that for all the natural numbers k between 0 and d, either $t + k$ is not included in I or $t + k$ is in the response.

The *until(stop)* semantics collects all the intervals such that, for each trigger t, the subinterval ranging from t to the first instant satisfying the stop condition *stop* is included in the response.

The semantics of timing *before(stop)* includes all the intervals that do not contain at least one stop condition and the intervals such that a stop condition never occurs after a trigger (*firststop(stop, t, ρ, I) = ub(I) + 1*) . In these cases the requirement trivially holds. In addition, it collects the intervals such that at least a trigger t exists, *stop* holds between t and the end of the interval, and there exists a time instant between the trigger and the stop condition that satisfies the response.

Finally, timing *after* semantics is derived from the semantics of *for* and *within*. In fact, *after d* time units is equivalent to not *for d* time units and *within d + 1* time units. In addition, for all the timings, the semantics includes all the intervals that do not contain any trigger. This reflects the fact that when no trigger occurs, the requirement holds trivially.

The semantics of a FRETish requirement is defined compositionally from the semantics of its fields. The function *dual* returns the dual of a timing field and it is defined in Table 1.

Definition 4.9 (FRETish semantics) *Let $r = \langle scope,$ timing, cond, res\rangle$ be a requirement. The semantics of r is defined as follows.*

$$\mathcal{F}[\![r]\!] = \{\rho \in \mathbb{T} \mid \mathcal{S}[\![scope]\!]_\rho \neq \epsilon \text{ and}$$
$$\forall i \text{ such that } 0 \leq i < |\mathcal{S}[\![scope]\!]_\rho| :$$
$$(\mathcal{S}[\![scope]\!]_\rho)_i \in \mathcal{T}[\![timing']\!]_\rho(cond, res')\}$$

where

$$timing' = \begin{cases} dual(timing) & \text{if scope is of type only} \\ timing & \text{otherwise} \end{cases}$$

$$res' = \begin{cases} \overline{res} & \text{if scope is of type only} \\ res & \text{otherwise} \end{cases}$$

As already mentioned, for scopes of type *only*, a requirement does *not* hold outside of their corresponding scope.

Table 1. Timing duality.

timing	dual(timing)
always	eventually
eventually	always
within(d)	for(d)
for(d)	within(d)
before(stop)	until(stop)
until(stop)	before(stop)
immediately	immediately
next	next

This means that the negation of the main body of the requirement must hold outside their scope. Therefore, the dual of the timing is considered and the result is negated.

The PVS formalization of FRETish includes an executable implementation of this semantics and theorems ensuring that it is equivalent to the denotational semantics presented in this section. Additionally, to improve the confidence that the PVS implementation is equivalent to the RTGIL semantics implemented in Fret, the proposed semantics has been checked with the help of the Fret extensive testing framework. To this aim, the testing framework has been enhanced to automatically generate test cases in the language of PVS. In particular, 10,000 test cases have been generated. Each of these test cases was composed of a *FRETish* specification r, a trace ρ, and a Boolean indicating if the trace belongs to the semantics of the specification. This Boolean value was computed by a component of Fret, called *oracle*, which implements the RTGIL semantics presented in [15]. Finally, the executable PVS semantics was executed for each test case, and a Boolean value was computed indicating if the trace ρ was included in the semantics of r.

5 From FRETish Specifications to MTL Formulas

The Fret tool generates a temporal logic formula from a FRETish specification. While the tool supports both future and past time metric linear temporal logic, in this work the past time fragment is considered. The formula is built compositionally on the FRETish requirement fields. The algorithm views a trace as a collection of disjoint intervals characterized by the scope field. The temporal requirement, characterized by the timing and condition fields, is imposed on each of these intervals.

First, the formula corresponding to the timing field and condition is generated. This formula is called *core formula*. The following functions are used to model the condition field. Given a condition *cond* $\in \mathbb{B}$ and a formula $\phi_{left} \in \mathbb{B}$ that models the left endpoint of the considered interval, the formula *trigger(cond, ϕ_{left})* characterizes a trigger, i.e., a point in time when *cond* becomes true from false or when it

$$\mathcal{T}[\![\,immediately\,]\!]_\rho(cond, res) = \{I \mid triggers(cond, \rho, I) = \emptyset\}$$
$$\cup \{I \mid triggers(cond, \rho, I) \neq \emptyset \land \forall t \in triggers(cond, \rho, I) : t \in resInt\}$$

$$\mathcal{T}[\![\,next\,]\!]_\rho(cond, res) = \{I \mid triggers(cond, \rho, I) = \emptyset\}$$
$$\cup \{I \mid triggers(cond, \rho, I) \neq \emptyset \land \forall t \in triggers(cond, \rho, I) : t + 1 \in I \rightarrow t + 1 \in resInt\}$$

$$\mathcal{T}[\![\,always\,]\!]_\rho(cond, res) = \{I \mid triggers(cond, \rho, I) = \emptyset\}$$
$$\cup \{I \mid triggers(cond, \rho, I) \neq \emptyset \land \forall j \in [min(triggers(cond, \rho, I)), ub(I)] : j \in resInt\}$$

$$\mathcal{T}[\![\,never\,]\!]_\rho(cond, res) = \mathcal{T}[\![\,always\,]\!]_\rho(cond, \neg res)$$

$$\mathcal{T}[\![\,eventually\,]\!]_\rho(cond, res) = \{I \mid triggers(cond, \rho, I) = \emptyset\}$$
$$\cup \{I \mid triggers(cond, \rho, I) \neq \emptyset \land \exists j \in [max(triggers(cond, \rho, I)), ub(I)] : j \in resInt\}$$

$$\mathcal{T}[\![\,within(d)\,]\!]_\rho(cond, res) = \{I \mid triggers(cond, \rho, I) = \emptyset\}$$
$$\cup \{I \mid triggers(cond, \rho, I) \neq \emptyset \land \forall t \in triggers(cond, \rho, I) :$$
$$t + d \in I \rightarrow \exists k \leq d : t + k \in I \land t + k \in resInt\}$$

$$\mathcal{T}[\![\,for(d)\,]\!]_\rho(cond, res) = \{I \mid triggers(cond, \rho, I) = \emptyset\}$$
$$\cup \{I \mid triggers(cond, \rho, I) \neq \emptyset \land \forall t \in triggers(cond, \rho, I) \forall k \leq d : t + k \in I \rightarrow t + k \in resInt\}$$

$$\mathcal{T}[\![\,until(stop)\,]\!]_\rho(cond, res) = \{I \mid triggers(cond, \rho, I) = \emptyset\}$$
$$\cup \{I \mid triggers(cond, \rho, I) \neq \emptyset \land \forall t \in triggers(cond, \rho, I) :$$
$$\forall x \in [t, firststop(stop, t, \rho, I) - 1] : x \in resInt\}$$

$$\mathcal{T}[\![\,before(stop)\,]\!]_\rho(cond, res) = \{I \mid triggers(cond, \rho, I) = \emptyset \lor stops(stop, \rho, I) = \emptyset\}$$
$$\cup \{I \mid triggers(cond, \rho, I) \neq \emptyset \land stops(stop, \rho, I) \neq \emptyset \land \forall t \in triggers(cond, \rho, I) :$$
$$firststop(stop, t, \rho, I) = ub(I) + 1 \lor \exists x : t \leq x < firststop(stop, t, \rho, I) \land x \in resInt\}$$

$$\mathcal{T}[\![\,after(n)\,]\!]_\rho(cond, res) = \mathcal{T}[\![\,for(n)\,]\!]_\rho(cond, \neg res) \cap \mathcal{T}[\![\,within(n + 1)\,]\!]_\rho(cond, res)$$

Figure 4. Timing field semantics.

holds at the beginning of the interval.

$$trigger(cond, \phi_{left}) = (cond \land \mathcal{Y}(\neg cond)) \lor (cond \land \phi_{left}) \tag{5.1}$$

The formula *noTriggers* captures the fact that the condition never occurs in the interval. In this case, the requirement is trivially true.

$$noTriggers(cond, \phi_{left}) = \neg cond \, \mathcal{S}_{inc}^{req} \, \phi_{left} \tag{5.2}$$

Given a timing field *timing*, a condition $cond \in \mathbb{B}$, a response $res \in \mathbb{B}$, and a left endpoint ϕ_{left}, Φ_{core} computes the core formula as illustrated in Table 2. Here, $\phi_{trigger}$ denotes $trigger(cond, \phi_{left})$, while $\phi_{noTriggers}$ denotes the formula $noTriggers(cond, \phi_{left})$.

The core formula is then interpreted in a *generic* finite interval defined by two endpoints: ϕ_{left}, and ϕ_{right}. The core formula is checked at each point in the target interval, except for the timing *eventually*, which is checked just once at the

right endpoint. The resulting formula is called *base formula*.

$$\Phi_{base}(timing, cond, res, \phi_{left}, \phi_{right})$$
$$= \begin{cases} \phi_{right} \rightarrow \mathcal{Y} \phi_{core} & \text{if } timing = eventually \\ \phi_{right} \rightarrow \mathcal{Y} (\phi_{core} \, \mathcal{S}_{inc}^{opt} \, \phi_{left}) & \text{otherwise} \end{cases}$$
$$\tag{5.3}$$

where $\phi_{core} = \Phi_{core}(timing, cond, res, \phi_{left})$. There is a special case of base formula that is defined when the interval spans to the end of the trace. In this case, the right endpoint is not defined and the core formula is required to hold from the left endpoint to the end of the trace.

$$\Phi_{baseLast}(timing, cond, res, \phi_{left})$$
$$= \begin{cases} (O \, \phi_{left}) \rightarrow \phi_{core} & \text{if } timing = eventually \\ \phi_{core} \, \mathcal{S}_{inc}^{opt} \, \phi_{left} & \text{otherwise} \end{cases} \tag{5.4}$$

Finally, the *general formula* imposes the base formula on the whole execution trace and the generic points ϕ_{left} and ϕ_{right} are replaced by formulas characterizing the given scope field. Given $mode \in \mathbb{B}$, Table 3 shows a list of abbreviations

Table 2. Core formula definition for *null* condition or $cond \in \mathbb{B}$.

timing	$\Phi_{core}(timing, null, res, \phi_{left})$	$\Phi_{core}(timing, cond, res, \phi_{left})$
immediately	$\phi_{left} \rightarrow res$	$\phi_{trigger} \rightarrow res$
next	$(\mathcal{Y}\,\phi_{left}) \rightarrow res$	$(\mathcal{Y}\,\phi_{trigger}) \rightarrow (res \vee \phi_{left})$
always	res	$\phi_{noTriggers} \vee (res\,\mathcal{S}_{inc}^{req}\,\phi_{trigger})$
eventually	$\neg(\neg res\,\mathcal{S}_{inc}^{req}\,\phi_{left})$	$\phi_{noTriggers} \vee \neg(\neg res\,\mathcal{S}_{inc}^{req}\,\phi_{trigger})$
until(stop)	$(\neg stop\,\mathcal{S}_{inc}^{req}\,\phi_{left}) \rightarrow res$	$\phi_{noTriggers} \vee ((\neg stop\,\mathcal{S}_{inc}^{req}\,\phi_{trigger}) \rightarrow res)$
before(stop)	$stop \rightarrow ((\neg\phi_{left}$ $\wedge \neg\,\mathcal{Y}\,(\neg res\,\mathcal{S}_{inc}^{req}\,\phi_{left})))$	$stop \rightarrow (\phi_{noTriggers} \vee (\neg\phi_{left} \wedge \neg\phi_{trigger}$ $\wedge \neg\,\mathcal{Y}\,(\neg res\,\mathcal{S}_{inc}^{req}\,\phi_{trigger})))$
for(m)	$(O_{[0,m]}\,\phi_{left}) \rightarrow res$	$(O_{[0,m]}\,\phi_{trigger}) \rightarrow (\phi_{noTriggers} \vee res)$
within(m)	$(\neg res\,\mathcal{S}_{inc}^{req}\,\phi_{left}) \rightarrow (O_{[0,m-1]}\,\phi_{left})$	$\mathcal{Y}^m(\phi_{trigger} \wedge \neg res)$ $\rightarrow (O_{[0,m-1]}(\phi_{left} \vee res))$

Table 3. Formulas for point of interest.

point	formula
ftp	$\neg\,\mathcal{Y}\,true$
fim(mode)	$mode \wedge (ftp \vee (\mathcal{Y}(\neg mode)))$
lim(mode)	$\neg mode \wedge \mathcal{Y}\,mode$
fnim(mode)	$\neg mode \wedge (ftp \vee (\mathcal{Y}\,mode))$
lnim(mode)	$mode \wedge \mathcal{Y}\,(\neg mode)$
ffim(mode)	$fim(mode) \wedge (ftp \vee \mathcal{Y}\,(\mathcal{H}\,\neg mode))$
flim(mode)	$lim(mode) \wedge \mathcal{Y}\,(\mathcal{H}\,\neg\,lim(mode))$

Table 4. Scope endpoints definition.

scope	*left(scope)*	*right(scope)*
null	*ftp*	-
before(mode)	*ftp*	*ffim(mode)*
after(mode)	*flim(mode)*	-
in(mode)	*fim(mode)*	*lim(mode)*
notIn/onlyIn(mode)	*fnim(mode)*	*lnim(mode)*
onlyBefore(mode)	*ffim(mode)*	-
onlyAfter(mode)	*ftp*	*flim(mode)*

used to model different points of interest in the execution trace. The abbreviation *ftp* denotes the first time point in execution, *fim(mode)* (respectively *lim(mode)*) is the first (last) state in which *mode* holds, *fnim(mode)* (*lnim(mode)*) is the first (last) state in which *mode* does not hold, and *ffim(mode)* (*flim(mode)*) is the first occurrence of *fim(mode)* (*lim(mode)*) in the execution.

Table 4 shows which points of interest in the trace, *left* and *right* correspond to, depending on the *scope* field. It is important to notice that for *null*, *after*, and *onlyBefore* scopes the right endpoint is not defined. In fact, in these cases, the interval spans until the end of the trace so there is no need to impose a right endpoint. The general formula is defined

by composing the base formula, depending on the timing and condition fields, and the information about the left and right endpoints, depending on the scope field. The function Φ computes the formula corresponding to a given FRETish requirement $\langle scope, timing, cond, res \rangle$.

$\Phi(\langle scope, timing, cond, res \rangle)$

$$= \begin{cases} \phi_{baseLast} & \text{if } scope \in \{null, after, before\} \\ \mathcal{H}(\phi_{base} \vee ftp)\wedge \\ ((\neg\phi_{right}\,\mathcal{S}_{inc}^{req}\,\phi_{left}) \rightarrow \phi_{baseLast}) & \text{otherwise} \end{cases}$$

$$(5.5)$$

where

$$\phi_{left} = left(scope)$$
$$\phi_{right} = right(scope)$$
$$\phi_{baseLast} = \Phi_{baseLast}(timing, cond, res, \phi_{left})$$
$$\phi_{base} = \Phi_{base}(timing, cond, res, \phi_{left}, \phi_{right}).$$

EXAMPLE 5.1 Consider the FRETish requirement of Example 3.1: "In *flight* mode, when *horizontal_distance* ≤ 250 & *vertical_distance* ≤ 50 the aircraft shall *within* 3 seconds satisfy *warning_alert*". The following steps computes the corresponding past-time MTL formula.

First of all, it is necessary to determine the endpoints of each interval in which the requirement is imposed. Since the scope is *in(flight_mode)*, the MTL formula characterizing the interval left endpoint is:

$fim(flight\_mode) = flight\_mode \wedge (ftp \vee (\mathcal{Y}(\neg flight\_mode)));$

while the formula characterizing the right endpoint is:

$lim(flight\_mode) = \neg flight\_mode \wedge \mathcal{Y}\,flight\_mode.$

The condition indicated in the requirement is:

$horizontal\_distance \leq 250 \wedge vertical\_distance \leq 50.$

Thus, a trigger of this condition is modeled by the formula

$\phi_{trigger} = (horizontal\_distance \leq 250 \wedge vertical\_distance \leq 50$

$$\land \, \mathcal{Y} \, (\neg(horizontal\_distance \leq 250)$$
$$\lor \neg(vertical\_distance \leq 50)))$$
$$\lor \, (horizontal\_distance \leq 250 \land vertical\_distance \leq 50$$
$$\land \, fim(flight\_mode)).$$

At this point, it is possible to compute the core formula for the timing $within(3)$:

$$\phi_{core} = \mathcal{Y}^3(\phi_{trigger} \land \neg warning\_alert)$$
$$\rightarrow (O_{[0,2]}(fim(flight\_mode) \lor warning\_alert)).$$

The regular base formula and the base formula for the last interval case are computed as follows:

$$\phi_{base} = lim(flight\_mode) \rightarrow \mathcal{Y} \, (\phi_{core} \, \mathcal{S}_{inc}^{opt} \, fim(flight\_mode))$$
$$\phi_{baseLast} = \phi_{core} \, \mathcal{S}_{inc}^{opt} \, fim(flight\_mode).$$

Finally, the general formula that imposes the requirement on the entire trace is:

$$\mathcal{H}(\phi_{base} \lor ftp) \land$$
$$((\neg \, lim(flight\_mode) \, \mathcal{S}_{inc}^{req} \, fim(flight\_mode)) \rightarrow \phi_{baseLast}).$$
∎

6 Proving the Semantic Equivalence

This section presents the main theorems ensuring the correctness of the FRET past-time MTL formula generation algorithm. As already mentioned, the entire formalization and the proofs have been mechanically checked in the PVS theorem prover.

FRETish currently supports 8 relationships for the *scope* field (including global scope), 2 options for the condition, and 10 options for the timing field, for a total of 160 combinations of semantic templates. Proving the equivalence of each one of these templates to the corresponding general formula would be extremely time-consuming. Therefore, to provide a reusable and extensible formalization, the proof has been structured in a compositional manner.

Following the same structure of the formula generation, the first step is to show that the base formula, which depends on the timing field, is equivalent to the semantics of the timing field in the context of an interval I. To this aim, it is convenient to notice that the definition of base formula in Equations (5.3) and (5.4), which use the *since inclusive optional* construct, are equivalent to imposing the core formula on the interval of interest with the historically operator.

Lemma 6.1 *Let $\rho \in \mathbb{T}$, $n \in \mathbb{N}$ such that $n = |\rho| - 1$, and let $I \in \mathbb{I}$ and $\phi, \phi_{left}, \phi_{right} \in \mathbb{MTL}$ such that $\rho \models_{lb(I)} \phi_{left}$, $\rho \models_{ub(I)+1} \phi_{right}$, and for all $t > lb(I)$, $\rho \not\models_t \phi_{left}$, then if $ub(I) \neq n$ then*

$$\rho \models_{ub(I)+1} \phi_{right} \rightarrow \mathcal{Y} \, (\phi \, \mathcal{S}_{inc}^{opt} \, \phi_{left})$$
$$\Longleftrightarrow \rho \models_n \mathcal{H}_{[n-ub(I),n-lb(I)]} \, \phi,$$

otherwise

$$\rho \models_{ub(I)} \phi \, \mathcal{S}_{inc}^{opt} \, \phi_{left} \Longleftrightarrow \rho \models_n \mathcal{H}_{[n-ub(I),n-lb(I)]} \, \phi.$$

It is worth noting that, since past-time temporal formulas are evaluated starting from the end of the trace at time n, when the operator historically ($\mathcal{H}$) is used, the formula is imposed on the interval $[n - ub(I), n - lb(I)]$. This is equivalent to using the future time always (or globally) operator on the interval I.

Corollary 6.2 *Given a timing field timing, a condition field $cond \in \mathbb{B}$, a response field $res \in \mathbb{B}$, a trace $\rho \in \mathbb{T}$, $n \in \mathbb{N}$ such that $n = |\rho| - 1$, an interval $I \in \mathbb{I}$, and $\phi, \phi_{left}, \phi_{right} \in \mathbb{MTL}$ such that $\rho \models_{lb(I)} \phi_{left}$, $\rho \models_{ub(I)+1} \phi_{right}$, and for all $t > lb(I)$, $\rho \not\models_t \phi_{left}$, if $ub(I) \neq n$ then*

$$\rho \models_{ub(I)+1} \Phi_{base}(timing, cond, res, \phi_{left}, \phi_{right})$$
$$\Longleftrightarrow \rho \models_n \mathcal{H}_{[n-ub(I),n-lb(I)]} \, \Phi_{core}(timing, cond, res, \phi_{left}),$$

otherwise

$$\rho \models_{ub(I)} \Phi_{baseLast}(timing, cond, res, \phi_{left})$$
$$\Longleftrightarrow \rho \models_n \mathcal{H}_{[n-ub(I),n-lb(I)]} \, \Phi_{core}(timing, cond, res, \phi_{left}).$$

The following lemma states that if the *trigger* formula defined in Equation (5.1) is satisfied at an index t of a trace ρ, then t belongs to the set of triggers for ρ, and vice-versa. Additionally, if there is no trigger occurring before or at index t, then the formula *noTriggers* is satisfied between the beginning of the scope interval of interest and t, and vice-versa.

Lemma 6.3 *Given $cond \in \mathbb{B}$, $\rho \in \mathbb{T}$, $I \in \mathbb{I}$, and $\phi_{left} \in \mathbb{MTL}$ such that $\rho \models_{lb(I)} \phi_{left}$ and for all $t > lb(I)$, $\rho \not\models_t \phi_{left}$, for all $t < |\rho|$:*

$$t \in triggers(cond, I, \rho) \Longleftrightarrow \rho \models_t trigger(cond, \phi_{left})$$

and

$$triggers(cond, I, \rho) \cap [lb(I), t] = \emptyset$$
$$\Longleftrightarrow \forall t_0 \in [lb(I), t] : \rho \models_{t_0} noTriggers(cond, \phi_{left}).$$

Proof Sketch. The first equivalence follows from the fact that either *cond* is satisfied at t but not at $t - 1$ or *cond* holds at the beginning of the interval I. Thus, by Definition 4.2, t is a lower bound of an interval in $\mathcal{I}(cond, \rho)$ or is the interval lower bound $lb(I)$. The second equivalence follows from the fact that, by definition of $\mathcal{S}_{inc}^{req}$, $\forall t_0 \in [lb(I), t]$, $\rho \not\models_t cond$. Thus, t is not included in $\mathcal{I}(cond, \rho)$. Vice-versa, if t was a member of $\mathcal{I}(cond, \rho)$, then it would satisfy *cond*, but $\rho \models_t \neg cond$ by definition of $\mathcal{S}_{inc}^{req}$. □

Table 2 distinguishes two cases for the definition of the core formula: condition omitted (*null*) or condition specified (*cond* $\in \mathbb{B}$). This distinction is convenient since the formulas generated when the condition is *null* are more compact. However, it can be shown that the simplified formula generated for the *null* condition is equivalent to the formula generated when the condition is *true*.

Lemma 6.4 *Given a timing field timing, a scope field scope, a condition cond* $\in \mathbb{B}$, *a response res* $\in \mathbb{B}$, *then*

$$\Phi(\langle scope, timing, null, res \rangle) \iff \Phi(\langle scope, timing, true, res \rangle).$$

Proof Sketch. From Equations (5.1) and (5.2), observe that $trigger(true, \phi_{left}) = \phi_{left}$ and $noTriggers(true, \phi_{left}) = false$. From Table 2 and Equation (5.1), the result follows by applying Boolean connectives manipulation and simplifications. □

The following lemma states the correctness of the base formula. In other words, it shows that the semantics of a timing field is equivalent to the semantics of its corresponding base formula generated as shown in Section 5.

Theorem 6.5 *Given* $\rho \in \mathbb{T}$, *a timing field timing, a scope field scope, a condition cond* $\in \mathbb{B}$, *a response res* $\in \mathbb{B}$, *an interval* $I \in \mathbb{I}$ *and* $\phi_{left}, \phi_{right} \in \mathbb{MTL}$ *such that* $\rho \models_{lb(I)} \phi_{left}$, $\rho \models_{ub(I)+1} \phi_{right}$, *and for all* $t > lb(I)$, $\rho \not\models_t \phi_{left}$, *if* $ub(I) \neq n$ *then*

$I \in \mathcal{T}[\![timing]\!]_\rho(cond, res)$

$\iff \rho \models_{ub(I)+1} \Phi_{base}(cond, timing, res, \phi_{left}, \phi_{right})$

otherwise

$I \in \mathcal{T}[\![timing]\!]_\rho(cond, res)$

$\iff \rho \models_{ub(I)} \Phi_{baseLast}(cond, timing, res, \phi_{left}).$

The proof proceeds by cases on the timing field. Below, the case in which $timing = always$ is shown. The proofs for the other timings are similar and they are available as part of the FRETish PVS formalization.

Proof. Assume that $timing = always$, $cond \in \mathbb{B}$, and $ub(I) \neq n$. From Lemma 6.4, the case when $cond = null$ is equivalent to the case when $cond = true$. Consider the $\Rightarrow$ direction. Let $I \in \mathcal{T}[\![always]\!]_\rho(cond, res)$ and split the proof into two cases. Assume, first, that $triggers(cond, \rho, I) = \emptyset$, thus:

$triggers(cond, I, \rho) = \emptyset$

$\Rightarrow triggers(cond, I, \rho) \cap [lb(I), ub(I)] = \emptyset$

  [By Lemma 6.3]

$\Rightarrow \forall t_0 \in [lb(I), ub(I)] : \rho \models_{t_0} \phi_{noTriggers}$

$\Rightarrow \forall t_0 \in [lb(I), ub(I)] : \rho \models_{t_0} \phi_{noTriggers} \vee (res \, \mathcal{S}_{inc}^{req} \, \phi_{trigger})$

  [By Table 2]

$\Rightarrow \forall t_0 \in [lb(I), ub(I)] : \rho \models_{t_0} \Phi_{core}(always, cond, res, \phi_{left})$

$\Rightarrow \rho \models_n \mathcal{H}_{[lb(I), ub(I)]} \Phi_{core}(always, cond, res, \phi_{left})$

By Corollary 6.2, if $ub(I) \neq n$ then it holds that $\rho \models_{ub(I)+1} \Phi_{base}(cond, always, res, \phi_{left}, \phi_{right})$, otherwise, it holds that $\rho \models_{ub(I)} \Phi_{baseLast}(cond, always, res, \phi_{left})\phi_{right}$.

If $triggers(cond, I, \rho) \neq \emptyset$, let $t \in [lb(I), ub(I)]$. If $t < min(triggers(cond, I, \rho))$, then $triggers(cond, \rho, I) \cap [lbI, t] = \emptyset$. By Lemma 6.3, it follows that for all $t_0 \in [lb(I), t]$, $\rho \models_{t_0} \phi_{noTriggers}$. From here the proof proceeds as the previous case in which $triggers(cond, \rho, I) = \emptyset$.

Otherwise, if $t \geq min(triggers(cond, \rho, I))$, let t_0 be the first trigger $min(triggers(cond, \rho, I))$. By Lemma 6.3, it follows that $\rho \models_{t_0} \phi_{trigger}$. By Table 2, it follows that for all $j \in [t_0, ub(I)]$, $j \in resInt$, and, by Definition 4.2, this means that $\rho \models_j res$. In particular, $\rho \models_{t_0} res$, and, thus, $\rho \models_{t_0} \phi_{trigger} \wedge res$. By definition of $\mathcal{S}_{inc}^{req}$, it follows that $\rho \models_t res \, \mathcal{S}_{inc}^{req} \, \phi_{trigger} \wedge res$ for all $t \in [lb(I), ub(I)]$. From here, the result follows from Corollary 6.2 and Table 2.

Now, consider the $\Leftarrow$ direction. By Corollary 6.2, it follows that $\rho \models_n \mathcal{H}_{[lb(I), ub(I)]} \phi_{noTriggers} \vee (res \, \mathcal{S}_{inc}^{req} \, \phi_{trigger})$. Let $t \in I$ and split the proof into two cases.

If $triggers(cond, I, \rho) = \emptyset$ the theorem follows directly. Otherwise, suppose $triggers(cond, I, \rho) = \{t_0, t_1, \ldots, t_k\}$ such that $t_i < t_{i+1}$ for all $i \in [0, k-1]$. It follows that the interval $[min(triggers(cond, I, \rho)), ub(I)]$ can be represented as a union of intervals according to the triggers. That is,

$$[min(triggers(cond, I, \rho)), ub(I)] = \bigcup_{i=0}^{k-1} [t_i, t_{i+1}] \cup [t_k, ub(I)]$$

where $t_0 = min(triggers(cond, I, \rho))$.

Let $j \in [min(triggers(cond, I, \rho)), ub(I)]$ and define t to be either t_{i+1} for $i \in [0, k-1]$ or $ub(I)$. By hypothesis, $\rho \models_t \phi_{noTriggers} \vee (res \, \mathcal{S}_{inc}^{req} \, \phi_{left})$, but $\rho \not\models_t \phi_{noTriggers}$, otherwise by Lemma 6.3, $triggers(cond, I, \rho) \cap [lb(I), t] = \emptyset$, but this is a contradiction since there exists at least one trigger before t. Thus, $\rho \models_t res \, \mathcal{S}_{inc}^{req} \, \phi_{left}$, and, by definition of $\mathcal{S}_{inc}^{req}$, it follows that there exists r_0 such that $r_0 \leq t$ and $\rho \models_{r_0} res \wedge \phi_{trigger}$ and for all $r_0 \leq r_1 \leq t, \rho \models_{r_1} res$. By Lemma 6.3, $\rho \models_{t_i} \phi_{trigger}$ for all $i \in [0, k]$ and, since $j \geq t_i$, it follows that $\rho \models_j res$, thus, $j \in resInt$ and the proof is complete. □

The following theorem ensures that the semantics of a FRETish requirement is equivalent to the semantics of the past-time MTL formula generated as defined in Section 5. That is, if a trace is in the semantics of a FRETish requirement, then the generated MTL formula holds at the end of the trace, and viceversa.

Theorem 6.6 *Let r be a FRETish requirement and* $\rho \in \mathbb{T}$,

$$\rho \in \mathcal{F}[\![r]\!] \iff \rho \models \Phi(r).$$

The proof proceeds by cases on the scope field. Each proof is split into two main cases to distinguish when the scope is an only scope or not. In the first case, *timing* and *res* are replaced with their duals in the proof. Below, the case in which $scope = in(mode)$ is shown. The PVS formalization contains the lemmas for the other scope fields.

Proof. Assume that the *scope* is $in(mode)$ for a given $mode \in \mathbb{B}$. Let $\rho \in \mathcal{F}[\![\langle in(mode), timing, cond, res \rangle]\!]$ and consider the $\Rightarrow$ direction. First, it is shown that $\rho \models_n \mathcal{H}(\phi_{base} \vee ftp)$. Let $t \in [0, n]$, if $t = 0$, then $\rho \models_n ftp$ and the result follows directly. If $t \neq 0$ it is possible to distinguish three cases. If $modeInt = \epsilon$, it means that there is no scope interval, thus $\rho \not\models_t \phi_{right}$. By Equation (5.3), ϕ_{base} is of the

form $\phi_{right} \to \phi$ for some ϕ, thus $\rho \models_t \phi_{base}$. Similarly, if there exists $I \in modeInt$ such that $t \neq ub(I) + 1$, then, from Table 3 and Table 4 $\rho \not\models_t \phi_{right}$ and the proof proceeds in the same way. Otherwise, if $t = ub(I) + 1$, since $I \in modeInt$, it follows that $I \in \mathcal{T}[\![timing]\!]_\rho(cond, res)$, and, by Theorem 6.5, it follows that $\rho \models_{ub(I)+1} \phi_{base}$. In the following, it is shown that $\rho \models_n (\neg\phi_{right} \mathcal{S}_{inc}^{req} \phi_{left}) \to \phi_{baseLast}$. If $modeInt = \epsilon$, then ϕ_{left} is never satisfied and thus, by Table 3 and Table 4, $\rho \not\models_n \neg\phi_{right} \mathcal{S}_{inc}^{req} \phi_{left}$, and the theorem follows directly. If there exists $I \in modeInt$ such that $ub(I) = n$, then $\rho \models_n \phi_{baseLast}$. Otherwise, for all I it holds that $ub(I) < n$, thus $\rho \models_{ub(I)} \phi_{right}$ and the result follows directly since $\rho \not\models_n \neg\phi_{right} \mathcal{S}_{inc}^{req} \phi_{left}$.

Consider the $\Leftarrow$ direction. If $modeInt = \epsilon$, the theorem holds directly. Otherwise, let $I \in modeInt$, it is possible to distinguish two cases. Let $ub(I) \neq n$. By hypothesis, $\rho \models_n \mathcal{H}(\phi_{base} \vee ftp)$, this means that for all $t \in [0, n]$, $\rho \models_t \phi_{base} \vee ftp$. Since $ub(I) + 1 \neq 0$, it follows that $\rho \not\models_{ub(I)+1} ftp$ and, therefore, $\rho \models_{ub(I)+1} \phi_{base}$. The theorem follows from Theorem 6.5 since $I \in \mathcal{T}[\![timing]\!]_\rho(cond, res)$. Otherwise, consider $ub(I) = n$, this means that $I = [lb(I), n]$. By hypothesis, it holds that $\rho \models_n (\neg\phi_{right} \mathcal{S}_{inc}^{req} \phi_{left}) \to \phi_{baseLast}$. Additionally, $\rho \models_{lb(I)} \phi_{left}$ and for all $t \in I$, $\rho \models_{lb(I)} \neg\phi_{right}$, thus, $\rho \models_n \neg\phi_{right} \mathcal{S}_{inc}^{req} \phi_{left}$. In addition, $\rho \models_n \phi_{baseLast}$, by Theorem 6.5, the proof is complete. □

7 FRETish Verification: Lessons Learned

In [15], a modular and extensible verification framework has been developed to check that the formulas generated by the FRET tool conform to the RTGIL semantics of the FRETish language. The framework automatically generates large numbers of example traces using a variety of strategies, in order to cover as many corner cases as possible. For each generated trace, and for each template combination of FRETish, an oracle produces the expected truth value of requirements corresponding to this combination according to the RTGIL semantics. This expected value is compared to the value produced when evaluating, with a model checker, the formula computed by FRET for this requirement, in the particular trace. The expected and obtained values are compared, and discrepancies are reported.

As reported in [15], this framework was extremely valuable in detecting even subtle errors in the translation algorithms during the development of FRET.

Nevertheless, testing, even when extensive, is not proving. And proving, when possible, is extremely valuable for safety-critical applications. This work developed a robust, compositional proof framework, which significantly increases the confidence in the current version of the tool and facilitates a continuity of this trust in future extensions of the language. During the development of this proof framework, the following two oversights were discovered in the algorithm description of [15]. First, the baseform definition for

the timing field *eventually* was defined as ϕ_{core}. In the case where the scope field is *after(mode)*, the condition is not *null*, and *mode* is never satisfied, the scope is empty, so the requirement should be trivially true. In examining the actual implementation, it turned out that the definition was instead: $O \phi_{left} \to \phi_{core}$. Since in this case ϕ_{left} is never satisfied, the formula is trivially true, as expected. The reason this detail was missed in the documentation of the algorithm is that it was hidden in a method invocation that in fact could be simplified. As a result, that part of the implementation was updated to explicitly match the structure of the algorithm in [15]. A clear structure that matches the documented algorithms is essential for their maintenance and extensibility.

Additionally, the formula *ffim(mode)* was incorrectly defined as $fim(mode) \wedge (\mathcal{Y}(\mathcal{H} \neg mode))$ instead of $fim(mode) \wedge (ftp \vee \mathcal{Y}(\mathcal{H} \neg mode))$. This would exclude the case in which the mode holds immediately at the beginning of the trace (*ftp*). Again, the implementation was correct and this was an omission in the paper.

The development of a rigorous proof framework in PVS had several additional benefits. The implementation of the FRETish language semantics in PVS was guided by the FRET's diagrammatic explanations and simulation capabilities (see Figure 1 and Figure 3), and some ambiguities were discovered in the semantics visualization along the way. In the latest version of FRET these ambiguities were resolved, resulting in a clearer user description.

From this research effort, it emerged that the combined use of a rigorous formalization and an extensive testing framework was an excellent strategy for improving the FRET tool. The testing framework was suitable for exploring new ideas, and for providing feedback for discrepancies in an intuitive fashion during the development of FRET. In addition, it provided an oracle for the PVS semantics implementation. Even though the PVS formalization was designed to be modular and as compact as possible, the effort of proving all the results is considerable. So it was essential to start with a semantics that was already tested and did not contain substantial errors.

The rigorous PVS formalization completed the results obtained in the testing phase ensuring the coverage of all corner cases. Moreover, even though RTGIL diagrams give a more intuitive general picture of the semantics of the language, PVS constructs provide a more explicit enumeration of all special cases that are considered.

The compositional nature of the translation algorithms and proof framework means that it is possible to focus on specific parts of the verification when desired. For example, it is convenient to start modeling and implementing simple timing operators like *eventually* or *always*, and gradually expand the implementation to *for*, *after*, *until*, and *before*. Every newly added field value will naturally combine with all the existing ones which are already verified. Any new detected bug would then be easily attributed to the new value.

Finally, the compositional design of the proofs is essential to provide continued theoretical support to FRET and to incorporate, with minor effort, a theorem prover based verification step in the FRET development process.

8   Related Work

The FRETish language borrows ideas from the Specification Pattern System (SPS) literature [10], Easy Approach to Requirements Syntax (EARS, EARS-CTRL) work [21, 22], and NASA experience. The SPS work derives a set of patterns from a property specification. The patterns are structured as a *scope*, which specifies the time intervals where the requirement holds (such as *after* an event or *before* an event), and a *pattern*. The pattern is specified either as an *occurrence* pattern, e.g., specifying that a proposition occurs in the interval, or an *ordering* pattern, e.g., an event being a response to a preceding event. Each pattern/scope combination is translated into different temporal logics, such as Linear Temporal Logic (LTL) and Graphical Interval Logic (GIL). This pattern and scope translation to LTL and then to first-order logic has been used for analysis [33].

SpeAR [11] (Specification and Analysis of Requirements) captures requirements that read like natural language. It supports SPS patterns in its specification language but has reduced these patterns to a very small set, since most of them were never used in practice. Its formal language gets translated to pure past-time LTL. SpeAR provides a formal logical entailment analysis that proves that specified properties, which define desired behaviors of the system, are consequences of the set of captured assumptions and requirements. This is done by translating a SpeAR specification into an equivalent Lustre [16] model and analyzing the Lustre model using infinite-state model checking. This technique provides useful insights on the completeness of the Spear specification.

Other extensions of the SPS include: *real-time property patterns* [18] which can be translated to metric LTL, Timed CTL, RTGIL, Duration Calculus, Phase Event Automata (PEA) [27], and to Boogie [20]; *composite propositions* [23, 28, 29], which address relationships among multiple propositions and are translated to optimized future-time LTL; and *specification of semantic subtleties* [7, 31].

In [29], abstract LTL templates are introduced to support automated generation of LTL formulas for complex properties in Prospec [28]. Manual formal proofs and model-checking based testing are used to check that these templates generate the intended LTL formulas. In contrast with the work presented in this paper, the proofs in [29] are done by hand and are not formalized in a theorem prover. In addition, the structure of the proposed LTL templates is not compositional, therefore it is necessary to prove every single pattern combination.

FORM-L [25] is a formal requirements language implemented with Extended Temporal Logic (ETL) [4]. The latter has relevant constructs (e.g., constraints on the number of events occurring), but is manually translated into requirement monitors expressed using blocks in a Modelica library.

Several commercial tools are also available for formal requirements engineering. The ASSERT tool [8, 30], proprietary to GE, uses an ontology-based approach both for formalizing domains through the language SADL, and the requirements themselves, through the language SRL. Requirements are in the form of assignments to attributes conditioned on (possibly temporal) Boolean conditions. The STIM-ULUS tool [17] enables its user to specify a formal requirement by combining phrases from a library whose underlying semantics are hierarchical state machines and dataflow constraints. The phrases can specify metric temporal conditions. The behavior of the resulting set of requirements can be simulated allowing the user to observe the system behavior as specified. The BTC Embedded*Platform*® tool [32] provides a GUI to construct requirements according to a graphical "simplified universal pattern". The pattern consists of a trigger and an action, both with specified events, conditions, and timing constraints. The requirements can then be analyzed, using model-checking, for consistency, completeness, and correctness, and tests can be automatically generated.

Examples of tools that parse and formalize more general natural language are VARED [1] and ARSENAL [12]. VARED aimed at translating natural language to LTL via SALT [2]. SALT translates to LTL via rewriting rules, but, to the best of the authors' knowledge, it doesn't prove that the transformation preserves any kind of semantics. ARSENAL translates natural language to SAL [3] models.

To the best of the authors' knowledge, FRET is the only structured natural language elicitation tool that is supported by a rigorous formalization in a theorem prover.

9   Conclusion

This paper presents a formalization of the FRETish structured natural language and a proof of the correctness of the past-time MTL formula generation algorithm implemented in FRET. This proof improves the confidence in the FRET tool for its use in the requirement elicitation of safety-critical systems.

The proposed formalization provides a rigorous theoretical basis to support current and future FRETish features. The modular structure of the proof has been designed to facilitate the task of extending the language while also maintaining the correctness of the formula generation algorithm. For example, just one new lemma is needed for any new timing or scope field construct.

In [15], the equivalence between FRET generated future-time and past-time formulas is checked on finite traces of specified length. As future work, the authors plan to prove

the correctness of the algorithm also for future-time formulas over traces of *arbitrary* length. This proof can be done by leveraging the semantic equivalence between FRETish and past-time MTL shown in this paper. In this way, it will be sufficient to show that the future-time and past-time temporal formulas generated for a given requirement are semantically equivalent.

Support for infinite trace semantics has been recently added to Fret in [15]. The PVS formalization can be extended in the future to target infinite traces by replacing bounded FRETish traces with unbounded ones. Finally, the proposed formalization will be extended as new constructs and features are added to the FRETish language, keeping, in this way, a robust theoretical framework behind Fret.

References

[1] J. Badger, D. Throop, and C. Claunch. 2014. VARED: Verification and analysis of requirements and early designs. In *IEEE 22nd International Requirements Engineering Conference (RE 2014)*. 325–326. https://doi.org/10.1109/RE.2014.6912279

[2] A. Bauer and M. Leucker. 2011. The Theory and Practice of SALT. In *Proceedings of the Third International NASA Formal Methods Symposium, NFM 2011 (Lecture Notes in Computer Science, Vol. 6617)*. Springer, 13–40. https://doi.org/10.1007/978-3-642-20398-5_3

[3] S. Bensalem, V. Ganesh, Y. Lakhnech, C. Muñoz, S. Owre, H. Rueß, J. Rushby, V. Rusu, H. Saïdi, N. Shankar, E. Singerman, and A. Tiwari. 2000. An Overview of SAL. In *5th NASA Langley Formal Methods Workshop (LFM 2000)*. 187–196.

[4] D. Bouskela and A. Jardin. 2018. ETL: A new temporal language for the verification of cyber-physical systems. *Annual IEEE International Systems Conference (SysCon 2018)* (2018), 1–8. https://doi.org/10.1109/SYSCON.2018.8369502

[5] A. Champion, A. Gurfinkel, T. Kahsai, and C. Tinelli. 2016. CoCoSpec: A Mode-Aware Contract Language for Reactive Systems. In *Proceedings of the 14th International Conference on Software Engineering and Formal Methods (SEFM 2016) (Lecture Notes in Computer Science, Vol. 9763)*. Springer, 347–366. https://doi.org/10.1007/978-3-319-41591-8_24

[6] A. Cimatti, E. M. Clarke, F. Giunchiglia, and M. Roveri. 2000. NUSMV: A New Symbolic Model Checker. *International Journal on Software Tools for Technology Transfer (STTT)* 2, 4 (2000), 410–425. https://doi.org/10.1007/s100090050046

[7] R. L. Cobleigh, G. S. Avrunin, and L. A. Clarke. 2006. User Guidance for Creating Precise and Accessible Property Specifications. In *Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering (SIGSOFT '06/FSE-14)*. ACM, 208–218. https://doi.org/10.1145/1181775.1181801

[8] A. Crapo, A. Moitra, C. McMillan, and D. Russell. 2017. Requirements Capture and Analysis in ASSERT. In *25th IEEE International Requirements Engineering Conference (RE 2017)*. 283–291. https://doi.org/10.1109/RE.2017.54

[9] A. Dutle, C. A. Muñoz, E. Conrad, A. Goodloe, L. Titolo, I. Perez, S. Balachandran, D. Giannakopoulou, A. Mavridou, and T. Pressburger. 2020. From Requirements to Autonomous Flight: An Overview of the Monitoring ICAROUS Project. In *Proceedings Second Workshop on Formal Methods for Autonomous Systems, FMAS 2020, Virtual, 7th of December 2020 (EPTCS, Vol. 329)*. 23–30. https://doi.org/10.4204/EPTCS.329.3

[10] M. B. Dwyer, G. S. Avrunin, and J. C. Corbett. 1999. Patterns in Property Specifications for Finite-state Verification. In *Proceedings of the 21st International Conference on Software Engineering (ICSE '99)*. ACM, 411–420. https://doi.org/10.1145/302405.302672

[11] A. W. Fifarek, L. G. Wagner, J. A. Hoffman, B. D. Rodes, M. A. Aiello, and J. A. Davis. 2017. SpeAR v2.0: Formalized Past LTL Specification and Analysis of Requirements. In *Proceedings of the 9th International NASA Formal Methods Symposium (NFM 2017) (Lecture Notes in Computer Science, Vol. 10227)*. 420–426. https://doi.org/10.1007/978-3-319-57288-8_30

[12] S. Ghosh, D. Elenius, W. Li, P. Lincoln, N. Shankar, and W. Steiner. 2016. ARSENAL: Automatic Requirements Specification Extraction from Natural Language. In *Proceedings of the 8th International NASA Formal Methods Symposium (NFM 2016)*. 41–46. https://doi.org/10.1007/978-3-319-40648-0_4

[13] D. Giannakopoulou, T. Pressburger, A. Mavridou, J. Rhein, J. Schumann, and N. Shi. 2020. Formal Requirements Elicitation with FRET. In *Joint Proceedings of REFSQ-2020 Workshops, Doctoral Symposium, Live Studies Track, and Poster Track co-located with the 26th International Conference on Requirements Engineering: Foundation for Software Quality (REFSQ 2020)*.

[14] D. Giannakopoulou, T. Pressburger, A. Mavridou, and J. Schumann. 2020. Generation of Formal Requirements from Structured Natural Language. In *26th International Working Conference on Requirements Engineering: Foundation for Software Quality, REFSQ 2020 (Lecture Notes in Computer Science, Vol. 12045)*. Springer, 19–35. https://doi.org/10.1007/978-3-030-44429-7_2

[15] D. Giannakopoulou, T. Pressburger, A. Mavridou, and J. Schumann. 2021. Automated formalization of structured natural language requirements. *Inf. Softw. Technol.* 137 (2021), 106590. https://doi.org/10.1016/j.infsof.2021.106590

[16] E. Jahier, P. Raymond, and N. Hawlbwachs. 2020. *The Lustre V6 Reference Manual*. Technical Report.

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

[18] S. Konrad and B. H. C. Cheng. 2005. Real-time Specification Patterns. In *Proceedings of the 27th International Conference on Software Engineering (ICSE '05)*. ACM, 372–381. https://doi.org/10.1145/1062455.1062526

[19] R. Koymans. 1990. Specifying Real-Time Properties with Metric Temporal Logic. *Real Time Systems* 2, 4 (1990), 255–299. https://doi.org/10.1007/BF01995674

[20] V. Langenfeld, D. Dietsch, B. Westphal, J. Hoenicke, and A. Post. 2019. Scalable Analysis of Real-Time Requirements. In *IEEE 27th International Requirements Engineering Conference (RE 2019)*. 234–244. https://doi.org/10.1109/RE.2019.00033

[21] L. Lúcio, S. Rahman, C. Cheng, and A. Mavin. 2017. Just Formal Enough? Automated Analysis of EARS Requirements. In *Proceedings of the 9th International NASA Formal Methods Symposium (NFM 2017) (Lecture Notes in Computer Science, Vol. 10227)*. Springer, 427–434. https://doi.org/10.1007/978-3-319-57288-8_31

[22] A. Mavin, P. Wilkinson, A. Harwood, and M. Novak. 2009. Easy Approach to Requirements Syntax (EARS). In *Proceedings of the 2009 17th IEEE International Requirements Engineering Conference (RE '09)*. 317–322. https://doi.org/10.1109/RE.2009.9

[23] O. Mondragon and A. Q. Gates. 2004. Supporting Elicitation And Specification Of Software Properties Through Patterns And Composite Propositions. *International Journal of Software Engineering and Knowledge Engineering* 14, 1 (2004), 21–41. https://doi.org/10.1142/S0218194004001567

[24] L. E. Moser, P. M. Melliar-Smith, Y. S. Ramakrishna, G. Kutty, and L. K. Dillon. 1996. The Real-Time Graphical Interval Logic Toolset. In *Proceedings of the 8th International Conference on Computer Aided Verification (CAV '96) (Lecture Notes in Computer Science, Vol. 1102)*. Springer, 446–449. https://doi.org/10.1007/3-540-61474-5_99

[25] T. Nguyen. 2019. Formal Requirements and Constraints Modelling in FORM-L for the Engineering of Complex Socio-Technical Systems. In

27th IEEE International Requirements Engineering Conference Workshops (REW 2019). 123–132. https://doi.org/10.1109/REW.2019.00027

[26] S. Owre, J. Rushby, and N. Shankar. 1992. PVS: A Prototype Verification System. In *Proceeding of the 11th International Conference on Automated Deduction (CADE-11) (Lecture Notes in Computer Science, Vol. 607)*. Springer, 748–752. https://doi.org/10.1007/3-540-55602-8_217

[27] A. Post and J. Hoenicke. 2012. Formalization and Analysis of Real-Time Requirements: A Feasibility Study at BOSCH. In *Proceedings of the 4th International Conference on Verified Software: Theories, Tools, Experiments (VSTTE 2012) (Lecture Notes in Computer Science, Vol. 7152)*. Springer, 225–240. https://doi.org/10.1007/978-3-642-27705-4_18

[28] S. Salamah, A. Gates, and V. Kreinovich. 2012. Validated Templates for Specification of Complex LTL Formulas. *J. Syst. Softw.* 85, 8 (2012), 1915–1929. https://doi.org/10.1016/j.jss.2012.02.041

[29] S. Salamah, A. Gates, S. Roach, and M. Engskow. 2011. Towards Support for Software Model Checking: Improving the Efficiency of Formal Specifications. *Adv. Software Engineering* 2011 (June 2011). https://doi.org/10.1155/2011/869182

[30] K. Siu, A. Moitra, M. Durling, A. Crapo, M. Li, H. Yu, H. Herencia-Zapana, M. Castillo-Effen, S. Sen, C McMillan, D. Russel, S. Roy, and P.

Manolios. 2017. Flight critical software and systems development using ASSERT™. In *IEEE/AIAA 36th Digital Avionics Systems Conference (DASC 2017)*. IEEE, 1–10. https://doi.org/10.1109/DASC.2017.8102059

[31] R. L. Smith, G. S. Avrunin, L. A. Clarke, and L. J. Osterweil. 2002. PRO-PEL: An Approach Supporting Property Elucidation. In *Proceedings of the 24th International Conference on Software Engineering (ICSE '02)*. ACM, 11–21. https://doi.org/10.1109/ICSE.2002.1007952

[32] T. Teige, T. Bienmüller, and H. J. Holberg. 2016. Universal Pattern: Formalization, Testing, Coverage, Verification, and Test Case Generation for Safety-Critical Requirements. In *19th GI/ITG/GMM Workshop Methoden und Beschreibungssprachen zur Modellierung und Verifikation von Schaltungen und Systemen (MBMV 2016)*. Albert-Ludwigs-Universität Freiburg, 6–9. https://doi.org/10.6094/UNIFR/10633

[33] B. Walter, Hammes J., M. Piechotta, and S. Rudolph. 2017. A Formalization Method to Process Structured Natural Language to Logic Expressions to Detect Redundant Specification and Test Statements. In *25th IEEE International Requirements Engineering Conference (RE 2017)*. IEEE Computer Society, 263–272. https://doi.org/10.1109/RE.2017.38