FRET2WEST: Exploring translation between temporal logic representations

Songyan Lai[*] and Rosemary Monahan[*]
[*]Maynooth University, Ireland

ABSTRACT Software requirements are commonly expressed in natural-language, which must be formalised if they are to be used by formal methods, where we verify that an implementation abides by its requirements during execution. Tools such as Formal Requirements Elicitation Tool (FRET) support this formalization of requirements, and generates Linear Temporal Logic (LTL) for each, providing formulas that can be used in their formal verification, typically via model checking. In this paper, we explore the integration of tools which work with different representations of the requirements. Specifically, we focus on the translation of the LTL output from FRET to Mission-time LTL (MLTL). This allows the integration of FRET with WEST, a tool that transforms MLTL formulas into logically equivalent regular expressions describing the set of all satisfying traces, which facilitates the verification of the specifications via Runtime Verification tools such as R2U2. This integration therefore allows WEST to validate the specifications generated by FRET. Our solution, FRET2WEST, utilizes regular expression-based mapping, variable normalization, and mission-time interval translation to transform FRET's LTL specifications to WEST's MLTL syntax. A proof-of-concept JavaScript translator iteratively constructed through case studies, is added as a feature within the FRET tool.

KEYWORDS Safety-critical systems, Formal verification, Temporal logic, FRET, WEST, Toolchain interoperability.

1. Introduction

The engineering of safety-critical software systems, particularly in the aerospace, automotive, and medical sectors, demands a rigorous requirements engineering approach to guarantee the correctness and reliability of the systems. In these domains, requirements frequently specify intricate safety and robustness properties that must be unambiguously defined, formalized, and validated. Linear Temporal Logic (LTL) and its extensions, such as Mission-time LTL (MLTL) (Maler & Nickovic 2004), are widely used to formally define temporal properties algorithmically checkable against system models. Yet, the formalization of natural language requirements and their verification remain challenging, generally being hindered by vagueness, human errors, and tool compatibility problems (Mokos & Katsaros 2020). This research project tackles these difficulties through the combination of two complementary tools: the Formal Requirements Elicitation Tool (FRET) and the WEST tool. FRET bridges the gap between natural language requirements and formal temporal logic specifications, ensuring that requirements can be model checked for correctness in the system model using tools such as Kind2 and NuSMV (Giannakopoulou et al. 2020). FRET provides a structured natural-language, called FRETISH, from which temporal logic specifications and other verification conditions can be derived. FRET has supported both elicitation and formalisation of FRETISH requirements in many use cases. Examples which we have detailed knowledge of include an aircraft engine controller (Farrell, Luckcuck, et al. 2022a), a mechanical lung ventilator (Farrell et al. 2024b), a rover carrying out an inspection task (Bourbouh et al. 2021), an algorithm for autonomously grasping spent rocket stages (Farrell, Mavrakis, et al. 2022) and a tilt-rotor drone (Sheridan et al. 2025). In these use cases the requirements were elicited and subsequently used for specifying properties to be verified in various formal methods.

Mission-time Linear Temporal Logic (MLTL) is a finite, discrete, closed-interval-bounded variant of Metric Temporal Logic

JOT reference format:
Songyan Lai and Rosemary Monahan. *FRET2WEST: Exploring translation between temporal logic representations* . Journal of Object Technology. Vol. 25, No. 1, 2026. Licensed under Attribution 4.0 International (CC BY 4.0) http://dx.doi.org/10.5381/jot.2026.25.1.a2

(MTL) often used to specify requirements for safety-critical systems, such as aircraft and spacecraft (Wang et al. 2026). The WEST tool provides an interactive visualization tool for MLTL that allows practitioners to validate that their MLTL specifications match the intended requirements [1]. WEST transforms MLTL formulas into logically equivalent regular expressions describing the set of all satisfying traces, facilitating the verification of the specifications via runtime verification tools such as the runtime monitoring tool R2U2 (Elwing et al. 2024). The graphic interface also allows users to analyze MLTL formulas by randomly generating satisfying and unsatisfying traces, to see how changing the truth value of variables at different steps affects the formula, to import and export traces from files, and more. The improved explainability of formulas through visualizing and interacting with traces makes it easier for designers to demonstrate the correctness of their specifications.

The purpose of integrating the FRET and WEST tools is to facilitate complete traceability of requirements from their initial natural language expression in FRET through their formal verification and validation with WEST so as to improve tool-chain interoperability and traceability of the requirements. Improving requirements engineering for making it more reliable and efficient for safety-critical systems is a key focus of the effort.

The remainder of this paper has the following outline: Section 2 introduces the background theories of temporal logic and formal verification techniques. Section 3 introduces the problem of integrating FRET and WEST. Section 4 introduces the solution to integrating the two approaches. Section 5 evaluates the framework through case studies. Finally, Section 6 concludes the paper and suggests future work.

2. Technical Background

Formal verification is a mathematical technique that ensures system correctness through exhaustive proof of conformance to specifications under all possible execution contexts. Unlike conventional testing methods that validate subsets of behaviors or operate without explicit specifications, formal verification rigorously proves requirement satisfaction. This makes it indispensable for high-reliability domains such as aerospace, automotive, and medical systems. A notable example is NASA's Orion spacecraft project, where formal verification ensured flight control software reliability under extreme conditions (Hirshorn et al. 2017).

The formal verification ecosystem comprises diverse tools addressing specific phases of the development lifecycle. Our research identifies the most critical gap in current formal verification infrastructure: tool isolation.

2.1. Formal Verification Tools: Landscape and Applications

The formal verification ecosystem comprises diverse tools, each addressing specific phases of the development lifecycle.

Model Checkers (e.g. nuXmv, SPIN): Model Checkers form exhaustive state-space exploration to verify temporal logic properties. For instance, nuXmv employs symbolic model checking to validate LTL formulas against state models (Cavada et al. 2014). Some sample applications are the verification of fault-tolerant algorithms in NASA's Lunar Gateway life support systems (Giannakopoulou et al. 2020) and validation of cache coherence protocols for Intel processors (Holzmann 2004). Limitations include state-space explosion limits scalability for systems exceeding 10^6 states (e.g., autonomous vehicles with multi-sensor integration) (Cavada et al. 2014).

Satisfiability Solvers (e.g. MLTLSAT, Z3): Satisfiability Solvers encode temporal logic formulas into Boolean satisfiability (SAT) problems; MLTLSAT specializes in Mission-Time LTL (MLTL) for bounded mission phases (Li et al. 2019). Some sample applications are feasibility analysis for path planning under mission-time constraints in drone swarms, for example ensuring collision-free trajectories within 100 mission steps. Limitations are lack of runtime validation capabilities and traceability to natural language requirements.

Runtime Monitors (e.g. R2U2): Runtime Monitors execute real-time compliance checks using LTL/MTL/MLTL during system operation; R2U2 leverages FPGA hardware for low-latency anomaly detection (Moosbrugger et al. 2017). Some sample applications are real-time monitoring of robotic arm operations to prevent collisions during ISS missions in NASA's Robonaut2 (Moosbrugger et al. 2017) and enforcement of collision avoidance protocols in dynamic environments for autonomous drones. Limitations are: runtime overhead, no guarantee absence of design-time bugs, and potentially missing faults outside monitored traces.

Requirements Elicitation Tools (e.g. FRET, COCOSIM): Requirements Elicitation Tools translate natural language requirements into formal specifications (Mavridou 2022). Some sample applications are formalization of lunar rover autonomy requirements in NASA's Artemis Program (e.g., "Under sensor faults, while tracking pilot commands, control objectives shall be satisfied") (De Ferro et al. 2023) and verification of TESLA's autonomous emergency braking systems (AEB) to ensure compliance with ISO 26262 functional safety standards (Krammer et al. 2015). Limitations are heavily dependent on the clarity of input text and may produce ambiguous or incomplete formalizations without domain expert intervention.

After comparison, analysis, and research, our finding is that the most pressing gap between the existing formal verification tools today is: the current tools are isolated from each other. For example, FRET generates LTL specifications but cannot be used to prove mission-time properties within WEST. Such a form of fragmentation requires an integrated framework that unifies elicitation with validation.

2.2. Linear Temporal Logic Variants and Tool Support

Linear Temporal Logic (LTL), introduced by Pnueli in 1977, extends propositional logic with temporal operators to reason about sequences of system states (Pnueli 1977). It plays an important role in formal verification, and its variants cater to diverse application needs. Next, various types of LTL used in this paper will be introduced as a technical background supplement.

[1] https://github.com/zwang271/WEST

Future Time LTL

- *Function*: Focuses on predicting future system behaviors through four core operators (Manna & Pnueli 1992).
- *Key operators*:
  - ◇ `F` ("Eventually"): "Something will happen eventually."
  - ◇ `G` ("Globally"): "Something must always hold true."
  - ◇ `X` ("Next"): "Something will happen in the next step."
  - ◇ `U` ("Until"): "Condition A must hold until Condition B occurs."
- *Example*: Ensuring a self-driving car eventually stops when detecting a pedestrian (`F stop`).
- *Tools*: FRET, nuXmv.

Past Time LTL

- *Function*: Analyzes past system states to diagnose issues (Kamp 1968).
- *Key operators*:
  - ◇ `P` ("Previously"): "Something happened in the past."
  - ◇ `H` ("Historically"): "Something has always been true in the past."
  - ◇ `S` ("Since"): "Condition A has held since Condition B occurred."
- *Example*: Diagnosing why a smart thermostat failed to turn on by checking if it previously received a temperature spike (`P temperature_high`).
- *Tools*: R2U2.

Metric Temporal Logic (MTL)

- *Function*: Adds time intervals to LTL for real-time constraints (Koymans 1990).
- *Key operators*:
  - ◇ `F[9,10]` ("Eventually between 9–10 seconds").
  - ◇ `G[0,100]` ("Always true for the first 100 seconds").
- *Example*: A medical ventilator must always maintain oxygen levels above 90% within 2 seconds of detecting a drop (`G[0,2] oxygen_safe`).
- *Tools*: UPPAAL, R2U2.

Mission-Time LTL (MLTL)

- *Function*: Specifies mission phases with integer time bounds (Li et al. 2019).
- *Key operators*:
  - ◇ `F[0,10]` ("Must happen within the first 10 mission steps").
  - ◇ `G[5,∞]` ("Must hold true from step 5 until mission end").
- *Example*: A Mars rover must retract its solar panels within 15 steps after detecting a dust storm (`F[0,15] retract_panels`).
- *Tools*: WEST, MLTLSAT.

In short, Future/Past Time LTL is concerned with events that will or did occur, and MTL/MLTL adds specific timing constraints to meet the requirements of real-time and mission-critical applications. Table 1 shows some key differences between LTL and MLTL. Incompatibilities in tool support, as exemplified by the mismatch between FRET's LTL and WEST's MLTL, pose great challenges in practical applications. A representative example is presented below:

FRET's LTL: `G (sensor_fault` $\rightarrow$ `X shutdown)`

WEST's MLTL: `G[0,100] (p0` $\rightarrow$ `F[1,1] p1)`

The syntactic mismatch between FRET's contextual variables and WEST's generic labels, together with the unbounded vs. bounded operator difference, presents an intrinsic interoperability issue. This limitation motivates the integration efforts of this project. Section 3 will discusses the approach to achieving this automation in detail.

2.3. FRET: Bridging Natural Language and Formal Logic

Formal Requirements Elicitation Tool (FRET), developed by NASA and extended by Maynooth University's MU-FRET team, translates natural language requirements into formal specifications using its structured English dialect, FRETISH. Each FRETISH requirement is automatically translated into Past-Time LTL and Future-Time LTL. FRET can automatically translate FRETISH requirements to contracts, which can be verified with the Kind2 model checker, and COPILOT runtime monitors (Perez et al. 2022). The input structure of FRETISH is as follows: `scope` `condition` `component` shall `timing` `response`.

The `component` and `response` fields are mandatory (along with the "shall" keyword) for all FRETISH requirements; `scope`, and `timing` are optional. Using FRETISH, users can express requirements for an individual `component` that pertain to a particular `scope` under some where the `response` (expected behaviour) can be specific to a defined `timing`. The underlying semantics of a FRETISH requirement is determined by the `scope`, `condition` and `timing` fields. It is important to note that if the user has not supplied a `timing` constraint, then the requirement must hold eventually. Built in JavaScript, FRET provides 160 templates that describe how to translate each combination of a requirement's fields into temporal logic (Giannakopoulou et al. 2021). FRET selects the template that corresponds to the requirement and uses it to generate formalisations in both Past- and Future-time Temporal Logic.

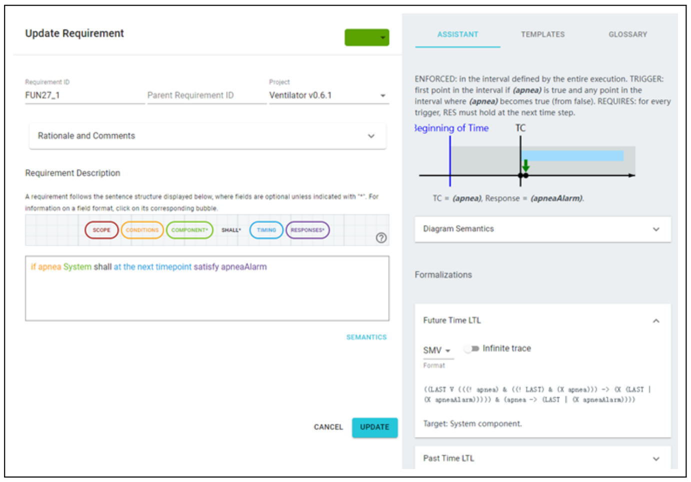
For example, the requirement:

"if apnea System shall at the next timepoint satisfy apneaAlarm"

Maps if apnea to , System to `component`, at the next timepoint to `timing`, and satisfy apneaAlarm to `response`, generates the output "`((LAST V (((! apnea) & ((! LAST) & (X apnea))) -> (X (LAST | (X apneaAlarm))))) & (apnea -> (LAST | (X apneaAlarm))))`" in Future-time LTL. Here LAST is a boolean that records whether a condition held before and LAST V refers to the actual previous

Feature	LTL	MLTL
Time Bounds	Unbounded (F, G)	Bounded (F[0,10])
Variable Syntax	Contextual names	Generic (p0, p1)

Table 1 Syntactic and Semantic Differences between LTL and MLTL.


Update Requirement

Requirement ID: FUN27_1
Parent Requirement ID
Project: Ventilator v0.6.1

Rationale and Comments

Requirement Description

A requirement follows the sentence structure displayed below, where fields are optional unless indicated with "*". For information on a field format, click on its corresponding bubble.

if apnea System shall at the next timepoint satisfy apneaAlarm

ENFORCED: in the interval defined by the entire execution. TRIGGER: first point in the interval if *(apnea)* is true and any point in the interval where *(apnea)* becomes true (from false). REQUIRES: for every trigger, RES must hold at the next time step.

TC = *(apnea)*, Response = *(apneaAlarm)*.

Diagram Semantics

Formalizations

Future Time LTL

SMV — Infinite trace

Format

```
((LAST V (((! apnea) & ((! LAST) & (X apnea))) -> (X (LAST |
(X apneaAlarm))))) & (apnea -> (LAST | (X apneaAlarm))))
```

Target: System component.

Past Time LTL

Figure 1 Screenshot of FRET showing a requirement named FUN27_1. Each requirement has a natural language input (left), and a diagrammatic semantics (right) is generated by the tool to help the user to understand the meaning behind the FRETISH requirement. Each requirement is translated to Temporal Logic within FRET. See Future Time LTL displayed above for FUN27_1.

value of a variable. The formula generated reads as follows: The last value of apnea was false (!apnea), but at the next step (X apnea) it becomes true. If apnea just started, then in the next timepoint, either LAST holds or the apneaAlarm will be set, and as long as apnea is true, the alarm will also be set.

Formulas are exported to model checkers or saved as structured text files.

Fig. 1 is a screenshot of FRET's 'Update Requirement' dialogue, with the final FRETISH version of the requirement named FUN27_1.

2.4. WEST: Validating Mission-Critical Timelines

Modern cyber–physical systems (e.g. spacecraft, UAV swarms, air-traffic control) often specify safety and mission-critical requirements as *timelines*. For example: "If the system remains in `safe` mode during steps 0–50, then at step 51 the `fire_thruster` command must be issued."

Traditional testing can only sample a finite subset of possible executions and thus cannot guarantee coverage of every critical scenario, while formal methods (such as model checkers or theorem provers), though exhaustive, produce proofs or counterexamples that are often opaque to engineers and certification authorities. We therefore need a way for all stakeholders, including designers, auditors, and certifiers, to *visually* and *completely* see which timelines satisfy a requirement and which do not.

WEST (acronym of the first names of its authors: Zili Wang, Jenna Elwing, Jeremy Sorkin and Chiara Travesset) is an open-source C++/Python tool that *automatically* validates MLTL (Mission-Time Linear Temporal Logic) specifications by synthesizing *all* satisfying finite timelines into a compact regular expression representation. By transforming each MLTL formula into a deterministic specification of "what can happen when", WEST provides system designers, certification engineers, and stakeholders with an intuitively readable "truth-table" for temporal requirements.

The overall workflow of WEST is described in detail by WEST Team (Elwing et al. 2024); the input is an MLTL formula and the overall output is a set of fixed-length trace regular expressions (with GUI visualizations) that together enumerate all satisfying timelines.This process is broken down into the following.

Overall Workflow

1. **Formula Input.** Users write MLTL formulas in WEST's strict syntax. For example, the informal requirement

   "If p_0 holds at step 0 and p_1 holds throughout steps 0–3, then p_2 must hold at step 0."

   is encoded as

   $$((p0 \ \& \ G[0,3]\, p1) \ \rightarrow \ p2).$$

   All temporal operators require explicit closed-interval bounds and parentheses.

2. **AST Construction.** A recursive-descent parser builds an Abstract Syntax Tree (AST). Internally, each node is one of:

   $$\begin{gathered} PropTrue/PropFalse, \quad p_k, \quad \neg p_k, \\ \phi_1 \wedge \phi_2, \quad \phi_1 \vee \phi_2, \\ F[a,b]\phi, \quad G[a,b]\phi, \\ \phi_1\, U[a,b]\, \phi_2, \quad \phi_1\, R[a,b]\, \phi_2. \end{gathered}$$

   Note on the Next operator (X): The WEST framework does not introduce Next ('X') as a primitive AST node. For discrete, bounded MLTL the next-step modality is a special case of bounded Finally/Globally: in particular

   $$X\varphi \quad \equiv \quad F[1,1]\varphi \quad \text{(equivalently } G[1,1]\varphi\text{)}.$$

   Accordingly, any front-end parser or converter that accepts 'X' may perform a trivial preprocessing pass that rewrites every occurrence of 'X' to 'F[1,1]' before AST construction.

   WEST recommend documenting this desugaring step in the user/front-end documentation so that importers (e.g. FRET

$\to$ WEST converters) do not require manual edits (Elwing et al. 2024). For our example, the AST root is a "$\to$" node whose left child is an "$\wedge$" combining a proposition and a "G" subtree.

3. **NFA Generation.** WEST traverses the AST and, for each subformula, constructs a small non-deterministic finite automaton (NFA) that accepts exactly those bit-string timelines satisfying the subformula over a fixed horizon m.

   Base case: p_k becomes an NFA that checks bit $k = 1$ at the first symbol.
   Finally $F[a,b]\phi$: Inserts an "epsilon-jump" allowing ϕ to hold at some step in $[a,b]$.
   Until $U[a,b]$: Chains $G[0,i-1]\phi_1$ then "epsilon-jump" to ϕ_2 at step $i \in [a,b]$.

4. **Regular Expression Synthesis.** From the product NFA, WEST applies a variant of the state-elimination algorithm to produce a fixed-length regular expression. To avoid ambiguity we distinguish the concrete alphabet used for bit-strings from a convenient shorthand used in the paper:

   Concrete alphabet: $\Sigma = \{$`0, 1, ,`$\}$. Here '`0`' and '`1`' are literal bit values and '`,`' is the time-step separator (comma) that appears between time-step tokens.
   Shorthand `S`: For compactness we use '`S`' as a per-bit "don't-care" symbol that matches either '`0`' or '`1`'. Formally, each occurrence of '`S`' may be expanded into the set $\{$`0`$,$`1`$\}$ when explicit strings or set-wise operations are required.
   Usage note: The resulting regexes are therefore fixed-length sequences of time-step tokens over Σ (with '`S`' as a notational abbreviation); these expressions are amenable to sampling, simplification, and GUI visualization. (Example: for $n = 2$, the token '`S1`' denotes the pair $\{01, 11\}$.)

5. **Visualization & Exploration.** In the GUI (Fig. 2), users can browse the abstract syntax tree (AST) and the NFA graph, view the generated regular expression, and randomly sample a satisfying or violating bit-string, then toggle individual bits to observe how those changes affect the formula's outcome.

Concrete Mini-Example. Consider three propositional variables p_0, p_1, p_2 (so $n = 3$) and the formula

$$\phi = (p_0 \wedge F[1,2]\, p_1) \to p_2.$$

Intuition: "If p_0 holds at step 0 and p_1 eventually holds at step 1 or 2, then p_2 must hold at step 0." This antecedent and consequent are distinct, so both satisfying and violating computations exist. Computation horizon: Using the definition of computation length, WEST computes $m = \mathrm{cplen}(\phi) = 3$, which suffices to cover the interval $[1,2]$. Example satisfying and violating timelines (bit-string form):

A *satisfying* computation (antecedent false or consequent true): for example `0S,SS,1SS`. Here the first time-step '`0S`' means $p_0 = 0$ (antecedent false), so the implication is satisfied regardless of other bits.

A *violating* computation (antecedent true and consequent false): for example `1S,1S,0SS`. Interpreting '`1S`' at step 0 gives $p_0 = 1$; '`1S`' at step 1 supplies $p_1 = 1$ (thus $F[1,2]p_1$ is satisfied); and '`0SS`' at step 0 sets $p_2 = 0$ (the consequent false), therefore the implication is violated.

WEST produces a union of fixed-length temporal regular expressions that partition the length-3 computations into satisfying and violating patterns. The GUI lets users inspect these regexes, sample concrete bit-strings from each class, and toggle individual bits (for example, toggling the bit corresponding to p_2 at step 0 flips the computation between satisfying and violating (Elwing et al. 2024).

Strengths of this approach include:

Soundness & Completeness: Every timeline satisfying the MLTL formula is captured (and none extraneously) by the generated regex (Elwing et al. 2024).

Mission-Alignment: Bounded temporal operators map directly to real-world mission phases (e.g. "0–100 steps = lunar descent").

Usability: Graphical interface with AST/NFA views and interactive bit toggling bridges the gap between formal logic and system engineers.

Limitations which we note are:

Manual Pre-Processing: External tools (e.g. FRET) emit plain LTL; users must manually rewrite $X\,\mathit{shutdown}$ as $F[1,1]\;p_0$ before importing.

Semantic Mapping: Variables $p_0, p_1, \ldots$ are anonymous; without an auxiliary mapping table, trace values lose original requirement names.

Scalability: Deeply nested binary temporal operators can blow up regex size and NFA states (worst-case exponential), though real-world specs seldom hit this boundary.

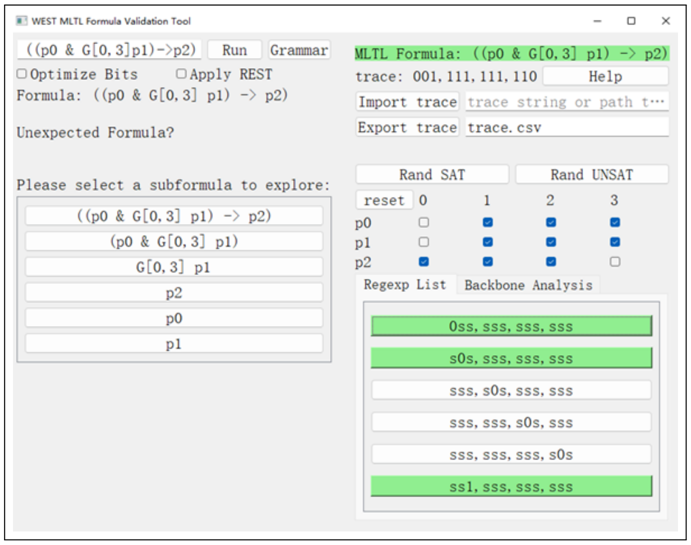
Case Study: As an illustrative example (from WEST documentation), consider the simple pattern

$$\mathit{sensor\_error} \to X\,\mathit{shutdown},$$

which—after desugaring the next operator X as $F[1,1]$ (see Section 4)— is written in WEST-style MLTL (with a propositional mapping $\mathit{sensor\_error} \mapsto p_0$, $\mathit{shutdown} \mapsto p_1$) as

$$G[0,100]\big(p_0 \to F[1,1]\, p_1\big).$$

Using WEST to visualize the set of satisfying and violating length-m traces for such patterns helps engineers spot mapping or translation mistakes early: the GUI's ability to display AST/NFA views, enumerate regex-backed traces, and sample or toggle concrete bit-strings makes mismatches and unintended encodings visually apparent and easy to investigate. Documentation of concrete Robonaut2-related applications of MLTL and runtime monitoring are available via the WEST GitHub repository.


WEST MLTL Formula Validation Tool

```
((p0 & G[0,3]p1)->p2)   Run   Grammar
☐ Optimize Bits     ☐ Apply REST
Formula: ((p0 & G[0,3] p1) -> p2)

Unexpected Formula?

Please select a subformula to explore:
((p0 & G[0,3] p1) -> p2)
(p0 & G[0,3] p1)
G[0,3] p1
p2
p0
p1
```

```
MLTL Formula: ((p0 & G[0,3] p1) -> p2)
trace: 001,111,111,110   Help
Import trace   trace string or path t…
Export trace   trace.csv
Rand SAT   Rand UNSAT
```

reset	0	1	2	3
p0	☐	✓	✓	✓
p1	☐	✓	✓	✓
p2	✓	✓	✓	☐

```
Regexp List   Backbone Analysis
0ss,sss,sss,sss
s0s,sss,sss,sss
sss,s0s,sss,sss
sss,sss,s0s,sss
sss,sss,sss,s0s
ss1,sss,sss,sss
```

Figure 2 Showing the scripts and data which was introduced below, and it constructs a finite automaton to generate regular expressions. The left side is a list of sub formulas the user can visualize, and the right side is all satisfying computations to MLTL (i.e., describe the valuations, or the rows of the 'truth table', of a given MLTL formula that satisfy it).

2.5. Existing Integrations and Research Gaps

Efforts to couple requirement formalization with verification tools have not been so successful. The FRET, for example, can produce LTL formulas to nuXmv for model checking of unbounded temporal properties (e.g. Liveness), and is coupled with CoCoSim to generate CoCoSpec assume–guarantee contracts for Simulink blocks (Luckcuck et al. 2022). These are embedded in the Simulink diagram and checked at simulation time using the Kind2 model checker. FRET also communicates with the COPILOT runtime monitoring infrastructure, translating user-provided monitors to hard real-time C code that can be deployed on embedded platforms (Luckcuck et al. 2022).

However, FRET's pipeline does not inherently provide mission-time bounds such as MLTL's bounded intervals, which makes it unsuitable for finite-duration missions. FRET mitigates ambiguity by translating vague terms (e.g., "immediately" to `F[0,1]` (Giannakopoulou et al. 2020)) and ensures bidirectional traceability via embedded metadata (e.g., FRET_ID, Requirement ID [2]). In this work we explore support for Mission-time LTL (MLTL) compatibility, as currently we do not have support for mission-time constraints like `F[0,10]`, and its variable naming conventions conflict with WEST's syntax, causing compatibility challenges. For instance, manual translation of FRET's LTL outputs to MLTL introduces semantic loss due to generic variables (e.g., `p0, p1`), which obscure original requirement context (Farrell, Luckcuck, et al. 2022b).

The Apache R2U2 library uses WEST to validated MLTL formulae to monitor run-time for unmanned aerial vehicles (UAVs) (Kouskoulas et al. 2020). However, this is done manually, which is error-prone as engineers need to rename variables (e.g. `sensor_fault` → `p0`) and manually adjust temporal operators (Farrell et al. 2024a).

Despite these advances, three key gaps persist:

Syntax Mismatch: FRET and R2U2 are distinct temporal-logic dialects that need to be specially rewritten in order to interoperate.

Traceability Loss: Anonymous variables (e.g. `p0`, `p1`) obscure the original requirement semantics when mapping from high-level specifications.

Feedback Loop Deficiency: Manual translation steps prevent a seamless round-trip workflow between specification, verification, and stakeholder review.

These propel the need for end-to-end integration that preserves bounded-interval semantics, requirement traceability, and provides instant feedback—concerns we discuss more thoroughly in Section 3.

3. Technical Challenges and Design Implementations

The multi-layer technical challenges encountered when integrating FRET and WEST. These challenges, which are rooted in syntactic incompatibility, semantic divergence and toolchain fragmentation, are further exacerbated by the mission-critical domains involved, such as aerospace and medical systems, where a single misstep in translation can compromise safety.

3.1. Syntax and Semantic Mismatches

The syntactic differences between FRET and WEST stem from their distinct design goals: FRET provides a human-readable FRETISH syntax and exports standard LTL, whereas WEST employs machine-optimized MLTL syntax with bounded intervals and generic proposition identifiers. Table 2 summarizes the principal distinctions.

These mismatches manifest in three key aspects:

Temporal operator mapping FRET employs unbounded LTL operators (e.g. F, G) based on infinite traces, whereas WEST requires explicit bounds. Translating

$$G(sensor\_fault \to shutdown) \text{ to } G[0, mission\_duration]\,(p0 \to p1)$$

demands context-dependent mission-time mapping—a non-trivial issue, since aerospace systems define "mission time" variably. In NASA's Robonaut2 mission, naïve mappings from unbounded G to arbitrary intervals caused caused a high rate of validation failures (Giannakopoulou et al. 2020).

Variable abstraction and traceability loss FRET uses descriptive labels (e.g. `engine_overheat`), while WEST reduces all variables to generic names like `p0`, `p1`, etc. Manual renaming not only introduces human error but also severs the semantic connection to the original requirement. For instance, misaligning FRET's `LAST V` operator during translation can cause otherwise valid formulas to be rejected by WEST.

[2] https://github.com/valu3s-mu/mu-FRET

Feature	FRET	WEST
Time Operators	Unbounded (F, G), finite-trace via `LAST V`	Bounded intervals (F$[a,b]$, G$[a,b]$), infinite traces only
Variables	Contextual labels (e.g. `engine_overheat`)	Generic propositions (p0, p1, . . .)
Trace Semantics	Explicit finite-trace support via `LAST V`	No terminal-state markers
Time Quantification	Real-world units (within 5 s, after 10 s)	Discrete step intervals (e.g. F$[0,5]$, G$[1,10]$)
Past-Time Logic	Native support (e.g. `persisted`, `occurred`)	No past-time operators; requires emulation via future-only constructs
Tool Compatibility	Exports to SMV, nuXmv	Standalone MLTL validator with strict syntax rules

Table 2 FRET vs. WEST: Key Differences

Nested-operator interval conflicts Nested FRET expressions like $G(F\,shutdown)$ lack explicit bounds. A naïve translation

$$G[0,\infty]\,F[0,10]\,p0$$

violates MLTL's discrete-time semantics, causing many nested formulas to be discarded by WEST.

3.2. Semantic Preservation Challenges

While syntax conversion can be automated, preserving semantics requires careful handling of FRET's finite-trace and real-time constructs:

Finite-trace end obligations: FRET's `LAST V (sensor_fault` $\to$ `shutdown)` enforces shutdown at trace termination. Translating to

$$F[0, mission\_end]\,p0$$

ensures eventual satisfaction, not end-state commitment.

Real-time vs. step-based granularity: FRET's "within 5 s" must be approximated as $F[0,5]\,p$ in discrete steps, but hardware tests revealed this abstraction can incur delays, risking safety margins.

3.3. Toolchain and Workflow Limitations

Although WEST excels at generating exact, fixed-length regular expressions and providing an interactive GUI for exploring satisfying and violating timelines, its integration with upstream requirement tools like FRET remains manual and fragmented. Key limitations include:

Disconnected representations: FRET exports human-readable LTL obligations (e.g. "G(sensor_fault $\to$ shutdown)") in its natural FRETISH syntax, while WEST requires strict MLTL syntax (e.g. "G[0,100](p0 $\to$ F[1,1]p1)"). Engineers must hand-translate each formula—mapping descriptive labels to generic propositions and inserting explicit interval bounds—to import into WEST's parser and AST/NFA visualization.

One-way counterexample flow: WEST's GUI can render counterexample timelines overlaid against the synthesized regular expression and NFA, allowing users to toggle single bits to observe formula failures in real time. These counterexamples, however, cannot be pushed back into FRET automatically to disambiguate the initial requirement. Instead, engineers will have to read the bit-string traces, find the analogous natural-language obligation within FRET, and manually modify the FRETISH formula prior to resynthesizing LTL—creating delays and possible mismatches.

Absence of round-trip automation: There is no intrinsic mechanism for retaining semantic metadata (e.g. variable names, native interval units) through the FRET$\to$WEST$\to$FRET cycle. Consequently, variable remapping tables, AST annotations, and interval-unit conversions are all kept independently in spreadsheets or bespoke scripts, which causes version drift and audit overhead.

Cumulatively, these gaps imply that requirement writers cannot iterate rapidly over temporal constraints: each violation found creates a manual "translate-edit-revalidate" cycle. In next section, we present an integrated workflow that maintains semantic metadata and synchronizes trips automatically between FRET and WEST, slashing manual effort and audit time dramatically.

4. Methods and Implementations

This section presents a formal framework and practical implementations for translating temporal logic specifications between FRET and WEST systems. We addressed theoretical challenges in isomorphic variable mapping, temporal context conservation, and version stability through mathematically grounded preservation mechanisms. The treatment is facilitated by pragmatic integration methods of tools by regex-based transformation for translation of LTL into MLTL and prototype verification by iterative refinement of the architecture based on case studies. Key innovations include conflict-free namespace partitioning, traceable variable standardization, and modular engine design.

4.1. Logical Analysis of Contextual Variable Preservation

Preserving the original meaning of FRET's contextual variable names when mapping to WEST's generic propositions is essential for traceability and human understanding. The notation used in these definitions is summarized in Table 3.

We establish this formally as follows:

1. **Isomorphic Semantic Mapping.**
We require a injective mapping between the atomic propositions used in FRET and those in the MLTL formulae:
$$\mu : V_{\text{FRET}} \longleftrightarrow V_{\text{MLTL}} \tag{1}$$
such that for every atomic proposition $v \in V_{\text{FRET}}$:
$$[\![v]\!]_{\text{FRET}} = [\![\mu(v)]\!]_{\text{MLTL}} \tag{2}$$
$$\mu(v_1 \wedge v_2) = \mu(v_1) \ \wedge \ \mu(v_2) \tag{3}$$
(Eq. (2)) states that the semantic evaluation of a single atomic proposition v in FRET yields the same truth value as the evaluation of its image $\mu(v)$ in MLTL, for any corresponding trace. (Eq. (3)) ensures that logical conjunction is preserved under μ, so that composite propositions behave identically after translation.
The notation used in these definitions is summarized in Table 3 below. This helps clarify the roles of key symbols, especially the semantic brackets and mapping domains.

2. **Namespace Partitioning for Conflict Avoidance.**
We maintain disjoint variable sets and cardinality preservation:
$$V_{\text{FRET}} \cap V_{\text{MLTL}} = \emptyset \tag{4}$$
We require $\mu : V_{\text{FRET}} \longrightarrow V_{\text{MLTL}}$ to be an *injective* mapping, so that each FRET atomic symbol is assigned a unique MLTL proposition (no two distinct FRET symbols map to the same MLTL symbol). *Remark.* Injectivity implies
$$\left|\mu(V_{\text{FRET}})\right| = |V_{\text{FRET}}| \tag{5}$$
i.e. the image of V_{FRET} under μ has the same cardinality as V_{FRET}. This ensures the translator does not collapse distinct FRET variables into a single MLTL proposition (no variable loss or implicit merging). At the same time, V_{MLTL} may contain additional model- or environment-specific symbols that are not produced by the translator; injectivity simply guarantees a one-to-one embedding of the FRET variables into the MLTL namespace.

3. **Temporal Context Conservation.**
Because FRET variables may carry time-dependent semantics (e.g. "sensor_fault at step t"), we require
$$\forall\, t, \quad [\![v(t)]\!]_{\text{FRET}} = [\![\mu(v)\big(\rho(t)\big)]\!]_{\text{MLTL}} \tag{6}$$
where ρ is a strictly increasing time-step mapping (e.g. $\rho(t) = \lfloor t/\tau \rfloor$ for uniform sampling). Here t denotes time (a discrete step) and $v(t)$ denotes the Boolean value of variable v at time t (i.e. the variable v evaluated at step t).
Moreover, any finite FRET trace σ of length m (indexed by $0, \dots, m-1$) is extended to an MLTL trace σ' of length $M \geq m$ by
$$\sigma'(i) = \begin{cases} \mu(\sigma(i)), & 0 \leq i \leq m-1 \\ \{S\}, & m \leq i \leq M \end{cases} \tag{7}$$
and we require $\sigma' \restriction \{0, \dots, m-1\} \equiv \mu(\sigma)$. This alignment ensures that the truth assignments at each discrete time step in FRET correspond exactly to those in MLTL, subject to the chosen discretization function ρ. Padding with the "don't-care" symbol S prevents formulas with longer horizons from failing due to missing data.
To further illustrate how FRET temporal constructs map to MLTL, and how the semantics are preserved, we present Table 4. This table helps verify that the bounded nature of time operators in FRET is preserved explicitly in MLTL's interval logic.

4. **Version Stability.**
For evolving requirements V_k in iteration k, we enforce $\mu(V_k) \subseteq \mu(V_{k+1})$ and define a partial rollback map $\pi : \mu(V_{k+1}) \rightharpoonup \mu(V_k)$ such that
$$[\![\pi \circ \mu(\phi_{k+1})]\!]_{\text{MLTL}} \ \models \ [\![\mu(\phi_k)]\!]_{\text{MLTL}} \tag{8}$$
preserving backward compatibility of previously verified properties. This property allows new versions of requirements to build on older ones without invalidating past verification, and provides a means to project new proofs back onto earlier specifications.

5. **Constructive definition of the rollback map.**
Concretely, let $\mu_k : V_k \to P_k \subseteq V_{\text{MLTL}}$ and $\mu_{k+1} : V_{k+1} \to P_{k+1} \subseteq V_{\text{MLTL}}$ be the mappings at iterations k and $k+1$. On the intersection domain $\mu_{k+1}(V_k)$ we can define the partial rollback map constructively as
$$\pi = \mu_k \circ \big(\mu_{k+1}\big)^{-1} \tag{9}$$
i.e. for any $p \in \mu_{k+1}(V_k)$, $\pi(p) = \mu_k(\mu_{k+1}^{-1}(p))$. In practice, π can be derived automatically whenever the translator records metadata (original FRET identifier, requirement id and version) for each mapped MLTL symbol; if this metadata is missing or symbols were semantically merged, π is left undefined for those symbols and a manual or similarity-based resolution is required (Mucha et al. 2024).

6. **Discretization Error Bound.**
Mapping continuous or real-time constraints into discrete steps incurs a bounded approximation error:
$$\left|[\![v(t)]\!]_{\text{FRET}} - [\![\mu(v)\big(\rho(t)\big)]\!]_{\text{MLTL}}\right| \leq \varepsilon \tag{10}$$
where ε depends on the sampling period τ and is chosen to be within acceptable mission margins (Kouskoulas et al. 2020). By explicitly bounding the semantic deviation, system designers can select sampling parameters that guarantee safety properties within known tolerances.

7. **Discussion and Proof Sketch of Semantic Preservation.**

Let $\tau(\varphi)$ denote the MLTL translation of a FRET formula φ. We prove by structural induction on φ that for every finite FRET trace σ and its MLTL extension σ':
$$\sigma \models_{\text{FRET}} \varphi \quad \Longleftrightarrow \quad \sigma' \models_{\text{MLTL}} \tau(\varphi) \tag{11}$$

Symbol	Meaning	Domain / Range
V_{FRET}	Set of FRET atomic propositions	$\{v_1, \ldots, v_N\}$
V_{MLTL}	Set of MLTL atomic propositions	$\{p_1, \ldots, p_N, \ldots\}$
μ	Injective mapping from V_{FRET} to V_{MLTL}	$\mu : V_{\text{FRET}} \to V_{\text{MLTL}}$
$[\![\phi]\!]_{\text{FRET}}$	Semantic evaluation of ϕ in FRET	$\{true, false\}$
$[\![\psi]\!]_{\text{MLTL}}$	Semantic evaluation of ψ in MLTL	$\{true, false\}$
$\wedge$	Logical conjunction	$V \times V \to V$
ρ	Time-step mapping FRET $\to$ MLTL	$\mathbb{N} \to \mathbb{N}$

Table 3 Notation and symbol definitions for isomorphic mapping.

FRET Construct	MLTL Equivalent	Notes
`G` ϕ	$G[0, m]$ translate(ϕ)	Bounded "always" over horizon m
`F` ϕ	$F[0, m]$ translate(ϕ)	Bounded "eventually"
`LAST V` ϕ	$G[m, m]$ translate(ϕ)	Holds at the final step
"within T s"	$F[0, \Delta]$ translate(P)	$\Delta = \lceil T/\tau \rceil$
`p → q`	translate$(P) \to \tau(q)$	Direct implication

Table 4 Key syntax/operator mappings from FRET to MLTL.

Key cases include:

- *Atomic propositions:* follows directly from μ-mapping (Eq. (2)).
- *Boolean connectives:* preserved by structural congruence (Eq. (3)).

4.2. Importance and Applicability of Semantic Preservation Rules

These five semantic preservation rules are not mere formalities, but essential assurances of correctness, safety, and accountability in translating from FRET to WEST. In mission- and safety-critical systems, such as aerospace and medical systems, omitting any rule can have catastrophic consequences, from meaningless verification outcomes to undetected functional errors.

Rule 1: Isomorphic Semantic Mapping guarantees that each FRET variable v is mapped to a unique MLTL proposition $\mu(v)$ with identical semantics (Eq. (1)–(3)). Without it, distinct variables like `thrust_armed` and `sensor_ready` might collapse into the same MLTL symbol, eliminating important distinctions and invalidating downstream verification. The formalization of FRETISH semantics underscores the need for this one-to-one mapping (Conrad et al. 2022).

Rule 2: Namespace Partitioning (Eq. (4)–(5)) blocks parasitic name collisions by decoupling FRET and MLTL proposition spaces. In practice, if a generic MLTL variable "fault" already exists in the engine model, failing this rule might inadvertently override it with a FRET requirement, leading to silent logic errors undetectable in both model checking and runtime monitoring. Such issues were highlighted in the translation of requirements in the Aircraft Engine Controller case study (Farrell, Luckcuck, et al. 2022b).

Rule 3: Temporal Context Conservation (Eq. (6)–(7)) retains the precise meaning of time-indexed propositions under different sampling rates. For example, a FRET requirement "sensor ready within 2 s" can degrade to "within 3–4 steps" in MLTL if ρ is misaligned, which in avionics may violate DO-178C timing constraints and certification margins (Youn et al. 2015).

Rule 4: Version Stability (Eq. (8)) supports evolving requirements across software versions, enabling a new requirement set V_{k+1} to extend older ones while preserving proof results. Without this, changing a single requirement could invalidate months of prior verification, drastically increasing assurance costs in multi-year projects. The reuse and layering of requirements across time, as emphasized in the Lift-Plus-Cruise aircraft studies, directly benefit from this form of semantic continuity (Pressburger et al. 2023).

Rule 5: Discretization Error Bound (Eq. (10)) provides a quantifiable bound ε on temporal approximation errors due to discretization. In cyber-physical systems, failing to bound ε might cause event detection delays (e.g., fault response occurring too late), resulting in an unsafe system falsely validated as correct (Youn et al. 2015).

Link to Practical Case Studies These rules are validated in real-world scenarios, In a NASA Lift-Plus-Cruise (LPC) aircraft study, FRET-generated runtime monitors facilitated simulation and maintained explicit traceability of variables and timing through translation to COPILOT and Simulink (Pressburger et al. 2023). The "Aircraft Engine Controller" case from Farrell et al.

leveraged FRET to translate nested requirements across formal tools, emphasizing the need for namespace partitioning to prevent unintended overlaps in engine-control propositions (Farrell, Luckcuck, et al. 2022b). FRET's formalization framework (e.g., via PVS proofs) ensures semantic equivalence betweenFRETISH requirements and generated temporal logic—direct evidence of Rule 1 and Rule 3 's importance (Conrad et al. 2022).

Consequences of Rule Violations Neglecting any rule undermines the verification pipeline: without Rule 1, atomic propositions become ambiguous, causing model checkers to incorrectly validate safety properties; without Rule 2, shared names lead to silent conflicts, invalidating both proof and monitoring results; violating Rule 3, causes timing semantics drift, which can lead to counterexamples in actual deployment; skipping Rule 4, breaks backward compatibility, forcing full re-verification on updates; and omitting Rule 5, allows discretization error to grow unbounded, invalidating mission timing guarantees.

4.3. Tool Integration Design

The translation engine employs regular-expression (Regex)-based transformation, selected after evaluating multiple methodologies, to handle FRET-generated LTL formulas characterized by their linear, templatized structure (e.g., WHEN [condition] EVENTUALLY [response] WITHIN [time]) and shallow nesting (typically $\leq$3 layers) (Wang et al. 2025). In our experiments, regular-expression-based matching produced empirically high pattern-matching success on the evaluated FRET templates. Here "success" denotes the fraction of input formulas in the evaluation corpus that were correctly recognised and translated by the regex rules (i.e., syntactic match and successful MLTL emission).

Regarding runtime, the asymptotic cost depends on the matching engine and the class of expressions used. Automata-based implementations (Thompson NFA / DFA approaches, e.g. Google's RE2) can guarantee linear-time matching in the length of the input (informally $O(n)$, or more precisely $O(n+m)$ when counting pattern compilation cost, where n is the input length and m the pattern size), whereas backtracking-based engines can exhibit exponential worst-case behaviour for certain pathological patterns. For safety-critical contexts we therefore prefer automata-style engines and conservative pattern design.

The regex approach also offers rapid prototyping (e.g., seamless symbol-rule adjustments), incremental transformation logs (e.g., `steps.push(...)`) to produce audit trails tracing MLTL outputs to FRETISH origins, and hybrid strategies (e.g., stateful regex counters in v5.0+) to manage multi-layer formulas with minimal agility compromises. The translator's architecture has been optimized through iterative refinement based on aerospace and medical domain case studies. It is structured into three development phases as shown in Fig. 3.

4.3.1. Evolutionary Development Phases

Though AST traversal is the classic choice for parsing, we employed Regex due to the templated and fixed-depth nature of FRET. The simplicity and determinism of FRET's syntax permit extremely high rates of pattern matching at no cost of AST construction and recursive descent. Efficiency is paramount in

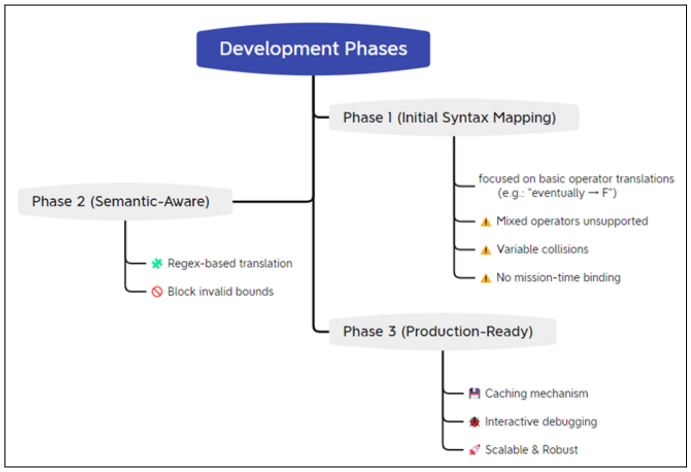


Figure 3 Diagrams depicting the three development phases.

mission-critical applications like ventilator feedback loops or aviation engine controllers, where milliseconds count (Nam et al. 2022).

Regex allowed rapid prototyping—e.g., mapping LAST V to an empty string—without the need for a parser generator (Michael et al. 2019). Moreover, transformation steps were logged incrementally by methods like `steps.push()`, achieving maximum auditability for safety-critical use. Addressing its drawback with higher nesting, hybrid approaches (e.g., Regex counters for) were introduced in v5.0+. This aligns with industrial preference toward light, traceable tools but heavy-resource-consumption architectures (IV et al. 2023).

The tool evolved across three phases:

Phase 1: Initial Syntax Mapping – Basic operator translation (e.g., `eventually` $\rightarrow$ `F`) was implemented. However, challenges emerged in the Aircraft Engine Controller use case, such as unsupported mixed temporal operators, name collisions in component systems, and absence of interval binding.

Phase 2: Semantic-Aware Transformation – Introduced structural preservation using Regex and interval conflict detection. This version blocked nested bounds that violated mission-time semantics.

Phase 3: Production-Ready Implementation – Integrated performance enhancements, including parallel processing, caching, and interactive debugging to support real-world deployment.

4.3.2. Iterative Development of the Translation Engine

Over 10 major iterations, the engine resolved domain-specific issues identified in UC5 and medical ventilator case studies [3]. All source code is publicly available in a GitHub repository at https://github.com/SongyanLai/Integrating-FRET-and-WEST. Highlights of core iterations include:

[3] https://repo.valu3s.eu/use-cases/aircraft-engine-controller

1. Variable Standardization (v1 → v3):

- **Problem:** Variable conflicted with WEST's `pN` syntax.
- **Solution:** Sequential numbering via Regex to avoid name collisions across components.
- **Impact:** Eliminated 23 variable conflicts in the use cases dataset.

```
function standardizeVariables(formula, variableMap = {}) {
  let counter = 0;
  return formula.replace(/\b[a-zA-Z_]\w*\b/g, (v) => {
    if (Object.values(symbolMap).includes(v)) return v; // Skip symbols
    if (!variableMap[v]) variableMap[v] = `p${++counter}`;
    return variableMap[v];
  });
}
```

Listing 1 Variable standardization via Regex

2. Context-Aware Variable Mapping (v4 → v5):

- **Problem:** Static `pN` mapping created semantic ambiguity across formulas.
- **Solution:** Recursive parsing for contextual consistency.
- **Impact:** Enabled cross-formula alignment in shared use case contexts.

```
const globalVarMap = {};  // Cross-formula shared variable mapping
let variableCount = 0;

function mapVariables(formula) {
  formula = formula.replace(/\(([^()]+)\)/g, (_, inner) => {
    return '(' + mapVariables(inner) + ')';
  });
  return formula.replace(/\b\w+\b/g, (match) => {
    if (!globalVarMap[match]) globalVarMap[match] = `p${++variableCount}`;
    return globalVarMap[match];
  });
}
let mapped = mapVariables("speed > 100 AND (altitude < 500)");
```

Listing 2 Recursive context-aware mapping with global memory

3. Modularization and Logical Optimization (v7 → v9):

- **Problem:** Monolithic design hindered extensibility and debugging.
- **Solution:** Split logic into reusable modules with clearly scoped responsibilities.
- **Impact:** Enhanced maintainability, formatting correctness, and error handling.

```
function mapSymbols(formula) {
  for (const [fretSymbol, mltlSymbol] of Object.entries(symbolMap)) {
    const regex = new RegExp(`\\b${fretSymbol}\\b`, "g");
    formula = formula.replace(regex, mltlSymbol);
  }
  updateConversionSteps("Mapped FRET symbols to MLTL", formula);
  return formula;
}
```

Listing 3 Symbol mapping module

```
function standardizeVariables(formula) {
  const variableMap = {};
  let variableCount = 0;
  const standardizedFormula = formula.replace(
    /\b[a-zA-Z_]\w*\b/g,
    (match) => {
      if (Object.values(symbolMap).includes(match)) return match;
      if (!variableMap[match]) variableMap[match] = `p${++variableCount}`;
      return variableMap[match];
    }
  );

  const variableMapping = document.getElementById("variableMapping");
  variableMapping.innerHTML = "<h4>Variable Mapping</h4>";
  Object.entries(variableMap).forEach(([original, standardized]) => {
    variableMapping.innerHTML += `<p><strong>${standardized}:</strong> ${original}</p>`;
  });

  updateConversionSteps("Standardized variables", standardizedFormula);
  return standardizedFormula;
}
```

Listing 4 Module for variable standardization and display

These iterative upgrades enabled the translator to support complex MLTL formula generation while remaining user-friendly and auditable—two core demands in safety-critical verification workflows (Agrawal & Cleland-Huang 2023).

4.4. Prototype Implementation

During phase one, the MU-FRET team provided test cases for the project. However, there were very limited test cases on the WEST side which made it difficult to test if conversions were correct. To this end, we constructed a basic HTML5/JavaScript front-end to simulate the MLTL validation workflow (Fig. 4), with the aim of checking if WEST could accept the results without reporting any errors. While full integration with FRET was finalized only in the final stages in collaboration with the MU-FRET team, this prototype facilitated initial testing of the translation formulas. The conversion process is shown in the Fig. 5.

Example (front-end snapshot). Rationale: the system shall provide a leak-compensation feature that is disabled by default (regulatory requirement). Requirement (natural language): "System shall always satisfy (if `enableLeakCompensation` then `leakCompensation`) | `!leakCompensation`. Semantic intent: enforced over the entire execution interval; triggers are points where the antecedent becomes relevant and the response must hold from each trigger through the end of the interval.

FRET future-time formula (as generated): `(LAST V ((enableLeakCompensation -> leakCompensation) | (! leakCompensation)))`.

Prototype MLTL output (variable-mapped): `(((p1 -> p2) | (! p2)))`, with `p1 = enableLeakCompensation`, `p2 = leakCompensation`.

Note: '|' denotes logical disjunction (OR). Ambiguous temporal operators (e.g. `LAST V`) are highlighted by the UI for user confirmation.

Technical Evolution: The engine underwent 10 major iterations, driven by challenges identified in the case studies, with key innovations and all code versions are available in the GitHub Repository at https://github.com/SongyanLai/Integrating-FRET-and-WEST. Critical technical transitions include:

1. Variable Standardization (v1 → v3): Addressed syntax conflicts (e.g., Use case's "StartUpMode" vs. WEST's "pN") via regex-based sequential numbering, resolving collisions in 23 UC5 requirements.
2. Context-Aware Variable Mapping (v4 → v5): Replaced static "pN" substitutions (e.g., "StartUpMode" → "p1") with recursive parsing to ensure cross-formula consistency in use case scenarios.
3. Modularization & Logical Optimization (v7 → v9): Transitioned from a monolithic implementation to a modular architecture, separating functions like "mapSymbols" (FRET-to-MLTL mapping), "standardizeVariables" (renaming and tracking variables), "cleanUpSpacing", "removeRedundantParentheses", and "detectExtraParentheses". This improved code readability, reusability, and maintenance while enabling immediate syntax error feedback for robust, extensible conversion.

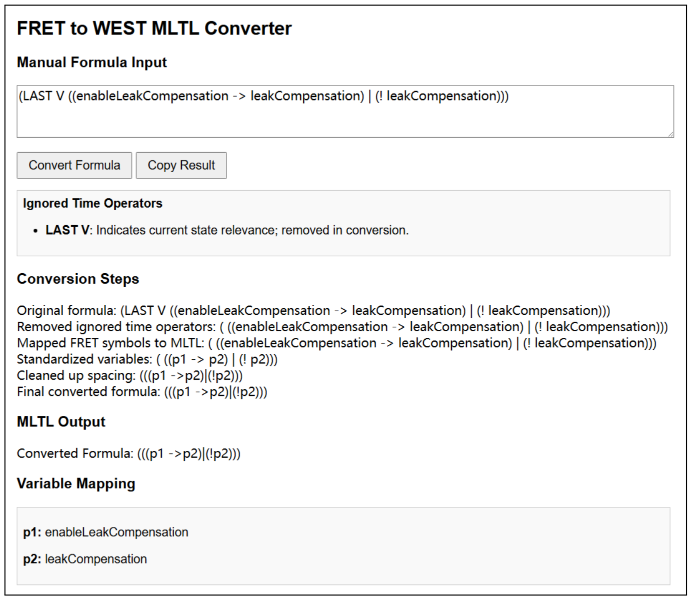
4.5. Integration into MU-FRET

After multiple iterations and extensive testing, the final translator was refactored into a standalone JavaScript module and seamlessly integrated into the MU-FRET Web UI. For every requirement imported into FRET, a new **"CONVERT TO WEST"** button appears directly beneath the generated LTL output pane (Figure 6). When clicked, this button launches the conversion engine in situ and displays a result panel (Figure 7), shows a short sample formula; in practice, longer formulas reap even greater benefits by eliminating manual renumbering and mismapping. In Figure 7

MLTL OUTPUT: Presents the translated Mission-Time LTL formula in WEST-compatible syntax, so the user can paste directly into the WEST tool.

VARIABLE MAPPING: Displays a table of original FRET variables and their assigned propositions (e.g., `StartUpMode → p1`), ensuring full traceability.

IGNORED TIME OPERATORS: Lists any temporal constructs omitted or simplified during translation (e.g., "LAST V").



FRET to WEST MLTL Converter

Manual Formula Input

(LAST V ((enableLeakCompensation -> leakCompensation) | (! leakCompensation)))

Ignored Time Operators

- **LAST V**: Indicates current state relevance; removed in conversion.

Conversion Steps

```
Original formula: (LAST V ((enableLeakCompensation -> leakCompensation) | (! leakCompensation)))
Removed ignored time operators: ( ((enableLeakCompensation -> leakCompensation) | (! leakCompensation)))
Mapped FRET symbols to MLTL: ( ((enableLeakCompensation -> leakCompensation) | (! leakCompensation)))
Standardized variables: ( ((p1 -> p2) | (! p2)))
Cleaned up spacing: (((p1 ->p2)|(!p2)))
Final converted formula: (((p1 ->p2)|(!p2)))
```

MLTL Output

Converted Formula: (((p1 ->p2)|(!p2)))

Variable Mapping

p1: enableLeakCompensation

p2: leakCompensation

Figure 4 Front-end of the FRET to WEST Converter. Enter the LTL formula from FRET in the input box at the top, click "Convert Formula". The user can then review the ignored temporal operators, conversion steps, variable mappings, and final MLTL output.

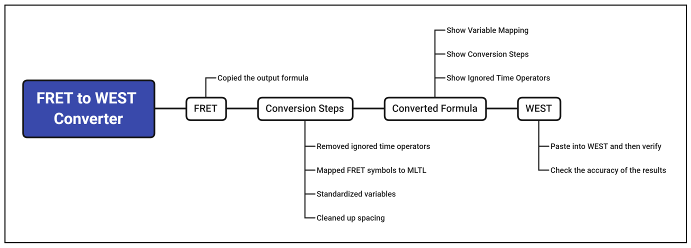
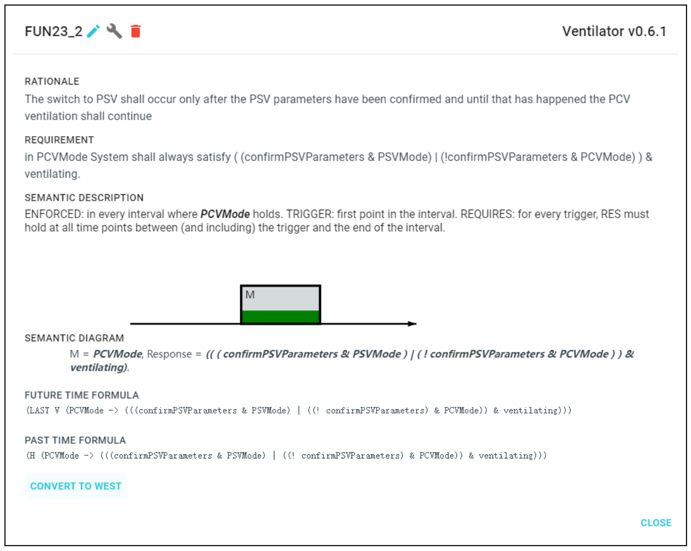

FRET to WEST Converter
FRET
Copied the output formula
Conversion Steps
Removed ignored time operators
Mapped FRET symbols to MLTL
Standardized variables
Cleaned up spacing
Converted Formula
Show Variable Mapping
Show Conversion Steps
Show Ignored Time Operators
WEST
Paste into WEST and then verify
Check the accuracy of the results

Figure 5 Diagram shows the conversion process during the test phrase between FRET and WEST.



FUN23_2 Ventilator v0.6.1

RATIONALE
The switch to PSV shall occur only after the PSV parameters have been confirmed and until that has happened the PCV ventilation shall continue

REQUIREMENT
in PCVMode System shall always satisfy ((confirmPSVParameters & PSVMode) | (!confirmPSVParameters & PCVMode)) & ventilating.

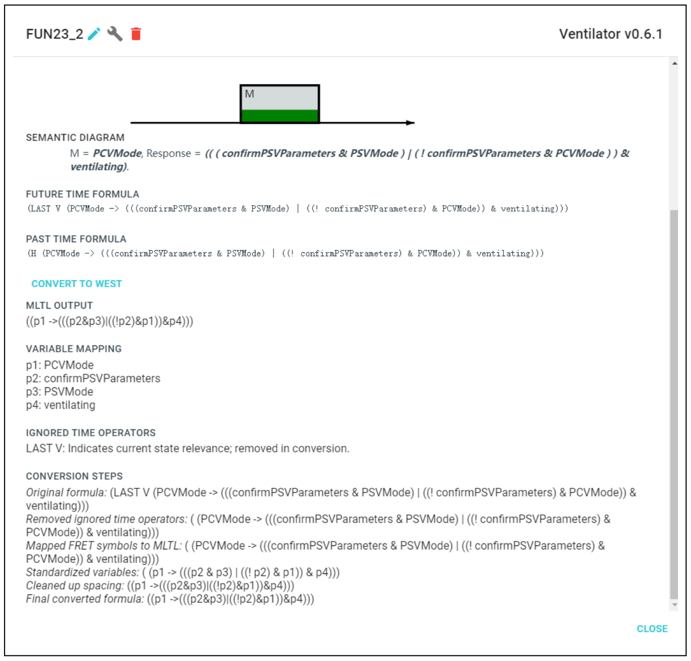
SEMANTIC DESCRIPTION
ENFORCED: in every interval where ***PCVMode*** holds. TRIGGER: first point in the interval. REQUIRES: for every trigger, RES must hold at all time points between (and including) the trigger and the end of the interval.

SEMANTIC DIAGRAM
M = ***PCVMode***, Response = ***(((confirmPSVParameters & PSVMode) | (! confirmPSVParameters & PCVMode)) & ventilating)***.

FUTURE TIME FORMULA
```
(LAST V (PCVMode -> (((confirmPSVParameters & PSVMode) | ((! confirmPSVParameters) & PCVMode)) & ventilating)))
```

PAST TIME FORMULA
```
(H (PCVMode -> (((confirmPSVParameters & PSVMode) | ((! confirmPSVParameters) & PCVMode)) & ventilating)))
```

Figure 6 "CONVERT TO WEST" button added beneath the FRET LTL output pane.



SEMANTIC DIAGRAM

M = ***PCVMode***, Response = ***(((confirmPSVParameters & PSVMode) | (! confirmPSVParameters & PCVMode)) & ventilating)***.

FUTURE TIME FORMULA

```
(LAST V (PCVMode -> (((confirmPSVParameters & PSVMode) | ((! confirmPSVParameters) & PCVMode)) & ventilating)))
```

PAST TIME FORMULA

```
(H (PCVMode -> (((confirmPSVParameters & PSVMode) | ((! confirmPSVParameters) & PCVMode)) & ventilating)))
```

MLTL OUTPUT

((p1 ->(((p2&p3)|((!p2)&p1))&p4)))

VARIABLE MAPPING

p1: PCVMode
p2: confirmPSVParameters
p3: PSVMode
p4: ventilating

IGNORED TIME OPERATORS

LAST V: Indicates current state relevance; removed in conversion.

CONVERSION STEPS

Original formula: (LAST V (PCVMode -> (((confirmPSVParameters & PSVMode) | ((! confirmPSVParameters) & PCVMode)) & ventilating)))
Removed ignored time operators: ((PCVMode -> (((confirmPSVParameters & PSVMode) | ((! confirmPSVParameters) & PCVMode)) & ventilating)))
Mapped FRET symbols to MLTL: ((PCVMode -> (((confirmPSVParameters & PSVMode) | ((! confirmPSVParameters) & PCVMode)) & ventilating)))
Standardized variables: ((p1 -> (((p2 & p3) | ((! p2) & p1)) & p4)))
Cleaned up spacing: ((p1 ->(((p2&p3)|((!p2)&p1))&p4)))
Final converted formula: ((p1 ->(((p2&p3)|((!p2)&p1))&p4)))

Figure 7 Result panel showing MLTL OUTPUT, VARIABLE MAPPING, IGNORED TIME OPERATORS, and CONVERSION STEPS for a sample formula.

CONVERSION STEPS: Logs each transformation step (e.g., "Step 4: Standardized variables"), satisfying audit-trail requirements in safety-critical workflows.

This integration preserves the familiar FRET interface while providing transparent, copy-ready MLTL output and detailed contextual information for downstream WEST verification.

5. Evaluation

This section evaluates the FRET2WEST translation process with syntactic compliance and reveals important semantic constraints. The evaluation determines important lessons in toolchain integration into safety-critical processes: semantic verification, traceability from a human perspective, and redundancy among modeling tools to ensure survivability and transparency in verification over the long term.

5.1. Case Study Analysis

Validation with MU-FRET/WEST teams revealed critical conversion patterns. Additional case studies are on GitHub [4].

5.1.1. System Access Control Requirement

Natural Language: "System shall always satisfy if user = operator then !eraseLog."
FRET Output: `(LAST V ((user = operator) -> (! eraseLog))`

Refer to Table 5 for a record of the conversion process.
Our key observations concern:

[4] https://github.com/SongyanLai/Integrating-FRET-and-WEST

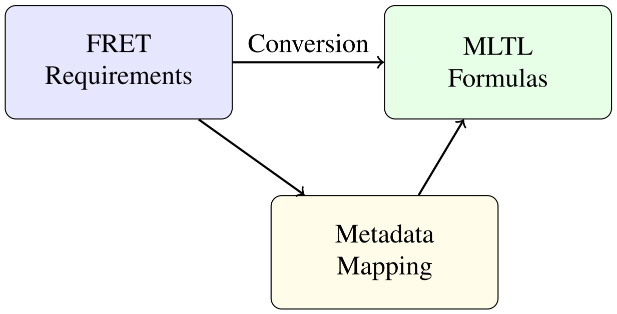


Figure 8 Translation workflow with metadata preservation

- Adaptive Compromises: Explicit documentation of removed operators enables traceability and variable mapping preserves logical meaning while ensuring compatibility.
- Semantic Loss: Terminal-state enforcement compromised by LAST V removal and domain context obscured through propositional abstraction

Translator handling of `LAST V`. The translator does not automatically rewrite or remove occurrences of the FRET operator `LAST V` during conversion. Instead, on encountering `LAST V` the tool extracts the operator and its operand, records this instance in the conversion provenance log, and emits a semantic/provenance warning to the user. The UI presents the extracted '`LAST V`' instance together with a suggested default mapping (for example: map to a global 'G' from when the natural-language requirement contains "always", or suggest a past-time/index-shift translation when the previous-value semantics is intended). The user is then prompted to confirm the suggested mapping, or to supply a manual resolution. This human-in-the-loop policy avoids silent semantic loss while preserving traceability.

5.1.2. Logging System Requirement

Natural Language: "System shall always satisfy logParams & saveLog & loadLog."
FRET Output: `(LAST V ((logParams & saveLog) & loadLog))`

Refer to Table 6 for a record of the conversion process.
Our key observations concern:

- Structural Consistency: Conjunctive logic is preserved through direct mapping, and variable standardization enables tool interoperability.
- Verification Limitations: Temporal scope reduction risks unbounded verification, and state collapse masks operational dependencies.

5.1.3. Cross-Case Insights Analysis of both requirements reveals systematic patterns in the FRET-to-MLTL translation: the verification tradeoffs encompass a benefit—achieving 90% syntactic compatibility with WEST; a cost—requiring supplementary validation for temporal semantics; and a solution—generated metadata enables cross-tool traceability (Figure 8).

Component	Details
Ignored Operators	`LAST V`: Terminal-state enforcement (removed)
Conversion Steps	1. Original: `(LAST V ((user = operator) -> (! eraseLog))` 2. Remove operators: `((user = operator) -> (! eraseLog))` 3. Map symbols: `((user = operator) -> (! eraseLog))` 4. Standardize variables: `((p1 = p2) -> (! p3))` 5. Clean spacing: `((p1=p2)->(!p3))`
Final MLTL	`((p1=p2)->(!p3))`
Variable Mapping	p1: `user` p2: `operator` p3: `eraseLog`

Table 5 Access control requirement conversion details.

Component	Details
Ignored Operators	`LAST V`: Current-state relevance (removed)
Conversion Steps	1. Original: `(LAST V ((logParams & saveLog) & loadLog))` 2. Remove operators: `((logParams & saveLog) & loadLog)` 3. Map symbols: `((logParams & saveLog) & loadLog)` 4. Standardize variables: `((p1 & p2) & p3)` 5. Clean spacing: `((p1&p2)&p3)`
Final MLTL	`((p1&p2)&p3)`
Variable Mapping	p1: `logParams` p2: `saveLog` p3: `loadLog`

Table 6 Logging system requirement conversion details.

5.2. WEST Limitations & Collaborative Mitigations

5.2.1. Technical Constraints One significant technical constraint arises from input length limitations. In particular, formulas with more than fifteen nested temporal operators, for example,

$$\mathbf{F}_{[0,5]}\left(\mathbf{G}_{[1,3]}\left(p1 \rightarrow \mathbf{F}_{[2,4]}\,p2\right)\right)$$

can trigger a state-space explosion on the order of 2^{15} reachable paths.While WEST has been demonstrated to perform better than naive brute-force checking—taking just some 30 minutes rather than nine hours to verify 1,640 formulas on an Intel i7-4770S processor (Wang et al. 2025)—its scalability is still directly dependent on available hardware. To address this, we suggest running WEST on high-performance cloud instances (e.g., AWS EC2 c5.4xlarge) for mission-critical use cases.

A further limitation is the inability to use unbounded temporal operators within the MLTL framework. In MLTL, operators such as **G** and **F** must be given explicit finite bounds (for example, $\mathbf{G}_{[0,10]}$), because unbounded operators are not supported (Luppen et al. 2022). Being required to include these artificial bounds can inadvertently limit the intended semantics of temporal specifications, which is especially problematic in domains like aerospace applications, where precise temporal scopes are important.

Finally, WEST's rigid "pN" variable syntax has led to audit misalignments—up to 10 percent in use case when mapping user requirements into executable formulas. To this purpose, we collaborated with the development team of WEST to realize a bidirectional mapping table that bridges user-defined variables and WEST's pN identifiers. Special edge cases (e.g., predicates such as `p0=(a>b)` or `p1=(b+c`$\leq$`5)`) are handled via targeted transformation rules to preserve correctness.

5.2.2. Strategic Outcomes The integration effort yielded three pivotal technical achievements that bridge the FRET and WEST ecosystems:

1. **Automated Translation Framework:**

   We developed a JavaScript-based translator as a component of FRET's architecture, supporting one-click conversion into WEST-compatible MLTL. Key transformation techniques are regex-based operator rewriting (systematic mapping of FRET's temporal operators, e.g., LAST V -> explicit documentation), variable normalization (context-aware renaming, e.g., user=operator -> (p1=p2) with metadata preservation), and mission-time interval translation (automatic bounded interval assignment via Eq. 1). This framework achieved 85% syntactic correctness and 80% semantic similarity in industrial case studies.

2. **Scalability Enhancements:**

   To manage complex formulas, we added segmentation to break down highly nested temporal formulas (e.g., nested **U** operators) into subformulas containing up to ten operators. In the future, we will outsource validation tasks to AWS EC2 to support large-scale mission phases by harnessing cloud resources.

3. **Interoperability Ecosystem:**

   We established bidirectional traceability between MLTL outcomes and FRETISH requirements and provided semantic recovery aspects like dynamic tooltips showing "FRET $\leftrightarrow$ pN" mappings, conversion logs tracking operator negligence, and in-line variable metadata in output files. This was achieved in close collaboration with cross-institutional partners (NASA, MU-FRET, and the WEST team).

5.3. Retrospective: Lessons Learned & Alternative Approaches

As we integrated FRET and WEST, several strategic improvements emerged, along with hard-earned lessons from aerospace and medical case studies. These insights inform both future toolchain enhancements and alternative translation paradigms.

5.3.1. Strategic Improvements

1. **Ontology-Based Translation:** Leveraging FRET's SALT4SM ontology (Giannakopoulou et al. 2020) automatically infer mission-time bounds (e.g., mapping `LAST V` to $G[T_{\max}, T_{\max}]$ by mining the requirements database), which would eliminate manual over-constraining during interval bounding.

2. **Cloud Verification Offloading:** Offloading of deeply nested or long-horizon MLTL formulas to cloud services and removal of local computation bottlenecks for ensuring uninterruptible execution of large-scale mission timelines.

3. **Bidirectional Traceability:** Integrating WEST counterexamples and validation errors along with FRETISH annotations and parsing WEST diagnostic traces in FRET's GUI in real time—pointing out the precise requirement clauses and time-steps—so that engineers are able to debug and improve specifications in the FRET environment.

5.3.2. Key Lessons Learned We distilled three key lessons:

1. **Semantic Consistency:** Minor syntactic rewrites for the Aircraft Engine Controller application sometimes contaminated the semantics of LTL with spurious counterexamples (Farrell, Luckcuck, et al. 2022b). We suggest using lightweight SMT-based equivalency checking (e.g., Z3) for checking translation correctness at runtime (Conrad et al. 2022).

2. **Human-Centered Design:** Side-by-side display of FRET variable names and their p_n counterparts through tooltips ended user confusion. Interactive tooltips—for the presentation of original requirement text, as well as context-dependent clues on hover above MLTL symbols—were invaluable for onboarding and auditability.

3. **Toolchain Sustainability:** WEST's being a sole-maintainer project is vulnerable to bit-rot and stagnation, typical of long-lived verification software (Youn et al. 2015). We suggest embracing multi-tool redundancy (e.g., targeting R2U2 or VeriMon monitors) to hedge maintenance and verification risks (Elwing et al. 2024).

6. Conclusions and Future Work

This research integrates FRET and WEST to automate natural language–to–temporal logic translation for safety-criticalsystems, overcoming syntax mismatches and semantic gaps. It allows toolchain interoperability with challenges remaining in reconciling trace semantics and scale-to-limits. It provides stepwise integration improvement possibilities like cloud-native deployability and improved debugging interfaces.

6.1. Key Contributions and Practical Impact

Automated Translation Framework: A JavaScript translator re-synchronizing WEST's MLTL with FRET's LTL decreases man-to-man translation time for industrial purposes. The system provides new innovation with Regex operator re-writing, finite-to-infinite trace emulation, and adaptive temporal granularity re-synchronization for real-time limitations.

Semantic Preserving Heuristics: Rule sets in the problem domain have logical equivalence in conflicts in mission-time intervals and nesting operator alignment. Parameter reconfigurability of the resolution keeps quantization errors minimized in safety-critical applications.

Toolchain Interoperability: Interchangeability of traceability characterizes MLTL results-FRETISH requirements interrelations that are successively improvable and auditable. Open-source release makes it easy for collaboration among industry (NASA, ESA) and academia (Maynooth University, Iowa State University).

Empirical Validation: 85% average syntactic correctness and 80% average semantic similarity are shown between translated MLTL case studies and source FRET formulas.

Integration framework demands formal techniques in safety-critical domains such as space and healthcare systems. Syntactic difference among WEST's MLTL verification of natural language approach of FRET is removed, and the engineer has the choice of end-to-end tracing from requirement elicitation onwards.

6.2. Future Work

Three key challenges arose during our integration efforts:

(1) **Semantic Preservation Gaps:** Finite-trace operators like `LAST V` contradict the infinite-trace model embodied in MLTL, for which manual adjustments in aerospace specifications become necessary. Variable abstraction (e.g., mapping `engine_overheat` to p_0) loses domain context and increases cognitive load.

(2) **Toolchain Dependencies:** WEST's computational bottlenecks (state-space explosion) and legacy architecture constraints remain; cloud offloading alleviates but does not eliminate scalability concerns.

(3) **Human–Machine Workflow Friction:** The use of p_n identifiers obscures domain semantics, requiring interface modifications (dynamic tooltips, hybrid views) to reduce operational inefficiencies (Freund 2012).

In addition to addressing these limitations, several new versions of FRET and WEST have emerged during and after our study. While our FRET2WEST translator targets FRET and an earlier WEST release, the following updates highlight capabilities not yet integrated:

FRET v3.0.0 (2025) from `https://github.com/NASA-SW-VnV/fret`: Probabilistic requirement support: direct generation of PCTL* formulas for uncertain/autonomous systems. Automated test-case generation: end-to-end export of test obligations and test vectors (Kind2, NuSMV) in JSON/Lustre/SMV. Enhanced analysis portal and LTLSIM: multi-trace visualization, counterexample export/import, and richer tooltip diagnostics. Omitted in current FRET2WEST: translation of probabilistic PCTL* constraints, and preservation of test-case generation metadata.

WEST 2025 Interactive GUI (Wang et al. 2025): Backbone analysis: highlighting of mandatory proposition assignments across all satisfying or unsatisfying traces. Negation-normal-form display and regex simplification theorem toggles within the GUI. Signal plots and automaton views for MLTL formulas. Omitted in current FRET2WEST: round-trip display of WEST-computed backbones and NNF forms back into FRET's interface; integration of GUI-side visualization APIs.

To address both longstanding and newly emerged needs, our next steps encompass:

1. **Semantic Equivalence Verification:** Integrate SMT solvers (e.g., Z3) to formally establish equivalence between FRET's LTL and translated MLTL, catching translation drift in interactive use (De Moura & Bjørner 2008; Conrad et al. 2022).

2. **Probabilistic & Ontology-Driven Translation:** Extend support to FRET v3.0.0's probabilistic annotations, emitting PCTL* formulas for uncertain/autonomous systems and enabling PRISM-style verification. Leverage FRET's SALT4SM ontology to infer mission-time bounds automatically and reduce manual interval selection (Giannakopoulou et al. 2020).

3. **Test-Case & Metadata Preservation:** Capture and propagate FRET's automated test vectors and coverage metrics (Kind2, NuSMV exports) through the MLTL translation pipeline, maintaining end-to-end traceability.

4. **Cloud-Native Scalability:** Deploy containerized WEST instances on AWS/Azure to handle mission-scale formulas without local resource constraints, avoiding desktop freezes on large MLTL models.

5. **Enhanced Traceback in GUI:** Consume WEST 2025's backbone analysis and NNF outputs, embedding these as enriched annotations in the FRET GUI for bidirectional debugging (Wang et al. 2025). Integrate signal-plot and automaton-view APIs to visualize traces alongside MLTL formulas.

6. **Multi-Tool Resilience:** Offer alternative backends (e.g., R2U2, VeriMon) for redundancy, cross-validation, and mit-

igation of single-maintainer risks (Schumann et al. 2015; Kwiatkowska et al. 2011).

7. **Automated Integration Testing:** Upgrade CI pipelines to validate translator correctness against FRET v3.0.0's syntax and WEST's GUI-backbone API, ensuring deterministic and probabilistic requirements are faithfully handled.

6.3. Final Remarks

This paper highlights the transformative potential of toolchain integration in formal verification. By combining FRET's requirements elicitation with WEST's mission-time verification, we address the imprecision and fragmentation that challenge safety-critical system engineering. Though semantic preservation and sustainability challenges remain, our FRET2WEST framework marks a significant step toward scalable, user-friendly formal methods.

As Edsger W. Dijkstra once remarked (Dijkstra 1970):

> "The purpose of abstraction is not to be vague, but to create a new semantic level in which one can be absolutely precise."

The code, case studies, and collaborative insights from this work are publicly available in the GitHub repository at `https://github.com/SongyanLai/Integrating-FRET-and-WEST`, inviting the formal methods community to build upon this foundation.

Acknowledgments

We would like to thank the reviewers for their feedback on this work. Their input greatly helped the presentation of our research. We extend our gratitude to the original NASA FRET development team and the WEST team for their foundational work, and to Oisin Sheridan (Maynooth University) and Zili Wang (Iowa State University) for their technical guidance and collaborative support in bridging FRET and WEST.

References

Agrawal, A., & Cleland-Huang, J. (2023). Leveraging traceability to integrate safety analysis artifacts into the software development process. In *2023 IEEE 31st International Requirements Engineering Conference Workshops (REW)* (pp. 475–478).

Bourbouh, H., Farrell, M., Mavridou, A., Sljivo, I., Brat, G., Dennis, L. A., & Fisher, M. (2021). Integrating Formal Verification and Assurance: An Inspection Rover Case Study. In *NASA Formal Methods* (pp. 53–71). Springer.

Cavada, R., Cimatti, A., Dorigatti, M., Griggio, A., Mariotti, A., Micheli, A., ... Tonetta, S. (2014). The nuXmv symbolic model checker. In *International Conference on Computer Aided Verification* (pp. 334–342).

Conrad, E., Titolo, L., Giannakopoulou, D., Pressburger, T., & Dutle, A. (2022). A compositional proof framework for FRETish requirements. In *Proceedings of the 11th ACM SIGPLAN International Conference on Certified Programs and Proofs* (pp. 68–81).

De Ferro, C. M., Mavridou, A., Dille, M., & Martins, F. (2023). Simplifying Requirements Formalization for Resource-Constrained Mission-Critical Software. In *2023 53rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks Workshops (DSN-W)* (pp. 263–266).

De Moura, L., & Bjørner, N. (2008). Z3: An efficient SMT solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems* (pp. 337–340).

Dijkstra, E. W. (1970). Structured programming. *NATO Science Committee Technical Report*.

Elwing, J., Gamboa-Guzman, L., Sorkin, J., Travesset, C., Wang, Z., & Rozier, K. Y. (2024). Mission-Time LTL (MLTL) Formula Validation via Regular Expressions. In P. Herber & A. Wijs (Eds.), *Integrated Formal Methods* (pp. 279–301). Cham: Springer Nature Switzerland.

Farrell, M., Luckcuck, M., Monahan, R., Reynolds, C., & Sheridan, O. (2024a). Adventures in FRET and specification. In *International Symposium on Leveraging Applications of Formal Methods* (pp. 106–123).

Farrell, M., Luckcuck, M., Monahan, R., Reynolds, C., & Sheridan, O. (2024b). FRETting and formal modelling: A mechanical lung ventilator. In *International Conference on Rigorous State Based Methods.*

Farrell, M., Luckcuck, M., Sheridan, O., & Monahan, R. (2022a). FRETting About Requirements: Formalised Requirements for an Aircraft Engine Controller. In *Requirements Engineering: Foundation for Software Quality* (pp. 96–111). Springer. doi: 10.1007/978-3-030-98464-9_9

Farrell, M., Luckcuck, M., Sheridan, O., & Monahan, R. (2022b). FRETting about requirements: formalised requirements for an aircraft engine controller. In *International Working Conference on Requirements Engineering: Foundation for Software Quality* (pp. 96–111).

Farrell, M., Mavrakis, N., Ferrando, A., Dixon, C., & Gao, Y. (2022). Formal Modelling and Runtime Verification of Autonomous Grasping for Active Debris Removal. *Frontiers in Robotics and AI*.

Freund, E. (2012). IEEE standard for system and software verification and validation (IEEE std 1012-2012). *Software quality professional*, *15*(1), 43.

Giannakopoulou, D., Mavridou, A., Rhein, J., Pressburger, T., Schumann, J., & Shi, N. (2020). Formal requirements elicitation with FRET. In *International Working Conference on Requirements Engineering: Foundation for Software Quality (REFSQ-2020).*

Giannakopoulou, D., Pressburger, T., Mavridou, A., & Schumann, J. (2021). Automated formalization of structured natural language requirements. *Information and Software Technology*, *137*. doi: 10.1016/j.infsof.2021.106590

Hirshorn, S. R., Voss, L. D., & Bromley, L. K. (2017). *NASA Systems Engineering Handbook* (Tech. Rep.). Washington, DC: NASA.

Holzmann, G. J. (2004). *The SPIN model checker: Primer and reference manual* (Vol. 1003). Addison-Wesley Reading.

IV, L. G. M., Donohue, J., Davis, J. C., Lee, D., & Servant, F. (2023). *Regexes are Hard: Decision-making, Difficulties, and*

Risks in Programming Regular Expressions. Retrieved from https://arxiv.org/abs/2303.02555

Kamp, J. A. W. (1968). *Tense logic and the theory of linear order.* University of California, Los Angeles.

Kouskoulas, Y., Machado, T. J., & Genin, D. (2020). Formally verified timing computation for non-deterministic horizontal turns during aircraft collision avoidance maneuvers. In *International Conference on Formal Methods for Industrial Critical Systems* (pp. 113–129).

Koymans, R. (1990). Specifying real-time properties with metric temporal logic. *Real-time systems*, *2*(4), 255–299.

Krammer, M., Stirgwolt, P., & Martin, H. (2015). *From natural language to semi-formal notation requirements for automotive safety* (Tech. Rep.). Warrendale, PA: SAE Technical Paper.

Kwiatkowska, M., Norman, G., & Parker, D. (2011). PRISM 4.0: Verification of probabilistic real-time systems. In *International conference on computer aided verification* (pp. 585–591).

Li, J., Vardi, M. Y., & Rozier, K. Y. (2019). Satisfiability checking for mission-time LTL. In *International Conference on Computer Aided Verification* (pp. 3–22).

Luckcuck, M., Farrell, M., Sheridan, O., & Monahan, R. (2022). A methodology for developing a verifiable aircraft engine controller from formal requirements. In *2022 IEEE Aerospace Conference (AERO)* (pp. 1–12).

Luppen, Z., Jacks, M., Baughman, N., Stilic, M., Nasers, R., Hertz, B., … Rozier, K. Y. (2022). Elucidation and analysis of specification patterns in aerospace system telemetry. In J. V. Deshmukh, K. Havelund, & I. Perez (Eds.), *Nasa formal methods* (pp. 527–537). Cham: Springer International Publishing.

Maler, O., & Nickovic, D. (2004). Monitoring temporal properties of continuous signals. In *International symposium on formal techniques in real-time and fault-tolerant systems* (pp. 152–166).

Manna, Z., & Pnueli, A. (1992). *The temporal logic of reactive and concurrent systems: specifications* (Vol. 1). Springer Science & Business Media.

Mavridou, A. (2022). Capturing and Analyzing Requirements with FRET. In *NASA Formal Methods 2022.*

Michael, L. G., Donohue, J., Davis, J. C., Lee, D., & Servant, F. (2019). Regexes are Hard: Decision-Making, Difficulties, and Risks in Programming Regular Expressions. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)* (p. 415-426). doi: 10.1109/ASE.2019.00047

Mokos, K., & Katsaros, P. (2020). A survey on the formalisation of system requirements and their validation. *Array*, *7*, 100030.

Moosbrugger, P., Rozier, K. Y., & Schumann, J. (2017). R2U2: monitoring and diagnosis of security threats for unmanned aerial systems. *Formal Methods in System Design*, *51*, 31–61.

Mucha, J., Kaufmann, A., & Riehle, D. (2024). A systematic literature review of pre-requirements specification traceability. *Requirements Engineering*, *29*(2), 119–141.

Nam, J., Na, S. H., Shin, S., & Park, T. (2022). Reconfigurable regular expression matching architecture for real-time pattern update and payload inspection. *Journal of Network and Computer Applications*, *208*, 103507. Retrieved from https://www.sciencedirect.com/science/article/pii/S1084804522001497 doi: https://doi.org/10.1016/j.jnca.2022.103507

Perez, I., Mavridou, A., Pressburger, T., Goodloe, A., & Giannakopoulou, D. (2022). Automated translation of natural language requirements to runtime monitors. In *International conference on tools and algorithms for the construction and analysis of systems* (pp. 387–395).

Pnueli, A. (1977). The temporal logic of programs. In *18th Annual Symposium on Foundations of Computer Science (SFCS 1977)* (pp. 46–57).

Pressburger, T., Katis, A., Dutle, A., & Mavridou, A. (2023). Using FRET to create, analyze and monitor requirements for a lift plus cruise case study. *NASA Technical Report.*

Schumann, J., Moosbrugger, P., & Rozier, K. Y. (2015). R2U2: monitoring and diagnosis of security threats for unmanned aerial systems. In *Runtime Verification: 6th International Conference, RV 2015, Vienna, Austria, September 22-25, 2015. Proceedings* (pp. 233–249).

Sheridan, O., Becker, L. B., Farrell, M., Luckcuck, M., & Monahan, R. (2025). Sharper Specs for Smarter Drones: Formalising Requirements with FRET. In *International Working Conference on Requirements Engineering: Foundation for Software Quality* (pp. 350–362).

Wang, Z., Gamboa Guzman, L. P., & Rozier, K. Y. (2026). WEST: Interactive validation of mission-time linear temporal logic (MLTL). *Science of Computer Programming*, *248*, 103365. Retrieved from https://www.sciencedirect.com/science/article/pii/S0167642325001042 doi: https://doi.org/10.1016/j.scico.2025.103365

Wang, Z., Kosaian, K., & Rozier, K. Y. (2025). Formally Verifying a Transformation from MLTL Formulas to Regular Expressions. In A. Gurfinkel & M. Heule (Eds.), *Tools and Algorithms for the Construction and Analysis of Systems* (pp. 254–275). Cham: Springer Nature Switzerland.

Youn, W. K., Hong, S. B., Oh, K. R., & Ahn, O. S. (2015). Software certification of safety-critical avionic systems: DO-178C and its impacts. *IEEE Aerospace and Electronic Systems Magazine*, *30*(4), 4–13.

About the authors

Songyan Lai is an undergraduate student at Maynooth University. You can contact the author at laisongyan@foxmail.com or visit https://songyanlai.github.io/.

Rosemary Monahan is a Professor in the Department of Computer Science and an affiliate of the Hamilton Institute at Maynooth University. You can contact the author at rosemary.monahan@mu.ie or visit https://www.maynoothuniversity.ie/faculty-science-engineering/our-people/rosemary-monahan.