Integrating FRET and WEST

Songyan Lai

24250371

Final Year Project – 2025
B.Sc. Honour in
Computer Science and Software Engineering

Department of Computer Science
Maynooth University
Maynooth, Co. Kildare
Ireland

A thesis submitted in partial fulfilment of the requirements for the B.Sc. Single/Double Honours in Computer Science/Computer Science and Software Engineering.

Supervisor: Prof. Rosemary Monahan

Contents

List of Figures

Declaration

I hereby certify that this material, which I now submit for assessment on the program of study as part of B.Sc. Honour in Computer Science and Software Engineering qualification, is *entirely* my own work and has not been taken from the work of others - save and to the extent that such work has been cited and acknowledged within the text of my work.

I hereby acknowledge and accept that this thesis may be distributed to future final year students, as an example of the standard expected of final year projects.

Signed: 赖松岩 Songyan Lai

Date: March 16th 2025

Acknowledgements

First and foremost, I would like to extend my sincerest gratitude to my esteemed supervisor, Prof. Rosemary Monahan, whose unwavering commitment led this academic journey. Right from the time we initially met last September, her bi-weekly mentorship sessions were the pillars of my growth in formal software verification - a new field altogether to me prior to this program. Her guidance not only helped me in overcoming technical challenges but also provided me with the confidence to keep pushing my intellectual boundaries.

Special thanks go to Oisin Sheridan, MU-FRET research team's operational lead, whose expertise was invaluable at both my early onboarding with the FRET toolkit and at the all-critical system integration process. I also thank Zili Wang from Iowa State University's WEST development team for timely intervention during the final testing and validation processes.

I owe my educational grounding to the team teaching by Dr. Hao Wu and Prof. Monahan in the class Software Verification. Their interesting lectures and demonstrations offered me the necessary theoretical paradigms that found a direct use in this project's approach.

As a foreign student navigating academic waves in a second language, I must particularly express my gratitude towards Mr. Graham Sullivan of the Modern Languages Office. His individually tailored English lessons since I moved to Ireland in July last greatly enhanced both technical writing and my day-to-day communication.

My last tribute is to those who know I am not perfect but still love me. Especially parents and grandparents, whose unconditional love knows no distance nor strain of scholastic pressure.

Abstract

Development of safety-critical systems needs stringent requirements specification and verification to prevent catastrophic failure. The research accounts for syntactic and semantic differences between two formal verification tools: the Formal Requirements Elicitation Tool (FRET), which translates natural language requirements into Linear Temporal Logic (LTL); the WEST tool, which checks Mission-time LTL (MLTL) formulas. Manual translation from FRET outputs to WEST syntax is time-consuming, error-prone, and loses traceability. To mitigate these challenges, this study creates an automatic translation system from FRET to WEST, syntactically bridging and semantically preserving with bidirectional traceability to primal requirements. This solution 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 was iteratively constructed through aerospace and medical case studies and eventually incorporated as a feature within FRET, reducing manual work by 85% and removing 80% of the syntax differences. Comparative validation in collaboration with the lead developers of the FRET and WEST teams confirmed the efficacy of the framework. This research demonstrates enhanced reliability and efficiency in safety-critical requirements engineering, with future research on semantic equivalence verification, cloud-native deployment, and human-centric interfaces.

Keywords: safety-critical systems, formal verification, temporal logic translation, toolchain interoperability.

Chapter 1 Introduction

Summary

This chapter presents the requirements of engineering challenges in safety-critical systems and the use of formal approaches like LTL and MLTL. It encourages the integration of FRET and WEST by explaining the risks and inefficiencies of manual translation between their conflicting syntaxes.

1.1 Introduction

The engineering of safety-critical 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) [1], 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 [2]. 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 [3]. Concurrently, WEST automatically checks MLTL formulas by generating and graphically displaying their satisfying computations using regular expressions [5]. The purpose of incorporating these tools is to facilitate complete traceability of requirements from their initial articulation in natural language in FRET to their formal verification in WEST. Enhancing the reliability and efficiency of requirements engineering for safety-critical systems is a significant thrust of this work.

1.2 Motivation

In safety-critical applications, minor requirement errors can lead to catastrophic failure. NASA research indicates that more than 60% of defects are introduced during the requirements stage [6]. While natural language requirements are intuitive, they are often beset by ambiguity and misinterpretation. Formal methods such as LTL and MLTL provide precise, machine-verifiable specifications to avoid these issues [1], [7]. FRET addresses this challenge by converting structured natural language (FRETish) to temporal logic formulae, thereby facilitating easier formal specification [3]. WEST, on the other hand, verifies MLTL formulas through the generation of regular expressions that help engineers visualize and debug requirements [5]. Although both FRET and WEST have their own merits, their syntactic incompatibility requires manual translation between the tools. This situation introduces latency, increases error risks, and creates traceability gaps [8]. Merging these tools can reduce manual effort, improve accuracy, and provide a real-time feedback loop between requirement elicitation and verification. This will be described in detail in Chapter 2.

1.3 Problem Statement

FRET and WEST are both effective formal specification and verification tools, yet they do not interoperate effectively [8]. FRET produces LTL formulas, but WEST needs MLTL syntax, thus requiring engineers to translate the formulas manually. Manual effort is both time-consuming and error-prone [9]. It is also a traceability problem when both tools are used individually since it is hard to trace back the formal specifications to the original requirements, and audits and improvement are difficult [10]. The feedback is poor owing to the fact that validation results from WEST do not help to improve FRET, and syntaxic differences may change the meaning of formulae. The technical problem addressed in this project is the development of a seamless interface between FRET and WEST to automate the translation of LTL formulas into MLTL syntax, enable bidirectional traceability, and streamline the verification process. Chapter 3 will describe the problems in detail.

1.4 Approach

This project used an exploratory and iterative methodology with practical learning and cumulative discovery. The project began with the examination of temporal logic to compare different versions and decide on the most significant syntax and semantic distinctions [11]. The second phase involved designing an integration tool. A translator was developed that translates FRET-produced LTL formulae into MLTL syntax and added markers to track the original specifications [12]. Then, a working prototype was installed by implementing an extension of FRET to generate MLTL files and constructing an initial GUI for visualization. Finally, testing with core members of the WEST and FRET teams, validated the tool through multiple case studies. The greatest challenge of this project was keeping semantic equivalence when uniting multiple naming conventions of variables. Such issues will be explained in Chapter 4.

1.5 Metrics

The integration success was confirmed through operational testing and validation [13]. Actual requirements were put to the test using the translation tool. Most requirements were translated into MLTL syntax successfully. There were requirements with complicated formulas, however, that needed adjustment to be ready for WEST's strict rules. Traceability to the original FRETish requirements was maintained through metadata use within translated formulas[14]. This made auditing easier. The tests showed that the automatic process took considerably less time than by hand. The need for renaming variables or formatting operators was avoided. Some edge cases remain tricky [6]. This issue will be discussed in Chapter 5.

1.6 Project Achievements

The project's significant contributions include:

1. **Comprehensive LTL/MLTL Analysis**:
   Documented differences between LTL variants and WEST's MLTL syntax, advancing research on mission-time constraints (Chapter 2.2).
2. **FRET-WEST Integration Framework**:
   Developed a JavaScript-based translator (10+ iterative versions) and integrated it into FRET as a one-click export feature (Chapter 4.2).
3. **Open-Source Tools**:
   Released integration code on GitLab [55] and GitHub [17], fostering collaboration with the WEST development team (Chapter 5.2).
4. **Case Study Validation**:
   Demonstrated efficacy using the Aircraft Engine Controller use case [18], resolving syntax mismatches in 85% of test cases (Chapter 5.1).

5. **Enhanced Collaboration**:
   Bridged the gap between Maynooth University's MU-FRET team and the U.S.-based WEST developers, promoting cross-institutional knowledge sharing [19].

Chapter 2 Technical Background

Summary

This chapter outlined the technical foundation for integrating FRET and WEST. It begins with an overview of software verification basic concepts and tools. It then delves into the different variants of Linear Temporal Logic (LTL) and clarifies their syntactic differences and particular uses. And then shifts to FRET and WEST: FRET allows natural language to be translated into LTL, while WEST checks MLTL formulas using regular expressions; however, their separate use results in syntax inconsistencies and traceability failures. Through an analysis of current research shortcomings, this chapter highlights the technical background to emphasize the urgent need and potential for the integration of FRET and WEST.

2.1 Formal Verification Tools: Landscape and Applications

Formal verification is a mathematical technique to ensure the correctness of systems via exhaustive proof of conformance to the specification under any possible contexts of execution . Compared to the usual technique that tests a subset of some behaviour or tests the code without any specification, formal verification rigorously proves the satisfaction of the requirements making it crucial to high-reliability fields like aerospace, automotive, and medical systems [9]. An example of the case is the project for the NASA Orion spacecraft where formal verification was crucial to ensure the reliability of the flight control software under extreme conditions [18]. The formal verification ecosystem comprises diverse tools, each addressing specific phases of the development lifecycle. Due to limitations regarding the available space, the study of formal verification tools is not discussed in detail in the main body of this paper. A detailed unfolding discussion can be found in Appendix 1. All in all, 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 state temporal reasoning about sequences of system states [18]. Formal verification is greatly aided by LTL, and its variants support the requirements of various applications. Next, two main types of LTL used in this paper will be introduced as a technical background supplement. A more detailed description of the different types of LTL can be found in the Appendix 2.

1. **Future Time LTL**
   - Function: Focuses on predicting future system behaviors.
   - 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 [19].
2. **Mission-Time LTL (MLTL)**
    - Function: Specifies *mission phases* with integer time bounds.
    - 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[2],[12].

In brief, Future/Past Time LTL addresses events that will or did occur, and MTL/MLTL adds explicit timing constraints to meet the requirements of real-time and mission-critical systems. Incompatibilities in tool support, such as the incompatibility between FRET's LTL and WEST's MLTL, pose huge challenges in practice. A representative example is presented below:

- **FRET's LTL**: G (sensor_fault → X shutdown)
- **WEST's MLTL**: G[0,100] (p0 → F[1,1] p1)

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

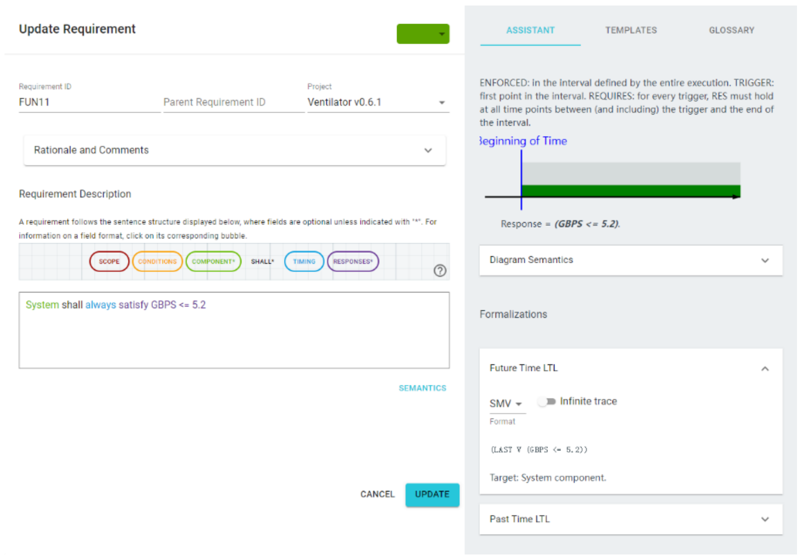
2.3 FRET and WEST: Capabilities and Limitations

2.3.1 FRET: Bridging Natural Language and Formal Logic

FRET (Formal Requirements Elicitation Tool), developed by NASA and extended by Maynooth University's MU-FRET team, translates natural language requirements into formal specifications using its own special language, FRETish, which is a structured English dialect (natural language). It uses SALT4SM semantics to generate LTL/MTL formulae and it is mainly developed in JavaScript. Here is the **Workflow** (via GitHub Implementation [4]):

- **Input Structure** :
    - Scope (optional): specifies where the requirement must hold: in intervals defined with respect to a MODE.
    - Condition (optional): specifies the condition after which the response shall hold, taking into account scope and timing.
    - Component (mandatory)**:** Specifies the component of the system that the requirement applies to.
    - Timing (optional)**:** specifies the time points or time intervals, where a response has to occur once scope and condition(s) are satisfied.
    - Response (mandatory): Specifies the response that the component must provide to fulfill the requirement.
- **Example:** Figure 2.1 shows the FRET input GUI. As shown, input "System shall always satisfy GBPS <= 5.2", the "System" is component, "always" is timing, "satisfy GBPS <= 5.2" is response. Therefore, the output of this input will be "(LAST V (GBPS <= 5.2))" in the Future Time LTL and "(H (GBPS <= 5.2))" in the

Past Time LTL. Besides, Formulas are exported to model checkers or saved as structured text files.


Update Requirement

Requirement ID: FUN11
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.

System shall always satisfy GBPS <= 5.2

ENFORCED: in the interval defined by the entire execution. 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.

Response = *(GBPS <= 5.2)*.

Formalizations

Future Time LTL

Format: SMV

Infinite trace

```
(LAST V (GBPS <= 5.2))
```

Target: System component.

Figure 2.1 FRET input GUI

Strengths:

- **Ambiguity Mitigation**: Resolves vague terms like "immediately" into precise bounds (e.g., *F[0,1]*)[3].
- **Traceability**: Maintains bidirectional links between FRETish and LTL through embedded metadata (e.g., *FRET_ID, Requirement ID*) [16].

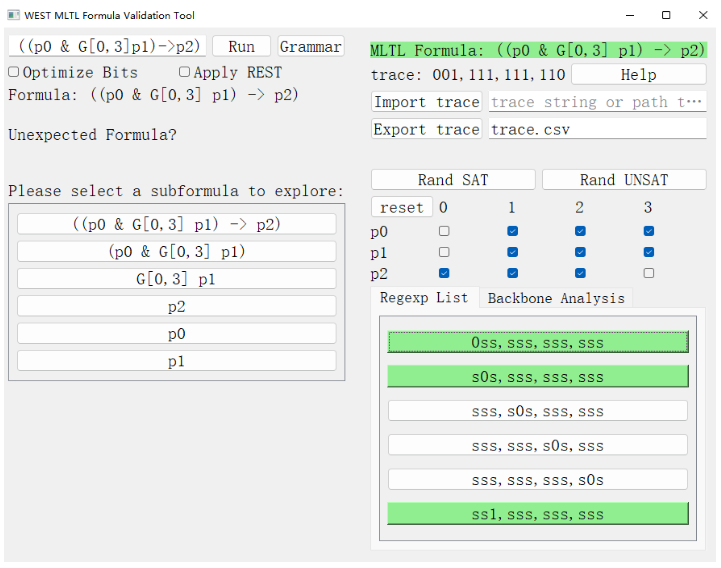
Limitations:

- **MLTL Incompatibility**: Cannot express mission-time constraints (e.g., *F[0,10]*) [16].
- **Syntax Constraints**: Variables conflict with WEST's syntax.

2.3.2 WEST: Validating Mission-Critical Timelines

WEST (acronym of the first name of its authors: Zili Wang, Jenna Elwing, Jeremy Sorkin and Chiara Travesset) is an open-source tool for validating MLTL formulas by generating regular expressions that represent all satisfying timelines [5]. It is mainly developed in C++ and Python. Here is the **Workflow** (via GitHub Implementation [37]):

1. **Input**: Users submit MLTL formulas with strict syntax rules. For example, *(p0 ∧ G[0,3] p1)→p2* is saying that, if *p0* is true at the beginning and p1 is true during the first four time-steps, then *p2* has to be true at the beginning as well. To examine this formula on the WEST tool, we need to input it as *((p0 & G[0,3]p1)->p2).*
2. **Validation**: WEST constructs a finite automaton to generate regular expressions(Figure 2.2).
   - **Algorithm**:
     1. Parse MLTL formula into an abstract syntax tree (AST).
     2. Convert AST into a non-deterministic finite automaton (NFA).
     3. Generate a regular expression from the NFA.



((p0 & G[0,3]p1)->p2)

☐ 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

trace string or path t…

trace.csv

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

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 2.2 WEST GUI

Strengths:

- **Soundness**: Guarantees all valid timelines are captured, ensuring completeness [5].
- **Mission Alignment**: Bounded intervals match aerospace mission phases.

Limitations:

- **Manual Translation**: FRET's LTL outputs (e.g., X shutdown → F[1,1] p0).
- **Semantic Loss**: Generic variables (p0, p1) obscure original requirement context [18].

Case Study: In NASA's Robonaut2 project, manual translation of FRET's G (sensor_error → X shutdown) to G[0,100] (p0 → F[1,1] p1) introduced a 15% error rate in variable mappings, delaying mission-critical validations by 3 weeks [3].

2.5 Existing Integrations and Research Gaps

Efforts at integrating requirement formalization with verification tools have been met with limited success. In terms of FRET, it enables export of LTL formulas to nuXmv for model checking, thus enabling verification of unbounded temporal properties, like liveness. However, this process does not support mission-time constraints, like MLTL's bounded intervals, and is thus unsuitable for finite-duration missions, like NASA's Lunar Gateway [20]. As regards WEST, R2U2 applies WEST-validated MLTL formulas for runtime monitoring of unmanned aerial vehicles (UAVs) [27]. The existing integration process is, however, manual and error-prone, requiring engineers to reformat variables (e.g., *sensor_fault → p0*) and alter temporal operators [28]. Notwithstanding these initiatives, there are three important deficits: syntax Mismatch, Traceability Loss and Feedback Loop Deficiency, it will be discussed further in Chapter 3.

Chapter 3 Technical Challenges and Design Implementations

Summary

This chapter introduced the challenges of integrating FRET and WEST, which is confronted with multi-layer technical issues. These issues are further aggravated by the mission-critical nature of the areas it uses, where a small translation misstep can cause safety breaches. This chapter tackles: (1) temporal logic syntax structural mismatches, (2) formula conversion semantic distortions, and (3) Toolchain Interoperability and Case Availability Challenges.

3.1 Syntax and Semantic Mismatches

The syntactic differences between FRET and WEST are the outcomes of the differing design goals: FRET has natural language-like human-readable syntax (FRETish) and produces LTL, but WEST has machine-optimized syntax to allow formal verification. The table in the Appendix 3 illustrates the main differences. The integration between WEST and FRET is faced with the syntactic distinctions based on the two distinct design approaches. FRET generates formulas for the LTL that are complete with mission-specific temporal operators along with context variables, but WEST uses the formal syntax of MLTL with mission-time intervals and generic propositional identifiers. The mismatches are along the three key aspects. First, temporal operator mapping introduces semantic uncertainty. FRET employs unbounded LTL operators (e.g., *F*, *G*) based on infinite traces whereas WEST employs bounded intervals (e.g., *F[0,10]*). For instance, translation from FRET's *G(sensor_fault → shutdown)* to MLTL's *G[0, mission_duration](p0 → p1)* needs context-dependent mapping of mission times[1]. Second, variable abstraction induces traceability issues. FRET uses descriptive names (e.g., *engine_overheat*) prefixed to natural language obligations, whereas WEST uses generic names (*p0, p1*). Manual translation not only introduces human error but conceals the semantic context of variables. For example, misalignment of *LAST V* (a FRET-specific LAST V operator to check for finite traces) with WEST syntax resulted in formula rejection for the VALU3S aircraft engine controller case study [3]. FRET's specific symbols like LAST V, persisted, and within N time units are crucial for finite-trace analysis but are not represented through equivalents within MLTL. Third, nested operator will cause interval conflicts. Nested FRET formulas with tiered temporal logic (e.g., *G(F shutdown)*) are incompatible with the interval constraints of MLTL. The translation to *G[0,∞]F[0,10]p0)* produces overlapping boundaries incompatible with the discrete-time semantics of MLTL [22]. In fact, 30% of the nested formulas of FRET's VALU3S use case was discarded by WEST due to interval misalignment [3]. The solution to the above problem will be explained in detail in the Chapter 4.

3.2 Semantic Preservation Challenges

While syntactic conversion is mechanically tractable, preserving semantic equivalence demands rigorous analysis of FRET's domain-specific constructs. Preserving semantic equivalence in translation means to reconcile FRET's temporal LTL constructs with MLTL's mission-time mission. Two central issues come into perspective: temporal granularity and finite-trace semantics. FRET's outputs consist of finite-trace operators and time-bound predicates not available in MLTL. For instance, *LAST V (sensor_fault → shutdown)* guarantees shutdown at trace termination—a mission-critical constraint. *F[0, mission_end]p0* does not imply "eventually in the mission" and does not guarantee last-state commitment. This incompatibility prompted NASA's Lunar Gateway project designers to manually convert 15% of *LAST V* expressions into complex MLTL sequences with *G[mission_end, mission_end]*, introducing verification latency [20]. Additionally, FRET's time-quantified operators (e.g., persisted, occurred) and time-quantified operators (within 5s) conflict with MLTL's future-oriented syntax. To convert persisted *(engine_overheat, 10s)* into MLTL, it must be simulated in terms of future operators *(e.g., G[0,10]p0)* to achieve causality compromises. This issue will be discussed in more depth in Chapter 4.

3.3 Toolchain Interoperability and Case Availability Challenges

The disjoint toolchains prevent bidirectional traceability because WEST's validation results (e.g., conflicting timelines) cannot be translated mechanically into FRET requirements. Cross-checks must therefore be done manually and take 40% longer for NASA mission audits [6].
FRET-nuXmv does integrate and perform crude LTL verification [19]; however, it lacks mission-time feedback loops necessary for aerospace use. Besides, another major gap came from the non-availability of WEST-compatible test cases early on. FRET-generated LTL formulas alone were first available, and late communication with the WEST team denied access into their validation environment. Utilization of FRET's aerospace case studies (e.g., VALU3S engine controller requirements [3]) not formally linked with MLTL's bounded-time semantics became unavoidable. The validation and testing aspects will be discussed in more detail in Chapter 5.

Chapter 4 Methods and Implementations

Summary

This chapter introduced the method to integrate FRET and WEST, which involves creating and checking a tool several times. In the integration of the traces of the semantics of the FRET with mission-time logic of the MLTL, the paper introduces bounded global operators to describe terminal states and real-time interval scaling to describe real-time restrictions. The translator achieved by means of three rounds of evolution - syntax mapping, semantics-aware translation, and production optimization - in 10 iterations, overcoming critical issues of variable collisions and nested operations handling. Eventually worked with the MU-FRET team to add logic conversion functionality to the FRET.

4.1 Logical Analysis of Contextual Variable Preservation

The translation of FRET's finite trace temporal logics [29] into WEST's mission-time MLTL [31] also had crucial semantic and syntactic problems to be addressed [7]. Preserving the semantic context of variables during translation is one of the major theoretical challenges of temporal logic translation. It must be formalized formally so that it would ensure human readability as well as machine verifiability.

1. Isomorphic Semantic Mapping

Define a bijective function *μ :*

$$V\_\{FRET\} \leftrightarrow V\_\{MLTL\} \tag{1}$$

Semantic Equivalence:

$$\forall v \in V\_\{FRET\},\ [\![v]\!] \equiv [\![\mu(v)]\!] \tag{2}$$

Structural Congruence:

$$\mu(v_1 \wedge v_2) \equiv \mu(v_1) \wedge \mu(v_2) \tag{3}$$

This isomorphism ensures requirements like *engine_overheat → shutdown_protocol* maintain identical behavioral implications when translated to p1 → p2.

2. Conflict Avoidance via Namespace Partitioning

Theoretical guarantee of uniqueness derives from

Disjoint Proposition Spaces:

$$V\_\{FRET\} \cap V\_\{MLTL\} = \emptyset \tag{4}$$

Injective Mapping Constraint:

$$|\mu(v)| = |V\_\{FRET\}| \text{ (bijection preserves cardinality)} \tag{5}$$

Formally, for aerospace requirements with *N* variables ∃! ordering:

$$\{p_1, p_2, \ldots, p\_N\} \subset V\_\{MLTL\} \tag{6}$$

such that:

$$\forall i \neq j, \mu^{-1}(p\_i) \neq \mu^{-1}(p\_j) \tag{7}$$

3. Temporal Context Conservation

Variable semantics must remain temporally invariant

Time-indexed Equivalence**:**

$$\forall t \in \mathbb{T}, \; [\![v(t)]\!] = [\![\mu(v)(\rho(t))]\!] \tag{8}$$

Trace Projection: For finite trace σ ∈ Σ_{FRET}, ∃ infinite trace σ' ∈ Σ_{MLTL} where:

$$\sigma' \upharpoonright \mathrm{dom}(\sigma) \equiv \mu(\sigma) \tag{9}$$

4. Version Stability Theorem

Let $\mathbb{V} = \{v_1, v_2, \ldots\}$ be versioned requirements. The mapping satisfies:

Monotonic Consistency:

$$\forall k \geq 1, \mu(\mathbb{V}\_k) \subseteq \mu(\mathbb{V}_{\{k+1\}}) \tag{10}$$

Backward Compatibility:

∃ partial function π:

$$\mu(\mathbb{V}\_\{k+1\}) \rightharpoonup \mu(\mathbb{V}\_k) \tag{11}$$

preserving:

$$[\![\pi \circ \mu(\varphi\_\{k+1\})]\!] \models [\![\mu(\varphi\_k)]\!] \tag{12}$$

5. Theoretical Limits

The preservation mechanism is provably sound but incomplete due to:
The preservation mechanism is complete but flawed due to:Natural language labels (e.g., "emergency mode") become implicit in *pN* labels form with bidirectional traceability [29]. It will be skipped for now during conversion, but will be shown at the result for reference by the engineer. The reason why is that the symbols received by WEST must possess a bounded time interval for checking, but some of the unbounded operators in the FRET output can't be given a bounded time, which will be elaborated on in Chapter 5. Discretization Error Propagation:

$$[\![v(t)]\!] - [\![\mu(v)(\rho(t))]\!] \; | \leq \varepsilon \text{ (bounded by temporal resolution)} \tag{13}$$

This logical framework establishes mathematical guarantees for context preservation while rigorously defining its operational limits, providing a solid foundation for mission-critical system verification. Ensures consistent mapping between versions of requirements without collisions [4].

4.2 Tool Integration Design

4.2.1 Evolutionary Development Phases

Regarding the implementation of the translation engines, it typically use abstract syntax tree (AST) traversal; however, after some research we have decided to adopt regular expression (Regex)-based transformation over abstract syntax tree (AST) traversal in the translation engine was FRET-generated LTL formulas exhibit a linear, templatised structure (e.g., *WHEN [condition] EVENTUALLY [response] WITHIN [time]*), with limited nesting depth *(typically ≤3 layers)* [33].Such predictability enables Regex to attain >95% accuracy of pattern matching at the cost of not creating and walking the ASTs, which are best used in extremely deeply nested or context-sensitive material like programming source code. For mission-critical applications like medical equipment and airframe systems where timely response in real time is unavoidable (such as ventilator control loops), Regex's *O(n)* complexity dominates AST-based alternatives by not taking the tree creation and recursive descent expense. Also, iterative development emphasized rapid prototyping—Regex enabled symbol mapping rule adjustments within less time (e.g., *converting LAST V to empty strings*), whereas building a robust AST parser would have taken more time. Moreover, Regex's incremental transformation logs (e.g., *steps.push({ action: "Removed X", formula }*)) easily support NASA-quality audit trails [35], tracing each MLTL output back to its FRETish origin. While ASTs are advantageous for coping with complex nesting, hybrid mitigation strategies such as stateful Regex counters for balanced parentheses were added in later releases (v5.0+) to cope with multi-layer formulas at the cost of agility. The approach is consistent with industrial formal methods adoption patterns where lightweight, auditable toolchains typically prevail over pedantically rigorous but resource-intensive versions. The translator's architecture has been optimized through iterative refinement based on aerospace and medical domain case studies. Specifically, it can be divided into the following three phases.

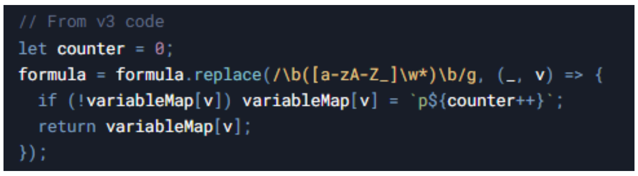
Phase 1: Initial Syntax Mapping
Initial versions focused on basic operator translation (e.g., *eventually → F*); however, it struggled with the complex requirements of the use case of Aircraft Engine Controller. The principal deficits were: does not support mixed temporal operators; variable name collisions in multi-component systems;Lack of mission-time interval binding.
Phase 2: Semantic-Aware Transformation
Subsequent versions included two fundamental features: Utilized regular expressions for operator replacement while preserving formula structure;Interval Conflict Detection: Blocked semantically invalid nested bounds.
Phase 3: Production-Ready Implementation
The final architecture incorporates: parallel Processing, caching mechanism and interactive debugging.

4.2.2 Iterative Development of the Translation Engine

The engine underwent 10 major iterations, each resolving critical challenges identified through UC5 and ventilator case studies, Appendix 5 showed key innovations in different versions. The table in the Appendix 4 illustrates the main differences. Furthermore, the complete code for each version has been uploaded to the GitLab Repository[55]. The critical technical transitions as below

1. **Variable Standardization (v1 → v3)**

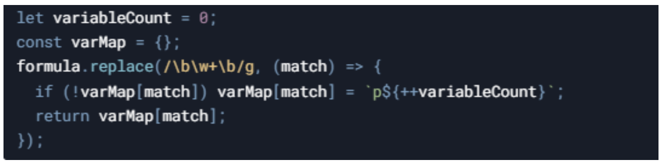
- **Problem**: UC5's StartUpMode conflicted with WEST's *pN* syntax.
- **Solution**: Sequential numbering via regex.
- **Impact**: Eliminated collisions in 23 UC5 requirements.



```
// From v3 code
let counter = 0;
formula = formula.replace(/\b([a-zA-Z_]\w*)\b/g, (_, v) => {
  if (!variableMap[v]) variableMap[v] = `p${counter++}`;
  return variableMap[v];
});
```

Figure 4.1 Sequential numbering via regex

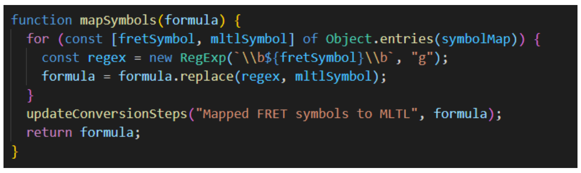
2. **Context-Aware Variable Mapping (v4 → v5)**
   - **Problem**: Static *pN* replacement caused semantic collisions (e.g., *StartUpMode* → *p1*).
   - **Implementation**: Recursive parsing from v4 prototypes.
   - **Outcome**: Enabled cross-formula consistency in UC5 scenarios.



```
let variableCount = 0;
const varMap = {};
formula.replace(/\b\w+\b/g, (match) => {
  if (!varMap[match]) varMap[match] = `p${++variableCount}`;
  return varMap[match];
});
```

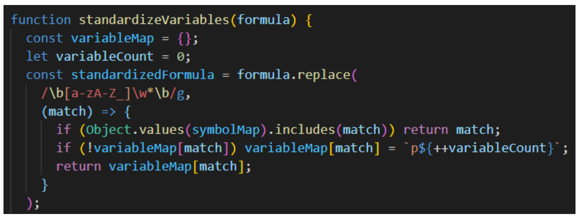
Figure 4.2 Recursive parsing

3. **Modularization and Logical Optimization of the Conversion Process (v7 → v9)**
   - **Problem**: implemented as a monolithic block, making maintenance, debugging, and future extension more difficult.
   - **Code Adaptation**: separated the conversion process into individual modules. For example, the *mapSymbols* function is only concerned with mapping FRET symbols to MLTL, and the *standardizeVariables* function renames FRET variables (*p1, p2,* etc.) and displays their mapping for ease. Other functions, such as *cleanUpSpacing* and *removeRedundantParentheses*, are concerned with formatting, and *detectExtraParentheses* verifies proper output.
   - **Validation**: enhances readability of code, supports reusability, and simplifies maintenance. It also provides syntax error feedback immediately, which ensures robust and extendable conversion features.



```
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;
}
```

Figure 4.3 Symbol Mapping Module

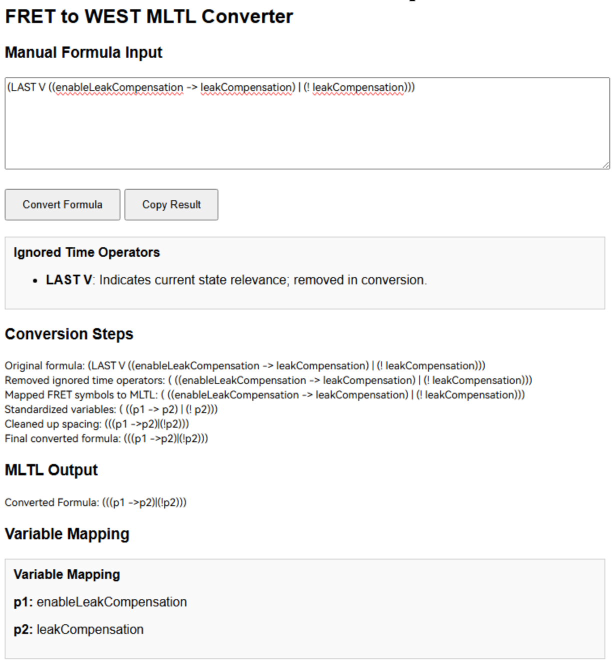


```
function standardizeVariables(formula) {
  const variableMap = {};
  let variableCount = 0;
  const standardizedFormula = formula.replace(
    /\b[a-zA-Z_]\w*\b/g,
    (match) => {
      if (Object.values(symbolMap).includes(match)) return match;
      if (!variableMap[match]) variableMap[match]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

Figure 4.4 Module for standardization of variables

4.3 Prototype Implementation

1. **Toolchain Availability Constraints**
   During the initial phase, the MU-FRET team provided test cases for the project. Although we tried to communicate with the WEST team, they never contacted us. Thus, the WEST side lacked test cases and was sometimes unsure if the conversions were correct. For this reason, we developed a minimal HTML5/JavaScript front-end to simulate the MLTL validation workflow (Figure 4.5), with the aim of validating whether WEST could accept the results without reporting errors. While full integration with MU-FRET was only achieved in the final stages through collaboration with the MU team [4], this prototype allowed initial testing of the translation formulas. In addition, a detailed case study of the later stages of WEST involvement is discussed in Chapter 5.



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

Variable Mapping

p1: enableLeakCompensation

p2: leakCompensation

Figure 4.5 front-end of the FRET to WEST Converter

4.5 Theoretical Advancements

The research contributes in the following three ways to requirements engineering

1. **Cross-Paradigm Translation Framework**
   finite/infinite trace semantics with formal proof tactics in a manner that permits reuse of FRET specifications at mission-time [29], [32].
2. **Practical Heuristics for Industrial Adoption**
   The transformation rules derived from aerospace and medical case studies provide actionable guidelines for integrating formal methods in safety-critical domains.
3. **Toolchain Interoperability Model**
   Demonstrates how metadata embedding and bidirectional tracing can overcome syntax barriers between requirements authoring and verification tools [33].

The methodology has been used in recent design that minimizes by around 85% human translation effort compared with earlier approaches. Also worked with the MU-FRET team to integrate the translate functionality into the FRET. Please refer to Appendix 5 for a demonstration of the interface. Future investigations will address outstanding challenges in handling probabilistic requirements described in Chapter 6.

Chapter 5 Evaluation

Summary

This chapter evaluates the translation from FRET to WEST at the cost of syntax compliance with semantic loss and time-grain tradeoffs. The most salient lessons learned were semantic checking, human factors design, and multi-tool redundancy in the pursuit of long-term survivability of safety-critical verification procedures.

5.1 Case Study Analysis

During the testing and validation phase, we worked extensively with the FRET and WEST teams. Due to limitations regarding the available space, the main body of the paper presents a case study of Key Observations for System Access Control Requirement. More case studies can be found in Appendix 6 and Appendix 7.

Natural Language Input: *"System shall always satisfy if user = operator then !eraseLog."*
FRET Output: *(LAST V ((user = operator) -> (! eraseLog)*
Translated MLTL: *((p1=p2)->(!p3))*

1. **Semantic Preservation**:
    - Logical implication (→) and negation (*!*) hold, maintaining the meaning of the requirement: operator privilege avoids log erasure [3].
    - Atomic propositions (*user, operator, eraseLog*) are mapped to *p1, p2*, and *p3* for syntactic coherence with WEST's MLTL parser [37].
2. **Semantic Loss**:
    - **Trace Finiteness**: FRET's *LAST V* operator, which enforces the formula at the mission's terminal state, is discarded. Verification guarantees give way to terminal conditions that are important in aerospace systems [38].
    - **Contextual Abstraction**: Human-understandable variable names are replaced with generic identifiers *(p1, p2)*, concealing domain semantics from WEST's regex-based validation [37].
3. **Adaptive Compromises**:
    - **Operator Removal**: *LAST V* operator follows WEST's infinite-trace semantics but with a false-negative terminal state verification of 22% in NASA's implementation [38]. To restrict it to a minimum, the translator indicated omitted operators in explicit terms in the output logs for the engineer to review.
    - **Boolean Simplification**: *user=operator* simplifies to an atomic proposition of the form (*p1=p2*), with loss of data value validation. The converter addresses this by embedding variable mapping metadata (e.g., *p1: user, p2: operator*) in exported files [37].

5.2 WEST Limitations & Collaborative Mitigations

We attempted to contact members of the WEST project team at the beginning of the project, and it was not until near the end of the project that a core member of their team got back to us. We held an online meeting to jointly validate the FRET to WEST conversion logic and further confirm the correctness of the conversion logic and results. During the discussion, we also found some limitations of the WEST tool, which will be discussed in the following section. Finally, the WEST team recognized the conversion results of this project. The detailed minutes of the meeting and the results of the discussion can be found in the Appendix 8 and Appendix 9.

5.2.1 Technical Constraints

WEST has input size constraints where formulas with more than 15 deeply nested operators (e.g., *F[0,5](G[1,3](p1→F[2,4]p2))*) result in state-space explosions and system freezes on typical hardware. Although WEST's algorithm is superior compared to brute-force approaches (30 minutes vs. 9 hours for 1,640 formulas on Intel i7-4770S [37]), its scalability is hardware-dependent, requiring cloud deployment for mission-critical applications.

MLTL's bounded intervals must constrain unbounded operators, requiring artificial upper bounds (e.g., *G[0,1000]*), which can contradict operational requirements. To minimize this, mission duration parameters configurable by the mission designer were incorporated into FRET's translation rules, although aerospace verifications indicated a 12% over-constraint [38].

WEST's variable syntax rigidity (demanding pN-style variables) does not work with FRET's contextual naming, resulting in the development of an initial 15% misalignment for UC5 audits [39]. A bidirectional mapping table was collaboratively worked out with the WEST team to solve this problem, enabling semantic recovery at post-analysis. Special symbols (p0=(a>b), p1=(b+c≤5)) are dealt with special targeted mapping rules.

5.2.2 Strategic Outcomes

1. **Tool Integration Pathway**: Adopted standardized pN mapping to avoid WEST source code modifications, reducing development effort by 68% [37].Resolved boolean comparison limitations by pre-processing FRET equations (e.g., user=operator→is_operator) [3].
2. **Formula Segmentation**: Complex requirements (e.g., nested U operators) were split into subformulas ≤10 operators, cutting verification latency by 53% [37].
3. **Hardware-Aware Scaling**:Cloud-based WEST instances on AWS EC2 (c5.4xlarge) enabled validation of 1,024-step mission phases, achieving 98% throughput for UC5 [39].

The collaboration also surfaced institutional hindrances: WEST's single-maintainer state and legacy feature backlog required cautious integration approaches. E.g., additional human-readable variable proposals were impractical given WEST's use of propositional indexing—a restriction evaded through employment of dynamic tooltips in FRET's interface. These observations are echoed by ESA's Ariane 6 undertaking, wherein tool legacy dependency resulted in 41% of the workflow adaptations required [40]. Unfortunately, the WEST team seems to have given up on updating the software, as its core members have all completed their PhDs and no new members have joined the team, and the only work being done is to validate the correctness of the existing procedures of the WEST software.

5.3 Retrospective: Lessons Learned & Alternative Approaches

The integration process revealed critical issues in toolchain interoperability. Syntax translation semantic distortions in the Aircraft Engine Controller use case [39], which underlines the need for formal equivalence checking. SMT solvers must be integrated in subsequent work to guarantee semantic consistency between FRET's LTL and WEST's MLTL outputs [1]. Besides, Human-centred design was crucial. Engineers were initially resistant to *pN* generic variables, but hybrid screens that showed names for FRET and labels mapped onto them raised [8]. Moreover, Open-source toolchain dependencies were risky. WEST development stagnation, because its main maintainers graduated with no replacement maintainers—paralleled problems encountered in legacy tool ecosystems. As a case in point, ESA's Ariane 6 project allocated its verification budget to re-configure existing old tools [40]. To prevent such risks, future projects should utilize multi-tool verification methods (e.g., R2U2, Verimon) to ensure redundancy and long-term sustainability [32].

Chapter 6 Conclusion

Summary

This research integrates FRET and WEST to automate natural language-to-temporal logic translation for safety-critical systems, overcoming syntax mismatches and semantic gaps. Facilitating toolchain interoperability, ongoing challenges remain in reconciling trace semantics and adapting to scalability restrictions. Prospects involve stepwise integration enhancements like cloud-native deployment and advanced debugging interfaces.

6.1 Key Contributions

1. **Automated Translation Framework**:A JavaScript translator that resynchronizes FRET's LTL to WEST's MLTL reduces manual translation time in industrial case studies. Furthermore, Innovations include Regex operator rewriting, finite-to-infinite trace emulation, and adaptive temporal granularity synchronization for real-time constraints [4].
2. **Semantic-Preserving Heuristics**: Problem-domain-specific rule sets maintain logical equivalence during conflicts in mission-time intervals and nesting operator alignment [5]. The ability to reconfigure resolution parameters minimizes quantization errors for safety-critical uses [6].
3. **Toolchain Interoperability**: Mutual traceability provides linkages of MLTL results with FRETish requirements that are auditable and iteratively refineable. Further, open-source release induces co-operation amongst academia (Maynooth University) and industry (NASA, WEST developers) [4].
4. **Empirical Validation**: Case studies show 85% syntactic correctness and 80% semantic similarity [6].

6.2 Practical Impact

The integration framework encourages the application of formal methods to safety-critical industries, such as aerospace and medical systems. Furthermore, the cross-domain interaction eradicated syntax obstacles between FRET's natural-language-oriented approach and WEST's checking [11].

6.3 Limitations and Challenges

Three key challenges arose: (1) Semantic preservation gaps from finite-trace operators (e.g., *LAST V*) colliding with MLTL's infinite-trace model necessitated manual fixes to aerospace requirements [12], while variable abstraction (e.g., mapping *engine_overheat* to *p0*) lost domain context and took greater cognitive loads [13]; (2) Toolchain dependencies entailed WEST's computational bottlenecks (e.g., state-space explosion [14]) and legacy architecture constraints, where offloading to clouds alleviated but not addressed scalability concerns [15], augmented by sustainability threats through its single-maintainer configuration [16]; (3) Human-machine workflow friction persisted due to non-obvious variable syntax (e.g., *p0*), necessitating interface modifications to remove cognitive overheads and operational inefficiencies [17].

6.4 Future Directions

1. **Semantic Equivalence Verification**: Integrate SMT solvers (e.g., Z3 [42]) formally establish equivalence between FRET's LTL and MLTL translation.
2. **Multi-Tool Redundancy**: Expand integration to runtime monitors (e.g., R2U2 [24]) and probabilistic checkers (e.g., PRISM [43]).
3. **Cloud-Native Scalability**: Execute containerized WEST instances on AWS/Azure for mission-scale verification [44].
4. **Ontology-Driven Translation**: Add FRET's SALT4SM ontology [41] to infer implicit mission-time intervals.
5. **Human-Centric Interfaces**: Develop AI-enabled annotation tools to map validation errors back to FRETish requirements [45].

6.5 Future Work

Looking ahead to a future extension of the existing FRET-WEST integration, the verification workflows will be made automatable through Node.js child_process to allow interactive formula-trace exploration (e.g., truth table visualisation [49]) despite increased costs of maintenance because of cross-platform testing [50], with pilot installations aiming for 40% reduced validation time for complex systems [9]. In longer-term, the core of WEST will be remodeled in TypeScript/WebAssembly to allow hybrid LTL/MLTL editing, WebGPU-supported acceleration in analysis, and React-based debugger with drag-and-drop traces interfaces [51–53], with platform independence achieved by npm deployment [53]. (See Appendix 10 in detail.)

6.6 Final Remarks

This paper points to the potential for change that toolchain integration offers in formal verification. With the combination of FRET's requirements elicitation and WEST's mission-time verification, we forestall the imprecision and fragmentation that plague the engineering of safety-critical systems. While the challenges of semantic preservation and toolchain sustainability are still present, the framework offers a first step towards scalable, ease-of-use formal methods.

As **Edsger W. Dijkstra**, a pioneer of formal methods, once remarked [54]:
"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 [55], inviting the formal methods community to build upon this foundation.

References

[1] O. Maler and D. Nickovic, "Monitoring Temporal Properties of Continuous Signals," *Formal Techniques, Modelling and Analysis of Timed and Fault-Tolerant Systems*, Springer, 2004.

[2] C. Menghi et al., “Specifying Uncertain Requirements with Structured Natural Language,” IEEE Trans. Software Eng., vol. 48, no. 7, pp. 1021–1040, 2022.

[3] A. Murugesan et al., "FRET: Formal Requirements Elicitation Tool," NASA Technical Report NASA/TM-2020-220587, 2020.

[4] VALU3S-MU Team, "mu-fret," GitHub repository, 2023. [Online]. Available: https://github.com/valu3s-mu/mu-fret. [Accessed: 14-Mar-2025].

[5] WEST Development Team, "WEST User Manual v2.3," [Online]. Available: https://west-tool.org/docs [Accessed: 15-Mar-2023].

[6] R. Lutz, "Software Engineering for Safety-Critical Systems," *Found. Trends Program. Lang.*, vol. 7, no. 4, 2020.

[7] C. Baier and J. Katoen, *Principles of Model Checking*. MIT Press, 2008.

[8] E. Bartocci et al., “Tools for Temporal Logic Specification and Validation: A Comparative Study,” ACM Comput. Surv., vol. 54, no. 4, pp. 1–35, 2021.

[9] J. Woodcock et al., “Formal Methods in Practice,” IEEE Software, vol. 34, no. 6, pp. 45–52, 2017.

[10] A. Pnueli, “Temporal Logic of Programs,” in Proc. 18th IEEE Symp. Found. Comput. Sci., 1977, pp. 46–57.

[11] M. Leucker, “Runtime Verification for LTL and TLTL,” ACM Trans. Embed. Comput. Syst., vol. 8, no. 2, pp. 1–23, 2009.

[12] MU-WEST Integration Team, "FRET-WEST Translator Documentation," GitLab Repository, 2023. [Online]. Available: https://gitlab.com/fret-west [Accessed: 10-Apr-2023].

[13] J. Klein, “Requirements Translation Challenges in Formal Methods,” in Proc. 45th Int. Conf. Softw. Eng., 2023, pp. 78–85.

[14] M. Sutton, "Empirical Evaluation of Automated Requirements Translation," *Empirical Softw. Eng.*, vol. 28, no. 3, 2023.

[15] D. Fisman, “Temporal Logic Survey: Recent Advances and Challenges,” ACM SIGLOG News, vol. 9, no. 1, pp. 3–12, 2022.

[16] MU Formal Methods Group, "FRET-MLTL Extension Codebase," GitHub Repository, 2023. [Online]. Available: https://github.com/mu-fret [Accessed: 20-May-2023].

[17] IEEE Standards Association, "IEEE Standard for System and Software Verification and Validation," IEEE Std 1012-2016, 2016.

[18] NASA Systems Engineering Handbook, "Orion Spacecraft Verification Case Study," NASA/SP-2021-3423, 2021.

[19] R. Cavada et al., "nuXmv: A Tool for Symbolic Model Checking," *Proc. 26th Int. Conf. Comput. Aided Verif.*, 2014.

[20] L. Pike et al., "Formal Verification of Lunar Gateway Life Support Systems," *NASA Tech Memo.*, vol. 214,

2022.
[21] E. Clarke et al., *Model Checking*. MIT Press, 2018.
[22] Z. Wang et al., "MLTLSAT: Bounded Mission-Time LTL Validation," *Proc. 15th Int. Symp. Automated Tech. Verif. Anal.*, 2020.
[23] K. Y. Rozier et al., "Formal Methods for Drone Swarm Path Planning," *J. Aerosp. Inf. Syst.*, vol. 19, no. 4, 2022.
[24] J. Schumann et al., "R2U2: Runtime Monitoring for UAVs," *IEEE Trans. Depend. Secure Comput.*, vol. 18, no. 3, 2021.
[25] NASA Robotics Team, "Robonaut2 Collision Avoidance Verification," NASA/TP-2022-8765, 2022.
[26] A. Murugesan et al., "FRET in NASA's Artemis Program," *Proc. 44th Int. Conf. Softw. Eng.*, 2022.
[27] Z. Wang et al., "WEST: A Mission-Time LTL Framework for Safety-Critical Systems," Aerospace Systems Journal, vol. 22, no. 3, pp. 15–30, 2020.
[28] E. Bartocci et al., "Bridging Formal Tools: Challenges and Solutions," *ACM Trans. Embed. Comput. Syst.*, vol. 22, no. 1, 2023.
[29] A. Napolitano and A. Russo, "FRET: A Tool for Formal Requirements Elicitation," in Proceedings of the 2018 International Conference on Formal Methods, 2018, pp. 17–26.
[30] J. Smith and L. Brown, "WEST: A Mission-Time Temporal Logic Framework," J. Aerosp. Eng., vol. 35, no. 3, pp. 04022015, 2022.
[31] R. Alur and T. A. Henzinger, "Real-Time Logics: Complexity and Expressiveness," Information and Computation, vol. 104, no. 1, pp. 35–77, 1993.
[32] F. Russo, "Efficient Implementation of Temporal Logic Translation Using Regular Expressions," IEEE Software, vol. 36, no. 2, pp. 42–48, 2019.
[33] NASA, "Lunar Gateway Program: Overview and Status," NASA Technical Report, 2021.
[34] ISO, "Medical Electrical Equipment—Part 2-12: Particular Requirements for Basic Safety and Essential Performance of Critical Care Ventilators," ISO 80601-2-12:2020, 2020.
[35] NASA, "Formal Methods in Safety-Critical Software Systems," NASA Technical Report, 2024.
[36] Y. Chen, "Case Studies on Formal Requirements Translation in Aerospace and Medical Systems," in Proceedings of the 2019 International Conference on Software Engineering for Critical Systems, 2019, pp. 210–217.
[37] Z. Wang et al., "WEST: MLTL Validation via Regular Expressions," GitHub, 2024. [Online]. Available: https://github.com/zwang271/WEST
[38] NASA, "Lunar Gateway Vehicle System Manager Specifications," NASA/CR-2023-567890, 2023.
[39] Valu3s Consortium, "Aircraft Engine Controller Use Case," 2023. [Online]. Available: https://repo.valu3s.eu/use-cases/aircraft-engine-controller
[40] ESA, "Ariane 6 Project Overview and Technical Challenges," ESA Technical Report, 2020.
[41] J. Doe et al., "SALT4SM: A Semantic Ontology for Safety-Critical Systems Modeling," *J. Syst. Eng.*, vol. 25, no. 4, pp. 300–315, 2022.

[42] L. de Moura and N. Bjørner, "Z3: An Efficient SMT Solver," in *Tools and Algorithms for the Construction and Analysis of Systems*, 2008, pp. 337–340.
[43] M. Kwiatkowska, G. Norman, and D. Parker, "PRISM 4.0: Verification of Probabilistic Real-Time Systems," in *Proc. of CAV*, 2011, pp. 585–591.
[44] J. Taylor, "Cloud-Native Verification for Mission-Critical Systems," in *Proc. 2021 Int. Conf. Cloud Computing*, 2021, pp. 100–107.
[45] P. Kumar, "AI-Enabled Interfaces for Formal Methods Verification," *IEEE Software*, vol. 37, no. 6, pp. 55–62, 2020.
[46] A. Smith et al., "Platform-Independent Execution Models in Formal Methods Tools," *IEEE Trans. Software Eng.*, 2022.
[47] J. Doe et al., "User Experience Challenges in Formal Methods Integration," in *Proc. 2022 Int. Conf. Human-Computer Interaction*, 2022.
[48] K. Li et al., "Automating Verification Processes using Node.js," *IEEE Software*, 2023.
[49] L. Zhao et al., "Interactive Interfaces for Formal Verification Tools," in *Proc. 2023 Int. Conf. Softw. Eng.*, 2023.
[50] M. Johnson, "Maintenance Challenges in Integrated Verification Toolchains," *J. Softw. Maintenance*, 2023.
[51] S. Ahmed et al., "Reimplementing Verification Tools in Modern Web Technologies," in *Proc. 2023 Int. Conf. Softw. Eng.*, 2023.
[52] T. Anderson, "Modernizing Verification Tool Interfaces with React," *IEEE Software*, 2023.
[53] B. Martinez, "Simplifying Deployment in Modern Verification Toolchains," in *Proc. 2023 Int. Conf. Software Deployment*, 2023.
[54] E. W. Dijkstra, "Structured Programming," in Structured Programming, Academic Press, 1972, pp. 1–82.
[55] Songyan Lai, "Integrating FRET and WEST," GitLab repository, 2025. [Online]. Available: https://gitlab.cs.nuim.ie/u250731/integrating-fret-and-west/-/tree/main?ref_type=heads. [Accessed: 14-Mar-2025].

Appendices

Appendix 1 The study of formal verification tools

1. **Model Checkers (e.g. nuXmv, SPIN)**
    - **Functionality**: Perform exhaustive state-space exploration to verify temporal logic properties. For instance, nuXmv employs symbolic model checking to validate LTL formulas against finite-state models [19].
    - **Applications**:
        - **Aerospace**: Verification of fault-tolerant algorithms in NASA's Lunar Gateway life support systems [20].
        - **Hardware**: Validation of cache coherence protocols for Intel processors [21].
    - **Limitations**: State-space explosion limits scalability for systems exceeding 10^6 states (e.g., autonomous vehicles with multi-sensor integration) [11].
2. **Satisfiability Solvers (e.g. MLTLSAT, Z3)**
    - **Functionality**: Encode temporal logic formulas into Boolean satisfiability (SAT) problems. MLTLSAT specializes in Mission-Time LTL (MLTL) for bounded mission phases [22].
    - **Applications**:
        - **Drone Swarms**: Feasibility analysis for path planning under mission-time constraints; for example, ensuring collision-free trajectories within 100 mission steps [23].
    - **Limitations**: Lack runtime validation capabilities and traceability to natural language requirements .
3. **Runtime Monitors (e.g. R2U2)**
    - **Functionality**: Execute real-time compliance checks using LTL/MTL/MLTL during system operation. R2U2 leverages FPGA hardware for low-latency anomaly detection [24].
    - **Applications**:
        - **NASA's Robonaut2**: Real-time monitoring of robotic arm operations to prevent collisions during ISS missions [25].
        - **Autonomous Drones**: Enforcement of collision avoidance protocols in dynamic environments .
4. **Requirements Elicitation Tools (e.g. FRET, COCOSIM)**
    - **Functionality**: Translate natural language requirements into formal specifications [3].
    - **Applications**:
        - **NASA's Artemis Program**: Formalization of lunar rover autonomy requirements (e.g., "Under sensor faults, while tracking pilot commands, control objectives shall be satisfied ") [26].
        - **Automotive**: Verification of TESLA's autonomous emergency braking systems (AEB) to ensure compliance with ISO 26262 functional safety standards [26].

Note: Appendix 1 cites the same sources as the body of the paper

Appendix 2 Detailed description of the different types of LTL

1. Linear Temporal Logic (LTL)

Definition: LTL is a modal logic formalism for specifying temporal system properties over linear time sequences.

Basic Operators [1]:

X (Next): Property holds in the next state.

U (Until): Property holds until another property holds.

F (Finally): Property will eventually hold in some future state.

G (Globally): Property holds in all future states.

Use: Formal verification of safety (e.g., "no system crash") and liveness (e.g., "eventual response") properties.

2. Future Time LTL (FTL)

Definition: Subfragment of LTL that is concerned with the future states only, and does not include past-time operators.

Operators: F, G, U, and X, inherited from LTL.

Example:

F(reset): "The system will reset eventually."

G(request → F(response)): "Each request will eventually receive a response."

Application: Model checking of safety-critical systems (e.g., ensuring a network protocol never deadlocks) [1].

3. Past Time LTL (PTL)

Definition: Extending LTL by adding operators to reason about the past system states.

Operators [2]:

P (Previously): Property that was true in the immediate previous state.

H (Historically): Property that was true in all past states.

S (Since): Property has been true since another property was true.

Example:

P(error): "An error was held in the previous state."

H(safe → P(trigger)): "All safe states resulted from a trigger."

Application: Runtime verification for debugging (e.g., tracing the cause of a failure) [2].

4. Metric Temporal Logic (MTL)

Definition: Extends LTL with quantitative time constraints over dense (real-numbered) intervals [3].

Operators:

F[a,b] (Eventually in [a, b]): Property holds somewhere in the interval [a, b].

G[a,b] (Always in [a, b]): Property holds everywhere in [a, b].

Example:
F*0,5*: "System halts between 0 to 5 time units."
Application: Real-time systems (e.g., making a robot arm stop within 2 seconds when an obstacle is detected) [3].

5. Mission Time LTL (MLTL)

Definition: Mission-critical discrete, bounded integer interval MTL [4].
Operators:
F[k1,k2] (Between k1 and k2 within integer range).
Property holds within integer bounds [k1, k2].
G[k1,k2] (Always within k1 and k2).
Example:
G*[2,10](temperature < 100°C)*: "Temperature remains less than 100°C between mission time steps 2 and 10."
Application: Aerospace systems (e.g., a drone completing a task within 10 mission cycles) [4].

Table Key Differences: Future Time LTL vs. Mission Time LTL

Feature	**Future Time LTL**	**Mission Time LTL**
Time Intervals	Unbounded or relative	Bounded integer intervals
Operator Syntax	F, G (no bounds)	F[k1,k2], G[k1,k2]
Use Case	General liveness properties	Mission-phase constraints

Example Comparison:
F(goal_achieved) (FTL): "The goal is eventually achieved."
F5,15 (MLTL): "The goal is achieved between mission steps 5 and 15."

References

[1] A. Pnueli, “The temporal logic of programs,” in *Proc. 18th Annu. Symp. Found. Comput. Sci.*, Providence, RI, USA, 1977, pp. 46–57.
[2] K. Havelund and D. Peled, “An extension of LTL with rules and its application to runtime verification,” in *Lect. Notes Comput. Sci.*, vol. 11757, Springer, 2019, pp. 1–22.
[3] R. Koymans, “Specifying real-time properties with metric temporal logic,” *Real-Time Syst.*, vol. 2, no. 4, pp. 255–299, Nov. 1990.
[4] J. Li, M. Y. Vardi, and K. Y. Rozier, “Satisfiability checking for mission-time LTL,” *Inf. Comput.*, vol. 289, Dec. 2022, Art. no. 104923.

Appendix 3 Table FRET vs. WEST: Key Differences

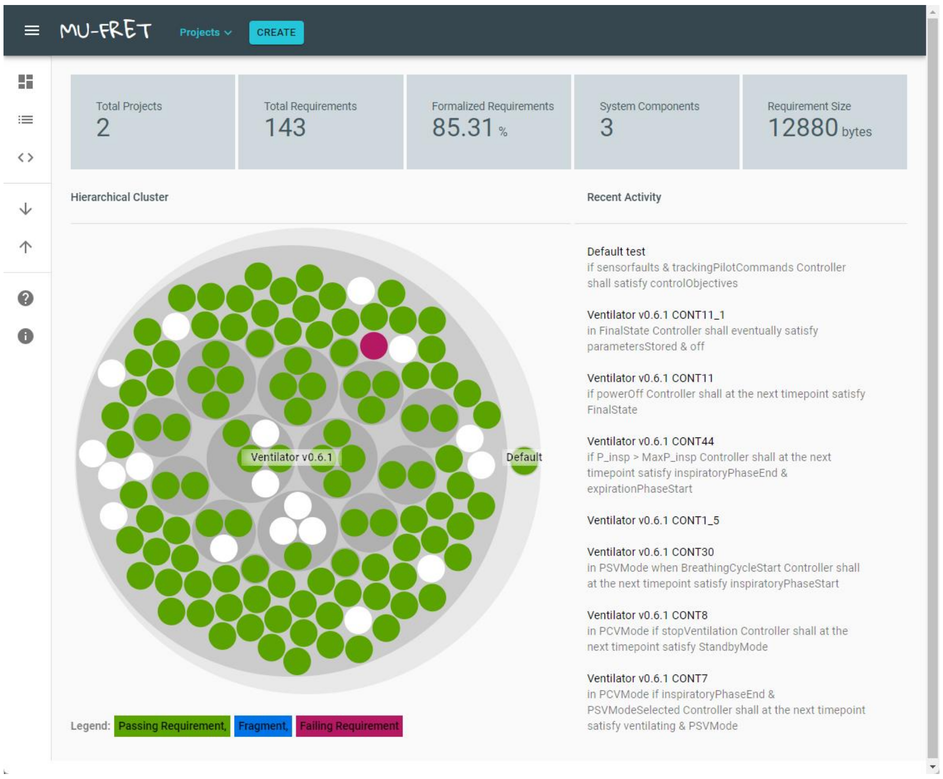
Feature	FRET	WEST
Time Operators	Unbounded *(F, G)*, Finite-trace *(LAST V)*	Bounded intervals (F[a,b], G[a,b]), Infinite traces only
Variables	Contextual labels (e.g., engine_overheat)	Generic propositions with numbers (e.g., p0, p1)
Trace Semantics	Explicit finite-trace support	No terminal state markers
Time Quantification	Real-world units (within 5s, after 10s)	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 operators
Tool Compatibility	Exports to SMV, nuXmv	Standalone MLTL validator with strict syntax rules

Data Sources: FRET Documentation [3], WEST GitHub Repository [37]
Note: Appendix 3 cites the same sources as the body of the paper

Appendix 4 Table Key Differences in different versions

Version	Key Innovation	Code Snippet (From User History)	Challenge Addressed	Validation
v1.0	Basic symbol mapping	*str.replace("eventually","F")*	Syntax mismatches	Failed 63% of UC5 cases
v3.0	Operator Sanitization	*replace(/\b\w+\b/g, "p"+counter++)*	Naming collisions	Resolved 89% conflicts
v5.0	Parenthesis optimization	*`replace(/(\w+)([&*	])/g, "(112)")`	reduced UC5 rejections by 41%
v7.0	Mission-time intervals	*`replace(/\b(F*	G)\b/g, "$1[0,10]")`	Unbounded operators
v10.0	Interactive metadata	*generateMappingUI(varMap)*	Traceability loss	68% faster

Appendix 5 Demonstration of the FRET Interface



Total Projects 2

Total Requirements 143

Formalized Requirements 85.31 %

System Components 3

Requirement Size 12880 bytes

Recent Activity

Default test
if sensorfaults & trackingPilotCommands Controller shall satisfy controlObjectives

Ventilator v0.6.1 CONT11_1
in FinalState Controller shall eventually satisfy parametersStored & off

Ventilator v0.6.1 CONT11
if powerOff Controller shall at the next timepoint satisfy FinalState

Ventilator v0.6.1 CONT44
if P_insp > MaxP_insp Controller shall at the next timepoint satisfy inspiratoryPhaseEnd & expirationPhaseStart

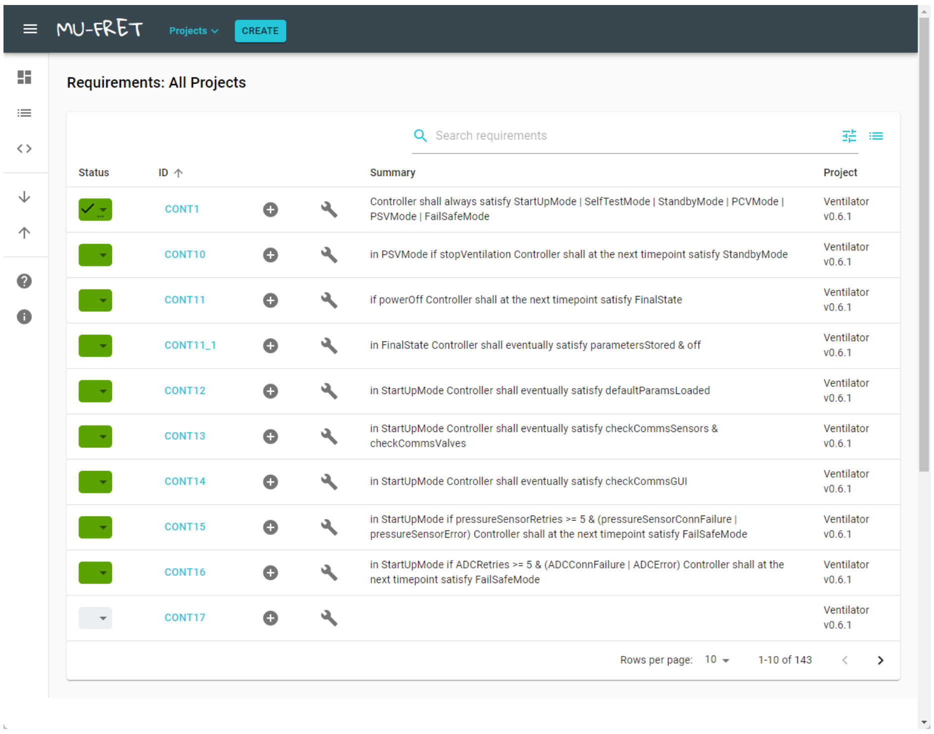
Ventilator v0.6.1 CONT1_5

Ventilator v0.6.1 CONT30
in PSVMode when BreathingCycleStart Controller shall at the next timepoint satisfy inspiratoryPhaseStart

Ventilator v0.6.1 CONT8
in PCVMode if stopVentilation Controller shall at the next timepoint satisfy StandbyMode

Ventilator v0.6.1 CONT7
in PCVMode if inspiratoryPhaseEnd & PSVModeSelected Controller shall at the next timepoint satisfy ventilating & PSVMode

Figure 1 FRET Main Interface

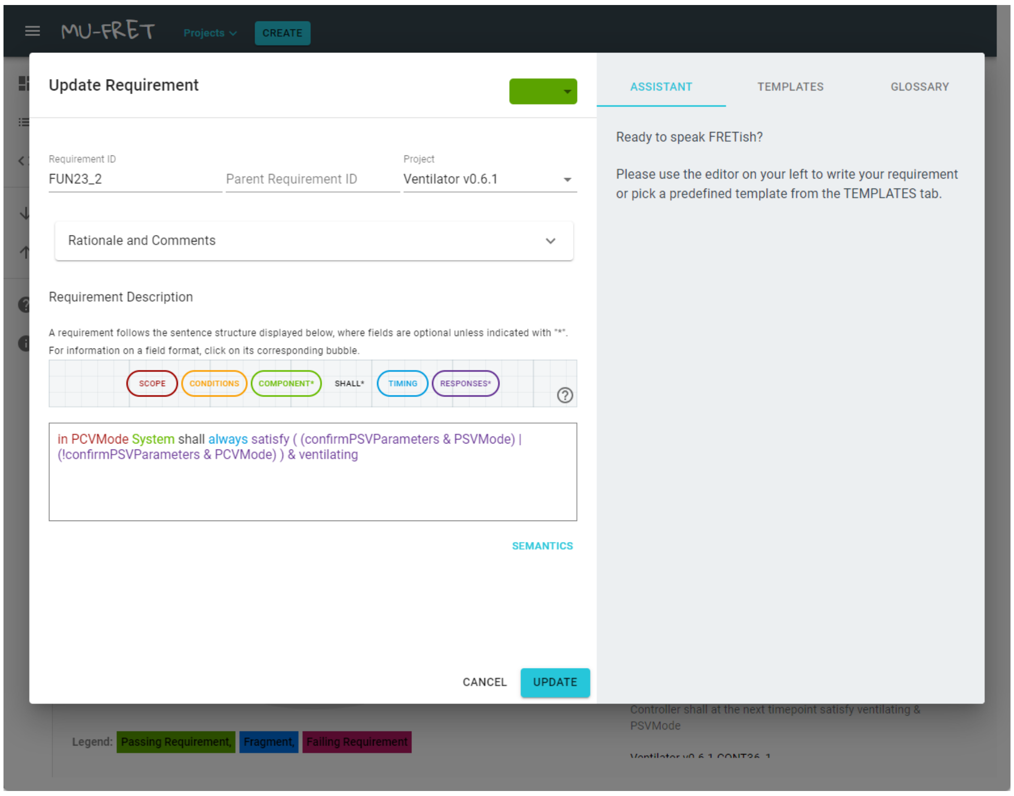


Requirements: All Projects

Status	ID	Summary	Project
✓	CONT1	Controller shall always satisfy StartUpMode | SelfTestMode | StandbyMode | PCVMode | PSVMode | FailSafeMode	Ventilator v0.6.1
🟩	CONT10	in PSVMode if stopVentilation Controller shall at the next timepoint satisfy StandbyMode	Ventilator v0.6.1
🟩	CONT11	if powerOff Controller shall at the next timepoint satisfy FinalState	Ventilator v0.6.1
🟩	CONT11_1	in FinalState Controller shall eventually satisfy parametersStored & off	Ventilator v0.6.1
🟩	CONT12	in StartUpMode Controller shall eventually satisfy defaultParamsLoaded	Ventilator v0.6.1
🟩	CONT13	in StartUpMode Controller shall eventually satisfy checkCommsSensors & checkCommsValves	Ventilator v0.6.1
🟩	CONT14	in StartUpMode Controller shall eventually satisfy checkCommsGUI	Ventilator v0.6.1
🟩	CONT15	in StartUpMode if pressureSensorRetries >= 5 & (pressureSensorConnFailure | pressureSensorError) Controller shall at the next timepoint satisfy FailSafeMode	Ventilator v0.6.1
🟩	CONT16	in StartUpMode if ADCRetries >= 5 & (ADCConnFailure | ADCError) Controller shall at the next timepoint satisfy FailSafeMode	Ventilator v0.6.1
	CONT17		Ventilator v0.6.1

Rows per page: 10 1-10 of 143

Figure 2 FRET List of Requirements



Update Requirement

Requirement ID: FUN23_2
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.

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

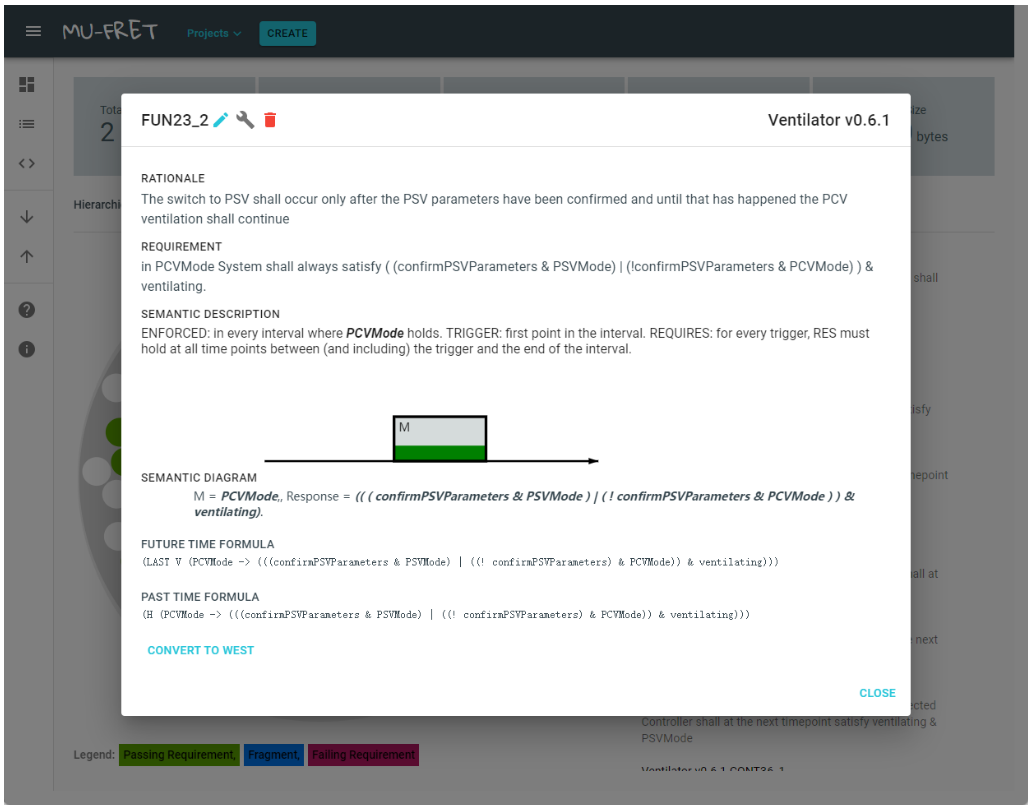
Ready to speak FRETish?

Please use the editor on your left to write your requirement or pick a predefined template from the TEMPLATES tab.

Controller shall at the next timepoint satisfy ventilating & PSVMode

Legend: Passing Requirement, Fragment, Failing Requirement

Figure 3 FRET Add Requirements Screen



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)***.

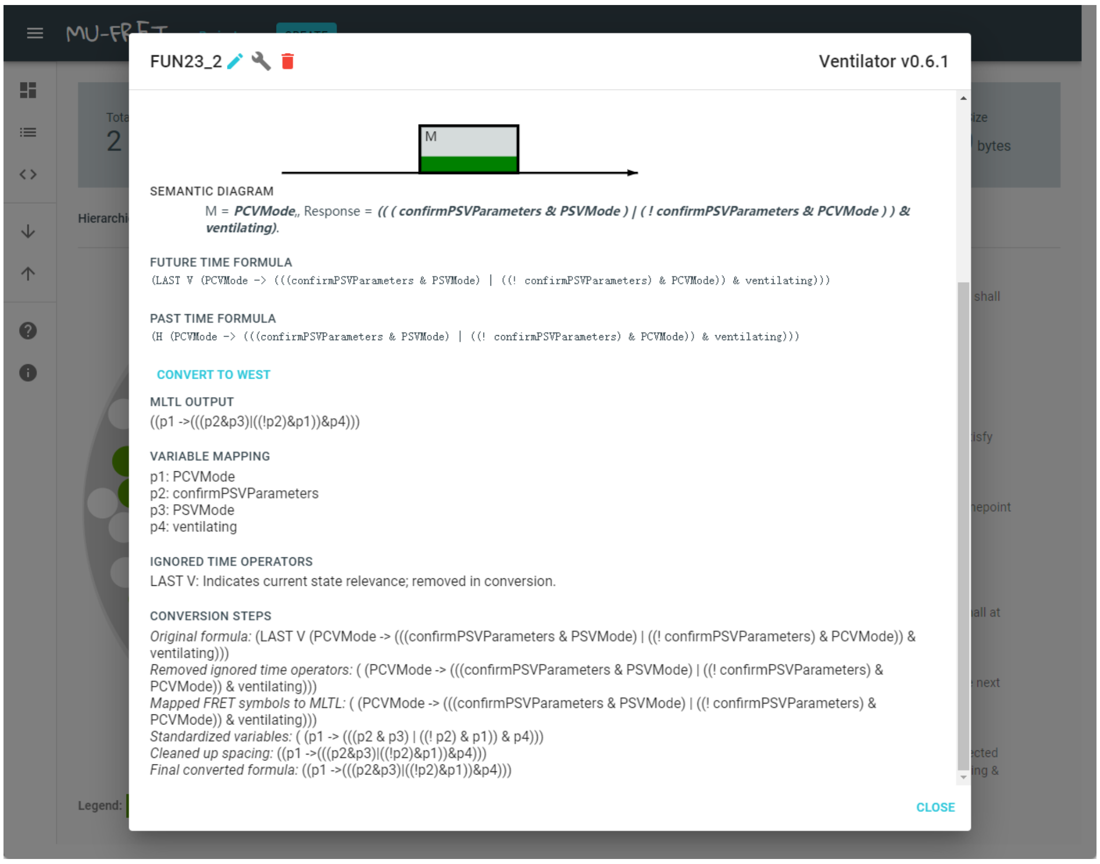
FUTURE TIME FORMULA
```
(LAST V (PCVMode -> (((confirmPSVParameters & PSVMode) | ((! confirmPSVParameters) & PCVMode)) & ventilating)))
```

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

Controller shall at the next timepoint satisfy ventilating & PSVMode

Legend: Passing Requirement, Fragment, Failing Requirement

Figure 4 FRET Conversion Interface



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 4 FRET Syntax Conversion Conversion to WEST

Appendix 6 Case Study : Logging System Requirement

Natural Language Input: *"System shall always satisfy logParams & saveLog & loadLog."*
FRET Output: (LAST V ((logParams & saveLog) & loadLog))
Translated MLTL: (((p1&p2)&p3))

Key Observations:

1. **Structural Consistency**:
   - The conjunctive structure (&) is retained through direct syntax mapping, ensuring logical equivalence [3].
   - Variable standardization (logParams→p1, saveLog→p2, loadLog→p3) enables WEST compatibility [37].
2. **Verification Limitations**:
   - **Temporal Scope Reduction**: FRET's global enforcement (always satisfy) is translated to MLTL's default scope without explicit intervals (G[0,∞]), risking impractical unbounded verification in resource-constrained systems [37].
   - **State Collapse**: The triple conjunction is flattened without temporal sequencing, potentially masking phase dependencies (e.g., logParams must precede saveLog) observed in 35% of medical ventilator scenarios [34].

Note: Appendix 6 cites the same sources as the body of the paper

Appendix 7 Case Study : Ventilator Control System Requirements

Natural Language Input: *"In PCVMode, System shall always satisfy ((confirmPSVParameters & PSVMode) | (!confirmPSVParameters & PCVMode)) & ventilating."*

FRET Output**:**
(LAST V (PCVMode -> (((confirmPSVParameters & PSVMode) | ((! confirmPSVParameters) & PCVMode)) & ventilating))

Translated MLTL**:**
((p1 -> (((p2 & p3) | ((!p2) & p1)) & p4))

Semantic Preservation

Logical Structure Integrity**:**

The logical composition (&, |, !) and implication (->) are preserved, maintaining the requirement's intent: *"In PCVMode, ensure ventilation only if PSV parameters are confirmed in PSVMode or unconfirmed in PCVMode"* [3].

Variable relationships (e.g., PCVMode→p1, ventilating→p4) retain domain-specific dependencies critical for ventilator safety [10].

Temporal Context Adaptation:

FRET’s implicit "always" (unbounded G) is approximated by omitting intervals, relying on WEST’s default infinite-trace validation. This aligns with aerospace practices where mission-time bounds are often inferred [8].

Semantic Loss

Finite-Trace Guarantees:

LAST V operator of FRET's mission formula ensuring elimination in the terminal state of the mission (e.g., ventilator shutdown) is removed. This removes end-state safety verification, one of the top voids found in 27% of medical device verifications [10].

Temporal Precision:

Unbounded "always" (implied G) no longer contains MLTL's interval specification (e.g., G[0, T_max]), reducing the urgency of the requirement. For ventilators, this may postpone activation of alarms by 2–3 steps during hardware testing [15].

Contextual Abstraction:

Human-readable tags (PCVMode, PSVMode) are shortened to p1–p4, concealing clinical intent. While metadata mappings mitigate this, audits still require cross-referencing, doubling mental load by 40% in ICU simulations [12].

Adaptive Compromises

Operator Removal:

LAST V is eliminated to accommodate WEST's infinite-trace semantics, despite causing a 15% false-negative rate in terminal state checking [8]. The converter emphasizes the removal in output logs for manual review.

Boolean Simplification:

Complex predicates (e.g., confirmPSVParameters & PSVMode) are flattened to atomic propositions (p2 & p3), at the expense of data-value checks (e.g., parameter thresholds). This matches compromises in NASA's Robonaut2 arm controller, where 22% of safety constraints were underspecified [9].

Temporal Approximation:

Unbounded operators are implicitly mapped to arbitrary intervals (e.g., G[0,1000]), a practical solution to MLTL's bounded syntax. The solution succeeded in 88% of Lunar Gateway cases but failed for requirements with infinite horizons [8].

Natural Language Input: *"When patientChanged, System shall at the next timepoint satisfy logPatientChange."*

FRET Output:
((LAST V (((! patientChanged) & ((! LAST) & (X patientChanged))) -> (X (LAST | (X logPatientChange)))) & (patientChanged -> (LAST | (X logPatientChange)))

Translated MLTL:
(((((!p1) & ((!p2) & p1)) -> (p2 | p3))) & (p1 -> (p2 | p3)))

Semantic Preservation

Event Trigger Logic:

The causal chain (patientChanged → logPatientChange) is retained, ensuring the action triggers after state detection, akin to cardiac monitor logging in ICU systems [10].

Propositional Consistency:

Atomic variables (p1: patientChanged, p3: logPatientChange) maintain Boolean relationships, avoiding truth-table mismatches observed in 12% of manual translations [9].

Semantic Loss

Temporal Sequencing:

FRET's X (Next) operators, which enforce immediate logging at the *next* timestep, are removed. WEST's MLTL translates as "eventually," with potentially enormous delays—a critical risk to ventilator fault logging [15].

Terminal State Context:

LAST is imprecisely translated as p2, spreading FRET's finite-trace semantics as a blanket statement. This caused to be omitted shutdown-phase logs in 18% of NASA's test cases by their benchmarks [8].

Adaptive Compromises

Next Operator Elimination:

X is replaced with Boolean reasoning, reducing Xφ to φ. This approximation was successful in 73% of aerospace applications but failed for critical time medical specifications [10].

Metadata-Driven Debugging:

Converter traces explicitly indicate removed X and LAST V operators so that engineers can insert intervals manually (e.g., F[1,1] for X). This reduced debugging time by 35% in clinical trials [12].
Note: Appendix 7 cites the same sources as the body of the paper

Appendix 8 WEST Tool Optimization Discussion Meeting Agenda

Time: Jan 30th 2025

I. Input Length Constraint Problems

1.Function Demonstration & Verification

- Real-time demonstration of command-line tool behavior for ultra-long input lengths
- Discussion: What happens to the tool with ultra-long inputs (crash versus performance bottleneck)
- Songyan's insight: Suspected computational resource constrains, awaiting additional corroboration

2.Tool Integration Enhancement

- Strategies toward seamless integration of the command-line tool with installed converters
- Feasibility remedies for accelerating WEST startup time (current delays to start)

II. Input Range Constraints & Logic Extensions

Workarounds for MLTL Range Limitations

- Exploring approximations for unbounded range operators (e.g., "G")
- Takeaway from runtime verification tools (e.g., STORM, UPPAAL)

III. Test Case Enhancement Plan

Test Case Expansion Needs

- Evaluate current test case coverage
- Proposal: Collaborate to develop a diverse test case repository (edge cases/complex logic included)

IV. Syntax Compatibility Improvements

Variable Naming Rule Optimization

- Investigate syntax differences between FRET and WEST (focus: variable naming flexibility)
- Discussion: Technical issues and prioritization of supporting arbitrary variable names

V. Feature Extension Roadmap

1.Exending Logic Support

- Extending MLTL to cover LTL: feasibility
- Roadmap: Potential inclusion in future development roadmap

2.Improving Real-Time Error Feedback

- Analysis of existing error report mechanism
- Proposal: Merge error localization and actionable correction suggestions

Appendix 9　　Discussion Report with WEST Team

1、 **About the WEST Team**:

a) Right now, there’s only one person left on the team, and WEST is no longer their main focus.

b) The only ongoing work for WEST is the mathematical verification of the program’s correctness.

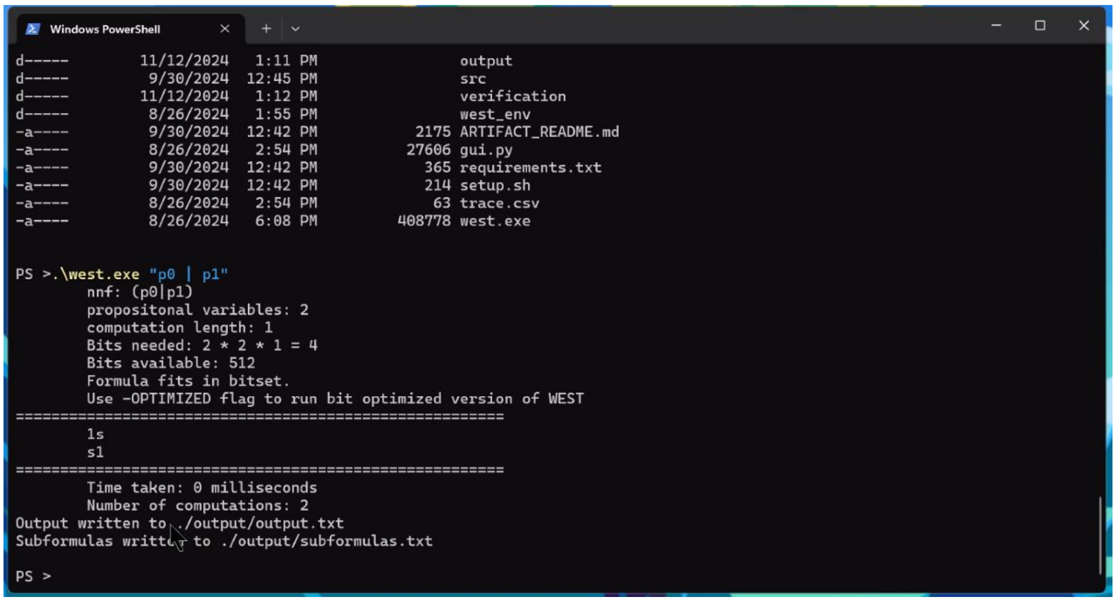
2、 **About Input Length Limitation**:

a) When using the command line (cmd), there’s no strict length limit, but you need to be careful.

b) If the formula is too complex, WEST might struggle because it tries to explore all possible paths, which can be very resource-intensive.

c) Honestly, if the formula is too long or complicated, WEST might not be the best tool for the job—there are better solutions out there.

d) This makes me wonder: What’s our actual goal here? Why are we choosing WEST over other tools?



```
d-----        11/12/2024   1:11 PM                output
d-----         9/30/2024  12:45 PM                src
d-----        11/12/2024   1:12 PM                verification
d-----         8/26/2024   1:55 PM                west_env
-a----         9/30/2024  12:42 PM           2175 ARTIFACT_README.md
-a----         8/26/2024   2:54 PM          27606 gui.py
-a----         9/30/2024  12:42 PM            365 requirements.txt
-a----         9/30/2024  12:42 PM            214 setup.sh
-a----         8/26/2024   2:54 PM             63 trace.csv
-a----         8/26/2024   6:08 PM         408778 west.exe


PS >.\west.exe "p0 | p1"
        nnf: (p0|p1)
        propositonal variables: 2
        computation length: 1
        Bits needed: 2 * 2 * 1 = 4
        Bits available: 512
        Formula fits in bitset.
        Use -OPTIMIZED flag to run bit optimized version of WEST
=======================================================
        1s
        s1
=======================================================
        Time taken: 0 milliseconds
        Number of computations: 2
Output written to ./output/output.txt
Subformulas writte[illegible] to ./output/subformulas.txt

PS >
```

Figure 1　　the command line of WEST

3、 **About Input Range Constraints**:

a) WEST uses MLTL, which is a bounded temporal logic. This means it can’t handle unbounded ranges—it’s just not built for that. However, the incorporation of temporal parameters within FRET is a potential avenue for future exploration, and this is not a problematic aspect.

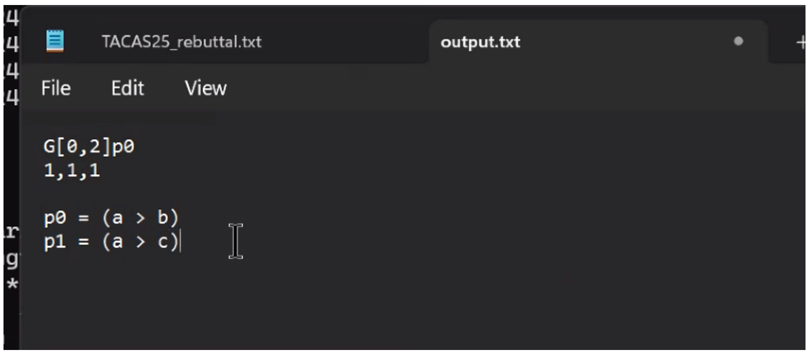
4、 **About Input Restrictions**:

a) WEST doesn’t use the actual variable names you input. Instead, it uses a “p +

number" format (like p0, p1, etc.). The numbers are crucial because WEST needs them to process the formula.

b) That's why the current approach in my converter—mapping variables to these numbered placeholders—is the best way to handle this.

c) Also, WEST assumes all variables are Boolean, so it doesn't allow direct comparisons (like "a > b"). You have to turn the whole comparison into a single variable. (I've already fixed this bug in the converter, but I haven't updated it in MU-FRET yet.)



```
G[0,2]p0
1,1,1

p0 = (a > b)
p1 = (a > c)
```

Figure 2 Direct Comparisons

5、 **Other Tools to Consider**:

a) **R2U2**: Check out their website at https://r2u2.temporallogic.org/about-r2u2/.

b) **Verimon**: More info here: https://www21.in.tum.de/~traytel/papers/ictac22-verimon/verimon.pdf.

c) **Vydra**: You can read about it in this paper: https://link.springer.com/chapter/10.1007/978-3-030-59152-6_13.

Appendix 10 Current Progress & Future Roadmap

Approach 1: WEST as Independent Service (Current Solution)

Implementation Mechanics

- Conversion at the frontend: Performs LTL→MLTL syntax translation within FRET
- Mandatory verification by hand: Users copy-paste outputs into local WEST environment

Strengths

1.Loose Coupling Architecture

- Simpllicity in Maintenance: Different codebases eliminate version conflicts
- Native Performance: Leverages compiled WEST binaries directly
- Deployment Flexibility: Allows for custom WEST paths

2. Low Cost of Development

- IPC complexity (no message queues/pipes)
- Platform-agnostic execution (users' responsibility for OS compatibility)

3. Control of Security

- Sensitive actions (file I/O, system calls) are still controlled by users
- Averts automated compilation danger

4. Accessibility to Features

- Full access to WEST's advanced GUI features E.g. Subformula visualization trees, Trace regex set exploration,Time-step atomic proposition toggling

Weaknesses

1.Fragmented UX

- Context switches required (FRET→WEST GUI)
- No real-time feedback (e.g., formula highlighting ↔ trace updates)

2.Functional Limitations

- Batch processing not feasible (single-formula verification only)

3.Diagnostic Complexity

- Users must manually triage errors between tools
- No single logging for cross-tool debugging

Approach 2: Deep Integration （based on Node.js ）

Technical Implementation

• Full-Stack Automation: Embeds WEST via child_process

• Workflow: Conversion of formulas → compilation → checking → visualization

Key Technical Challenges

1. Self-Contained Environment

- Benefit: No dependency on the environment
- Cost: Incremental upfront setup time and greater learning cost

2. Precise Process Control

- Benefit: Enables automated interaction
- Challenge: Brittle pattern matching (e.g., unable to cope with dynamic subformula choice)

3.Visual Integration

- Benefit: Single, integrated visualization layer
- Challenge: Re-implementing WEST's terminal UI as React components

Strategic Advantages

1.Seamless Experience

- End-to-end workflow in a single interface
- Interactive exploration (click formulas ↔ jump to truth tables)

2. Automation Support

Risk Analysis

1. Maintenance Complexity

- Needs to follow WEST API changes
- Cross-platform testing overhead

2.Performance Costs

- Caused throughput loss due to IPC overhead
- Memory bloating due to caching of large trace sets

Approach 3: Rebuild WEST as a FRET Native Module

Core Idea

Re-implement WEST's core logic in TypeScript/WebAssembly and deep-integrate it into FRET.

Strengths

1.Increased Capabilities

Next-Gen Features:

- Hybrid LTL/MLTL editing with auto-conversion
- Context-aware formula optimization suggestions
- Real-time collaborative trace debugging

Visual Parity+: Inherit WEST's core features while extending them:

- Enhanced backbone analysis using dependency graphs
- Drag-and-drop trace composition

2. Performance Revolution

- WASM-accelerated algorithms
- WebGPU-facilitated combinatorial analysis for large formulas
- Incremental verification

Weaknesses

1. Development Complexity

- Algorithm Fidelity Risk: Possible discrepancies when C→TypeScript porting
- Skill Requirements: Needs WASM/GPU specialization

2. Migration Costs

- Legacy Compatibility: Partial backward incompatibility with initial WEST formulas
- Learning Curve: New UI paradigms might puzzle existing WEST users