An Integrated Ecosystem for Mission-time LTL Specification Elicitation and Validation

Zili Wang[1], Alec E. Rosentrater[1], Alexis Aurandt[2], Christopher Johannsen[1], and Kristin Yvonne Rozier[1]

[1] Iowa State University, Ames, Iowa 50011, USA
{ziliw1, alecrose, cgjohann, kyrozier}@iastate.edu
[2] Collins Aerospace, USA
Alexis.Aurandt2@collins.com
MLTL.TEMPORALLOGIC.ORG

Abstract. Formal methods often stall not on verification algorithms, but on the harder task of turning informal engineering intent into precise, trustworthy requirements. Mission-time Linear Temporal Logic (MLTL) strikes a practical balance for this setting: it is expressive enough to capture many common reactive-system requirements while directly encoding bounded mission-time properties in a form that aligns well with practical engineering workflows. We present MLTL.TEMPORALLOGIC.ORG, a zero-install, browser-based ecosystem for MLTL that supports the workflow from initial specification elicitation through validation and downstream verification. The ecosystem supports an iterative loop that starts with a myriad of elicitation methods spanning manual drafting, inference from labeled system traces, translation from structured natural language, and AI-assisted generation, and then feeds into a suite of techniques to help validate that the candidate specification matches the original intent. Using a sounding rocket requirement as a running example, we show how this loop helps engineers move from an informal English requirement to a formal, validated MLTL specification, and then carry it forward into runtime monitoring and model checking for verification. We argue that this kind of integrated, easily accessible tooling is an important research contribution that lowers the practical barrier to formal specification.

Keywords: Mission-time Linear Temporal Logic · Specification Elicitation · Runtime Verification · Formal Methods Ecosystem

1 Introduction

As automated systems grow both increasingly complex and ubiquitous, we critically depend on robust capabilities for their verification, certification, and regulation. All of these activities, in turn, centrally build upon a well-defined set of system requirements that we can transparently tie to system development, including emergent system behaviors. This is not just desirable, but *mandated* by regulation and certification authorities [26]. Tools that automate system verification or analyze requirements can only do so if the requirements are specific and unambiguous, e.g., written in formal mathematical logic as opposed to informal natural language. Requirements elicitation is notoriously tedious and error-prone activity [66]; *formal* requirements elicitation therefore can trigger cultural resistance in traditional engineering workflows under time pressure and short-term performance goals.

While the benefits of implementing formal verification far outweigh the cost of formalizing system requirements [14,33,61], there is still room to further streamline that process. We focus on Mission-time Linear Temporal Logic (MLTL) as our requirement specification language [49, 50, 63] because it strikes a comfortable balance between expressiveness and complexity. Many common requirements of reactive systems are expressible in MLTL, yet the logic remains sufficiently easy to analyze, e.g., using model checking at system design time or runtime verification at deployment time. Additionally, translation algorithms [50] from MLTL to other logics, such as LTL [69], LTLf [28], as well as solver input formats like SMT-LIB2 [7] allow us to leverage tools that use these logics and formats to analyze MLTL specifications. Furthermore, MLTL represents the most commonly-used fragment [63] of MTL [1] and STL [57], so alleviating the drudgery of specification elicitation and validation for MLTL directly benefits those logics.

One of the biggest barriers-to-entry of creating requirements in general, and formal requirements in particular, is the question of how to start. Initial system concepts can take many forms, including fuzzy ideas, English descriptions, sets of positive and negative examples, or combinations of those. Some engineers are in the habit of drafting formal requirements directly; others are not. For ease of workflow integration, we need to accommodate a range of requirements elicitation styles. Once we have an initial set of system requirements, we encounter the second major barrier-to-entry: requirements validation. For a large, complex system, it can be a daunting task to rigorously ensure that each requirement is correct (means what we think it means) and that the set of requirements as a whole is correct (allows expected emergent behaviors and prohibits unexpected ones). A third and much less commonly acknowledged barrier-to-entry is the ease of accessing existing tooling for requirements elicitation and validation: many available tool lives in its own silo, requiring separate installation, its own input format, and its own learning curve.

We present MLTL.TEMPORALLOGIC.ORG as a direct response to all three barriers: a zero-install, browser-based ecosystem integrating the complete life cycle of MLTL specification elicitation, validation, and deployment. The entire platform is based on open-source tools and algorithmic implementations [4, 5, 38, 50, 58, 64, 74, 76, 79] that we now continuously integrate and update. Fig. 1 gives an overview of the complete ecosystem. Multiple **elicitation** methods—ranging from manual formula composition, to inference from example traces, to structured natural language, to AI-assisted generation—feed into a candidate MLTL specification. That candidate is then analyzed using an integrated suite of **validation** tools (again spanning manual to fully automated), including satisfiability checking, trace enumeration, formula progression, language partitioning, and stream-based timeline visualization. Once validated, the specification can be deployed for **verification** via model checking or runtime monitoring. We argue that integrating these existing tools and algorithms into a single ecosystem is itself a research contribution, and that the ecosystem as a whole is more than the sum of its parts. This paper is an explicit attempt to close the gap between the theoretical research contributions of algorithms and the practical research contributions of usable tools, demonstrating the ecosystem through a single running example from informal engineering intent to a deployed runtime monitor.

We structure the rest of the paper as follows: Section 2 provides background on MLTL. Then, we walk through the elicitation (Section 3), validation (Section 4), and verification (Section 5) stages in turn, and conclude with a discussion of related efforts, the case for ecosystem papers, and future work in Section 6.

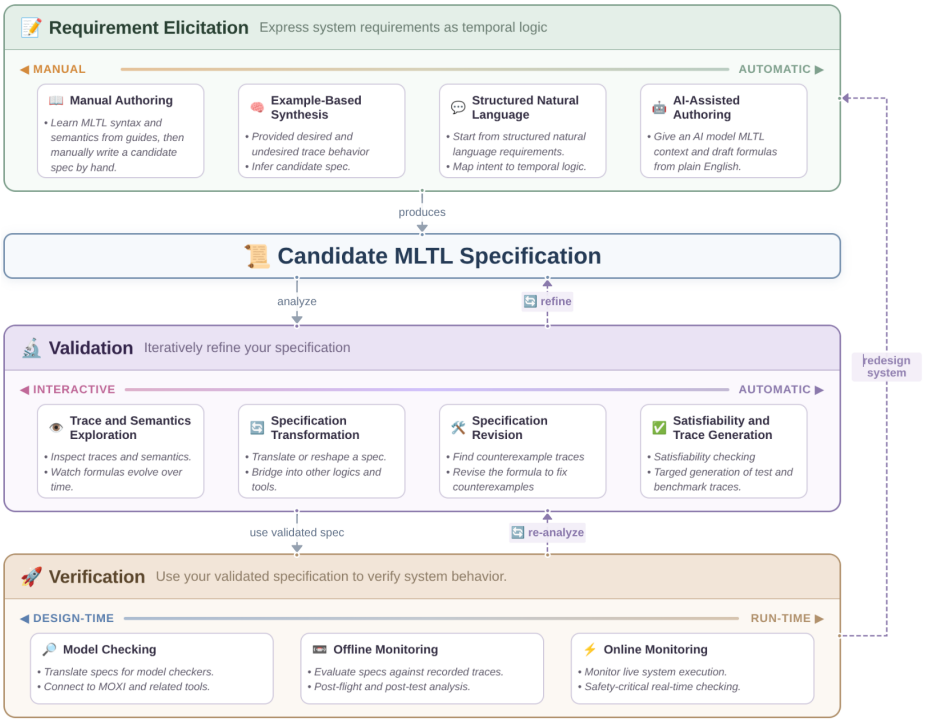


Fig. 1: Overview of the MLTL tools ecosystem, capturing the specification elicitation, validation, and verification life cycle. This figure is a direct screenshot of MLTL.TEMPORALLOGIC.ORG/TOOLS; clicking any box on the website reveals a list of all currently integrated tools offering that functionality.

2 MLTL Background

Mission-time Linear Temporal Logic (MLTL) [50,63] is designed for reactive systems operating over finite time horizons. While standard LTL reasons over infinite traces and may leave temporal operators unbounded, MLTL attaches a bounded, closed, discrete interval to every temporal operator, explicitly encoding the time window where a property must hold. This makes MLTL specifications directly actionable: the interval corresponds to physical mission-time steps, and the logic's finite-trace semantics matches how aerospace, automotive, and robotics systems will always have a end to the mission.

Definition 1 (MLTL Syntax). *The syntax of an MLTL formula φ over a finite set of atomic propositions $\mathcal{AP}$ is recursively defined as follows:*

$$\varphi ::= \mathsf{true} \mid \mathsf{false} \mid p \mid \neg\psi \mid \psi \wedge \xi \mid \psi \vee \xi \mid \Box_I\, \psi \mid \Diamond_I\, \psi \mid \psi\, \mathcal{U}_I\, \xi \mid \psi\, \mathcal{R}_I\, \xi$$

where $p \in \mathcal{AP}$ is an atomic proposition, ψ and ξ are MLTL formulas, and $I = [lb,ub]$ is a closed interval such that $lb \leq ub$ and $lb,ub \in \mathbb{N}_0$.

Definition 2 (Finite Trace). *A finite trace, denoted by π, is a finite sequence of sets of atomic propositions $\mathcal{AP}$. The i^{th} set is denoted by $\pi(i)$ and contains the atomic propositions that are satisfied at the i^{th} time step. $|\pi|$ denotes the length of π.*

For $|\mathcal{AP}| = n$, each time step $\pi(i)$ can be encoded as a bitstring of length n, where the j^{th} bit is 1 if the j^{th} atomic proposition (for some fixed total ordering on $\mathcal{AP}$) is

satisfied and 0 otherwise. Thus a trace π of length $|\pi|$ can be represented as a bitstring of length $n{\cdot}|\pi|$; for readability, we separate time steps with commas when $n{>}1$. We further generalize this representation to a *trace regular expression* [76] by introducing the symbol `s` to denote a *don't-care* value, i.e., a position that may be either 0 or 1. A single regex containing k occurrences of `s` therefore compactly represents 2^k distinct traces. For example, let $\mathcal{AP}=\{p,q\}$ (with ordering $p \prec q$) and consider the regex `10,s1,01` of length 3. The commas delimit time steps, each of which is a bitstring of length 2. This regex represents 2 traces: `10,11,01`, corresponding to $\pi{=}\{p\},\{p,q\},\{q\}$; and `10,01,01`, corresponding to $\pi{=}\{p\},\{q\},\{q\}$.

Definition 3 (MLTL Semantics). *We recursively define $\pi,i \models \varphi$ (finite trace π starting from time index $i{\geq}0$ satisfies, or "models" MLTL formula φ) as*

- $\pi,i \models \mathsf{true}$
- $\pi,i \models p$ *for* $p{\in}\mathcal{AP}$ *iff* $p{\in}\pi(i)$
- $\pi,i \models \neg\psi$ *iff* $\pi,i \not\models \psi$
- $\pi,i \models \psi \wedge \xi$ *iff* $\pi,i \models \psi$ *and* $\pi,i \models \xi$
- $\pi,i \models \Box_{[lb,ub]}\ \psi$ *iff* $|\pi| \leq i{+}lb$ *or* $\forall j{\in}[i{+}lb,i{+}ub]$ *such that* $\pi,j \models \psi$
- $\pi,i \models \psi\ \mathcal{U}_{[lb,ub]}\ \xi$ *iff* $|\pi| > i{+}lb$ *and* $\exists j{\in}[i{+}lb,i{+}ub]$ *such that* $\pi,j \models \xi$ *and* $\forall k{<}j$ *where* $k{\in}[i{+}lb,i{+}ub]$, $\pi,k \models \psi$.

Given two MLTL formulas ψ and ξ, they are semantically equivalent (denoted by $\psi{\equiv}\xi$) if and only if $\pi,0{\models}\psi \leftrightarrow \pi,0{\models}\xi$ for all finite traces π. To complete the semantics, $\mathsf{false}{\equiv}\neg\mathsf{true}$, $\psi \vee \xi{\equiv}\neg(\neg\psi \wedge \neg\xi)$, $\neg(\psi\ \mathcal{U}_I\ \xi){\equiv}(\neg\psi\ \mathcal{R}_I\ \neg\xi)$, *and* $\neg\Diamond_I\ \psi{\equiv}\Box_I\ \neg\psi$. *MLTL also holds the standard operator equivalences of* $\Diamond_I\ \psi \equiv (\mathsf{true}\ \mathcal{U}_I\ \psi)$, *and* $\Box_I\ \psi{\equiv}(\mathsf{false}\ \mathcal{R}_I\ \psi)$.

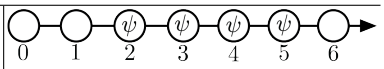
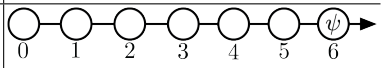
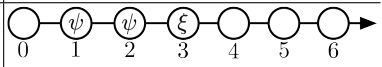
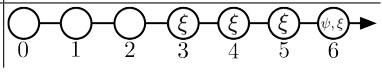
MLTL contains four future-time temporal operators (Always, Eventually, Until, and Release); Table 1 provides a visualization of example formulas containing these future-time operators. Intuitively, these temporal operators can be translated into plain English as follows:

- $\Box_{[lb,ub]}\ \psi$ means "ψ *always* holds at every point from *lb* to *ub* time steps in the future"
- $\Diamond_{[lb,ub]}\ \psi$ means "ψ *eventually* holds at some point between *lb* and *ub* time steps in the future"
- $\psi\ \mathcal{U}_{[lb,ub]}\ \xi$ means "ψ holds *until* ξ holds between *lb* to *ub* time steps in the future"
- $\psi\ \mathcal{R}_{[lb,ub]}\ \xi$ means "ψ *releases* the requirement that ξ holds between *lb* to *ub* time steps in the future"

Notably, the bounded, closed, discrete intervals in MLTL also guarantees that the number of distinct propositional variables across time steps is finite and enumerable—a property that enables tools like truth tables, AllSAT enumeration, and SAT-based test generation to run in bounded time.

Two evaluation modes. There is an important distinction that governs how MLTL specifications are evaluated in practice; we categorize this distinction into two evaluation modes: (1) single-point evaluation and (2) stream-based evaluation. In *single-point* evaluation (used by validation tools such as WEST [74], FPROGG [64], Language Partitioning [76], and Formula Progression [49]), a formula φ is checked once at time index $i{=}0$ (i.e., designed to only consider if $\pi,0{\models}\varphi$), asking the question "does this

Table 1: Pictorial representation of the four future-time temporal operators with satisfying traces at $i=0$.

Formula	Operator	Timeline
$\Box_{[2,5]}\psi$	ALWAYS$_{[2,5]}$	
$\Diamond_{[0,6]}\psi$	EVENTUALLY$_{[0,6]}$	
$\psi\ \mathcal{U}_{[1,4]}\ \xi$	UNTIL$_{[1,4]}$	
$\psi\ \mathcal{R}_{[3,6]}\ \xi$	RELEASE$_{[3,6]}$	

trace satisfy φ from the beginning?" In *stream-based* evaluation (the standard mode in runtime monitoring), φ is re-evaluated at every time index i such that φ is incrementally re-evaluted based on a continuous stream of input data. In other words, stream-based evaluation asks the question "does φ hold from *now*?". Notably, stream-based evaluation of φ is semantically equivalent to single-point evaluation of $\Box_{[0,M]}\varphi$, where M is the expected mission length as defined in Definition 4 below; therefore, an engineer who validates φ at $i=0$ and then monitors how φ is evaluated in a stream-based manner typically adds an external $\Box_{[0,M]}$ (i.e., result is $\Box_{[0,M]}\ \varphi$), which may or may not match intent. We highlight this distinction where relevant in Sections 3 and 4.

Definition 4 (Mission-time (M) [63]). *All MLTL formulas are mission-time bounded; therefore, let $M\in\mathbb{N}_0$ denote the end of mission-time (i.e., the time when the mission ends) and all trace π satisfy the trace length condition $|\pi|\leq M+1$.*

Past-time MLTL (ptMLTL). While MLTL strictly reasons over the future, ptMLTL [4] strictly reasons over the past. System requirements are more-commonly expressed in future time (e.g., [52,55,73]); therefore, this ecosystem mainly focuses on the future-time variant of MLTL. Nevertheless, we introduce ptMLTL here as ptMLTL is a supported logic for runtime monitoring with R2U2 (Realizable Responsive Unobtrusive Unit). Extending the elicitation and validation tools in the MLTL ecosystem to also support ptMLTL is ongoing work.

Definition 5. *(ptMLTL Syntax)* The syntax of a ptMLTL formula φ over a set of atomic propositions $\mathcal{AP}$ is recursively defined as:

$$\varphi ::= \mathsf{true} \mid \mathsf{false} \mid p \mid \neg\psi \mid \psi\wedge\xi \mid \psi\vee\xi \mid \boxminus_I\ \psi \mid \diamondminus_I\ \psi \mid \psi\ \mathcal{S}_I\ \xi \mid \psi\ \mathcal{T}_I\ \xi$$

where $p\in\mathcal{AP}$ is an atom, ψ and ξ are ptMLTL formulas, and I is a closed interval $[lb,ub]$ where lb and ub denote the lower and upper bound, respectively, such that $lb\leq ub$ and $lb,ub\in\mathbb{N}_0$.

Definition 6. *(ptMLTL Semantics)* We recursively define $\pi,i\models\varphi$ (finite trace π starting from time index $i\geq 0$ satisfies, or "models" ptMLTL formula φ) as

- $\pi,i\models\mathsf{true}$
- $\pi,i\models p$ for $p\in\mathcal{AP}$ iff $p\in\pi(i)$
- $\pi,i\models\neg\psi$ iff $\pi,i\not\models\psi$
- $\pi,i\models\psi\ \wedge\ \xi$ iff $\pi,i\models\psi$ and $\pi,i\models\xi$

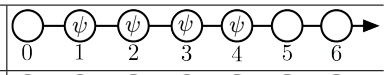
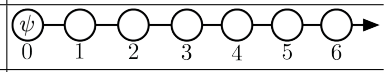
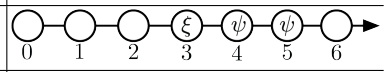
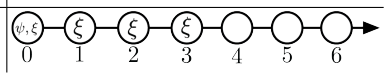
- $\pi,i \models \boxminus_{[lb,ub]} \psi$ iff $|\pi| \leq i-ub$ or $\forall j \in [i-ub,i-lb]$, $\pi,j \models \psi$
- $\pi,i \models \psi\ \mathcal{S}_{[lb,ub]}\ \xi$ iff $|\pi| > i-ub$ and $\exists j \in [i-ub,i-lb]$ such that $\pi,j \models \xi$ and $\forall k > j$ where $k \in [i-ub,i-lb]$, $\pi,k \models \psi$

Given two ptMLTL formulas ψ and ξ, they are semantically equivalent (denoted by $\psi \equiv \xi$) if and only if $\pi,0 \models \psi \leftrightarrow \pi,0 \models \xi$ for all finite traces π. To complete the semantics, $\mathsf{false} \equiv \neg\mathsf{true}$, $\psi \vee \xi \equiv \neg(\neg\psi \wedge \neg\xi)$, $\neg(\psi\ \mathcal{S}_I\ \xi) \equiv (\neg\psi\ \mathcal{T}_I\ \neg\xi)$, and $\neg\diamondminus_I \psi \equiv \boxminus_I \neg\psi$.

ptMLTL contains past-time operators that mirror the future-time operators of MLTL - Historically, Once, Since, and Trigger; Table 2 provides a visualization of example formulas containing these past-time operators. Intuitively, these temporal operators can be translated into plain English as follows:

- $\boxminus_{[lb,ub]} \psi$ means "ψ *historically* held at every time step from *lb* to *ub* time steps in the past"
- $\diamondminus_{[lb,ub]} \psi$ means "ψ *once* held at some between *lb* and *ub* time steps in the past"
- $\psi\ \mathcal{S}_{[lb,ub]}\ \xi$ means "ψ held *since* ξ held between *lb* to *ub* time steps in the past"
- $\psi\ \mathcal{T}_{[lb,ub]}\ \xi$ means "ψ *releases* the requirement that ξ held between *lb* to *ub* time steps in the past"

Table 2: Pictorial representation of the four past-time temporal operators with a satisfying trace timeline where $i=6$

Formula	Operator	Timeline
$\boxminus_{[2,5]} \psi$	Historically$_{[2,5]}$	
$\diamondminus_{[0,6]} \psi$	Once$_{[0,6]}$	
$\psi\ \mathcal{S}_{[1,4]}\ \xi$	Since$_{[1,4]}$	
$\psi\ \mathcal{T}_{[3,6]}\ \xi$	Trigger$_{[3,6]}$	

3 Specification Elicitation Strategies and Tools

With the syntax and semantics of MLTL in hand, the natural next question is: how does a practitioner actually *write* a specification? Oftentimes, this turns out to be the hardest step [66]. Engineers come to specification with vastly different starting points: some are comfortable with formal logic, others may have sample system traces, and still others begin with natural language requirements. A single elicitation pathway cannot serve all of them. We suggest four potentially overlapping approaches to eliciting MLTL specifications, spanning a spectrum from manual to fully automated: (1) learning MLTL via tutorials and manually composing specifications, (2) automatically inferring an MLTL specification from desirable and undesirable system traces, (3) translating structured natural language requirements into MLTL using the Formal Requirements Elicitation Tool (FRET) [2], and (4) enlisting the aid of large language models (LLMs). These approaches are complementary: an engineer may use one or all to figure out what works the best for their specific application domain.

We demonstrate all four approaches using a single running example drawn from the *Nova Somnium* sounding rocket's aerobraking control system [32]:

"When the Nova Somnium rocket enters the BOOST state, the state will change to COAST within seven seconds."

As we will see, each approach yields a slightly different candidate formula(s) for this requirement, illustrating both the value of having multiple on-ramps and the need for the validation tools of Section 4.

3.1 Learn and Write

The most direct path to writing a candidate specification is by learning MLTL and attempting to write formulas corresponding to your requirements by hand. The homepage of the site invites a user to "Learn MLTL" by going through an interactive MLTL Tutorial at MLTL.TEMPORALLOGIC.ORG/TUTORIAL.

The Interactive MLTL Tutorial is structured as a self-contained introduction to the logic. It opens by motivating temporal logic at a high level in the context of system verification, presents the formal syntax of MLTL, and then individually walks through the semantics of all four future-time temporal operators, and concrete example formulas. The tutorial notes basic equivalences such as that $\mathsf{true}\ \mathcal{U}_I\ \varphi \equiv \Diamond_I\ \varphi$, and duality properties, such as between $\Box/\ \Diamond$ and $\mathcal{U}/\mathcal{R}$.

Throughout the tutorial , hovering over any MLTL formula shows an automated English translation, giving an immediate intuitive reading. Fig. 2 shows an example of this feature. The tutorial also addresses common misconceptions, such as the distinction in time scope of the formulas $p_0 \rightarrow \Box_{[0,3]} p_1$ (a one-time check on p_0 at time 0) and $\Box_{[0,3]}(p_0 \rightarrow p_1)$ (a per-timestep implication).

The tutorial concludes with two interactive components: (1) a *quiz* that presents a formula and a concrete trace and asks the user to determine whether the trace satisfies the formula, and (2) a *Semantics Playground* where users type any MLTL formula φ, construct a trace π by toggling variables at various timesteps, and evaluating $\pi \models \varphi$ with an MLTL evaluator [58]. These components give the user hands-on practice with the semantics before they attempt to write their own specification.

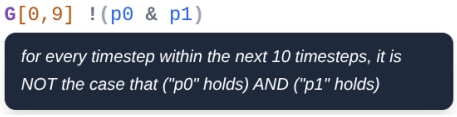


Fig. 2: Hovering over any MLTL formula on the site produces an automated English translation. This is a direct screenshot of an example formula in the tutorial page.

Specification Patterns. Specification patterns are shapes of MLTL formulas that encapsulate common requirements arising in real systems, and are a great starting point to drafting MLTL specifications. For instance, [21, 22] catalogs recurring patterns such as absence, existence, universality, response, precedence, and more, providing formula templates for LTL and CTL. These are extended to real-time temporal logics such as MTL and Timed CTL (TCTL) for patterns such as minimum and maximum duration, bounded recurrence, response, and invariance [44]. In the context of runtime verification (RV) for unmanned aircraft systems for MLTL, [66] identifies additional recurring patterns for runtime monitors such as allowable ranges for sensor readings, sensor rates, control sequences, and system-wide consistency.

Running Example. Armed with general MLTL knowledge and common specification patterns, an engineer can attempt to formalize the sounding rocket requirement by hand. The requirement says that *whenever* BOOST occurs, COAST must follow *within seven*

seconds. The "whenever" naturally maps to a $\Box$ over the mission-time, and "within seven seconds" maps to an eventually operator $\Diamond_{[0,7]}$. Letting $p_0 = \text{BOOST}$ and $p_1 = \text{COAST}$ and assuming single-point evaluation, a first attempt yields $\varphi = \Box_{[0,M]}(p_0 \rightarrow \Diamond_{[0,7]} p_1)$. If we intend to think about the formula in a stream-based setting, then we may simply write $\varphi = p_0 \rightarrow \Diamond_{[0,7]} p_1$.

3.2 Generating MLTL via Inference from Example System Behaviors

Oftentimes, an engineer may have a collection of system traces (e.g., logs, telemetry recordings, or simulation runs) that can each be individually classified as desirable or undesirable system behaviors. *Specification inference* - also known as programming by example (PBE) or specification mining - exactly solves this problem: given a set of positive traces P (desirable behaviors) and negative traces N (undesirable behaviors), automatically infer a specification that separates P and N (based on single-point evaluation). For MLTL, this means finding some formula φ such that for every $\pi \in P$, $\pi,0 \models \varphi$ and for every $\pi \in N$, $\pi,0 \not\models \varphi$.

This is an active area of research with a rich body of work spanning LTL/LTLf inference [24, 54, 65], STL inference [9, 10, 43, 48, 60], and MTL inference [35, 36, 62]. Our ecosystem provides an MLTL-specific inference engine [79], accessible at MLTL.TEMPORALLOGIC.ORG/MLTL-INFERENCE, that searches for candidate formulas using two complementary strategies: *beam search* over the space of MLTL syntax trees, which systematically expands partial formulas by scoring them against the training traces, and *genetic search* (implementing the DSGE algorithm [53] for MLTL), which evolves a population of candidate formulas using crossover and mutation operators defined over MLTL's grammar.

The user provides positive and negative traces through the web interface and configures search parameters (maximum formula depth, interval bounds, search budget). Then, the inference returns a ranked list of candidate formulas sorted first by the fraction of input traces correctly classified, and then by formula length (shorter specs are usually preferred over longer ones because they are easier to understand and validate).

Running Example. Returning to the sounding rocket example, suppose an engineer does not yet know the formal MLTL specification but has access to system traces (perhaps from simulation runs or flight logs) that can be classified as desirable (the rocket transitions from BOOST to COAST within a timely manner) or undesirable (BOOST occurs without a timely COAST response). Fig. 3 shows the trace-building interface with four positive and three negative traces over two propositional variables ($p_0 = \text{BOOST}$, $p_1 = \text{COAST}$) and nine timesteps. The positive traces include scenarios where COAST appears within seven timesteps of BOOST (e.g., at $t=1$, $t=4$, or $t=6$), as well as a trace with no BOOST at all (vacuously desirable). The negative traces capture a BOOST with no subsequent COAST, a BOOST where COAST arrives too late ($t=8$), and a case in which BOOST and COAST simultaneous hold at $t=0$.

Running the inference engine on these traces yields the five candidate formulas that achieve 100% training accuracy, shown in Fig. 4. Candidates #1–#4 are discovered by the beam search (`tree_search`) strategy, while candidate #5 is found by the grammar-guided genetic search (`dsge`). Notably, *none* of the inferred formulas exactly match the manually drafted specification from Section 3.1, although candidate #2, $\Box_{[0,1]}(\Diamond_{[1,7]} p_1 \,|\, \neg p_0)$, is close; it is equivalent to $\Box_{[0,1]}(p_0 \rightarrow \Diamond_{[1,7]} p_1)$. Compared with the hand-written formula $\Box_{[0,M]}(p_0 \rightarrow \Diamond_{[0,7]} p_1)$, it differs in two ways. First, the

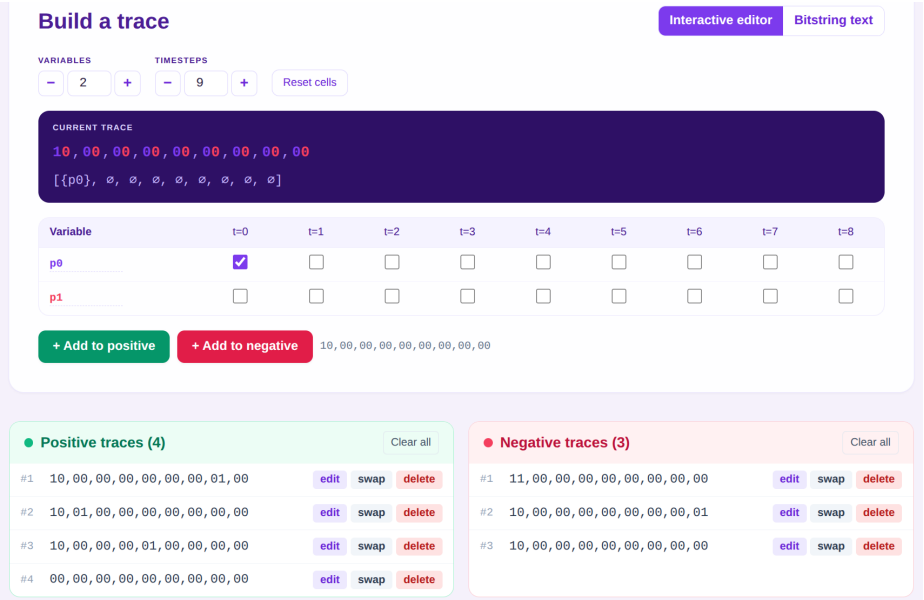


Fig. 3: The MLTL inference trace-building where the user can construct traces via an interactive editor and classifies each as a positive or negative example. Available online at MLTL.TEMPORALLOGIC.ORG/MLTL-INFERENCE.

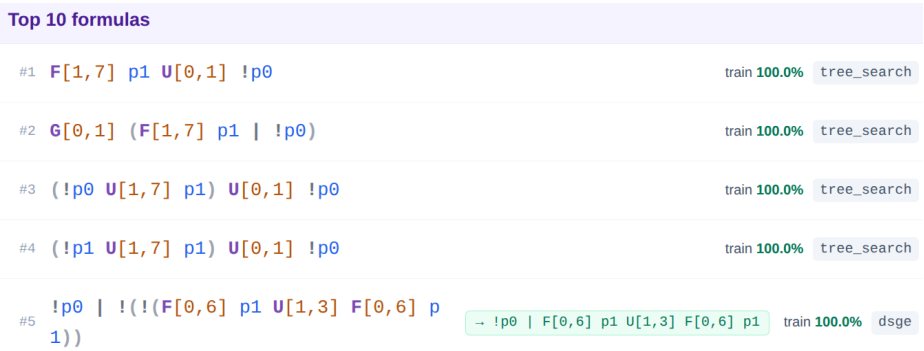


Fig. 4: The five candidate MLTL formulas inferred from the traces in Fig. 3 that achieve 100% training accuracy. Candidates #1–#4 are found by beam search (`tree_search`); candidate #5 is found by grammar-guided genetic search (`dsge`).

inference procedure has no direct notion of the mission bound M from Definition 4, and indirectly infers the outer interval as $\Box_{[0,1]}$. Second, because the training data treats simultaneous BOOST and COAST at $t=0$ as a negative example, the inferred response window becomes $\Diamond_{[1,7]}$ rather than $\Diamond_{[0,7]}$. The other candidate specifications need further investigation to determine whether they capture the intended requirement.

3.3 Eliciting MLTL from Structured NL Requirements with FRET

System requirements are often expressed in ambiguous natural language, which are challenging to formalize. To address this challenge, NASA's Formal Requirements Elicitation Tool (FRET) leverages the *structured* natural language known as FRETISH

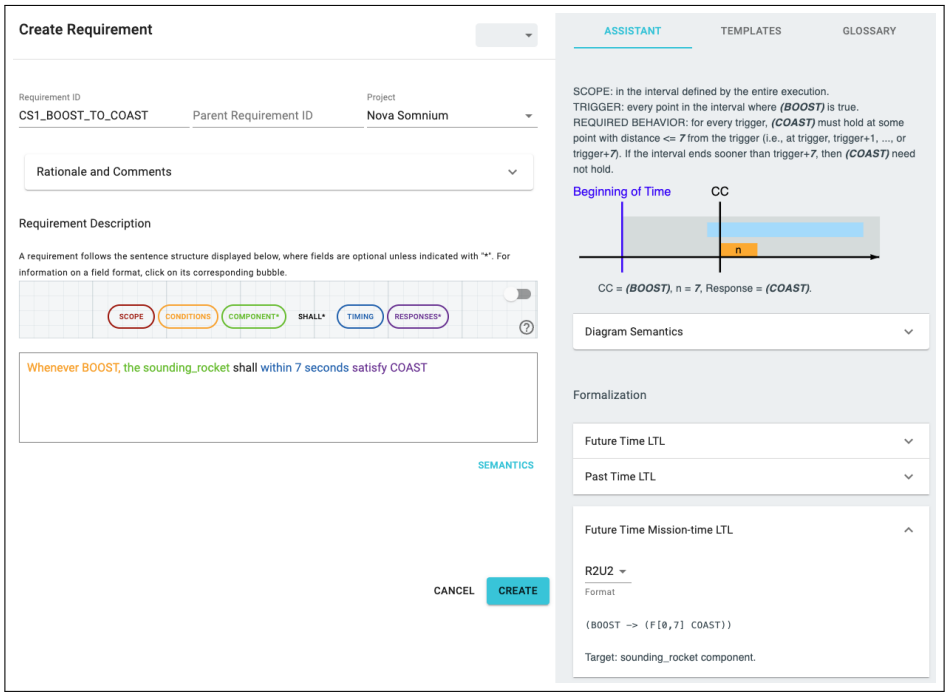


Fig. 5: FRETish requirement and MLTL formalization for stream-based evaluation

[29,30]. FRETish is a controlled subset of natural language such that it is composed of five fields: `scope`, `condition`, `component`, `timing`, and `response`. Utilizing FRETish, users can specify a requirement for a specific `component` within a certain `scope` of operation under a particular `condition` where the `response` must be satisfied within the defined `timing` constraints. We translate our running example into FRETish as "*whenever BOOST, the sounding_rocket* shall *within 7 seconds satisfy COAST*" as demonstrated in Fig. 5.

A translation from FRETish to MLTL is introduced in [2]. It is essential to recognize that this translation is formally verified with the aid of the PVS theorem prover [15] and SMT [2]; therefore, given the underlying semantics of FRETish, we have a correct MLTL formalization of FRETish requirements. Significantly, this translation to MLTL is also directly available for use in *both* single-point and stream-based evaluation. In Fig. 5, the MLTL formalization is shown for stream-based evaluation with R2U2. This formalization captures that whenever the *sounding_rocket* is in *BOOST*, then *COAST* must be satisfied *within 7 seconds*. Recall that since this is designed for stream-based evaluation, this formalization will be re-evaluated for each time index i. Comparatively, Fig. 6 exhibits the MLTL formalization for single-point evaluation in

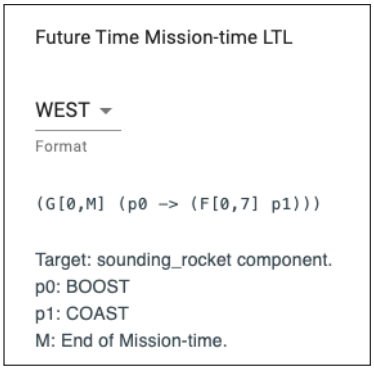


Fig. 6: MLTL formalization in FRET for single-point evaluation

the WEST tool. Since the MLTL formalization is given for both evaluation modes, one can easily utilize this formalization within any of the other tools within the MLTL ecosystem without worrying about incorrect assumptions or translation between the two modes. Its worth noting that the resulting MLTL formula presented in this Section is also identical to the formula produced in Section 3.1.

3.4 Generating MLTL from NL Requirements with AI-assistance

Finally, the most automated way is simply just prompting any large language model (LLM). The rapid advancement of such models means that the best model today may not be the best the model tomorrow, and so instead we provide a MLTL.md markdown file (available at MLTL.TEMPORALLOGIC.ORG/MLTL-REFERENCE) that supplement a prompt with information about MLTL. As of time of writing, we prompted Claude Sonnet 4.6 (free model) with the exact English spec and with the markdown file attached, and received the formula $\square[0,N](boost \rightarrow \diamond[1,7]coast)$ (see Fig. 7), where it tells us that N should be a "mission window of N seconds."[3] Note that this formula is meant to be evaluated from the beginning of the mission (i.e., single-point evaluation), and otherwise corresponds to the same specification as the one we manually drafted in Section 3.1.

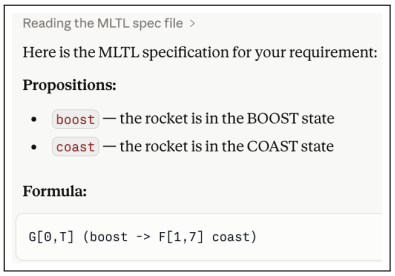


Fig. 7: Screenshot of Claude Sonnet 4.6 generating an MLTL specification from a natural language requirement

4 Specification Validation Strategies and Tools

The previous section introduced four different approaches to eliciting MLTL specifications that yielded many candidate formulas, all of which capture the intended requirement to some degree. It is now our job to determine which one is correct, if any, through careful validation. What does correctness even mean when the ground truth is an informal English sentence?

At its heart, validation is the process of identifying *counterexample traces*. Are there traces that satisfy a candidate formula but shouldn't (meaning the specification is too permissive), or traces that violate the candidate formula but shouldn't (meaning the specification is too strict)? An imprecise specification can "cry wolf" often enough that engineers stop trusting the monitor, or worse, stay silent and lend false confidence when the wrong behavior occurs. In this section, we begin with basic sanity checks and techniques for understanding formula semantics on concrete traces, then move to methods for surfacing counterexample classes more systematically, and finally discuss rewriting and revision once a mismatch has been identified.

4.1 Trace and Semantics Exploration

A variety of methods exist for exploring traces and formula semantics, ranging from interactive tools that let users step through individual traces to automatic techniques that systematically generate traces exposing different formula behaviors.

[3] Full conversation available here: https://claude.ai/share/b9bcaaa0-894e-458e-bb75-07fbc1b5f66f

Satisfiability Checking. For a given formula φ, MLTL Satisfiability Checking (MLTL-SAT) asks the question "does there exist a trace π such that $\pi \models \varphi$?" This is the most primitive sanity-check for any candidate specification [70]; if the spec is unsatisfiable or the *negation* of the spec is unsatisfiable (i.e., the spec is a tautology), we better revise the specification.

In the case that a given specification is satisfiable, the output of the underlying MLTL-SAT techniques is a specification-satisfying trace. This trace-generating behavior allows MLTL-SAT to serve as a basis for other techniques beyond just sanity checking, including benchmark generation [49, 64]. If a specification is unsatisfiable, this implies there is no satisfying trace, a fact that can help prove properties about formulas. For example, one can show that two formulas φ and ψ are equivalent if the formula $\neg(\varphi \leftrightarrow \psi)$ is unsatisfiable [2].

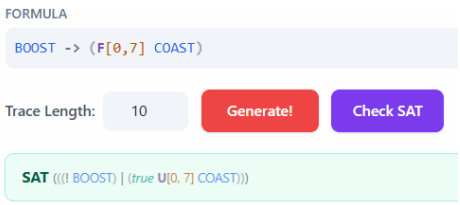


Fig. 8: SAT-Checking for $\varphi = BOOST \rightarrow \Diamond_{[0,7]} COAST$

All current techniques for MLTL-SAT are reductions to other problems, namely: LTL and LTLf satisfiability checking [50], invariant model checking [50], propositional satisfiability checking [31], and various SMT-based encodings [41, 50]. The artifacts of [31, 50] provide implementations of their presented reductions, and the Configuration Compiler for Property Organization (C2PO), the formula compiler for R2U2 and an MLTL analysis tool, provides a number of SMT encodings [38].

Formula Progression. Formula progression [47, 49] offers a trace-local perspective to how a formula can be partially evaluated as each timestep of a concrete trace is consumed. Given a formula φ and a trace π, progression rewrites φ at each timestep t into a residual formula φ_t that captures the remaining obligation after observing the first t steps of π. The ecosystem exposes this as an interactive visualization at MLTL.TEMPORALLOGIC.ORG/FORMULA-PROGRESSION: the user enters a formula and a trace, and the tool displays, step by step, how the formula simplifies (or grows) at each timestep until it reduces to true or false.

This is particularly useful for debugging: if a trace unexpectedly violates (or satisfies) a formula, formula progression pinpoints the exact timestep where the formula's evaluation diverges from expectation, providing insight into which part of the formula is responsible for the discrepancy.

Stream-based Timeline Visualization. Both WEST and Formula Progression consider only single-point evaluation of MLTL formulas. The R2U2 Playground [4] is designed to simulate how MLTL specifications are evaluated in a stream-based manner, and provides an interactive web-based playground that visualizes how the R2U2 runtime monitor evaluates MLTL specifications. In fact, the Rust realization of R2U2 is being directly run in the background. The web-based interface offers a wide range of features to be able to interact with R2U2, but most significantly, the R2U2 Playground offers stepwise execution coupled with reactive timeline plotting to enable easier specification understanding and debugging. Fig. 9 demonstrates the timeline visualization of our running example $\varphi = BOOST \rightarrow \Diamond_{[0,7]} COAST$ over the given trace. The trace given in Fig. 9 defines BOOST being true at time index $i \in [1,5]$ and COAST being true

at time $i=9$. We can break down this example into three different cases of how this MLTL specification is evaluated for each time index i:

Table 3: Explanation of stream-based evaluation from Fig. 9

Time index i	Verdict of φ	Explanation
$i=0$ and $i\in[6,9]$	true	BOOST is false; therefore $\rightarrow$ is vacuously true
$i=1$	false	BOOST is true but $\Diamond_{[0,7]}COAST$ is false for $i=0$
$i\in[2,5]$	true	BOOST is true and $\Diamond_{[0,7]}COAST$ is true for $i\in[2,5]$



Fig. 9: R2U2 Playground Timeline Visualization for $\varphi=BOOST\rightarrow\Diamond_{[0,7]}COAST$

4.2 Counterexample Identification

Identifying which potential counterexamples are worth exploring in depth makes up a majority of the validation effort. Below, we first consider trace enumeration via regular expressions, which exposes the satisfying language directly; then trace generation for stream-based evaluation, which more automatically proposes behaviors to inspect; and finally language partitioning, an orthogonal technique that decomposes a specification into sub-behaviors that can be explored further by either method.

Trace Enumeration via Regular Expressions. The exploration techniques above help an engineer understand what a formula means for individual traces. To systematically identify mismatches between the requirement's intent and the formal specification, a more exhaustive approach is to enumerate *all* satisfying traces via trace regular expressions [25,78]. Each trace regular expression captures a structural class of satisfying behavior, so the engineer can scan the complete set of permitted behaviors and ask: is there any class here that should *not* be permitted, or any expected behavior that is

missing? This makes trace enumeration a direct method for surfacing counterexamples: a regex pattern that describes clearly unintended behavior is an immediate signal that the specification is too permissive, while the absence of an expected pattern signals that the specification is too strict.

The scalability of this approach is inherently limited by the formula's propositional blowup, since it solves the AllSAT problem for MLTL. When only a single-trace verdict is needed ("does this particular trace satisfy φ?"), lightweight single-point evaluation (e.g., the Semantics Playground from Section 3.1) is more appropriate.

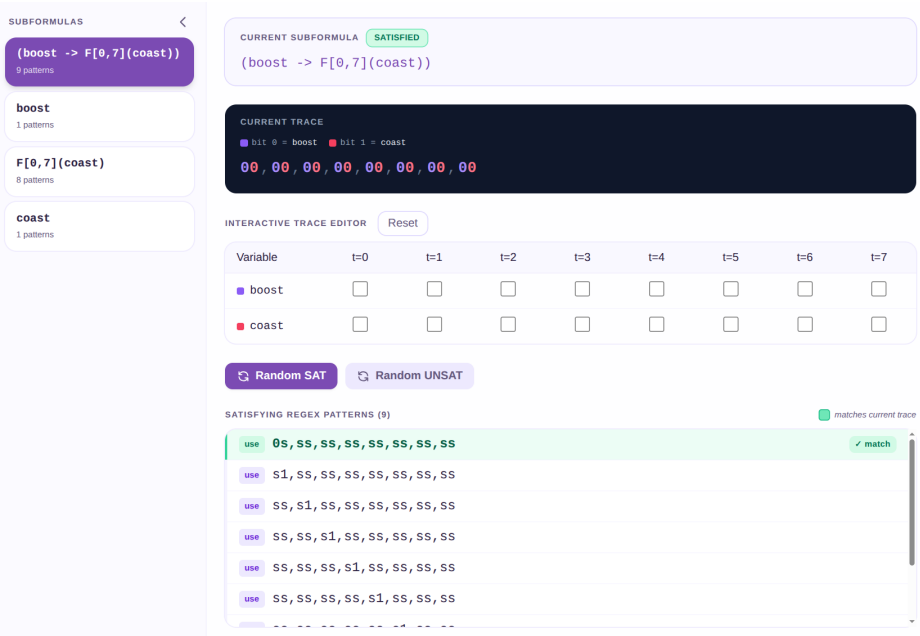


Fig. 10: WEST output for $\varphi = \mathit{boost} \rightarrow \Diamond_{[0,7]} \mathit{coast}$. Nine regex patterns enumerate all satisfying traces; the first six are visible. The first pattern (highlighted green) matches the current all-zeros trace, which vacuously satisfies φ since BOOST is false. The three patterns cut off at the bottom are `ss,ss,ss,ss,ss,s1,ss,ss`, `ss,ss,ss,ss,ss,ss,s1,ss`, and `ss,ss,ss,ss,ss,ss,ss,s1`.

The WEST tool [74] (now available online at MLTL.TEMPORALLOGIC.ORG/WEST) provides this capability: given a formula φ, WEST presents the complete set of satisfying traces grouped by regex pattern. The user can then interactively select any regex pattern to see which concrete traces it represents, randomly generate satisfying or dissatisfying traces, and check whether a particular trace of interest matches a given pattern.

Running Example. Fig. 10 shows the output of WEST on our running example $\varphi = \mathit{boost} \rightarrow \Diamond_{[0,7]} \mathit{coast}$. The tool produces nine regex patterns. The first pattern, `0s,ss,ss,ss,ss,ss,ss,ss`, captures all traces where BOOST is false at $t=0$ and the remaining seven timesteps are unconstrained: this is the vacuous satisfaction case.

Each of the remaining eight patterns has the form `ss,...,s1,...,ss`, with exactly one position where COAST is forced to be true; these cover the cases where COAST occurs at $t=0,t=1,...,t=7$ respectively, regardless of whether BOOST is true. Together, the nine patterns exhaustively characterize the formula's language. Scanning them immediately reveals that vacuous satisfaction (no BOOST) produces a large class of satisfying traces. If the engineer considers vacuous truth undesirable, this is a counterexample class that prompts strengthening the specification.

Trace Generation for Stream-based Evaluation While all MLTL-SAT tools are able to provide counterexample and witness traces to become positive and negative traces for exploring system behavior, they only consider single-point evaluations, not the stream-based repeated evaluations which would model monitoring use cases, as seen with the R2U2 Playground. This is an important capability, as some specifications are not able to be satisfied at every timestep, such as $p \wedge \mathcal{F}_{[1,1]} \neg p$.

The Formula Progression Generator (FPROGG) tool utilizes both MLTL-SAT tools and Formula progression to generate traces which satisfy a formula under the repeated evaluations of monitoring. To further explore potential system behaviors, FPROGG offers automated generation of traces that minimize satisfying assignments for potentially finding counterexample traces. FPROGG also offers the ability to generate traces which switch from satisfying evaluations to dissatisfying the formula and vice-versa to enable engineers to see the system states that would be required for the system to move from a satisfying state to a dissatisfying one, and back (recoverability).

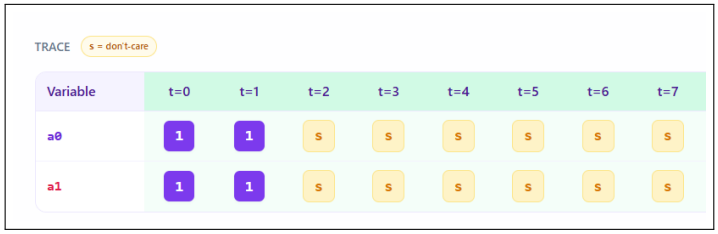


Fig. 11: A satisfying trace generated by FPROGG for $\varphi = BOOST \rightarrow \Diamond_{[0,7]} COAST$. The benchmark directly shows that the requirement as formalized permits the rocket to be in both BOOST mode and COAST mode simultaneously

Taken together, the MLTL formula, trace, and evaluation vector are known as *MLTL RV Benchmarks* because they enable the development and profiling of RV monitors outside of this ecosystem.

Language Partitioning. Language partitioning [76] takes a different approach to understanding a formula: rather than rewriting it into an equivalent single formula, it decomposes the formula into a set of *co-formulas* $\psi_0, \psi_1, ..., \psi_k$ whose satisfying trace sets are pairwise disjoint and whose union equals the language of the original formula. That is, $\mathcal{L}(\varphi) = \mathcal{L}(\psi_0) \cup \mathcal{L}(\psi_1) \cup \cdots \cup \mathcal{L}(\psi_k)$. This lets the engineer examine more fine-grained categories of satisfying behavior: instead of asking "does this trace satisfy φ?", one can ask "*how* does this trace satisfy φ?"

The tool is available at MLTL.TEMPORALLOGIC.ORG/LANGUAGE-PARTITIONING. The user supplies an MLTL formula, along with *interval compositions* that specify how each temporal operator's interval should be decomposed into sub-intervals.

Running Example. We partition the inner formula of our running example, $p_0 \rightarrow \Diamond_{[1,7]} p_1$, using the composition $\langle 2,3,2 \rangle$ to split the interval [1,7] into early ($[1,2]$), mid ($[3,5]$), and late ($[6,7]$) sub-intervals. The tool produces seven co-formulas, shown in Fig. 12. The first three (ψ_0–ψ_2) cover the case where BOOST is absent ($\neg p_0$): COAST arrives early, mid, or late. The next three (ψ_3–ψ_5) cover the case where BOOST is present (p_0): again with COAST arriving early, mid, or late. The final co-formula $\psi_6 = \neg p_0 \wedge \Box_{[1,7]} \neg p_1$ captures the vacuously-satisfying case: no BOOST and no COAST at all. Notice that the case $p_0 \wedge \Box_{[1,7]} \neg p_1$ (BOOST with no COAST) is absent, precisely because it violates the original formula.



Fig. 12: Language partitioning result for $p_0 \rightarrow \Diamond_{[1,7]} p_1$ with interval composition $\langle 2,3,2 \rangle$. The seven co-formulas partition the formula's language into categories of satisfying behavior.

Each co-formula can be individually analyzed by any suitable tool, giving the engineer a structured way to inspect more fine-grained classes of satisfying behavior. For instance, an engineer might realize that ψ_5 (BOOST with COAST arriving only at timestep 6 or 7) is cutting it dangerously close and decide to tighten the bound from $\Diamond_{[1,7]}$ to $\Diamond_{[1,5]}$. Or, they might notice that ψ_6 (no BOOST and no COAST) is an undesirable vacuous case and decide to rewrite the original formula to exclude it.

4.3 Revising Specifications via Counterexamples

At this point, once we have identified concrete counterexamples pointing to a flaw in a candidate specification, the workflow naturally loops back to the elicitation strategies of Section 3. An engineer may return to the tutorial and specification patterns to manually rewrite the formula, refine the underlying FRETish requirement, ask an LLM for a revised formalization, or use inference to search for a new candidate consistent with the counterexample.

For the manual-revision route, it can help to first view the formula in a simpler but equivalent form, since obvious simplifications may reveal a cleaner way to repair it; the ecosystem automatically applies such simplifications, as illustrated in Fig. 13. This lightweight simplifier is only the first layer: C2PO (Configuration Compiler for Property Organization), which we return to in Section 5, implements a much richer collection of rewrite and simplification rules for MLTL formulas [38, 39]. For example, it can factor out common temporal contexts, rewriting $\Box_{[0,3]} \varphi \wedge \Box_{[0,3]} \psi$ as $\Box_{[0,3]} (\varphi \wedge \psi)$, and more generally applies rules that merge redundant



Fig. 13: Automatic formula simplification on the ecosystem site. The input formula (top) is rewritten to an equivalent, more readable form (bottom) by applying De Morgan's laws, temporal duality, and trivial operand elimination.

temporal structure and simplify nested operators. The recent FRET–R2U2 integration extends this rewrite-based workflow to specifications that originate as FRETish requirements [2]. Thus, whether a candidate specification came from direct MLTL entry or from structured natural language, rewriting support can help expose a simpler, equivalent formula that is easier to understand and modify.

The inference route is particularly easy to chain with the validation tools above. The current (flawed) formula φ can be sampled to produce an initial dataset: satisfying traces generated via WEST or FPROGG become positive examples, and dissatisfying traces become negative examples. Newly discovered counterexamples are then added to the appropriate set, positive if φ was too strict and negative if φ was too loose, and inference is re-run on the augmented dataset. In this way, WEST and FPROGG can help surface counterexamples, while inference proposes revised formulas that are consistent with both the original intended behavior and the new evidence. The process continues iteratively until the engineer is satisfied with the specification.

5 Verification

Once we have a validated specification, we want to use it to do useful formal verification of our system in a spectrum of settings ranging from design-time to run-time. We focus on two options: (1) *model checking* against a system model, and (2) *deployment* as a runtime monitor.

5.1 Translation and Rewriting for Verification

Rewriting is an effective way to obtain syntactically different but semantically equivalent formulas. This enables automatic simplification, reducing the size and complexity of a given formula, and exploring the space of formulas equivalent to the original. Beyond simplification, rewriting can also serve as an optimization depending on the end application. For example, for runtime monitoring with R2U2, rewrite rules can reduce the require memory for the resulting monitors [2, 39]. Rewrites for optimization are implemented in C2PO and FRET.

Additionally, translation algorithms [50] from MLTL to other logics, such as LTL [69], LTLf [28], as well as solver input formats like SMT-LIB2 [7] allow us to leverage tools that use these logics and formats to analyze MLTL specifications. This capability is already available at MLTL.TEMPORALLOGIC.ORG/TRANSLATION. Thus, once a specification has been validated, the ecosystem can translate it into other logics or input formats to support downstream verification and deployment.

5.2 Model Checking

Model checking exhaustively verifies that *all* possible behaviors of a system model M satisfy a specification φ, i.e., $M \models \varphi$. Model checking is both sound and complete within the context of the given system model: when it returns a verdict, that verdict is correct, and it always returns a verdict.

As discussed in the previous subsection, validated MLTL specifications can already be translated into both LTL and LTLf. This lets engineers reuse existing model-checking backends rather than waiting on a dedicated MLTL-only model checker: the LTL translation can be checked with mature tools such as SPIN [34] and nuXmv [13, 34], and can also feed explicit-state model-checking workflows through deterministic compilation of temporal safety properties [68], while the LTLf translation opens finite-trace model-checking workflows such as CPD_{LTLf} [80]. In this sense, the ecosystem's translation layer is the bridge from a validated MLTL requirement to downstream design-time

verification in whatever LTL- or LTLf-based tool best matches the engineer's model and workflow. In practice, nuXmv also provides limited direct support for MLTL-style bounded temporal operators, including bounded versions of $\Box$ and $\Diamond$, although this still falls short of full native MLTL model checking [11].

Proof Assistants. A substantial body of work already formalizes MLTL in PVS [15], and especially Isabelle/HOL [45,46,75,77]. These provide nearly arbitrary expressiveness for encoding a system model and proving that it satisfies an MLTL specification, at the cost of signifcant manual effort and expertise in the proof assistant.

5.3 Runtime Monitoring

While model checking is a complete technique for verifying that all possible behaviors of a system model satisfy an MLTL specification, it's not always feasible to construct such a full model of the system. An alternative to model checking is to check if various system runs satisfy our specifications through a technique known as *runtime monitoring*. Furthermore, there are two variations of runtime monitoring: (1) *offline* runtime monitoring and (2) *online* runtime monitoring. *Offline* runtime monitoring requires that all system data has already been previously captured either through simulation or logging of a physical system, while *online* runtime monitoring observes system data during execution as new data becomes available.

Offline Runtime Monitoring. SABRe (Stream Analyzer via Bitwise Reasoning) is an MLTL runtime monitoring tool designed to efficiently check system runs *offline* [41]. The tool uses C2PO to generate a C program from an MLTL formula. That C program then takes as input the trace (generated via simulation or logging) and outputs a sequence of true/false verdicts for each timestamp, as in stream-based monitoring (Sec. 2). SABRe operates at a throughput of up to 25 million verdicts per second on an ASCII text trace [41], and up to 250 million verdicts per second on raw bytes (i.e., each bit in the file corresponds to whether a proposition is true or false at that point in the trace).

Online Runtime Monitoring. R2U2 (Realizable Responsive Unobtrusive Unit) is a runtime monitoring tool designed to evaluate system behavior against a set of MLTL or ptMLTL specifications in real-time [4,38,42]. While R2U2 can be utilized for *offline* runtime monitoring as seen with the R2U2 Playground, R2U2 has been specifically designed and optimized for *online* runtime monitoring. To this extent, *online* runtime monitors must frequently fit within the tight timing and memory constraints of the target system in order to produce verdicts in real-time [8,23,42], and this is where R2U2 proves particularly effective. R2U2 requires minimal resource overhead while also being readily available in both C [38] and Rust [4] with proofs of correctness available in [4,42,81], enabling effective deployment to a wide range of systems. As a result, R2U2 has been successfully deployed on various systems such as NASA's Robonaut2 [42], NASA's Lunar Gateway Vehicle System Manager [16–19], NASA's Swift UAS [27,63,72], NASA's DragonEye UAS [59,71], Iowa State's Nova Somnium sounding rocket [32], Iowa State's CySat-I CubeSat [3], and more [12,37,55,56].

6 Discussion and Future Work

6.1 The Case for Ecosystem Papers

Formal methods research has long produced sophisticated algorithms, correctness proofs, and high quality software artifacts. Yet adoption remains stubbornly low, and the

bottleneck is rarely theoretical: it is a user experience problem. Getting started is the hardest step: even before a user is ready to write a specification, if a tool is difficult to install, platform-dependent, or only offers command-line interfaces, engineers will hesitate. Most research outputs stop at "algorithm + CLI prototype," leaving the last mile from algorithm to usable tool unfunded, unrewarded, and largely unattempted. Static artifacts are crucial for research reproducibility, but they are not sufficient for adoption beyond the formal methods community.

Ecosystem papers fill this gap by presenting the *deployment* of formal methods as a research contribution in its own right. We argue that the whole is greater than the sum of its parts: integrating elicitation, validation, and verification around a shared workflow creates a feedback loop that is invaluable for getting formal methods into the hands of practitioners. The four elicitation on-ramps (Section 3) are a UX response to the fact that different users arrive with different backgrounds and preferences.

The current wave of AI enthusiasm frames large language models as an oracle that will "solve" specification. We think this overstates the case: AI is better understood as a *translator*: one on-ramp among several, not a replacement for human understanding. The heavy lifting is done by the underlying algorithms; AI closes the last mile by lowering the barrier to reaching them. We should be candid as well: much of the ecosystem's frontend was built with significant assistance from Claude Opus 4.6. But even with AI assistance, defining proper API contracts, deciding what backends should talk to what, and wiring it all together in a way that easily scales (to support more users and integrate more tools) was a nontrivial systems engineering effort.

We acknowledge the responsibilities associated with the use of AI-assisted tools. Certain aspects of dataset provenance and licensing require careful consideration, and we apply such tools conservatively, following best practice guidelines. More practically, we believe that integrating someone else's research tool into a new interface (AI-assisted or otherwise) should require their explicit consent. Tools should be integrated into new interfaces only with the explicit knowledge and consent of their creators. We take the pragmatic stance that AI is a useful engineering tool when applied with care, transparency, and respect for the community whose work it builds on.

Finally, a deployed web ecosystem is itself a living artifact: reviewers and readers can interact with it immediately, without build instructions or environment setup. To our knowledge, no prior work provides a browser-based *ecosystem* for temporal logic specification that spans elicitation, validation, and verification under a unified workflow; MLTL.TEMPORALLOGIC.ORG is designed to fill exactly this gap. This liveness is both the primary advantage and the defining challenge of the ecosystem model.

6.2 Limitations and Future Work

Presenting a living, evolving website as a research contribution means that the specific tools described in this paper will inevitably shift. New state of the art tools will be integrated, old tools may become deprecated, and the UI will be iteratively redesigned and refined based on user feedback. We view this not as a weakness but as the defining feature of the ecosystem model, in which the centralized framework serves as a common connection point between past and future tools.

The list of tools that can and should be integrated is long and growing. Multimodal Model Predictive Runtime Verification (MMPRV), together with support for Predictive MLTL (PMLTL), would add predictive and probabilistic monitoring capabilities within R2U2 [6]. When a large specification set collectively becomes unsatisfiable, engineers

need to know the *maximum* subset of requirements that can simultaneously hold, i.e., the MLTL MaxSAT problem [31]; integrating its solver is a natural next step. Since MLTL already translates to LTL, existing LTL-to-timeline visualization tools [51] and the automata representations available through Spot [20] would immediately extend to MLTL timelines and automata via the existing translation pipeline. A direct MLTL-to-MoXI translation is another natural extension: MoXI was designed as an intermediate language for symbolic model checking, and its model-exchange tool suite would provide a principled route for connecting MLTL to a broader ecosystem of symbolic backends without committing to a single target verifier [40, 67]. Each of these can and should slot into the existing ecosystem architecture with minimal friction, and the ecosystem's liveness means that new tools can be made available to users immediately upon integration.

Integration is an ongoing process: it continually evolves as new tools emerge and existing ones mature. A healthy ecosystem is one that continuously absorbs new research contributions and makes them immediately accessible to practitioners, while needing to maintain infrastructure and support for the tools it already has. We argue that sustaining this perpetual integration should be more widely incentivized by the community, unglamorous as it may be. It is one of the most impactful investments the formal methods community can make toward broader adoption.

AI-Use Transparency Statement. Portions of this work were supported through the use of Claude Sonnet 4.6 and Claude Opus 4.6. All interactions with these models were performed exclusively by the co-authors affiliated with Iowa State University. The models were used solely for code assistance or other computational support as described in the paper. The co-authors who utilized these models verified the accuracy, validity, and appropriateness of all material derived from it prior to inclusion in the final manuscript.

Disclaimer. All activities involving the use of Claude Sonnet 4.6 and Claude Opus 4.6 were conducted exclusively by the co-authors affiliated with Iowa State University. Co-authors affiliated with Collins Aerospace did not participate in, contribute to, supervise, review, or otherwise engage in any work involving the use of Claude in the development of this paper. Collins Aerospace neither endorses, sponsors, approves of, nor assumes any responsibility for the use of Claude or for any outcomes, analyses, or conclusions derived from it.

References

1. Alur, R., Henzinger, T.A.: Real-time Logics: Complexity and Expressiveness. In: LICS. pp. 390–401. IEEE (1990)
2. Aurandt, A., Johannsen, C., Katis, A., Mavridou, A., Rozier, K.Y., Jones, P.H.: From natural language requirements to runtime monitors for resource-constrained systems: Integrating FRET and R2U2. In: NASA Formal Methods Symposium. Springer (2026)
3. Aurandt, A., Jones, P.H., Rozier, K.Y.: Runtime verification triggers real-time, autonomous fault recovery on the CySat-I. In: NASA Formal Methods Symposium. pp. 816–825. Springer (2022). https://doi.org/https://doi.org/10.1007/978-3-031-06773-0_45
4. Aurandt, A., Jones, P.H., Rozier, K.Y.: Towards a safe, verified runtime monitor for embedded systems: R2U2 in embedded rust. In: NASA Formal Methods Symposium. pp. 31–53. Springer (2025). https://doi.org/https://doi.org/10.1007/978-3-031-93706-4_3
5. Aurandt, A., Rozier, K.Y., Jones, P.H.: R2U2 playground: Visualization of a real-time, temporal logic runtime monitor. In: Formal Methods in Computer Aided Design. pp. 126–132. TU Wien Academic Press (2025). https://doi.org/10.34727/2025/isbn.978-3-85448-084-6_18

6. Aurandt, A.A., Jones, P.H., Rozier, K.Y., Wongpiromsarn, T.: Multimodal model predictive runtime verification for safety of autonomous cyber-physical systems. In: Haxthausen, A.E., Serwe, W. (eds.) Formal Methods for Industrial Critical Systems - 29th International Conference, FMICS 2024, Milan, Italy, September 9-11, 2024, Proceedings. pp. 220–244. Lecture Notes in Computer Science, Springer (2024). https://doi.org/10.1007/978-3-031-68150-9_13, https://doi.org/10.1007/978-3-031-68150-9_13
7. Barrett, C.W., Stump, A., Tinelli, C.: The smt-lib standard version 2.0 (2010), https://api.semanticscholar.org/CorpusID:7943149
8. Bartocci, E., Deshmukh, J., Donzé, A., Fainekos, G., Maler, O., Ničković, D., Sankaranarayanan, S.: Specification-based monitoring of cyber-physical systems: a survey on theory, tools and applications. Lectures on Runtime Verification: Introductory and Advanced Topics pp. 135–175 (2018). https://doi.org/10.1007/978-3-319-75632-5_5
9. Bombara, G., Belta, C.: Offline and online learning of signal temporal logic formulae using decision trees. ACM Trans. Cyber Phys. Syst. **5**(3), 22:1–22:23 (2021). https://doi.org/10.1145/3433994, https://doi.org/10.1145/3433994
10. Bombara, G., Vasile, C.I., Penedo, F., Yasuoka, H., Belta, C.: A decision tree approach to data classification using signal temporal logic. In: Abate, A., Fainekos, G. (eds.) Proceedings of the 19th International Conference on Hybrid Systems: Computation and Control, HSCC 2016, Vienna, Austria, April 12-14, 2016. pp. 1–10. ACM (2016). https://doi.org/10.1145/2883817.2883843, https://doi.org/10.1145/2883817.2883843
11. Bozzano, M., Cavada, R., Cimatti, A., Dorigatti, M., Griggio, A., Mariotti, A., Micheli, A., Mover, S., Roveri, M., Tonetta, S.: nuXmv 2.1.0 User Manual. Fondazione Bruno Kessler (2024), https://es-static.fbk.eu/tools/nuxmv/downloads/nuxmv-user-manual.pdf
12. Cauwels, M., Hammer, A., Hertz, B., Jones, P., Rozier, K.Y.: Integrating Runtime Verification into an Automated UAS Traffic Management System. In: DETECT. Springer, L'Aquila, Italy (September 2020)
13. Cavada, R., Cimatti, A., Dorigatti, M., Griggio, A., Mariotti, A., Micheli, A., Mover, S., Roveri, M., Tonetta, S.: The nuxmv symbolic model checker. In: Biere, A., Bloem, R. (eds.) Computer Aided Verification - 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings. pp. 334–342. Lecture Notes in Computer Science, Springer (2014). https://doi.org/10.1007/978-3-319-08867-9_22, https://doi.org/10.1007/978-3-319-08867-9_22
14. Clarke, E.M., Wing, J.M.: Formal methods: state of the art and future directions. ACM Comput. Surv. **28**, 626–643 (1996), https://api.semanticscholar.org/CorpusID:5534240
15. Conrad, E., Titolo, L., Giannakopoulou, D., Pressburger, T., Dutle, A.: A compositional proof framework for FRETish requirements. In: Proceedings of the 11th ACM SIGPLAN International Conference on Certified Programs and Proofs. pp. 68–81 (2022). https://doi.org/10.1145/3497775.3503685
16. Dabney, J.B., Badger, J.M., Rajagopal, P.: Adding a verification view for an autonomous real-time system architecture. In: Proceedings of SciTech Forum. p. Online. 2021-0566, AIAA (January 2021)
17. Dabney, J.B., Badger, J.M., Rajagopal, P.: Trustworthy autonomy for gateway vehicle system manager. In: 2023 IEEE Space Computing Conference (SCC). pp. 57–62. IEEE (2023)
18. Dabney, J.B.: Using assume-guarantee contracts in autonomous spacecraft. Flight Software Workshop (FSW) Online: https://www.youtube.com/watch?v=zrtyiyNf674 (February 2021)
19. Dabney, J.B., Rajagopal, P., Badger, J.M.: Using assume-guarantee contracts for developmental verification of autonomous spacecraft. Flight Software Workshop (FSW) Online: https://www.youtube.com/watch?v=HFnn6TzblPg (February 2022)

20. Duret-Lutz, A., Lewkowicz, A., Fauchille, A., Michaud, T., Renault, E., Xu, L.: Spot 2.0—a framework for LTL and ω-automata manipulation. In: International Symposium on Automated Technology for Verification and Analysis. pp. 122–129. Springer (2016), https://spot.lre.epita.fr/
21. Dwyer, M.B., Avrunin, G.S., Corbett, J.C.: Specification patterns. https://matthewbdwyer.github.io/psp/patterns.html, accessed: 2026-04-16
22. Dwyer, M.B., Avrunin, G.S., Corbett, J.C.: Patterns in property specifications for finite-state verification. In: Proceedings of the 21st International Conference on Software Engineering (ICSE). pp. 411–420. ACM (1999). https://doi.org/10.1145/302405.302672
23. Ehlers, R.: Efficient temporal logic runtime monitoring for tiny systems. In: International Conference on Tests and Proofs. pp. 3–21. Springer (2024), https://www.ruediger-ehlers.de/papers/tap2024.pdf
24. Ehlers, R., Gavran, I., Neider, D.: Learning properties in LTL $\cap$ ACTL from positive examples only. In: 2020 Formal Methods in Computer Aided Design, FMCAD 2020, Haifa, Israel, September 21-24, 2020. pp. 104–112. IEEE (2020). https://doi.org/10.34727/2020/ISBN.978-3-85448-042-6_17, https://doi.org/10.34727/2020/isbn.978-3-85448-042-6_17
25. Elwing, J., Guzman, L.P.G., Sorkin, J., Travesset, C., Wang, Z., Rozier, K.Y.: Mission-time LTL (MLTL) formula validation via regular expressions. In: Herber, P., Wijs, A. (eds.) iFM 2023 - 18th International Conference, iFM 2023, Leiden, The Netherlands, November 13-15, 2023, Proceedings. pp. 279–301. Lecture Notes in Computer Science, Springer (2023). https://doi.org/10.1007/978-3-031-47705-8_15, https://doi.org/10.1007/978-3-031-47705-8_15
26. Fisher, M., Mascardi, V., Rozier, K.Y., Schlingloff, H., Winikoff, M., Yorke-Smith, N.: Towards a framework for certification of reliable autonomous systems. In: 20th International Conference on Autonomous Agents and Multiagent Systems (AAMAS). Journal-first (JAAMAS) track, Springer (May 2021)
27. Geist, J., Rozier, K.Y., Schumann, J.: Runtime observer pairs and bayesian network reasoners on-board FPGAs: flight-certifiable system health management for embedded systems. In: International Conference on Runtime Verification. pp. 215–230. Springer (2014)
28. Giacomo, G.D., Vardi, M.Y.: Linear temporal logic and linear dynamic logic on finite traces. In: Rossi, F. (ed.) IJCAI 2013, Proceedings of the 23rd International Joint Conference on Artificial Intelligence, Beijing, China, August 3-9, 2013. pp. 854–860. IJCAI/AAAI (2013), http://www.aaai.org/ocs/index.php/IJCAI/IJCAI13/paper/view/6997
29. Giannakopoulou, D., Mavridou, A., Rhein, J., Pressburger, T., Schumann, J., Shi, N.: Formal requirements elicitation with FRET. In: International Working Conference on Requirements Engineering: Foundation for Software Quality (REFSQ-2020) (2020)
30. Giannakopoulou, D., Pressburger, T., Mavridou, A., Schumann, J.: Automated formalization of structured natural language requirements. Information and Software Technology **137**, 106590 (2021). https://doi.org/10.1016/j.infsof.2021.106590
31. Hariharan, G., Jones, P.H., Rozier, K.Y., Wongpiromsarn, T.: Maximum satisfiability of mission-time linear temporal logic. In: International Conference on Formal Modeling and Analysis of Timed Systems. pp. 86–104. Springer (2023)
32. Hertz, B., Luppen, Z., Rozier, K.Y.: Integrating runtime verification into a sounding rocket control system. In: Proceedings of the 13th NASA Formal Methods Symposium (NFM 2021) (May 2021). https://doi.org/https://doi.org/10.1007/978-3-030-76384-8_10
33. Hinchey, M., Wang, C., McNeil, J.: Formal methods for system/software engineering: NASA & Army experiences. Presentation, NASA Goddard Space Flight Center (august 2011), https://www.nasa.gov/wp-content/uploads/2016/10/585641main_formalmethodsforsystemsoftwareengineering.pdf, accessed: 2026-04-12
34. Holzmann, G.J.: The model checker SPIN. IEEE Trans. Software Eng. **23**(5), 279–295 (1997). https://doi.org/10.1109/32.588521, https://doi.org/10.1109/32.588521

35. Hoxha, B., Dokhanchi, A., Fainekos, G.: Mining parametric temporal logic properties in model-based design for cyber-physical systems. Int. J. Softw. Tools Technol. Transf. **20**(1), 79–93 (2018). https://doi.org/10.1007/S10009-017-0447-4, https://doi.org/10.1007/s10009-017-0447-4
36. Huo, X., Liu, R., Zhang, H., Tang, Q., Liu, Q., Liu, X.: A flexible linear temporal logic-based data inference architecture for industrial process prediction. Complex Intell. Syst. **12**(3) (2026). https://doi.org/10.1007/S40747-025-02212-0, https://doi.org/10.1007/s40747-025-02212-0
37. Johannsen, C., Anderson, M., Burken, W., Diersen, E., Edgren, J., Glick, C., Jou, S., Kumar, A., Levandowski, J., Moyer, E., et al.: Openuas version 1.0. In: 2021 International Conference on Unmanned Aircraft Systems (ICUAS). pp. 1449–1458. IEEE (2021)
38. Johannsen, C., Jones, P., Kempa, B., Rozier, K.Y., Zhang, P.: R2U2 Version 3.0: Reimagining a toolchain for specification, resource estimation, and optimized observer generation for runtime verification in hardware and software. In: International Conference on Computer Aided Verification. pp. 483–497. Springer (2023), https://doi.org/10.1007/978-3-031-37709-9_23
39. Johannsen, C., Kempa, B., Jones, P.H., Rozier, K.Y., Wongpiromsarn, T.: Impossible made possible: Encoding intractable specifications via implied domain constraints. In: International Conference on Formal Methods for Industrial Critical Systems. pp. 151–169. Springer (2023). https://doi.org/10.1007/978-3-031-43681-9_9
40. Johannsen, C., Nukala, K., Dureja, R., Irfan, A., Shankar, N., Tinelli, C., Vardi, M.Y., Rozier, K.Y.: The moxi model exchange tool suite. In: Gurfinkel, A., Ganesh, V. (eds.) Computer Aided Verification. pp. 203–218. Springer Nature Switzerland, Cham (2024)
41. Johannsen, C., Jones, P.H., Rozier, K.Y., Wongpiromsarn, T.: Scalable mltl runtime monitoring and satisfiability via bit-vector encoding. In: 2025 Formal Methods in Computer-Aided Design (FMCAD). pp. 1–11. TU Wien Academic Press (2025). https://doi.org/10.34727/2025/isbn.978-3-85448-084-6_9
42. Kempa, B., Zhang, P., Jones, P.H., Zambreno, J., Rozier, K.Y.: Embedding Online Runtime Verification for Fault Disambiguation on Robonaut2. In: Proceedings of the 18th International Conference on Formal Modeling and Analysis of Timed Systems (FORMATS). Lecture Notes in Computer Science (LNCS), vol. 12288, pp. 196–214. Springer, Vienna, Austria (September 2020). https://doi.org/10.1007/978-3-030-57628-8_12, http://research.temporallogic.org/papers/KZJZR20.pdf
43. Kong, Z., Jones, A., Ayala, A.M., Gol, E.A., Belta, C.: Temporal logic inference for classification and prediction from data. In: Fränzle, M., Lygeros, J. (eds.) 17th International Conference on Hybrid Systems: Computation and Control (part of CPS Week), HSCC'14, Berlin, Germany, April 15-17, 2014. pp. 273–282. ACM (2014). https://doi.org/10.1145/2562059.2562146, https://doi.org/10.1145/2562059.2562146
44. Konrad, S., Cheng, B.H.C.: Real-time specification patterns. In: 27th International Conference on Software Engineering (ICSE 2005). pp. 372–381. ACM (2005). https://doi.org/10.1145/1062455.1062526
45. Kosaian, K., Wang, Z., Sloan, E.: Mission-time linear temporal logic. Archive of Formal Proofs (January 2025), https://isa-afp.org/entries/Mission_Time_LTL.html, Formal proof development
46. Kosaian, K., Wang, Z., Sloan, E.: Mission-time linear temporal logic formula progression. Archive of Formal Proofs (July 2025), https://isa-afp.org/entries/Mission_Time_LTL_Formula_Progression.html, Formal proof development
47. Kosaian, K., Wang, Z., Sloan, E., Rozier, K.Y.: Formalizing MLTL formula progression in isabelle/hol. In: de Paiva, V., Koepke, P. (eds.) Intelligent Computer Mathematics - 18th International Conference, CICM 2025, Brasilia, Brazil, October 6-10, 2025, Proceedings. pp. 371–391. Lecture Notes in Computer Science, Springer (2025). https://doi.org/10.1007/978-3-032-07021-0_21, https://doi.org/10.1007/978-3-032-07021-0_21

48. Li, D., Cai, M., Vasile, C., Tron, R.: Learning signal temporal logic through neural network for interpretable classification. In: American Control Conference, ACC 2023, San Diego, CA, USA, May 31 - June 2, 2023. pp. 1907–1914. IEEE (2023). https://doi.org/10.23919/ACC55779.2023.10156357, https://doi.org/10.23919/ACC55779.2023.10156357
49. Li, J., Rozier, K.Y.: MLTL Benchmark Generation via Formula Progression. In: Proceedings of the 18th International Conference on Runtime Verification (RV18). Springer-Verlag, Limassol, Cyprus (November 2018)
50. Li, J., Vardi, M.Y., Rozier, K.Y.: Satisfiability checking for Mission-time LTL. In: Proceedings of 31st International Conference on Computer Aided Verification (CAV). LNCS, vol. 11562, pp. 3–22. Springer, New York, NY, USA (July 2019). https://doi.org/https://doi.org/10.1007/978-3-030-25543-5_1
51. Li, R., Gurushankar, K., Heule, M.J.H., Rozier, K.Y.: What's in a name? linear temporal logic literally represents time lines. In: IEEE Working Conference on Software Visualization, VISSOFT 2023, Bogotá, Colombia, October 1-2, 2023. pp. 73–83. IEEE (2023). https://doi.org/10.1109/VISSOFT60811.2023.00018, https://doi.org/10.1109/VISSOFT60811.2023.00018
52. Li, R., Gurushankar, K., Heule, M.J., Rozier, K.Y.: What's in a name? linear temporal logic literally represents time lines. In: 2023 IEEE Working Conference on Software Visualization (VISSOFT). pp. 73–83. IEEE (2023). https://doi.org/10.1109/VISSOFT60811.2023.00018
53. Lourenço, N., Assunção, F., Pereira, F.B., Costa, E., Machado, P.: Structured Grammatical Evolution: A Dynamic Approach, pp. 137–161. Springer International Publishing (2018). https://doi.org/10.1007/978-3-319-78717-6_6
54. Luo, W., Liang, P., Du, J., Wan, H., Peng, B., Zhang, D.: Bridging ltlf inference to GNN inference for learning ltlf formulae. In: Thirty-Sixth AAAI Conference on Artificial Intelligence, AAAI 2022, Thirty-Fourth Conference on Innovative Applications of Artificial Intelligence, IAAI 2022, The Twelveth Symposium on Educational Advances in Artificial Intelligence, EAAI 2022 Virtual Event, February 22 - March 1, 2022. pp. 9849–9857. AAAI Press (2022). https://doi.org/10.1609/AAAI.V36I9.21221, https://doi.org/10.1609/aaai.v36i9.21221
55. Luppen, Z., Jacks, M., Baughman, N., Stilic, M., Nasers, R., Hertz, B., Cutler, J., Lee, D.Y., Rozier, K.Y.: Elucidation and analysis of specification patterns in aerospace system telemetry. In: NASA Formal Methods Symposium. pp. 527–537. Springer (2022). https://doi.org/10.1007/978-3-031-06773-0_28
56. Luppen, Z.A., Lee, D.Y., Rozier, K.Y.: A Case Study in Formal Specification and Runtime Verification of a CubeSat Communications System. In: SciTech. AIAA, Nashville, TN, USA (January 2021). https://doi.org/10.2514/6.2021-0997.c1
57. Maler, O., Nickovic, D.: Monitoring temporal properties of continuous signals. In: Proceedings of Formal Techniques, Modelling and Analysis of Timed and Fault-Tolerant Systems (FORMATS). pp. 152–166 (2004)
58. Marzen, L.: libmltl: A high-performance library for mission-time linear temporal logic (MLTL) parsing, AST manipulation, and formula evaluation. https://github.com/lmarzen/libmltl (2025), gitHub repository
59. Moosbrugger, P., Rozier, K.Y., Schumann, J.: R2U2: Monitoring and Diagnosis of Security Threats for Unmanned Aerial Systems pp. 1–31 (April 2017). https://doi.org/10.1007/s10703-017-0275-x
60. Nenzi, L., Silvetti, S., Bartocci, E., Bortolussi, L.: A robust genetic algorithm for learning temporal specifications from data. In: McIver, A., Horváth, A. (eds.) Quantitative Evaluation of Systems - 15th International Conference, QEST 2018, Beijing, China, September 4-7, 2018, Proceedings. pp. 323–338. Lecture Notes in Computer Science, Springer (2018). https://doi.org/10.1007/978-3-319-99154-2_20, https://doi.org/10.1007/978-3-319-99154-2_20

61. Newcombe, C., Rath, T., Zhang, F., Munteanu, B., Brooker, M., Deardeuff, M.: How amazon web services uses formal methods. Communications of the ACM (2015), https://www.amazon.science/publications/how-amazon-web-services-uses-formal-methods
62. Raha, R., Roy, R., Fijalkow, N., Neider, D., Pérez, G.A.: Synthesizing efficiently monitorable formulas in metric temporal logic. In: Dimitrova, R., Lahav, O., Wolff, S. (eds.) Verification, Model Checking, and Abstract Interpretation - 25th International Conference, VMCAI 2024, London, United Kingdom, January 15-16, 2024, Proceedings, Part II. pp. 264–288. Lecture Notes in Computer Science, Springer (2024). https://doi.org/10.1007/978-3-031-50521-8_13, https://doi.org/10.1007/978-3-031-50521-8_13
63. Reinbacher, T., Rozier, K.Y., Schumann, J.: Temporal-logic based runtime observer pairs for system health management of real-time systems. In: Proceedings of the 20th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS). Lecture Notes in Computer Science (LNCS), vol. 8413, pp. 357–372. Springer-Verlag (April 2014)
64. Rosentrater, A., Rozier, K.Y.: FPROGG: A formula progression-based MLTL benchmark generator (May 2024). https://doi.org/10.5281/zenodo.11100118, https://doi.org/10.5281/zenodo.11100118
65. Roy, R., Gaglione, J., Baharisangari, N., Neider, D., Xu, Z., Topcu, U.: Learning interpretable temporal properties from positive examples only. In: Williams, B., Chen, Y., Neville, J. (eds.) Thirty-Seventh AAAI Conference on Artificial Intelligence, AAAI 2023, Thirty-Fifth Conference on Innovative Applications of Artificial Intelligence, IAAI 2023, Thirteenth Symposium on Educational Advances in Artificial Intelligence, EAAI 2023, Washington, DC, USA, February 7-14, 2023. pp. 6507–6515. AAAI Press (2023). https://doi.org/10.1609/AAAI.V37I5.25800, https://doi.org/10.1609/aaai.v37i5.25800
66. Rozier, K.Y.: Specification: The biggest bottleneck in formal methods and autonomy. In: Proceedings of 8th Working Conference on Verified Software: Theories, Tools, and Experiments (VSTTE 2016). LNCS, vol. 9971, pp. 1–19. Springer-Verlag, Toronto, ON, Canada (July 2016). https://doi.org/10.1007/978-3-319-48869-1_2
67. Rozier, K.Y., Dureja, R., Irfan, A., Johannsen, C., Nukala, K., Shankar, N., Tinelli, C., Vardi, M.Y.: Moxi: An intermediate language for symbolic model checking. In: Proceedings of the 30th International Symposium on Model Checking Software (SPIN). LNCS, Springer (April 2024) (2024)
68. Rozier, K.Y., Vardi, M.Y.: Deterministic compilation of temporal safety properties in explicit state model checking. In: Biere, A., Nahir, A., Vos, T.E.J. (eds.) Hardware and Software: Verification and Testing - 8th International Haifa Verification Conference, HVC 2012, Haifa, Israel, November 6-8, 2012. Revised Selected Papers. pp. 243–259. Lecture Notes in Computer Science, Springer (2012). https://doi.org/10.1007/978-3-642-39611-3_23, https://doi.org/10.1007/978-3-642-39611-3_23
69. Rozier, K.: Linear Temporal Logic Symbolic Model Checking. Computer Science Review Journal **5**(2), 163–203 (May 2011). https://doi.org/doi:10.1016/j.cosrev.2010.06.002, http://dx.doi.org/10.1016/j.cosrev.2010.06.002
70. Rozier, K., Vardi, M.: LTL satisfiability checking. International Journal on Software Tools for Technology Transfer (STTT) **12**(2), 123 – 137 (March 2010). https://doi.org/DOI10.1007/s10009-010-0140-3, http://dx.doi.org/10.1007/s10009-010-0140-3
71. Schumann, J., Roychoudhury, I., Kulkarni, C.: Diagnostic reasoning using prognostic information for unmanned aerial systems. In: Annual Conference of the PHM Society. vol. 7 (2015)
72. Schumann, J., Rozier, K.Y., Reinbacher, T., Mengshoel, O.J., Mbaya, T., Ippolito, C.: Towards real-time, on-board, hardware-supported sensor and software health management for unmanned aerial systems. In: International Journal of Prognostics and Health Management (2015)

73. Vázquez, G., Mavridou, A., Farrell, M., Pressburger, T., Calinescu, R.: Robotics: A new mission for FRET requirements. In: NASA Formal Methods Symposium. pp. 359–376. Springer (2024). https://doi.org/10.1007/978-3-031-60698-4_22
74. Wang, Z., Guzman, L.P.G., Rozier, K.Y.: WEST: interactive validation of mission-time linear temporal logic (MLTL). Sci. Comput. Program. **248**, 103365 (2026). https://doi.org/10.1016/J.SCICO.2025.103365, https://doi.org/10.1016/j.scico.2025.103365
75. Wang, Z., Kosaian, K.: Mission-time linear temporal logic to regular expressions. Archive of Formal Proofs (January 2025), https://isa-afp.org/entries/Mission_Time_LTL_to_Regular_Expression.html, Formal proof development
76. Wang, Z., Kosaian, K., Rosentrater, A.: Language partitioning for mission-time linear temporal logic. Arch. Formal Proofs **2025** (2025), https://www.isa-afp.org/entries/Mission_Time_LTL_Language_Partition.html
77. Wang, Z., Kosaian, K., Rosentrater, A.: Language partitioning for mission-time linear temporal logic. Archive of Formal Proofs (March 2025), https://isa-afp.org/entries/Mission_Time_LTL_Language_Partition.html, Formal proof development
78. Wang, Z., Kosaian, K., Rozier, K.Y.: Formally verifying a transformation from MLTL formulas to regular expressions. In: Gurfinkel, A., Heule, M. (eds.) Tools and Algorithms for the Construction and Analysis of Systems - 31st International Conference, TACAS 2025, Held as Part of the International Joint Conferences on Theory and Practice of Software, ETAPS 2025, Hamilton, ON, Canada, May 3-8, 2025, Proceedings, Part I. pp. 254–275. Lecture Notes in Computer Science, Springer (2025). https://doi.org/10.1007/978-3-031-90643-5_13, https://doi.org/10.1007/978-3-031-90643-5_13
79. Wang, Z., Marzen, L.: MLTL inference. https://github.com/zwang271/mltl-inference (2026), gitHub repository
80. Zeng, X., Wang, M., Wang, M., Lu, X.: $\text{Cpd}_{\text{rmltlf}}$: a linear temporal logic over finite traces guided model checking tool. Softw. Qual. J. **33**(4), 32 (2025). https://doi.org/10.1007/S11219-025-09729-0, https://doi.org/10.1007/s11219-025-09729-0
81. Zhang, P., Aurandt, A., Dureja, R., Jones, P.H., Rozier, K.Y.: Model predictive runtime verification for cyber-physical systems with real-time deadlines. In: International Conference on Formal Modeling and Analysis of Timed Systems. pp. 158–180. Springer (2023), https://research.temporallogic.org/papers/ZADJR23.pdf