Formally Verifying a Transformation from MLTL Formulas to Regular Expressions

Zili Wang[1[0000-0003-1730-6180]], Katherine Kosaian[2[0000-0002-9336-6006]], and Kristin Yvonne Rozier[1[0000-0002-6718-2828]]

[1] Iowa State University, Ames, IA, USA
[2] University of Iowa, Iowa City, IA, USA
{ziliw1,kyrozier}@iastate.edu, katherine-kosaian@uiowa.edu

Abstract. Mission-time Linear Temporal Logic (MLTL), a widely used subset of popular specification logics like STL and MTL, is often used to model and verify real world systems in safety-critical contexts. As the results of formal verification are only as trustworthy as their input specifications, the WEST tool was created to facilitate writing MLTL specifications. Accordingly, it is vital to demonstrate that WEST itself works correctly. To that end, we verify the WEST algorithm, which converts MLTL formulas to (logically equivalent) regular expressions, in the theorem prover Isabelle/HOL. Our top-level result establishes the correctness of the regular expression transformation; we then generate a code export from our verified development and use this to experimentally validate the existing WEST tool. To facilitate this, we develop some verified support for checking the equivalence of two regular expressions.

Keywords: MLTL · Regular Expressions · Interactive Theorem Proving · Isabelle/HOL · Code Generation · Tool Validation

1 Introduction

As formal methods tools become increasingly integrated into system development life cycles, it is necessary to offer stronger demonstrations of their correct implementation than piecemeal code analysis and experimental validation. After all, these are the tools justifying and verifying, e.g., the certification of systems; these tools must obey a higher standard for correctness. This starts with their input languages and specification validation.

Many formal methods tools, such as model checkers and runtime verification engines, reason over behavior specifications in LTL or related linear-time logics that extend LTL, e.g., to add intervals on the temporal operators like Signal Temporal Logic (STL) [32], Metric Temporal Logic (MTL) [1], and Metric Interval Temporal Logic (MITL) [2]. Mission-time Linear Temporal Logic (MLTL) [40,30] represents a commonly used subset of these timed logics, and has a conversion to LTL [30]. Several tools use MLTL as a core specification language; these include the Formal Requirements Elicitation Tool (FRET) [19,34,4], the

Realizable Responsive Unobtrusive Unit (R2U2) [40,43,23], and the Ogda runtime monitoring tool [37,35,36]. Popular symbolic model checker nuXmv [9] supports a subset of MLTL [25] by allowing bounds on the Globally and Future operators (but not on Until or Release). The WEST tool [17,51] transforms MLTL formulas into logically equivalent (and easier to analyze) regular expressions and facilitates the validation of MLTL specifications with an interactive GUI. Since WEST validates specifications, which are the fundamental basis for formal verification, it is especially critical to rigorously establish its correctness.

The research community has long recognized that specification is the biggest bottleneck in formal methods [42]; to that end LTL is formalized in Coq [14], in PVS [38], and in Isabelle/HOL [47], along with many algorithms for its use in formal verification [48,45,46,44,18]. Libraries for related linear-time logics were inspired by, or directly built upon those for LTL, including formalizations of MTL in Coq [10] and PVS [12,50]; a PVS formalization of MITL [41]; and Isabelle formalizations of the 3-valued variant LTL3 [3] and MLTL [27]. Further, the importance of ensuring correctness of formal methods tools naturally prompts using these formalizations to generate tools. For instance, an Isabelle/HOL formalization of the VeriMon tool for monitoring metric first-order temporal logic (MFOTL) generates (via code export) VeriMon's codebase [8]. An Isabelle/HOL formalization of a metric dynamic logic (MDL) runtime monitoring algorithm also generated the Vydra tool [39].[3] In Coq, a formalization of monitoring past-time MTL generates an OCaml monitoring engine [11].

We enrich this space by formalizing the WEST algorithm for specification validation. Building on an existing MLTL library in Isabelle/HOL [27,26], we formally prove that the WEST algorithm generates regular expressions that are logically equivalent to the input MLTL formulas, filling in details omitted from the original tool's correctness proofs. From our formalized algorithms, we generate a new implementation of WEST to validate the (unverified) implementations of WEST: the proof-of-concept original [17] and a highly optimized refactoring [51]. As WEST validates other MLTL tools, most notably the runtime verification engine R2U2 [40], our work helps to foster trust in a safety-critical space. Our experiments also show that our Isabelle-generated code is (in aggregate) close in performance to the optimized, unverified version of WEST.

Section 2 recaps the existing Isabelle/HOL MLTL library [27,26], introduces the trace regular expressions fundamental to the WEST algorithm, and sets up the definitions underlying our formalization. Section 3 presents our formalization of the WEST algorithm. Section 4 gathers our formalization insights to inform future efforts that build on our contributions. Section 5 experimentally evaluates the new version of WEST generated via Isabelle's code export utility in comparison with two previous, hand-coded versions [17,51], while Section 6 concludes with a discussion. Our formalization (totaling $\approx$ 7400 lines of code) is available on the Archive of Formal Proofs (AFP) [52].

[3] Vydra also reasons with regular expressions in the input language, rather than using regular expression to represent the input, as WEST does.

2 MLTL and Regular Expressions

In this section, we present the syntax and semantics of MLTL and explain our formalization of the WEST regular expressions used by the WEST tool, highlighting some key datatypes; when appropriate, we intersperse mathematical definitions with Isabelle/HOL code. We also introduce some useful functions that are important in the correctness proofs later on.

Other works formalize regular expressions in different contexts. An algorithm for matching extended regular expressions via symbolic derivatives was formalized in Lean [56], and the Myhill-Nerode theorem was restated in Isabelle/HOL using regular expressions (instead of automata, which is more common) [54]. There has also been work formalizing decision procedures to check equivalence of regular expressions in Rocq [13] and Isabelle/HOL [29]. The latter is particularly relevant; we are interested in potentially incorporating it in future work to improve our (currently naive) regular expression checking procedure.

2.1 Syntax and Semantics of MLTL

Let $\mathtt{AP}$ be a finite set of atomic propositions. Let $\mathtt{p} \in \mathtt{AP}$ be an atomic proposition, and $a, b \in \mathbb{N}$ be natural numbers such that $a \leq b$; MLTL formulas are defined by the following grammar; the temporal operators $\mathtt{F}, \mathtt{G}, \mathtt{U}, \mathtt{R}$ denote "Future", "Globally", "Until", and "Release", respectively.

$$\phi, \psi := \mathtt{True} \mid \mathtt{False} \mid \mathtt{p} \mid \neg\phi \mid \phi \wedge \psi \mid \phi \vee \psi \mid \mathtt{F}_{[a,b]}\phi \mid \mathtt{G}_{[a,b]}\phi \mid \phi \mathtt{U}_{[a,b]}\psi \mid \phi \mathtt{R}_{[a,b]}\psi.$$

A **trace** π is a finite sequence $\pi = \pi[0], \pi[1], \ldots$ of sets of atomic propositions, where $\pi[i] \subseteq \mathtt{AP}$ for all i. We refer to the i-th element of a trace π as the i-th state of the trace, and intuitively interpret $\pi[i]$ as the set of propositions that are true at time i. We denote the length of a trace π by $|\pi|$, and the suffix of a trace π starting at time i by π_i; that is, $\pi_i = \pi[i], \pi[i+1], \ldots$ and $\pi_0 = \pi$. The existing MLTL library in Isabelle/HOL [26] encodes a trace as a list of sets of natural numbers; each set represents the atomic propositions that are true at each timestep. For example, the trace $\pi = \{p_0, p_1\}, \{p_0\}$ is encoded in Isabelle as the `[{0, 1}, {0}]`, which has type `nat set list`.

A trace π satisfies an MLTL formula ϕ, denoted $\pi \models \phi$, as follows [40,30], where ψ is another MLTL formula:

$\pi \models p$ iff $p \in \pi[0]$ $\qquad$ $\pi \models \neg\phi$ iff $\pi \not\models \phi$

$\pi \models \phi \wedge \psi$ iff $\pi \models \phi$ and $\pi \models \psi$ $\qquad$ $\pi \models \phi \vee \psi$ iff $\pi \models \phi$ or $\pi \models \psi$

$\pi \models \mathtt{F}_{[a,b]}\phi$ iff $|\pi| > a$ and $\exists i \in [a, b].\ \pi_i \models \phi$

$\pi \models \mathtt{G}_{[a,b]}\phi$ iff $|\pi| \leq a$ or $\forall i \in [a, b].\ \pi_i \models \phi$

$\pi \models \phi\ \mathtt{U}_{[a,b]}\psi$ iff $|\pi| > a$ and $\exists i \in [a, b].\ (\pi_i \models \psi$ and $\forall j \in [a, i-1].\ \pi_j \models \phi)$

$\pi \models \phi\ \mathtt{R}_{[a,b]}\psi$ iff $|\pi| \leq a$ or $(\forall i \in [a, b].\ \pi_i \models \psi)$ or $\exists j \in [a, b-1].\ (\pi_j \models \phi$ and $\forall k \in [a, j]\ \pi_k \models \psi)$

2.2 Trace Regular Expressions

The WEST algorithm [17] takes an MLTL formula as input and recursively computes a WEST regular expression representing exactly the set of traces that satisfy that formula. Intuitively, we can think of this as happening in two steps. First, we represent traces as **bit strings**; here, instead of encoding each state in a trace as a set, we encode each state as a bit string of length n (where n is the number of variables in the formula). Next, we define **WEST regular expressions (WEST regexes)** as a compact way to represent a set of traces.

More precisely, we assume that $\texttt{AP} = \{p_0, p_1, \ldots, p_{n-1}\}$ and impose (without loss of generality) an ordering on these atomic propositions; we use this ordering to construct the **bit string** of a trace π of length m as the length mn string of 0's and 1's such that the value of atomic proposition p_k at timestep i corresponds to the $(ni + k)$-th character of the bit string [17, Definition 2]. We visualize an example in Fig. 1. We encode bit strings in Isabelle as lists of lists.

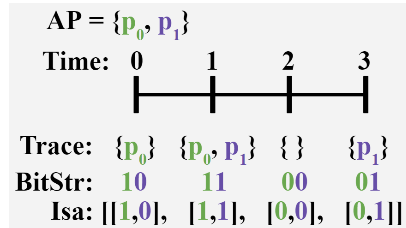

Fig. 1: For $\texttt{AP} = \{p_0, p_1\}$, the bit string of trace $\{p_0\}, \{p_0, p_1\}, \{\}, \{p_1\}$ is `10,11,00,01` (following the source material [17], we use commas to separate timesteps for readability) which is encoded in Isabelle as *`[[1,0], [1, 1], [0, 0], [0, 1]]`* (type *`nat list list`*).

In Isabelle/HOL, we obtain an ordering on our set of atomic propositions by constraining them to be natural numbers, of type *`nat`*. Following WEST's implementation [51], we choose not to fix n globally (which we could accomplish using a locale [6,7]) but instead pass the number of variables as an argument to the helper functions in the WEST algorithm (in the top-level function, we compute the right value to pass to the helper functions).

We then collate these bit string representations in **trace regular expressions**,[4] or trace regexes for short, which are strings consisting of `0`, `1`, and `S`, where `S` is a shorthand for the regular expression `0`|`1`. For example, fixing the number of atomic propositions to be $n = 3$, the trace regex `10S` matches only the two bit strings `101` and `100` (each representing a trace of length 1), and the trace regex `S00,0S0` matches the four bit strings (each representing a length 2 trace) "`100,010`", "`100,000`", "`000,010`", and "`000,000`".

In Isabelle/HOL, trace regexes have type *`WEST_bit list list`*, where our custom datatype *`WEST_bit`* is comprised by *`Zero`*, *`One`*, and *`S`*. We represent trace regexes with *`WEST_bit list list`* and not *`WEST_bit list`* because the number of atomic propositions, n, is critical for the interpretation of traces from their bit string representations. We must ensure that each *`WEST_bit list`*, referred to as a *state regex*, has length n in the overall list; having a list of lists facilitates this check. For this, we define the function *`trace_regex_of_vars`* which takes as inputs trace regex *`r`* and the number of atomic propositions *`n`*, and checks that each state regex in *`r`* has length *`n`*. Here, *`!`* is Isabelle/HOL syntax for the `i`-th element of *`L`*.

[4] Also called temporal regular expressions [17, Definition 4].

```
definition trace_regex_of_vars::"trace_regex ⇒ nat ⇒ bool"
  where "trace_regex_of_vars r n = (∀i<length r. length (r!i) = n)"
```

Then, we build a list of trace regexes as a `WEST_regex` of type `WEST_bit list list list`, the final return type of the WEST algorithm. A WEST regex `L` is well-defined for n atomic propositions if each trace regex `r` in `L` satisfies `trace_regex_of_vars r n`. We summarize the datatypes of objects in our encoding in Table 1. While the nested lists may seem unwieldy at first glance, they ensure modularity in the implementation and, more crucially, in the correctness proofs. We turn to an example of this modularity now, as we build up to formalizing the notion of a WEST regex matching a trace.

Terminology	**Description**	**Isabelle Type**
WEST bit	Custom Isabelle datatype	`WEST_bit`
state regex	List of WEST bits that encodes states as bit strings	`WEST_bit list`
trace regex	List of WEST states that represents sets of traces compactly as regular expressions	`WEST_bit list list`
WEST regex	List of WEST traces that represents the union of all sets of traces represented by the WEST traces	`WEST_bit list list list`

Table 1: Summary of the datatypes of each object in our encoding.

2.3 Useful Definitions

The notion of matching is foundational to the WEST algorithm because it is crucial for connecting the semantics of MLTL formulas to the semantics of WEST regexes. We define that a state regex r **matches** a state if r equals the bit string representation of the state or if r generalizes the bit string by replacing some characters in the bit string with `S`'s. This notion lifts to traces: a trace regex r matches a trace π iff r matches the bit string representation of π. Furthermore, we may lift this to WEST regexes. For trace regexes $r_1, r_2, ..., r_k$, we can combine them by alternations as $r_1|r_2|...|r_k$; we abbreviate this as the WEST regex $L = [r_1, r_2, ..., r_k]$, and define that L matches a trace π iff some r_i matches π.

We contribute a formal mathematical definition of the notion of matching, which previous work [17] supplied only an intuition for. We do this in three steps. First, we define matching a state regex (of type `WEST_bit list`) to a state in a trace (of type `nat set`) in the definition `match_timestep`:

```
definition match_timestep:: "nat set ⇒ state_regex ⇒ bool"
    where "match_timestep state r = (∀i < length r.
  (r ! i = One ⟶ i ∈ state) ∧ (r ! i = Zero ⟶ i ∉ state))"
```

This definition checks that for all `i`, `r!i` equaling `One` implies the i-th atomic proposition p_i holds at the input state (i.e., $p_i \in$ `state`), and `r!i` equaling `Zero` implies p_i does not hold at this state. If `r!i` is `S`, then p_i can be either true or false at this state. For example, the state regex $[0, 1, S]$ matches $\{1\}$ and $\{1, 2\}$.

Next we define matching a trace regex (of type `WEST_bit list list`) to a trace (of type `nat set list`) in the definition `match_regex`:

```
definition match_regex:: "trace ⇒ trace_regex ⇒ bool"
  where "match_regex π r = ((∀ time<length r.
  (match_timestep (π ! time) (r ! time)))∧(length π ≥ length r))"
```

This definition takes as input a trace π and a trace regex *r*, and checks that *match_timestep* holds for all regex states in *trace* (i.e., for all *r ! time*) on the corresponding state in the trace (π *! time*). It also checks that the length of π is at least the length of *r* (a well-definedness condition, as we need to access π *! time* for all time up to the length of *r*).

Finally, we define matching a WEST regex (of type *WEST_bit list list list*) to a trace (of type *nat set list*) in the definition *match*:

```
definition match:: "trace ⇒ WEST_regex ⇒ bool"
    where "match π L = (∃i < length L. match_regex π (L ! i))"
```

This definition checks that *match_regex* holds for some trace regex *L ! i* in *L* and the trace π. We may intuitively view WEST regexes as compactly representing the behavior of a set of traces; then, the WEST algorithm transforms a given MLTL formula into a WEST regex that captures the set of satisfying traces.

Another important function, *WEST_num_vars*, counts the number of atomic propositions in a given MLTL formula by recursively computing the maximum number of atomic propositions in all subformulas. For example, *WEST_num_vars* of an atomic proposition *p* is *p+1* (as atomic propositions are indexed from 0), and *WEST_num_vars* of *And_mltl* φ ψ is the maximum of *WEST_num_vars* φ and *WEST_num_vars* ψ. This function is used frequently in our correctness results.

3 Formalizing the WEST Algorithm

Intuitively, the WEST algorithm recursively computes a list of trace regexes for the subformulas of an MLTL formula, and then combines these lists using the `WEST_and` and `WEST_or` operations for taking intersections and unions of sets of traces. The finite semantics of MLTL formulas ensures that all existential and universal quantifiers can be translated to a finite number of `WEST_and` and `WEST_or` operations on trace regexes; thus the WEST algorithm directly defines the temporal operators in terms of `WEST_and` and `WEST_or`. For these temporal operators, we also need a *shifting* operation, *shift*, which the source material [17] implicitly uses but does not explicitly define. Intuitively, *shift* ensures that we are analyzing the locations

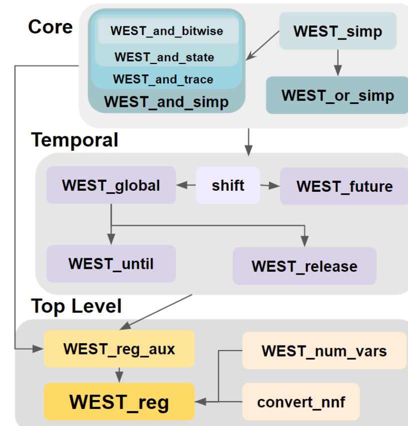

Fig. 2: High-level overview of key components in our formalization of the WEST algorithm.

in the trace specified by the temporal operators; we will see this in an example in Sect. 3.2. Fig. 2 visualizes the overall structure of the WEST algorithm.

We first discuss our formalization of the core operators *WEST_and* and *WEST_or* along with our formalization of an important simplification step in Sect. 3.1. Then, we present how the temporal operators are built on top of these core operators in Sect. 3.2, using the *shift* operation. Finally, we discuss the top-level WEST algorithm *WEST_reg* and our overall correctness result in Sect. 3.3.

3.1 The Core Operations of WEST

The `WEST_or` operation simply combines two WEST regexes (i.e., lists of trace regexes) into one WEST regex. We implement this in Isabelle/HOL using the built-in *@* operator for list concatenation. The top-level correctness theorem shows that for two WEST regexes *L1* and *L2*, *L1* matches a trace π or *L2* matches π iff *L1@L2* matches π. We formally state this as the *WEST_or_correct* lemma.

```
lemma WEST_or_correct:
      fixes π::"trace" and L1 L2::"WEST_regex"
      shows "match π (L1@L2) ⟷ (match π L1) ∨ (match π L2)"
```

Next, the `WEST_and` operation takes as input two lists of trace regexes and computes a list of trace regexes representing the intersection of the sets of traces represented by the input lists. We visualize the intended semantics of this operation in Fig. 3. One notable point here is that *WEST_and Zero One* is *None*, because it is impossible for a bit in a trace regex to simultaneously equal *Zero* and *One*. In Isabelle/HOL, we formalize `WEST_and` in four steps: first we define an operation between two bits, then between two regex states, then between two trace regexes, and finally between two WEST regexes.

The lowest-level operation between two bits (each of type *WEST_bit*) is defined in the function *WEST_and_bitwise* as follows:

```
fun WEST_and_bitwise:: "WEST_bit⇒WEST_bit⇒WEST_bit option" where
   "WEST_and_bitwise b One = (if b=Zero then None else Some One)"
   | "WEST_and_bitwise b Zero = (if b=One then None else Some Zero)"
   | "WEST_and_bitwise b S = Some b"
```

This operation reflects the desired semantics visualized in Fig. 3 by using option types to return *None* when the set intersection is empty. For example, *WEST_and_bitwise S Zero* is *Some Zero*, while *WEST_and_bitwise One Zero* is *None*.

This operation is then lifted to two regex states in *WEST_and_state*; here, we apply *WEST_and_bitwise* to each pair of corresponding bits in the two regex states. If *None* is returned for any pair, then the function returns *None* for the entire regex state. Note that the lengths of the two regex states must be the same (i.e., equal to n, the number of atomic propositions), and this operation returns *None* if they are not. Then, we again lift *WEST_and_state* to operate on two trace regexes in the function *WEST_and_trace* by applying *WEST_and_state* to each pair of corresponding regex states in the two trace regexes, returning *None*

if any of the calls to *WEST_and_state* returns *None*. The input trace regexes are allowed to have different lengths, and the shorter trace regex is treated as if the missing regex states are all *S*, following [17, Definition 4, Pad]. The full formal definitions can be found in our formalization [52].

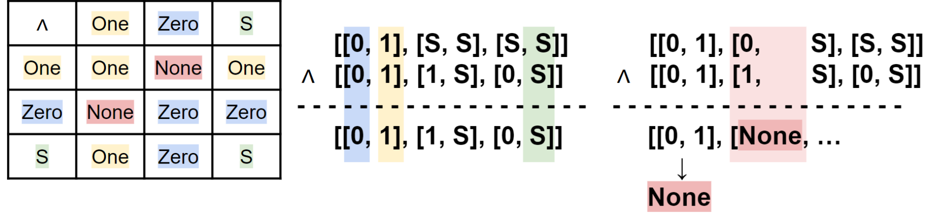

Fig. 3: Operations table for `WEST_and_bit` operation for bits (left), and two examples of `WEST_and` between regex states and traces (middle and right).

To establish the correctness of *WEST_and*, we prove the following lemma:

```
lemma WEST_and_correct:
    fixes π::"trace" and L1 L2:: "WEST_regex"
    assumes L1_of_num_vars: "WEST_regex_of_vars L1 n"
    assumes L2_of_num_vars: "WEST_regex_of_vars L2 n"
    shows "(match π L1 ∧ match π L2) ⟷ match π (WEST_and L1 L2)"
```

This shows that for input WEST regexes *L1* and *L2*, both *L1* and *L2* match trace π iff the *WEST_and* of *L1* and *L2* matches π. In other words, the set of traces that the *WEST_and* of *L1* and *L2* matches is exactly the intersection between the set of traces that *L1* matches and the set of traces that *L2* matches. The assumptions on *L1* and *L2* are well-definedness conditions that ensure all state regexes have length n (the number of atomic propositions), as required by *WEST_and_state*.

To keep the sizes of WEST regexes small, WEST implements an additional simplification step which collects together related trace regexes. If two trace regexes differ only by a single bit, then they may be combined into one trace regex where the differing bit is *S*. For example, fixing the number of atomic propositions to $n = 2$, the WEST regex *[[[0,0],[0,1]], [[0,0],[0,0]], [[0,1],[0,S]]]* may first be reduced (by combining the first two trace regexes) to *[[[0,0],[0,S]], [[0,1],[0,S]]]*, and then to *[[[0,S],[0,S]]]*. This is crucial for improving the tool performance, as it helps to mitigate blowup in the length of the list of trace regexes during the `WEST_and` and `WEST_or` operations [17, Section 4].

The underlying idea is straightforward: greedily simplify pairs of regexes until no more pairs can be simplified; we implement this in the *WEST_simp* function. It is crucial that the simplification step does not change the set of traces that a WEST regex matches. The following lemma shows that, for a well-defined WEST regex *L*, a trace π matches *L* iff π matches the simplification of *L*:

```
lemma WEST_simp_correct:
    fixes L::"WEST_regex" and π::"trace" and n::"nat"
```

```
      assumes "WEST_regex_of_vars L n"
      shows "match π (WEST_simp L n) ⟷ match π L"
```

Finally, we define the functions *WEST_and_simp* and *WEST_or_simp* by passing the output of *WEST_and* and *WEST_or* (respectively) to *WEST_simp*. The correctness of *WEST_and_simp* and *WEST_or_simp* follows directly from the correctness results for *WEST_and*, *WEST_or*, and *WEST_simp*.

3.2 Temporal Operators

Our formalization of the temporal operators in the WEST algorithm uses the *WEST_and_simp* and *WEST_or_simp* operators. It also uses an operation to shift regular expressions to later timesteps, which we call *shift*. Though the source material never explicitly defines this *shift* operation, it uses it implicitly and defines an analogous operation [17, Definition 5]. We formalize *shift* as follows:

```
fun shift:: "WEST_regex ⇒ nat ⇒ nat ⇒ WEST_regex"
   where "shift L n t = map (λtrace. (arbitrary_trace n t)@trace) L"
```

Here, we refer to a state regex of all *S*'s as an *arbitrary state*, and we refer to a trace regex of all arbitrary states as an *arbitrary trace* [17, Section 6]. In this snippet, *arbitrary_trace n t* constructs an arbitrary trace regex containing *t* arbitrary states of length *n*. Then, *shift* takes as input a WEST regex *L*, and appends an arbitrary trace of *t* arbitrary states to the front of each trace regex in *L*. As intuitively named, *shift* shifts all trace regexes in *L* by *t* timesteps.

For example, fixing the number of atomic propositions at $n = 2$, the WEST regex *L = [[[1,1]], [[0,0], [0,0]]]* captures that either p_0 and p_1 both need to be true at timestep 0, or p_0 and p_1 both need to be false at timesteps 0 and 1. If instead we want to delay this behavior for p_0 and p_1 by 3 timesteps, we can compute *shift L 2 3*, which returns *[[[S,S],[S,S],[S,S],[1,1]], [[S, S],[S,S],[S,S],[0,0],[0,0]]]*. The following lemma formalizes the connection between the *shift* operation for WEST regexes and the suffix of a trace:

```
lemma shift_match_property:
      assumes "length π ≥ t"
      shows "match (drop t π) L ⟷ match π (shift L num_vars t)"
```

More precisely, *shift_match_property* establishes that a sufficiently long trace π matches a WEST regex *L* shifted by *t* timesteps iff the suffix of π with *t* states removed, denoted *drop t* π, matches *L*.

Now, we demonstrate how the temporal operators are built on top of the core WEST operators. We provide for an example *WEST_global*, defined as follows:

```
fun WEST_global:: "WEST_regex ⇒ nat ⇒ nat ⇒ nat ⇒ WEST_regex"
   where "WEST_global L a b n = (if (a = b) then (shift L n a)
   else (if (a < b) then (WEST_and_simp (shift L n b)
```

```
                        (WEST_global L a (b-1) n) n) else []))"
```

WEST_global takes as input a WEST regex *L*, lower and upper interval bounds *a* and *b*, and the number of atomic propositions *n*. *WEST_global* then uses the *shift* operation to shift the input regex *L* by *b* timesteps, and computes the *WEST_and* of the shifted *L* and *WEST_global* with *b-1*. Intuitively, *L* captures a set of traces specifying some behavior at timestep 0, and the successive *shift* and *WEST_and* operations ensures that *L*'s behavior happens at all timesteps between *a* and *b*. The remaining temporal operators are defined in a similar manner, using *shift* and the core WEST operators.

We establish the correctness of the *WEST_global* operator as follows:

```
lemma WEST_global_correct:
   fixes L::"WEST_regex" and φ::"nat mltl" and π::"trace"
   assumes semantics_φ: "⋀π. (length π ≥ complen_mltl φ ⟶
                     (match π L ⟷ semantics_mltl π φ))"
   assumes L_vars: "WEST_regex_of_vars L n"
   assumes φ_vars: "WEST_num_vars φ ≤ n" and "a≤b"
   assumes trace_len: "length π ≥ (complen_mltl φ) + b"
   shows "match π (WEST_global L a b n) ⟷
          semantics_mltl π (Global_mltl φ a b)"
```

This lemma says that for a WEST regex *L* over *n* variables (assumption *L_vars*) that captures the semantics of an MLTL formula φ of at most *n* variables (assumption *semantics_*φ and φ*_vars*), and a trace π of sufficient length, *WEST_global* φ *a b n* matches π iff π satisfies the semantics of *Global_mltl* φ *a b* (representing the formula $\mathtt{G}_{[a,b]}\phi$).

Likewise, each of the remaining temporal operators has a correctness lemma that establishes the connection between the WEST regex it computes and its corresponding temporal operator. The correctness lemmas for the temporal operators totaled about 850 lines of code.

3.3 Top-Level Algorithm and Correctness

The WEST algorithm takes as input an MLTL formula ϕ in negation normal form (NNF) and recursively computes the WEST regex representing the set of traces with length at least the computation length of φ that satisfy the formula. The existing Isabelle/HOL MLTL library [27] already formalizes the computation length[5] of ϕ, denoted $\mathtt{complen}(\phi)$, which intuitively measures how much time is needed to decide the satisfiability of ϕ [17,24,27].

We formalize the WEST algorithm in the function *WEST_reg* as follows:

[5] This is also known as the *worst-case propagation delay* in the context of runtime verification [24,55].

```
fun WEST_reg:: "nat mltl ⇒ WEST_regex"
     where "WEST_reg φ = (let nnf_φ = convert_nnf φ in
                 WEST_reg_aux nnf_φ (WEST_num_vars φ))"
```

Although input formulas to the WEST algorithm must be in NNF, we allow formulas of all shapes as input and apply the `convert_nnf` function from the existing MLTL formalization [27] to transform the input formula to NNF. The resultant NNF formula `nnf_`φ and the number of atomic propositions, computed as `WEST_num_vars` φ, are then passed to the auxiliary function `WEST_reg_aux`. This auxiliary function takes two inputs (a `nat mltl` formula φ and a natural number `n` for the number of atomic propositions) and cases on the structure of φ to apply the appropriate core operators and return a WEST regex.

We consider here a few representative cases: `True`, `Prop_mltl`, `And_mltl`, and `Global_mltl` (corresponding to the cases of `True`, an atomic proposition, a conjunction, and the global operator). Mathematically, these cases are defined in the source material as follows [17]: $\texttt{reg}(\texttt{True}) = S^n$, $\texttt{reg}(p_k) = S^k 1 S^{n-k-1}$, and $\texttt{reg}(\phi \wedge \psi) = \texttt{reg}(\phi) \wedge \texttt{reg}(\psi)$. The global operation, $\texttt{reg}(\texttt{G}_{[a,b]}\phi)$ computes (recursively) the `WEST_and` of $\texttt{reg}(\phi)$ shifted by i timesteps for all i with $a \leq i \leq b$ (note this is essentially what `WEST_global` computes). In Isabelle/HOL, we have:

```
WEST_reg_aux:: "(nat) mltl ⇒ nat ⇒ WEST_regex"
 where "WEST_reg_aux True_mltl n = [[(map (λ j. S) [0 ..< n])]]"
 | "WEST_reg_aux (Prop_mltl p) n =
    [[(map (λj. (if (p=j) then One else S)) [0 ..< n])]]"
 | "WEST_reg_aux (And_mltl φ ψ) n = (WEST_and_simp
    (WEST_reg_aux φ n) (WEST_reg_aux ψ n) n)"
 | "WEST_reg_aux (Global_mltl φ a b) n =
    WEST_global (WEST_reg_aux φ n) a b n"
```

Here, `map f L` applies a function `f` on every element of a list `L`, so the base case for `True_mltl` creates a WEST regex containing a trace regex of all `S`'s. In the case `Prop_mltl p`, the map function takes as input `j` and returns `One` if the propositional variable `p` equals the index `j`, and otherwise `S`. In `And_mltl`, we directly call the `WEST_and` operator; likewise in `Global_mltl`.

Top-Level Correctness. A central contribution of our work is proving (and even slightly generalizing) the correctness of the `WEST_reg_aux` function and elucidating many of the details omitted in the original proof of correctness. Theorem 2 in the source material states the correctness result as follows: for a MLTL formula ϕ in negation normal form, a trace π with length $\texttt{complen}(\phi)$ satisfies ϕ iff π matches $\texttt{reg}(\phi)$ [17]. We formalize this in the theorem `WEST_reg_aux_correct`:

```
theorem WEST_reg_aux_correct:
   fixes π::"trace" and φ::"nat mltl" and n::"nat"
   assumes π_long_enough: "length π ≥ complen_mltl φ"
   assumes is_nnf: "∃ ψ. φ = (convert_nnf ψ)"
   assumes φ_nv: "WEST_num_vars φ ≤ n"
```

```
    assumes "intervals_welldef φ"
    shows "match π (WEST_reg_aux φ n) ⟷ semantics_mltl π φ"
```

This theorem states that for MLTL formula φ in NNF (assumption *is_nnf*) with at most *n* variables (assumption *φ_nv*) and well-defined interval bounds (assumption *intervals_welldef φ*) and a trace π of length at least complen(φ) (assumption *π_long_enough*), the trace π satisfies φ iff the trace π matches the WEST regex computed by *WEST_reg_aux φ n*. Here, the functions *convert_nnf*, *complen_mltl*, and *intervals_welldef* are from the existing MLTL formalization [26]. The *φ_nv* is an implicit assumption in the source material, which globally fixes the number of atomic propositions.[6] We slightly generalize the original correctness result, as our formal result holds for all traces of length at least the computation length of φ rather than just the traces of length equal to the computation length of φ.

We prove this by structural induction on the input formula φ. The *is_nnf* assumption allows us to use the custom induction rule *nnf_induct* from the existing MLTL formalizing [26], simplifying the induction proof. The base cases are straightforward, and the inductive cases are proven by applying the inductive hypothesis on the subformulas and using the correctness lemmas for the core WEST operators. For instance, for input formula *φ = Global_mltl ψ a b* (which is $\mathtt{G}_{[a,b]}\psi$), the inductive hypothesis gives us that the trace π satisfies ψ iff the WEST regex *L* computed by *WEST_reg_aux ψ n* matches π. Next, in order to apply the correctness result of the *WEST_global* operator, we need to show that *L* is a WEST regex over *n* atomic propositions (i.e., each state regex in each trace regex in *L* is of length *n*). For this, we prove the lemma *WEST_reg_aux_num_vars*:

```
lemma WEST_reg_aux_num_vars:
    fixes φ::"nat mltl"
    assumes is_nnf: "∃ ψ. φ = (convert_nnf ψ)"
    assumes "WEST_num_vars φ ≤ n" and "intervals_welldef φ"
    shows "WEST_regex_of_vars (WEST_reg_aux φ n) n"
```

This lemma states that, for a formula φ in NNF with at most *n* atomic propositions, the WEST regex computed by *WEST_reg_aux φ n* is a WEST regex over *n* atomic propositions. With this, we can apply the correctness result of the *WEST_global* operator on *L* and complete the proof of the *Global_mltl* case.

Finally, we present the top-level correctness result for the WEST algorithm:

```
theorem WEST_reg_correct:
     fixes φ::"nat mltl" and π::"trace"
     assumes "intervals_welldef φ"
     assumes π_long_enough: "length π ≥ complen_mltl φ"
     shows  "match π (WEST_reg φ) ⟷ semantics_mltl π φ"
```

[6] Note that we crucially assume that the number of variables of φ is $\leq n$ instead of $= n$ in order to satisfy the inductive hypothesis in our (inductive) proof.

This theorem states that for any MLTL formula φ with well-defined interval bounds [27] and any trace π of length at least the computation length of φ, π satisfies φ iff the WEST regex `WEST_reg` φ matches π. The correctness of the top-level WEST algorithm took about 600 LOC in Isabelle/HOL compared to the 60 or so lines of proof sketches in the source material [17, Appendix III].[7]

4 Formalization Insights

Retrospectively viewing our formalization at a high level, we highlight a few notable points. First, our modular definitions did considerably streamline our correctness proofs. Many proofs have relatively similar structures, which helped guide the formalization at a high level. However, we also found that relatively short proofs in the source material became lengthy in the formalization, in part because they often split into many subcases. For example, the notion of a WEST regex matching a trace is intuitively simple, but the formalization used several helper functions. As another example, the proof of `WEST_and_correct` is approximately 15 lines of a proof sketch in the source material [17, Theorem 4]. However, our formal development took approximately 1800 LOC to state and prove this result level by level, starting from the correctness of the `and` operation on state regexes, then on trace regexes, and finally on WEST regexes. Although these proofs had structural similarities, subtle differences between the operators complicated the low-level details of the proofs; for instance, the option types of `WEST_and_state` required careful analysis in the correctness proofs.

Second, our formalization makes *all* details explicit, including details omitted in the source material. Many of our formal proofs are by induction; setting up the "right" inductive structure in a formal setting requires careful analysis that is often glossed over in source material. For instance, the top-level correctness theorem required making a mathematically implicit assumption on `num_vars` explicit. Setting up this assumption in the wrong way leads to an ineffective inductive structure. As another example, in the proof of `WEST_simp_correct`, we perform a tricky induction on the difference between the length of the input WEST regex and the output simplified WEST regex. Additionally, we are required to prove that all functions terminate. For many functions, Isabelle/HOL proves this automatically [28], but we occasionally ran into cases where we had to explicitly construct a measure to prove termination. For example, the `WEST_reg_aux` function and the `WEST_simp` function required such manual termination proofs. Intuitively, `WEST_reg_aux` recurses on all subformulas in NNF, converting subformulas to NNF as necessary; accordingly, we use a termination measure that is similar to the number of nodes in the abstract syntax tree (AST) of the formula, but weighs nodes that are not in NNF more heavily. This allows us to prove that `WEST_reg_aux` terminates, as this measure strictly decreases on every recursive call. Further, for `WEST_simp`, the length of the input list is not strictly decreasing, but the list of candidate pairs for simplification will be exhausted at some point,

[7] The results leading up to this top-level theorem required an additional ≈ 5300 LOC.

so we use a measure that combines the length of the input list with the length of remaining candidate pairs.

Overall, integrating `WEST_simp` into our formalization was rather involved. Our initial formalization did not include `WEST_simp`, but we ultimately realized that it is crucial for speed and thus also important for tool validation. While the modular nature of our formalization easily allowed us to add in this function to the algorithm, its correctness proofs were intricate. Similarly to `WEST_and`, we proved the correctness of `WEST_simp` level by level, totaling around 1300 LOC.

As a final interesting point, we found during our tool validation that `WEST_reg` and the (unverified) WEST tool sometimes produce trace regexes that differ only by a string of `S`'s at the end. In such cases, because these trace regexes have different length, our equivalence checking methods spuriously identify a mismatch. The WEST tool always produces trace regexes that have the same length as the computation length of the input formula, while `WEST_reg` does not.[8] To account for this, we define a function `simp_pad_WEST_reg` which pads trace regexes to this computation length (and then simplifies). We extend the top-level correctness theorem from `WEST_reg` to `simp_pad_WEST_reg`; in our experiments, we work with `simp_pad_WEST_reg` so as to eliminate these spurious mismatches.

5 Experiments

The functions `simp_pad_WEST_reg` and `naive_equivalence` are executable in Isabelle/HOL, and we use Isabelle/HOL's code generator [20] to export these functions to Haskell.[9] We choose Haskell both to facilitate our experimental setup and because the GHC compiler [33] produces reasonably fast native machine code. We use our code export to validate two versions of the WEST tool—the initial version of WEST [17], and also a more recent version that has been highly optimized [51]. We also compare the different implementations for speed. We run all of our experiments in WSL2 on a Windows machine with an 11th generation Intel Core i7 processor and 32GB of RAM. We use an unverified parsing script to transform input MLTL formulas into the format required by our code export.[10]

Previous Validation Efforts. The most recent (and fastest) version of WEST was validated against several MLTL tools [51]: ① the original version of WEST [17], ② the runtime verification engine R2U2 [43,24,23] ③ a direct C++ implementation of MLTL semantics [51], and ④ translating MLTL formulas to propositional logic [21] and applying a BDD based AllSAT solver. The validation works by analyzing, for each formula in the test suite, whether the trace

[8] This is because we implicitly treat shorter trace regexes to have all `S`'s at the end (recall our discussion of the `WEST_and_trace` operator in Sect. 3.1).

[9] Note that, although Isabelle/HOL's code generator is not yet fully verified, exporting a formalized function is more trustworthy than simply coding a function. Additionally, some work has considered verifying Isabelle's code generator [22].

[10] There has been some recent work [49] on improving support for verified parsing in Isabelle/HOL, so verifying this parsing step might be an interesting future direction.

set of regexes produced by WEST is equivalent to the set of satisfying traces produced by other tools. The equivalence checking is a crucial step performed between outputs that can be in different formats (depending on the output format of each tool). The test suite of 1662 MLTL formulas was designed to capture every possible combination of MLTL operators [17].

5.1 Verified Equivalence Checking

Our tool validation is set up to check the outputs of our verified implementation of WEST against the two existing implementations. For this, we need to be able to check equivalences between WEST regexes. It is not always enough to merely check set equality, as implementation differences can lead to different (but logically equivalent) outputs. For instance, the two WEST regexes $[[[S, S]]]$ and $[[[S, 1]], [[1, S]], [[0, 0]]]$ are equivalent, but `WEST_simp` does not simplify the second into the first. The order in which `WEST_simp` simplifies pairs of trace regexes within a WEST regex is what causes these differences.

Developing a fully verified and optimized equivalence checking algorithm is out of scope of our work, but we still wanted a lightweight trustworthy implementation of regex equivalence checking. Accordingly, we formalize a naive equivalence checking function for WEST regexes, called `naive_equivalence`. This function works by explicitly enumerating all the trace regexes that each WEST regex produces and then checking set equality.

We then prove the experimentally relevant direction of correctness: If two WEST regexes are equivalent under our (executable) naive equivalence checking function, then they are indeed equivalent under the (non-executable) mathematical definition. Formally, we have the following lemma:

```
lemma regex_equivalence_correct:
  fixes A B::"WEST_regex"
  shows "(naive_equivalence A B) ⟶ (∀π. match π A = match π B)"
```

The proof was approximately 1150 lines of code. Although establishing both directions of equivalence here (i.e, $\longleftrightarrow$ instead of $\longrightarrow$) is theoretically desirable, the direction we verify is the experimentally significant one, since we encounter no instances where `naive_equivalence` failed in our test suite. More specifically, `naive_equivalence` holds on all but 4 of the 1662 input formulas and times out (after 4 hours) on the remaining 4 formulas. Often the outputs are identical; for example, the Isabelle implementation and the optimized WEST tool produced identical WEST regexes on 1547 of the formulas. We additionally ran the previous (unverified) equivalence checking procedure, which succeeded on all of the formulas. Collectively, these results establish strong confidence in the correctness of the (unverified) WEST tools [17,51].

5.2 Speed Comparison

The original C++ version of WEST [17] performed string-based operations, and the optimized version of WEST takes advantage of highly parallelized computa-

tions by using bitsets [51]. Although fast performance is not our primary goal, preliminary experiments demonstrate how our formalized code compares to the two unverified versions of WEST. Overall, we find that the optimized version of WEST is fast (as expected). Our Isabelle implementation also performs quite

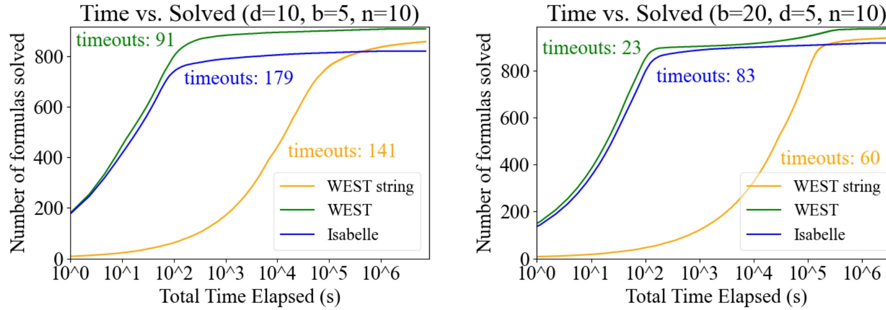

Fig. 4: Two cactus plots, each comparing the three WEST implementations on 1000 random formulas of varying nesting depth d, interval bounds b, and number of atomic propositions n. The number of total solved instances is shown on the y-axis, and the cumulative time taken is shown on the x-axis, with the number of timeouts labeled.

respectably; it is, in aggregate, close in performance to the optimized version of WEST. We perform extensive experiments to compare the performance of the three tools on large randomly generated benchmark sets. We use a script to generate random MLTL formulas [51], varying the parameters of the maximum depth and the maximum interval time bounds. Our results are in Fig. 4. As the primary focus of our work is tool validation, we do not envision our contribution as replacing the WEST tool, but its relative efficiency is encouraging nonetheless.

However, we did find that, on individual examples, our code export has somewhat unpredictable behavior (whereas the optimized version of WEST appears to be uniformly fast), and our code export seems to incur timeouts more frequently than the unverified WEST implementations. For example, in Fig. 5, we evaluate the speed of the three tools based on varying values of d, the depth of the formula, while fixing the number of atomic propositions at $n = 5$ and the maximum interval bound at $b = 2$. Here, we observe that the Isabelle implementation begins to timeout much more frequently than the other two tools when $d = 4$ and $d = 5$.

Additional results, including aggregate cactus plots on easier but larger test suites, an extension of Fig. 5 on higher values of formula depth d, and experiments where we vary the value of maximum interval bound b (instead of d), can be found in Appendix A of the full version of this paper [53].

6 Conclusion

Our work produces a third, open-source, freely available implementation of the WEST algorithm, this time *formally verified* [52]. Given the popularity of MLTL as a formal specification language for safety-critical applications [24,15,31,16,5],

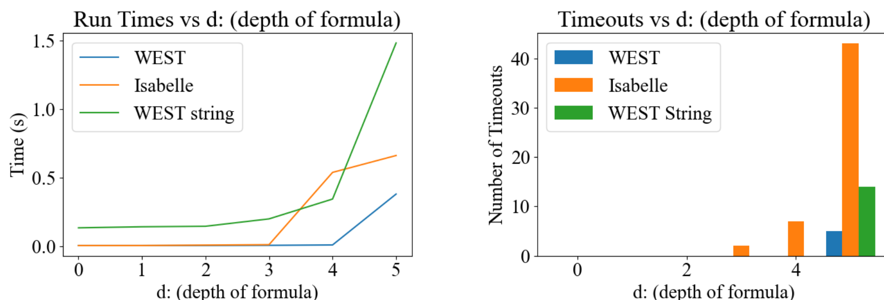

Fig. 5: Results for $n = 5$, $b = 2$, and varying values of d from 0 to 5, with a batch size of 300 formulas per value of d. The Isabelle implementation is faster than the unoptimized WEST tool on most values of d, but times out on many formulas for $d = 5$.

verifying significant algorithms like WEST, which facilitates MLTL specification, is well-justified. We build on an existing formalization of MLTL in Isabelle/HOL [27] to further develop the library of verified MLTL algorithms and properties, which could help facilitate future verified developments in this space. Our development validates the existing (unverified) WEST tool [17,51] on benchmarks from the literature, bringing us a step closer to validating other MLTL tools like R2U2 [40,23]. Though our primary focus was not on speed, the aggregate performance of our Isabelle-generated code is promising, and optimizing our formalization could be interesting future work. It would be particularly beneficial to further optimize (and verify the reverse direction of) our naive WEST regex equivalence checking, possibly using existing work [29] which verifies regex equivalence checking in a general setting. Verified parsing (to transform input formulas into the syntax required by our code export) would also be welcome. Additionally, a deeper analysis of the performance of the WEST tools and of our verified code on different classes of benchmarks could inform future verified tool generation efforts. For example, it would be interesting to experimentally compare a code export to some of the other languages supported by Isabelle/HOL, like SML and OCaml, to see if a different target language could help avoid timeouts. Importantly, our formalization of MLTL rewriting, equivalence checking, and regular expression manipulation could serve as a basis for formalizing similar utilities in logics like MTL and STL that extend MLTL.

Acknowledgments. Thanks to NSF CAREER Award CNS-1552934, NSF CCRI-2016592, and GRFP-2024364991 for supporting this work. We thank the anonymous TACAS reviewers as well as Alec Rosentrater and Laura Gamboa Guzman for their helpful feedback on the paper, and the TACAS artifact evaluators for their time.

References

1. Alur, R., Henzinger, T.A.: Real-time Logics: Complexity and Expressiveness. In: LICS. pp. 390–401. IEEE (1990)

2. Alur, R., Feder, T., Henzinger, T.A.: The Benefits of Relaxing Punctuality. In: Logrippo, L. (ed.) Proceedings of the Tenth Annual ACM Symposium on Principles of Distributed Computing, Montreal, Quebec, Canada, August 19-21, 1991. pp. 139–152. ACM (1991). https://doi.org/10.1145/112600.112613, https://doi.org/10.1145/112600.112613
3. Amjad, R., van Glabbeek, R., O'Connor, L.: Definitive set semantics for LTL3. Archive of Formal Proofs (August 2024), https://isa-afp.org/entries/LTL3_Semantics.html, Formal proof development
4. Anastasia Mavridou: Capturing and Analyzing Requirements with FRET. Presentation, nasa formal methods symposium, https://github.com/NASA-SW-VnV/fret, National Aeronautics and Space Agency, Pasadena, California, USA (May 2022)
5. Aurandt, A., Jones, P., Rozier, K.Y.: Runtime Verification Triggers Real-time, Autonomous Fault Recovery on the CySat-I. In: Proceedings of the 14th NASA Formal Methods Symposium (NFM 2022). Lecture Notes in Computer Science (LNCS), vol. 13260. Springer, Cham, Caltech, California, USA (May 2022). https://doi.org/10.1007/978-3-031-06773-0_45
6. Ballarin, C.: Locales and locale expressions in Isabelle/Isar. In: Berardi, S., Coppo, M., Damiani, F. (eds.) TYPES. LNCS, vol. 3085, pp. 34–50. Springer (2003). https://doi.org/10.1007/978-3-540-24849-1_3, https://doi.org/10.1007/978-3-540-24849-1_3
7. Ballarin, C.: Locales: A module system for mathematical theories. J. Autom. Reason. **52**(2), 123–153 (2014). https://doi.org/10.1007/S10817-013-9284-7, https://doi.org/10.1007/s10817-013-9284-7
8. Basin, D.A., Dardinier, T., Hauser, N., Heimes, L., y Munive, J.J.H., Kaletsch, N., Krstic, S., Marsicano, E., Raszyk, M., Schneider, J., Tirore, D.L., Traytel, D., Zingg, S.: VeriMon: A formally verified monitoring tool. In: Seidl, H., Liu, Z., Pasareanu, C.S. (eds.) ICTAC. LNCS, vol. 13572, pp. 1–6. Springer (2022). https://doi.org/10.1007/978-3-031-17715-6_1, https://doi.org/10.1007/978-3-031-17715-6_1
9. 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.) CAV. LNCS, vol. 8559, pp. 334–342. Springer (2014). https://doi.org/10.1007/978-3-319-08867-9_22, https://doi.org/10.1007/978-3-319-08867-9_22
10. Chattopadhyay, A., Mamouras, K.: A Verified Online Monitor for Metric Temporal Logic with Quantitative Semantics. In: Runtime Verification: 20th International Conference, RV 2020, Los Angeles, CA, USA, October 6–9, 2020, Proceedings. p. 383–403. Springer-Verlag, Berlin, Heidelberg (2020). https://doi.org/10.1007/978-3-030-60508-7_21, https://doi.org/10.1007/978-3-030-60508-7_21
11. Chattopadhyay, A., Mamouras, K.: A verified online monitor for metric temporal logic with quantitative semantics. In: Deshmukh, J., Ničković, D. (eds.) Runtime Verification. pp. 383–403. Springer International Publishing, Cham (2020)
12. Conrad, E., Titolo, L., Giannakopoulou, D., Pressburger, T., Dutle, A.: A compositional proof framework for FRETish requirements. In: Popescu, A., Zdancewic, S. (eds.) CPP '22: 11th ACM SIGPLAN International Conference on Certified Programs and Proofs, Philadelphia, PA, USA, January 17 - 18, 2022. pp. 68–81. ACM (2022). https://doi.org/10.1145/3497775.3503685, https://doi.org/10.1145/3497775.3503685

13. Coquand, T., Siles, V.: A Decision Procedure for Regular Expression Equivalence in Type Theory. In: Jouannaud, J., Shao, Z. (eds.) CPP. LNCS, vol. 7086, pp. 119–134. Springer (2011). https://doi.org/10.1007/978-3-642-25379-9_11, https://doi.org/10.1007/978-3-642-25379-9_11
14. Coupet-Grimal, S.: An axiomatization of linear temporal logic in the calculus of inductive constructions. J. Log. Comput. **13**(6), 801–813 (2003). https://doi.org/10.1093/LOGCOM/13.6.801, https://doi.org/10.1093/logcom/13.6.801
15. 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). https://doi.org/https://doi.org/10.2514/6.2021-0566
16. 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)
17. Elwing, J., Gamboa-Guzman, L., 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. LNCS, vol. 14300, pp. 279–301. Springer (2023). https://doi.org/10.1007/978-3-031-47705-8_15, https://doi.org/10.1007/978-3-031-47705-8_15
18. Esparza, J., Lammich, P., Neumann, R., Nipkow, T., Schimpf, A., Smaus, J.G.: A fully verified executable LTL model checker. Archive of Formal Proofs (May 2014), https://isa-afp.org/entries/CAVA_LTL_Modelchecker.html, Formal proof development
19. 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). No. ARC-E-DAA-TN77785 (2020)
20. Haftmann, F.: Code generation from specifications in higher-order logic. Ph.D. thesis, Technical University Munich (2009), http://mediatum2.ub.tum.de/node?id=886023
21. Hariharan, G., Jones, P.H., Rozier, K.Y., Wongpiromsarn, T.: Maximum satisfiability of Mission-time Linear Temporal Logic. In: Petrucci, L., Sproston, J. (eds.) FORMATS. LNCS, vol. 14138, pp. 86–104. Springer (2023). https://doi.org/10.1007/978-3-031-42626-1_6, https://doi.org/10.1007/978-3-031-42626-1_6
22. Hupel, L., Nipkow, T.: A Verified Compiler from Isabelle/HOL to CakeML. In: Ahmed, A. (ed.) ESOP. LNCS, vol. 10801, pp. 999–1026. Springer (2018). https://doi.org/10.1007/978-3-319-89884-1_35, https://doi.org/10.1007/978-3-319-89884-1_35
23. Johannsen, C., Jones, P., Kempa, B., Rozier, K.Y., Zhang, P.: R2U2 Version 3.0: Re-Imagining a Toolchain for Specification, Resource Estimation, and Optimized Observer Generation for Runtime Verification in Hardware and Software. In: Enea, C., Lal, A. (eds.) Computer Aided Verification. pp. 483–497. Springer Nature Switzerland, Cham (2023)
24. Kempa, B., Zhang, P., Jones, P.H., Zambreno, J., Rozier, K.Y.: Embedding Online Runtime Verification for Fault Disambiguation on Robonaut2. In: FORMATS. pp. 196–214. LNCS, Springer, Vienna, Austria (September 2020), http://research.temporallogic.org/papers/KZJZR20.pdf
25. Kessler, F.B.: nuXmv 1.1.0 (2016-05-10) Release Notes. https://es-static.fbk.eu/tools/nuxmv/downloads/NEWS.txt (2016)

26. 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
27. Kosaian, K., Wang, Z., Sloan, E., Rozier, K.: Formalizing MLTL formula progression in Isabelle/HOL (2024), https://arxiv.org/abs/2410.03465
28. Krauss, A.: Automating Recursive Definitions and Termination Proofs in Higher-Order Logic. Ph.D. thesis, Technische Universität München (2009)
29. Krauss, A., Nipkow, T.: Proof Pearl: Regular Expression Equivalence and Relation Algebra. J. Autom. Reason. **49**(1), 95–106 (2012). https://doi.org/10.1007/S10817-011-9223-4, https://doi.org/10.1007/s10817-011-9223-4
30. 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 2019). LNCS, Springer, New York, NY, USA (July 2019)
31. Luppen, Z., Jacks, M., Baughman, N., Hertz, B., Cutler, J., Lee, D.Y., Rozier, K.Y.: Elucidation and Analysis of Specification Patterns in Aerospace System Telemetry. In: Proceedings of the 14th NASA Formal Methods Symposium (NFM 2022). Lecture Notes in Computer Science (LNCS), vol. 13260. Springer, Cham, Caltech, California, USA (May 2022). https://doi.org/10.1007/978-3-031-06773-0_28
32. Maler, O., Nickovic, D.: Monitoring temporal properties of continuous signals. In: Formal Techniques, Modelling and Analysis of Timed and Fault-Tolerant Systems, pp. 152–166. Springer (2004)
33. Marlow, S., Jones, S.L.P.: The Glasgow Haskell Compiler (2012), https://api.semanticscholar.org/CorpusID:35370
34. NASA Technology Transfer Program: FRET : Formal Requirements Elicitation Tool (ARC-18066-1). Online: https://software.nasa.gov/software/ARC-18066-1 (2024)
35. Perez, I.: Runtime verification with ogma. In: Invited Talk to University of California (2023)
36. Perez, I., Goodloe, A.: OGMA. https://github.com/nasa/ogma (2021)
37. Perez, I., Mavridou, A., Pressburger, T., Goodloe, A., Giannakopoulou, D.: Automated translation of natural language requirements to runtime monitors. In: Fisman, D., Rosu, G. (eds.) Tools and Algorithms for the Construction and Analysis of Systems. pp. 387–395. Springer International Publishing, Cham (2022)
38. Pnueli, A., Arons, T.: TLPVS: A PVS-based LTL verification system. In: Dershowitz, N. (ed.) Verification: Theory and Practice, Essays Dedicated to Zohar Manna on the Occasion of His 64th Birthday. LNCS, vol. 2772, pp. 598–625. Springer (2003). https://doi.org/10.1007/978-3-540-39910-0_26, https://doi.org/10.1007/978-3-540-39910-0_26
39. Raszyk, M., Basin, D., Traytel, D.: Multi-head monitoring of metric dynamic logic. In: International Symposium on Automated Technology for Verification and Analysis. pp. 233–250. Springer (2020)
40. 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)
41. Roohi, N., Viswanathan, M.: Revisiting MITL to fix decision procedures. In: Dillig, I., Palsberg, J. (eds.) VMCAI. LNCS, vol. 10747, pp. 474–494. Springer (2018). https://doi.org/10.1007/978-3-319-73721-8_22, https://doi.org/10.1007/978-3-319-73721-8_22

42. 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
43. Rozier, K.Y., Schumann, J.: R2U2: Tool Overview. In: Proceedings of International Workshop on Competitions, Usability, Benchmarks, Evaluation, and Standardisation for Runtime Verification Tools (RV-CUBES). vol. 3, pp. 138–156. Kalpa Publications, Seattle, WA, USA (September 2017), https://easychair.org/publications/paper/Vncw
44. Schimpf, A., Lammich, P.: Converting linear-time temporal logic to generalized Büchi automata. Archive of Formal Proofs (May 2014), https://isa-afp.org/entries/LTL_to_GBA.html, Formal proof development
45. Seidl, B., Sickert, S.: A compositional and unified translation of LTL into ω-automata. Archive of Formal Proofs (April 2019), https://isa-afp.org/entries/LTL_Master_Theorem.html, Formal proof development
46. Sickert, S.: Converting linear temporal logic to deterministic (generalized) Rabin automata. Archive of Formal Proofs (September 2015), https://isa-afp.org/entries/LTL_to_DRA.html, Formal proof development
47. Sickert, S.: Linear temporal logic. Archive of Formal Proofs (March 2016), https://isa-afp.org/entries/LTL.html, Formal proof development
48. Sickert, S.: An efficient normalisation procedure for linear temporal logic: Isabelle/HOL formalisation. Archive of Formal Proofs (May 2020), https://isa-afp.org/entries/LTL_Normal_Form.html, Formal proof development
49. Tilscher, S., Wimmer, S.: LL(1) parser generator. Archive of Formal Proofs (May 2024), https://isa-afp.org/entries/LL1_Parser.html, formal proof development
50. Titolo, L., Conrad, E., Giannakopoulou, D., Pressburger, T., Dutle, A.: FRET Proof Framework. https://lauratitolo.github.io/project/fret-proof-framework/ (2022)
51. Wang, Z., Gamboa-Guzman, L.P., Rozier, K.Y.: WEST: Interactive Validation of Mission-time Linear Temporal Logic (MLTL) (2024), https://temporallogic.org/research/WEST/
52. 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
53. Wang, Z., Kosaian, K., Rozier, K.Y.: Formally verifying a transformation from MLTL formulas to regular expressions (2025), https://arxiv.org/abs/2501.17444
54. Wu, C., Zhang, X., Urban, C.: A formalisation of the Myhill-Nerode theorem based on regular expressions (proof pearl). In: van Eekelen, M., Geuvers, H., Schmaltz, J., Wiedijk, F. (eds.) Interactive Theorem Proving. pp. 341–356. Springer Berlin Heidelberg, Berlin, Heidelberg (2011)
55. Zhang, P., Aurandt, A.A., Dureja, R., Jones, P.H., Rozier, K.Y.: Model predictive runtime verification for cyber-physical systems with real-time deadlines. In: Petrucci, L., Sproston, J. (eds.) Formal Modeling and Analysis of Timed Systems - 21st International Conference, FORMATS 2023, Antwerp, Belgium, September 19-21, 2023, Proceedings. Lecture Notes in Computer Science, vol. 14138, pp. 158–180. Springer (2023). https://doi.org/10.1007/978-3-031-42626-1_10, https://doi.org/10.1007/978-3-031-42626-1_10

56. Zhuchko, E., Veanes, M., Ebner, G.: Lean formalization of extended regular expression matching with lookarounds. In: Proceedings of the 13th ACM SIGPLAN International Conference on Certified Programs and Proofs. p. 118–131. CPP 2024, Association for Computing Machinery, New York, NY, USA (2024). https://doi.org/10.1145/3636501.3636959, https://doi.org/10.1145/3636501.3636959