Mission-time LTL (MLTL) Formula Validation Via Regular Expressions⋆

Jenna Elwing[1], Laura Gamboa-Guzman[2], Jeremy Sorkin[3], Chiara Travesset[4], Zili Wang[2], and Kristin Y. Rozier[2]

[1] University of Michigan, Ann Arbor
[2] Iowa State University
[3] University of Amsterdam
[4] University of Wisconsin, Madison

Abstract. Mission-time Linear Temporal Logic (MLTL) represents the most practical fragment of Metric Temporal Logic; MLTL resembles the popular logic LTL with finite closed-interval integer bounds on the temporal operators. Increasingly, many tools reason over MLTL specifications, yet these tools are only useful when system designers can validate the input specifications. We design an automated characterization of the structure of the computations that satisfy a given MLTL formula using regular expressions. We prove soundness and completeness of our structure. We also give an algorithm for automated MLTL formula validation and analyze its complexity both theoretically and experimentally. Additionally, we generate a test suite using control flow diagrams to robustly test our implementation and release an open-source tool. The result of our contributions are improvements to existing algorithms for MLTL analysis applicable to many other tools and the WEST tool for automated, efficient MLTL formula validation.

Keywords: Mission-time Linear Temporal Logic (MLTL) · MLTL Validation · Temporal Logic Validation.

1 Introduction

System specifications, such as aerospace operational concepts, often utilize timelines to express critical requirements. We can cite examples of this from NASA's Automated Airspace Concept [11], the U. S. Navy's Aircraft Carrier Deck Scheduler [33], the JAXA-NASA Global Precipitation Measurement (GPM) Observatory [10], and many others. Formal methods provide continuously advancing tools and techniques to rigorously analyze timelines expressed in the form of temporal logic requirements, from early design-time model checking and theorem proving to on-board runtime verification. The U. S. Federal Aviation Administration (FAA) even advocates the use of formal methods for flight certification of these critical systems [28,29,27]. Yet, a significant hurdle to the use of formal methods remains: how to convincingly demonstrate to the humans in the loop, from system designers to certifiers, that the analyzed formulas truly represent the system requirements [31]. We creatively address this validation question using regular expressions.

NASA, for example, has developed several tools that operate over temporal logic requirements, such as FRET[12], R2U2[32], and a PVS library [7] for the logic MLTL

⋆ Work supported in part by NSF grant DMS-1950583 and NSF CAREER Award CNS-1552934.

(Mission-time Linear Temporal Logic) [30,19]. MLTL was the specification logic for NASA's Robonaut2 verification project [16] and is currently the specification logic for both design-time and runtime verification of the NASA Lunar Gateway Vehicle System Manager [8]. Other recent verification efforts involving MLTL include a JAXA autonomous satellite [24], a UAS Traffic Management (UTM) system involving Collins and Mosaic Aerospace [13], a sounding rocket [14], and multiple small satellites [21,20,2]. However, all of these successful verification efforts were carried out by groups specializing in formal methods research. To enable broader application of formal verification, and adoption across larger projects, we critically need better validation, e.g., so that analysis over MLTL-specified requirements can transparently contribute to flight certification.

Many specifications from case studies, in logics such as MTL [1] and STL [22], fall within the MLTL fragments of these logics. Variations on Metric Temporal Logic (MTL) such as MLTL have grown increasingly popular, in part due to their comparatively tractable complexity-to-expressibility trade-offs [25]. The model checker nuXmv encodes a popular subset of MLTL for use in symbolic model checking [17].

There exists a SAT solver for MLTL, MLTLSAT [19], but there are currently no tools for MLTL formula validation. This paper introduces the WEST tool [GitHub][5], which produces a description of the set of all finite time lines (of a fixed length) that satisfy a given MLTL formula, similar to a truth table for propositional formulas. MLTL validation can be done by verifying that the output of the WEST program indeed matches the behaviour of the specification in question.

We show that our contributions not only fill a critical gap in temporal logic validation but also directly connect to parallel developments to enable better temporal logic formula analysis, benchmark generation, proof generation (e.g., in ACL2), and synthesis of verified C++ code from temporal logic behavior descriptions.

We structure the paper as follows. Section 2 builds on the semantics of MLTL to define a computation and its bit string representation. Section 3, recursively defines regular expressions encapsulating the satisfying computations of MLTL formulas. We provide a calculation for the minimum computation length required to describe all the satisfying computations of an MLTL formula that slightly improves upon existing calculations in the literature. Finally, we show an application of the regular expressions by using them to prove an MLTL theorem. We introduce the WEST tool that implements automated validation in Section 4 and calculate its space and time complexity, both theoretically and experimentally. Section 5 proves the correctness of WEST and provides a test suite to show correctness of implementation with high confidence. Intelligent fuzzing techniques contribute to test suite construction from a state diagram representing the control flow of WEST. We also verify the correctness of outputs of the WEST program against a naïve brute force implementation. Section 6 provides a combinatorial theorem for simplifying certain outputs of the WEST program to the trivial computation. Section 7 demonstrates a specific use case of the WEST tool and explores the currently supported features. Section 8 discusses impacts and future work.

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

2 Preliminaries: Mission-time LTL and Bit String Computations

Mission-time Linear Temporal Logic (MLTL) [19] is a finite variation of LTL over bounded, closed, discrete intervals of the form $[a, b]$ where $a, b \in \mathbb{N}$ and $0 \leqslant a \leqslant b$. The syntax of MLTL formulas, φ and ψ, over a set of atomic propositions $\mathcal{AP}$, where $p \in \mathcal{AP}$ is a propositional variable, is given by the following BNF grammar:

$$\varphi, \psi := \top \mid \bot \mid p \mid \neg\varphi \mid \varphi \wedge \psi \mid \varphi \vee \psi \mid \mathcal{F}_{[a,b]}\varphi \mid \mathcal{G}_{[a,b]}\varphi \mid \varphi\mathcal{U}_{[a,b]}\psi \mid \varphi\mathcal{R}_{[a,b]}\psi.^6$$

The symbols $\mathcal{F}, \mathcal{G}, \mathcal{U}, \mathcal{R}$ respectively denote the temporal operators Finally, Globally (or Always), Until, and Release. MLTL formulas can be interpreted using both finite and infinite "time lines" that are called *computations*, which represent a discrete sequence of time instances and the truth values for the propositional variables on each one of these. For the purpose of this paper, we are only going to deal with *finite* computations, that is, computations that represent only finitely many time steps.

Definition 1 **(*Finite computations*).** *A computation π of length m is a sequence $\{\pi[i]\}_{i=0}^{m-1}$ of sets of propositional variables, $\pi[i] \subseteq \mathcal{AP}$, where the i^{th} set holds the propositional variables that are satisfied at the i^{th} time step. That is, a propositional variable p is true at time step i if and only if $p \in \pi[i]$. We denote the suffix of π starting at i (including i) by π_i. Note that $\pi_0 = \pi$.*

We now provide the semantics for MLTL. A computation π satisfies a given MLTL formula α, written $\pi \models \alpha$, in the following cases[7]:

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

$\pi \models \alpha \wedge \beta$ iff $\pi \models \alpha$ and $\pi \models \beta$ $\qquad$ $\pi \models \alpha \vee \beta$ iff $\pi \models \alpha$ or $\pi \models \beta$

$\pi \models \mathcal{F}_{[a,b]}\alpha$ iff $|\pi| > a$ and $\exists i \in [a, b]$ such that $\pi_i \models \alpha$

$\pi \models \mathcal{G}_{[a,b]}\alpha$ iff $|\pi| \leqslant a$ or $\forall i \in [a, b]$ $\pi_i \models \alpha$

$\pi \models \alpha\,\mathcal{U}_{[a,b]}\beta$ iff $|\pi| > a$ and $\exists i \in [a, b]$ such that $\pi_i \models \beta$ and $\forall j \in [a, i-1]$ $\pi_j \models \alpha$

$\pi \models \alpha\,\mathcal{R}_{[a,b]}\beta$ iff $|\pi| \leqslant a$ or $\forall i \in [a, b]$ $\pi_i \models \beta$ or $\exists j \in [a, b-1]$ such that $\pi_j \models \alpha$ and $\forall a \leqslant k \leqslant j$ $\pi_k \models \beta$

Definition 2 **(*Bit String Representation of a Computation*).** *Let $p_0, p_1, ..., p_{n-1}$ be propositional variables for a fixed $n \in \mathbb{N}$. We represent a (finite) computation π of length $m \in \mathbb{N}$ using a bit string representation as follows:*

- *For each time step $j \in [0, 1, \dots, m-1]$, we have a bit string of length n where the k^{th} bit represents the truth assignment of the proposition p_{k-1}.*
- *We separate each time step by a comma and order the time steps chronologically.*

Example 1. Suppose $n = 2$. The bit string representation $\pi = 10, 01$ means that p_0 is true in the first time step and false in the second one, and that p_1 is false in the first time step and true in the second one.

[6] For simplicity, we do not include parentheses in the grammar, but the WEST program requires parentheses (see Section 4). We encode Release directly rather than as the dual of Until. See Appendix IV for a straightforward proof of equivalence using the semantics.

[7] We do not include the Next operator, often denoted by χ, as it can be defined as $\mathcal{G}_{[1,1]}$.

3 MLTL Regular Expressions

We modify the standard definition of Regular Expressions (regex) to introduce notation that describes the satisfying computations of an MLTL formula. We begin by quoting the parts of the standard definition of a regex from [34] that we use to describe our computations:

Definition 3 (***Regular Expression***). *Let Σ denote an alphabet, and let $a \in \Sigma$. We say that R is a* regular expression *if one of the following holds:*

- $R = a$
- $R = \epsilon$*, where ϵ is the language containing the empty string*
- $R = \varnothing$
- $R = (R_1 \vee R_2)$*, where R_1, R_2 are regexes and $\vee$ denotes* alternation*; the set union of all the strings described by R_1 and R_2*
- $R = R_1 R_2$*, which denotes* concatenation*, i.e., the set of strings obtained by concatenating any string generated by R_1 with any string generated by R_2, in that order*

Now we introduce the additions to the definition of a regex that we utilize to describe the computations. These additions allow us to write regexes of a known, finite, fixed length, which is required when describing the computations in MLTL.

Definition 4 (***Temporal Regular Expression***). *Let R and T denote regular expressions, and let S be an abbreviation for $(0 \ \vee \ 1)$.*
Let fixed $n \in \mathbb{N}$ denote the number of propositional variables in an MLTL formula. We use the following operations to describe the form of satisfying computations of the formula in the bit string representation:

- *Pad(R, T) determines which regular expression is longer and concatenates $(, S^n)$ repeatedly to the end of the shorter regular expression until the two regular expressions are the same length. Note that in the bit string representation, $(, S^n)$ denotes a time step in which the truth value of all n propositional variables does not matter.*
- *$R \wedge T$ is the intersection of the sets of strings described by R and T. To perform this operation, we first use Pad(R, T), and then take the set intersection of the sets of strings described by the two regular expressions.*
- *R^i denotes regular expression consisting of R repeated i times for $i \geqslant 0$. $R^0 = \epsilon$.*

Note that our regular expressions do not use the Kleene star. This is because our computations are of a fixed, finite length.

Example 2. Let $n = 2$, and let $R = S1$ and $T = 1S, 1S \vee S1, S1$. To compute $R \wedge T$ and $R \vee T$, we use Pad(R,T). Since T is the longer regex by one time step, we extend R by one time-step. Thus $T = 1S, 1S \vee S1, S1$ and $R = S1, SS$. Now we can perform set intersection and alternation on the two regular expressions of equal length:

- $R \wedge T = 11, 1S \vee S1, S1$.
- $R \vee T = S1, SS \vee 1S, 1S \vee S1, S1 = S1, SS \vee 1S, 1S$.

To describe the bit string representations of the satisfying computations for a given MLTL formula, we use the alphabet $\Sigma = \{$"0" , "1", ","$\}$ and define S as an abbreviation for $(0 \ \vee \ 1)$. Let $p_0, p_1, ..., p_{n-1}$ be propositional variables and φ, ψ well-formed

MLTL formulas in negation normal form (NNF). Note that any MLTL formula can easily be converted into NNF. We define the regular expression of satisfying computations for an MLTL formula as follows:

$$\text{reg}(\top) = S^n \qquad \text{reg}(\bot) = \varnothing$$

$$\text{reg}(p_k) = S^k 1 S^{n-k-1} \qquad \text{reg}(\neg p_k) = S^k 0 S^{n-k-1}$$

$$\text{reg}(\varphi \vee \psi) = \text{reg}(\varphi) \vee \text{reg}(\psi) \qquad \text{reg}(\varphi \wedge \psi) = \text{reg}(\varphi) \wedge \text{reg}(\psi)$$

$$\text{reg}(\mathcal{G}_{[a,b]}\varphi) = \bigwedge_{i=a}^{b} (S^n,)^i \text{reg}(\varphi) \qquad \text{reg}(\mathcal{F}_{[a,b]}\varphi) = \bigvee_{i=a}^{b} (S^n,)^i \text{reg}(\varphi)$$

$$\text{reg}(\varphi\, \mathcal{U}_{[a,b]}\psi) = \bigvee_{i=a}^{b} \text{reg}\left(\mathcal{G}_{[a,i-1]}\varphi \wedge \mathcal{G}_{[i,i]}\psi\right)$$

$$\text{reg}(\varphi \mathcal{R}_{[a,b]}\psi) = \text{reg}\left(\mathcal{G}_{[a,b]}\psi\right) \vee \bigvee_{i=a}^{b-1} \text{reg}\left(\mathcal{G}_{[a,i]}\psi \wedge \mathcal{G}_{[i,i]}\varphi\right)$$

Definition 5 **(*Computation Length*).** *We define the computation length* cplen(φ) *of an MLTL formula* φ *recursively:*

$$\text{cplen}(p_k) = \text{cplen}(\neg p_k) = 1,$$
$$\text{cplen}(\varphi \wedge \psi) = \text{cplen}(\varphi \vee \psi) = \max(\text{cplen}(\varphi), \text{cplen}(\psi)),$$
$$\text{cplen}(\mathcal{G}_{[a,b]}\varphi) = \text{cplen}(\mathcal{F}_{[a,b]}\varphi) = b + \text{cplen}(\varphi),$$
$$\text{cplen}(\varphi\mathcal{U}_{[a,b]}\psi) = \text{cplen}(\varphi\mathcal{R}_{[a,b]}\psi) = b + \max(\text{cplen}(\varphi) - 1, \text{cplen}(\psi)).$$

cplen(φ) is the minimum computation length required to ensure that none of the intervals in φ are out of bounds. A computation that is of length cplen(φ) or greater will reach the end of every interval in φ. Our minimum computation length for Until and Release are slight optimizations of what was previously considered the minimum computation length in the literature. In [16],

$$\text{cplen}(\varphi\mathcal{U}_{[a,b]}\psi) = \text{cplen}(\varphi\mathcal{R}_{[a,b]}\psi) = b + \max(\text{cplen}(\varphi), \text{cplen}(\psi))$$

whereas Theorem 1 proves our minimum computation length for Until and Release,

$$\text{cplen}(\varphi\mathcal{U}_{[a,b]}\psi) = \text{cplen}(\varphi\mathcal{R}_{[a,b]}\psi) = b + \max(\text{cplen}(\varphi) - \mathbf{1}, \text{cplen}(\psi)).$$

Theorem 1 **(*Minimum Computation Length of Until and Release*).** *Let* $0 \leqslant a \leqslant b \in \mathbb{N}$ *and let* φ, ψ *be well-formed MLTL formulas in NNF. The minimum computation length of* Until *and* Release *is given by* $\text{cplen}(\varphi\mathcal{U}_{[a,b]}\psi) = \text{cplen}(\varphi\mathcal{R}_{[a,b]}\psi) = b + \max(\text{cplen}(\varphi) - 1, \text{cplen}(\psi))$.

Proof. The formulas for the minimum computation length follow directly from the regular expressions for the satisfying computations for Until and Release and the minimum computation lengths for Finally, Globally, AND, and OR. See Appendix I for details.

We can reduce the minimum computation length of Until because at the final time step for $\varphi\mathcal{U}_{[a,b]}\psi$, ψ must be assigned true if it hasn't been at a prior time step due to Until being a strong operator, and the truth value of φ at this final time step doesn't matter. Likewise, in the case for $\varphi\mathcal{R}_{[a,b]}\psi$, if ψ is true from the first to the last time step, the

truth value of φ at the final time step doesn't matter — the computation satisfies the formula regardless.
We write $\mathscr{L}(\text{reg}(\varphi))$ to denote the language of reg(φ). That is, the set of computations represented by the regular expression reg(φ).

Theorem 2 **(*Soundness and Completeness*).** *For any well formed MLTL formula φ in negation normal form, a computation π with $|\pi| = \text{cplen}(\varphi)$ satisfies φ if and only if $\pi \in \mathscr{L}(\text{reg}(\varphi))$.*

Proof. Straightforward proof by induction on the length of a formula. See Appendix II for details.

As an application of the regular expressions in the above theorem, we prove a previously known MLTL rewriting theorem. This demonstrates the utility of our regular expressions for theoretical analysis.

Theorem 3 **(*Nested Until and Release Rewriting Theorem*).** *Any MLTL formula using the Until or Release operator can be rewritten with right-nested subformulas. Let $a, b, c \in \mathbb{Z}_{\geqslant 0}$, $a \leqslant b$, and φ, ψ be well-formed MLTL formulas in NNF. Then $\varphi\, \mathcal{U}_{[a,b+c]}\psi \equiv \varphi\, \mathcal{U}_{[a,b]}(\varphi\, \mathcal{U}_{[0,c]}\psi)$ and $\varphi\, \mathcal{R}_{[a,b+c]}\psi \equiv \varphi\, \mathcal{R}_{[a,b]}(\varphi\, \mathcal{R}_{[0,c]}\psi)$.*

Proof. Follows directly from the regular expressions for MLTL. See III for details.

4 WEST Algorithm and Analysis

Algorithm 1 (Fig. 1) recursively computes all satisfying temporal regular expressions to an input formula φ. We use sets to represent alternation of regular expressions; for n regular expressions $t_0, ..., t_{n-1}$, we'll write $\{t_0, ..., t_{n-1}\} = \bigcup_{i=0}^{n-1} t_i$. Additionally, we provide details for performing set intersection of temporal regular expressions, and the algorithms for the temporal operators $\mathcal{G}$ and $\mathcal{U}$. Note how `reg_U` parallels the regular expression defined for $\mathcal{U}$, and the algorithms (see appendix VI) for the other three temporal operators follow an identical structure.
For regular expressions w_0 and w_1, a useful reduction is that $\{w_0 1 w_1, w_0 0 w_1\} = \{w_0 s w_1\}$. Each time we perform set intersection, we greedily apply this reduction to all appropriate pairs of regular expressions in the set. This prevents repeated set intersection operations from blowing up exponentially most of the time, and drastically improves running time. We call this simple algorithm `simplify` and use it extensively throughout the WEST code.

4.1 Proof of Correctness of WEST

Theorem 4 **(*Theoretical Correctness of WEST*).** *Given a well-formed MLTL formula, The WEST Algorithm outputs the regular expressions of the satisfying computations as described in Section 3.*

Correctness of the WEST algorithm is dependent on the correctness of subroutines `reg_prop_cons`, `reg_prop_var`, `join`, `set_intersect`, `reg_F`, `reg_G`, `reg_U`, and `reg_R`. The routines `reg_prop_cons` and `reg_prop_var` take as input an MLTL formula of the appropriate form and return the regular expression defined in Section 3.

Algorithm 1 WEST Algorithm

Inputs: φ - MLTL formula in NNF
φ_1 and φ_2 below are subformulas of φ
n - number of propositional variables
Output: set of REGEX satisfying φ

procedure REG(string φ, int n)
  if φ is $\top$ or $\bot$ **then**
    return reg_prop_const(φ, n)
  if φ is p_k or $\neg p_k$ **then**
    return reg_prop_var(φ, n)
  if $\varphi = \varphi_1 \wedge \varphi_2$ **then**
    return set_intersect(reg(φ_1), reg(φ_2), n)
  if $\varphi = \varphi_1 \vee \varphi_2$ **then**
    return join(reg(φ_1), reg(φ_2), n)
  if $\varphi = \mathcal{F}_{[a,b]}\varphi_1$ **then**
    return reg_F(reg(φ_1), a, b, n)
  if $\varphi = \mathcal{G}_{[a,b]}\varphi_1$ **then**
    return reg_G(reg(φ_1), a, b, n)
  if $\varphi = \varphi_1 \mathcal{U}_{[a,b]} \varphi_2$ **then**
    return reg_U(reg(φ_1), reg(φ_2), a, b, n)
  if $\varphi = \varphi_1 \mathcal{R}_{[a,b]} \varphi_2$ **then**
    return reg_R(reg(φ_1), reg(φ_2), a, b, n)

Algorithm 2 set_intersect

Inputs: R, T - two sets of REGEX
n - number of propositional variables
Output: set of REGEX equal to $R \wedge T$

procedure SET_INTERSECT(R, T, n)
  Pad(R, T, n), ret $\leftarrow \{\}$
  for $(r, t) \in R \times T$ **do**
    add bit_wise_and(r, t) to ret
  return simplify(ret)

Algorithm 3 reg_G

Inputs: r_φ - set of REGEX for MLTL formula φ (after calling reg)
a, b - interval bounds
n - number of propositional variables
Output: set of REGEX for $G_{[a,b]}\varphi$

procedure REG_G(set r_φ, int a, int b, int n)
  pre $\leftarrow$ ((‘S’)^n + ‘,’)^a
  comp $\leftarrow r_\varphi$
  if $a > b$ **then** return $\{S^n\}$
  for $(1 \leqslant i \leqslant b - a)$ **do**
    temp$_\varphi \leftarrow$ ((‘S’)^n + ‘,’)^i + r_φ
    comp $\leftarrow$ set_intersect(comp, temp$_\varphi$, n)
  return pre + comp

Algorithm 4 reg_U

Inputs: r_φ, r_ψ - sets of REGEX for MLTL formulas φ and ψ (after calling reg)
a, b - integers representing interval bound
n - number of propositional variables
Output: set of REGEX for $\varphi \mathcal{U}_{[a,b]} \psi$

procedure REG_U(r_φ, r_ψ, a, b, n)
  comp $\leftarrow$ ((‘S’)^n + ‘,’)^a+ r_ψ
  if $a > b$ **then** return $\{\}$
  for $(a \leqslant i \leqslant b - 1)$ **do**
    G1 $\leftarrow$ reg_G(r_φ, a, i, n)
    G2 $\leftarrow$ reg_G(r_ψ, $i + 1$, $i + 1$, n)
    temp_comp $\leftarrow$ set_intersect(G1, G2, n)
    comp $\leftarrow$ join(comp, temp_comp)
  return comp

Fig. 1: Pseudocode for WEST, set_intersect, reg_G, and reg_U

`Join` concatenates two sets of regular expressions R and T, which is equivalent to $\mathscr{L}(R) \cup \mathscr{L}(T)$. `set_intersect` takes as input two sets of regular expressions $R = \{r_0, ..., r_{a-1}\}$ and $T = \{t_0, ..., t_{b-1}\}$, such that each r_i and t_j are regular expressions over $\Sigma = \{$“0” , “1”, “S”, “,”$\}$. Assume that all strings of regular expressions are right padded to equal length. We show that $\mathscr{L}(\texttt{set\_intersect}(R, T)) = \mathscr{L}(R) \cap \mathscr{L}(T)$:

$$\mathscr{L}(R) \cap \mathscr{L}(T) = \left(\bigcup_{i=0}^{a-1} \mathscr{L}(r_i)\right) \cap \left(\bigcup_{j=0}^{b-1} \mathscr{L}(t_j)\right) = \bigcup_{i=0}^{a-1} \bigcup_{j=0}^{b-1} \left(\mathscr{L}(r_i) \cap \mathscr{L}(t_j)\right).$$

The loop in line 3 of `set_intersect` computes the union of $\texttt{bit\_wise\_and}(r_i, t_j)$ over all such pairs, and so it suffices to show $\mathscr{L}(\texttt{bit\_wise\_and}(r_i, t_j)) = \mathscr{L}(r_i) \cap \mathscr{L}(t_j)$. Given a computation π, $\pi \in \mathscr{L}(r_i) \cap \mathscr{L}(t_j)$ if and only if π matches every character of both r_i and t_j. $\texttt{Bit\_wise\_and}(r_i, t_j)$ compares r_i and t_j character by character and computes their intersection, which is defined naturally: $0 \cap 1 = \varnothing,\ 0 \cap S = 0,\ 1 \cap S = 1,\ 0 \cap 0 = 0,\ 1 \cap 1 = 1,$ and $S \cap S = S$. Note that this operation is commutative. This exhaustively captures all the cases for which π must match corresponding characters from r_i and t_j. Thus $\mathscr{L}(\texttt{bit\_wise\_and}(r_i, t_j)) = \mathscr{L}(r_i) \cap \mathscr{L}(t_j)$ and the claim holds.

The correctness for `reg_F`, `reg_G`, `reg_U`, `reg_R`, and `reg` will be proven by induction on depth of recursion to `reg`. The depth of recursion is exactly the depth of the parse tree of the input formula. For the base case (depth 0), `reg_prop_var` and `reg_prop_cons` are called to handle input formulas that consist of a propositional variable, the negation of a propositional variable, or a propositional constant. Then assume `reg` is correct on all formulas of depth at most d, for some integer $d \geqslant 0$. Let γ be an MLTL formula in negation normal form of depth $d + 1$. γ must be of the form $\varphi \vee \psi$, $\varphi \wedge \psi$, $\mathcal{G}_{[a,b]}\varphi$, $\mathcal{F}_{[a,b]}\varphi$, $\varphi\mathcal{U}_{[a,b]}\psi$, or $\varphi\mathcal{R}_{[a,b]}\psi$, such that φ and ψ are of depth at most d, and a, b are nonnegative integers. Correctness of the first two cases have been proven. The proof for the four temporal cases are of similar structure, and it suffices to verify that the algorithms compute appropriate regular expressions correctly using `join` and `set_intersect`.

We give the explicit proof for the case $\gamma = \varphi\mathcal{U}_{[a,b]}\psi$ as an example. `reg_U` takes as input r_φ = reg(φ) and r_ψ = reg(ψ), which by the induction hypothesis are correctly and recursively computed. The regular expression for the Until operator may be rewritten as $\text{reg}(\varphi\,\mathcal{U}_{[a,b]}\psi) = \text{reg}\left(\mathcal{G}_{[a,a]}\psi\right) \vee \bigvee_{i=a}^{b-1} \text{reg}\left(\mathcal{G}_{[a,i]}\varphi \wedge \mathcal{G}_{[i+1,i+1]}\psi\right)$. In line 2 of `reg_U`, the variable `comp` is initialized to $(\text{“}S\text{”}^n + \text{“},\text{”})^a$ pre-concatenated to r_ψ, and is the regular expression for $\mathcal{G}_{[a,a]}\psi$. Next, the $\vee$ from $i = a + 1$ to $b - 1$ is computed by the for loop in line 4; lines 5 through 7 computes $\text{reg}\left(\mathcal{G}_{[a,i]}\varphi \wedge \mathcal{G}_{[i+1,i+1]}\psi\right)$. This shows correctness of `reg_U` (note that in a complete proof, correctness of `reg_G` needs to be shown first since lines 5 and 6 calls `reg_G`). Continuing in the same fashion to prove the other three cases, `reg` is correct on all depth $d + 1$ inputs, and thus `reg` is correct on all inputs by induction.

4.2 Theoretical Complexity

In order to reason about the complexity of our algorithms, we make a few reasonable assumptions about our inputs. Suppose that the lower and upper intervals of temporal operators are bounded by some constant $d \in \mathbb{N}$, and that the difference between any bound is less than some constant $\delta \in \mathbb{N}$. These are reasonable assumptions because MLTL is a finite temporal logic with a known mission end. We provide a summary of the complexity of each of the helper functions that contribute to the worst case behavior of final output.

For any function $f(\varphi)$ taking a string argument φ, we use $|\varphi|$ to denote the number of characters in φ and $S(f(\varphi))$ to denote the space complexity of f in terms of the number of characters in the output.

If φ is a propositional constant or variable, it's easy to see that $S(\texttt{reg}(\bot)) = 0$ since only the empty set is returned. By definition $\texttt{reg}(\top) = S^n$, so we have that

$S(\texttt{reg}(\top)) = n$. Similarly, $\texttt{reg}(p_k)$ and $\texttt{reg}(\neg p_k)$ both return strings of the same length and thus $S(\texttt{reg}(p_k)) = S(\texttt{reg}(\neg p_k)) = S(\texttt{reg}(\top)) = n$.

If φ is "$\varphi_1 \vee \varphi_2$" we return $\texttt{join}(\texttt{reg}(\varphi_1), \texttt{reg}(\varphi_2))$, which simply concatenates the contents of the two sets. Thus $S(\texttt{reg}(\varphi_1 \vee \varphi_2)) = S(\texttt{reg}(\varphi_1)) + S(\texttt{reg}(\varphi_2))$.

If φ is "$\varphi_1 \wedge \varphi_2$" $\texttt{set\_intersect}(\texttt{reg}(\varphi_1), \texttt{reg}(\varphi_2), n)$ returns a set of size $S(A) \cdot S(B)$ in the worst case when no simplification can be made. Thus, our space complexity is $S(\texttt{reg}(\varphi_1 \wedge \varphi_2)) = S(\texttt{reg}(\varphi_1)) \cdot S(\texttt{reg}(\varphi_2))$.

For the next cases, we use these two bounds and define the constants $\mathcal{C}_G$ and $\mathcal{C}_F$:

$$\prod_{i=a}^{b}(n+1)i = (n+1)^{b-a} \cdot \frac{b!}{(a-1)!} \leqslant (n+1)^{\delta} b! \leqslant (n+1)^{\delta} d! = \mathcal{C}_G$$

$$\sum_{i=a}^{b}(n+1)i \leqslant (n+1)b\delta \leqslant (n+1)d\delta = \mathcal{C}_F$$

If φ is "$\mathcal{G}_{[a,b]}\varphi_1$": recall that $\texttt{reg}(\mathcal{G}_{[a,b]}\varphi_1) = \bigwedge_{i=a}^{b}(S^n,)^i \texttt{reg}(\varphi_1)$. From the analysis of $\texttt{set\_intersect}$, worst case space complexity is multiplicative. Thus $S(\texttt{reg}(\varphi)) = \prod_{i=a}^{b}(n+1)i \cdot S(\texttt{reg}(\varphi_1)) < \mathcal{C}_G \cdot S(\texttt{reg}(\varphi_1))^{\delta}$. In this calculation, $(n+1)i$ counts the concatenation of the padded component of the computation.

If φ is "$\mathcal{F}_{[a,b]}\varphi_1$": Recall that $\texttt{reg}(\mathcal{F}_{[a,b]}\varphi_1) = \bigvee_{i=a}^{b}(S^n,)^i \texttt{reg}(\varphi_1)$.
From the analysis of $\texttt{join}$, worst case space complexity is additive and thus $S(\texttt{reg}(\varphi)) = \sum_{i=a}^{b}(n+1)i \cdot S(\texttt{reg}(\varphi_1)) < \mathcal{C}_F \cdot S(\texttt{reg}(\varphi_1))$.

If φ = "$\varphi_1 \mathcal{U}_{[a,b]} \varphi_2$": Recall that
$\texttt{reg}(\varphi_1 \, \mathcal{U}_{[a,b]}\varphi_2) = \bigvee_{i=a}^{b} \texttt{reg}\left(\mathcal{G}_{[a,i-1]}\varphi_1 \wedge \mathcal{G}_{[i,i]}\varphi_2\right)$. We can bound $S(\texttt{reg}(\mathcal{G}_{[i,i]}\varphi_2))$ by $(n+1) \cdot i \cdot S(\texttt{reg}(\varphi_2))$ because the operation is equivalent to simply prepending $(S^n,)^i$. Thus using our previous results for the $\mathcal{G}$, $\wedge$ and $\vee$ operators, we have that:

$$\begin{aligned}
S(\texttt{reg}(\varphi)) &\leqslant \sum_{i=a}^{b}[\mathcal{C}_G S(\texttt{reg}(\varphi_1))^{\delta} \cdot (n+1)iS(\texttt{reg}(\varphi_2))] \\
&\leqslant \delta[\mathcal{C}_G \cdot \delta(n+1)d \cdot S(\texttt{reg}(\varphi_1))^{\delta} S(\texttt{reg}(\varphi_2))] \\
&= \mathcal{C}_U \cdot S(\texttt{reg}(\varphi_1))^{\delta} S(\texttt{reg}(\varphi_2))
\end{aligned}$$

where $\mathcal{C}_U = \mathcal{C}_G \delta(n+1)d$.

If φ = "$\varphi_1 \mathcal{R}_{[a,b]} \varphi_2$":
$\texttt{reg}(\varphi_1 \mathcal{R}_{[a,b]}\varphi_2) = \texttt{reg}\left(\mathcal{G}_{[a,b]}\varphi_2\right) \vee \bigvee_{i=a}^{b-1} \texttt{reg}\left(\mathcal{G}_{[a,i]}\varphi_2 \wedge \mathcal{G}_{[i,i]}\varphi_1\right)$. A similar argument to the $\mathcal{U}$ case shows $S(\texttt{reg}(\varphi)) < \mathcal{C}_R \cdot S(\texttt{reg}(\varphi_1)) \cdot S(\texttt{reg}(\varphi_2))^{\delta}$, where $\mathcal{C}_R = \mathcal{C}_G \cdot (1 + \delta(n+1)d)$.

Theorem 5 **(*Space Complexity*).** *Given a well formed MLTL formula φ, $\texttt{reg}(\varphi)$ has worst-case space complexity that is $O(\mathcal{C}_R^{\delta^\ell} \cdot (\ell+1)^{\delta^{\ell+1}})$, where ℓ is the number of logical connectives ($\wedge, \vee, \mathcal{F}, \mathcal{G}, \mathcal{R}, \mathcal{U}$) in φ.*

Proof. To analyze worst-case complexity, it's clear from the analysis above that $\mathcal{U}$ and $\mathcal{R}$ give the worst complexity. In the previous analysis, we defined $\mathcal{C}_U = \mathcal{C}_G \delta(n+1)d$ and $\mathcal{C}_R = \mathcal{C}_G \cdot (1 + \delta(n+1)d)$. Observe that $\mathcal{C}_R > \mathcal{C}_U$, and so we'll analyze only repeated nesting of the operator $\mathcal{R}$.
However, notice that the structure of the parse tree is important. For example, formulas similar to $(p_3 \mathcal{R} p_1)\mathcal{R}(p_1 \mathcal{R} p_0)$ generate a balanced binary parse tree where the maximum depth of recursion is $O(\log \ell)$. However if the nesting is only from one side, such as formulas similar to $p_3\mathcal{R}(p_2\mathcal{R}(p1\mathcal{R}p_0))$, then the maximum depth of recursion is

$O(\ell)$. Thus we'll focus on the formula

$$\varphi = p_\ell \mathcal{R}_{[a_\ell,b_\ell]}(p_{\ell-1}\mathcal{R}_{[a_{\ell-1},b_{\ell-1}]}...\mathcal{R}_{[a_3,b_3]}(p_2\mathcal{R}_{[a_2,b_2]}(p_1\mathcal{R}_{[a_1,b_1]}p_0))...)$$

where there are ℓ logical connectives $\mathcal{R}$ and $n = \ell + 1$ propositional variables.
We'll derive the complexity of $S(\texttt{reg}(\varphi))$ by defining the recursive sequence $\{s_k\}_{k=1}^{\ell}$ such that $s_1 = S(\texttt{reg}(p_1\mathcal{R}_{[a_1,b_1]}p_0))$ and $s_{k+1} = \mathcal{C}_R S(\texttt{reg}(p_{k+1}))(s_k)^\delta$. The recurrence relation captures an extra nesting of the $\mathcal{R}$ operator, based on the complexity of $\mathcal{R}$ defined above. We calculate $S(p_m) = n = \ell + 1$ for all m such that $0 \leqslant m \leqslant \ell$, thus $s_1 = \mathcal{C}_R(\ell+1)^{\delta+1}$ and $s_{k+1} = \mathcal{C}_R(\ell+1)(s_k)^\delta$.
The explicit formula is given by $s_k = \mathcal{C}_R^{\sum_{i=0}^{k-1}\delta^i} \cdot (\ell+1)^{\sum_{i=0}^{k}\delta^i}$. It's easy to check that the base case $k = 1$ holds, and we prove the claim by induction:

$$\begin{aligned} S_{k+1} &= \mathcal{C}_R(\mathcal{C}_R^{\sum_{i=0}^{k-1}\delta^i} \cdot (\ell+1)^{\sum_{i=0}^{k}\delta^i})^\delta \cdot (\ell+1) \\ &= \mathcal{C}_R^{1+\sum_{i=0}^{k-1}\delta^{i+1}} \cdot (\ell+1)^{1+\sum_{i=0}^{k}\delta^{i+1}} \\ &= \mathcal{C}_R^{\sum_{i=0}^{k}\delta^i} \cdot (\ell+1)^{\sum_{i=0}^{k+1}\delta^i}. \end{aligned}$$

Thus we have that $S(\texttt{reg}(\varphi)) = \mathcal{C}_R^{\sum_{i=0}^{\ell-1}\delta^i} \cdot (\ell+1)^{\sum_{i=0}^{\ell}\delta^i} = O(\mathcal{C}_R^{\delta^\ell} \cdot (\ell+1)^{\delta^{\ell+1}})$.

Through a similar analysis we've found that the time complexity of all of the above functions is the same as their space complexity.

Theorem 6 **(*Time Complexity*).** *Given a well formed MLTL formula φ, $\texttt{reg}(\varphi)$ has worst-case time complexity that is $O(\mathcal{C}_R^{\delta^\ell} \cdot (\ell+1)^{\delta^{\ell+1}})$, where ℓ is the number of logical connectives ($\wedge, \vee, \mathcal{F}, \mathcal{G}, \mathcal{R}, \mathcal{U}$) in φ.*

If in the worst case no simplification occurs in any call of `set_intersect`, space complexity remains unchanged, but simplifying a set of regular expressions is cubic in input size. In practice however, both time and space complexities are much more optimistic than worst-case estimates.

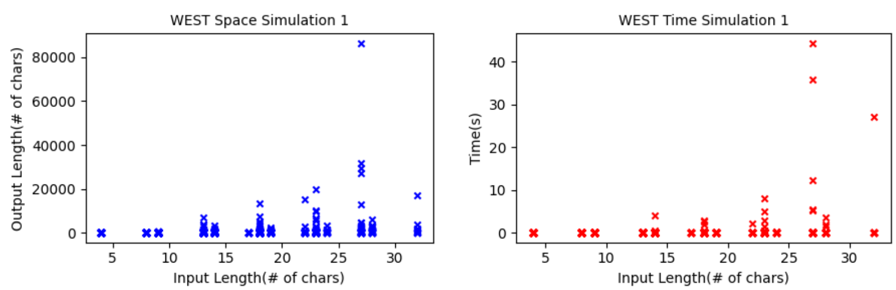
4.3 Experimental Benchmarking

We accompany theoretical space and time complexity with experimental evaluation of these complexities using randomly-generated MLTL formulas. What we observed from the simulations is that the worst case complexity, both for space and time, is relatively rare, and that otherwise the program has good complexity. WEST ran in under 10 seconds for nearly all the inputted random formulas. The number of characters outputted was typically under 5000, and often less. This is approximately the length of a single paragraph. We also observe that these worst cases are extreme outliers and that in nearly every case, are examples of nested binary temporal operators. As seen in [14], [16], [2], and [13], nested binary temporal operators do not appear in any specifications, and thus are likely rare in the literature and in practical applications.

We ran these experiments on an Intel(R) Xeon(R) Gold 6140 CPU @ 2.30GHz with 376 GB RAM. For each simulation, we randomly generated 1000 MLTL formulas using the parameters `delta`, `interval_max`, number of propositional variables, and number of iterations. `delta` is the maximum length we allow for any interval.

`interval_max` is the largest allowed upper bound for any interval. Number of iterations refers to the level of nesting in the generated formulas. For example, $\mathcal{G}_{[0,2]}p_0$ is a formula generated with one iteration, while $\mathcal{G}_{[0,2]}(p_0 \wedge p_1)$ is a formula generated with two iterations. We measure the number of characters in the output and the amount of time in microseconds taken to run the program. Note that these simulations run the `reg` function with the subformula boolean set to false and the simplify boolean set to true. For the pseudocode of the program that generated the random formulas, please see Appendix VI.

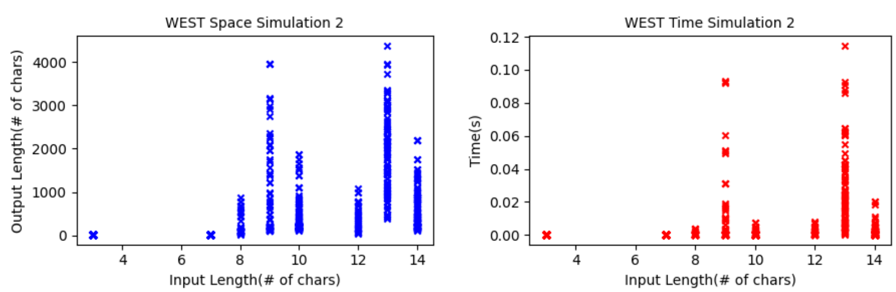
Simulation 1 For the first simulation, we consider 2 iterations, 5 propositional variables, 10 for `delta`, and 10 for `interval_max`. We obtain plots 2a and 2b.



(a) (b)

Fig. 2: $(p_0 = p_1)\mathcal{U}_{[2,9]}(p_1\mathcal{U}_{[7,9]}p_3)$ is an outlier in both. $(p_4 \rightarrow p_2)\mathcal{R}_{[1,8]}(p_3\mathcal{U}_{[3,4]}p_0)$ and $(p_3\mathcal{R}_{[2,7]}p_4)\mathcal{R}_{[1,9]}(p_4\mathcal{R}_{[4,9]}p_3)$ are outliers only in b.

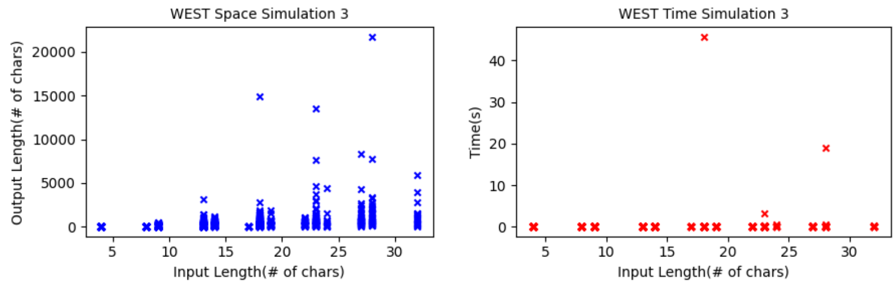
Simulation 2 For the second simulation, we consider 1 iteration, 10 propositional variables, 20 for `delta`, and 20 for `interval_max`. We obtain plots 3a and 3b.



(a) (b)

Fig. 3: We observe no outliers. The zero second run times are observed for MLTL formulas that consist of a single propositional variable or its negation.

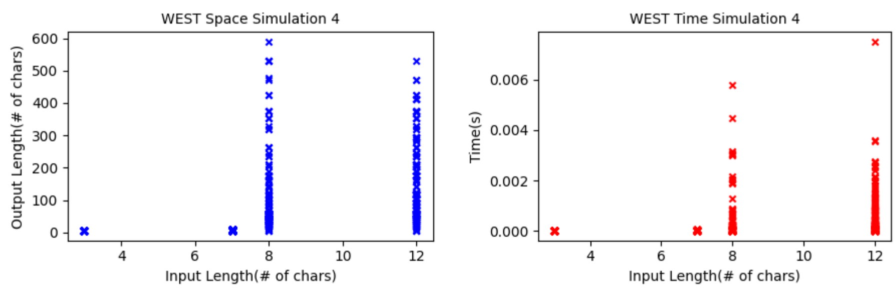
Simulation 3 For the third simulation, we consider 2 iterations, 10 propositional variables, 5 for `delta`, and 10 for `interval_max`. We obtain plots 4a and 4b.



(a) (b)

Fig. 4: $(p_7\mathcal{U}_{[5,7]}p_9)\mathcal{U}_{[3,7]}(\mathcal{F}_{[1,4]}p_4)$ and $\mathcal{G}_{[3,7]}(p_9\mathcal{U}_{[0,4]}p_0)$ are outliers in both. $\mathcal{G}_{[3,7]}(p_9\mathcal{U}_{[0,4]}p_0)$ is an outlier in 4a only.

Simulation 4 For the fourth simulation, we consider 1 iteration, 5 propositional variables, 10 for `delta`, and 10 for `interval_max`. We obtain plots 5a and 5b.



(a) (b)

Fig. 5: We observe no outliers. The zero second run times are observed for MLTL formulas that consist of a single propositional variable or its negation.

What we conclude from the simulations is that for most practical purposes, the WEST algorithm demonstrates good space and time complexity.

5 Correctness of WEST Tool Implementation

We accompany our proof of algorithmic correctness with a rigorous evaluation of implementation correctness, showing that our WEST tool correctly implements our WEST algorithm. Since our proof of algorithmic correctness is manual and our focus is on usability for validation by humans, we utilize more traditional techniques for robust software engineering with testing-based evaluation. The naïve approach is to test all inputs up to a certain size and verify the outputs, but this strategy would generate an unnecessarily large and redundant test suite. For instance, there is little sense in testing all MLTL formulas of the form $p_0\mathcal{U}_{[0,t]}p_1$ for all t such that $0 \leqslant t \leqslant 99$; verification of a few should give sufficient confidence of correctness of the program. Instead, we test our implementation with a test suite that explores all possible sequences of lines of code that are executed (up to a certain depth).

5.1 Intelligent Fuzzing

Traditional black-box fuzzing is defined by Miller [23]: *"If we consider a program to be a complex finite state machine, then our testing strategy can be thought of as a random walk through the state space, searching for undefined states."*
Instead we utilize intelligent fuzzing, an alternate approach that leverages knowledge about program structure to generate valid inputs and increase coverage. Borrowing the words of Miller, our approach to testing the WEST program can be thought of as walking all possible paths up to a certain depth of the state space of our algorithm. We first outline our overall approach to intelligent fuzzing:

1. Construct a directed graph representation of our algorithm. The edges capture control flow of our algorithm, and vertices represent non-branching blocks of code.
2. Construct a test suite that explores all possible paths in the digraph up to a certain depth. Run the WEST program on the test suite to produce a set of output files.
3. Run a naïve brute force generator of satisfying computations on the test suite and verify that both outputs match for all test cases.

State Diagram Construction We can represent the state space of the WEST algorithm as a directed graph with the edges representing the control flow and vertices representing blocks of contiguous code without branching statements. The core of the WEST program lies in the recursive routine, `reg`, which calls the 8 different subroutines as shown in Fig. 6. In order to construct the intelligent fuzzing test suite, we make the

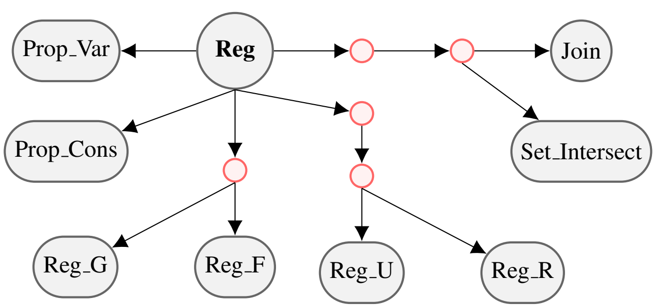


Fig. 6: Abstracted graph of the `reg` main routine. Red nodes signal recursive calls to `reg` on sub formulas of the input formula.

design choice to abstract away the eight subroutines in the overall state space diagram, despite the fact that they may have different possible execution paths within them. Without this abstraction, attempting to explore all execution paths in this finer graph is infeasible due to the explosion in the number of paths [26], some of which are provably impossible to construct a test input to explore.

Creating the Test Suite To generate our intelligent fuzzing test suite, we first count the number of formulas $\varphi(d)$ to be generated as a function of the exact depth d of recursion desired. For $d = 0$, only the paths leading to `prop_var` and `prop_cons` can be explored, so $\varphi(0) = 2$. For $d \geqslant 1$, we recursively calculate $\varphi(d+1) = 2\varphi(d)+4\varphi(d)^2$; paths to `reg_G` and `reg_F` is counted by the linear term, and paths to `reg_U`, `reg_R`, `set_intersect`, and `join` is counted by the quadratic term. This gives $\varphi(1) = 20$, $\varphi(2) = 1640$, and $\varphi(3) = 10761680$. $d = 3$ is computationally infeasible, and $d = 1$

doesn't give us assurance about operators interacting with each other through nesting. Thus we select $d = 2$ as a happy medium. We generate the full test suite in a similar recursive manner. Firstly, the $d = 0$ test suite consists of two formulas: a propositional variable or its negation, and a propositional constant. Then for any $d \geqslant 1$, we iterate through all formulas in the depth $d-1$ test suite for `reg_G` and `reg_F`, and all pairs of formulas from the $d-1$ test suite for the remaining four recursive paths. To ensure wider coverage, we randomly generate each of the propositional variables or their negation and propositional constants.

Verifying against Naïve Brute Force A relatively straightforward approach to generating the set of all satisfying computations of an MLTL formula φ over n variables, such that $m = \text{cplen}(\varphi)$, is to iterate over all $2^{m \cdot n}$ possible computations, which counts all possible length $m \cdot n$ bit strings. An interpreter function takes computation π and MLTL formula φ and determines if $\pi \models \varphi$ purely based on the semantics that define MLTL. Our test program translates every first-order quantifier into a loop; then checking for satisfying conditions of the suffix of a computation naturally lends itself to recursion. The full implementation details are available in the WEST GitHub repository [Link][8]. On an Intel(R) Core(TM) i7-4770S CPU at 3.10GHz with 32gb RAM, the brute force program took nearly nine hours to execute the depth two test suite of 1640 formulas. For this test suite, we fixed the number of propositional variables at $n = 4$ and the largest computation length was $m = 5$, from formulas with doubly-nested temporal operators. In comparison, the WEST program executed the same test suite in under thirty minutes on the same machine. Note that the brute force program outputs only computations of zeros and ones, and thus comparing the outputs of the WEST program requires expanding out the "S" characters in the regular expressions. It's important to state that although the full test suite matches between both implementations, absolute correctness on all inputs is not guaranteed for either program. However, the successful execution of the test suite gives us a much higher confidence in correctness of both the WEST program and the brute force program.

6 Regular Expression Simplification Theorem

As a final result, we provide a regular expression simplification theorem. This theorem describes the form of a set of computations that simplify down to all "S" characters. This theorem may help users identify tautologies, as the WEST program does not always output a single string of "S" characters when a formula holds true for every computation. We first define some vocabulary. We say a regular expression composed entirely of "S" characters and commas is an *arbitrary computation*. For the purposes of the following theorem, we remove all commas from computations. We say a "0" or a "1" in a regular expression is a *fixed truth value*. We overload the definition of a matrix and say that a *matrix* is a representation of a union of regular expressions, where each row is a regular expression. This aids significantly in the description and proof of the theorem; note, however, that matrix algebra does not apply in this definition.

Theorem 7 ***(Regular Expression Simplification Theorem).*** *Let M be a $n+1$ by n matrix, where each of the $n+1$ rows represents a regular expression of length n with*

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

commas stripped. If each column has one "1," one "0," and $n-1$ "S" characters, then the union of this set of regular expressions can be simplified to S^n, the arbitrary computation of length n.

Proof. Follows from induction and the pigeon hole principle. See Appendix V for details.

This theorem gives us a sufficient but not a necessary condition for simplification to the arbitrary computation. One such example is the regular expression $101 \vee S1S \vee 1S0 \vee 0SS$, which fails the hypothesis of the simplification theorem but is still equivalent to SSS.

6.1 Theoretical and Experimental Analysis of REST

We present an algorithm based on Theorem 7 for simplifying disjunctions of regular expressions and provide theoretical analysis as well as experimental benchmarking. We determine that REST runs exponentially with respect to the length of the inputted regular expressions, both in the worst case and average case. Since REST does not apply for many inputted MLTL formulas, however, and WEST demonstrates good time and space complexity in the average case without REST (see section 4.3), this is not of much concern.

Algorithm 5 Regular Expression Simplification Algorithm (REST)

Inputs: vector v of m regular expressions each of length n
Output: simplified set of equivalent regular expressions using Theorem 7

procedure REST(set v)
  for $r \in [3, \min(m, n+1)]$ **do**
    for all vectors of regular expression $w \subseteq v$ s.t. $|v| = r$ **do**
      diff_cols ← indices of all columns of w that aren't uniformly the same character
      if |diff_cols| $= r - 1$ **then**
        $w' \leftarrow w$ containing only columns $\in$ diff_cols
        if w' satisfies Theorem 7 **then**
          In w, replace all columns $\in$ diff_cols with s
          $v \leftarrow$ remove_duplicates(v)
  return v

Theorem 8. *On input vector of regular expressions v of m strings each of length n, REST has a worst case runtime of $O(n^2 2^m)$.*

Proof. We'll first analyze statements in the innermost loop. In line 4, constructing diff_cols can be done in $O(nr)$ by iteratively scanning the columns of w. In line 7 checking if w' satisfies the conditions of Theorem 7 is done in $O(r^2)$ time by keeping a count of the number of $0, 1, S$ in each column. Thus, the innermost portion of the loop has run time $= O(nr + r^2) = O(n^2)$. The total run time is bounded as follows:

$$\text{runtime(REST}(v)) = \sum_{r=3}^{min(m,n+1)} \binom{m}{r} O(n^2) = O(n^2 2^m).$$

We provide an experimental evaluation of run time using randomly-generated sets of regular expressions satisfying the conditions of REST. Unfortunately, results suggest that the average case time complexity of is the worst case.
We generated 100 sets of regular expressions satisfying the REST conditions, with n between 10 and 25. We measured the amount of time in seconds taken to run the program. We ran these experiments on an Intel(R) Xeon(R) Gold 6140 CPU @ 2.30GHz with 376 GB RAM, taking over one hour. We conclude that REST is not advisable to use as a part of the WEST program because it is often too computationally expensive.

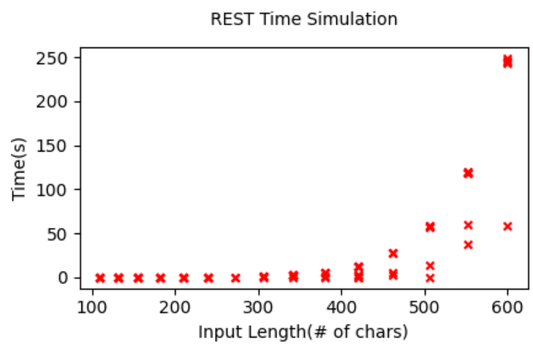


Fig. 7: For most inputs, REST's run time is exponential with respect to the input length.

7 Using WEST: An Example

In this section we include a [Video][9] tutorial to show how a user can use the WEST program to explore specifications. The video uses the example formula $(p_0 \wedge \mathcal{G}_{[0,3]} p_1) \rightarrow p_2$ from [16]. The WEST program interface can aid a user in the process of validation by allowing them to explore the behavior of the formula and its subformulas. The user can input specific computations and confirm whether or not they satisfy the given formula.

8 Closing Remarks

The primary goal of this work is to visually represent MLTL formulas to aide in debugging of MLTL specifications in industrial domains. We have accomplished this with our regular expressions framework, which captures many structural patterns of satisfying computations for a given MLTL formula. The tool itself has demonstratively reasonable run time for most inputs, and the correctness of outputs has been verified to a high degree of confidence through intelligent fuzzing.

8.1 Additional Research Directions

The WEST algorithm and release of our open-source tool open many promising avenues for future research.

Similar Analysis of LTL and LTLf. ω-regular expressions match only infinite words, and can be used to describe satisfying computations of LTL formulas.Our work on validation for MLTL formulas lays the groundwork for future work on validation of the

[9] https://youtu.be/HoBJwdCq42c

finite-trace logic LTLf [9] as well as LTL formulas. For the infinite-trace semantics of LTL, the particular difficulties revolve around the Kleene star operation, which behave poorly due to computing infinite unions and intersections of regular expressions.

Simplifying unions of regular expressions to a minimal representation. Theorem 7 addresses a non-trivial situation in which a set of regular expressions may be simplified. A natural question to ask is what is the minimum number of regular expressions needed to represent a language of computations. However, such a minimal representation is not unique. For instance, $\{S1, 1S\} = \{S1, 10\} = \{01, 10, 11\}$. Other schemes may be needed to simplify any arbitrary union of regular expressions to a minimal representation.

Intelligent Fuzzing to General Recursive Algorithms. A tool to systematically convert a recursive algorithm into a directed graph representation of the state space would be a helpful aid for generating test suites. Additional care should be put into allowing for varying levels of abstractions of execution due to concerns of the path explosion problem. We note that such avenues for code-level verification will still be necessary even upon the completion of potential future work avenues like synthesizing the core implementation of the WEST algorithm from an interactive theorem prover. This is because the goal of explainability to humans will require at least some manual code authorship for the foreseeable future.

Synthesizing verified code from MLTL behavior descriptions. The truth tables generated by the WEST tool can now serve as input to a recently-published toolchain [3]. This newly-enabled workflow would produce encodings of the represented MLTL behavior for the interactive theorem prover ACL2, including automatically generating related properties of general interest such as unambiguousness [4,15,6]. Next, a synthesis pipeline consisting of a verified program transformation suite [18] along with a proof-generating C code generator [5] (both built on ACL2) generates verified software implementing the behaviors originally described in MLTL. By providing a new front-end for this tool chain, we have now enabled a path to generating provably correct software from validated MLTL formulas describing the desired behaviors of a system. Considering the rising popularity of MLTL for describing such behaviors, we expect this to be a rewarding avenue for future exploration.

References

1. Alur, R., Henzinger, T.A.: Real-time logics: complexity and expressiveness. Information and Computation **104**(1), 35–77 (1993)
2. Aurandt, A., Jones, P.H., Rozier, K.Y.: Runtime Verification Triggers Real-Time, Autonomous Fault Recovery on the CySat-I. In: Proceedings of the 13th NASA Formal Methods Symposium (NFM). pp. 816–825. LNCS, Springer International Publishing (2022). https://doi.org/10.1007/978-3-031-06773-0_45
3. Cofer, D., Sattigeri, R., Amundson, I., Babar, J., Hasan, S., Smith, E., Nukala, K., Osipychev, D., Timmerman, L., Margineantu, D., Paunicka, J., Boeing, M.: Flight test of a collision avoidance neural network with run-time assurance. In: 2022 IEEE/AIAA 41st Digital Avionics Systems Conference (DASC) (09 2022)
4. Coglio, A.: A complex java code generator for ACL2 based on a shallow embedding of ACL2 in java. In: Sumners, R., Chau, C. (eds.) Proceedings Seventeenth International Workshop on the ACL2 Theorem Prover and its Applications, Austin, Texas, USA, 26th-27th May 2022. EPTCS, vol. 359, pp. 168–184 (2022). https://doi.org/10.4204/EPTCS.359.14, https://doi.org/10.4204/EPTCS.359.14
5. Coglio, A.: A proof-generating C code generator for ACL2 based on a shallow embedding of C in ACL2. In: Sumners, R., Chau, C. (eds.) Proceedings Seventeenth International Workshop on the ACL2 Theorem Prover and its Applications, Austin, Texas, USA, 26th-27th May 2022. EPTCS, vol. 359, pp. 185–201 (2022). https://doi.org/10.4204/EPTCS.359.15, https://doi.org/10.4204/EPTCS.359.15
6. Community, T.A.: The acl2 theorem prover and community books: Documentation. https://www.cs.utexas.edu/~moore/acl2/manuals/current/manual/, accessed: 2022-10-09
7. 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)
8. Dabney, J.B., Badger, J.M., Rajagopal, P.: Adding a verification view for an autonomous real-time system architecture. In: AIAA Scitech 2021 Forum. p. 0566 (2021)
9. De Giacomo, G., Vardi, M.: Linear Temporal Logic and Linear Dynamic Logic on Finite Traces. In: IJCAI. pp. 2000–2007. AAAI Press (2013)
10. Dion, S.: Global Precipitation Measurement (GPM) Safety Inhibit Timeline Tool. Tech. Rep. GSFC.ABS.7501.2012, NASA Goddard Space Flight Center, Greenbelt, MD, United States (2013), https://ntrs.nasa.gov/citations/20130000831
11. Erzberger, H., Heere, K.: Algorithm and operational concept for resolving short-range conflicts. Proc. IMechE G J. Aerosp. Eng. **224**(2), 225–243 (2010). https://doi.org/10.1243/09544100JAERO546, http://pig.sagepub.com/content/224/2/225.abstract
12. 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 in NTRS (2020)
13. Hammer, A., Cauwels, M., Hertz, B., Jones, P., Rozier, K.Y.: Integrating Runtime Verification into an Automated UAS Traffic Management System. Innovations in Systems and Software Engineering: A NASA Journal (July 2021). https://doi.org/10.1007/s11334-021-00407-5
14. 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). pp. 151–159. LNCS, Springer International Publishing (May 2021). https://doi.org/10.1007/978-3-030-76384-8_10

15. Kaufmann, M., Moore, J.S.: The acl2 theorem prover: Website. https://www.cs.utexas.edu/users/moore/acl2/, accessed: 2022-10-09
16. 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
17. Kessler, F.B.: nuXmv 1.1.0 (2016-05-10) Release Notes. https://es-static.fbk.eu/tools/nuxmv/downloads/NEWS.txt (2016)
18. Kestrel Institute: APT: Automated Program Transformations. https://www.kestrel.edu/home/projects/apt/ (2020)
19. Li, J., Vardi, M.Y., Rozier, K.Y.: Satisfiability checking for Mission-time LTL (MLTL). Information and Computation p. 104923 (2022). https://doi.org/https://doi.org/10.1016/j.ic.2022.104923
20. 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
21. 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
22. Maler, O., Nickovic, D.: Monitoring temporal properties of continuous signals. In: FORMATS, pp. 152–166. Springer (2004)
23. Miller, B.P., Fredriksen, L., So, B.: An empirical study of the reliability of UNIX utilities. Communications of the ACM **33**(12), 32–44 (December 1990). https://doi.org/10.1145/96267.96279
24. Okubo, N.: Using R2U2 in JAXA program. Electronic correspondence (November–December 2020), series of emails and zoom call from JAXA with technical questions about embedding MLTL formula monitoring into an autonomous satellite mission with a provable memory bound of 200KB
25. Ouaknine, J., Worrell, J.: Some Recent Results in Metric Temporal Logic. In: Cassez, F., Jard, C. (eds.) Formal Modeling and Analysis of Timed Systems. pp. 1–13. Springer Berlin Heidelberg, Berlin, Heidelberg (2008)
26. Pham, V.T., Böhme, M., Roychoudhury, A.: Model-based whitebox fuzzing for program binaries. In: Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering. ACM (August 2016). https://doi.org/10.1145/2970276.2970316
27. Radio Technical Commission for Aeronautics: DO-333 – formal methods supplement to DO-178C and DO-278A (2011), https://www.rtca.org/content/standards-guidance-materials
28. Radio Technical Commission for Aeronautics: DO-178C/ED-12C – software considerations in airborne systems and equipment certification (2012), https://www.rtca.org/content/standards-guidance-materials
29. Radio Technical Commission for Aeronautics (RTCA): DO-254: Design assurance guidance for airborne electronic hardware (April 2000)
30. 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)

31. 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
32. 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://doi.org/10.29007/5pch, https://easychair.org/publications/paper/Vncw
33. Ryan, J., Cummings, M., Roy, N., Banerjee, A., Schulte, A.: Designing an interactive local and global decision support system for aircraft carrier deck scheduling. In: Infotech@Aerospace. AIAA (2011)
34. Sipser, M.: Introduction to the theory of Computation. Course Technology (2020)

Appendix

I Minimum Computation Length

Theorem 1 (***Minimum Computation Length of Until and Release***). *Let* $0 \leqslant a \leqslant b \in \mathbb{N}$ *and let* φ, ψ *be well-formed MLTL formulas in NNF. The minimum computation length of* Until *and* Release *is given by* $\text{cplen}(\varphi \mathcal{U}_{[a,b]} \psi) = \text{cplen}(\varphi \mathcal{R}_{[a,b]} \psi) = b + \max(\text{cplen}(\varphi) - 1, \text{cplen}(\psi))$.

Proof. Recall that $\text{reg}(\varphi\, \mathcal{U}_{[a,b]} \psi) = \bigvee_{i=a}^{b} \text{reg}\left(\mathcal{G}_{[a,i-1]}\varphi \wedge \mathcal{G}_{[i,i]}\psi\right)$. Thus

$$\begin{aligned}
\text{cplen}(\varphi \mathcal{U}_{[a,b]} \psi) &= \max_{a \leqslant i \leqslant b} \left(\text{cplen}(\mathcal{G}_{[a,i-1]}\varphi \wedge \mathcal{G}_{[i,i]}\psi)\right) \\
&= \max_{a \leqslant i \leqslant b} \left(\max(i - 1 + \text{cplen}(\varphi), i + \text{cplen}(\psi))\right) \\
&= \max_{a \leqslant i \leqslant b} \left(i + \max(\text{cplen}(\varphi) - 1, \text{cplen}(\psi))\right) \\
&= b + \max(\text{cplen}(\varphi) - 1, \text{cplen}(\psi)).
\end{aligned}$$

Now recall that $\text{reg}(\varphi \mathcal{R}_{[a,b]} \psi) = \text{reg}\left(\mathcal{G}_{[a,b]}\psi\right) \vee \bigvee_{i=a}^{b-1} \text{reg}\left(\mathcal{G}_{[a,i]}\psi \wedge \mathcal{G}_{[i,i]}\varphi\right)$. Thus

$$\begin{aligned}
\text{cplen}(\varphi \mathcal{R}_{[a,b]} \psi) &= \max\left(\text{cplen}(\mathcal{G}_{[a,b]}\psi), \text{cplen}\left(\bigvee_{i=a}^{b-1} \text{reg}\left(\mathcal{G}_{[a,i]}\psi \wedge \mathcal{G}_{[i,i]}\varphi\right)\right)\right) \\
&= \max\left(b + \text{cplen}(\psi), \max_{a \leqslant i \leqslant b-1} \left(\mathcal{G}_{[a,i]}\psi \wedge \mathcal{G}_{[i,i]}\varphi\right)\right) \\
&= \max\left(b + \text{cplen}(\psi), \max_{a \leqslant i \leqslant b-1} \left(\max(i + \text{cplen}(\psi), i + \text{cplen}(\varphi))\right)\right) \\
&= \max\left(b + \text{cplen}(\psi), \max(b - 1 + \text{cplen}(\varphi), b - 1 + \text{cplen}(\psi))\right) \\
&= \max(b + \text{cplen}(\psi), b - 1 + \text{cplen}(\varphi), b - 1 + \text{cplen}(\psi)) \\
&= b + \max(\text{cplen}(\varphi) - 1, \text{cplen}(\psi)).
\end{aligned}$$

II Soundness and Completeness

Theorem 2 (***Soundness and Completeness***). *For any well formed MLTL formula* φ *in negation normal form, a computation* π *with* $|\pi| = \text{cplen}(\varphi)$ *satisfies* φ *and if and only if* $\pi \in \mathscr{L}(\text{reg}(\varphi))$.

Proof. We proceed by induction on the length of the formula.

Base Cases.

reg($\top$**)**

Any computation π satisfies $\top$, and $\mathscr{L}(\text{reg}(\top))$ is the set of all computations of one time step. Padding conventions extends this to the set of all computations of any positive number of time steps, so we are done.

reg($\perp$**)**

There are no computations that satisfy $\perp$ and $\mathscr{L}(\text{reg}(\perp))$ is the empty string, so we are done.

reg(p_k**)**

For $0 \leqslant k \leqslant n-1$, consider the propositional variable p_k. If π satisfies p_k, then p_k evaluates to true at $\pi[0]$. This is equivalent to writing that $\pi[0]$ is of the form

$$S^k 1 S^{n-k-1}$$

, which is precisely reg(p_k).
reg($\neg p_k$)
If π satisfies $\neg p_k$, then $\neg p_k$ evaluates to true at $\pi[0]$. This is equivalent to writing that $\pi[0]$ is of the form

$$S^k 0 S^{n-k-1}$$

, which is precisely reg($\neg p_k$).
Inductive Step.
Suppose the theorem holds for MLTL formulas φ and ψ. We now show that it holds for $\varphi \wedge \psi$, $\varphi \vee \psi$, $\mathcal{F}_{[a,b]}\varphi$, $\mathcal{G}_{[a,b]}\varphi$, $\varphi\, \mathcal{U}_{[a,b]}\psi$, and $\varphi \mathcal{R}_{[a,b]}\psi$.
reg($\varphi \wedge \psi$)
We know that $\pi \models \varphi \wedge \psi$ iff $\pi \models \varphi$ and $\pi \models \psi$.
By the inductive hypothesis, $\pi \models \varphi$ iff $\pi \in \mathscr{L}(\text{reg}(\varphi))$ and
$\pi \models \psi$ iff $\pi \in \mathscr{L}(\text{reg}(\psi))$, so

$$\pi \models \varphi \wedge \psi \text{ iff } \pi \in (\mathscr{L}(\text{reg}(\varphi)) \wedge \mathscr{L}(\text{reg}(\psi))) = \mathscr{L}(\text{reg}(\varphi \wedge \psi)).$$

reg($\varphi \vee \psi$)
We know that $\pi \models \varphi \vee \psi$ iff $\pi \models \varphi$ or $\pi \models \psi$.
By the inductive hypothesis, $\pi \models \varphi$ iff $\pi \in \mathscr{L}(\text{reg}(\varphi))$ and
$\pi \models \psi$ iff $\pi \in \mathscr{L}(\text{reg}(\psi))$, so

$$\pi \models \varphi \vee \psi \text{ iff } \pi \in (\mathscr{L}(\text{reg}(\varphi)) \vee \mathscr{L}(\text{reg}(\psi))) = \mathscr{L}(\text{reg}(\varphi \vee \psi)).$$

reg($\mathcal{F}_{[a,b]}\varphi$)
We know that $\pi \models \mathcal{F}_{[a,b]}\varphi$ iff $|\pi| > a$ and $\exists i \in [a,b]$ such that $\pi_i \models \varphi$.
If $|\pi| = \text{cplen}(\mathcal{F}_{[a,b]}\varphi)$, $|\pi| > a$. Likewise, $\pi \in \mathscr{L}\left(\text{reg}\left(\mathcal{F}_{[a,b]}\varphi\right)\right)$ implies $|\pi| = \text{cplen}(\mathcal{F}_{[a,b]}\varphi)$, so the length condition is satisfied.
By the inductive hypothesis, $\pi_i \models \varphi$ iff $\pi_i \in \mathscr{L}(\text{reg}(\varphi))$, so $\pi \in \mathscr{L}((S^n,)^i\text{reg}(\varphi))$ for some $i \in [a,b]$. Equivalently,

$$\pi \in \mathscr{L}\left(\bigvee_{i=a}^{b}(S^n,)^i\text{reg}(\varphi)(,S^n)^{b-i}\right) = \mathscr{L}(\text{reg}(\mathcal{F}_{[a,b]}\varphi)).$$

reg($\mathcal{G}_{[a,b]}\varphi$)
We know that $\pi \models \mathcal{G}_{[a,b]}\varphi$ iff $|\pi| \leqslant a$ or $\forall i \in [a,b]$ $\pi_i \models \varphi$.
Since $|\pi| = \text{cplen}(\mathcal{G}_{[a,b]}\varphi)$ and $\text{cplen}(\mathcal{G}_{[a,b]}\varphi) > a$, the first option for satisfying the formula never occurs.
By the inductive hypothesis, $\pi_i \models \varphi$ iff $\pi_i \in \mathscr{L}(\text{reg}(\varphi))$, so $\pi \in \mathscr{L}((S^n,)^i\text{reg}(\varphi))$ for all $i \in [a,b]$. Equivalently,

$$\pi \in \mathscr{L}\left(\bigwedge_{i=a}^{b}(S^n,)^i\text{reg}(\varphi)(,S^n)^{b-i}\right) = \mathscr{L}(\text{reg}(\mathcal{F}_{[a,b]}\varphi)).$$

reg($\varphi\,\mathcal{U}_{[a,b]}\psi$)
We have that

$$\pi \models \varphi\,\mathcal{U}_{[a,b]}\psi \text{ iff } |\pi| > a \text{ and } \exists i \in [a,b] \text{ such that } (\pi_i \models \psi \text{ and } \forall a \leqslant j < i, \pi_j \models \varphi).$$

As argued in the Finally case, the length condition is satisfied.
By the inductive hypothesis, $\pi_i \models \psi$ iff $\pi_i \in \mathscr{L}(\text{reg}(\psi))$, or equivalently, $\pi \in \mathscr{L}(\text{reg}(\mathcal{G}_{[i,i]}\psi))$. Also, $\pi_j \models \varphi$ if and only if $\pi_j \in \mathscr{L}(\text{reg}(\varphi))$.
By the argument used in the Global case, we see that $\forall a \leqslant j < i,\ \pi_j \models \varphi$ is equivalent to $\pi \in \mathscr{L}(\text{reg}(\mathcal{G}_{[a,i-1]}\varphi))$.
Thus $\pi \models \varphi\,\mathcal{U}_{[a,b]}\psi$ iff $\pi \in \mathscr{L}((\text{reg}(\mathcal{G}_{[i,i]}\psi) \wedge \text{reg}(\mathcal{G}_{[a,i-1]}\varphi)))$ for some $i \in [a,b]$, or equivalently,

$$\pi \in \mathscr{L}\left(\bigvee_{i=a}^{b} \text{reg}\left(\mathcal{G}_{[a,i-1]}\varphi \wedge \mathcal{G}_{[i,i]}\psi\right)\right) = \mathscr{L}\left(\text{reg}(\varphi\,\mathcal{U}_{[a,b]}\psi)\right).$$

reg($\varphi\mathcal{R}_{[a,b]}\psi$)
We have that

$$\begin{aligned}&\pi \models \varphi\mathcal{R}_{[a,b]}\psi \text{ iff } |\pi| \leqslant a \text{ or } \forall i \in [a,b]\ \pi_i \models \psi \text{ or}\\ &\exists j \in [a,b] \text{ such that } (\pi_j \models \varphi \text{ and } \forall a \leqslant k \leqslant j, \pi_k \models \psi).\end{aligned}$$

As argued in the Global case, the first option for satisfying the formula never occurs.
By the Global case, the statement $\forall i \in [a,b]\ \pi_i \models \psi$ is equivalent to $\pi \in \mathscr{L}(\text{reg}(\mathcal{G}_{[a,b]}\psi))$ and, by the Finally case. the statement $\exists j \in [a,b]$ such that ($\pi_j \models \varphi$ and $\forall a \leqslant k \leqslant j$, $\pi_k \models \psi$) is equivalent to $\pi \in \mathscr{L}\left(\bigvee_{i=a}^{b} \text{reg}\left(\mathcal{G}_{[a,i]}\psi \wedge \mathcal{G}_{[i,i]}\varphi\right)\right)$. Hence

$$\begin{aligned}\pi \models \varphi\mathcal{R}_{[a,b]}\psi \text{ iff } \pi \in \mathscr{L}&\left(\text{reg}(\mathcal{G}_{[a,b]}\psi) \vee \bigvee_{i=a}^{b} \text{reg}\left(\mathcal{G}_{[a,i]}\psi \wedge \mathcal{G}_{[i,i]}\varphi\right)\right)\\ = \mathscr{L}&\left(\text{reg}(\mathcal{G}_{[a,b]}\psi) \vee \bigvee_{i=a}^{b-1} \text{reg}\left(\mathcal{G}_{[a,i]}\psi \wedge \mathcal{G}_{[i,i]}\varphi\right)\right)\\ = \mathscr{L}&\left(\text{reg}(\varphi\mathcal{R}_{[a,b]}\psi)\right).\end{aligned}$$

This completes the inductive step, and thus the proof. Since this proof addresses all possible MLTL formulas in negation normal form, it shows completeness along with soundness.

III Nested Until and Release Rewriting Theorem

Theorem 3 **(*Nested Until and Release Rewriting Theorem*).** *Any MLTL formula using the Until or Release operator can be rewritten with right-nested subformulas. Let* $\mathcal{B} = \mathcal{R}$ *or* $\mathcal{U}$. *Let* $a, b, c \in \mathbb{Z}_{\geqslant 0}$, $a \leqslant b$, *and* φ, ψ *be well-formed MLTL formulas in NNF. Then,* $\varphi\ \mathcal{B}_{[a,b+c]}\psi \equiv \varphi\ \mathcal{B}_{[a,b]}(\varphi\ \mathcal{B}_{[0,c]}\psi)$. *That is,* $\varphi\ \mathcal{U}_{[a,b+c]}\psi \equiv \varphi\ \mathcal{U}_{[a,b]}(\varphi\ \mathcal{U}_{[0,c]}\psi)$ *and* $\varphi\ \mathcal{R}_{[a,b+c]}\psi \equiv \varphi\ \mathcal{R}_{[a,b]}(\varphi\ \mathcal{R}_{[0,c]}\psi)$.

The proof of Theorem 3 appears below.

Proof. **Case 1:** $\mathcal{B} = \mathcal{U}$
Let $\gamma = \varphi\,\mathcal{U}_{[0,c]}\psi$. Then, $\varphi\,\mathcal{U}_{[a,b]}(\varphi\,\mathcal{U}_{[0,c]}\psi) = \varphi\,\mathcal{U}_{[a,b]}\gamma$ and
$\mathrm{reg}\,(\gamma) = \bigvee_{j=0}^{c} \mathrm{reg}\left(\mathcal{G}_{[0,j-1]}\varphi \wedge \mathcal{G}_{[j,j]}\psi\right)$. Thus

$$\begin{aligned}\mathrm{reg}\left(\varphi\,\mathcal{U}_{[a,b]}\gamma\right) &= \bigvee_{i=a}^{b} \mathrm{reg}\left(\mathcal{G}_{[a,i-1]}\varphi \wedge \mathcal{G}_{[i,i]}\mathrm{reg}\,(\gamma)\right)\\ &= \bigvee_{i=a}^{b} \mathrm{reg}\left(\mathcal{G}_{[a,i-1]}\varphi \wedge \mathcal{G}_{[i,i]}\left(\bigvee_{j=0}^{c} \mathrm{reg}\left(\mathcal{G}_{[0,j-1]}\varphi\right) \wedge \mathrm{reg}\left(\mathcal{G}_{[j,j]}\psi\right)\right)\right).\end{aligned}$$

$\mathcal{G}_{[i,i]}$ distributes over $\wedge$ and $\vee$, so

$$\begin{aligned}\mathrm{reg}\left(\varphi\,\mathcal{U}_{[a,b]}\gamma\right) &= \bigvee_{i=a}^{b} \mathrm{reg}\left(\mathcal{G}_{[a,i-1]}\varphi \wedge \bigvee_{j=0}^{c} \mathcal{G}_{[i,i]}\left(\mathrm{reg}\left(\mathcal{G}_{[0,j-1]}\varphi\right) \wedge \mathrm{reg}\left(\mathcal{G}_{[j,j]}\psi\right)\right)\right)\\ &= \bigvee_{i=a}^{b} \mathrm{reg}\left(\mathcal{G}_{[a,i-1]}\varphi \wedge \left(\bigvee_{j=0}^{c} \mathcal{G}_{[i,i]}\mathrm{reg}\left(\mathcal{G}_{[0,j-1]}\varphi\right) \wedge \mathcal{G}_{[i,i]}\mathrm{reg}\left(\mathcal{G}_{[j,j]}\psi\right)\right)\right).\end{aligned}$$

Since $\mathcal{G}_{[t_1,t_1]}\mathcal{G}_{[t_2,t_3]}\varphi \equiv \mathcal{G}_{[t_1+t_2,t_1+t_3]}\varphi$, we have

$$\begin{aligned}\mathrm{reg}\left(\varphi\,\mathcal{U}_{[a,b]}\gamma\right) &= \bigvee_{i=a}^{b} \mathrm{reg}\left(\mathcal{G}_{[a,i-1]}\varphi \wedge \left(\bigvee_{j=0}^{c} \mathrm{reg}\left(\mathcal{G}_{[i,i+j-1]}\varphi\right) \wedge \mathrm{reg}\left(\mathcal{G}_{[i+j,i+j]}\psi\right)\right)\right)\\ &= \bigvee_{i=a}^{b}\bigvee_{j=0}^{c} \mathrm{reg}\left(\mathcal{G}_{[a,i-1]}\varphi\right) \wedge \mathrm{reg}\left(\mathcal{G}_{[i,i+j-1]}\varphi\right) \wedge \mathrm{reg}\left(\mathcal{G}_{[i+j,i+j]}\psi\right).\end{aligned}$$

Since $\mathcal{G}_{[t_1,t_2-1]}\varphi \wedge \mathcal{G}_{[t_2,t_3]}\varphi \equiv \mathcal{G}_{[t_1,t_3]}\varphi$, we have

$$\begin{aligned}\mathrm{reg}\left(\varphi\,\mathcal{U}_{[a,b]}\gamma\right) &= \bigvee_{i=a}^{b}\bigvee_{j=0}^{c} \mathrm{reg}\left(\mathcal{G}_{[a,i+j-1]}\varphi\right) \wedge \mathrm{reg}\left(\mathcal{G}_{[i+j,i+j]}\psi\right)\\ &= \bigvee_{i+j=a}^{b+c} \mathrm{reg}\left(\mathcal{G}_{[a,i+j-1]}\varphi\right) \wedge \mathrm{reg}\left(\mathcal{G}_{[i+j,i+j]}\psi\right).\end{aligned}$$

Finally, let $k = i + j$. Thus

$$\begin{aligned}\mathrm{reg}\left(\varphi\,\mathcal{U}_{[a,b]}\gamma\right) &= \bigvee_{k=a}^{b+c} \mathrm{reg}\left(\mathcal{G}_{[a,k-1]}\varphi\right) \wedge \mathrm{reg}\left(\mathcal{G}_{[k,k]}\psi\right)\\ &= \mathrm{reg}\left(\varphi\,\mathcal{U}_{[a,b+c]}\psi\right).\end{aligned}$$

This shows that $\varphi\,\mathcal{U}_{[a,b]}(\varphi\,\mathcal{U}_{[0,c]}\psi) \equiv \varphi\,\mathcal{U}_{[a,b+c]}\psi$.
Case 2: $\mathcal{B} = \mathcal{R}$
We begin by rewriting the regex for Release in an equivalent form to better mirror the

structure of the Until case. The Release operator is the dual of the Until operator. Thus,

$$\begin{aligned}\operatorname{reg}\left(\varphi\,\mathcal{R}_{[a,b]}\psi\right) &= \operatorname{reg}\left(\neg\left(\neg\varphi\,\mathcal{U}_{[a,b]}\neg\psi\right)\right)\\ &= \neg\left(\bigvee_{i=a}^{b}\operatorname{reg}\left(\mathcal{G}_{[a,i-1]}\neg\varphi\wedge\mathcal{G}_{[i,i]}\neg\psi\right)\right).\end{aligned}$$

Global is the dual of Finally, so

$$\begin{aligned}\operatorname{reg}\left(\varphi\,\mathcal{R}_{[a,b]}\psi\right) &= \neg\left(\bigvee_{i=a}^{b}\operatorname{reg}\left(\neg\mathcal{F}_{[a,i-1]}\varphi\wedge\neg\mathcal{F}_{[i,i]}\psi\right)\right)\\ &= \neg\left(\bigvee_{i=a}^{b}\operatorname{reg}\left(\neg\left(\mathcal{F}_{[a,i-1]}\varphi\vee\mathcal{F}_{[i,i]}\psi\right)\right)\right)\\ &= \neg\neg\left(\bigwedge_{i=a}^{b}\operatorname{reg}\left(\mathcal{F}_{[a,i-1]}\varphi\vee\mathcal{F}_{[i,i]}\psi\right)\right)\\ &= \bigwedge_{i=a}^{b}\operatorname{reg}\left(\mathcal{F}_{[a,i-1]}\varphi\vee\mathcal{F}_{[i,i]}\psi\right).\end{aligned}$$

This completes the rewriting of the regex for Release. Now let $\gamma = \varphi\,\mathcal{R}_{[0,c]}\psi$. Then $\varphi\,\mathcal{R}_{[a,b]}(\varphi\,\mathcal{R}_{[0,c]}\psi) = \varphi\,\mathcal{R}_{[a,b]}\gamma$ and $\operatorname{reg}(\gamma) = \bigwedge_{j=0}^{c}\operatorname{reg}\left(\mathcal{F}_{[0,j-1]}\varphi\vee\mathcal{F}_{[j,j]}\psi\right)$. Thus

$$\begin{aligned}\operatorname{reg}\left(\varphi\,\mathcal{R}_{[a,b]}\gamma\right) &= \bigwedge_{i=a}^{b}\operatorname{reg}\left(\mathcal{F}_{[a,i-1]}\varphi\vee\mathcal{F}_{[i,i]}\operatorname{reg}(\gamma)\right)\\ &= \bigwedge_{i=a}^{b}\operatorname{reg}\left(\mathcal{F}_{[a,i-1]}\varphi\vee\mathcal{F}_{[i,i]}\left(\bigwedge_{j=0}^{c}\operatorname{reg}\left(\mathcal{F}_{[0,j-1]}\varphi\right)\vee\operatorname{reg}\left(\mathcal{F}_{[j,j]}\psi\right)\right)\right).\end{aligned}$$

$\mathcal{F}_{[i,i]}\equiv\mathcal{G}_{[i,i]}$, so this operation distributes over $\wedge$ and $\vee$. Thus

$$\begin{aligned}\operatorname{reg}\left(\varphi\,\mathcal{R}_{[a,b]}\gamma\right) &= \bigwedge_{i=a}^{b}\operatorname{reg}\left(\mathcal{F}_{[a,i-1]}\varphi\vee\bigwedge_{j=0}^{c}\mathcal{F}_{[i,i]}\left(\operatorname{reg}\left(\mathcal{F}_{[0,j-1]}\varphi\right)\vee\operatorname{reg}\left(\mathcal{F}_{[j,j]}\psi\right)\right)\right)\\ &= \bigwedge_{i=a}^{b}\operatorname{reg}\left(\mathcal{F}_{[a,i-1]}\varphi\vee\left(\bigwedge_{j=0}^{c}\mathcal{F}_{[i,i]}\operatorname{reg}\left(\mathcal{F}_{[0,j-1]}\varphi\right)\vee\mathcal{F}_{[i,i]}\operatorname{reg}\left(\mathcal{F}_{[j,j]}\psi\right)\right)\right).\end{aligned}$$

Since $\mathcal{F}_{[t_1,t_1]}\mathcal{F}_{[t_2,t_3]}\varphi\equiv\mathcal{F}_{[t_1+t_2,t_1+t_3]}\varphi$, we have

$$\begin{aligned}\operatorname{reg}\left(\varphi\,\mathcal{R}_{[a,b]}\gamma\right) &= \bigwedge_{i=a}^{b}\operatorname{reg}\left(\mathcal{F}_{[a,i-1]}\varphi\vee\left(\bigwedge_{j=0}^{c}\operatorname{reg}\left(\mathcal{F}_{[i,i+j-1]}\varphi\right)\vee\operatorname{reg}\left(\mathcal{F}_{[i+j,i+j]}\psi\right)\right)\right)\\ &= \bigwedge_{i=a}^{b}\bigwedge_{j=0}^{c}\operatorname{reg}\left(\mathcal{F}_{[a,i-1]}\varphi\right)\vee\operatorname{reg}\left(\mathcal{F}_{[i,i+j-1]}\varphi\right)\vee\operatorname{reg}\left(\mathcal{F}_{[i+j,i+j]}\psi\right).\end{aligned}$$

Since $\mathcal{F}_{[t_1,t_2-1]}\varphi \wedge \mathcal{F}_{[t_2,t_3]}\varphi \equiv \mathcal{F}_{[t_1,t_3]}\varphi$, we have

$$\begin{aligned}\text{reg}\left(\varphi\,\mathcal{R}_{[a,b]}\gamma\right) &= \bigwedge_{i=a}^{b}\bigwedge_{j=0}^{c}\text{reg}\left(\mathcal{F}_{[a,i+j-1]}\varphi\right) \vee \text{reg}\left(\mathcal{F}_{[i+j,i+j]}\psi\right) \\ &= \bigwedge_{i+j=a}^{b+c}\text{reg}\left(\mathcal{F}_{[a,i+j-1]}\varphi\right) \vee \text{reg}\left(\mathcal{F}_{[i+j,i+j]}\psi\right).\end{aligned}$$

Let $k = i + j$. Thus

$$\begin{aligned}\text{reg}\left(\varphi\,\mathcal{R}_{[a,b]}\gamma\right) &= \bigwedge_{k=a}^{b+c}\text{reg}\left(\mathcal{F}_{[a,k-1]}\varphi\right) \vee \text{reg}\left(\mathcal{F}_{[k,k]}\psi\right) \\ &= \text{reg}\left(\varphi\,\mathcal{R}_{[a,b+c]}\psi\right).\end{aligned}$$

Thus $\varphi\,\mathcal{R}_{[a,b]}(\varphi\,\mathcal{R}_{[0,c]}\psi) \equiv \varphi\,\mathcal{R}_{[a,b+c]}\psi$, so $\varphi\,\mathcal{B}_{[a,b+c]}\psi \equiv \varphi\,\mathcal{B}_{[a,b]}(\varphi\,\mathcal{B}_{[0,c]}\psi)$. This completes the proof.

IV Until and Release Duality Lemma

Lemma 1 **(*Until and Release Duality*).** *The definition of Release is equivalent to the dual of Until: $\varphi\mathcal{R}\psi \equiv \neg(\neg\varphi\mathcal{U}\neg\psi)$. That is to say, $\varphi\mathcal{R}_{[a,b]}\psi$ if and only if $|\pi| \leqslant a$ or $\forall s \in [a,b], (\pi_s \models \psi$ or $\exists t \in [a,s-1], \pi_t \models \varphi)$.*

Proof.
(⇒):

Suppose $\pi \models \varphi\mathcal{R}_{[a,b]}\psi$, so:

$$|\pi| \leqslant a \text{ or } \forall i \in [a,b], (\pi_i \models \psi) \text{ or } \exists j \in [a,b-1], (\pi_j \models \varphi \text{ and } \forall k \in [a,j]\pi_k \models \psi)$$

We proceed by cases to show that:

$$|\pi| \leqslant a \text{ or } \forall s \in [a,b], (\pi_s \models \psi \text{ or } \exists t \in [a,s-1], \pi_t \models \varphi) \tag{A1}$$

Case 0: If $|\pi| \leqslant a$, then we are immediately done. Case 1: Suppose $\forall i \in [a,b], \pi_i \models \psi$.

Through re-labeling, we have $\forall s \in [a,b], \pi_s \models \psi$. Then we clearly have:

$$|\pi| \leqslant a \text{ or } \forall s \in [a,b], (\pi_s \models \psi \text{ or } \exists t \in [a,s-1], \pi_t \models \varphi) \tag{A1}$$

Case 2: Suppose $\exists i \in [a,b], \pi_i \not\models \psi$.

Then we must have that:

$$\exists j \in [a,b-1], (\pi_j \models \varphi \text{ and } \forall k \in [a,j]\ \pi_k \models \psi) \tag{1}$$

We want to show that $\forall s \in [a,b], (\exists t \in [a,s-1], \pi_t \models \varphi)$.

Suppose by contradiction that:

$$\exists s \in [a,b], (\forall t \in [a,s-1], \pi_t \not\models \varphi) \tag{2}$$

Since $s \in [a, b]$ and $t \in [a, s-1]$, we have that $t \in [a, b-1]$.

Since $j \in [a, b-1]$ (from Line 1), we have that $\pi_j \not\models \varphi$ from Line 2.

However from Line 1 we have that $\pi_j \models \varphi$ and have thus derived a contradiction.

Thus, we now have that $\forall s \in [a, b]$, $(\exists t \in [a, s-1], \pi_t \models \varphi)$.

From this, we clearly have that:

$$|\pi| \leqslant a \text{ or } \forall s \in [a, b], (\pi_s \models \psi \text{ or } \exists t \in [a, s-1], \pi_t \models \varphi) \tag{A1}$$

Since these 3 cases exhaustively capture all cases fo the assumption, the ($\Rightarrow$) direction is proved.

($\Leftarrow$):

Suppose that $\pi \models \neg(\neg\varphi \mathcal{U} \neg\psi)$:

$$|\pi| \leqslant a \text{ or } \forall s \in [a, b], (\pi_s \models \psi \text{ or } \exists t \in [a, s-1], \pi_t \models \varphi) \tag{1}$$

We want to show $\pi \models \varphi R_{[a,b]} \psi$, that is:

$$|\pi| \leqslant a \text{ or } \forall i \in [a, b], (\pi_i \models \psi) \text{ or } \exists j \in [a, b-1], (\pi_j \models \varphi \text{ and } \forall k \in [a, j]\ \pi_k \models \psi) \tag{B1}$$

Case 0: If $|\pi| \leqslant a$, again we are immediately done.

Case 1: Suppose $\forall s \in [a, b]$, $(\pi_s \models \psi)$.

Relabeling s to i, we now have that $\forall i \in [a, b]$, $(\pi_s \models \psi)$, which implies line B1.

Case 2: Suppose $\exists s \in [a, b], \pi_s \not\models \psi$.

By re-labeling s to i, we have that $\exists i \in [a, b], \pi_i \not\models \psi$.

Since $[a, b]$ is a finite-discrete interval, there must exist a first i_1 s.t. $\pi_{i_1} \not\models \psi$, that is:

$$\exists i_1 \in [a, b], (\pi_{i_1} \not\models \psi \text{ and } \forall k \in [a, i_1 - 1], \pi_k \models \psi) \tag{2}$$

Since (line 2) $\exists i_1 \in [a, b], \pi_{i_1} \not\models \psi$, by Line 1 we have that: $\pi_{i_1} \models \psi$ or $\exists t \in [a, i_1 - 1]$, $\pi_t \models \varphi$.

Thus, we have that:

$$\exists t \in [a, i_1 - 1], \pi_t \models \varphi \tag{3}$$

Since $[a, t] \subseteq [a, i_1 - 1]$ and $\forall k \in [a, i_1 - 1]$, $\pi_k \models \psi$, we have that:

$$\forall k \in [a, t], \pi_k \models \psi \tag{4}$$

Since $[a, t] \subseteq [a, i_1 - 1] \subseteq [a, b-1]$, we have that $t \in [a, b-1]$, so let $j := t$.

Then from lines 3 and 4, we have that:

$$\exists j \in [a, b-1], (\pi_j \models \varphi \text{ and } \forall k \in [a, j]\ \pi_k \models \psi) \tag{5}$$

From line 5, we now get:

$$|\pi| \leqslant a \text{ or } \forall i \in [a, b], (\pi_i \models \psi) \text{ or } \exists j \in [a, b-1], (\pi_j \models \varphi \text{ and } \forall k \in [a, j]\ \pi_k \models \psi) \tag{B1}$$

Again the 3 cases are exhaustive of all cases in the assumption, this we have the ($\Leftarrow$) direction.
This finishes the proof.

V Regular Expression Simplification Theorem

Theorem 7 (***Regular Expression Simplification Theorem***)**.** *Let M be a $n+1$ by n matrix, where each of the $n+1$ rows represents a regular expression of length n with commas stripped. If each column has one "1," one "0," and $n-1$ "S" characters, then the union of this set of regular expressions can be simplified to S^n, the arbitrary computation of length n.*

Proof. Assume no row is the arbitrary computation, because then the union of the set of regular expressions would trivially simplify to the arbitrary computation.
We begin by showing that there must be at least one row in the matrix M that is composed of one fixed truth value and $n-1$ 'S's. This will be of use later in the proof. Because there are n columns and 2 fixed truth values in each column, there are $2n$ fixed truth values. There are $n+1$ rows, so the average number of fixed truth values per row is strictly less than 2 since $\frac{2n}{n+1} < 2$. Thus, there must exist at least one row composed of one fixed truth value and $n-1$ 'S's.
We now proceed by induction on the length of computations, n.
Base Cases: $n=1$ and $n=2$.
The $n=1$ case holds by definition:

$$\begin{bmatrix} 1 \\ 0 \end{bmatrix} \equiv S.$$

For $n=2$, we can manually verify that each possible matrix indeed satisfies the theorem. Note that because the union of regular expressions is commutative, any permutation of rows is equivalent:

$$\begin{bmatrix} 1 & S \\ S & 1 \\ 0 & 0 \end{bmatrix} \equiv \begin{bmatrix} 0 & S \\ S & 0 \\ 1 & 1 \end{bmatrix} \equiv \begin{bmatrix} 1 & S \\ S & 0 \\ 0 & 1 \end{bmatrix} \equiv \begin{bmatrix} 0 & S \\ S & 1 \\ 1 & 0 \end{bmatrix} \equiv SS.$$

Inductive Hypothesis:
Let $n \geqslant 2$. Assume that a matrix of regular expressions, H, with the following characteristics is equivalent to the arbitrary computation of length $n-1$: n rows, $n-1$ columns, one '1' per column, one '0' per column, and $n-2$ 'S's per column.
Inductive Step:
Consider a matrix of regular expressions, J, with the following characteristics: $n+1$ rows, n columns, one '1' per column, one '0' per column, and $n-1$ 'S's per column. We show J is equivalent to the arbitrary computation of length n.
As aforementioned, there must exist at least one row composed of one fixed truth value and $n-1$ 'S's, and the union of regular expressions is commutative. Thus, WLOG, let the first row of J, r_1, be a row with one known truth value. Suppose this known truth value is in column k, c_k, where $1 \leqslant k \leqslant n$.. Assume WLOG that this value is a 0.

Matrix J can be represented as follows:

$$
\begin{array}{c}
\quad\;\; \mathbf{c_1} \qquad \mathbf{c_{k-1}}\; \mathbf{c_k}\; \mathbf{c_{k+1}} \qquad \mathbf{c_n} \\
J = \left(\begin{array}{ccccccc}
S & \dots & S & 0 & S & \dots & S \\
 & & & S & & & \\
 & & & \vdots & & & \\
 & & & S & & & \\
 & \vdots & & 1 & & \vdots & \\
 & & & S & & & \\
 & & & \vdots & & & \\
 & & & S & & &
\end{array}\right)
\begin{array}{l}
\mathbf{r_1} \\ \mathbf{r_2} \\ \\ \\ \\ \\ \\ \mathbf{r_{n+1}}
\end{array}
\end{array}
$$

The first row of J represents half of the regular expressions contained in the arbitrary computation. The other half would be represented by a regular expression of all 'S's, except for a '1' at the k^{th} position. Thus, if the rest of J, rows r_2 through r_{n+1}, represents the other half of the arbitrary computation, then the matrix - the union of the set of the regular expressions - represents the arbitrary computation of length n. We show that this is indeed the case: Because the first row represents all the computations with '0' at the k^{th} position, every 'S' in c_k can be replaced with a '1', to avoid redundancy; the case for which each 'S_k' is '0' is a subset of r_1. Thus, matrix J can be represented as follows:

$$
\begin{array}{c}
\quad\;\; \mathbf{c_1} \qquad \mathbf{c_{k-1}}\; \mathbf{c_k}\; \mathbf{c_{k+1}} \qquad \mathbf{c_n} \\
J = \left(\begin{array}{ccccccc}
S & \dots & S & 0 & S & \dots & S \\
 & & & 1 & & & \\
 & \vdots & & \vdots & & \vdots & \\
 & & & 1 & & &
\end{array}\right)
\begin{array}{l}
\mathbf{r_1} \\ \mathbf{r_2} \\ \\ \mathbf{r_{n+1}}
\end{array}
\end{array}
$$

The problem reduces to showing that $J - r_1$, that is, J with the row r_1 removed, represents the other half of the arbitrary computation. Thus, let $J' = J - r_1$:

$$
\begin{array}{c}
\quad\;\; \mathbf{c_1}\; \mathbf{c_k}\; \mathbf{c_n} \\
J' = \left(\begin{array}{ccc}
 & 1 & \\
\vdots & \vdots & \vdots \\
 & 1 &
\end{array}\right)
\begin{array}{l}
\mathbf{r_2} \\ \\ \mathbf{r_{n+1}}
\end{array}
\end{array}
$$

Again, we want to show that J' represents the other half of the arbitrary computation. Recall that we specify this to be the union of regular expressions of all 'S's except for a '$1'$ at the k^{th} position. Because each row indeed contains a '$1'$ at the k^{th} position, the problem reduces to showing that $r_2 - c_k$ through $r_{n+1} - c_k$ represents the arbitrary computation, where $r_j - c_k$ is the row r_j with the k^{th} entry removed. Thus, we remove

c_k from J' and call this new matrix J'':

$$J'' = \begin{pmatrix} r_2 - c_k \\ . \\ . \\ . \\ r_{n+1} - c_k \end{pmatrix}$$

Because J'' is the result removing one row and one column from J, J'' has n rows and $n-1$ columns. In each column of J'', there remains one '1' and one '0'. Also, there are now $n-2$ 'S's in each column, because r_1 was removed. Thus, J'' is equivalent to H, and is therefore equivalent to the arbitrary computation by the inductive hypothesis. Because r_1 is equivalent to half of the arbitrary computation and J' is equivalent to the other half of the arbitrary computation, J is equivalent to the arbitrary computation, as the union of r_1 and J' is equivalent to J. Therefore, by induction, the theorem holds.

VI Pseudocode for the WEST Algorithm Functions

To compute the satisfying computations of an MLTL formula, many of the functions in the WEST program require the regular expressions of the satisfying computations of the subformulas as inputs. We denote these regexes by R and T. Additionally, n will always refer to the number of propositional variables, and nnf refers to an input formula in negation normal form.

Algorithm 6 WEST Algorithm

Inputs: φ - MLTL formula in NNF
φ_1 and φ_2 below are subformulas of φ
n - number of propositional variables
Output: set of REGEX satisfying φ

```
procedure REG(string φ, int n)
    if φ is ⊤ or ⊥ then
        return reg_prop_const(φ, n)
    if φ is p_k or ¬p_k then
        return reg_prop_var(φ, n)
    if φ = φ1 ∧ φ2 then
        return set_intersect(reg(φ1), reg(φ2),        n)
    if φ = φ1 ∨ φ2 then
        return join(reg(φ1), reg(φ2), n)
    if φ = F_[a,b] φ1 then
        return reg_F(reg(φ1), a, b, n)
    if φ = G_[a,b] φ1 then
        return reg_G(reg(φ1), a, b, n)
    if φ = φ1 U_[a,b] φ2 then
        return reg_U(reg(φ1), reg(φ2), a, b, n)
    if φ = φ1 R_[a,b] φ2 then
        return reg_R(reg(φ1), reg(φ2), a, b, n)
```

Algorithm 7 Pad a set to elements of all equal length

Input: set of strings that represents a regex, number of propositional variables
Output: set of strings padded to equal length

procedure PAD(set R, int n)
  max_Length ← $\max_{\{r\in R\}}$(r.length())
  for ($r \in R$) **do**
    diff ← (max_length−r.length()) / $(n+1)$
    $r \leftarrow r + (, S^n)^{\text{diff}}$
  return R

Algorithm 8 Computes regex for propositional constant

Input: String that is either "true" or "false", number of propositional variables
Output: set of strings that represents the appropriate satisfying computations

procedure REG_PROP_CONS(string nnf, int n)
  if (nnf = "true" and $n \neq 0$) **then** return $\{S^n\}$
  else return $\{\}$

Algorithm 9 Output the set of computation satisfying a propositional variable.

Input: String that represents a propositional variable or the negation of one, number of propositional variables
Output: set of strings that represents the appropriate satisfying computations

procedure REG_PROP_VAR(string nnf, int n)
  if (nnf = "pk", where k is a nonnegative integer) **then** return $\{S^k 1 S^{n-k-1}\}$
  if (nnf = "∼pk", where k is a nonnegative integer) **then** return $\{S^k 0 S^{n-k-1}\}$

Algorithm 10 Takes the intersection of two computations

Input: Two strings representing regexes
Output: Bitwise AND of the inputted strings

procedure BIT_WISE_AND(string r, string t)
  ret ← ""
  for (i ∈ $[0,\ r$.length()$]$) **do**
    if ($r[i] \wedge t[i] =$ "") **then** return ""
    else ret ← ret + $r[i] \wedge t[i]$
  return ret

Algorithm 11 set_intersect

Inputs: R, T - two sets of REGEX
n - number of propositional variables
Output: set of REGEX equal to $R \wedge T$

procedure SET_INTERSECT(R, T, n)
  Pad(R, T, n), return ← {}
  for $(r, t) \in R \times T$ **do**
    add bit_wise_and(r, t) to ret
  return simplify(ret)

Algorithm 12 Takes union of two regexes (combines two sets into one)

Input: sets of strings that represent regexes, simplify boolean
Output: set of strings that represents union of inputted regexes

```
procedure JOIN(set R, set T, bool simp)
    ret ← {}
    for (r ∈ R) do add r to ret
    for (t ∈ T) do add t to ret
    if (simp is true) then return simplify(ret)
    else return ret
```

Algorithm 13 Computes the regex for an MLTL formula F[a,b]φ

Inputs: set of strings representing the regex for φ, interval bounds, number of propositional variables, simplify boolean
Output: set of strings that represents the appropriate satisfying computations

```
procedure REG_F(set r_φ, int a, int b, int n, bool simp)
    pre ← ((‘S’)^n + ‘,’)^a
    comp ← r_φ
    if a > b then return {}
    for (1 ⩽ i ⩽ b − a) do
        temp_φ ← ((‘S’)^n + ‘,’)^i + r_φ
        comp ← join(comp, temp_φ, simp)
    return pre + comp
```

Algorithm 14 Computes the regex for an MLTL formula φU[a,b]ψ

Inputs: r_φ, r_ψ - sets of REGEX for MLTL formulas φ and ψ (after calling reg)
a, b - integers representing interval bound
n - number of propositional variables
Output: set of REGEX for $\varphi\mathcal{U}_{[a,b]}\psi$

```
procedure REG_U(r_φ, r_ψ, a, b, n)
    comp ← ((‘S’)^n + ‘,’)^a + r_ψ
    if a > b then return {}
    for (a ⩽ i ⩽ b − 1) do
        G1 ← reg_G(r_φ, a, i, n)
        G2 ← reg_G(r_ψ, i + 1, i + 1, n)
        temp_comp ← set_intersect(G1,      G2, n)
        comp ← join( comp, temp_comp)
    return comp
```

Algorithm 15 Computes the regex for an MLTL formula φR[a,b]ψ

Inputs: sets of strings representing the regexes for φ and ψ, interval bounds, number of propositional variables, simplify boolean
Output: set of strings that represents the appropriate satisfying computations

```
procedure REG_R(set r_φ, set r_ψ, int a, int b, int n, bool simp)
    comp ← reg_G(r_ψ, a, b, n, simp)
    if a > b then return {S^n}
    for (a ⩽ i ⩽ b − 1) do
        temp_comp ← set_intersect(reg_G(r_ψ, a, i, n), reg_G(r_φ, i, i, n), n, simp)
        comp ← join(comp, temp_comp, simp)
    return comp
```

Algorithm 16 Combines two strings that differ only by one character into one

Input: Two strings that represent regexes
Output: Single string that represents computations represented by both input strings or FAIL

```
procedure SIMPLIFY_STRING(string r, string t)
    if (r.length() ≠ t.length()) then exit
    for (0 ⩽ i < r.length()) do
        pre_r ← r[0, i − 1]
        char_r ← r[i]
        post_r ← r[i + 1, r.length() − 1]
        pre_t ← t[0, i − 1]
        char_t ← t[i]
        post_t ← t[i + 1, t.length() − 1]
        if (pre_r = pre_t and post_r = post_t) then
            if (char_r ≠ char_t) then
                return pre_r +"S" + post_r
            else
                return r
    return FAIL
```

Algorithm 17 Simplifies a set of strings using simplify_string

Input: set of strings representing a regex
Output: set of strings representing a regex simplified using simplify_string

```
procedure SIMPLIFY(set R)
    Pad all strings in R to the same length
    if (R.length() ⩽ 1) then return R
    i = R.length − 1
    j = i − 1
    START
    while (i ⩾ 1) do
        while (j ⩾ 0) do
            simplified = simplify_string(R[i], R[j])
            if (simplified ≠ FAIL) then
                replace string at index j with simplified
                remove string at index i from R
                i = R.length() − 1
                j = i − 1
                goto START
            --j
        --i
        j = i − 1
    return R
```

Algorithm 18 Generates test suite template without propositional constants, propositional variables, or negations of propositional variables filled in

Inputs: Depth of desired test suite template to generate, interval bounds
Output: set of MLTL formulas (without propositional constants, propositional variables, or negations of propositional variables)

```
procedure GENERATE_TEST_TEMPLATE(int depth, int a, int b)
    if (depth = 0) then return {p, q}
    template ← {}
    V ← generate_test_template(depth−1, a, b)
    for (string φ ∈ V) do
        add "G[a:b]" + φ to template
        add "F[a:b]" + φ to template
        for string ψ ∈ V do
            add φ + "U[a:b]" + ψ to template
            add φ + "R[a:b]" + ψ to template
            add φ + "∨" + ψ to template
            add φ + "&" + ψ to template
    return template
```

Algorithm 19 Generates a complete test suite of MLTL formulas in negation normal form up to a certain depth

Inputs: Depth of desired test suite template to generate, interval bounds, number of propositional variables
Output: set of MLTL formulas

```
procedure GENERATE_TEST(int depth, int a, int b, int n)
    tests ← generate_test_template(depth, a, b)
    for string t ∈ tests do
        for char ch ∈ t do
            if ch = p then
                k ← rand()%n
                if rand() %2 = 0 then replace ch with pk
                else replace ch with ∼pk
            else if ch = q then
                if rand() %2 = 0 then replace ch with T
                else replace ch with !
    return tests
```

VII State Diagram Graphs

Below we provide the state diagram graphs of the functions examined in our intelligent fuzzing. Nodes in the graph represent portions of the code without control flow statements, so a single node can represent large chunks of code. Directed edges represent branching of control flow, such as IF statements and loops. Each graph directly corresponds with the pseudocode of their corresponding function, with nodes and edges being labeled accordingly. Note that the option to run `simplify` has been omitted for clarity.

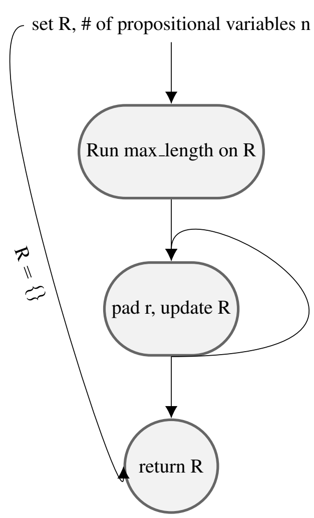


Fig. 8: Pad Function

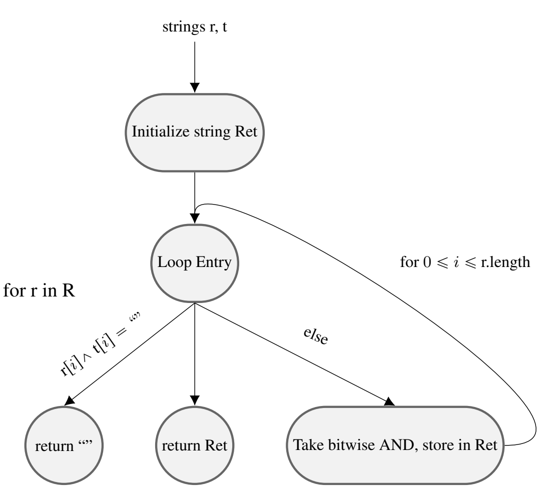


Fig. 9: Bitwise_And Function

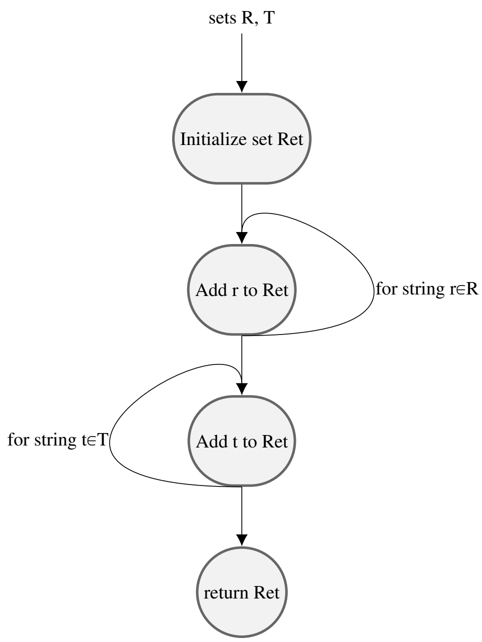


Fig. 10: Join Function

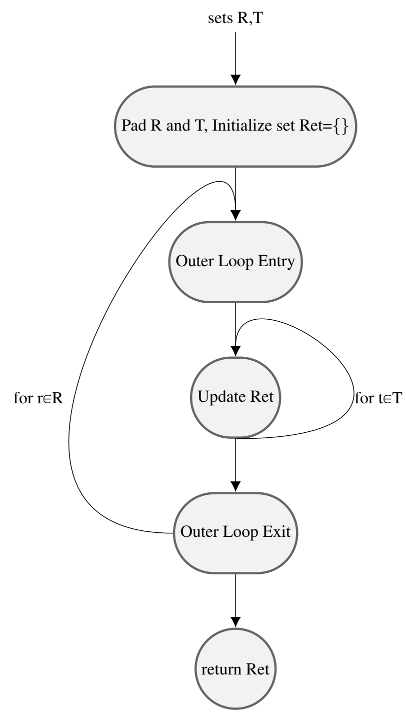


Fig. 11: Set_Intersect Function

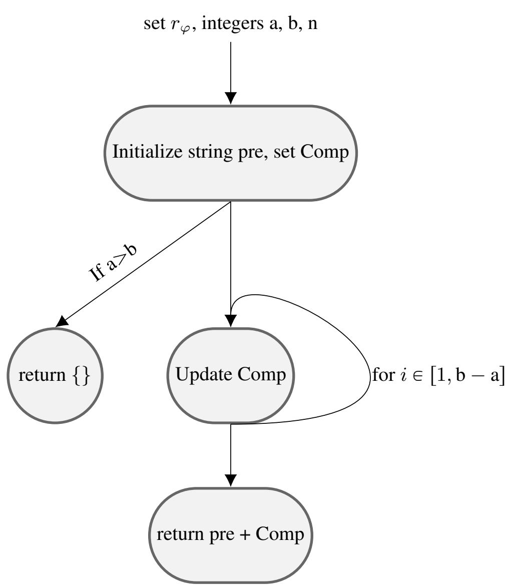


Fig. 12: Reg_F Function

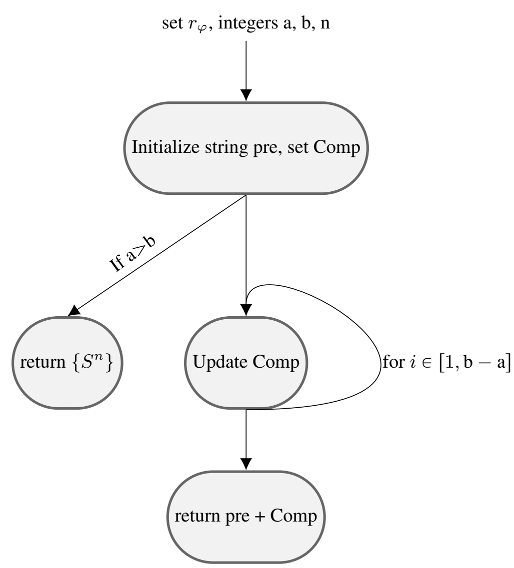


Fig. 13: Reg_G Function

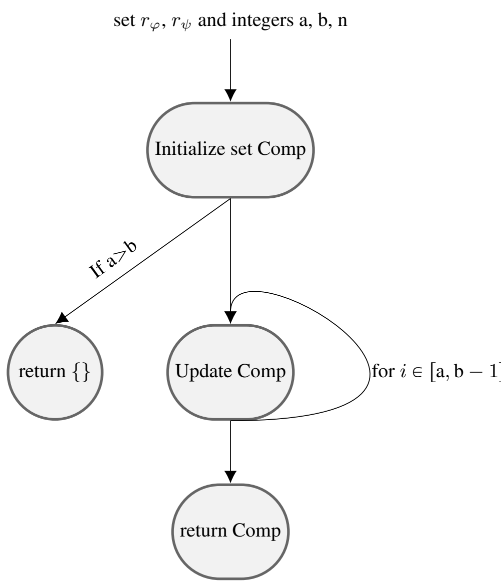


Fig. 14: Reg_U Function

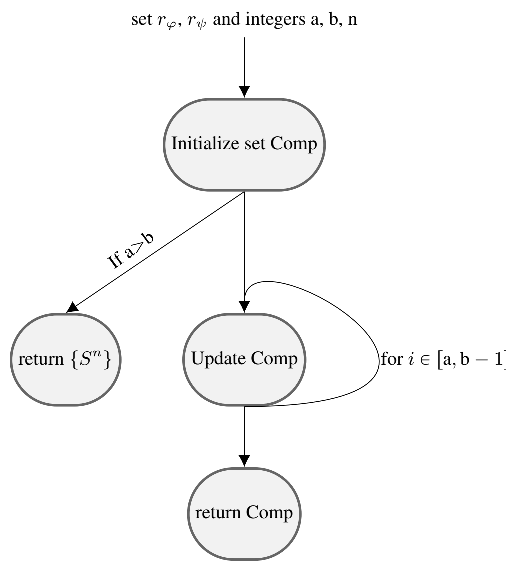


Fig. 15: Reg_R Function

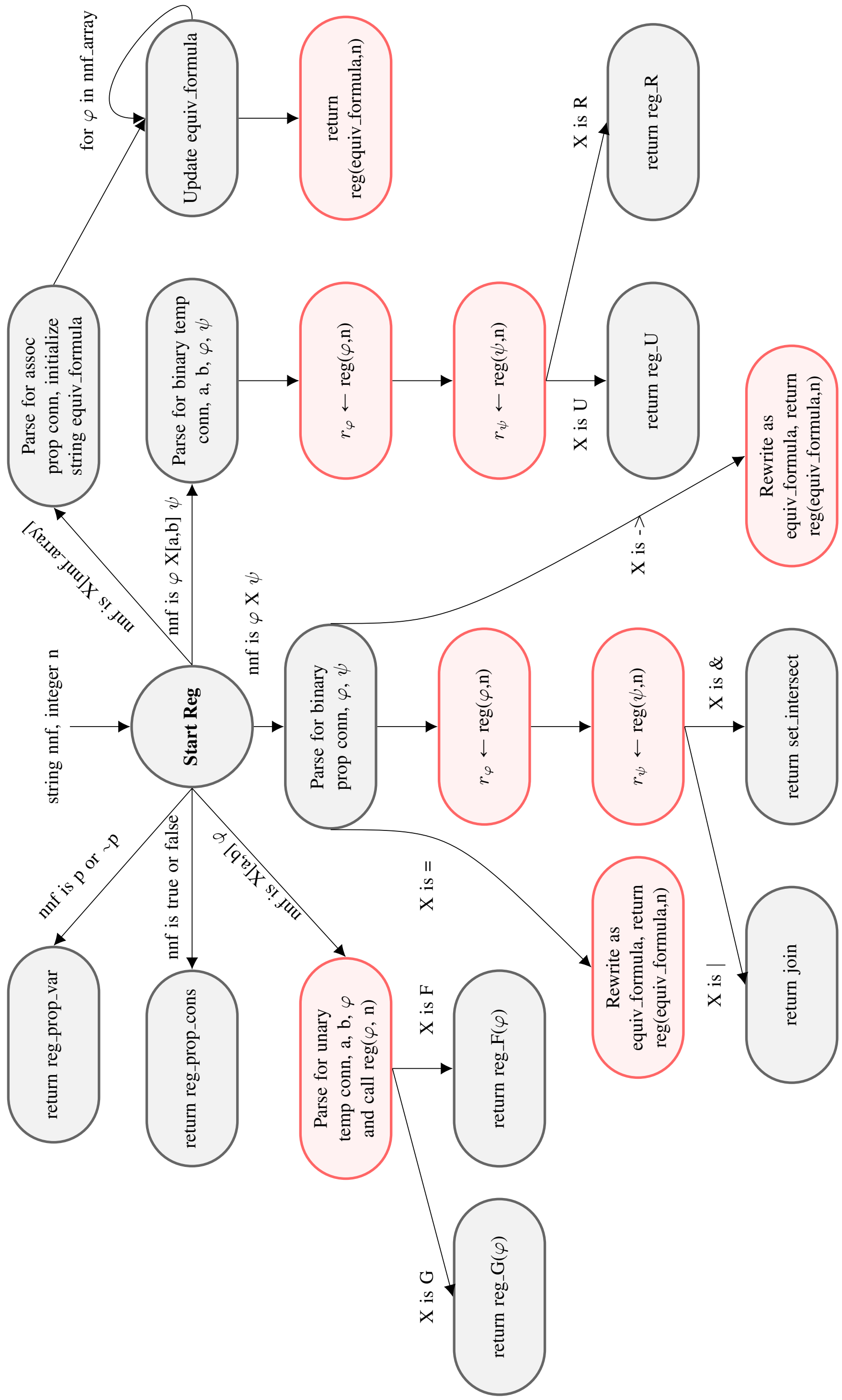


Fig. 16: Control Flow of Reg. Red nodes indicate a recursive call to Reg.