Describing Mission-time LTL Formulas with Regular Expressions using the WEST Program

Iowa State University REU

Jenna Elwing[*] Laura Gamboa Guzmán[†] Jeremy Sorkin[‡]
Chiara Travesset[§] Zili Wang[¶] Kristin Y. Rozier[†]

June 2022

Abstract

Mission-time Linear Temporal Logic (mLTL) is an extension of propositional logic that allows temporal quantifiers over finite discrete intervals. A computation is a finite sequence of truth assignments. In this paper, we use regular expressions to describe the structure of the computations that satisfy a given mLTL formula. We prove soundness and completeness of our structure. We also give an implemented algorithm (the WEST program) and analyze its complexity both theoretically and experimentally. Finally, we generate a test suite using control flow diagrams to robustly test the code.

Contents

[*]University of Michigan, Ann Arbor
[†]Iowa State University
[‡]The Ohio State University
[§]Purdue University
[¶]University of California, Berkeley

1 Introduction

Mission-time Linear Temporal Logic (mLTL) extends propositional logic by including temporal operators over finite discrete intervals. It is useful for writing specifications for spacecraft, rovers, aircraft, and other robotic missions [7]. In standard propositional logic, a convenient way to check that a formula captures a specification correctly is to write out its truth table and check that the formula behaves in the desired manner. The same strategy can be employed for mLTL, but the truth tables increase in size at a much faster rate, making them too large to be useful. For instance, an mLTL formula with just two propositional variables and 10 time steps would require $\left(2^{10}\right)^2 = 2^{20}$ rows. This is often called the "State Explosion Problem" [4].

In this paper, we offer a remedy to the state explosion problem by using regular expressions to describe the set of all computations satisfying a mLTL formula, which significantly reduces the space needed to express the computations. We implement these regular expressions into the WEST program, which allows the user to input a mLTL formula and obtain the satisfying computations. The WEST program can be found on its Github repository.

The structure of the paper is as follows: In section 2, we provide the necessary background for this paper. We give the semantic definitions for all the mLTL operators, define a computation, and describe the bit string representation of a computation that is used throughout the paper and the WEST program. Finally, we describe negation normal form (NNF), a standardized way to express mLTL formulas. By writing the mLTL formulas in NNF, the WEST program is able to run much more efficiently.

In section 3, we recursively define the regular expressions of the satisfying computations of a mLTL formula. We prove that these regular expressions are both sound and complete. In addition, we provide a calculation for the minimum computation length required to describe all the satisfying computations of a mLTL formula that slightly improves upon existing calculations in the literature. Finally, we show an application of the regular expressions of the satisfying computations by using them to prove a mLTL theorem. In particular, we show that $\alpha\mathcal{U}_{[a,b]}(\alpha\mathcal{U}_{[0,c]}\beta) = \alpha\mathcal{U}_{[a,b+c]}\beta$ and $\alpha\mathcal{R}_{[a,b]}(\alpha\mathcal{R}_{[0,c]}\beta) = \alpha\mathcal{R}_{[a,b+c]}\beta$.

In section 4, we give the WEST code specifications. In particular, we describe the grammar that the program follows and give an overview of the important functions in the program.

In section 5, we calculate the space and time complexity, both theoretically and experimentally, of the WEST program. We show that the theoretical worst case complexity for both space and time is doubly exponential, but that the average case fares far better.

In section 6, we first prove correctness of the algorithms and provide a test suite to show correctness of implementation with high confidence. Intelligent fuzzing techniques are used to construct the test suite from a state diagram representing the control flow of the algorithm. We then verify the correctness of outputs of the WEST program against a naive brute force implementation.

Lastly, in section 7, we provide a combinatorial theorem for simplifying certain outputs of the WEST program. Formulas of the form $\alpha \equiv \beta$ to check for equivalence between two expressions can lead to outputs that are non-trivial to simplify.

2 Mission-time LTL and Regular Expressions

2.1 Mission-time Linear Temporal Logic

Mission-time Linear Temporal Logic (mLTL) [7] is a finite variation of Linear Temporal Logic (LTL) [2] over bounded discrete intervals. Instead of taking the LTL operators over an infinite time line, we restrict the operators to a finite time line, taking each temporal operator over a closed interval $[a, b]$ where $a, b \in \mathbb{N}$ and $0 \leq a \leq b$. The syntax of an mLTL formula over a set of atomic propositions $\mathcal{AP}$, where $p \in \mathcal{AP}$ is a propositional variable, is given by the following BNF grammar:

$$\alpha := \top \mid \bot \mid p \mid \neg\alpha \mid \alpha \wedge \alpha \mid \alpha \vee \alpha \mid \mathcal{F}_{[a,b]}\alpha \mid \mathcal{G}_{[a,b]}\alpha \mid \alpha\mathcal{U}_{[a,b]}\alpha \mid \alpha\mathcal{R}_{[a,b]}\alpha$$[1]

Definition 1 (*Computation*)**.** A computation of length m is a sequence $\{\pi[i]\}_{i=0}^{m-1}$ of sets of propositional variables, $\pi[i] \subseteq \mathcal{AP}$, where the ith set holds the propositional variables that are satisfied at the ith 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 use a bit string representation for our computations. This representation is easy to describe using regular expressions and is convenient to code.

Definition 2 (*Bit String Representation*)**.** 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:

- We represent a true assignment to a propositional variable by 1 and a false assignment by 0.
- For each time step $j \in [0, 1, \ldots, m-1]$, 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 in chronological order.

We write that a computation π is equal to its bit string representation. An example is given below.

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.

[1]For simplicity, we do not include parentheses in the grammar, but they may used as needed. The WEST program uses parentheses (see section 4 for more details).

2.1.1 Semantics

We say a computation π as satisfies a given mLTL formula α, written $\pi \vDash \alpha$, when[2]:

$$\pi \vDash p \text{ iff } p \in \pi[0] \tag{1}$$
$$\pi \vDash \neg\alpha \text{ iff } \pi \nvDash \alpha \tag{2}$$
$$\pi \vDash \alpha \wedge \beta \text{ iff } \pi \vDash \alpha \text{ and } \pi \vDash \beta \tag{3}$$
$$\pi \vDash \alpha \vee \beta \text{ iff } \pi \vDash \alpha \text{ or } \pi \vDash \beta \tag{4}$$
$$\pi \vDash \mathcal{F}_{[a,b]}\alpha \text{ iff } |\pi| > a \text{ and } \exists i \in [a,b] \text{ such that } \pi_i \vDash \alpha \tag{5}$$
$$\pi \vDash \mathcal{G}_{[a,b]}\alpha \text{ iff } |\pi| \leq a \text{ or } \forall i \in [a,b]\ \pi_i \vDash \alpha \tag{6}$$
$$\pi \vDash \alpha\ \mathcal{U}_{[a,b]}\beta \text{ iff } |\pi| > a \text{ and } \exists i \in [a,b] \text{ such that } \pi_i \vDash \beta \text{ and } \forall j \in [a, i-1]\ \pi_j \vDash \alpha \tag{7}$$
$$\pi \vDash \alpha\ \mathcal{R}_{[a,b]}\beta \text{ iff } |\pi| \leq a \text{ or } \forall i \in [a,b]\ \pi_i \vDash \beta \text{ or } \exists j \in [a, b-1] \text{ such that } \pi_j \vDash \alpha \text{ and } \forall a \leq k \leq j\ \pi_k \vDash \beta \tag{8}$$

The reader may observe that in this paper, we define Release independently from Until. It is not uncommon for Release instead to be defined as the dual of Until [2] [7]. These two definitions are in fact equivalent, and the full proof is provided in Appendix I.

2.2 Negation Normal Form

All mLTL formulas can be written in negation normal form (NNF) [2]. We require mLTL formulas to be written in NNF to generate the satisfying computations using regular expressions. Without this requirement, generating the satisfying computations requires taking complements of the languages described by regular expressions, which is a very expensive operation.

Definition 3 (*Negation Normal Form [9]*)**.** An mLTL formula α is in *negation normal form* if

- Any negation symbols appear only in front of propositional variables.
- We allow only the operations $\top$, $\bot$, $\wedge$, and $\vee$ from propositional logic.
- We allow only the operations $\mathcal{U}$ and $\mathcal{R}$ from mLTL.

In the literature, NNF does not include the Finally and Global operators, as $\bot\mathcal{R}_{[a,b]}\alpha = \mathcal{G}_{[a,b]}\alpha$ and $\top\mathcal{U}_{[a,b]}\alpha = \mathcal{F}_{[a,b]}\alpha$. In this paper, we stray from the literature and include Finally and Global as part of NNF, as this is far more natural for writing our regular expressions that describe the satisfying computations of a mLTL formula.

2.3 Regular Expressions

We use regular expressions (regex) to describe the form of the satisfying computations.

Definition 4 (*Regular Expression [11]*)**.** A *regular expression* is a string that describes the form of a language. Let Σ denote a finite alphabet. The empty set $\emptyset$, the empty string $\epsilon =$ "", and any character in Σ are constants defined to be a regular expression. Beyond that, we define regular expressions recursively. Suppose R and T are regular expressions. Then the following operations generate regular expressions:

- *(Concatenation)* Denoted RT, this operation describes the set of strings obtained by concatenating any string generated by R and any string generated by T in that order.
- *(Alternation)* Denoted $R|T$, this operation describes the set union of the strings generated by R and T.
- *(Kleene Star)* Denoted R^*, this operation describes the set of all strings made by concatenating any finite number (including zero) of strings described by R together.

[2]In mLTL, we do not include the Next (χ) operator since $\mathcal{G}_{[1,1]}$ is equivalent to it.

- R^i denotes regular expression consisting of R repeated i times for $i \geq 0$. $R^0 = \epsilon$.

We provide an example of each operation.

Example 2. Let R describe {'*a*', '*b*'} and T describe { '*c*', '*d*' }.

- RT describes the set { '*ac*', '*ad*', '*bc*', '*bd*' }.
- $R|T$ describes the set {'*a*', '*b*', '*c*', '*d*' }.
- R^* describes the set { ϵ, '*a*', '*b*', '*aa*', '*bb*', '*ab*', '*ba*', '*aaa*', '*aba*',... }.
- R^2 describes the set { '*aa*', '*bb*', '*ab*', '*ba*' }.

We provide examples of different regex and possible combinations of the above operations.

Example 3. Let '*a*', '*b*', and '*c*' denote characters in a finite alphabet.

- ab^* describes the set { '*a*', '*ab*', '*abb*', '*abbb*', ... }.
- $a^*|b$ describes the set { ϵ, '*b*', '*a*', '*aa*', '*aaa*', ...}.
- $a(b^*)c$ describes the set { '*ac*', '*abc*', '*abbc*', '*abbbc*', ... }.
- $(a|b)^3$ describes the set { '*aaa*', '*aba*', '*aab*', '*abb*', '*bbb*', ... }.

3 mLTL Regular Expressions

We introduce the following additional notation for our regular expressions that describe the satisfying computations of a mLTL formula:

Definition 5 (*Additional Notation*)**.** Let R and T denote regular expressions, and let S be an abbreviation for $(0 \mid 1)$. Let fixed $n \in \mathbb{N}$ denote the number of propositional variables in a mLTL formula. We use the following operations in the next section to describe the form of satisfying computations of the formula in the bit string representation:

- $R \vee T$ denotes alternation. We use this notation to match our notation for set intersection defined below.
- Pad(R, T) determines which regular expression is longer and concatenate $(, S^n)$ 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.
- For $R \wedge T$, we first use Pad(R, T), and then take the set intersection of the sets of strings described by the two regular expressions.
- We do not use the Kleene star since our computations are a fixed finite length.

An example of these operations follows.

Example 4. Let there be $n = 2$ propositional variables, and let $R = S1$ and $T = 1S, 1S|S1, S1$. To compute $R \wedge T$ and $R \vee T$, we use Pad(R,T). Clearly T is the longer regex by one time step, so we concatenate $, S^2$ to the right of R. Thus $T = 1S, 1S|S1, S1$ and $R = S1, SS$. Now the regexes are the same length, so we can perform set intersection and alternation:

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

For the purposes of describing 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 $(0 \mid 1)$. Let $p_0, p_2, ..., p_{n-1}$ be propositional variables, α, β well-formed mLTL formulas in negation normal form. We define the regular expression of satisfying computations for an mLTL formula as follows:

$$\text{reg}(\top) = S^n \tag{9}$$

$$\text{reg}(\bot) = \emptyset \tag{10}$$

$$\text{reg}(p_k) = S^k 1 S^{n-k-1} \tag{11}$$

$$\text{reg}(\neg p_k) = S^k 0 S^{n-k-1} \tag{12}$$

$$\text{reg}(\alpha \vee \beta) = \text{reg}(\alpha) \vee \text{reg}(\beta) \tag{13}$$

$$\text{reg}(\alpha \wedge \beta) = \text{reg}(\alpha) \wedge \text{reg}(\beta) \tag{14}$$

$$\text{reg}(\mathcal{F}_{[a,b]}\alpha) = \bigvee_{i=a}^{b} (S^n,)^i \text{reg}(\alpha) \tag{15}$$

$$\text{reg}(\mathcal{G}_{[a,b]}\alpha) = \bigwedge_{i=a}^{b} (S^n,)^i \text{reg}(\alpha) \tag{16}$$

$$\text{reg}(\alpha\ \mathcal{U}_{[a,b]}\beta) = \bigvee_{i=a}^{b} \text{reg}\left(\mathcal{G}_{[a,i-1]}\alpha \wedge \mathcal{G}_{[i,i]}\beta\right) \tag{17}$$

$$\text{reg}(\alpha\mathcal{R}_{[a,b]}\beta) = \text{reg}\left(\mathcal{G}_{[a,b]}\beta\right) \vee \bigvee_{i=a}^{b-1} \text{reg}\left(\mathcal{G}_{[a,i]}\beta \wedge \mathcal{G}_{[i,i]}\alpha\right) \tag{18}$$

3.1 Minimum Computation Length

These regular expressions describe all the satisfying computations of an mLTL formula of length complen(α), which is defined recursively below:

$$\text{complen}(p_k) = \text{complen}(\neg p_k) = 1 \tag{19}$$

$$\text{complen}(\alpha \wedge \beta) = \text{complen}(\alpha \vee \beta) = \max(\text{complen}(\alpha), \text{complen}(\beta)) \tag{20}$$

$$\text{complen}(\mathcal{G}_{[a,b]}\alpha) = \text{complen}(\mathcal{F}_{[a,b]}\alpha) = b + \text{complen}(\alpha) \tag{21}$$

$$\text{complen}(\alpha\mathcal{U}_{[a,b]}\beta) = \text{complen}(\alpha\mathcal{R}_{[a,b]}\beta) = b + \max(\text{complen}(\alpha) - 1, \text{complen}(\beta)) \tag{22}$$

Intuitively, complen(α) is the minimum computation length required to ensure that none of the intervals in α are out of bounds. Our minimum computation length for Until and Release are slight optimizations of what was previously considered the minimum computation length in the literature. In [6],

$$\text{complen}(\alpha\mathcal{U}_{[a,b]}\beta) = \text{complen}(\alpha\mathcal{R}_{[a,b]}\beta) = b + \max(\text{complen}(\alpha), \text{complen}(\beta))$$

whereas our minimum computation length for Until and Release is

$$\text{complen}(\alpha\mathcal{U}_{[a,b]}\beta) = \text{complen}(\alpha\mathcal{R}_{[a,b]}\beta) = b + \max(\text{complen}(\alpha) - \mathbf{1}, \text{complen}(\beta)).$$

We provide the proof for the minimum computation length of Until and Release below.

Theorem 1 (*Minimum Computation Length of Until and Release*)**.** Let $0 \leq a \leq b \in \mathbb{N}$ and let α, β be well-formed mLTL formulas in NNF. The minimum computation length of Until and Release is given by $\text{complen}(\alpha\mathcal{U}_{[a,b]}\beta) = \text{complen}(\alpha\mathcal{R}_{[a,b]}\beta) = b + \max(\text{complen}(\alpha) - 1, \text{complen}(\beta))$.

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, Global, And, and Or.
Recall that $\text{reg}(\alpha\, \mathcal{U}_{[a,b]}\beta) = \bigvee_{i=a}^{b} \text{reg}\left(\mathcal{G}_{[a,i-1]}\alpha \wedge \mathcal{G}_{[i,i]}\beta\right)$. Thus

$$\begin{aligned}
\text{complen}(\alpha\mathcal{U}_{[a,b]}\beta) &= \max_{a\le i\le b}\left(\text{complen}(\mathcal{G}_{[a,i-1]}\alpha \wedge \mathcal{G}_{[i,i]}\beta)\right)\\
&= \max_{a\le i\le b}\left(\max(i-1+\text{complen}(\alpha), i+\text{complen}(\beta))\right)\\
&= \max_{a\le i\le b}\left(i+\max(\text{complen}(\alpha)-1, \text{complen}(\beta))\right)\\
&= b+\max(\text{complen}(\alpha)-1, \text{complen}(\beta)).
\end{aligned}$$

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

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

This completes the proof. □

Intuitively, this optimization of the minimum computation length of Until and Release makes sense:

- At the final time-step for $\alpha\mathcal{U}_{[a,b]}\beta$, if β has not yet been assigned to be true in the computation, it will be assigned true because $\alpha\mathcal{U}_{[a,b]}\beta$ is a strong operator. Semantically, the truth value of α at this time interval doesn't matter; whether α is true or false, the computation for $\alpha\mathcal{U}_{[a,b]}\beta$ is satisfied.
- Likewise, in the case for $\alpha\mathcal{R}_{[a,b]}\beta$, if β is true from the first to the last time-step, the truth value of α at the final time-step doesn't matter - the computation satisfies $\alpha\mathcal{R}_{[a,b]}\beta$ regardless of whether α is true or false.

3.2 Soundness and Completeness

We write $\mathscr{L}(\text{reg}(\alpha))$ to denote the language of $\text{reg}(\alpha)$. That is, the set of computations represented by the regular expression $\text{reg}(\alpha)$.

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

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

reg(p_k)
For $0 \leq k \leq n-1$, 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 $\pi \vDash \varphi \wedge \psi$ iff $\pi \vDash \varphi$ and $\pi \vDash \psi$.
By the inductive hypothesis, $\pi \vDash \varphi$ iff $\pi \in \mathscr{L}(\mathrm{reg}(\varphi))$ and
$\pi \vDash \psi$ iff $\pi \in \mathscr{L}(\mathrm{reg}(\psi))$, so

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

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

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

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

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

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

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

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

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

As argued in the Finally case, the length condition is satisfied.
By the inductive hypothesis, $\pi_i \vDash \psi$ iff $\pi_i \in \mathscr{L}(\mathrm{reg}(\psi))$, or equivalently,

$\pi \in \mathscr{L}(\text{reg}(\mathcal{G}_{[i,i]}\psi))$. Also, $\pi_j \vDash \varphi$ if and only if $\pi_j \in \mathscr{L}(\text{reg}(\varphi))$.
By the argument used in the Global case, we see that $\forall a \leq j < i,\ \pi_j \vDash \varphi$ is equivalent to $\pi \in \mathscr{L}(\text{reg}(\mathcal{G}_{[a,i-1]}\varphi))$.
Thus $\pi \vDash \varphi\ \mathcal{U}_{[a,b]}\psi$ 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,

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

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

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

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

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

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. □

3.3 Nested Until and Release Rewriting Theorem

We prove the following theorem using the regular expressions above. As detailed in section 5, nested binary temporal operators create the worst case time and space complexity for the WEST program. Although the WEST program does not implement this theorem, we would expect an implementation of this theorem to improve runtime. On its own, this theorem can be used to simplify mLTL formulas, making them easier to interpret.

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 $a, b, c \in \mathbb{Z}_{\geq 0}$, $a \leq b$, and α, β be well-formed mLTL formulas in NNF. Then, $\alpha\, \mathcal{B}_{[a,b+c]}\beta = \alpha\, \mathcal{B}_{[a,b]}(\alpha\, \mathcal{B}_{[0,c]}\beta)$. That is, $\alpha\ \mathcal{U}_{[a,b+c]}\beta \equiv \alpha\ \mathcal{U}_{[a,b]}(\alpha\ \mathcal{U}_{[0,c]}\beta)$ and $\alpha\ \mathcal{R}_{[a,b+c]}\beta \equiv \alpha\ \mathcal{R}_{[a,b]}(\alpha\ \mathcal{R}_{[0,c]}\beta)$.

Proof. **Case 1:** $\mathcal{B} = \mathcal{U}$

Let $\gamma = \alpha\ \mathcal{U}_{[0,c]}\beta$. Then, $\alpha\ \mathcal{U}_{[a,b]}(\alpha\ \mathcal{U}_{[0,c]}\beta) = \alpha\ \mathcal{U}_{[a,b]}\gamma$ and $\text{reg}\left(\gamma\right) = \bigvee_{j=0}^{c} \text{reg}\left(\mathcal{G}_{[0,j-1]}\alpha \wedge \mathcal{G}_{[j,j]}\beta\right)$. Thus

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

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

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

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

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

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

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

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

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

This shows that $\alpha\,\mathcal{U}_{[a,b]}(\alpha\,\mathcal{U}_{[0,c]}\beta) \equiv \alpha\,\mathcal{U}_{[a,b+c]}\beta$.

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, and we'll write $\neg\text{reg}(\alpha)$ to denote the compliment of $\text{reg}(\alpha)$. Thus,

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

Global is the dual of Finally, so

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

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

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

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

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

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

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

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

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

Let $k = i + j$. Thus

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

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

4 WEST Code Specifications

The code for the WEST program can be found on its Github repository. The mLTL syntax for our program is given as the context free grammar below. In the alphabet, T and ! refer to the propositional constants 'true' and 'false', respectively. &, v, =, > refer to propositional logic operators 'and', 'or, 'equivalent', and 'implies', respectively. Note that the last comma listed is not a typo and indeed part of the alphabet.

Alphabet:

$$\begin{aligned}\Sigma = \{&\text{'0' , '1' , '2' , '3' , '4' , '5' , '6' , '7' , '8' , '9' , 'T' , '!' , 'p' , '$\sim$' , 'v' , '\&' , '=' , '¿' ,}\\ &\text{'F' , 'G' , 'U', 'R' , '(' , ')' , '[' , ']' , ':' , ','}\}\end{aligned}$$

$$\begin{aligned}\text{Digit} &\rightarrow 0\,|\,1\,|\,2\,|\,3\,|\,4\,|\,5\,|\,6\,|\,7\,|\,8\,|\,9\\ \text{Num} &\rightarrow \text{Digit Num}\,|\,\text{Digit}\\ \text{Interval} &\rightarrow [\ \text{Num}\ :\ \text{Num}\ ]\\ \text{Prop\_cons} &\rightarrow \text{T}\ |\ !\\ \text{Prop\_var} &\rightarrow \text{p Num}\\ \text{Unary\_Prop\_conn} &\rightarrow \sim\\ \text{Binary\_Prop\_conn} &\rightarrow \text{v}\ |\ \&\ |\ =\ |\ >\\ \text{Assoc\_Prop\_conn} &\rightarrow \text{v}\ |\ \&\ |\ =\\ \text{Array\_entry} &\rightarrow \text{Wff , Array\_entry}\ |\ \text{Wff}\\ \text{Unary\_Temp\_conn} &\rightarrow \text{F}\ |\ \text{G}\\ \text{Binary\_Temp\_conn} &\rightarrow \text{U}\ |\ \text{R}\\ \text{Wff} &\rightarrow \text{Prop\_cons}\ |\ \text{Prop\_var}\\ &\quad |\ \text{Unary\_Prop\_conn}\ \ \text{Wff}\\ &\quad |\ \text{Wff}\ \ \text{Binary\_Prop\_conn}\ \ \text{Wff}\\ &\quad |\ (\ \text{Assoc\_Prop\_conn}\ [\ \text{Array\_entry}\ ]\ )\\ &\quad |\ \text{Unary\_Temp\_conn}\ \ \text{Interval}\ \ \text{Wff}\\ &\quad |\ (\ \text{Wff}\ \ \text{Binar\_Temp\_conn}\ \ \text{Interval}\ \ \text{Wff}\ )\end{aligned}$$

In the grammar, associative propositional connectives (*Assoc_Prop_conn*) are included for the ease of the user. Given well-formed formulas $\alpha_1, \alpha_2, \alpha_3$, the following formula in prefix notation $(\vee[\alpha_1, \alpha_2, \alpha_3])$ is parsed into $((\alpha_1 \vee \alpha_2) \vee \alpha_3)$, which is in turn equivalent to $(\alpha_1 \vee \alpha_2 \vee \alpha_3)$.

Remark 1. For two propositional variables, the associative equivalence operator functions identically to the binary propositional connective equivalence operator. For formulas with 3 or more propositional variables, however , the associative equivalence operator does not mean that each propositional variable in the list is equivalent. Instead, it means that the equivalence of the first two propositional variables in the list is equivalent to the next propositional variable in the list, and the truth value for this expression is equivalent to the next propositional variable, and so on. For example, =[p1,p2,p3] is equivalent to (p1=p2)=p3 rather than p1=p2=p3.

In Appendix III, we provide the full pseudocode of the WEST program. Below we summarize only the important points to provide analysis of time and space complexity in the next section. The regular expression $R = \{R_1|R_2|...|R_k\}$, where for all $1 \leq i \leq k$, $R_i \in \{\text{'0'}, \text{'1'}, \text{'S'}, \text{','}\}^*$, is represented as a vector $R = R_1, R_2, ..., R_k$.

In order to compute mLTL operations, we define functions for handling union and intersection of such vectors. The *join* function performs simple concatenation of two vectors, while the *set_intersect* function calculates pairwise intersection of strings from two vectors. The intersection of two strings is performed by *bit_wise_and*, which computes intersection character by character. For example, $\textit{bit\_wise\_and}(1s, 0s)$ is the empty string while $\textit{bit\_wise\_and}(s11, 0s1) = 011$.

The main focus is on the recursive function *reg*, which is based on the structure of the grammar. It computes the propositional logic operators $>$ and $=$ through formula rewriting in terms of And and Or. The four temporal operators in the grammar are computed according to the definitions of their regexes (See section 3).

If the user would like to see the satisfying computations for all the subformulas in their original formula, *reg* contains a boolean to enable this functionality.

It is often the case that a vector of regular expression strings contains many redundancies, and can be simplified to a smaller vector of regular expressions that improves space complexity and drastically enhances user readability. As such, the *simplify* function performs this by iterating through pairs of strings from the vector to search for valid simplifications. If a pair of strings(w, v) can be written in the form $w = w_1 c_w w_2$ and $v = v_1 c_v v_2$ such that $w_1 = v_1$ and $w_2 = v_2$, then they are combined into one string $w_1(c_w|c_v)w_2$.

5 Complexity

5.1 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 $\delta \in \mathbb{N}$. We provide a summary of the complexity of each of the helper functions described for *reg*. For any function $f(\alpha)$ taking a string argument α, we use $|\alpha|$ to denote the number of characters in α and $S(f(\alpha))$ to denote the space complexity of f in terms of the number of characters in the output.

If α is a propositional constant or variable:
It's easy to see that $S(\textit{reg}(!)) = 0$ since only the empty vector is returned.
By definition $\textit{reg}(T) = S^n$, so we have that $S(\textit{reg}(T)) = O(n)$. Similar to $\alpha = T$, $\textit{reg}(p_k)$ and $\textit{reg}(\sim p_k)$ both return strings of the same length and thus we also have $S(\textit{reg}(p_k)) = S(\textit{reg}(\sim p_k)) = S(\textit{reg}(T)) = n$.

If α is "$\alpha_1 \vee \alpha_2$" :
For this case in our *reg* function, we return $\textit{join}(\textit{reg}(\alpha_1), \textit{reg}(\alpha_2))$
Thus, our space complexity is:

$$S(\textit{reg}(\alpha_1 \vee \alpha_2)) = S(\textit{reg}(\alpha_1)) + S(\textit{reg}(\alpha_2)).$$

If α is "α_1 & α_2" :

For this case in our *reg* function, we return $\mathit{set\_intersect}(\mathit{reg}(\alpha_1), \mathit{reg}(\alpha_2), n)$
In the worst case, $\mathit{set\_intersect}(A, B, n)$ will return a vector of size $S(A) \cdot S(B)$.
Thus, our space complexity is:

$$S(\mathit{reg}(\alpha_1 \ \& \ \alpha_2)) = S(\mathit{reg}(\alpha_1)) \cdot S(\mathit{reg}(\alpha_2)).$$

If α **is** "$\alpha_1 > \alpha_2$" **:**
For this case in our *reg* function, we first rewrite $\alpha_1 > \alpha_2$ to the equivalent formula:

$$\mathit{equiv\_formula} = \text{“(”} + \mathit{wff\_to\_nnf}(\sim\alpha_1) + \text{“} \vee \text{”} + \alpha_2 + \text{“)”}.$$

Then we return $\mathit{reg}(\mathit{equiv\_formula})$. So for space complexity, we have:

$$S(\mathit{reg}(\alpha_1 > \alpha_2)) = S(\mathit{reg}(\mathit{wff\_to\_nnf}(\sim\alpha_1)) + S(\mathit{reg}(\alpha_2)).$$

If α **is** "$\alpha_1 = \alpha_2$" **:**
For this case in our *reg* function, we first rewrite $\alpha_1 = \alpha_2$ to the equivalent formula:

$$\mathit{equiv\_formula} = \text{“((”}+\alpha_1+\text{“\&”}+\alpha_2+\text{“)}\vee\text{(”}+\mathit{wff\_to\_nnf}(\sim\alpha_1)+\text{“\&”}+\mathit{wff\_to\_nnf}(\sim\alpha_2)+\text{“))”}.$$

Then we return $\mathit{reg}(\mathit{equiv\_formula})$. So for space complexity, we have:

$$S(\mathit{reg}(\alpha_1 = \alpha_2)) = S(\mathit{reg}(\alpha_1)) \cdot S(\mathit{reg}(\alpha_2)) + S(\mathit{reg}(\mathit{wff\_to\_nnf}(\sim\alpha_1)) \cdot S(\mathit{reg}(\mathit{wff\_to\_nnf}(\sim\alpha_2)).$$

If α **is** "$\vee$ [α_1, .., α_k]" **:**
For this case in our *reg* function, we first rewrite $\vee[\alpha_1, .., \alpha_k]$ to the equivalent formula:

$$\mathit{equiv\_formula} = \text{“(...(}\alpha_1 \vee \alpha_2) \vee ...\alpha_k\text{)”}.$$

Using the space complexity of $\vee$, we have that:

$$S(\mathit{reg}(\vee[\alpha_1, .., \alpha_k])) = \sum_{i=1}^{k} S(\mathit{reg}(\alpha_i)).$$

If α **is** "& [α_1, .., α_k]" **:**
For this case in our *reg* function, we first rewrite $\&[\alpha_1, .., \alpha_k]$ to the equivalent formula:

$$\mathit{equiv\_formula} = \text{“(...(}\alpha_1 \& \alpha_2)\&...\alpha_k\text{)”}.$$

Using the space complexity of &, we have that:

$$S(\mathit{reg}(\&[\alpha_1, .., \alpha_k])) = \prod_{i=1}^{k} S(\mathit{reg}(\alpha_i)).$$

If α **is** "$= [\alpha_1$, .., $\alpha_k]$" **:**
For this case in our *reg* function, we first rewrite $= [\alpha_1, .., \alpha_k]$ to the equivalent formula:

$$\mathit{equiv\_formula} = \text{“(...(}\alpha_1 = \alpha_2) = ...\alpha_k\text{)”}.$$

Based on the space complexities of & and $\vee$, this gives:

$$S(\mathit{reg}(= [\alpha_1, .., \alpha_k])) = \prod_{i=1}^{k} S(\mathit{reg}(\alpha_i)) + \prod_{i=1}^{k} S(\mathit{reg}(\mathit{wff\_to\_nnf}(\sim\alpha_i))).$$

For the following cases, we use these two bounds:

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

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

If α is "$\mathcal{G}_{[a,b]}\alpha_1$"**:**

Recall that $\mathit{reg}(\mathcal{G}_{[a,b]}\alpha) = \bigwedge_{i=a}^{b}(S^n,)^i \mathit{reg}(\alpha)(,S^n)^{b-i}$.
As shown in the *Assoc_Prop_Conn* case, if α is "$\&[\alpha_1, ..., \alpha_k]$", then $\mathrm{S}(\mathit{reg}(\&[\alpha_1, .., \alpha_k])) = \prod_{i=1}^{k} \mathrm{S}(\mathit{reg}(\alpha_i))$. We have that:

$$\mathrm{S}(\mathit{reg}(\alpha)) = \prod_{i=a}^{b}(n+1)i \cdot \mathrm{S}(\mathit{reg}(\alpha_1)) \leq \mathcal{C}_G \cdot \mathrm{S}(\mathit{reg}(\alpha_1)^{\delta}$$

In the calculation, $(n+1)i$ counts the concatenation of the padded component of the computation in the worse case when we have a vector of only length 1 strings.

If α **is "$\mathcal{F}_{[a,b]}\alpha_1$":**
Recall that $\mathit{reg}(\mathcal{F}_{[a,b]}\alpha) = \bigvee_{i=a}^{b}(S^n,)^i \mathit{reg}(\alpha)(,S^n)^{b-i}$.
As shown in the *Assoc_Prop_Conn* case, if α is "$\vee[\alpha_1, ..., \alpha_k]$", then $S(\mathit{reg}(\vee[\alpha_1, .., \alpha_k])) = \sum_{i=1}^{k} S(\mathit{reg}(\alpha_i))$. Thus with similar calculation to $\mathcal{G}$,

$$\mathrm{S}(\mathit{reg}(\alpha)) = \sum_{i=a}^{b}(n+1)i \cdot \mathrm{S}(\mathit{reg}(\alpha_1)) \leq \mathcal{C}_F \cdot \mathrm{S}(\mathit{reg}(\alpha_1)$$

If α **= "$\alpha_1\mathcal{U}_{[a,b]}\alpha_2$":**
By our previous definitions, $\mathit{reg}(\alpha_1\ \mathcal{U}_{[a,b]}\alpha_2) = \bigvee_{i=a}^{b} \mathit{reg}\left(\mathcal{G}_{[a,i-1]}\alpha_1 \wedge \mathcal{G}_{[i,i]}\alpha_2\right)$.
We can bound $S(\mathit{reg}(\mathcal{G}_{[i,i]}\alpha_2))$ by $(n+1) \cdot i \cdot S(\mathit{reg}(\alpha_2))$ because the operation is equivalent to simply prepending $(S^n,)^i$. Thus using our previous results for the $\mathcal{G}$, & and $\vee$ operators, we have that:

$$\begin{aligned}
S(\mathit{reg}(\alpha)) &= \sum_{i=a}^{b}[((n+1)(i-1)+1) \cdot S(\mathit{reg}(\alpha_1))]^{\delta}[(n+1) \cdot i \cdot S(\mathit{reg}(\alpha_2))] \\
&< \delta[((n+1)(b-1)+1) \cdot S(\mathit{reg}(\alpha_1))]^{\delta}[(n+1) \cdot b \cdot S(\mathit{reg}(\alpha_2))] \\
&< \delta((n+1)(b-1)+1)^{\delta} \cdot (n+1) \cdot d \cdot S(\mathit{reg}(\alpha_1))^{\delta} \cdot S(\mathit{reg}(\alpha_2)) \\
&\leq \mathcal{C}_U \cdot S(\mathit{reg}(\alpha_1))^{\delta} \cdot S(\mathit{reg}(\alpha_2))
\end{aligned}$$

where $\mathcal{C}_U = \delta((n+1)(d-1)+1)^{\delta} \cdot (n+1) \cdot d$.
If α **= "$\alpha_1\mathcal{R}_{[a,b]}\alpha_2$":**
By our previous definitions, $\mathit{reg}(\alpha_1\mathcal{R}_{[a,b]}\alpha_2) = \mathit{reg}\left(\mathcal{G}_{[a,b]}\alpha_2\right) \vee \bigvee_{i=a}^{b-1} \mathit{reg}\left(\mathcal{G}_{[a,i]}\alpha_2 \wedge \mathcal{G}_{[i,i]}\alpha_1\right)$.
Thus, a similar argument to the $\mathcal{U}$ case gives us that:

$$\begin{aligned}
S(\mathit{reg}(\alpha)) &< \mathcal{C}_G \cdot S(\mathit{reg}(\alpha_1))^{\delta} + \sum_{i=a}^{b-1}[\mathcal{C}_G \cdot S(\mathit{reg}(\alpha_1))^{\delta} \cdot (n+1) \cdot i \cdot S(\mathit{reg}(\alpha_2))] \\
&< \mathcal{C}_G \cdot S(\mathit{reg}(\alpha_1))^{\delta} + \delta[\mathcal{C}_G \cdot S(\mathit{reg}(\alpha_1))^{\delta} \cdot (n+1)(b-1) \cdot S(\mathit{reg}(\alpha_2))] \\
&= \mathcal{C}_G \cdot S(\mathit{reg}(\alpha_1))^{\delta}[1 + \delta(n+1)(b-1)S(\mathit{reg}(\alpha_2))] \\
&\leq \mathcal{C}_G \cdot S(\mathit{reg}(\alpha_1))^{\delta}(\delta(n+1)d \cdot S(\mathit{reg}(\alpha_2))) \\
&= \mathcal{C}_R \cdot S(\mathit{reg}(\alpha_1))^{\delta} \cdot S(\mathit{reg}(\alpha_2))
\end{aligned}$$

where $\mathcal{C}_R = \mathcal{C}_G \cdot \delta(n+1)d = \mathrm{C}_G \cdot \mathcal{C}_F$.

Theorem 4 (*Space Complexity*)**.** Given a well formed mLTL formula α, $\mathit{reg}(\alpha)$ has worst-case space complexity that is $O(\mathcal{C}_U^{\delta^{\ell}} \cdot (\ell+1)^{\delta^{\ell+1}})$, where ℓ is the number of logical connectives $(\wedge, \vee, \mathcal{F}, \mathcal{G}, \mathcal{R}, \mathcal{U})$ in α.

Proof. To analyze worst-case complexity, it's clear from the analysis above that $\mathcal{U}$ and $\mathcal{R}$ give the largest complexity. From the constants in their respective bounds, we can show that $\mathcal{C}_U = ((n+1)d-n)^{\delta} \cdot \mathcal{C}_R > \mathcal{C}_R$, and so we'll analyze only repeated nesting of the operator $\mathcal{U}$.

However, notice that the structure of the parse tree is important. For example, formulas similar to $(a\mathcal{U}b)\mathcal{U}(c\mathcal{U}d)$ 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 $(((p_0\mathcal{U}p_1)\mathcal{U}p_2)\mathcal{U}p_3)$, then the maximum depth of recursion is $O(\ell)$. Thus we'll focus on the formula

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

where there are ℓ logical connectives $\mathcal{U}$ and $n = \ell + 1$ propositional variables.

We'll derive the complexity of $S(\mathit{reg}(\alpha))$ by defining the recursive sequence $\{s_k\}_{k=1}^{\ell}$ such that $s_1 = S(\mathit{reg}(p_0\mathcal{U}_{[a_1,b_1]}p_1))$, and $s_{k+1} = \mathcal{C}_U(s_k)^\delta \cdot S(\mathit{reg}(p_{k+1}))$. The recurrence relation captures an extra nesting of the $\mathcal{U}$ operator, based on the complexity of $\mathcal{U}$ defined above. We calculate $S(p_m) = n = \ell+1$ for all m such that $0 \le m \le \ell$, thus $s_1 = \mathcal{C}_U(\ell+1)^{\delta+1}$ and $s_{k+1} = \mathcal{C}_U(s_k)^\delta \cdot (\ell+1)$.

The explicit formula is given by $s_k = \mathcal{C}_U^{\sum_{i=0}^{k-1}\delta^i} \cdot (\ell+1)^{\sum_{i=0}^{k}\delta^i}$. It's easy to check that the base case $k = 1$ holds, and we prove the claim by induction:

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

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

□

For time complexity, through a similar analysis, we've found that the time-complexity of all of the above functions is the same as their space complexity. In particular, the time complexities of $\alpha = p_k, \sim p_k, T, !$, as well as that of set intersection, *join*, and the operator $\mathcal{C}_U$ match their respective space complexities. Thus the worst case time complexity is $O(\mathcal{C}_U^{\delta^\ell} \cdot (\ell+1)^{\delta^{\ell+1}})$ as well.

One feature within our implementation of *reg* is to apply the *simplify* function to all intermediary vector calculations in order to decrease the output size. In the worst case if no simplification occurs, space complexity remains unchanged, but time complexity of *simplify* is cubic in input size. In practice however, both time and space complexities are much more optimistic than worst-case estimates.

5.2 Experimental Benchmarking

Along with theoretical space and time complexity, we provide some experimental benchmarking of these complexities using randomly generated mLTL formulas. This was done on an Intel(R) Core(TM) i7-4770S CPU at 3.10GHz with 32gb RAM. For each simulation, we 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 milliseconds 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.

5.2.1 Simulation 1

For the first simulation, we consider 2 iterations, 5 propositional variables, 10 for DELTA, and 10 for INTERVAL_MAX. Below are some examples of generated formulas:

- $(p_4 \wedge p_0)\mathcal{U}_{[2,3]}(\neg p_1)$.
- $(\mathcal{G}_{[3,5]}p_3)\mathcal{R}_{[7,8]}(p_3 \rightarrow p_0)$.
- $\mathcal{F}_{[1,6]}(p_4 = p_3)$.

We obtain the following plots:

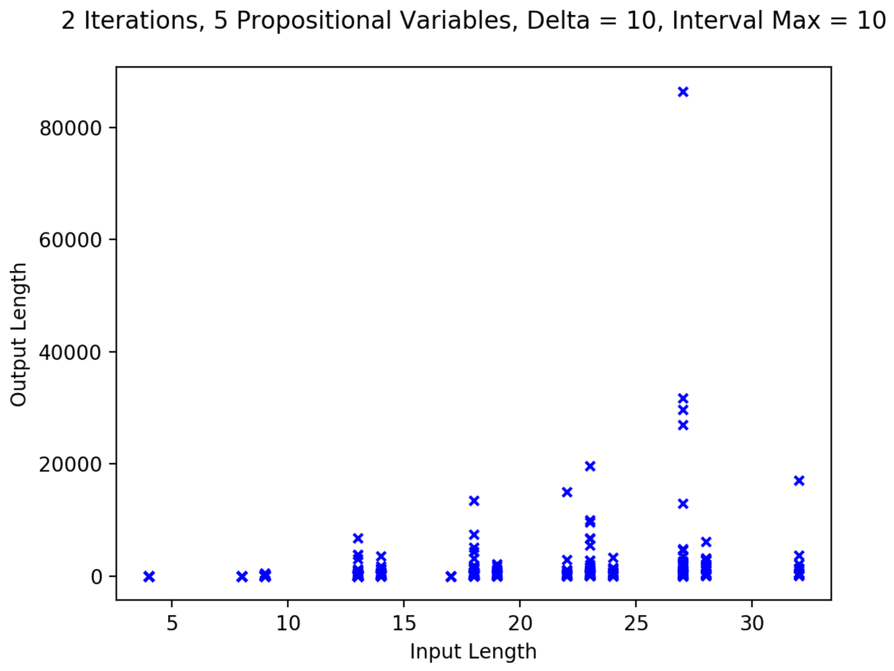


Figure 1: Input length in number of characters vs output length in number of characters for simulation 1. Input length counts all characters, including parentheses, of the inputted formulas. Output length counts all the characters outputted, including commas.

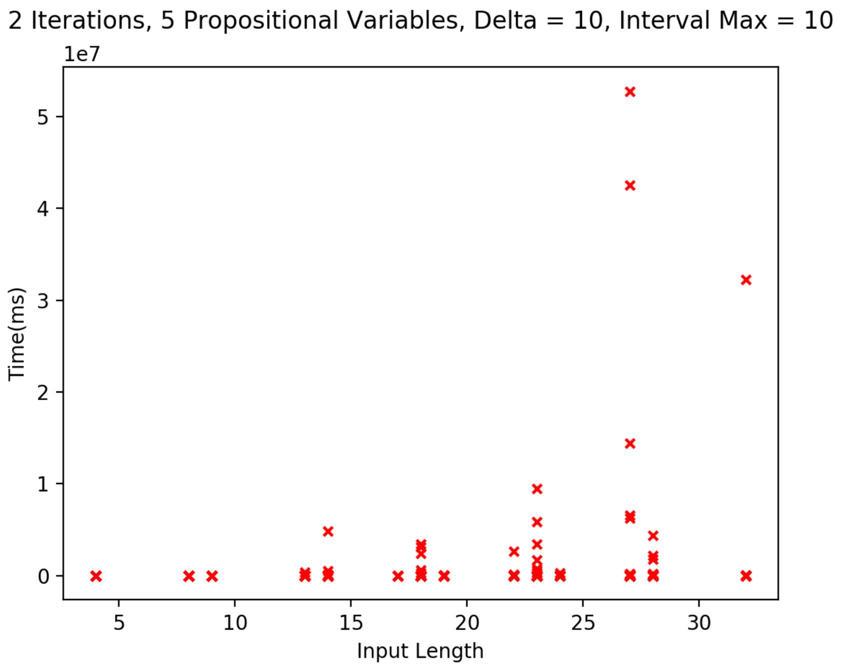


Figure 2: Input length in number of characters vs run time in milliseconds for simulation 1. Input length counts all characters, including parentheses, of the inputted formulas.

The extreme outlier seen in the output length plot corresponds to the formula $(p_0 = p_1)\mathcal{U}_{[2,9]}(p_1\mathcal{U}_{[7,9]}p_3)$. The three extreme outliers seen in the time plot correspond to the formulas

- $(p_0 = p_1)\mathcal{U}_{[2,9]}(p_1\mathcal{U}_{[7,9]}p_3)$ (same formula as the complexity outlier).
- $(p_4 \rightarrow p_2)\mathcal{R}_{[1,8]}(p_3\mathcal{U}_{[3,4]}p_0)$.
- $(p_3\mathcal{R}_{[2,7]}p_4)\mathcal{R}_{[1,9]}(p_4\mathcal{R}_{[4,9]}p_3)$.

5.2.2 Simulation 2

For the second simulation, we consider 1 iteration, 10 propositional variables, 20 for DELTA, and 20 for INTERVAL_MAX. Below are some examples of generated formulas:

- $\mathcal{F}_{[10,17]}p_4$.
- $\neg p_6$.
- $p_6\mathcal{U}_{[6,14]}p_8$.

We obtain the following plots:

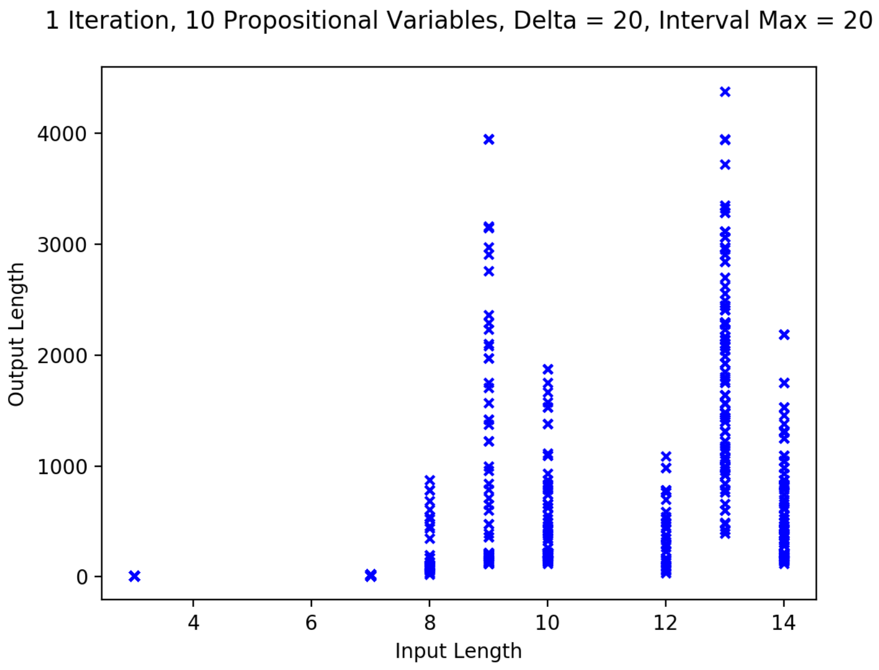


Figure 3: Input length in number of characters vs output length in number of characters for simulation 2. Input length counts all characters, including parentheses, of the inputted formulas. Output length counts all the characters outputted, including commas.

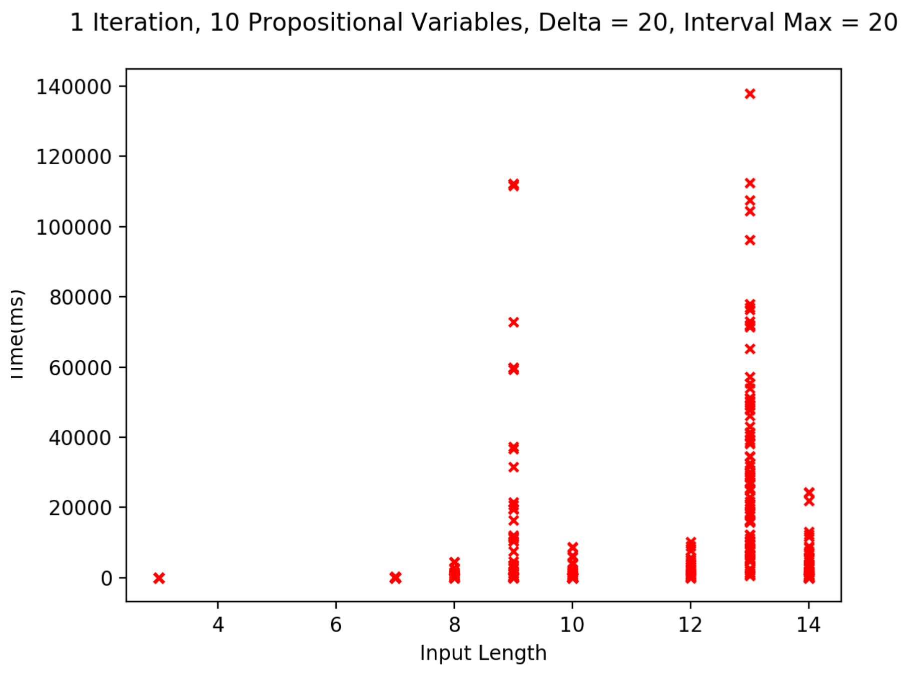


Figure 4: Input length in number of characters vs run time in milliseconds for simulation 2. Input length counts all characters, including parentheses, of the inputted formulas.

5.2.3 Simulation 3

For the third simulation, we consider 2 iterations, 10 propositional variables, 5 for DELTA, and 10 for INTERVAL_MAX. Below are some examples of generated formulas:

- $(p_5 \wedge p_0) = (p_8 \mathcal{U}_{[3,6]} p_8)$.
- $(\mathcal{G}_{[6,7]} p_4) \wedge (\mathcal{F}_{[5,7]} p_1)$.
- $\neg(p_0 \mathcal{U}_{[2,5]} p_7)$.

We obtain the following plots:

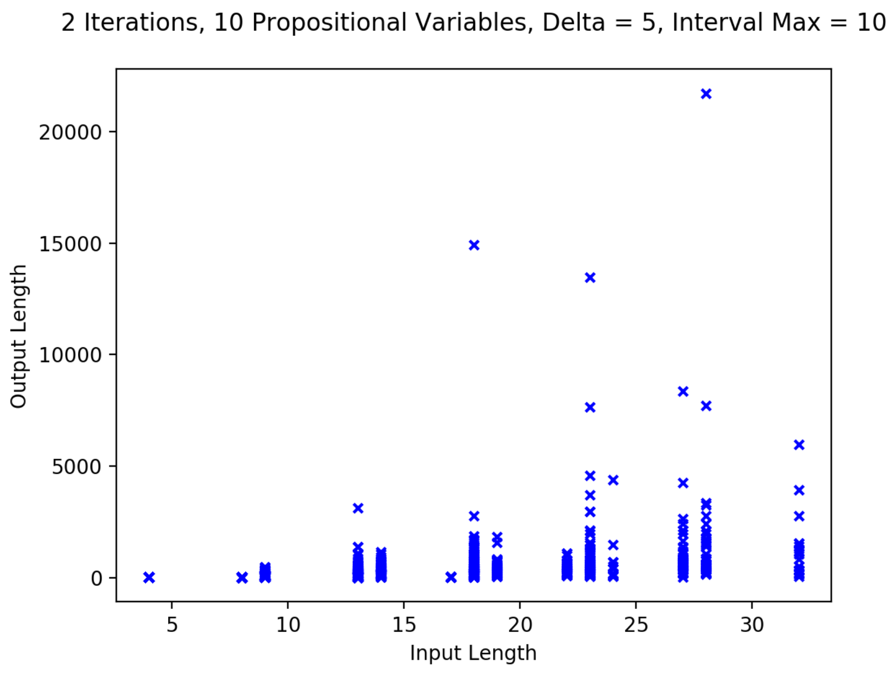


Figure 5: Input length in number of characters vs run time in milliseconds for simulation 3. Input length counts all characters, including parentheses, of the inputted formulas.

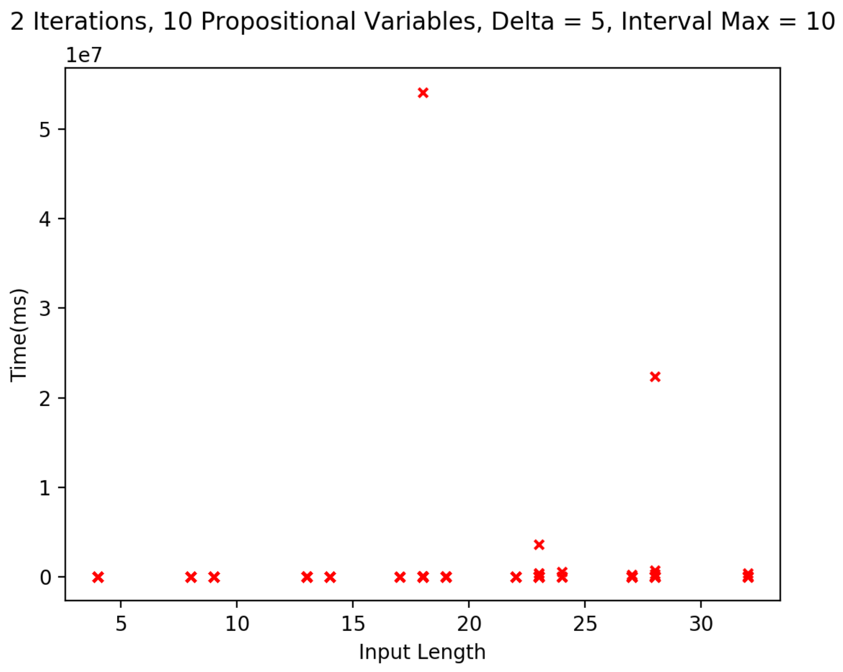


Figure 6: Input length in number of characters vs run time in milliseconds for simulation 3. Input length counts all characters, including parentheses, of the inputted formulas.

The three outliers in the output length plot are the formulas

- $(p_7\mathcal{U}_{[5,7]}p_9)\mathcal{U}_{[3,7]}(\mathcal{F}_{[1,4]}p_4)$.

- $\mathcal{G}_{[3,7]}(p_9 \mathcal{U}_{[0,4]} p_0)$.
- $(\neg p_9)\mathcal{R}_{[5,9]}(p_5 \mathcal{U}_{[2,4]} p_3)$.

The former two are also the time complexity outliers.

5.2.4 Simulation 4

For the fourth simulation, we consider 1 iteration, 5 propositional variables, 10 for DELTA, and 10 for INTERVAL_MAX. Below are some examples of generated formulas:

- $\neg p_1$.
- $p_1 \mathcal{R}_{[6,7]} p_3$.
- $\mathcal{F}_{[5,5]} p_3$.

We obtain the following plots:

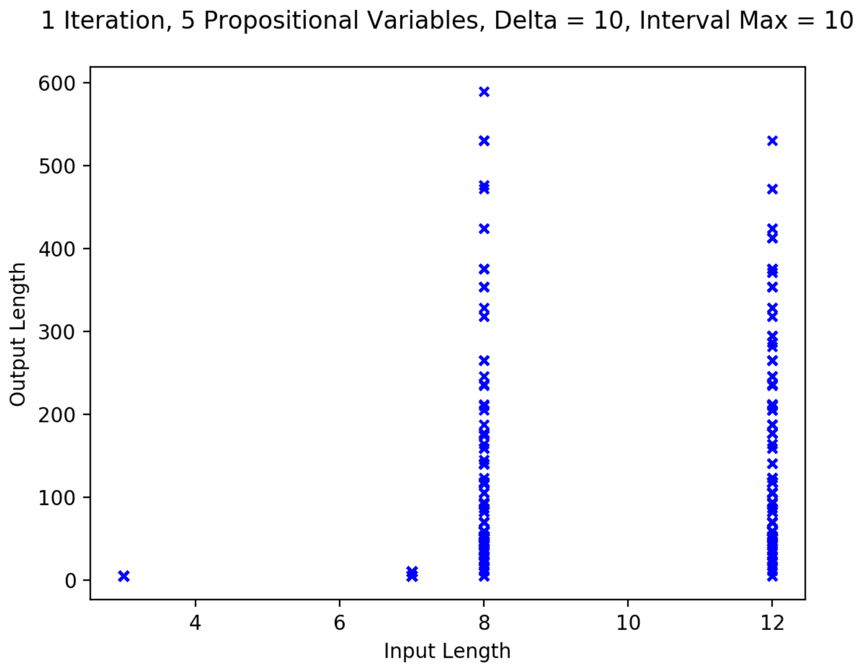


Figure 7: Input length in number of characters vs run time in milliseconds for simulation 4. Input length counts all characters, including parentheses, of the inputted formulas.

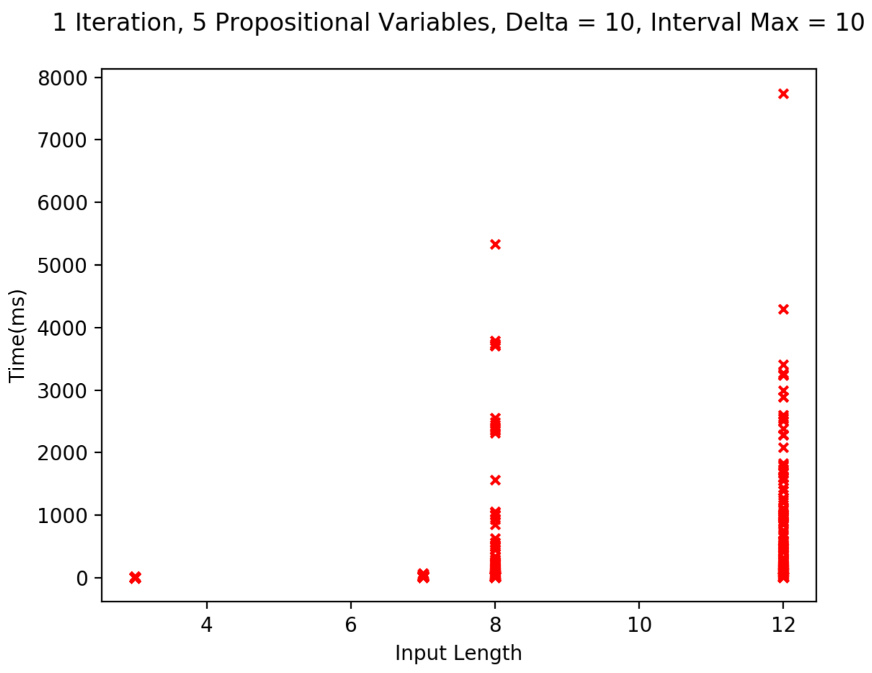


Figure 8: Input length in number of characters vs run time in milliseconds for simulation 4. Input length counts all characters, including parentheses, of the inputted formulas.

What we observe from the simulations is that the worst case complexity, both for space and time, is relatively sparse. 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 [5], [6], [1], and [3], nested binary temporal operators do not appear in any specifications, and thus are likely rare in practical applications.

6 Verification of Correctness

The correctness of an abstract algorithm one can describe on pencil and paper differs from the correctness of an actual implementation of the algorithm (the program). On one hand, correctness of an algorithm is assured by mathematical analysis of the algorithm on all possible inputs and showing that it behaves as desired. However, assuring correctness of the program is a difficult task. One naive approach is to test every possible input up to a certain size, and verify the output or behavior of the program. This often times involves a test suite that is too large to be computationally feasible, and the test suite may be overtly redundant. 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 $0 \leq t \leq 99$; verification of a few should give sufficient confidence of correctness of the program for any reasonable party.

We approach this problem more intelligently. In the following sections, we prove the correctness of our algorithm and also test our implementation with a test suite that explores all control paths (up to a certain depth) of the algorithm. A control path can be thought of as a sequence of lines of code that are executed in one calculation of an input formula. Differing structures of formulas will lead to different sequences of lines of code to be executed, and a test suite to explore all such paths provides high confidence in the correctness of the WEST program.

6.1 Theoretical Correctness of the WEST Algorithm

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* takes as input a mLTL formula of the appropriate form, and returns the regular expression as previously defined in section 3.

Join takes as input two vectors of computation strings R and T, and simply returns a vector containing all strings in R and T, which is equivalent to $R \cup T$.

Set_intersect takes as input two vectors of computation strings R and T, where R represents $r_1|...|r_a$ and T represents $t_1|...|t_b$ 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 $\mathscr{L}(set\_intersect(R,T)) = \mathscr{L}(R) \cap \mathscr{L}(T)$. Since a vector of regular expressions is synonymous to the collective Or of each of them, we have that

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

Set_intersect computes the union of $bit\_wise\_and(r_i, t_j)$ over all such pairs, and so it suffices to show $\mathscr{L}(bit\_wise\_and(r_i, t_j)) = \mathscr{L}(r_i) \cap \mathscr{L}(t_j)$. 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. $Bit\_wise\_and(r_i, t_j)$ compares r_i and t_j character by character and computes their intersection, which is defined naturally: $0 \cap 1 = \emptyset$, $0 \cap S = 0$, $1 \cap S = 1$, $0 \cap 0 = 0$, $1 \cap 1 = 1$, 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 $\mathscr{L}(bit\_wise\_and(r_i, t_j)) = \mathscr{L}(r_i) \cap \mathscr{L}(t_j)$ 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 $d \geq 0$. Let γ be a mLTL formula in negation normal form of depth $d+1$. γ must be of the form $\mathcal{G}_{[a,b]}\alpha$, $\mathcal{F}_{[a,b]}\alpha$, $\alpha\mathcal{U}_{[a,b]}\beta$, or $\alpha\mathcal{R}_{[a,b]}\beta$, such that α and β are of depth at most d, and a, b are positive integers. The proof for all four 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 $\gamma = \alpha\mathcal{U}_{[a,b]}\beta$ as an example. In this case, *reg* will call the helper function *reg_U* and pass in the parameters $reg\_alpha = reg(\alpha)$ and $reg\_beta = reg(\beta)$, which, by the induction hypothesis, are correctly and recursively computed. The regular expression for the Until operator may be rewritten as follows:

$$\begin{aligned}
\text{reg}(\alpha\, \mathcal{U}_{[a,b]}\beta) &= \bigvee_{i=a}^{b} \text{reg}\left(\mathcal{G}_{[a,i-1]}\alpha \wedge \mathcal{G}_{[i,i]}\beta\right) \\
&= \text{reg}\left(\mathcal{G}_{[a,a-1]}\alpha \wedge \mathcal{G}_{[a,a]}\beta\right) \vee \bigvee_{i=a+1}^{b} \text{reg}\left(\mathcal{G}_{[a,i-1]}\alpha \wedge \mathcal{G}_{[i,i]}\beta\right) \\
&= \text{reg}\left(\mathcal{G}_{[a,a]}\beta\right) \vee \bigvee_{i=a}^{b-1} \text{reg}\left(\mathcal{G}_{[a,i]}\alpha \wedge \mathcal{G}_{[i+1,i+1]}\beta\right)
\end{aligned}$$

The last line follows from the semantics of the Global operator, which defines out of bound time step behavior to be vacuously true. In the pseudocode for *reg_U* (III), the variable *comp* is initialized to $(`1S"^n + ``,")^a$ pre-concatenated to *reg_beta*, and is the regular expression for $\mathcal{G}_[a,a]\beta$. Next, the Or block from $i = a+1$ to $b-1$ is iteratively computed using *set_intersect* for the inner expression, and added to *comp* with the *join* function, which are both proven correct above. This shows correctness of *reg_U*. Continue 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.

6.2 Correctness of WEST

Negative testing is the process of checking that a program is able to handle unexpected inputs without crashing; fuzz testing (also know as fuzzing) is a form of negative testing. The earliest use of fuzzing by Miller [8] was to discover bugs in Unix utilities by randomly generating command line inputs, and monitoring for unexpected crashes or security. This style of black box testing assumes no prior knowledge of the target program, and is centered on generating a large quantity of inputs. Miller describes his approach in [8] as follows:

"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."

Intelligent fuzzing (sometimes referred to as whitebox fuzzing) is an alternate approach to fuzz testing 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 state diagram of our algorithm, represented as a directed graph. The edges of the graph capture control flow of our algorithm, and vertices represent the remaining blocks of the code without branching statements.
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 naive brute force generator of satisfying computations on the test suite and verify that both outputs match for all test cases.

6.2.1 State Diagram Construction

We choose to represent the states space of 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 is executed in the recursive routine, *reg*, which calls the 8 different subroutines discussed earlier, as shown in the graph below:

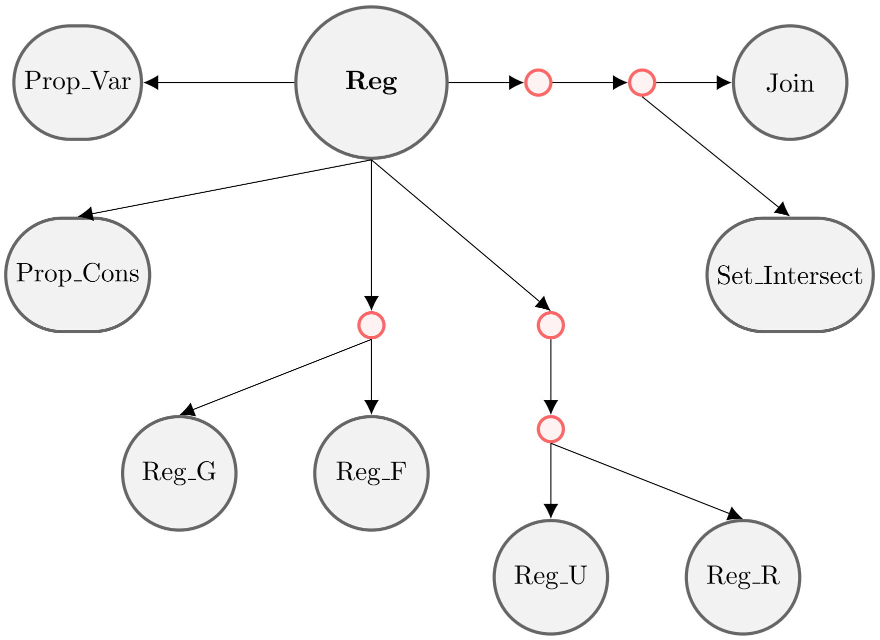


Figure 9: Abstracted graph of the *reg* main routine (refer to Appendix IV for expanded graph and graphs of helper functions). Red nodes signal recursive calls to *reg* on sub formulas of the input formula.

In order to construct the intelligent fuzzing test suite, we make the 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 [10], some of which are provably impossible to construct a test input to explore.

Additionally, the astute reader may observe that analysis of the $>$, $=$, and *assoc_prop_conn* operators are not included in the state diagram. These cases are handled in the algorithm through only formula rewriting to the core operators. This is done through simple string manipulation, which is relatively simple to guarantee and test correctness of through unit testing.

6.2.2 Creating the Test Suite

We now give the procedure of how the intelligent fuzzing test suite is generated. Firstly, we count the number of formulas $\phi(d)$ 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 $\phi(0) = 2$. For $d \geq 1$, we recursively calculate $\phi(d+1) = 2\phi(d) + 4\phi(d)^2$; paths to *reg_G* and *reg_F* is counted in the linear term, and paths *reg_U*, *reg_R*, *set_intersect*, and *join* is counted by the quadratic term. This gives $\phi(1) = 20$, $\phi(2) = 1640$, and $\phi(3) = 10761680$. $d = 3$ is computationally infeasible, and $d = 1$ doesn't give us assurance about operators interacting with each other through nesting. Thus $d = 2$ is a happy medium.

The full test suite is generated in a similar recursive manner. Firstly, the $d = 0$ test suite comprises of two formulas: a propositional variable or its negation, and a propositional constant. Then for any $d \geq 1$, the test suite is constructed by iterating 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, each of the propositional variables or their negation and propositional constants are randomly generated (full pseudocode description in Appendix III).

6.2.3 Verifying against Naive Brute Force

A relatively straight forward approach to generating the set of all satisfying computations of a mLTL formula α over n variables, such that $m = \text{complen}(\alpha)$, is to iterate over all $2^{m \cdot n}$ possible computations, which counts all possible length $m \cdot n$ bit strings. The main work is done by an interpreter function that takes arguments of computation π and mLTL formula α and determines if $\pi \vDash \alpha$ purely based on semantics that define mLTL. Every first-order quantifier is easily translated to loops, and checking for satisfying conditions of the suffix of a computation naturally lends itself to recursion. The full implementation details can be found on the GitHub repository.

As expected for a brute force implementation, extensive run time poses a major limitation; the previously mentioned infeasibility of not abstracting away subroutines and running test suites for deeper depths is primarily caused by this stage of verification. 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, the number of propositional variables was fixed 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. One thing to note is 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. For instance, the regular expression "$1SS0SS1$" yields $2^{\#\text{ of '}S\text{'s}} = 2^4 = 16$ computations after expansion.

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.

7 Regular Expression Simplification Theorem

As a final result of our paper, we provide a regular expression simplification theorem. This theorem describes a form of a set of computations that are guaranteed to simplify down to a string of all 'S's. Although not implemented in the WEST program, this theorem may help users identify tautologies, as the WEST program does not always output a single string of 'S's when a formula holds true for every computation. We first define some vocabulary.

Definition 6 (*Arbitrary Computation*)**.** We say a regular expression composed entirely of 'S's and commas is an *arbitrary computation.* For the purposes of the following theorem, we remove all commas from computations.

Definition 7 (*Fixed Truth Value*)**.** We say '0' or a '1' in a regular expression is a *fixed truth value.*

Definition 8 (*Matrix*)**.** 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 5 (*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's, then the union of this set of regular expressions can be simplified to S^n, the arbitrary computation of length n.*

Example:

$$\begin{pmatrix} 1 & S & S & S \\ S & 1 & S & S \\ S & S & 1 & S \\ S & S & S & 1 \\ 0 & 0 & 0 & 0 \end{pmatrix} = SSSS.$$

See Appendix II for the proof of the theorem.

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

7.1 Implementation Challenges of the Regular Expression Simplification Theorem

Example 5. Consider the mLTL formula: $T\ \mathcal{U}_{[0,3]}p_0 = \mathcal{F}_{[0,3]}p_0$

Our program outputs the following union of regular expressions to represent this formula:

$$\begin{pmatrix} 1 & S & S & S \\ S & 1 & S & S \\ S & S & 1 & S \\ S & S & S & 1 \\ 0 & 0 & 0 & 0 \end{pmatrix}.$$

Instances like this demonstrate the utility of the regular expression simplification theorem; a recursive function implementation of this theorem would simplify output to the arbitrary computation. However, we foresee some challenges to the implementation of this algorithm.

First, the proof of our algorithm relies on a set of $n+1$ regular expressions, yet it is obvious that the inclusion of redundant regular expressions of the same computation length in the union wouldn't alter this equivalence. A program implementation of this theorem would require an algorithm to parse through and identify which of these regular expressions are redundant and don't match the format of those described in our proof. For example, adding the row "$1,1,1,1$" to the matrix generated by example 5 would change the matrix to not match the specifications of the theorem, but the union would still simplify to the arbitrary computation.

Next, it is possible that a regular expression could be appended with a substring of 'S's at the beginning or end. Consider the following mLTL formula:

Example 6. $T\ \mathcal{U}_{[2,3]}p_0 \equiv \mathcal{F}_{[2,3]}p_0$

The WEST program generates the following union of regular expressions:

$$\begin{pmatrix} S & S & 1 & S \\ S & S & S & 1 \\ S & S & 0 & 0 \end{pmatrix}.$$

It can clearly be seen that this is equivalent to the arbitrary computation, as one can remove the first two columns and apply the simplification theorem to the remaining columns.

Furthermore, identical substrings may be interspersed throughout the set of regular expressions. Then, the union of them won't simplify to the arbitrary computation, but rather, the arbitrary computation "broken up" by common substrings. Consider the following modification to the matrix generated by example 5:

$$\begin{pmatrix} 0 & \mathbf{1} & \mathbf{S} & 1 & S & 0 & \mathbf{S} & \mathbf{S} & 1 \\ 0 & \mathbf{S} & \mathbf{1} & 1 & S & 0 & \mathbf{S} & \mathbf{S} & 1 \\ 0 & \mathbf{S} & \mathbf{S} & 1 & S & 0 & \mathbf{1} & \mathbf{S} & 1 \\ 0 & \mathbf{S} & \mathbf{S} & 1 & S & 0 & \mathbf{S} & \mathbf{1} & 1 \\ 0 & \mathbf{0} & \mathbf{0} & 1 & S & 0 & \mathbf{0} & \mathbf{0} & 1 \end{pmatrix} \equiv 0\mathbf{SS}1S0\mathbf{SS}1$$

The substrings "0", "1S0", and "1" are inserted at the beginning, middle, and end, respectively. In general, a vector of length m strings seeking to apply the theorem to n of its columns has $\binom{m}{n}$ combinations of columns to check for. Implementation of this algorithm can drastically increase run-time while proving its utility in only a small number of circumstances.

8 Closing Remarks

One goal of this research project is to visually represent mLTL formulas as truth tables, similar to classical truth tables in propositional logic. The goal is motivated by the fact that mLTL is widely used in industry by engineers and scientists to write specifications for safe critical systems, and the WEST program can aide in the debugging of such specifications. The WEST program accomplishes this: given mLTL formula α, we compute a way to capture all satisfying formulas π in a relatively readable format that additionally captures structural patterns in such regular expressions. The tool itself has demonstratively reasonable run time. Although worst case theoretical complexity is doubly exponential, our benchmarking shows that the most severe case is very sparse in practice.

We have also applied techniques in intelligent fuzzing to explore the state space of the algorithm in a structured way. For a reasonably complex program, a complete walk of the input space is infeasible. However, basing testing on a (manually constructed) graph of the algorithm provides a middle ground where some level of coverage is guaranteed. Although this isn't a magic box that generates test cases for arbitrary algorithms, the methods used in algorithmic analysis and test suite construction are easily adaptable to similar recursive algorithms.

8.1 Additional Research Directions

Describe all satisfying computations to LTL formulas using ω-regular expressions.

ω-regular expression are defined to match only infinite words, and can be used to describe satisfying computations to LTL formulas (which has infinite time steps). Our original efforts were spent reasoning about the general LTL case before restricting to mLTL. Particular difficulties revolves around the need for the Kleene star operation, which leads to messy arithmetic of infinite unions and intersections of regular expressions.

Simplifying unions of regular expressions to a minimal representation.

Theorem 5 addresses a non-trivial situation in which a vector of regular expressions may be simplified. A natural question to ask is what is the minimum number of strings of regular expressions needed to represent a language of computations. However, such a minimal representation is not unique. For instance, $\mathscr{L}(S1|1S) = \mathscr{L}(S1|10) = \{01, 10, 11\}$. There may be other schemes needed to simplify any arbitrary union of regular expressions to their minimal representation.

A more general application of intelligent fuzzing to recursive algorithms.
A tool to systematically convert a recursive algorithm (perhaps written in some strictly defined pseudocode syntax) into a directed graph representation of the state space would be a helpful aid for generating test suites. Significant efforts were made to construct the graph of the WEST algorithm by hand, and an automated process would invaluable. Additional care should be put into allowing for varying levels of abstractions of execution due to concerns of the path explosion problem.

8.1.1 Acknowledgements

This work was done as part of Iowa State University's Research Experience for Undergraduates funded by NSF grant DMS-1950583. We would like to thank Iowa State University and the NSF for this research opportunity.

The assistance of our mentors Dr. Kristin Yvonne Rozier and Laura Gamboa Guzmán was invaluable and greatly appreciated. In addition, we thank Dr. Steve Butler and Dr. Bernard Lidický for their management of this program and organization of its funding.

Finally, we want to thank Brian Kempa for his guidance and suggestions on the intelligent fuzzing portion of our research, and Dr. Steve Butler for his intuition on the Regular Expression Simplification Theorem.

References

[1] Alexis Aurandt, Phillip H. Jones, and Kristin Yvonne Rozier. Runtime verification triggers real-time, autonomous fault recovery on the CySat-i. In *Lecture Notes in Computer Science*, pages 816–825. Springer International Publishing, 2022.

[2] Armin Biere, Alessandro Cimatti, Edmund M Clarke, Ofer Strichman, and Yunshan Zhu. Bounded Model Checking. 9 1992.

[3] Matthew Cauwels, Abigail Hammer, Benjamin Hertz, Phillip H. Jones, and Kristin Y. Rozier. Integrating runtime verification into an automated UAS traffic management system. In *Communications in Computer and Information Science*, pages 340–357. Springer International Publishing, 2020.

[4] Keliang He, Morteza Lahijanian, Lydia E. Kavraki, and Moshe Y. Vardi. Towards manipulation planning with temporal logic specifications. In *2015 IEEE International Conference on Robotics and Automation (ICRA)*, pages 346–352, 2015.

[5] Benjamin Hertz, Zachary Luppen, and Kristin Yvonne Rozier. Integrating runtime verification into a sounding rocket control system. In *Lecture Notes in Computer Science*, pages 151–159. Springer International Publishing, 2021.

[6] Brian Kempa, Pei Zhang, Phillip H. Jones, Joseph Zambreno, and Kristin Yvonne Rozier. Embedding online runtime verification for fault disambiguation on robonaut2. In *Lecture Notes in Computer Science*, pages 196–214. Springer International Publishing, 2020.

[7] Jianwen Li, Moshe Y. Vardi, and Kristin Y. Rozier. Satisfiability checking for mission-time ltl (mltl). *Information and Computation*, page 104923, 2022.

[8] Barton P. Miller, Lars Fredriksen, and Bryan So. An empirical study of the reliability of UNIX utilities. *Communications of the ACM*, 33(12):32–44, dec 1990.

[9] Andreas Nonnengart and Christoph Weidenbach. Chapter 3.2 - negation normal forms. In Alan Robinson and Andrei Voronkov, editors, *Handbook of Automated Reasoning*, Handbook of Automated Reasoning, pages 335–367. North-Holland, Amsterdam, 2001.

[10] Van-Thuan Pham, Marcel Böhme, and Abhik Roychoudhury. Model-based whitebox fuzzing for program binaries. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*. ACM, aug 2016.

[11] Michael Sipser. *Introduction to the theory of Computation*. Course Technology, 2020.

Appendix

I Until and Release Duality Lemma

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

Proof.
($\Rightarrow$):
Suppose $\pi \models \alpha\mathcal{R}_{[a,b]}\beta$, so:

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

We proceed by cases to show that:

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

Case 0: If $|\pi| \leq a$, then we are immediately done.
Case 1: Suppose $\forall i \in [a,b]$, $\pi_i \models \beta$.
Through re-labeling, we have $\forall s \in [a,b]$, $\pi_s \models \beta$. Then we clearly have:

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

Case 2: Suppose $\exists i \in [a,b]$, $\pi_i \not\models \beta$.
Then we must have that:

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

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

Suppose by contradiction that:

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

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 $\pi_j \not\models \alpha$ from Line 2.
However from Line 1 we have that $\pi_j \models \alpha$ and have thus derived a contradiction.
Thus, we now have that $\forall s \in [a,b]$, $(\exists t \in [a,s-1],\ \pi_t \models \alpha)$.
From this, we clearly have that:

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

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

($\Leftarrow$):
Suppose that $\pi \models \neg(\neg\alpha\mathcal{U}\neg\beta)$:

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

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

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

Case 0: If $|\pi| \leq a$, again we are immediately done.
Case 1: Suppose $\forall s \in [a,b]$, $(\pi_s \models \beta)$.

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

Case 2: Suppose $\exists s \in [a,b], \pi_s \nvDash \beta$.
By re-labeling s to i, we have that $\exists i \in [a,b], \pi_i \nvDash \beta$.
Since $[a,b]$ is a finite-discrete interval, there must exist a first i_1 s.t. $\pi_{i_1} \nvDash \beta$, that is:

$$\exists i_1 \in [a,b],\ (\pi_{i_1} \nvDash \beta \text{ and } \forall k \in [a, i_1 - 1],\ \pi_k \vDash \beta) \tag{2}$$

Since (line 2) $\exists i_1 \in [a,b], \pi_{i_1} \nvDash \beta$, by Line 1 we have that: $\pi_{i_1} \vDash \beta$ or $\exists t \in [a, i_1 - 1]$, $\pi_t \vDash \alpha$. Thus, we have that:

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

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

$$\forall k \in [a,t],\ \pi_k \vDash \beta \tag{4}$$

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:

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

From line 5, we now get:

$$|\pi| \le a \text{ or } \forall i \in [a,b],\ (\pi_i \vDash \beta) \text{ or } \exists j \in [a, b-1],\ (\pi_j \vDash \alpha \text{ and } \forall k \in [a,j]\ \pi_k \vDash \beta) \tag{B1}$$

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

This finishes the proof. □

II Regular Expression Simplification Theorem

Proof of 5. 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 $n \ge 2$. 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 $1 \leq k \leq n$.. Assume WLOG that this value is a 0. Matrix J can be represented as follows:

$$J = \begin{array}{cccccccc}
\mathbf{c_1} & & \mathbf{c_{k-1}} & \mathbf{c_k} & \mathbf{c_{k+1}} & & \mathbf{c_n} & \\
S & \dots & S & 0 & S & \dots & S & \mathbf{r_1} \\
 & & & S & & & & \mathbf{r_2} \\
 & & & \vdots & & & & \\
 & & & S & & & & \\
 & \vdots & & 1 & & \vdots & & \\
 & & & S & & & & \\
 & & & \vdots & & & & \\
 & & & S & & & & \mathbf{r_{n+1}}
\end{array}$$

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 kth 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 kth 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:

$$J = \begin{array}{cccccccc}
\mathbf{c_1} & & \mathbf{c_{k-1}} & \mathbf{c_k} & \mathbf{c_{k+1}} & & \mathbf{c_n} & \\
S & \dots & S & 0 & S & \dots & S & \mathbf{r_1} \\
 & & & 1 & & & & \mathbf{r_2} \\
 & \vdots & & \vdots & & \vdots & & \\
 & & & 1 & & & & \mathbf{r_{n+1}}
\end{array}$$

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 $J' = J - r_1$:

$$J' = \begin{array}{cccc}
\mathbf{c_1} & \mathbf{c_k} & \mathbf{c_n} & \\
 & 1 & & \mathbf{r_2} \\
\vdots & \vdots & \vdots & \\
 & 1 & & \mathbf{r_{n+1}}
\end{array}$$

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 kth position. Because each row indeed contains a '$1'$ at the kth 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 kth 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. □

III 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 B, where $R = R_1|R_2|...|R_k$ is represented as a vector of strings $R = R_1, R_2, ..., R_k$, and $B = B_1|B_2|...|B_\ell$ is represented as a vector of strings $B = B_1, B_2, ..., B_\ell$. Additionally, n will always refer to the number of propositional variables, and nnf refers to an input formula in negation normal form.

Algorithm 1 Pad a vector to elements of all equal length

```
Input: Vector of strings that represents a regex, number of propositional variables
Output: Vector of strings padded to equal length
  procedure PAD(vector $R$, int $n$)
    max_Length ← $max_{\{r \in R\}}(r.\text{length}())$          ▷ determines length to pad all strings to
    for ($r \in R$) do
      diff ← (max_length−$r$.length()) / $(n+1)$          ▷ determines how many $S^n$s to append
      $r \leftarrow r + (, S^n)^{\text{diff}}$
    return $R$
```

Algorithm 2 Computes regex for propositional constant

```
Input: String that is either "T" or "!", number of propositional variables
Output: Vector of strings that represents the appropriate satisfying computations
  procedure REG_PROP_CONS(string nnf, int $n$)
    if (nnf = "T" and $n \neq 0$) then
      return $\{S^n\}$                    ▷ regex of satisfying computations for "T"
    else
      return {}                    ▷ regex of satisfying computations for "!"
```

Algorithm 3 Output the vector of computation satisfying a propositional variable.

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

```
procedure REG_PROP_VAR(string nnf, int $n$)
    if (nnf = “p$k$”, where $k$ is a nonnegative integer) then
        return $\{S^k 1 S^{n-k-1}\}$                ▷ regex of satisfying computations for “p$k$”
    if (nnf = “∼p$k$”, where $k$ is a nonnegative integer) then
        return $\{S^k 0 S^{n-k-1}\}$                ▷ regex of satisfying computations for “∼p$k$”
```

Algorithm 4 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 $b$)
    if ($r$.length() $\neq$ $b$.length()) then
        exit                                    ▷ cannot compare strings if they differ in length
    for ($0 \leq i < r$.length()) do
        pre_r ← $r[0, i-1]$                     ▷ stores string $r$ up to comparison character
        char_r ← $r[i]$                         ▷ stores character to compare in $r$
        post_r ← $r[i+1, r$.length() $- 1]$     ▷ stores remaining part of string $r$

        pre_b ← $b[0, i-1]$                     ▷ stores string $b$ up to comparison character
        char_b ← $b[i]$                         ▷ stores character to compare in $b$
        post_b ← $b[i+1, b$.length() $- 1]$     ▷ stores remaining part of string $b$

        if (pre_r = pre_b and post_r = post_b) then   ▷ checks that strings $r$ and $b$ match for
                                                        all but one character
            if (char_r $\neq$ char_b) then
                return pre_r +“$S$” + post_r    ▷ if the comparison characters differ, replace char-
                                                  acter with “$S$”
            else
                return $r$                      ▷ if the comparison character match, strings are identical
    return FAIL                                 ▷ Indicates strings differed by more than one character
```

Algorithm 5 Simplifies a vector of strings using simplify_string

```
Input: Vector of strings representing a regex
Output: Vector of strings representing a regex simplified using simplify_string
  procedure SIMPLIFY(vector $R$)
      Pad all strings in $R$ to the same length        ▷ ensures all inputs to simplify_string are valid
      if ($R$.length() $\leq 1$) then
          return $R$                                   ▷ insufficient strings in $R$ to use simplify_string
      $i = R$.length $- 1$
      $j = i - 1$
      START
      while ($i \geq 1$) do
          while ($j \geq 0$) do
              simplified = simplify_string($R[i]$, $R[j]$)    ▷ for each combination of strings in $R$,
                                                                apply simplify_string
              if (simplified $\neq$ FAIL) then
                  replace string at index $j$ with simplified    ▷ updating $R[j]$ to be combined string
                  remove string at index $i$ from $R$            ▷ remove $R[i]$, as it's redundant
                  $i = R$.length() $- 1$
                  $j = i - 1$
                  goto START
              --$j$
          --$i$
          $j = i - 1$
      return $R$
```

Algorithm 6 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 $b$)
      ret ← “”                                         ▷ initializing string to store bitwise AND
      for (i $\in [0,\ r$.length()]) do
          if ($r[i] \wedge b[i]$ = “”) then
              return “”             ▷ if bitwise AND is ever empty, entire string intersection is empty
          else
              ret ← ret + $r[i] \wedge b[i]$                   ▷ for each character, store bitwise AND
      return ret
```

Algorithm 7 Takes set intersection of the languages of two regexes

```
Input: Vectors of strings that represent regexes, simplify boolean
Output: Vector of strings that represents set intersection of the languages of the inputted regexes
  procedure SET_INTERSECT(vector $R$, vector $B$, bool simp)
      Pad $R$ and $B$ to the same length
      ret ← {}                                  ▷ initializing a vector of strings to store set intersection
      for ($r \in R$) do
          for ($b \in B$) do          ▷ take bitwise AND of every combination of strings from $R$ and $B$
              add bit_wise_and($r, b$) to ret                               ▷ store results in ret
      if (simp is true) then
          return simplify(ret)          ▷ if user requests simplified output, run ret through simplify
      else
          return ret
```

Algorithm 8 Takes union of two regexes (combines two vectors into one)

```
Input: Vectors of strings that represent regexes, simplify boolean
Output: Vector of strings that represents union of inputted regexes
  procedure JOIN(vector $R$, vector $B$, bool simp)
    ret ← {}                        ▷ initializing a vector of strings to store union
    for ($r \in R$) do
      add $r$ to ret                ▷ add all vectors in $R$ to ret
    for ($b \in B$) do
      add $b$ to ret                ▷ add all vectors in $B$ to ret
    if (simp is true) then
      return simplify(ret)          ▷ if user requests simplified output, run ret through simplify
    else
      return ret
```

Algorithm 9 Computes the regex for a mLTL formula G[a:b]α

```
Inputs: Vector of strings representing the regex for $\alpha$, interval bounds, number of propositional variables, simplify boolean
Output: Vector of strings that represents the appropriate satisfying computations
  procedure REG_G(vector reg_alpha, int $a$, int $b$, int $n$, bool simp)
    pre ← $((‘S’)^n + ‘,’)^a$         ▷ before the start of the interval, computation is arbitrary
    comp ← reg_alpha               ▷ initializing vector of strings for outputted regex with subformula regex
    if $a > b$ then                ▷ empty interval, Global is trivially true
      return $\{S^n\}$
    for ($1 \leq i \leq b - a$) do     ▷ computing $\bigwedge_{i=1}^{b-a}(S^n,)^i\text{reg}(\alpha)$
      temp_alpha ← $((‘S’)^n + ‘,’)^i$ + reg_alpha
      comp ← set_intersect(comp, temp_alpha, simp)
    return pre + comp              ▷ prepending string pre to every string in comp; result is regex for Global defined in section 3
```

Algorithm 10 Computes the regex for a mLTL formula F[a:b]α

```
Inputs: Vector of strings representing the regex for $\alpha$, interval bounds, number of propositional variables, simplify boolean
Output: Vector of strings that represents the appropriate satisfying computations
  procedure REG_F(vector reg_alpha, int $a$, int $b$, int $n$, bool simp)
    pre ← $((‘S’)^n + ‘,’)^a$         ▷ before the start of the interval, computation is arbitrary
    comp ← reg_alpha               ▷ initializing vector of strings for outputted regex with subformula regex
    if $a > b$ then                ▷ empty interval, Finally is trivially false
      return {}
    for ($1 \leq i \leq b - a$) do     ▷ computing $\bigvee_{i=1}^{b-a}(S^n,)^i\text{reg}(\alpha)$
      temp_alpha ← $((‘S’)^n + ‘,’)^i$ + reg_alpha
      comp ← join(comp, temp_alpha, simp)
    return pre + comp              ▷ prepending string pre to every string in comp; result is regex for Finally defined in section 3
```

Algorithm 11 Computes the regex for a mLTL formula αU[a:b]β

Inputs: Vectors of strings representing the regexes for α and β, interval bounds, number of propositional variables, simplify boolean
Output: Vector of strings that represents the appropriate satisfying computations

```
  procedure REG_U(vector reg_alpha, vector reg_beta int $a$, int $b$, int $n$, bool simp)
      comp ← ((‘$S$’)$^n$ + ‘,’)$^a$+ reg_beta      ▷ initializing vector of strings for outputted regex with regex for beta, prepended with ‘$S'$s up to start of interval
      if $a > b$ then
          return {}      ▷ empty interval, Until is trivially false
      for $(a \leq i \leq b-1)$ do      ▷ computing $\bigvee_{i=a}^{b}$ reg $\left(\mathcal{G}_{[a,i-1]}\alpha \wedge \mathcal{G}_{[i,i]}\beta\right)$, loop ends at $b-1$ because comp is initialized to $i = a$ case and loop is reindexed
          temp_comp ← set_intersect(reg_G(reg_alpha, $a$, $i$, $n$), reg_G(reg_beta, $i+1$, $i+1$, $n$), $n$, simp)
          comp ← join(comp, temp_comp, simp)
      return comp
```

Algorithm 12 Computes the regex for a mLTL formula αR[a:b]β

Inputs: Vectors of strings representing the regexes for α and β, interval bounds, number of propositional variables, simplify boolean
Output: Vector of strings that represents the appropriate satisfying computations

```
  procedure REG_R(vector reg_alpha, vector reg_beta int $a$, int $b$, int $n$, bool simp)
      comp ← reg_G(reg_beta, $a$, $b$, $n$, simp)      ▷ initializing vector of strings for output regex with regex for G[a:b]$\beta$
      if $a > b$ then
          return $\{S^n\}$      ▷ empty interval, Release is trivially true
      for $(a \leq i \leq b-1)$ do      ▷ computing reg $\left(\mathcal{G}_{[a,b]}\beta\right) \vee \bigvee_{i=a}^{b-1}$ reg $\left(\mathcal{G}_{[a,i]}\beta \wedge \mathcal{G}_{[i,i]}\alpha\right)$, first run through loop includes reg $\left(\mathcal{G}_{[a,b]}\beta\right)$
          temp_comp ← set_intersect(reg_G(reg_beta, $a$, $i$, $n$), reg_G(reg_alpha, $i$, $i$, $n$), $n$, simp)
          comp ← join(comp, temp_comp, simp)
      return comp
```

Algorithm 13 Computes the regex for an mLTL formula in NNF

Inputs: mLTL formula in negation normal form, number of propositional variables, boolean sub for storing subformula computations in global variable FORMULAS, simplify boolean
Output: Vector of strings that represents the appropriate satisfying computations

```
procedure REG(string nnf, int $n$, bool sub, bool simp)
    if (nnf is a prop. var. or ~prop. var) then
        if (sub) then
            Run reg on formula and store formula with its regular expression
        return reg_prop_var(nnf, $n$)    ▷ call helper function for propositional variable

    if (nnf is $T$ or !) then
        if (sub) then
            Run reg on formula and store formula with its regular expression
        return reg_prop_cons(nnf, $n$)    ▷ call helper function for propositional constant

    if (nnf is of the form unary_temp_conn interval alpha) then
        unary_temp_conn ← unary_temp_conn in nnf    ▷ parse formula to find unary operator
        $a$ ← start of interval
        $b$ ← end of interval
        reg_alpha ← reg(alpha, $n$, sub, simp)    ▷ recursive call to reg on subformula

        if (unary_temp_conn = ‘F’) then
            if (sub) then
                Run reg on formula and store formula with its regular expression
            return reg_F(reg_alpha, $a$, $b$, $n$, simp)    ▷ Call helper function for Finally operator

        if (unary_temp_conn = ‘G’) then
            if (sub) then
                Run reg on formula and store formula with its regular expression
            return reg_G(reg_alpha, $a$, $b$, $n$, simp)    ▷ Call helper function for Global operator

    if (nnf is of the form “(Assoc_prop_conn[nnf_array])”) then ▷ Rewrite nnf in non-array form
        assoc_prop_conn ← assoc_prop_conn in nnf
        equiv_formula ← “”    ▷ For instance, $\&[\alpha_1, \alpha_2, \alpha_3]$ is rewritten to $((\alpha_1\&\alpha_2)\&\alpha_3)$
        for (array entry alpha in nnf_array) do
            if (alpha is first entry) then
                equiv_formula ← alpha    ▷ First entry doesn’t need parentheses
            else
                equiv_formula = “(” + equiv_formula + assoc_prop_conn + alpha + “)”
        if (sub) then
            Run reg on formula and store formula with its regular expression
        return reg(equiv_formula, $n$, sub, simp)    ▷ recursively call reg on the rewritten formula
```

```
if (nnf is of the form alpha binary_prop_conn beta) then
    binary_prop_conn ← binary_prop_conn in nnf      ▷ parse formula to find binary operator
    reg_alpha ← reg(alpha, $n$, sub, simp)               ▷ recursive call to reg on subformula
    reg_beta ← reg(beta, $n$, sub, simp)                 ▷ recursive call to reg on subformula

    if (binary_prop_conn = “&”) then
        if (sub) then
            Run reg on formula and store formula with its regular expression
        return set_intersect(reg_alpha, reg_beta, $n$, simp)          ▷ Call helper function for &
operator

    if (binary_prop_conn = “∨”) then
        if (sub) then
            Run reg on formula and store formula with its regular expression
        return join(reg_alpha, reg_beta, $n$, simp)          ▷ Call helper function for ∨ operator

    if (binary_prop_conn = “=”) then                 ▷ Rewrite in terms of OR, AND, and NOT
        equiv_formula = “((” + alpha + “&” + beta + “)∨(” + wff_to_nnf(∼alpha) +
        “&” + wff_to_nnf(∼beta) + “))”
        if (sub) then
            Run reg on formula and store formula with its regular expression
        return reg(equiv_formula, $n$, sub, simp)    ▷ recursive call to reg on rewritten formula

    if (binary_prop_conn = “>”) then                 ▷ Rewrite in terms of OR, AND, and NOT
        equiv_formula = “(” + wff_to_nnf(∼alpha) + ”∨” + beta + “)”
        if (sub) then
            Run reg on formula and store formula with its regular expression
        return reg(equiv_formula, $n$, sub, simp)    ▷ recursive call to reg on rewritten formula

if (nnf is of the form alpha binary_temp_conn interval beta) then
    binary_temp_conn ← binary_temp_conn in nnf                    ▷ parse for binary operator
    $a$ ← start of interval
    $b$ ← end of interval
    reg_alpha ← reg(alpha, $n$, sub, simp)               ▷ recursive call to reg on subformula
    reg_beta ← reg(beta, $n$, sub, simp)                 ▷ recursive call to reg on subformula

    if (binary_temp_conn = ‘U’) then
        if (sub) then
            Run reg on formula and store formula with its regular expression
        return reg_U(reg_alpha, reg_beta, $a$, $b$, $n$, simp)          ▷ Call helper function for Until

    else if (binary_temp_conn = ‘R’) then
        if (sub) then
            Run reg on formula and store formula with its regular expression
        return reg_R(reg_alpha, reg_beta, $a$, $b$, $n$, simp)       ▷ Call helper function for Release

Throw exception          ▷ If none of the previous statements held, there’s an error with the
                           formula nnf
return {}
```

Algorithm 14 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: Vector 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 $\{p, q\}$                      ▷ $p$ represents a propositional variable or its negation,
                                                 $q$ represents a propositional constant
    template ← {}                              ▷ initialize variable to store final template
    $V$ ← generate_test_template(depth−1, $a$, $b$)    ▷ recursively generate all formulas of 1 lower
                                                 depth
    for (string $\alpha \in V$) do
        add "G[a:b]" + $\alpha$ to template
        add "F[a:b]" + $\alpha$ to template           ▷ Add unary temporal operators to template
        for string $\beta \in V$ do
            add $\alpha$ + "U[a:b]" + $\beta$ to template
            add $\alpha$ + "R[a:b]" + $\beta$ to template
            add $\alpha$ + "∨" + $\beta$ to template
            add $\alpha$ + "&" + $\beta$ to template      ▷ add binary operators to template
    return template
```

Algorithm 15 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: Vector of mLTL formulas

```
procedure GENERATE_TEST(int depth, int $a$, int $b$, int $n$)
    tests ← generate_test_template(depth, $a$, $b$)
    for string $t \in$ tests do
        for char ch $\in t$ do
            if ch = $p$ then                   ▷ replace $a$ with a random propositional variable or
                                                 the negation of a random propositional variable
                $k$ ← rand()%$n$                ▷ assign $k$ as a random int between 0 and $n-1$
                if rand() %2 = 0 then          ▷ imitate randomness of a coin toss
                    replace ch with p$k$
                else
                    replace ch with ∼p$k$
            else if ch = $q$ then              ▷ Replace $q$ with a random propositional constant
                if rand() %2 = 0 then
                    replace ch with T
                else
                    replace ch with !
    return tests
```

IV State Diagram Graphs

Below we provide the graphs of state diagrams of the critical functions examined in our intelligent fuzzing. In general, nodes in the graph represent portions of the code without control flow statements. As a result, 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 these graphs, the option to run the *simplify* function has been omitted for clarity. For the graph of the *reg* function, the option to store subformulas has been omitted as well.

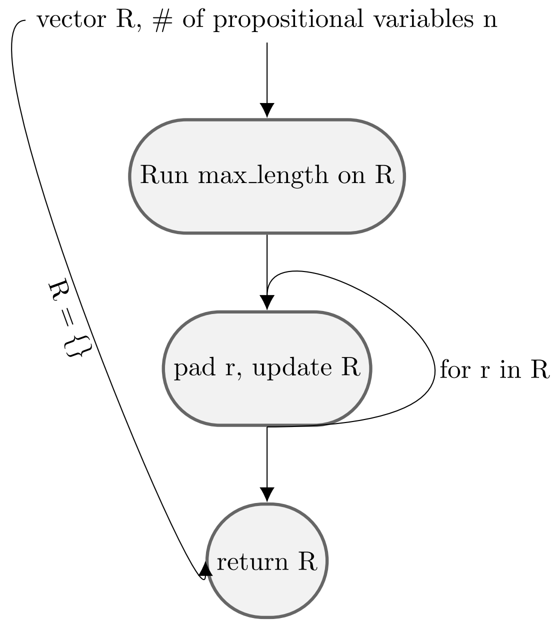


Figure 10: Pad Function

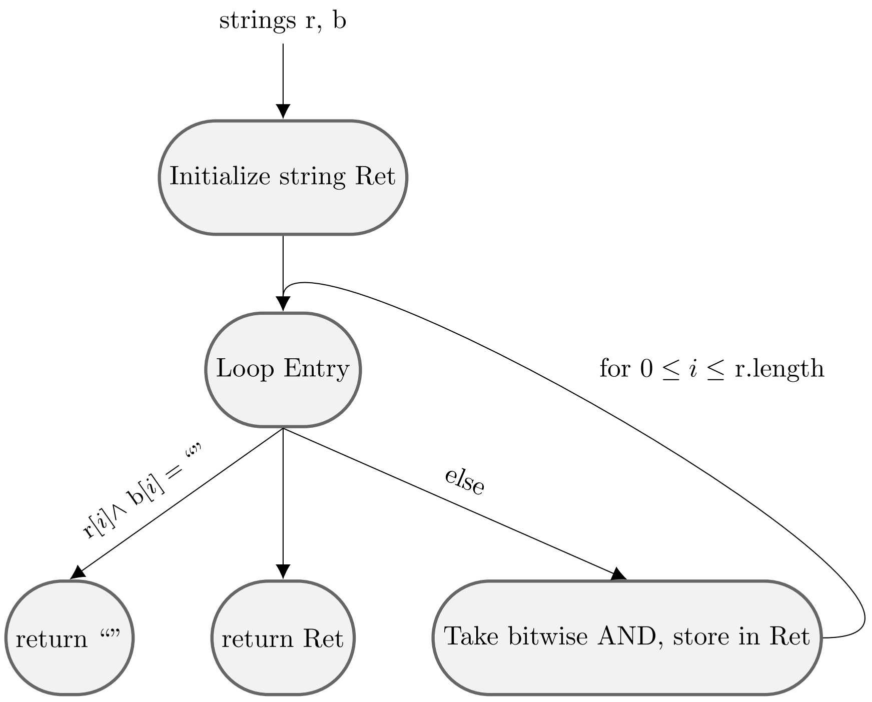


Figure 11: Bitwise_And Function

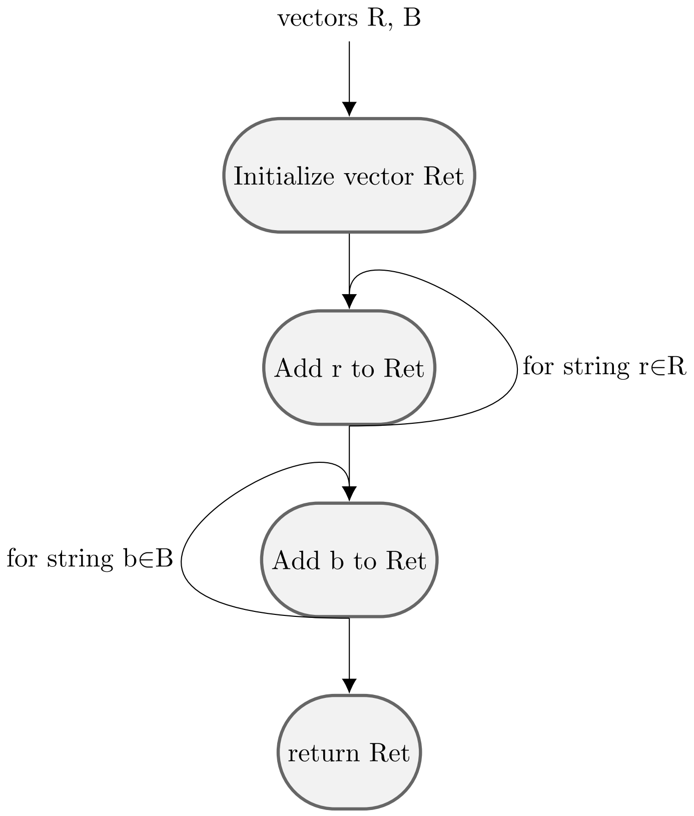


Figure 12: Join Function

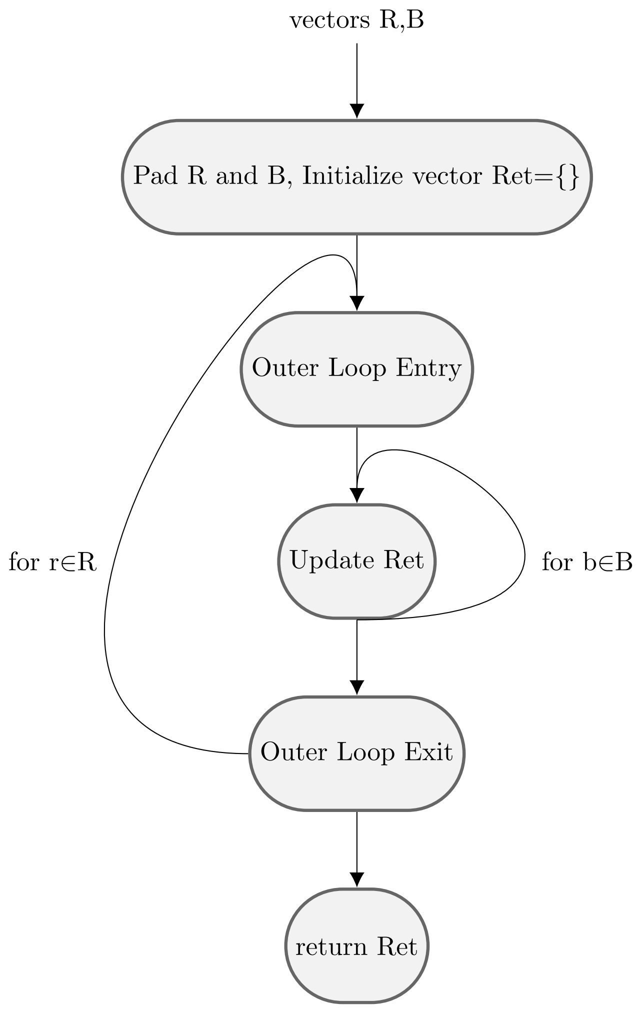
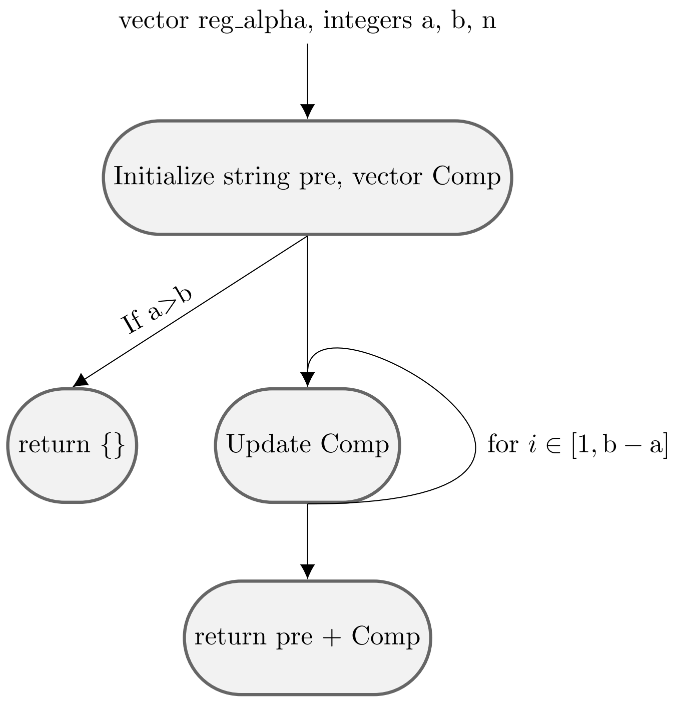
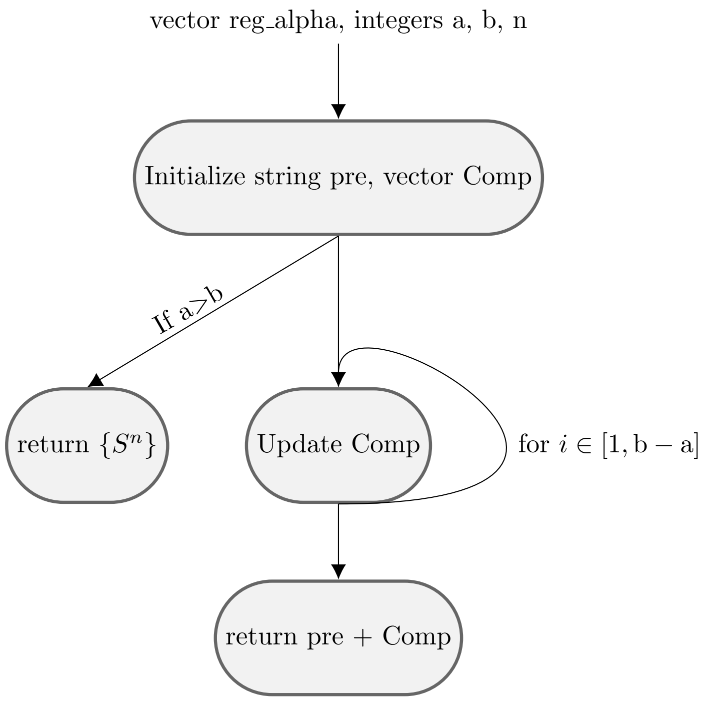


Figure 13: Set_Intersect Function



vector reg_alpha, integers a, b, n
Initialize string pre, vector Comp
If a>b
return {}
Update Comp
for $i \in [1, \mathrm{b}-\mathrm{a}]$
return pre + Comp

Figure 14: Reg_F Function



vector reg_alpha, integers a, b, n
Initialize string pre, vector Comp
If a>b
return $\{S^n\}$
Update Comp
for $i \in [1, \mathrm{b}-\mathrm{a}]$
return pre + Comp

Figure 15: Reg_G Function

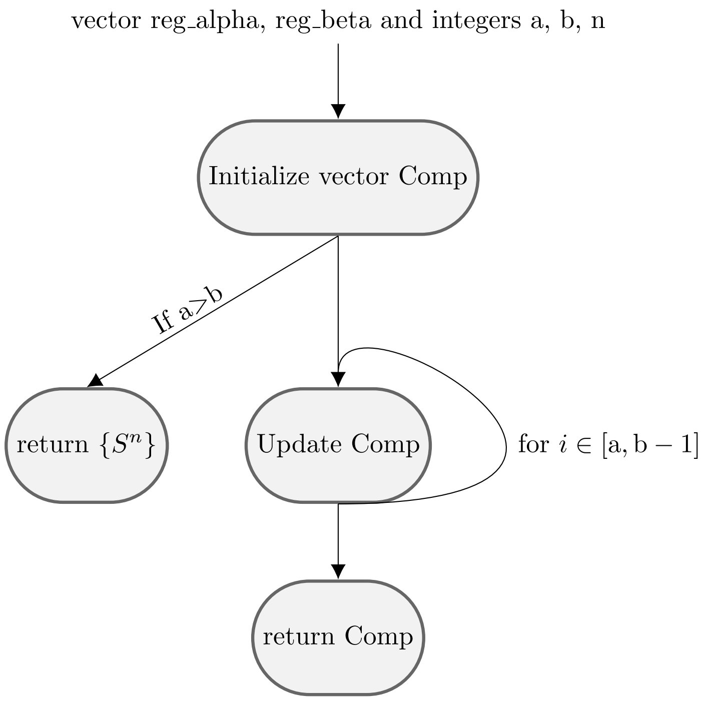

vector reg_alpha, reg_beta and integers a, b, n
Initialize vector Comp
If a>b
return $\{S^n\}$
Update Comp
for $i \in [\mathrm{a}, \mathrm{b}-1]$
return Comp

Figure 16: Reg_R Function

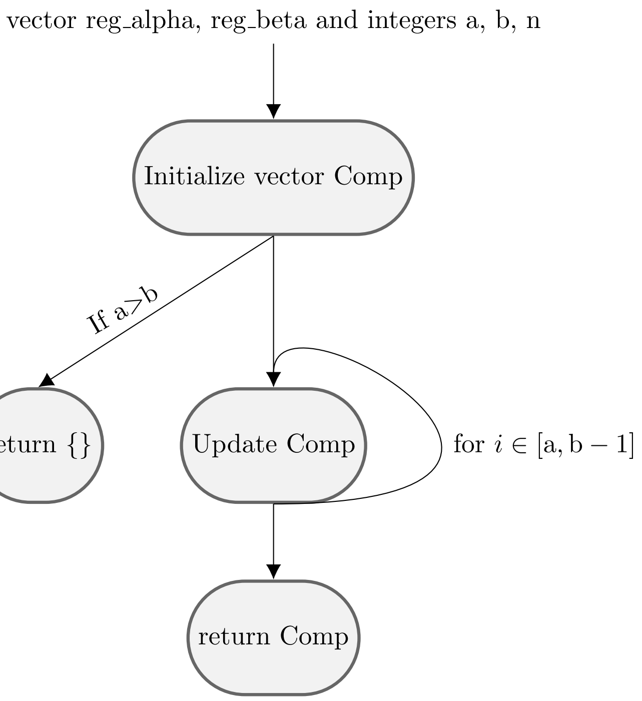

vector reg_alpha, reg_beta and integers a, b, n
Initialize vector Comp
If a>b
return {}
Update Comp
for $i \in [\mathrm{a}, \mathrm{b}-1]$
return Comp

Figure 17: Reg_U Function

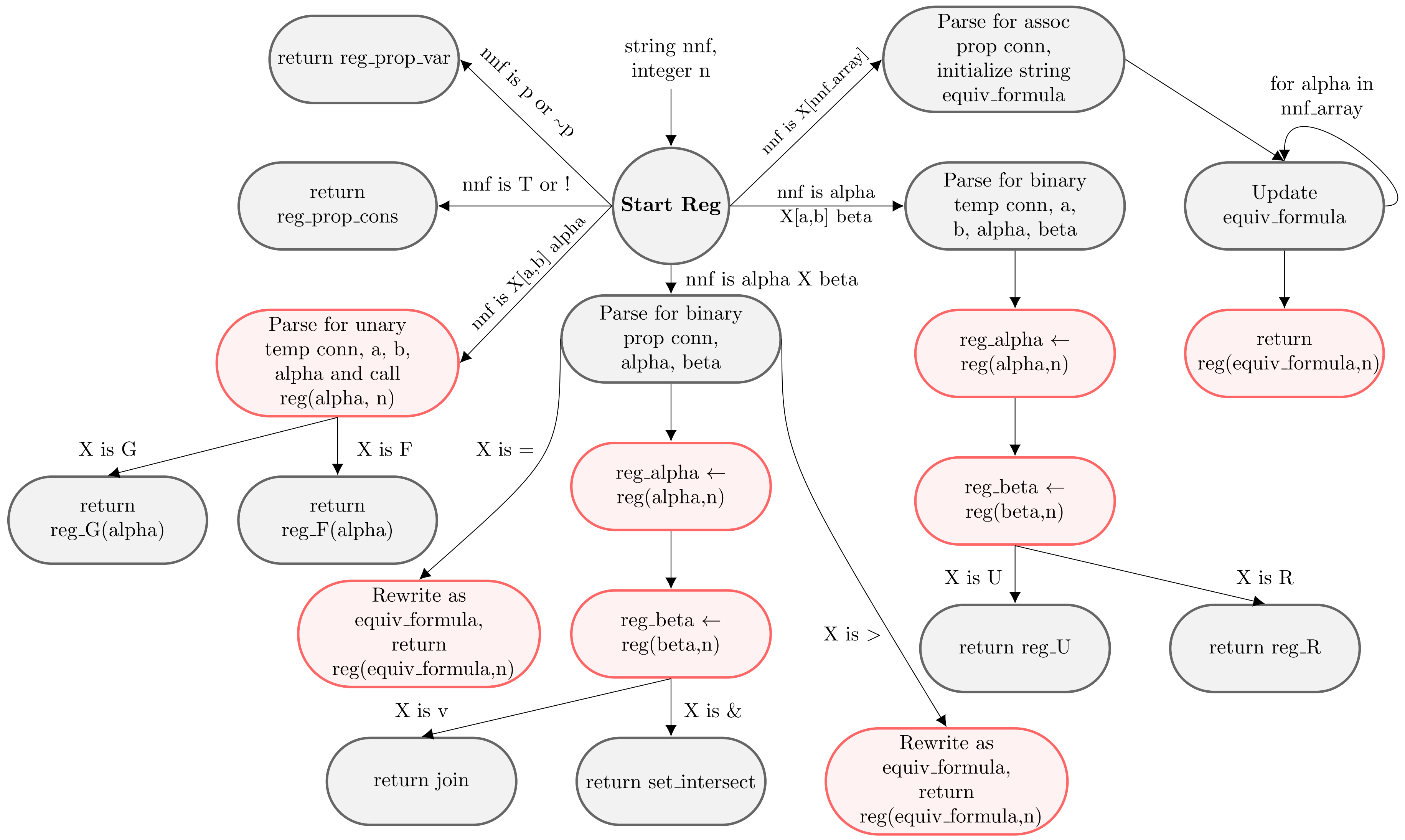


Figure 18: Control Flow of Reg. Red nodes indicate a recursive call to Reg.