RuSTL: Runtime Verification using Signal Temporal Logic

by

Waleed Qadir Khan

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Science
in
Electrical and Computer Engineering

Waterloo, Ontario, Canada, 2019

Examining Committee Membership

The following served on the Examining Committee for this thesis. The decision of the Examining Committee is by majority vote.

Supervisor(s):	Sebastion Fischmeister Professor, Dept. of Electrical and Computer Engineering, University of Waterloo
Internal Member:	Arie Gurfinkel Associate Professor, Dept. of Electrical and Computer Engineering, University of Waterloo
Internal Member:	Mark Crowley Assistant Professor, Dept. of Electrical and Computer Engineering, University of Waterloo

Author's Declaration

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

A system is classified to be a safety-critical system if its failure and/or malfunction of these devices may result in severe injuries or in extreme cases loss of human life. Such systems are all around us, examples of which include pacemakers, respiratory equipment, electrical locks, fire sprinklers and cars among many others. Runtime Verification (RV) is used to monitor the execution of such systems either while running or after execution to ensure that the system under observation does not violate any safety constraints.

RV employs formal specification languages to evaluate a real-world systems. Pnueli introduced the formal specification for Linear Temporal Logic (LTL) in 1977 for specifying propositional time properties of reactive and concurrent systems. Signal Temporal Logic (STL) is a popular extension of LTL, which analyzes dense-time real-valued signal properties with quantitative timing constraints.

In this thesis, we introduce Runtime Verification using Signal Temporal Logic (RuSTL), an offline qualitative semantic tool for monitoring STL properties. RuSTL is designed to parse any valid STL formula φ and create a stand-alone executable *monitor program*, which checks the property against a given trace σ. RuSTL also take in as input structured English text and convert it into an equivalent STL formula. The application also has the capability to automatically generate diagnostic plots that help the user visually inspect the results of the monitor against a given trace.

We prove that the *monitor program* generated by RuSTL is sound and it terminates for any given valid STL property. Furthermore, we prove that the parsing algorithm used to create the *monitor program* is complete.

We evaluated RuSTL's performance over traces collected from an autonomous self-driving vehicle. The experimental results for our RV monitor show that the execution time of the monitor grows linearly with respect to the length of the signal trace provided.

Acknowledgements

I am extremely grateful to my supervisor Professor Dr. Sebastian Fischmeister, for giving me this opportunity and allowing me to pursue my dream for higher education. He is a great mentor and his guidance has been invaluable during the course of my research and has shown me the importance of clarity in the written words. The words of some unknown author come to mind:

> To my supervisor, for whom no thanks is too much.
>
> ~ Some Unknown Author

I would also like to thank Sean Kauffman, Dr. Carlos Moreno and David Shin for their help and support during my time here. Their counsel on numerous occasions has helped me become a better researcher and am genuinely grateful to them for taking the time out of their schedule to assist me.

Dedication

This is dedicated in memory of my late grandfather Dr. Abdul Qadir Khan — and my surviving grandparents Mr.Asadullah Khan Durrani, Mrs.Sharafat Yasmeen and Mrs.Shakila Qadir. I would also like to dedicate this to my parents, my father Dr. Altaf Qadir Khan, my mother Dr. Faria Altaf.

It is only due to their continuous support and guidance that I am able to make it here today and no amount of thanks will ever be enough to show my gratitude.

Table of Contents

List of Tables

List of Figures

Abbreviations

ANTLR ANother Tool for Language Recognition 7, 17, 18, 20–25, 43, 44

AP Atomic Propositions 9–11, 15

CAN Control Area Network 15, 48

CLI Command-line interface 54

CPS Cyber Physical Systems 15

CSV Comma-separated values 36

CTL Computational Tree Logic [16] 2, 42

EBNF extended Backus-Naur form 17, 43

GIL Graphical Interval Logic [70] 2, 42

GPS Global Positioning System 46

GUI Graphical User Interface 41, 54, 55

IMU Inertial Measurement Unit 46

KF Kalman filters 46

LTL Linear Temporal Logic [68] 2, 9–12, 14, 42

MITL Metric Interval Temporal Logic [3] 13–15, 51, 53, 54

MTL Metric Temporal Logic [50] 2, 12–14, 42, 53

NLP Natural Language Processing 41, 42

QRE Quantified Regular Expressions [65] 2

RAD Renesas Autonomy Demonstrator xi, 45

ROS Robot Operating System 46–48, 51

RTGIL Real-Time Interval Logic [61] 2, 42

RuSTL Runtime Verification using Signal Temporal Logic iv, xi, 1, 3–7, 17, 23, 24, 28, 31, 36, 39, 42–44, 51, 55, 56

RV Runtime Verification 1–4, 6, 9, 11, 12, 15, 43, 55, 56

SPIDER Specification Pattern Instantiation and Derivation EnviRonment [48] 3, 42

STL Signal Temporal Logic [56] 2–7, 15–17, 23, 24, 27, 36, 40, 42, 50, 51, 53–57

TCTL Timed Computational Tree Logic [2] 2, 42

TL Temporal Logic [68] 2, 3, 7, 9, 41, 42, 53

TRE Timed Regular Expressions 54

UDP User Datagram Protocol 48

V2I Vehicle to Infrastructure 46

V2V Vehicle to Vehicle 46

xSTL Extended Signal Temporal Logix 54

List of Symbols

$\mathbb{N}$ Natural Numbers: Whole non-negative number 4

$\mathbb{Q}$ Rational Numbers: Numbers that can be represented as a fraction of two integers, with a non-zero denominator. 4, 13, 24

$\wedge$ Basic logical operator for conjunction 10, 11, 32, 37, 39

$\vee$ Logical operator for disjuncion 10, 11, 27, 32, 37

$\leftrightarrow$ Logical operator for equivalence 10

$\Diamond$ Temporal operator for *Eventual*, which specifies that the formula will hold eventually $\Diamond\varphi = \top\,\mathcal{U}\,\varphi$. 11, 14, 15, 27, 32, 37

φ A temporal property formula iv, 1, 3–7, 10, 11, 14–16, 23, 24, 26–28, 31, 36, 40, 55, 56

$\Box$ Temporal operator for *Global*, which specifies that the formula must always hold. $\Box\varphi = \neg\Diamond\neg\varphi = \neg(\top\,\mathcal{U}\,\neg\varphi)$ 11, 14, 16, 27, 32, 37

$\rightarrow$ Logical operator for implication 10, 11, 27, 32, 37

$\neg$ Basic logical operator for negation 10, 37, 39

$\bigcirc$ Temporal operator for *next*, which specifies what should hold true in the next step. 10

σ Execution trace iv, 1, 4–7, 9, 15, 24, 37, 40

$\mathcal{U}$ Temporal operator for *Until*, which specifies that the first formula should hold until the second formula becomes true. 10, 11, 37, 39

I Time Interval: A nonempty convex subset of $\mathbb{Q}_{\geq 0}$ 13, 24, 31, 33, 37, 42, 43

Chapter 1

Introduction

1.1 Runtime Verification

A system is classified to be a safety-critical system if its failure and/or malfunction of these devices may result in severe injuries or in extreme cases loss of human life. Such systems are all around us, examples of which include cars driving at 100 km/h, airplanes flying at 35,000 feet, fire sprinklers dispersing water to extinguish an uncontrolled flame or pacemakers in the human body ensuring the continued and rhythmic beating of the human heart among many others. Such systems must be tested rigorously to ensure safety. In Runtime Verification (RV) the execution of a system is monitored and analyzed to determine if it satisfies some pre-defined correctness properties. Correctness properties are derived from the requirements of the system or provided by engineers designing the application to ensure that the behavior of the running system complies with the safety specifications. An essential property for any distributed system to ensure is "*a process blocked on a resource will eventually get that resource*". The correctness property formula φ is most commonly defined in a formal specification language. A monitor takes as its input a trace σ, which is either the running execution of a system or prerecorded inputs and outputs. The monitor analyzes the trace σ against the correctness property, where the output denotes whether the given trace σ satisfies or violates the correctness property.

We propose Runtime Verification using Signal Temporal Logic (RuSTL), which performs offline analysis about the future evolution of a continuous time behavior of real-valued signals obtained from sources such as the safety-critical systems described above to verify if they are operating correctly. For such devices, the timing requirements are much stricter, and so these devices must be monitored to ensure that they are operating

as intended. More often than not, malfunctions leading to catastrophic failure proceeded with anomalous behavior or temporal relations between components that are not maintained immediately before failure [8]. An RV monitor addresses these concerns by raising an alert if for example, the temperature of a furnace exceeds the safety limits and the cooling system is not turning on.

Formal specifications such as Linear Temporal Logic [68] (LTL), Computational Tree Logic [16] (CTL), Graphical Interval Logic [70] (GIL) and Quantified Regular Expressions [65] (QRE) are widely used in the fields of formal methods, software verification and model checking and analysis. LTL specifically was developed by Pnueli in 1977 to reason about events in the temporal domain of reactive and concurrent systems. However, these specifications taken as is, do not quantitatively reason about time. When checking the observed behavior of real-time systems, a property in LTL can be defined that checks if "a request is always eventually followed by a response". But a property such as "a request is always followed by a response within 5-time units" cannot be defined using LTL.

There was a need to expand this formal specification for real-valued signals. To that aim, a number of formal specification developed to target real-time properties, such as Metric Temporal Logic [50] (MTL), Timed Computational Tree Logic [2] (TCTL) and Real-Time Interval Logic [61] (RTGIL). In RV various formal specifications are employed [68, 50, 15, 28]. MTL is a popular expansion of LTL, that analyzes real-time systems of Boolean signals. MTL allows temporal operators to have timing constraints and analyzing over an n-dimensional Boolean signal. MTL thus allows the creation of correctness properties of dense-time Boolean signals.

However sensor signals, analog circuits, control systems, etc. don't output data only as Boolean variables. So Signal Temporal Logic [56] (STL) was developed by expanding on the formalism created for MTL, to monitor dense-time real-valued signals. Usually, when observing a running system such as a vehicle or a manufacturing line, the trace being collected, is in the form of a sequence of time-stamped values, where each entry represents an entire snapshot of the entire system and contains Boolean, integer or real values pertaining to the different inputs.

1.2 Structured English

As mentioned earlier, leveraging Temporal Logic [68] (TL) allows the user to check if the system under observation satisfies the desired correctness properties. However, in industry, employing techniques based on formal methods is often met with resistance, due to the

lack of background knowledge and training. Specifications to check on an observed system can come from multiple sources, such as engineers, designers, managers, etc. Non-technical professionals are usually unfamiliar with the formal semantics of TL. It becomes difficult for those unfamiliar with the domain to understand the type of check that the property is trying to perform.

Structured English language sentences are intuitive to understand, and translating them into correctness properties, then provides an incentive for the use of this technology in RV. STL or TL properties can be complicated and highly nested, and just by looking at the formula φ itself, it may not be easy even for domain experts to discern with the property is trying to achieve. Structured text acts as a conduit between generating correctness properties from text requirements and deploying them on the observed system. Structured text that closely resembles the natural English language alleviates some of that complexity. Multiple techniques [40, 5, 35, 60, 62] have been developed that use parsing and natural language processing to compute the specifications from the structured text, in the domain of model checking, robot controllers and formal methods. Specification Pattern Instantiation and Derivation EnviRonment [48] (SPIDER) is a tool that takes Structured English text and converts them to the desired specification properties.

These techniques have been utilized in synthesizing robotic controllers and motion planning [52, 33, 29, 54]. Since they allow designers to enter requirements similar to if they were talking to a person "if in room1, go to room2 next" or "speed never exceeds 20 km/h".

1.3 Thesis Statement

In this thesis, we claim that future bounded STL properties are practically applicable for checking the integrity of dense-time real-valued signal traces from observed systems. The proposed application RuSTL does this by (1) parsing any valid STL property or structured English text, (2) generating a *monitor program* which when provided with a trace determines the satisfiability of the property.

The thesis statement leads to the following problem statement:

Problem Statement: Given a trace and a set of properties in STL, effectively and efficiently determine offline whether the properties are being violated.

1.3.1 Formal Model

For RV, we assume that there is a system under observation, as shown in Figure 1.1 that takes some input, processes the information and generates an output. Let $\mathbb{Q}$ be the set of rational numbers, $\mathbb{N}$ the set of natural numbers and $\mathbb{B} = \{True, False\}$ the Boolean domain. The time domain $\mathbb{T}$ is the set $\mathbb{Q}_{\geq 0}$ of non-negative rational numbers. The state of the system is described by a set of n state variables $V = \{x_1, x_2, x_3, \ldots, x_n\}$, where $n \in \mathbb{N}$. "*A state is a snapshot or instantaneous description of the system that captures the values of the variables at a particular instant of time*" [17]. Valid values of V are in the domain of $\mathbb{Q}$.

Figure 1.1: System under observation.

Definition 1 (Signal Trace σ)**.** *A signal trace (or trace) σ is the behaviour (state) of the system over a finite or infinite subset of $\mathbb{T}$. For a given trace σ, we define $\sigma(t_i) = x_i[t]$ as the value of the state variable x_i, $x_i \in V$ at a time instance t.*

Assumption 1 (Discrete-time signals)**.** *We assume the system is sampled such that the result of applying the formula on the discrete-time signal trace is equivalent to applying it on the continuous-time signal. For example we sample the system in such a way that every relevant event in the system is included within the trace.*

Assumption 2 (Finite length trace)**.** *It is important to note that there are certain properties that cannot be evaluated over a finite length trace. Example of such a property is one that checks "traffic light is green infinitely often", which requires an infinite length trace to analyze that it is satisfied. RuSTL is designed to analyze properties that can be evaluated over a finite length trace.*

Figure 1.2 provides a high-level overview of RuSTL. A STL specification φ is provided to a compiler program, which tokenizes the input string, and outputs a parse-tree. The algorithm traverses the parse-tree and generates a *monitor program*, which is a stand-alone script, which when executed, takes as input a signal trace σ and checks if the trace satisfied or violated the property and outputs the appropriate verdict over the Boolean domain $\mathbb{B} = \{True, False\}$ respectively.

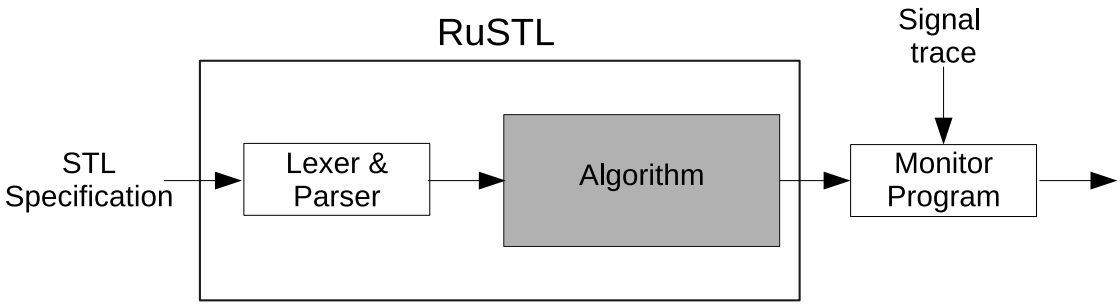

Figure 1.2: Abstract overview of RuSTL.

Effectively

The following metrics will be used to demonstrate that RuSTL executes its tasks effectively:

Definition 2 (Soundness)**.** *RuSTL will be deemed sound, if and only if, when given a valid trace σ and STL specification φ the monitor program always computes the correct answer, regarding the satisfiability or violation of the property.*

Definition 3 (Termination)**.** *RuSTL will be considered terminating, if and only if, when given a valid trace σ and STL specification φ the monitor program always halts.*

Definition 4 (Completeness)**.** *RuSTL will be deemed complete, if and only if, when given a valid STL specification φ, the Algorithm accepts any output generated by the Parser and creates a monitor program.*

Efficiently

Computation Time - This is the only metric used to measure the efficiency of the application. Time elapsed between the start of the *monitor program* (after loading the trace) till the end of execution (when the process halts and outputs the result). Ideally, the monitor should output the result as quickly as possible, so lower computation time values are desirable.

The complexity of the formula and the length of the trace heavily influence the performance of the system and are subjective to the user. So metrics such as memory allocation and CPU consumption are not considered to chart the performance of the application.

Violate a Property

The evaluation function $\theta : \sigma \times \varphi \times t \rightarrow \mathbb{B}$, maps a given trace σ, a valid STL formula φ and a time t to a value in the domain of $\mathbb{B}$. If $\theta(\sigma, \varphi, t)$ evaluates to *True* then the property was satisfied, and *False* means that it was violated.

1.4 Offline vs. Online Monitoring

The main focus of RV is to detect violations (or satisfaction) of correctness properties [9]. The detection can be achieved either offline or online. In offline RV, the process of verification starts after the *entire* trace σ has been collected "*(i.e., traces starting at time 0, and lasting till a user-specified time horizon)*" [19].

In online RV, the monitor runs in parallel to the running system, checking the currently running execution of the system. If the trace σ violates the correctness property, the monitor raises the alarm. In some cases, after the monitor raises the alarm, some entity may use this signal to begin the process to rectify the system. In some cases, a safety program may be triggered by the alarm to start the process to stabilize the system. Since the monitor is checking a running system, it is unknown how far into the future will the system keep running. So the property is always being monitored until either the system shuts down or the trace violates the property.

In embedded real-time systems, resources such as memory (non-volatile or volatile) are much smaller compared to everyday devices such as laptops or desktops. The main concern when doing monitoring online is to analyze its effect on the performance of the embedded device. The temporal depth requirement of the trace σ and the complexity of the property directly affect the computation time and memory allocation. For online RV the performance of the running system should be affected in the least amount possible. Whereas for offline monitoring the complexity cost can be passed on to offsite servers (e.g., *supercomputers*), that have more than enough resources and have been optimally designed to perform such computations.

1.5 Contribution

The main contribution of this work is RuSTL, an RV application using STL, which performs an offline qualitative analysis of STL formulas that takes in input as:

- Structured English text - The text is then analyzed and converted into a STL formula φ.
- Valid STL formula φ - The result of analysing the valid STL property is the generation of a stand-alone *monitor program*. During the generation of the *monitor program* RuSTL collects specifics regarding the formula φ and makes it available to the user. The collected information includes all the signals required for the formula φ, its temporal depth (the maximum time window that a property may require to compute a verdict) and general stack information.

  The generated *monitor program* is independent of the application and can be deployed headless on different platforms. The *monitor program* is currently generated as a Python program, but can easily be extended to other languages such as C or C++. The *monitor program* can then be run by providing it with a valid trace σ.

The novelty to our approach is that the created *monitor program*, is a stand-alone python script. Thus the *monitor program* can be deployed on any system that supports python, independently of RuSTL. To the best of our knowledge, there is no open-source implementation available that offers this feature.

After the *monitor program* analyzes a trace σ, it gives a binary response stating whether the trace satisfied or violated the property. But, an observer ensuring the system is running *safely* would like a more in-depth diagnostic alongside the *True* or *False* verdict. The verdict alone does not tell where in the sequence of events the property failed and by how much. To that end, RuSTL provides some diagnostic capability in the form of an interactive HTML plot. The tool takes the output from the *monitor program* and plots the checked data and also outputs the violated trace section.

1.6 Organization of the Thesis

The rest of the thesis is organized as follows. Chapter 2 gives background on TL, leading up to STL. In Chapter 3, we present the background information related to ANother Tool for Language Recognition (ANTLR), the tool used for parsing and lexing grammars.

In Chapter 4 and 5 we introduce the proposed solution that takes either structured text or an STL property and creates the corresponding checker. We explain its inner workings and present a proof for its effectiveness.

In Chapter 6 we describe a case study for implementing checks on data collected from a real-time self-driving autonomous vehicle and discuss the results.

Chapter 7 describes the related work done in the field. Finally, concluding remarks and discussion related to future work are in Chapter 8.

Chapter 2

Background

Temporal Logic [68] (TL) was first introduced in by Pnueli in 1977. It is a formalism for specifying propositional time properties of reactive and concurrent systems. Temporal logic has been studied extensively since its conception [10, 16, 53, 4].

TL can be divided into two categories:

- Linear Temporal Logic [68]
- Branching Time Logic

The first section in this chapter will explain the Linear Temporal Logic (LTL) specification and gives a few useful examples. Later sections will cover Metric Interval Temporal Logic (MITL), used to analyze real-time systems of Boolean signals and Signal Temporal Logic (STL) which is used to monitor dense-time real-valued signals.

2.1 Linear Temporal Logic (LTL)

Linear Temporal Logic [68] (LTL) specifications are checked on a finite set of Atomic Propositions (AP) over an infinite length trace σ. A trace can be either a particular execution of a program or data collected from sensors. As the name applies, in LTL there is a single sequence of events or system-states. Whereas the branching time logic, analysis properties over a computation tree model. Since the focus of this work centers on Runtime Verification (RV), we will not be looking into branching time logic and only focus on LTL.

The grammar for LTL can be expressed as follows:

$$\varphi := \top \mid p \mid \neg\varphi \mid \varphi_1 \wedge \varphi_2 \mid \bigcirc \varphi \mid \varphi_1 \,\mathcal{U}\, \varphi_2 \tag{2.1}$$

where

- φ is a temporal formula
- p is an AP from a pre-defined finite set
- $\neg$ is the basic logical operator for negation
- $\top$ is the Boolean operator True
- False ($\bot$) can be represented as: $\bot = \neg\top$
- $\wedge$ is the basic logical operator for conjunction
- The $\bigcirc$ or *next* is a basic temporal operator that specifies what should hold true in the next step
- $\mathcal{U}$ is a basic temporal operator for *until*, which specifies that the first formula should hold until the second formula becomes true

$$\sigma \models \varphi_1 \,\mathcal{U}\, \varphi_2 \leftrightarrow \exists i = 0, 1, \ldots \text{ such that } \varphi_2 \text{ hold true and } \forall 0 \leq j < i : \ \varphi_1 \text{ hold true} \tag{2.2}$$

Additional logical operators which build on top of $\neg$ (negation) and $\wedge$ (conjunction) are:

- $\vee$ is the logical operator for disjunction

$$\varphi_1 \vee \varphi_2 \equiv \neg(\neg\, \varphi_1 \wedge \neg\, \varphi_2) \tag{2.3}$$

- $\rightarrow$ is the logical operator for implication

$$\varphi_1 \rightarrow \varphi_2 \equiv \neg\varphi_1 \vee \varphi_2 \equiv \neg(\varphi_1 \wedge \neg\, \varphi_2) \tag{2.4}$$

- $\leftrightarrow$ is the logical operator for equivalence

$$\varphi_1 \leftrightarrow \varphi_2 \equiv (\varphi_1 \rightarrow \varphi_2) \wedge (\varphi_2 \rightarrow \varphi_1) \tag{2.5}$$

Additional temporal operators are also defined as follows:

- $\Diamond$ is the temporal operator *eventually*. $\Diamond\varphi$ specifies that the formula φ will eventually hold true at some point in the future. It is derived from the basic temporal operators and can be expressed as follows:

$$\Diamond\varphi \equiv \top \mathcal{U} \varphi \tag{2.6}$$

- $\Box$ is the temporal operator *always*, while some literature also refers to it as *globally*. $\Box\varphi$ specifies that the formula φ will always hold. It is also derived from the basic temporal operators and can be written as:

$$\Box\varphi \equiv \neg\Diamond\neg\varphi \equiv \neg(\top \mathcal{U} \neg\varphi) \tag{2.7}$$

2.1.1 Precedence Order

The precedence order on the operators are defined already. The $\mathcal{U}$ temporal operator has a higher precedence than the logical operators of $\wedge$ conjunction, $\vee$ disjunction and $\rightarrow$ implication. The $\mathcal{U}$ temporal operator is right-associative, so the formula $\varphi_1\mathcal{U}\varphi_2\mathcal{U}\varphi_3$ and $\varphi_1\mathcal{U}(\varphi_2\mathcal{U}\varphi_3)$ are equivalent.

2.1.2 Examples

Figure 2.1 outlines a few basic examples in LTL. Each circle represents a state. A state marked with an alphabet signifies that the AP is true in that state. Unmarked states can have any arbitrary AP true or false in them and does

LTL has been widely used for specification and verification of programs, RV over observed system behaviour and model checking. Popular model checking tools such as SPIN [41], SMV [59] and NuSMV [14] all have roots in LTL. It has also been used in mobile robot applications for dynamic controller and path planning algorithms in [29, 47, 51]. There are also Numerous RV techniques for LTL properties have also been proposed [36, 38] and tools such as EAGLE [7] exist for public use.

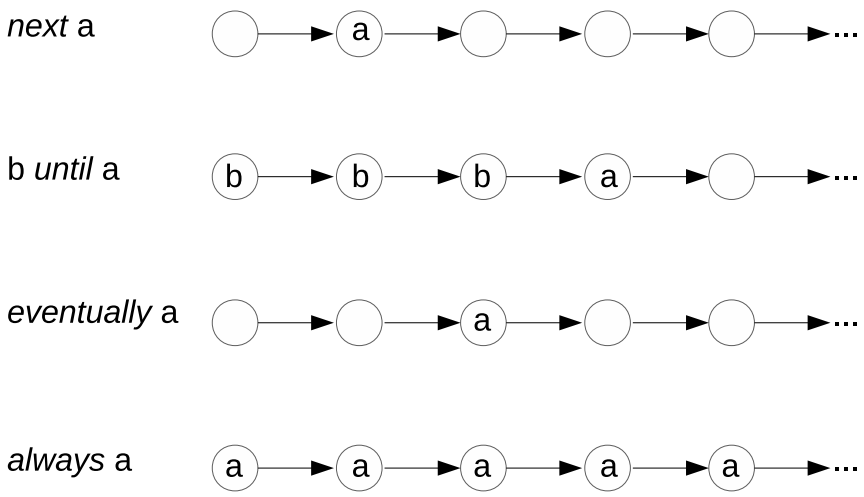

Figure 2.1: Sample example of LTL property evaluations.

2.2 Metric Interval Temporal Logic (MITL)

Metric Temporal Logic [50] (MTL) was introduced to analyze real-time systems of Boolean signals. Real-time systems such as vehicles, manufacturing plants, power stations, etc. have multiple sensory inputs during normal operation. These system needs to respond to these events asynchronously within a bounded time. While the domain of model checking and formal verification leveraged LTL extensively, there was an apparent lack of a formal specification method that involved reasoning about quantitative timing properties. Also, LTL properties are defined over infinite trace executions which is not the case when analyzing black box systems.

Many extensions have been proposed to the traditional LTL to deal with these issues in RV. One popular extension is MTL, proposed in [50] as a formal specification that subscripts a timing constraint on temporal operators.

Following are a few examples that are typical in the field of RV that would be difficult to express using the LTL specification:

- A request should always be followed by a response within at least 5-time units.
- When triggered the light should stay Amber for 3-time units and then turn Green.
- Each time someone opens the front door, someone will enter the correct security code

with the next 10 seconds or the alarm will start ringing for 5 seconds after the first 8 seconds have passed.

In [3] the authors intorduced Metric Interval Temporal Logic [3] (MITL). "*MITL is MTL where the timing constraints are not allowed to be singleton sets*" [21]. MITL restricts MTL by forbidding the use of single time constraint susbscripted to temporal operators and are instead susbscripted by time intervals.

Definition 5 (Time Interval I)**.** *A time interval is a nonempty convex subset of* $\mathbb{Q}_{\geq 0}$*. Intervals can be in the form of:*

- *Open* — $(a, b) = \{x \in \mathbb{Q} \mid a < x < b\}$
- *Half-open* — $[a, b) = \{x \in \mathbb{Q} \mid a \leq x < b\}$
- *Closed* — $[a, b] = \{x \in \mathbb{Q} \mid a \leq x \leq b\}$

We restrict ourselves to bounded timing interval $I = [a, b]$ *of finite length in the range of* $\subseteq [0, \infty)$*. An interval in the form of* $I = [a, a]$*, holds only a single value.*

The grammar for MTL is defined as follows:

$$\varphi := \top \mid p \mid \neg\varphi \mid \varphi_1 \wedge \varphi_2 \mid \varphi_1 \,\mathcal{U}_I\, \varphi_2 \mid \varphi_1 \,\mathcal{U}\, \varphi_2 \tag{2.8}$$

where

- A temporal operator with no interval equates to one for the entire time domain

$$\varphi_1 \,\mathcal{U}\, \varphi_2 \equiv \varphi_1 \,\mathcal{U}_{[0,\infty]}\, \varphi_2 \tag{2.9}$$

- The until formula $\varphi_1 \mathcal{U}_{[a,b]} \varphi_2$ equates to the following, where t, is the starting position:

$$\begin{aligned}(\sigma, t) \models \varphi_1 \,\mathcal{U}_{[a,b]}\, \varphi_2 \leftrightarrow \exists t^{'} \in [t + a, t + b] \text{ such that } \varphi_2 \text{ holds true} \\ \text{and } \forall t^{''} \in [t, t^{'}]\, \varphi_1 \text{ holds true}\end{aligned} \tag{2.10}$$

In this work the non-strict semantics of *until* $\mathcal{U}$ are employed, where both φ_1 and φ_2 hold true for some value $t^{'}$.

Thus the remaining temporal operators can be expressed as follows:

- The $\Diamond$ temporal operator for *eventually* is now expressed as $\Diamond_{[a,b]}\varphi$

$$\Diamond_{[a,b]}\varphi \equiv \top \mathcal{U}_{[a,b]}\varphi \tag{2.11}$$

  The formula φ will eventually hold true between $[t+a, t+b]$

- The $\Box$ temporal operator for *always* is now expressed as $\Box_{[a,b]}\varphi$

$$\Box_{[a,b]}\varphi \equiv \neg\Diamond_{[a,b]}\neg\varphi \equiv \neg(\top\, \mathcal{U}_{[a,b]}\, \neg\varphi) \tag{2.12}$$

  The formula φ will always hold true between $[t+a, t+b]$

Equation 2.13 is a valid property for MTL but not MITL, that checks that it is always the case that whenever φ_1 holds true, φ_2 will eventually hold true in exactly a time units.

$$\varphi := \Box(\varphi_1 \rightarrow \Diamond_{=a}\, \varphi_2) \tag{2.13}$$

It has been shown in [39] that both MTL and MITL are equally as expressive, so the semantics for only MTL is defined. In MITL the temporal operators are subscripted by a time interval. Equation 2.13 can be re-written in MITL notation as follows, where the bounded time interval $[a, a]$ contains exactly one point:

$$\varphi := \Box(\varphi_1 \rightarrow \Diamond_{[a,a]}\, \varphi_2) \tag{2.14}$$

With this expansion on the temporal logic specification that has the notion timing constraints, it gives the user more expressive power allowing them to define properties MITL that were not possible with LTL, such as those that require exact time distance between events or a specified time distance between two separate event occurrences.

With the grammar now defined we look at the examples given at the start of the section and construct their MITL formulas

- It is always the case that a response within 5-time units should follow a request.

$$\Box(request \rightarrow \Diamond_{[0,5]} response) \tag{2.15}$$

- When triggered the light should stay Amber for 3-time units and then turn Green.

$$\Box(trigger \rightarrow \Box_{[0,3]} Amber \wedge \Box_{[4,4]} Green) \tag{2.16}$$

- Each time someone opens the front door, someone will enter the correct security code with the next 10 seconds or the alarm will start ringing for 5 seconds after the first 8 seconds have passed.

$$\Box(OpenFrontDoor \rightarrow (\Diamond_{[0,10]} SecurityCode \vee \Box_{[8,13]} AlarmRinging)) \tag{2.17}$$

2.3 Signal Temporal Logic (STL)

Signal Temporal Logic [56] (STL) was developed as an extension to MITL to monitor dense-time real-valued signals. STL is similar to MITL, with the added benefit that the specification is no longer bound to evaluate only AP. In STL, the execution trace σ is a set of time-stamped real-valued signals. Since the intent is to deploy this RV monitor in embedded applications such as autonomous vehicles, Control Area Network (CAN) operated devices or Cyber Physical Systems (CPS) systems, this will be helpful.

The grammar for STL is identical to that of MITL presented in Equation 2.8 on page 13.

$$\varphi := \top \mid p \mid \neg\varphi \mid \varphi_1 \wedge \varphi_2 \mid \varphi_1 \, \mathcal{U}_I \, \varphi_2 \tag{2.18}$$

- The temporal formula for bounded STL *until* $(\varphi_1 \, \mathcal{U}_{[a,b]} \, \varphi_2)$ is the same as presented in 2.10 on page 13, but the formulas can now include real-valued signals and not just AP.

$$\begin{aligned}(\sigma, t) \models \varphi_1 \, \mathcal{U}_{[a,b]} \, \varphi_2 \leftrightarrow \exists t^{'} \in [t+a, t+b] \text{ such that } \varphi_2 \text{ holds true} \\ \text{and } \forall t^{''} \in [t, t^{'}] \, \varphi_1 \text{ holds true}\end{aligned} \tag{2.19}$$

- The $\Diamond$ temporal operator *eventually* is expressed as $\Diamond_{[a,b]}\varphi$

$$(\sigma, t) \models \Diamond_{[a,b]} \, \varphi \leftrightarrow \exists t^{'} \in [t+a, t+b] \text{ where } \varphi \text{ holds true} \tag{2.20}$$

  The formula φ will eventually hold true between $[t+a, t+b]$

- The $\square$ temporal operator *always* is expressed as $\square_{[a,b]}\varphi$

$$(\sigma, t) \models \square_{[a,b]}\varphi \leftrightarrow \forall t^{'} \in [t+a, t+b]\ \varphi \text{ holds true} \tag{2.21}$$

  The formula φ will always hold true between $[t+a, t+b]$

The formal specification provides a way to monitor hybrid systems, where the input signals can be either Boolean or real-valued. Following are a few properties that can be expressed using STL:

- The braking system should disengage in the vehicle as soon as wheel locking is detected.

$$\square((Wheel\_Locking == True) \rightarrow \square_{[1,1]}(Braking\_System\_Disengage == True)) \tag{2.22}$$

- Vehicle should not exceed a speed of 100 within the first 5 time units

$$\square_{[0,5]}(speed \leq 100) \tag{2.23}$$

- Whenever the signal goes above 1, within 2-time units it should settle under 0.5 for at least 3-time units

$$\square((signal > 1) \rightarrow \Diamond_{[0,2]}(\square_{[0,3]}(signal < 0.5))) \tag{2.24}$$

Chapter 3

ANTLR

Computer language parsing is the first step in writing a compiler. There is no restriction to writing a parser from scratch, but this is a highly error-prone and complex task. It is often preferred to use an existing program that performs this step. Bison and yaac are examples of popular parser generators used today. Bison is commonly used in conjunction with flex, which performs lexing and tokenizing.

Runtime Verification using Signal Temporal Logic (RuSTL) uses ANother Tool for Language Recognition (ANTLR) [67], a tool written in Java that handles structured text. ANTLR is an ALL(*) parser generator, which means it takes a top-down approach when parsing. The tool takes in as input a grammar file (with an extension "*.g4"), create a parser which can recognize structured text input of the language. The application provides additional targets for languages such as C, C++, Python along with Java. Our application is written using Python bindings.

3.1 Grammar

A grammar is a formal language description of a set of rules defining how the language is structured. It is used to define the syntax of programming languages and compiler construction. There should never be any ambiguity in the grammar of what constitutes a valid input for the language. Fortunately, the grammar for STL has already been defined as shown in Equation 2.18. ANTLR accepts grammar defined in extended Backus-Naur form (EBNF) notation.

Grammar 3.1 is an example sample grammar taken from the online [72] repository, that recognizes simple mathematical expressions. The grammar is conveniently called *Expression*. The general notation for ANTLR is that lexical (tokens) rules are written in uppercase letters and are defined near the end of the grammar while parser rules are denoted using lowercase letters.

ANTLR employs the concept of alternatives by having them to the right-hand side of a given rule. An *alternative* is a choice between possible matches that the parser can make from its current position. Ideally, only one of the alternatives will match; however, there can be cases where multiple alternatives can match the rule, and in such instances ANTLR matches it to the first alternative.

It starts with a *prog* rule (short for program), that comprises of one or more *stat* (short for statements). A *stat* has three alternatives. The first being an *expr* (short for expression) followed by a newline. The second an assignment statement for an *expr* followed by a newline. And the last being just a newline. The *expr* rule has 5 alternatives. They follow the logic of mathematical expressions. The multiply and divide rule is defined before the addition and subtract, and so they will be matched first. The last alternative matches and expression encapsulated in parenthesis.

The following lists examples of valid inputs for the grammar:

- 3
- a = (3 + 1)
- b = 5
- c = c + 1
- a = b + 2

The grammar is just displayed here to understand the basics of ANTLR and is in no way a complete. For instance, the fourth bullet point is a valid syntax for incrementing the value of variable *c* by 1. However, before this line, the variable has not been initialized. ANTLR will raise no error flags, since the grammar accepts the input but the resulting logic will break down. It is then the programmer's job to look for these cases when parsing the input.

Grammar for expression statements

```
grammar Expression;

prog :    stat+ ;


stat::=
    expr NEWLINE                    # printExpr
    | ID '=' expr NEWLINE           # assign
    | NEWLINE                       # blank


expr::=
    expr op=('*'|'/') expr          # MulDivExpr
    |expr op=('+'|'-') expr         # AddSubExpr
    |INT                            # int
    |ID                             # id
    |'('expr')'                     # parensExpr


MUL : '*' ;
DIV : '/' ;
ADD : '+' ;
SUB : '-' ;
ID  : [a-zA-Z]+;
INT : [0-9]+;
NEWLINE: '\r'? '\n' ;
WS  : [ \t]+ -> skip ;
```

3.2 Lexer and Tokenizing

A lexer is a program that tokenizes a stream of text. The job of a lexer is to read the input character by character and performs lexical analysis —transforming plain text into a stream of tokens. There are at least two main characteristics of a lexar. First is to chunk the text according to the rules and second is identifying the *type* of the token. Other information associated with a token includes the line number, start index, stop index and text width.

Equation 3.1 is an example of a simple variable assignment statement that is valid for the grammar defined in Grammar 3.1.

$$a = b + 2 \tag{3.1}$$

The tokens generated by the program for the input provided in Equation 3.1 are:

- ID: a
- ID: b
- op: +
- INT: 2

3.3 Parser

A parser then recognizes the sentence structure. The parsers input is the stream/sequence of tokens. The parser then checks the meaning of the tokens, analyzes their relative position and the context in which they appear. It is the job of the parser to determine if the structure of the sequence of tokens is valid with respect to the grammar of the language. The resulting output from a parser is a tree-like structure called a parse tree.

ANTLR is a recursive-descent parser, meaning that "*this is a specific kind of top-down parser implemented with a function for each rule in the grammar*" [66]. In Figure 3.1 nodes of the tree are labeled with the parser rule whereas the leaves of the parse tree are the input tokens. They correspond to the rules defined in the grammar shown in Grammar 3.1.

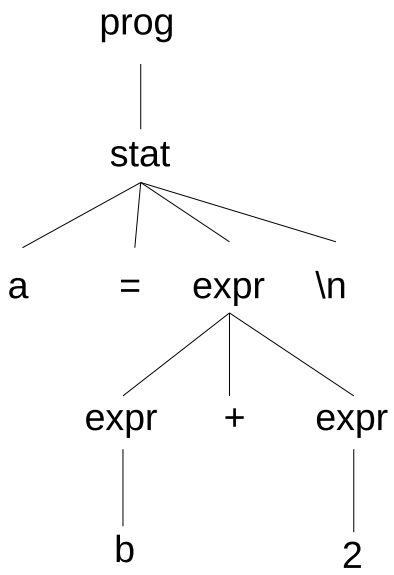

Figure 3.1: Parse tree generated after analyzing the input.

3.3.1 Avoiding Ambiguity

There is ambiguity in the grammar if an input string provided to a grammar results in more than one parse tree generated by the parser. Having ambiguity while defining the grammar can lead to unexpected and unintended consequences as it becomes more complex and supports a wide array of interconnected behavior. For a grammar with any uncertainty can lead to a scenario where multiple alternatives match with a single string input. These faults are not easy to identify by just viewing the grammar, and an extensive battery of user-provided test cases should be run to validate the claim that the grammar is unambiguous.

An unambiguous grammar is one whose parser generates a unique parse tree when provided with a valid input string. ANTLR deals with such ambiguities by matching the input to the first rule specified in the grammar. So these points must be taken into consideration when designing a grammar.

3.4 AST Listener

An *expr* rule has multiple alternatives, and so multiple nodes in Figure 3.1 are labelled with this rule. Once the parser creates the parse tree, ANTLR, provides a walking method to traverse it. The *ParseTreeListener* implementation provided by ANTLR creates an *entry* and *exit* method for each rule. If the alternatives for a rule are named, then ANTLR creates an individual entry and exit methods under those names.

The walker then starts at the root node and visits each of the children nodes in the tree. The mechanism works automatically, and the user does not have to call the methods explicitly. So when the method of *AddSub* is entered, it will have two *expr* child nodes that can be accessed explicitly.

ANTLR ensures that all rules (or alternatives) have an *entry* and *exit* function associated with them. And the appropriate method is called while walking the parse tree. The class method provides a context for of which rule (or alternative) ANTLR matched from the input.

Chapter 4

RuSTL

The entire workflow of RuSTL is outlined in Figure 4.1. The user has the option to either provide a valid STL formula φ or structured text as input. If the latter is provided, it is converted into an STL formula φ and will be presented in more detail in Chapter 5. If the input is an STL formula φ, the grammar specification analyzes it and generates a parse-tree. RuSTL performs some optimizations and updates the parse-tree, which is then traversed to create the *monitor program*.

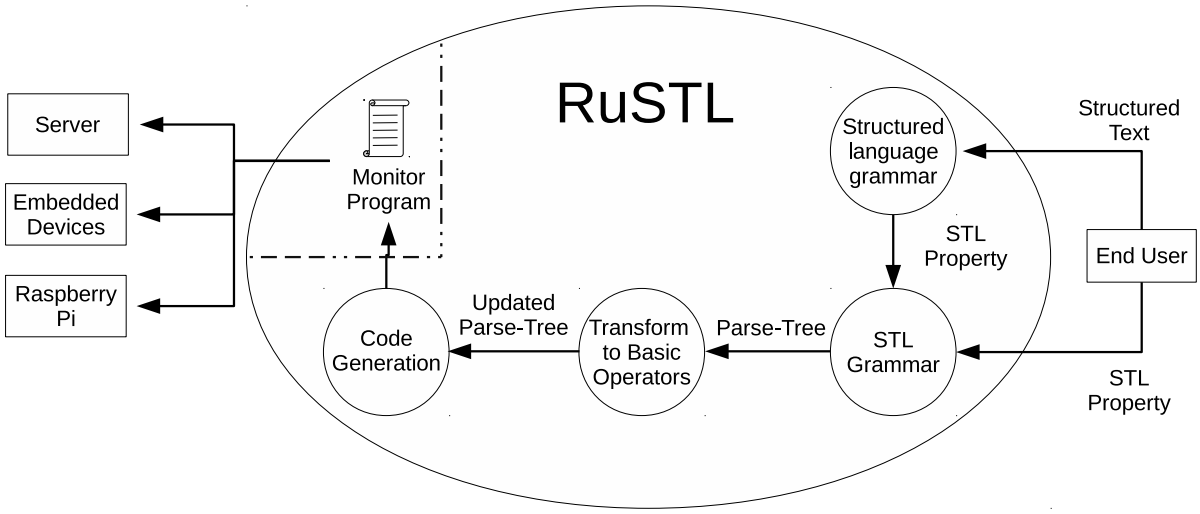

Figure 4.1: Workflow diagram of RuSTL.

This chapter first outlines the grammar as supplied to ANTLR, and explain its key components. Then moves on to describe how RuSTL traverses the parse-tree generated by

the parser and the logic used to create the *monitor program*. RuSTL makes the following two assumptions regarding the input trace σ and the correctness property φ being checked respectively:

Assumption 3 (Time Column)**.** *RuSTL makes the explicit assumption that the time values column of the trace σ is labelled as "Time", where each subsequent entry is a linearly-ordered unique ascending $\mathbb{Q}_{\geq 0}$.*

Assumption 4 (Future Bounded STL Properties)**.** *It is assume that the interval I subscripted to the temporal modalities are in the form of $[a, b]$. Where $0 \leq a \leq b$ and $a, b \in \mathbb{Q}_{\geq 0}$.*

4.1 STL Grammar

Defining the grammar is the first step. The designed grammar takes as input any valid STL property and creates a unique parse-tree. The basics of defining a grammar using ANTLR have already been discussed in Chapter 3, so this section will cover the critical components of *stlgrammar* and explain the design in detail. The entire grammar is available in Appendix A.

4.1.1 Expression Rule

We will start defining the grammar from the base and work way up to the complete STL formula. The grammar should be able to process expressions that are part of the formula φ to monitor real-valued signals. The following lists examples of valid expression in formulas for the grammar:

- 63
- 4/63
- $(4/63 * (4 + 3))$
- $4/63 * 4 + 3$

As discussed in Section 3.3.1, any ambiguity in the defined grammar can lead to parsers generating parse trees which are in-line with the defined grammar but not correct according

Expression grammar

```
expr::=
    expr op=('*'|'/') expr          # MulDivExpr
    |expr op=('+'|'-') expr         # AddSubExpr
    |REAL                           # realExprs
    |'(' expr ')'
```

to the specification. Since ANTLR matches the tokens to the first alternative in a rule, the implementation ensures that the multiply (*) and divide (\) operation have the same priority and need to be matched before the addition (+) and subtraction (-) operator. Grammar 4.1.1 shows the complete definition.

A rule alternative can have the rule itself as part of the definition. Figure 4.2 shows the parse tree generated by the parser for the input expression $4 + 3 * 7$.

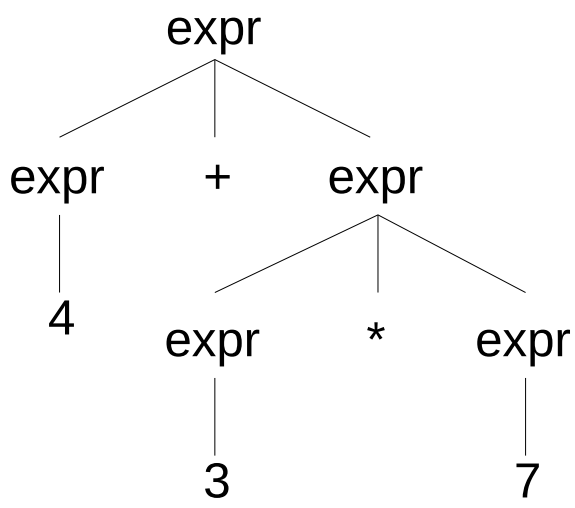

Figure 4.2: Parse tree generated for: $4 + 3 * 7$.

Signal comparison grammar

```
signalComp::=
    signal relOp expr          # signalExpr
    |expr relOp signal         # signalExpr
    |Bool relOp signal         # signalBool
    |signal relOp Bool         # signalBool
```

4.1.2 Signal Comparison Rule

Grammar 4.1.2 relates to signal comparison expressions that are accepted by the grammar. The following lists examples of valid inputs for the grammar:

- $a == True$
- $4/63 >= a$
- $(4/63 * (4 + 3)) != b$
- $(False != a)$

The time-stamped real-valued signal is to be on one side of the relational operator, while the other is either an expression or a Boolean value. Do note that multiple alternatives have the same label in the grammar. The inputs $a == True$ and $4/63 >= a$ are handled differently. The prior expression compares a signal to a real, while the latter to a Boolean string value. So $4/63 >= a$ and $a <= 4/63$ are associated with the same alternative method even though they are distinct inputs however the logic associated with them remains the same.

As an added step, the *entry* methods for the *signalComp* updates a unique list relating to all the signals used in the formula φ. It does this by verifying if the signal variable is already in the global signal list and appends the signal to the list if that is not the case. This step has no bearing on the generation of the *monitor program*. The tool uses this information in labeling the diagnostic plots after traversing the entire parse tree. The specifics of the diagnostic plots are discussed in Section 4.3.3.

STL formula grammar

```
stlFormula::=
    stlFormula 'U' timeSlice stlFormula      # stlUntilFormula
    |stlFormula implies stlFormula           # stlFormulaImplies
    |stlFormula andorOp stlFormula           # stlConjDisjFormula
    |NOT stlFormula                          # stlNotFormula
    |'G' timeSlice? '('stlFormula')'         # stlGlobalFormula
    |'F' timeSlice? '('stlFormula')'         # stlEventualFormula
    |signalComp                              # stlSignalComp
    |signal                                  # stlSignal
    |Bool                                    # stlProp
    |'('stlFormula')'                        # stlParens
```

4.1.3 STL Formula Rule

Section 2.1.1 outlines the precedence order of the operators and Grammar 4.1.3 follows the same rules while defining the alternatives for what constitutes an STL formula. The following lists examples of valid STL formula's φ for the grammar:

- $G[0,10]((pb == 1) \rightarrow F[1,2](qb == 1))$
- $G[3,5](F[2,5](x > 0)) \rightarrow (y > 0)$
- $(not(x > 9)\, U[10,15]y > 25) \rightarrow (a! = 3)$
- $(x > 12)\, and\, x > 9\, U[10,15]y > 25\, and\, (z[t] >= 25)$

4.2 Transforming Properties

Equation 2.18 on page 15 shows the basic temporal and logical operator to form any correctness property. The operators for $\Box$ (global) and $\Diamond$ (eventually), $\rightarrow$ (implies) and $\vee$ (or) can be generated using the basic operators.

The additional operators make the process of generating the properties easier for the user and help them understand the properties intuitively. In Equation 4.1, all three formulas are equivalent and will check that "during the first 30 time units the value of x remains

below 100". Although all three formula's are performing the same check, the first method of writing the formula is more compact.

$$\square_{[0,30]}(x < 100) \equiv \neg\Diamond_{[0,30]}\neg(x < 100) \equiv \neg(\top\, \mathcal{U}_{[0,30]}\, \neg(x < 100)) \tag{4.1}$$

De Morgan's Laws [73] outlines the principles where propositions are related to their negations and disjunctions. The transformation to the input adheres to those rules. There are two reasons for performing the transformation. Firstly, this reduces the number of alternatives for the *stlFormula* rule that will be called by the *stlgrammarListener* class. The *entry* and *exit* methods for all affected alternatives are skipped resulting in a leaner code which reduces debug times.

Secondly, when going through the proof that the *monitor program* generated by the application is sound and terminates in Section 4.4 these additional operators case will not have to be analyzed. The grammar supports the full syntax of the formal specification and accepts all valid formula φ, but the transformation helps reduce the complexity associated with proving the completeness of RuSTL.

The main focus of this intermediate step is to take a correctness property as a string input and restructure the formula so that it only uses the basic operators. If a given input formula provided by the user only consists of the basic logical and temporal operators, then the output of this step will be identical to the input.

A bottom-up approach on the parse tree achieves the desired outcome. For any given property, such as $\square(\varphi)$, there is no restriction as to how nested the formula φ can be. The *entry* method for any rule will have no inherent knowledge of the depth of the tree from its node if the parse-tree is traversed using a top-down approach.

By using only the exit methods for the rules, the callbacks will start from the leaf (token) nodes in the parse-tree. The nodes are annotated with the transformed formula and exit its callback, traversing up the tree is continued in this manner. The number of its children and their type are already known for each alternative's callback and can be accessed explicitly.

Only the non-basic type operators need to be altered leaving the remaining nodes untouched. Any rule that is not an alternative *stlFormula* gets annotated with the text that the parser associated with it by default.

Whenever an *exit* method for a node is reached, it is explicitly known that the sub-tree below that node has already been analyzed and the information annotated on the children's node. Figure 4.3 shows the parse tree that would be generated by taking as input Equation 4.2.

$$G[0, 10](x < 10) \tag{4.2}$$

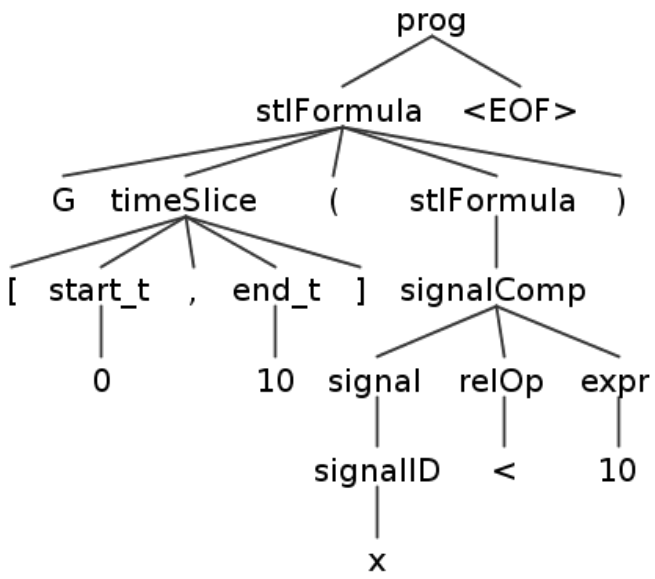

Figure 4.3: Parse tree generated for Equation 4.2.

In Figure 4.3, the *signalComp* node will be annotated with the expression computed so far: $x < 10$. The parent of the *signalComp* node is the *stlFormula* rule but is not the alternative that needs to be altered so will be annotated with the entire sub-tree without modifications. The first *stlFormula* node, that is the child of the *prog* node, is matched to the *stlGlobalFormula* alternative and is one of the operators to be transformed. The child *stlFormula* of this node is already annotated with the sub-tree and and thus mutates the operator according to the rule shown in Equation 2.7. The resultant output formula and parse-tree is shown in Equation 4.3 and Figure 4.4 respectively.

$$not(True\, U[0, 10](not(x < 10))) \tag{4.3}$$

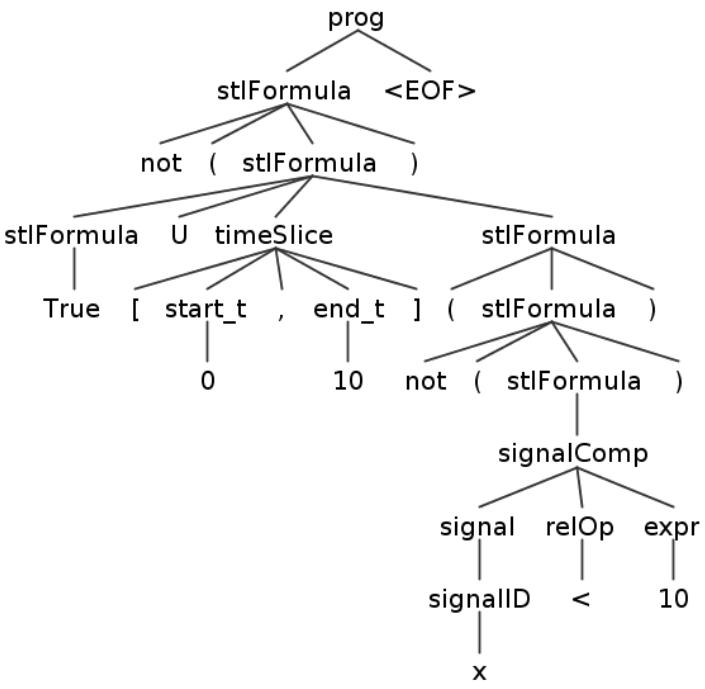

Figure 4.4: Parse tree generated for Equation 4.3.

4.3 Generating the Monitor Program

When generating the *monitor program*, the general approach is to create a function call for each alternative of the *stlFormula*, *signalComp* and *expr* rule. Each alternative for these rules has a code template associated with it. At each *exit* callback in the parse tree, the template is updated with the specific details. The logic then appends the function to the *monitor program* file.

This function-based template approach has multiple advantages. Calling any specific function is equivalent to checking a sub-formula from that node in the parse tree. When checking complex and highly nested properties, this allows the user to verify which sub-rule violated or satisfied.

Furthermore, since the functions are based on pre-defined templates, the output code becomes as lean as possible. This strategy allows the execution of the *monitor program* to be run on a variety of different platforms independently of the application.

4.3.1 Entering a Node

Every time an *entry* method for the *stlFormula* the *signalComp* is called, a unique function name is associated with that node. The function name is a concatenation of a string prefix (the rule name) and a unique integer value.

Here the application also performs some general housekeeping, such as ensuring the end time is not less than the start time in an interval I. RuSTL updates a list of all the unique signals that the property φ requires.

4.3.2 Exiting a Node

At the *exit* callback the function template is populated with the unique specifics. The function then gets appended to the *monitor program*. For each node, there is a separate function that will exist in the final output. The only parameter required by the function call is the time t. The trace is a global variable and can be accessed through all functions. Additionally, any call in the form of $func\_\varphi_1(t)$ || $func\_\varphi(t)$, refers to a function that evaluates φ_1 || φ formula respectively at time t.

Exiting expr

In the *expr* rule, the node of the parse-tree gets annotated with the complete expression processed from that node as the root till the token leaves. A simple python dictionary, with the node address as the key, is employed to store and access the information. The same strategy is used throughout the application to annotate nodes with relevant data that a parent or child node might require. The *exit* method for the alternative encapsulates the expression in brackets to ensure precedence order. So for an input that contains the expression $4 + 3 * 7$, the *monitor program* checks $(4 + (3 * 7))$.

Exiting signalComp Alternate

When exiting a *signalComp* method, for any of the two alternatives that get matched, there will always be three children nodes. Of the three, two will be the relational operator and the signal name. The last child will either be an expression or a Boolean value.

Algorithm 1 shows the function template that will be generated for the *signalComp* rule, when the alternative is *signalExpr*. The logic for the *signalBool* alternative is similar.

Algorithm 1: *signalComp* Function Template

```
   Input: t
   Result: True or False
1  if signal[t] relOp expr then
2  |  return True
3  else
4  |  return False
5  end
```

Exiting stlSignal Alternate

The *stlSignal* or *stlProp* alternate of the *stlFormula* are easily mapped in a single function since no temporal operator is associated with those cases. The resulting template is outlined in Algorithm 2:

Algorithm 2: *stlSignal* Function Template

```
   Input: t
   Result: True or False
1  if signal[t] == True then
2  |  return True
3  else
4  |  return False
5  end
```

Exiting Disjunction Alternate

Due to the pre-processing done on the input string as described in Section 4.2, a callback related to the $\Box$ (globally), $\Diamond$ (eventually), $\rightarrow$ (implies) and $\vee$ (or) will never be entered. If the alternate for *stlConjDisjFormula* is reached, it will have three children, two of which will be properties φ_1 and φ_2, and the remaining child will be the operator, in the form of $\varphi_1 \wedge \varphi_2$, and it can be implicitly assumed that the operator will be an $\wedge$ (and) operator. At Line 1 of Algorithm 3, $func\_\varphi_1(t)$, refers to the function that evaluates the φ_1 property at time t. As outline in Section 4.3.1 all the relevant nodes are annotated by a unique function call. The callback method here while exiting can access that value since the children have

all been assigned this information. So $func\_\varphi_1(t)$ is being passed the same time value t that the *stlConjDisjFormula* function received and the same is true for $func\_\varphi_2(t)$.

Algorithm 3: *stlConjDisjFormula* Function Template: $\varphi_1 \wedge \varphi_2$

Input: t
Result: True or False
1 **if** *(*$func\_\varphi_1(t)$ *and* $func\_\varphi_2(t)$*) ==* *True* **then**
2 | return True
3 **else**
4 | return False
5 **end**

Exiting Negation Alternate

The *stlNotFormula* case, shown in Algorithm 4, is similar to the one presented for *stlConjDisjFormula* alternative. It will *not* the result received by the $func\_\varphi(t)$ call and check the result.

Algorithm 4: *stlNotFormula* Function Template: $\neg\varphi$

Input: t
Result: True or False
1 **if** *not(*$func\_\varphi(t)$*)* **then**
2 | return True
3 **else**
4 | return False
5 **end**

Exiting Until Alternate

The *stlUntilFormula* alternate method shown in Algorithm 5 is the only case where a temporal operator is involved. The rule has three children. Two of them are the subformulas φ_1 and φ_2, while the last child is the time interval. Assumption 4 addresses the requirement of future bounded properties by restricting the values of the interval I. Values of the interval, $[a, b]$ are accessed, and are checked to ensure that both are greater than zero: $a, b \geq 0$, and that b is never less than a: $b \geq a$. An interval [6,3], will be accepted by the grammar but will correctly fail this check.

The Boolean variable initially gets set to *False*. This variable denotes the state of the property in the check so far, and ensure that the function always returns a valid output. Line 4 computes the dense-time range at which φ_2 will be evaluated. The array holds all time-stamped values in the range $[t + a, t + b]$. The for-loop at Line 6 iterates through the array and checks the result. If the $func\_\varphi_2(t)$ does not hold true for any value in the dense-time range then the Boolean value stays *False*. But if $func\_\varphi_2(t)$ holds true for any value in the array, the variable is set to *True* and break from this for-loop.

Next, the value of the Boolean variable gets evaluated at Line 14. If it is *False*, it indicates that φ_2 did not yield *True* in the required range and as such the property could not be satisfied. If however the check is passed, φ_2 holds true at some time value between $[t+a, t+b]$, and that is stored in the variable i. The next step is to calculate the dense-time range for φ_1.

According to the grammar, φ_1 needs to hold *True* at all time indexes in the range $[t, i]$. The for-loop at Line 17 then iterates through the array and evaluates $func\_\varphi_1(t = j)$ and if any return value evaluates to *False*, updates the Boolean variable with that answer and breaks from the for loop. Finally, the Boolean variable gets evaluated at Line 25, whose value signifies if the property is violated or satisfied.

Algorithm 5: *stlUntilFormula* Function Template: $\varphi_1 \, \mathcal{U}_{[a,b]} \, \varphi_2$

Input: t
Result: True or False

```
/* Initialize the Boolean variable to False                          */
until_check ← False
/* A list of time-stamps from the trace that fall in the range       */
dense_time_range_phe = [t+a, t+b]
/* Iterate over the list of states that are within the time range    */
for i in dense_time_range_phe do
    /* if func_φ2(t = i) equals to true for any value in the list, set
       the Boolean variable to True, and break from the loop         */
    if func_φ2(t = i) holds true then
        until_check ← True
        break
    end
end
/* if the Boolean variable is True, it means that func_φ2(t = i) did
   result True for some value in the time range of [t+a, t+b]        */
if until_check == True then
    /* calculate the subset of the first list till the time where
       func_φ2(t = i) resulted in True                               */
    dense_time_range_phi = [all values in dense_time_range_phe ≤ i]
    for j in dense_time_range_phi do
        /* Iterate over the second list and if func_φ1(t = j) returns
           False for any value, set the Boolean variable to False and
           break from the loop                                       */
        if not (func_φ1(t = j) holds true) then
            until_check ← False
            break
        end
    end
end
if until_check == True then
    return True
else
    return False
end
```

4.3.3 Diagnostic Plot

Presently RuSTL only provides a qualitative result in the form of *True* or *False* for any valid STL formula. If the property being checked is in the form of $\varphi_1 \wedge \varphi_2$, a Boolean *True* or *False* verdict does not give any insight into which sub-formula was violated. For that reason a diagnostic component is added to RuSTL, which allows the user to gain more insight regarding the sequence of event.

As outlined in Section 4.3, each node in the parse tree is assigned a unique function call. That unique name is appended to a global list. When the *monitor program* is created, RuSTL also outputs a dictionary, whose keys are the function call and the value associated with each key is the sub-formula that function checks. The *monitor program* also has these function call names available to it in the final output.

During the execution of the *monitor program*, each function updates the global data frame with its result, under the function name column, at time t in the trace. The results of the check are stored in a Comma-separated values (CSV) file. The CSV can then be used to plot the results by the user. RuSTL provides a way to plot the results using Plotly, an open-source JavaScript charting library. The plots are interactive, allowing the user to focus only on the desired signals and timeframes.

The graphing capability causes over-head in terms of memory usage and computation time, and therefore RuSTL provides a way to set this feature *on* or *off* depending on their need.

4.4 Soundness, Termination and Completeness

An algorithm is considered to be *sound* if the answer computed by the said algorithm is always correct. It should not ever yield the wrong answer. While, an algorithm is considered to *terminate* if, for a given valid input it will always halt.

Take as an example of an algorithm which sorts a finite unsorted array of integers. If an algorithm always sorts any finite unsorted integer list correctly, it can be proved to be sound. If an algorithm always halts for any given input within its domain, it is proved to terminate. If the algorithm returns a sorted array when provided with positive integer value array, but does not halt when the list contains negative values, then the algorithm is sound but does not terminate.

RuSTL handles any valid STL formula φ as input and generates a unique parse-tree followed by a *monitor program*. If the input is not a valid STL formula φ, it is not accepted

and terminates with a warning message. Due to the pre-processing done on the input as described in Section 4.2, soundness and termination is proven over the basic operators only.

As a reminder, for time an alternate for the *stlFormula* is being exit from the parse-tree in the logic, an associative function call is added to the *monitor program*. Each function requires a single input t, which is the time at which that sub-formula is being checked. The trace however is a global variable and can be access by all functions. Additionally, any call in the form of $func\_\varphi_1(t)$ || $func\_\varphi(t)$, refers to a function that evaluates φ_1 || φ formula respectively at time t. As an example, the disjunction formula is written as $\varphi_1 \wedge \varphi_2$. It has two sub-formula φ_1 and φ_2 respectively. The calls $func\_\varphi_1(t)$ and $func\_\varphi_2(t)$ as presented in Algorithm 3 return the result for both these sub-formulas and the disjunction formula *and's* the output and returns the appropriate result. Furthermore, as outlined in Section 4.2, the algorithm transforms all $\Box$ (globally), $\Diamond$ (eventually), $\rightarrow$ (implies) and $\vee$ (or) operators from the input specification and transforms them using the predefined rules so only the cases for $\wedge$ (conjunction), $\neg$ (negation), and $\mathcal{U}$ (until) operators need to be handled for the *monitor program*.

Theorem 4.4.1 (Soundness)**.** *The monitor program is sound.*

Proof. There are three cases to consider.

- The disjunction $(\varphi_1 \wedge \varphi_2)$ template, as presented in Algorithm 3 only performs a single check at Line 1, evaluating the returns from $func\_\varphi_1(t)$ and $func\_\varphi_2(t)$. It passes the same time value t to both functions as provided to itself.

- The negation $(\neg\varphi)$ template, as presented in Algorithm 4 evaluates φ at time t and checks the negated result against *True* before returning it.

- The until $(\varphi_1 \, \mathcal{U}_{[a,b]} \, \varphi_2)$ template is presented in Algorithm 5. As part of the logic in the *exit* method, the interval values $I = [a, b]$ are checked for $a, b \geq 0$ and $b \geq a$. Since the temporal operator for until $\mathcal{U}_{[a,b]}$ is the only modality that looks into future time it also needs to be checked that the trace is sufficiently long enough. If $b < max\_time\ of\ trace$, then the logic may yield a wrong answer. However since this is dependant on the trace σ, it is left up to the user at this point to ensure the temporal-depth does not exceed the trace length. The logic presented in Section 4.5 shows how the temporal depth can be calculated and the application outputs the required temporal depth for the trace when generating the *monitor program*.

The use-case of the formula can be broken down into three subcategories to prove its soundness.

1. $func\_\varphi_2(t^{'})$ holds for no value of time $t^{'}$ in the range of $[t+a, t+b]$. In this case the property should yield as *False*.

   The dense-time array calculated at Line 4 in Algorithm 5, will have either finite positive increasing time-stamped values from the trace or yield will empty if no time-stamps exist within the range $[a, b]$. Either scenario is valid.

   If the evaluation of the Boolean value at Line 14 is *False*, that means that there was no time in the range $[t + a, t + b]$, where $func\_\varphi_2(t^{'})$ held *True* or the list containing the trace within the time range was empty.

   The logic will move to Line 25, which will evaluate to *False*, which is the correct behavior.

2. $func\_\varphi_2(t^{'})$ holds for some value of time $t^{'}$ in the range of $[t + a, t + b]$, and $func\_\varphi_1(t^{''})$ does not hold for some value $t^{''}$ in the range $[t, t^{'}]$. In this case the property should yield as *False*.

   If this scenario is reached then some value of $t^{'}$ yielded $func\_\varphi_2(t^{'})$ as *True*. So the Boolean check at Line 14 will hold true and the code will loop through the values between $[t, t^{'}]$ from Line 17. If for any value $t^{''}$, $func\_\varphi_1(t^{''})$ yields *False*, the Boolean variable is update to *False* breaks from the loop.

   The logic will move to Line 25, which will evaluate to *False*, which is the correct behavior.

3. $func\_\varphi_2(t^{'})$ holds for some value of time $t^{'}$ in the range of $[t + a, t + b]$, and $func\_\varphi_1(t^{''})$ holds for all values of $t^{''}$ in the range $[t, t^{'}]$. In this case the property should yield as *True*.

   For this scenario, $func\_\varphi_2(t^{'})$ holds for some value of time $t^{'}$, otherwise option 1 would be matched. And $func\_\varphi_1(t^{''})$ holds for all values in the range $[t, t^{'}]$, otherwise option 2 would have matched.

   So the Boolean variable will be *True*, during the final check at Line 25, which results in the formula equating to *True*, which is the correct result.

□

Theorem 4.4.2 (Termination)**.** *The monitor program always terminates.*

Proof. The same three cases are considered to prove termination.

- The disjunction function template in Algorithm 3 has no loops and as such will always terminate
- The negation function template in Algorithm 4 has no loops and as such will always terminate
- The until function template presented in Algorithm 5 contains two *for-loops*. To prove that this function always halts the logic should never run into a situation where the code runs infinitely.

    1. The *for-loop* at Line 6 iterates over the list of finite time-stamped states from the states computed between the range $[t+a, t+b]$ derived on Line 4. There is no recursion, and calls a function to compute the result of $func\_\varphi_2(t^{'})$. Hence the code will always exit the loop.
    2. The *for-loop* at Line 17 if reached goes over the subset of the finite time-stamped states computed at Line 16, to compute the result for $func\_\varphi_1(t^{''})$. Using the same logic as presented for the first *for-loop*, it is shown that the code will never run into an infinite running sate in this function template.

□

Theorem 4.4.3 (Completeness)**.** *The Algorithm that handles the parse tree and generates the monitor program is complete.*

Proof. As discussed in Section 4.2, the parser will only be handling cases for $\wedge$ (conjunction), $\neg$ (negation), and $\mathcal{U}$ (until) operators. Section 4.3 outlines the steps used by RuSTL to handle each of these cases and how the resulting check is implemented.

Grammar 4.1.3 specifies that any alternative may include a nested formula without restrictions, so the complexity of the input formula is not constrained. The final output handles recursion by making a single individual call to each instance of the rule included in the alternative. So thereby we can prove that the algorithm handling the parse-tree is complete. □

4.5 Temporal Depth

The satisfaction or violation of any future bounded STL property φ at time t depends upon future inputs $t^{'} > t$. The work presented in [57], describes the concept of *temporal depth* as the sufficient length of time required from any time time t to satisfy a property φ, which they denoted as $D(\varphi)$. The required depth is always relative to the formula φ. Equations 4.4, outlines their approach on how to calculate the temporal depth of a bounded STL formula φ.

$$\begin{aligned} D(p) &= 0 \\ D(\neg\varphi) &= D(\varphi) \\ D(\varphi_1 \wedge \varphi_2) &= max\{D(\varphi_1), D(\varphi_2)\} \\ D(\varphi_1 \, \mathcal{U}_{[a,b]} \, \varphi_2) &= b + max\{D(\varphi_1), D(\varphi_2)\} \end{aligned} \tag{4.4}$$

A trace σ that satisfies or violates a property φ may not require the entire depth of the trace and can output the result earlier, but will not need a longer interval than calculated by the temporal depth to process the satisfiability. An input trace σ shorter than the calculated temporal depth may lead to the incorrectly evaluating the satisfiability of the property. The user can use the computed depth value to ensure that the trace σ is sufficiently long. Equations 4.5 shows the calculated temporal depth for a formula which checks that eventually during the first 10 time units the $safeState$ holds for at least 2-time units.

$$D(\varphi) \equiv D(\Diamond_{[0,10]}(\Box_{[0,2]}(safeState == True))) \equiv 12 \tag{4.5}$$

Chapter 5

Structured English

The formal specification processes have been around a long time and are widely used in industry. But while the logic behind TL and its variants are well understood, there is often difficulty in creating these properties. It is an accepted fact that developing formal specifications is an error-prone task [21]. Specifying TL properties is difficult for someone not intimate with the field and even the experts can have difficulty describing adequately what a rule might be trying to accomplish by just reading the property. When looking at the property specified in Equation 5.1 it is not immediately clear what the property aims to monitor. However, it is a common property to check the behavior of stabilizing spikes in a system. It verifies that whenever the signal value exceeds the value 1, it should eventually within the next 2 time units remain under 0.5 for at least 3 time units.

$$\square((signal > 1) \rightarrow \Diamond_{[0,2]}(\square_{[0,3]}(signal < 0.5))) \tag{5.1}$$

System requirements can be generated from multiple sources such as engineers, managers, sales, etc. targeted to monitor different aspects of the system. But the assumption cannot be made that the engineers or developers will be familiar with specifying and/or interpreting of TL properties. Often enough even engineer with a mathematical background face problems trying to compose TL specifications [34].

So we need ways for the end-user to leverage these formal specification based frameworks without the ambiguity. Multiple techniques [40, 5, 35, 60, 62] have been developed that use parsing, Graphical User Interface (GUI) and Natural Language Processing (NLP) to compute the specifications from the structured text, in the domain of model checking, robot controllers and formal methods.

TL specifications have a strong resemblance to natural languages so using structured-texts to generate properties is an advantageous approach. "*Specifying a query by selecting fragments of an English sentence allow users more easily to formulate required properties for a given model which gives him/her the possibility to focus on the specification of the properties rather than on the specification process itself*" [34]. Structured English sentences allows users to understand the meaning of a property without requiring them to interpret the the temporal logic representation [49]. "*Feedback from industry has indicated that a structured English representation is less intimidating than the temporal logic notation*" [49]. Many projects have undertaken this approach, one among them being Specification Pattern Instantiation and Derivation EnviRonment [48] (SPIDER).

SPIDER is one such tool that takes Structured English text and converts them to the desired specification properties. It supports specifications for LTL, Computational Tree Logic [16] (CTL), Graphical Interval Logic [70] (GIL), MTL, Timed Computational Tree Logic [2] (TCTL) and Real-Time Interval Logic [61] (RTGIL). The approach of using grammar over techniques such as NLP has many benefits over the latter. "*Since different notations are used for the same temporal operators, a structured English framework targeted for robotic applications can offer a uniform representation of temporal logic formulas*" [52].

RuSTL implements the grammar described in [49], shown in Figure 5.1 and enhances it to handle real-valued signals to accommodate for STL. Dwyer collected an impressive amount of real-time specification patterns from industry and published his finding [27]. The work categorizes the specification properties into several patterns and shows that the majority of the properties, if not all, conform to one of the prescribed patterns. The grammar developed in [49] supports those patterns and created additional mappings to categorize timing-based requirements since such requirements were originally not specified in the study [27].

The grammar starts by selecting the scope of the property, followed by whether if the property is qualitative or real-time. Qualitative properties can fall under the category of *occurrence* or *order*. While the real-time properties can be *duration* (bounds the interval I of the occurrence), *periodic* (addresses the periodicity of the occurrences) and *real-time order* (addresses the time-elapsed requirement between different occurrences).

One of the key points to note is that the grammar cannot be used to construct recursive and repetitive properties. Each sentence provided to the grammar gets mapped to a scoped specification pattern which is either qualitative or real-time. It also does not tailor to past-bounded temporal operators. RuSTL is based on the assumption to only handle future bounded STL properties, so these design decisions align with the core of RuSTL. As mentioned before, structured English representation is far more relatable than temporal

Start	1: property	::=	*scope* “, ” *specification* “.”
Scope	2: scope	::=	“Globally” | “Before ” R | “After ” Q | “Between ” Q “ and ” R | “After ” Q “ until ” R
General	3: specification	::=	*qualitativeType* | *realtimeType*
Qualitative	4: qualitativeType	::=	*occurrenceCategory* | *orderCategory*
	5: occurrenceCategory	::=	*absencePattern* | *universalityPattern* | *existencePattern* | *boundedExistencePattern*
	6: absencePattern	::=	“it is never the case that ” P “ holds”
	7: universalityPattern	::=	“it is always the case that ” P “ holds”
	8: existencePattern	::=	P “ eventually holds”
	9: boundedExistencePattern	::=	“transitions to states in which ” P “ holds occur at most twice”
	10: orderCategory	::=	“it is always the case that if ” P “ holds” (*precedencePattern* | *precedenceChainPattern1-2* | *precedenceChainPattern2-1* | *responsePattern* | *responseChainPattern1-2* | *responseChainPattern2-1* | *constrainedChainPattern1-2*)
	11: precedencePattern	::=	“, then ” S “ previously held”
	12: precedenceChainPattern1-2	::=	“ and is succeeded by ” S “, then ” T “ previously held”
	13: precedenceChainPattern2-1	::=	“, then ” S “ previously held and was preceded by ” T
	14: responsePattern	::=	“, then ” S “ eventually holds”
	15: responseChainPattern1-2	::=	“, then ” S “ eventually holds and is succeeded by ” T
	16: responseChainPattern2-1	::=	“ and is succeeded by ” S “, then ” T “ eventually holds after ” S
	17: constrainedChainPattern1-2	::=	“, then ” S “ eventually holds and is succeeded by ” T “, where ” Z “ does not hold between ” S “ and ” T
Real-time	18: realtimeType	::=	“it is always the case that ” (*durationCategory* | *periodicCategory* | *realtimeOrderCategory*)
	19: durationCategory	::=	“once ” P “ becomes satisfied, it holds for ” (*minDurationPattern* | *maxDurationPattern*)
	20: minDurationPattern	::=	“at least ” c “ time unit(s)”
	21: maxDurationPattern	::=	“less than ” c “ time unit(s)”
	22: periodicCategory	::=	P “ holds ” *boundedRecurrencePattern*
	23: boundedRecurrencePattern	::=	“at least every ” c “ time unit(s)”
	24: realtimeOrderCategory	::=	“if ” P “ holds, then ” S “ holds ” (*boundedResponsePattern* | *boundedInvariancePattern*)
	25: boundedResponsePattern	::=	“after at most ” c “ time unit(s)”
	26: boundedInvariancePattern	::=	“for at least ” c “ time unit(s)”

Figure 5.1: Structured English Grammar by [49]

logic notation. Not supporting recursive properties limits the formulas that can be constructed. Nested text that covers the complete semantics would lose their resemblance to English language sentences. That would retain that barrier for non-experts who wish to use this formal specification tool. A further point to consider is the fact that this grammar was developed by analyzing specification properties most often monitored by industry, giving confidence in the knowledge that the grammar covers correctness properties most frequently used by experts in RV.

The grammar is defined in the EBNF format and then uses ANTLR for lexing and parsing. A few minor modifications were made to facilitate the integration of the grammar within RuSTL. Firstly, due to Assumption 4, interval I in the form of $[\leq t]$ are not considered valid, since the original work maps the timing restriction to singleton sets.

RuSTL maps to this time value to the prescribed future bounded time-interval format $[0, t]$, to circumvent the error.

A slight text modification was made to Rule 18. The string is changed from "it is always the case that" to "it's always the case that". This modification is due to ANTLR's inherent limitation, where the parser only supports direct left recursion. Due to this, Rule 7 and Rule 18 which require the same start string cannot be told apart, since ANTLR does not go into deeper levels to differentiate an input. ANTLR will perform left recursion on a single rule and try to match the input to the first rule matched, and as a result, a valid string input for the grammar may cause an error. So by making this modification we overcome this issue while retaining the original architecture of the grammar.

We tested the implementation against the examples provided in [49] and supplemented them with additional test cases.

"**Globally, it's always the case that if** x == 0 **holds, then** y == 1 **holds after at most** 6 **time unit(s).**"

$$\square(x == 0 \rightarrow \Diamond_{[0,6]} y == 1) \tag{5.2}$$

"**Globally, it's always the case that** x == 0 **holds at least every** 10 **time unit(s).**"

$$\square(\Diamond_{[0,20]} x == 0) \tag{5.3}$$

"**Globally, it's always the case that if** RampDownInitiated == True **holds, then** AssistTorque == 0 **holds after at most** 20 **time unit(s).**"

$$\square(RampDownInitiated == True \rightarrow \Diamond_{[0,20]} AssistTorque == 0) \tag{5.4}$$

"**Globally, it's always the case that if** RampDownInitiated == True **holds, then** AssistTorque != 0 **holds for at least** 19 **time unit(s).**"

$$\square(RampDownInitiated == True \rightarrow \square[0, 19] AssistTorque! = 0) \tag{5.5}$$

Chapter 6

Case Study

The University of Waterloo has a highly advanced and established autonomous driving research platform. In 2017 and 2018, the University of Waterloo partnered with Renesas, QNX and other companies to showcase the project at the Consumer Electronics Show (CES), in Las Vegas, Nevada. The goal of the project was to showcase the Renesas Autonomy Demonstrator (RAD), a fully functional autonomous driving vehicle.

Figure 6.1: RAD - Photo credit: investStratford/Terry Manzo.

The demo consisted of the vehicle driving at low speeds (under 15 km/h), following a

predefined path and performing complex maneuvers such as lane-keep-assist, pedestrian identification, identifying and following traffic and road signs, forward and reverse parking in designated parking spots, along with communicating with other vehicles on the road and collision avoidance.

The vehicle used was a 2015 Lincoln MKZ. The company Dataspeed retrofitted the vehicle with a drive-by-wire system which gave the ability to control the car without any physical human interaction. The vehicle was instrumented with a wide array of sensors, including a Novatel Span GPS. The Global Positioning System (GPS) provided an accuracy of about 5 cm and updated at a frequency of 20 Hz. A Novatel Inertial Measurement Unit (IMU) for heading angle accuracy. A Velodyne LiDAR, which "*is a remote sensing technology which uses the pulse from a laser to collect measurements which can then be used to create 3D models and maps of objects and environments*" [1]. The instrumented Vehicle to Vehicle (V2V) and Vehicle to Infrastructure (V2I) radio system were used to establish communication with smart traffic lights and other vehicles on the road.

6.1 Robot Operating System (ROS)

Robot Operating System (ROS) is a popular, open-source robotics framework widely used in industry for robot applications such as drones, rumbas, and autonomous vehicles. It provides an extensive set of libraries and drivers that allow developers to design their applications without having to build everything from the ground up. These resources are available for applications targetted towards motion planning, 3D localization and mapping, state estimation and sensor fusion using sophisticated techniques such as Kalman filters (KF) for prediction and updates, image processing for classification, etc. Using ROS dramatically reduces the development time of robotic applications, and with its broad community base, and publicly available project code repositories it is easy to find helpful resources.

ROS was designed to be language neutral and binding are currently available in C++, Python, Octave, and LISP, and development in others is underway. Thus ROS can be used in wide variety of scenarios and environments, and although performance differs based on the language in use, the fact that the underlying methodologies remain the same, applications developed in one programming language can easily communicate with those developed in a different one.

ROS Nodes

A ROS node is akin to a process. Usually, a node is designed to perform a specific task. A single ROS application will typically have multiple nodes communicating with one another. A node does not know who is on the receiving end of its communication. All the nodes need not run on the same machine but can be running on distributed platform.

ROS Messages

Nodes communicate with one another by transmitting a message. Messages in ROS are generated using simple text files that describe the data structure, comprising typed fields. The message structure supports primitive types such as integer, floating point, Boolean, etc. The header in each message holds the time in nanoseconds at which it was published.

ROS works on a publish / subscribe model. The nodes specify what information they are going to publish and at what frequency. Subsequently, the nodes also list what information they will be listening for, and specify a callback associated for each such.

ROS Topics

Nodes exchange messages over named busses call *Topics*. Topics in ROS are an asynchronous mode of communication. Publisher nodes transmit pre-configured ROS message types to a specified topic and listener nodes then subscribe to said topic and execute the associated callback function in their code. More than one node may publish and/or subscribe to the same topic, and a node can also publish and/or subscribe to multiple topics. A key point to note is that subscriber nodes are unable to verify the identity of the publisher for any message they receive.

ROS Bags

The ROS *rosbag* utility gives the ability to record and playback ROS topics. A user can either choose to identify a subset of the topics or specify to *bag* all topics. The files collected from using this tool are usually referred to as *bags* and are used to record and play the bags. The data appears as JSON objects, and any text editor can be used to view the data. In this work the term *bagging* means using the *rosbag* utility to record ROS message.

6.2 Renesas Autonomy Demonstrator (RAD)

Figure 6.2 provides a high-level overview of the architecture. The three independent ROS autonomy stacks executed concurrently and were fed the sensor reading and feedback updating the current state of the vehicle. They processed the data and computed the *control commands* —the future heading including steering, brake, throttle and gear values that should be executed by the car. Each ROS stack packets its *control commands* message and sends the data to the micro-controller via the User Datagram Protocol (UDP) communication protocol. The microcontroller checks the incoming data for timing and authenticates the messages to ensure their integrity and chooses which of the received actuation command message to pass on to the Dataspeed module. Ideally, all three stacks should compute the future heading within an error threshold. The Dataspeed module then transmits the commands on the vehicle CAN bus, at 50 Hz, which ultimately actuates the vehicle, updating the throttle, brake, steering, and gear values accordingly.

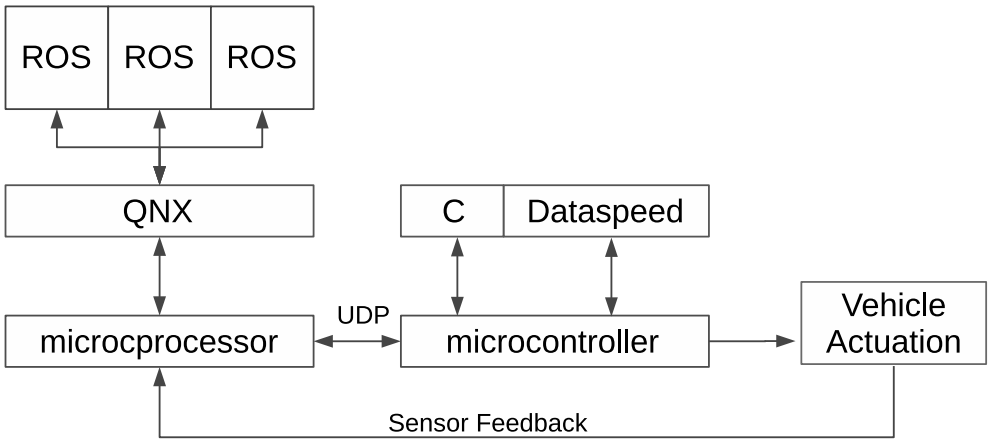

Figure 6.2: Architectural overview.

The vehicle was configured to start *bagging* a list of pre-defined ROS topics on power-up. We knew the responsibilities of each node in the Autonomosse ROS architecture. We also knew which topics each node publishes and/or subscribes to, at what frequency and what information is contained in the message sent on that topic. The list comprised of topics that provided a high-level state estimation of the vehicle. The data was cleaned and imported into a PostgreSQL database for analysis. Table 6.1 outlines a few of the fields in the dataset:

Data Field	Definition
autonomy_enabled	User requesting vehicle into autonomy mode
vehicle_brake_report_enabled	Signifies the vehicle is in autonomy mode
control_commands_gear_cmd_gear	Gear requested by autonomy stack
vehicle_gear_cmd_gear	Requested gear from Dataspeed to vehicle
vehicle_gear_state_gear	Current gear value of the vehicle

Table 6.1: Sample fields from collected dataset.

6.3 Experimental Results

Based on the architecture and the relationship between the components, multiple properties can be checked to ensure that the vehicle was running safely. One such property is presented in Equation 6.1, which specifies that: **Globally, it's always the case that if** autonomy_enabled **holds, then** vehicle_brake_report_enabled **holds after at most** 5 **time unit(s).**. Here *vehicle_brake_report_enabled* signifies a user input which commands the vehicle to go from manual to autonomous driving mode, while *vehicle_brake_report_enabled* is a field in the Dataspeed feedback loop, which if *True*, indicates that the vehicle is in autonomous driving mode. This check is performed to ensure that the acceptable worst-case timing behavior was never violated. Since the project is a prototype of an autonomous self-driving car, there are opportunities for optimization. So this specification checks the upper bound placed on getting the signal from the in-car display when the user pushes the button to car switching the driving mode to autonomous.

$$\square(autonomy\_enabled \rightarrow \Diamond_{[0,5]} vehicle\_brake\_report\_enabled == True) \tag{6.1}$$

Multiple tests were conducted before the demo to ensure that the described behavior was always satisfied. The data was queried from the database for increasingly longer signal traces, ranging from 1,000 to 100,000, sampled at a step size of 1 second. So, the number of rows in a trace is equal to the number of seconds the trace covers. An additional signal trace of 1,000,000 samples was created by replicating the data of the 100,000 second trace to stress test the system. Table 6.2 shows a summary of the timing results obtained from running the experiment multiple times, and Figure 6.3 is a plot depicting the results. The second column in the Table 6.2 refers to the number of times *autonomy_enabled* was *True* in the trace and SD stands for standard deviation. The timing results are consistent for the multiple executions so the mean time in seconds for the experiments is presented. The y-axis in Figure 6.3 showing the computation time is in log scale

Trace Length	# of Requests	Mean Computation Time (s)	SD (s)
1,000	97	0.1456	0.0026
10,000	950	1.1615	0.0239
100,000	9,600	11.0871	0.2611
1,000,000	96,000	234.9861	1.62

Table 6.2: Computation time vs trace length.

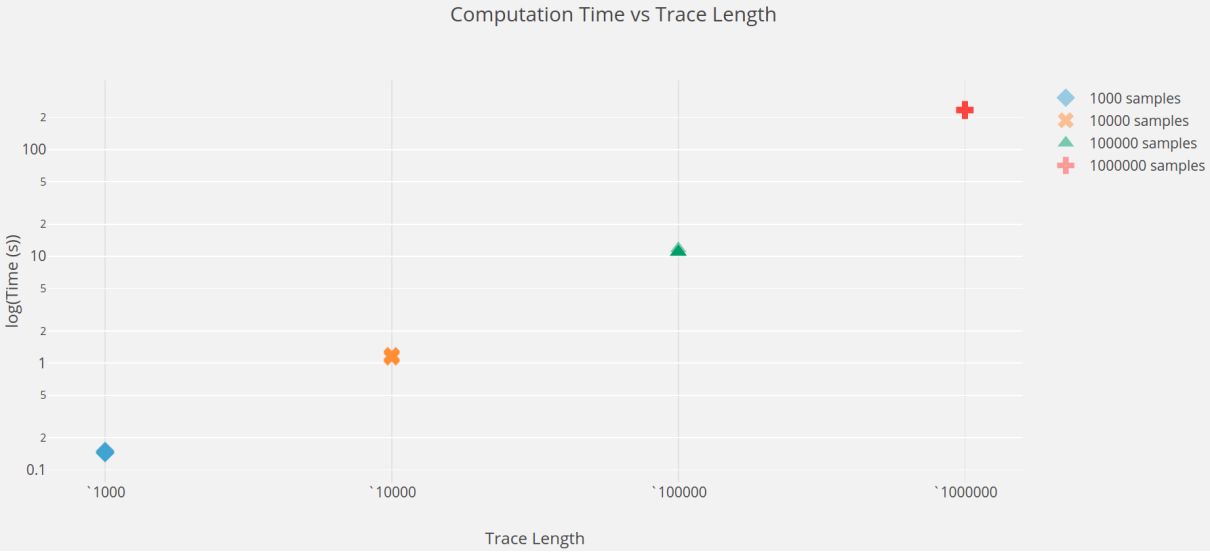

Figure 6.3: Computation time vs trace length.

All traces satisfied the STL formulas listed in Equation 6.1, and the results show that the computation time for a formula increases linearly with the length of the trace. Figure 6.4 shows the output from the diagnostic charts automatically created using Plotly, after running the *monitor program*. The user can choose which signals to plot and focus on a sub-section of the time. In Figure 6.4, *autonomy_enabled* was replaced by p and *vehicle_brake_report_enabled* by q so that the labels on the plot do not become too wide. Each curve in the plot corresponds to either the input signal used from the trace in the formula or to a node in the parse tree and can be toggled on or off. For any curve, $+1$ means that the formula equated to *True* at that time t, while -1 means it equated to *False*. 0 means that the formula was not computed for that time value. The range slider at the bottom of the chart allows the user to zoom into specific time windows of their choice.

The property checked in Equation 6.1 was chosen since it was always guaranteed to hold and could be used to test the performance of RuSTL. However, the property only contains atomic propositions and thus can be considered a MITL formula. Equation 6.2 checks that globally it is always the case that if current vehicle's gear value is *Drive*, and the current requested next gear value is not *Drive*, this implies that globally for at least the next 3 time units, the vehicle should not be requested to go into the *Drive* gear. The property concerns the safety of the vehicle, to ensure that it does not change gears too often.

The property contains real-value variable comparison on the current gear status and the next requested gear and is, therefore, a real STL property. This specification was not chosen at random but derived from the actual physical behavior exhibited by the car. Passengers in the vehicle could feel a "jerky motion" while it was moving in autonomous mode and, on the rare occasion, exit out of autonomous mode while performing a complicated maneuver.

$$\begin{aligned}\square((\_vehicle\_gear\_state\_gear = Drive \ \wedge \ \_vehicle\_gear\_cmd\_gear! = Drive) \\ \rightarrow \square_{[0,3]} \_vehicle\_gear\_cmd\_gear = \neg Drive)\end{aligned} \tag{6.2}$$

Monitoring the specification on the trace showed that the property is violated at multiple time instances. Note, that this does not reveal the cause of the behavior, since the logic of how the control commands are computed is not part of the scope of the runtime-monitor. The violation only reveals that the ROS autonomy stack requests the gears to be switched quickly, and a closer look should be taken at the internal algorithms computing the values.

Based on the knowledge of the system and its behavior many additional properties can be checked using RuSTL. We know that the node publishing the control commands sends its message at 50 Hz, so the property presented in Equation 6.3 can be used to verify that a new message is always sent at regular intervals.

$$\square(\Diamond_{[0,0.02]}(control\_command\_msg\_sent)) \tag{6.3}$$

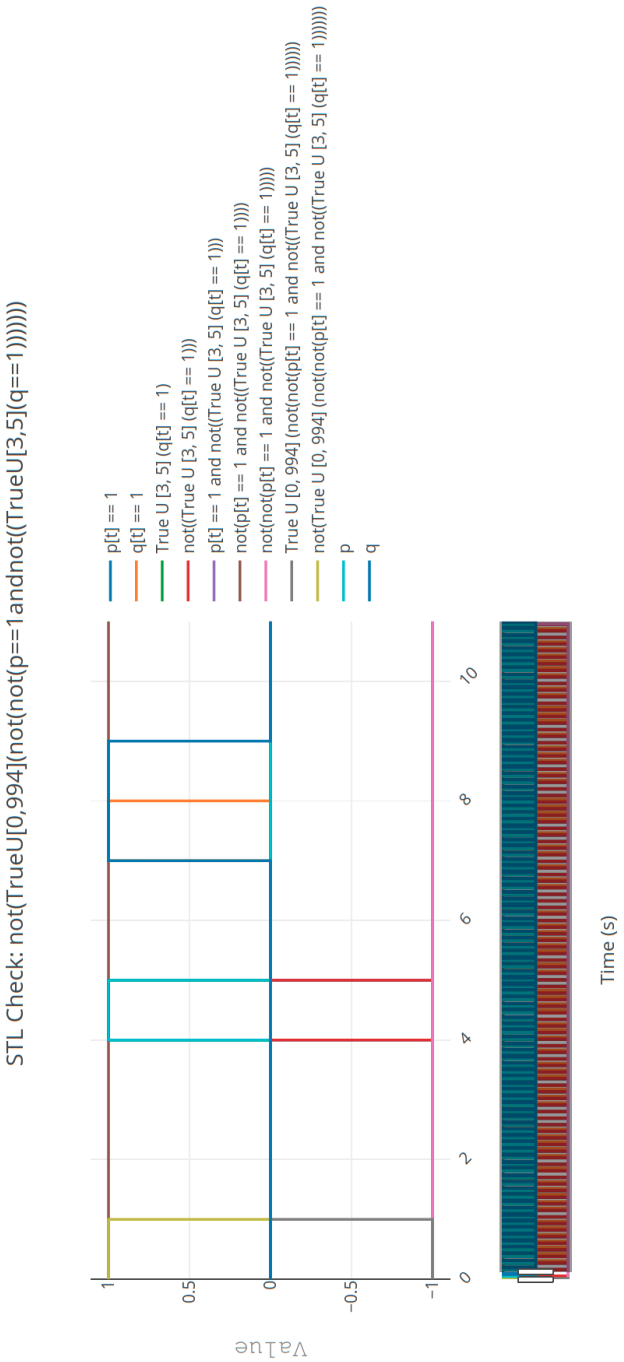

Figure 6.4: Diagnostic plot.

Chapter 7

Related Work

TL was first introduced four decades ago and STL more than one decade ago. So it is not surprising that a lot of work has been done to leverage these technologies over the years. Temporal Rover [26], is a commercial tool that can generate program code in either C, C++ or Java to monitor satisfiability of specifications at run time. LOLA [18] and MaC [55] are also one of the first tools developed for monitoring TL specifications. The original STL paper [56] provided an offline monitoring framework for bounded real-time MITL properties.

S-TALIRO [6] is an extensive tool built on the Matlab platform, used for monitoring, testing and verification of MTL properties both online [20] and offline. It was developed at Arizona State University and has had multiple updates with increased functionality over the years [42]. Along with a Boolean satisfaction, the tool also computes the robustness metric of trace over a MTL formula through a real-number [30, 31, 32].

The authors also acknowledge in their work, that writing formal specifications is an error-prone task, so S-TALIRO also provides a graphical formalism tool, VISPEC [43], providing non-expert users with an interactive system to visualize specification properties and develop MTL properties. "*The set of specifications that can be generated from this graphical formalism is a proper subset of the set of MTL specifications*" [42].

Breach [22] is another popular Matlab/C++ toolbox for simulation-based verification and analysis. It is a highly versatile tool, used for simulation-based analysis [23], mining requirements of Simulink models [46] and computing robustness of piecewise-continuous signal over an STL formula [24].

The Breach toolbox can be used to generate simulation traces for property falsification and parameter synthesis. Monitoring the satisfiability of STL formulas can be performed

both online [19] and offline [22]. It provides both a GUI and a Command-line interface (CLI) to define properties. In [24] the developers of Breach, provide experimental results that show their implementation outperforming S-TALIRO v1.3.

AMT [64] is another tool for offline monitoring temporal properties of continuous signals using the STL specification. It is written in C++ for Linux machines. It is a stand-alone application with a GUI implemented in QT library. It allows users to define properties and assertion and manage input signals. The tool evaluates the property for the given input trace and provides a visualization window showing graphical output related to the result. Online monitor techniques developed in [58] were later implemented in AMT.

AMT 2.0 [63] is a recent update of the same tool and perform qualitative and quantitative analysis with Extended Signal Temporal Logix (xSTL) for offline monitoring. xSTL integrates Timed Regular Expressions (TRE) within STL. TRE are used to match segments of the trace on which quantitative semantics are applied. The tool offers an upgraded GUI along with the capability to include *declarations* and *aliases* in the GUI itself.

U-Check [11] is an open-source software suite written in Java, that deals with stochastic models (Continuous-Time Markov Chains). U-Check performs property falsification and parameter synthesis on MITL and STL properties.

Both Stlinspector [71] and [69] are tools that automatically generate traces that either satisfy or violate an STL formula. Labeled traces help users cement their understanding of what the property is trying to monitor and update the property if need be.

STL has gained a lot of popularity in recent years, which is reflected by its applications in the domain of FPGA's [44], Chemical Reaction Networks (CRN) [12], frequency analysis and signal processing [13, 25], networked power systems [37] among many others. The broad breadth of applications spanning over multiple fields proves towards the versatility and usefulness of the specification.

Chapter 8

Conclusion

8.1 Summary

In this thesis we proposed RuSTL, an RV application for qualitative semantics of future-bounded STL properties. RuSTL generates a stand-alone *monitor program*, for any valid STL formula φ, that can be deployed on most modern computers headless (without the use of a GUI). RuSTL also has the capability to take structured English language text and convert the input into a corresponding STL formula following the grammar provided in [49]. RuSTL also provides the user with the option to automatically generate diagnostic plots for the *monitor program* for any valid input trace. The plots allow the user to visually inspect the results of the monitor.

The *monitor program* generated by RuSTL can be run independently of the application and we show that the *monitor program* is both sound and it terminates. Along with that, we show that our algorithm that computes the *monitor program* from the parse-tree is complete.

RuSTL was tested on traces collected from an autonomous self-driving vehicle. The experimental results also show that the computation time for the *monitor program* grows linearly with the size of the input trace.

8.2 Future Work

Based on the related work performed by other research groups there are several avenues for further research. Leveraging techniques such as vector-based operation for data manip-

ulation and dynamic programming by storing the results of searching the state-space once and using them for future operations can increase the performance of the application with respect to the computation time.

8.2.1 Online Monitoring

RuSTL currently performs offline RV of STL properties. The application can be enhanced to perform online RV. To that end, the temporal depth of a formula φ is already computed for any given temporal property as described in Section 4.5. Online RV gives the obvious advantage of alerting a user as soon as a property is violated. The monitor will stop the processing the trace whenever a violation occurs.

8.2.2 Robustness Metric

The application currently performs only qualitative satisfaction analysis for STL properties. Receiving just a binary response for monitoring real-values signals does not account for any degree of tolerance in the observed system.

Quantitative semantics adds a robustness metric, which is the degree of satisfaction/violation of the property. The robustness metric assigns a real-valued number to the formula over a given trace, whose magnitude indicates the satisfaction or violation by some distance metric. If the property being monitored is $x < c$, and at some time t, x exceeds that value, then the property is violated. But it is useful to know if the violation was marginal $x = c + \epsilon$, or if the violation was of a higher magnitude $x = c \times 10$.

Having the capability to quantify the degree of satisfaction/violation is a highly desirable feature, providing more insight into the observed system. Robustness for STL was introduced in [31]. Many works [24, 45, 42] have implemented this quantitative semantic using a variety of techniques. Robustness is a key metric for categorizing the performance of a system that goes beyond just the Boolean semantics.

A recent survey [8] discusses the prominent approaches for qualitative and quantitative measurements designed thus far and list the different tools available.

8.2.3 Multi-Language Process Monitor

The *monitor program* generated by RuSTL is in Python. Using Python was a design decision made due to the ever-increasing use of the language and the tools available for

data analytics and visualization.

However, for embedded systems, C is the logical programming language of choice. The application can be upgraded to generate the core *monitor program* in C, which can then deployed on micro-controllers as well as microprocessors.

Additional features to include are property falsification which automatically generates counterexamples that violate a given STL property. Parameter synthesis is also a useful feature to have which finds the range of parameters in a STL property that satisfies the behavior. Property falsification can be considered as a dual problem to parameter synthesis [8].

References

[1] 3dlasermapping. 3d laser mapping, 2019. Online documentation https://www.3dlasermapping.com/what-is-lidar-and-how-does-it-work/, Last accessed on 2019-02-17.

[2] Rajeev Alur. *Techniques for automatic verification of real-time systems*. PhD thesis, stanford university, 1991.

[3] Rajeev Alur, Tomás Feder, and Thomas A Henzinger. The benefits of relaxing punctuality. *Journal of the ACM (JACM)*, 43(1):116–146, 1996.

[4] Rajeev Alur, Thomas A Henzinger, and Orna Kupferman. Alternating-time temporal logic. *Journal of the ACM (JACM)*, 49(5):672–713, 2002.

[5] Vincenzo Ambriola and Vincenzo Gervasi. Processing natural language requirements. In *Proceedings 12th IEEE International Conference Automated Software Engineering*, pages 36–45. IEEE, 1997.

[6] Yashwanth Annpureddy, Che Liu, Georgios Fainekos, and Sriram Sankaranarayanan. S-taliro: A tool for temporal logic falsification for hybrid systems. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 254–257. Springer, 2011.

[7] Howard Barringer, Allen Goldberg, Klaus Havelund, and Koushik Sen. Rule-based runtime verification. In *International Workshop on Verification, Model Checking, and Abstract Interpretation*, pages 44–57. Springer, 2004.

[8] Ezio Bartocci, Jyotirmoy Deshmukh, Alexandre Donzé, Georgios Fainekos, Oded Maler, Dejan Ničković, and Sriram Sankaranarayanan. Specification-based monitoring of cyber-physical systems: a survey on theory, tools and applications. In *Lectures on Runtime Verification*, pages 135–175. Springer, 2018.

[9] Andreas Bauer, Martin Leucker, and Christian Schallhart. Runtime verification for ltl and tltl. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 20(4):14, 2011.

[10] Mordechai Ben-Ari, Amir Pnueli, and Zohar Manna. The temporal logic of branching time. *Acta informatica*, 20(3):207–226, 1983.

[11] Luca Bortolussi, Dimitrios Milios, and Guido Sanguinetti. U-check: Model checking and parameter synthesis under uncertainty. In *International Conference on Quantitative Evaluation of Systems*, pages 89–104. Springer, 2015.

[12] Luca Bortolussi, Alberto Policriti, and Simone Silvetti. Logic-based multi-objective design of chemical reaction networks. In *International Workshop on Hybrid Systems Biology*, pages 164–178. Springer, 2016.

[13] Lubos Brim, P Dluhoš, D Šafránek, and Tomas Vejpustek. Stl: Extending signal temporal logic with signal-value freezing operator. *Information and Computation*, 236:52–67, 2014.

[14] Alessandro Cimatti, Edmund Clarke, Enrico Giunchiglia, Fausto Giunchiglia, Marco Pistore, Marco Roveri, Roberto Sebastiani, and Armando Tacchella. Nusmv 2: An opensource tool for symbolic model checking. In *International Conference on Computer Aided Verification*, pages 359–364. Springer, 2002.

[15] Edmund M Clarke and E Allen Emerson. Design and synthesis of synchronization skeletons using branching time temporal logic. In *Workshop on Logic of Programs*, pages 52–71. Springer, 1981.

[16] Edmund M. Clarke, E Allen Emerson, and A Prasad Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 8(2):244–263, 1986.

[17] Edmund M Clarke Jr, Orna Grumberg, Daniel Kroening, Doron Peled, and Helmut Veith. *Model checking*. MIT press, 2018.

[18] Ben d'Angelo, Sriram Sankaranarayanan, César Sánchez, Will Robinson, Bernd Finkbeiner, Henny B Sipma, Sandeep Mehrotra, and Zohar Manna. Lola: Runtime monitoring of synchronous systems. In *Temporal Representation and Reasoning, 2005. TIME 2005. 12th International Symposium on*, pages 166–174. IEEE, 2005.

[19] Jyotirmoy V Deshmukh, Alexandre Donzé, Shromona Ghosh, Xiaoqing Jin, Garvit Juniwal, and Sanjit A Seshia. Robust online monitoring of signal temporal logic. *Formal Methods in System Design*, 51(1):5–30, 2017.

[20] Adel Dokhanchi, Bardh Hoxha, and Georgios Fainekos. On-line monitoring for temporal logic robustness. In *International Conference on Runtime Verification*, pages 231–246. Springer, 2014.

[21] Adel Dokhanchi, Bardh Hoxha, and Georgios Fainekos. Metric interval temporal logic specification elicitation and debugging. In *Formal Methods and Models for Codesign (MEMOCODE), 2015 ACM/IEEE International Conference on*, pages 70–79. IEEE, 2015.

[22] Alexandre Donzé. Breach, a toolbox for verification and parameter synthesis of hybrid systems. In *International Conference on Computer Aided Verification*, pages 167–170. Springer, 2010.

[23] Alexandre Donzé, Eric Fanchon, Lucie Martine Gattepaille, Oded Maler, and Philippe Tracqui. Robustness analysis and behavior discrimination in enzymatic reaction networks. *PloS one*, 6(9):e24246, 2011.

[24] Alexandre Donzé, Thomas Ferrere, and Oded Maler. Efficient robust monitoring for stl. In *International Conference on Computer Aided Verification*, pages 264–279. Springer, 2013.

[25] Alexandre Donzé, Oded Maler, Ezio Bartocci, Dejan Nickovic, Radu Grosu, and Scott Smolka. On temporal logic and signal processing. In *International Symposium on Automated Technology for Verification and Analysis*, pages 92–106. Springer, 2012.

[26] Doron Drusinsky. The temporal rover and the atg rover. In *International SPIN Workshop on Model Checking of Software*, pages 323–330. Springer, 2000.

[27] Matthew B Dwyer, George S Avrunin, and James C Corbett. Patterns in property specifications for finite-state verification. In *Proceedings of the 21st international conference on Software engineering*, pages 411–420. ACM, 1999.

[28] Cindy Eisner. Psl for runtime verification: Theory and practice. In *International Workshop on Runtime Verification*, pages 1–8. Springer, 2007.

[29] Georgios E Fainekos, Hadas Kress-Gazit, and George J Pappas. Temporal logic motion planning for mobile robots. In *Robotics and Automation, 2005. ICRA 2005.*

Proceedings of the 2005 IEEE International Conference on, pages 2020–2025. IEEE, 2005.

[30] Georgios E Fainekos and George J Pappas. Robustness of temporal logic specifications. In *Formal Approaches to Software Testing and Runtime Verification*, pages 178–192. Springer, 2006.

[31] Georgios E Fainekos and George J Pappas. Robustness of temporal logic specifications for continuous-time signals. *Theoretical Computer Science*, 410(42):4262–4291, 2009.

[32] Georgios E Fainekos, Sriram Sankaranarayanan, Koichi Ueda, and Hakan Yazarel. Verification of automotive control applications using s-taliro. In *2012 American Control Conference (ACC)*, pages 3567–3572. IEEE, 2012.

[33] Cameron Finucane, Gangyuan Jing, and Hadas Kress-Gazit. Ltlmop: Experimenting with language, temporal logic and robot control. In *Intelligent Robots and Systems (IROS), 2010 IEEE/RSJ International Conference on*, pages 1988–1993. IEEE, 2010.

[34] Stephan Flake, Wolfgang Müller, and Jürgen Ruf. Structured english for model checking specification. In *MBMV*, pages 99–108, 2000.

[35] Norbert E Fuchs and Rolf Schwitter. Attempto controlled english (ace). *arXiv preprint cmp-lg/9603003*, 1996.

[36] Dimitra Giannakopoulou and Klaus Havelund. Automata-based verification of temporal properties on running programs. In *Automated Software Engineering, 2001.(ASE 2001). Proceedings. 16th Annual International Conference on*, pages 412–416. IEEE, 2001.

[37] Iman Haghighi, Austin Jones, Zhaodan Kong, Ezio Bartocci, Radu Gros, and Calin Belta. Spatel: a novel spatial-temporal logic and its applications to networked systems. In *Proceedings of the 18th International Conference on Hybrid Systems: Computation and Control*, pages 189–198. ACM, 2015.

[38] Klaus Havelund and Grigore Roşu. Synthesizing monitors for safety properties. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 342–356. Springer, 2002.

[39] Thomas A Henzinger, J-F Raskin, and P-Y Schobbens. The regular real-time languages. In *International Colloquium on Automata, Languages, and Programming*, pages 580–591. Springer, 1998.

[40] Alexander Holt and Ewan Klein. A semantically-derived subset of english for hardware verification. In *Proceedings of the 37th annual meeting of the Association for Computational Linguistics on Computational Linguistics*, pages 451–456. Association for Computational Linguistics, 1999.

[41] Gerard J. Holzmann. The model checker spin. *IEEE Transactions on software engineering*, 23(5):279–295, 1997.

[42] Bardh Hoxha, Hoang Bach, Houssam Abbas, Adel Dokhanchi, Yoshihiro Kobayashi, and Georgios Fainekos. Towards formal specification visualization for testing and monitoring of cyber-physical systems. In *Int. Workshop on Design and Implementation of Formal Tools and Systems*, 2014.

[43] Bardh Hoxha, Nikolaos Mavridis, and Georgios Fainekos. Vispec: A graphical tool for elicitation of mtl requirements. In *Intelligent Robots and Systems (IROS), 2015 IEEE/RSJ International Conference on*, pages 3486–3492. IEEE, 2015.

[44] Stefan Jakšić, Ezio Bartocci, Radu Grosu, Reinhard Kloibhofer, Thang Nguyen, and Dejan Ničković. From signal temporal logic to fpga monitors. In *Formal Methods and Models for Codesign (MEMOCODE), 2015 ACM/IEEE International Conference on*, pages 218–227. IEEE, 2015.

[45] Susmit Jha, Ashish Tiwari, Sanjit A Seshia, Tuhin Sahai, and Natarajan Shankar. Telex: Passive stl learning using only positive examples. In *International Conference on Runtime Verification*, pages 208–224. Springer, 2017.

[46] Xiaoqing Jin, Alexandre Donzé, Jyotirmoy V Deshmukh, and Sanjit A Seshia. Mining requirements from closed-loop control models. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 34(11):1704–1717, 2015.

[47] Marius Kloetzer and Calin Belta. A fully automated framework for control of linear systems from temporal logic specifications. *IEEE Transactions on Automatic Control*, 53(1):287–297, 2008.

[48] Sascha Konrad and Betty HC Cheng. Facilitating the construction of specification pattern-based properties. In *13th IEEE International Conference on Requirements Engineering (RE'05)*, pages 329–338. IEEE, 2005.

[49] Sascha Konrad and Betty HC Cheng. Real-time specification patterns. In *Proceedings of the 27th international conference on Software engineering*, pages 372–381. ACM, 2005.

[50] Ron Koymans. Specifying real-time properties with metric temporal logic. *Real-time systems*, 2(4):255–299, 1990.

[51] Hadas Kress-Gazit, Georgios E Fainekos, and George J Pappas. Where's waldo? sensor-based temporal logic motion planning. In *Robotics and Automation, 2007 IEEE International Conference on*, pages 3116–3121. IEEE, 2007.

[52] Hadas Kress-Gazit, Georgios E Fainekos, and George J Pappas. Translating structured english to robot controllers. *Advanced Robotics*, 22(12):1343–1359, 2008.

[53] Leslie Lamport. What good is temporal logic? In *IFIP congress*, volume 83, pages 657–668, 1983.

[54] Stanislao Lauria, Guido Bugmann, Theocharis Kyriacou, Johan Bos, and Ewan Klein. Converting natural language route instructions into robot executable procedures. In *Robot and Human Interactive Communication, 2002. Proceedings. 11th IEEE International Workshop on*, pages 223–228. IEEE, 2002.

[55] Insup Lee, Sampath Kannan, Moonjoo Kim, Oleg Sokolsky, and Mahesh Viswanathan. Runtime assurance based on formal specifications. *Departmental Papers (CIS)*, page 294, 1999.

[56] Oded Maler and Dejan Nickovic. Monitoring temporal properties of continuous signals. In *Formal Techniques, Modelling and Analysis of Timed and Fault-Tolerant Systems*, pages 152–166. Springer, 2004.

[57] Oded Maler, Dejan Nickovic, and Amir Pnueli. On synthesizing controllers from bounded-response properties. In *International Conference on Computer Aided Verification*, pages 95–107. Springer, 2007.

[58] Oded Maler, Dejan Nickovic, and Amir Pnueli. Checking temporal properties of discrete, timed and continuous behaviors. In *Pillars of computer science*, pages 475–505. Springer, 2008.

[59] Kenneth L McMillan. Symbolic model checking. In *Symbolic Model Checking*, pages 25–60. Springer, 1993.

[60] James Bret Michael, Vanessa L Ong, and Neil C Rowe. Natural-language processing support for developing policy-governed software systems. In *Proceedings 39th International Conference and Exhibition on Technology of Object-Oriented Languages and Systems. TOOLS 39*, pages 263–274. IEEE, 2001.

[61] Louise E Moser, YS Ramakrishna, George Kutty, P Michael Melliar-Smith, and Laura K Dillon. A graphical environment for the design of concurrent real-time systems. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 6(1):31–79, 1997.

[62] Rani Nelken and Nissim Francez. Automatic translation of natural language system specifications into temporal logic. In *International Conference on Computer Aided Verification*, pages 360–371. Springer, 1996.

[63] Dejan Ničković, Olivier Lebeltel, Oded Maler, Thomas Ferrère, and Dogan Ulus. Amt 2.0: Qualitative and quantitative trace analysis with extended signal temporal logic. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 303–319. Springer, 2018.

[64] Dejan Nickovic and Oded Maler. Amt: A property-based monitoring tool for analog systems. In *International Conference on Formal Modeling and Analysis of Timed Systems*, pages 304–319. Springer, 2007.

[65] Kurt M. Olender and Leon J. Osterweil. Cecil: A sequencing constraint language for automatic static analysis generation. *IEEE Transactions on Software Engineering*, 16(3):268–280, 1990.

[66] Terence Parr. *The definitive ANTLR 4 reference*. Pragmatic Bookshelf, 2013.

[67] Terence J. Parr and Russell W. Quong. Antlr: A predicated-ll (k) parser generator. *Software: Practice and Experience*, 25(7):789–810, 1995.

[68] Amir Pnueli. The temporal logic of programs. In *Foundations of Computer Science, 1977., 18th Annual Symposium on*, pages 46–57. IEEE, 1977.

[69] Pavithra Prabhakar, Ratan Lal, and James Kapinski. Automatic trace generation for signal temporal logic. In *2018 IEEE Real-Time Systems Symposium (RTSS)*, pages 208–217. IEEE, 2018.

[70] Y Srinivas Ramakrishna, P Michael Melliar-Smith, Louise E Moser, Laura K Dillon, and George Kutty. Interval logics and their decision procedures: Part ii: A real-time interval logic. *Theoretical Computer Science*, 170(1-2):1–46, 1996.

[71] Hendrik Roehm, Thomas Heinz, and Eva Charlotte Mayer. Stlinspector: Stl validation with guarantees. In *International Conference on Computer Aided Verification*, pages 225–232. Springer, 2017.

[72] Terence Parr. Antlr4 python examples, 2016. Online documentation https://github.com/jszheng/py3antlr4book, Last accessed on 2019-02-17.

[73] Wikipedia. De morgan's laws, 2019. Online documentation https://en.wikipedia.org/wiki/De_Morgan%27s_laws, Last accessed on 2019-03-01.

APPENDICES

Appendix A

STL Grammar for ANTLR4

Complete STL Grammar for ANTLR

```
grammar stlGrammar;

prog : stlFormula+ ;

stlFormula::=
    stlFormula 'U' timeSlice stlFormula     # stlUntilFormula
    |stlFormula implies stlFormula          # stlFormulaImplies
    |stlFormula andorOp stlFormula          # stlConjDisjFormula
    |NOT stlFormula                         # stlNotFormula
    |'G' timeSlice? '('stlFormula')'        # stlGlobalFormula
    |'F' timeSlice? '('stlFormula')'        # stlEventualFormula
    |signalComp                             # stlSignalComp
    |signal                                 # stlSignal
    |Bool                                   # stlProp
    |'('stlFormula')'                       # stlParens


signalComp::=
    signal relOp expr       # signalExpr
    |expr relOp signal      # signalExpr
    |Bool relOp signal      # signalBool
    |signal relOp Bool      # signalBool


expr::=
    expr op=('*'|'/') expr      # MulDivExpr
    |expr op=('+'|'-') expr     # AddSubExpr
    |REAL                       # realExprs
    |'('expr')'
```