Formal Requirements Toolkit for Testing and Monitoring Temporal Logic-based Specifications

by

Jacob W. Anderson

A Thesis Presented in Partial Fulfillment
of the Requirements for the Degree
Master of Science

Approved April 2023 by the
Graduate Supervisory Committee:

Georgios Fainekos, Co-Chair
Giulia Pedrielli, Co-Chair
Zhe Xu

ARIZONA STATE UNIVERSITY

May 2023

ABSTRACT

Testing and verification is an essential procedure to assert a system adheres to some notion of safety. To validate such assertions, monitoring has provided an effective solution to verifying the conformance of complex systems against a set of properties describing what constitutes safe behavior. In authoring such properties, Temporal Logic (TL) has become a widely adopted specification language in many monitoring applications because of its ability to formally capture time-critical behaviors of reactive systems. This broad acceptance into the verification community and others, however, has naturally led to a lack of TL-based requirement elicitation standards as well as increased friction in tool interoperability.

In this thesis, I propose a standardization of TL-based requirement languages through the development of a Formal Requirements Toolkit (FoRek): a modular, extensible, and maintainable collection of TL parsers, translators, and interfaces. To this end, six propositional TL languages are supported in addition to their appropriate past-time variants to provide a framework for a variety of applications using TL as a specification language. Furthermore, improvements to the Pythonic Formal Requirements Language (PyFoReL) tool are performed in addition to a formal definition on the structure of a PyFoReL program. And lastly, to demonstrate the results of this work, FoRek is integrated into an offline monitor to showcase its intended use and potential applications into other domains.

To My

Niece Lucy & Nephew Owen

ACKNOWLEDGEMENTS

First and foremost, I would like to express my sincerest gratitude and appreciation towards my advisor, mentor, and professor Dr. Georgios Fainekos. I reminisce on the moment I first joined his lab in 2020 with no practical knowledge of what it means to perform research. In spite of this, his patience, compassion, and diligence to the field and his students has been an unwavering source of inspiration for me throughout all these years. I am indebtedly grateful for all the discussions, research, and opportunities provided by him that would otherwise not be possible without his hard work and dedication. I also would like to solemnly thank Dr. Giulia Pedrielli for her guidance, determination, and advocation of my work and future endeavors. Her perspective and candor is refreshing and has enabled me to thrive in my years of graduate school. I would also like to thank Dr. Zhe Xu for serving as my committee member and providing insightful feedback on my work. I have been fortunately alofted onto the shoulders of giants.

Within my graduate career, I had the invaluable opportunity of working for the Toyota Research Institute of North America alongside many experienced researchers. I am thankful to Bardh Hoxha, Tomoya Yamaguchi, Hideki Okamoto, Danil Prokhorov, Mitchell Black, Navid Hashemi, and Kandai Watanabe. I am especially grateful to Bardh for his benevolence and mentorship while balancing university work. This experience has shaped me in more ways than just my research capabilities.

I am deeply grateful for the friendships and connections I have formed throughout my career both at Arizona State University (ASU) and outside—they have greatly impacted me in the best way possible that one could only hope for. From the lab, thank you Elena Oikonomou, Daniel Pietz, Quinn Thibeault, Tanmay Khandait, Mina Jiang, Keyvan Majd, Aniruddh Chandratre, Surdeep Chotaliya, and Walstan Baptista. I would especially like to thank Quinn for his early support of my research, our countless discussions on all-things programming, and genuine interest in my work. I attribute a large growth in my software development skills to you. Furthermore, I would like to extend my gratitude to Elena who became a close friend in a short time. Her moral support, humble attitude, and experience has kept me inspired throughout much of my graduate career. I would like to thank Daniel for his candid advice and ideas that keep me assured and grounded through many aspects of life. Lastly, for experiences with those I treasure, thank you Celine, Edward, Riley, Grace, and many others with whom I have connected with.

To my close friends and family, I cannot fathom completing this degree without your moral and emotional support. To Daniel Vo, you truly are my ride or die. To Samuel Yauk, you are the respite through difficult periods. To my parents, Stephen and Laura, if not for everything else, thank you for taking care of me throughout my six years here at ASU. I am fortunate to have your physical presence by my side. To my brother, Nicholas, thank you for your brother-ly love. And to my sisters, Amanda and Kaitlin, I am officially twice the uncle! To my extended family, thank you all for the support—especially to my Uncle John and Aunt Lisa for showing interest and engagmement in my future.

This work was partially supported by DARPA ARCOS contract FA8750-20-C-0507. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the the Defense Advanced Research Projects Agency.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Figure Page

LIST OF DEFINITIONS

PREFACE

This thesis is organized as follows: Chapter 1 introduces the domain and motivation of recognizing formal requirements in monitoring, Chapter 2 provides a self-contained review of the core concepts related to this work, Chapter 3 contains a review of related work and approaches relevant to this thesis, Chapter 4 discusses the main contribution of this work, Chapter 5 discusses changes, modifications, and definitions for the formal requirements-based domain-specific language, Chapter 6 showcases results of the the main contribution of this work, and Chapter 7 provides a conclusion and potential future work.

Chapter 1

INTRODUCTION

Safety is a ubiquitous forethought in the design and development of complex systems [77]. This is especially true for computer-based systems that interact with the physical world. These aforementioned systems are most notably recognized and termed Cyber-Physical Systems (CPSs) [64]. Examples of such systems include power grids, automatic transmissions, autonomous vehicles, robots, and even thermostats. In designing a safe CPS, a notion of safety must first be expressed. The definition of safety for a system commonly originates from a set of desirable states often influenced by various parameters such as its dynamics, control inputs, and the environment it operates within. Considering these influences, a generalized solution or approach to certifying all existing and future systems safe is an infeasible feat. Hence, a vast amount of resources from the academic, industrial, and governmental sectors have been devoted to the verification and validation of numerous complex systems for safety and trust [20].

Of these efforts, formal methods have proven to be a successful and reliable approach in verifying the intended behavior of systems—specifically systems that are time-critical [54]. In most cases, formal methods such as *model checking* [23, 16] prove (exhaustively) the system satisfies the notion of safety set forth. While this methodology provides a rigid, provable, and reliable set of techniques to validate the safety of a system, its strength is its weakness. The analytical solutions provided by these techniques cannot be applied to all domains and problems due to its strict and resource intensive techniques. Furthermore, a model for the system under scrutiny may even not exist to accurately and efficiently capture its behavior effectively voiding formal approaches from being applicable. From these considerations, *simulation* and *testing* provide a numerical-based approach to solving problems that are otherwise limited by computational power, theory, or simply models when using formal methods. In the context of CPSs, *specification-based monitoring* (henceforth, monitoring) [14] is an effective requirement-based verification technique used to validate the conformance of a CPS against a safety requirement to assert safety guarantees.

In monitoring, generally a trace of the system (i.e., sequence of states) is checked against some requirement to verify whether the trace satisfies the requirement. This satisfaction constraint is dependent on the requirement expressed, its accuracy in capturing the intended property, and the capabilities of the language used. Natural Language (NL) is an effective framework to express requirements and has no limitations in this regard. However, as it is an inherently ambiguous language, evaluating the intention of an NL requirement is unreliable and troublesome. To resolve this issue of ambiguity and expressiveness, the use of Temporal Logic (TL) as a specification language for monitoring has become a popular choice. TL provides a formal approach to unambiguously express safety requirements for reactive systems and CPSs alike [62]. The popularity of the formal language has grown in recent decades due to its effective expressibility of time-based behaviors, foundations in formal philosophical logic, and relatively small grammar to manage. This widespread use can be seen through the development of numerous tools that utilize TL as a specification language to author formal requirements of safety properties [10, 31, 25, 66, 65]. While these tools illustrate the advantages of TL and the efforts towards safer and more robust CPSs, the development has, consequently, led to a divergence in authoring TL requirements at an implementation level. To further demonstrate this idea, consider the following example regarding a TL requirement.

Example. Consider a scenario where the system under scrutiny is a simple network server interface. The server has one job: Whenever a request is sent via a client, the server must eventually grant access to said client within the next 10 time units (inclusive). In other words, the TL requirement[1] can be written as so:

$$\square \, (req \;\Rightarrow\; \Diamond_{[0,10]} \, gnt) \tag{1.1}$$

where req is the event at which a request is received by the server and gnt is the event at which a grant is issued by the server. If this requirement under consideration were to be used by several different tools that support this TL language such as TLTk [25], RTAMT [66], and DP-TaLiRo [82], then the following issues arise. First, TLTk does not inherently support a TL-based interface. Therefore, such a requirement could not be easily transposed into the TLTk framework without additional work. Second, the requirement to be used in RTAMT would be written as follows:

```
always (req implies eventually[0:10] gnt)
```

where `always` corresponds to $\square$, `implies` corresponds to $\Rightarrow$, and `eventually` corresponds to $\Diamond$ in Equation (1.1). The same requirement, when written to be accepted by the DP-TaLiRo tool would be as follows:

```
[](req -> <>_[0,10] gnt)
```

where `[]` corresponds to $\square$, `->` corresponds to $\Rightarrow$, and `<>` corresponds to $\Diamond$ in Equation (1.1) leading to some obvious divisions in the preferred syntactic flavor for TL between RTAMT and DP-TaLiRo.

With the aforementioned example in mind, the issue of divergence has resulted in a lack of TL-based requirement elicitation standards, decreased tool interoperability,

[1]At this time, the meaning of the TL requirement is unimportant. A further discussion on TLs is performed in Section 2.2

and minor, yet impactful, syntactic differences in the preferred specification language. Furthermore, with an increasing number of tools released using TL, this gap is only widened throughout the verification community.

In this thesis, I advocate for the standardization of TL-based requirement tools through the development of the FoRek: a unified, modular, and extensible TL framework for parsing, translating, and authoring formal requirements in TL. In addition, updates to the Pythonic Formal Requirements Language (PyFoReL) tool [6] are showcased as well as a formal definition on the structure of a PyFoReL program.

1.1 Contributions

From the work performed in this thesis, the list of contributions (ordered by appearance) are as follows below:

1. A unified TL framework ("FoRek") for parsing, interpreting, and translating TL requirements.

2. A demonstration, by integration, of FoRek with the offline monitor TP-TaLiRo from the S-TaLiRo toolbox.

3. A series of new features, improvements, and changes to PyFoReL.

4. A formal definition by induction on the translational units of a PyFoReL requirement into a TL formula.

In this work, the set of TLs supported include: (1) Linear Temporal Logic (LTL), (2) Metric Temporal Logic (MTL), (3) Signal Temporal Logic (STL), (4) Timed Propositional Temporal Logic (TPTL), (5) Timed Quality Temporal Logic (TQTL), and (6) Spatio-Temporal Perception Logic (STPL).

1.2 Publications

In this section, an overview of publications (ordered most to least recent) received during the course of my M.S. degree are listed below as follows:

Publication 1.1. Jacob Anderson, Mohammad Hekmatnejad, and Georgios Fainekos. "PyFoReL: A Domain-Specific Language for Formal Requirements in Temporal Logic". In: *2022 IEEE 30th International Requirements Engineering Conference (RE)*. IEEE. 2022, pp. 266–267

Publication 1.2. Quinn Thibeault, Jacob Anderson, Aniruddh Chandratre, Giulia Pedrielli, and Georgios Fainekos. "Psy-taliro: A Python Toolbox for Search-based Test Generation for Cyber-Physical Systems". In: *Formal Methods for Industrial Critical Systems: 26th International Conference, FMICS 2021, Paris, France, August 24–26, 2021, Proceedings 26*. Springer. 2021, pp. 223–231

Chapter 2

BACKGROUND

In this chapter, a review of the core concepts used within this thesis is performed. Briefly, this includes a definition for the model of TL, an overview of the TLs directly supported in this work, monitoring with TL-based specifications, and parsing techniques and data structures.

2.1 Model for Temporal Logic

When discussing TLs, it is important to define a model of computation (henceforth, “model”) over which the framework operates on. This model for TL is traditionally and widely represented as a Kripke Structure (KS) [57]. A KS is a specialized automata [79] containing a finite set of states, a transition relation, and an interpretation function. The formal definition of the structure is provided below.

Definition 2.1.1 (Kripke Structure)**.** Let AP represent the set of *atomic propositions*. A KS is then represented as a tuple of the following four elements:

$$\mathfrak{M} = \langle S, R, I, \Im \rangle$$

where S is the set of states, $R \subseteq S \times S$ is the transition relation, $I \subseteq S$ is the set of initial states, and $\Im : S \mapsto 2^{AP}$ is the label (interpretation) function that maps a state to the set of valid (i.e., true) atomic propositions from AP.

In following the representation of a model for TL from [38] and [62], the KS is, henceforth, simplified to the following representation when considering the sequence of states as a linear ordering of moments in time that capture the state of the system as a set of atomic propositions that are satisfied:

Definition 2.1.2 (Model for Temporal Logic)**.** In considering a simplifed version of the KS, the resulting model for TL is represented as follows:

$$\mathfrak{M} = \langle s_0, s_1, s_2, ...\rangle$$

where s_i represents a set of valid atomic propositions from AP at moment i.

With this new representation, a specific moment i of the model $\mathfrak{M}$ from the discrete time domain $T \subseteq \mathbb{N}$ (e.g., the state of the system at a particular timestamp) may then be referred to with the following tuple $\langle \mathfrak{M},\ i\rangle$ such that $i \in T$ as will be commonly used when defining the semantics of each TL in Section 2.2.

2.2 Temporal Logics

TL, as it stands, is a branch of philosophical logic concerned with the metaphysical element time. The concept of time is not new and has been studied for centuries. However, most notably, it was not until the formalization of *tense logic* [74, 73] did the application and strength of TL begin to show.

From this temporal framework, many new temporal-based logics have been derived. In categorizing these TLs, there are several possible traits associated with a TL language to describe its capabilities and intended use. The set of mirrored-properties includes (i) propositional versus first-order, (ii) global versus compositional, (iii) branching versus linear, (iv) point-based versus interval-based, (v) discrete versus continuous, and (vi) past versus present [33]. Within this work, the assumed traits of the supported TL are propositional, linear, point-based, and discrete—both past and present modalities are supported.

Furthermore, there are a wide variety of TLs created for a variety of problems. However, this thesis focuses only on a subset of popularly utilized TLs—especially within the field of verification and testing for CPSs. This subset of supported TLs forms a hierarchical structure of dependencies that is illustrated in Figure 2.1 below with Propositional Logic (PL) being the root language.

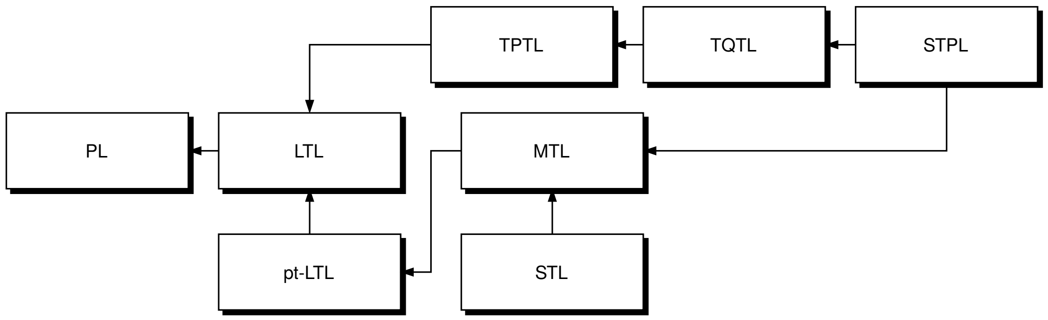


Figure 2.1: Hierarchy of Temporal Logic.

2.2.1 Propositional Logic

Propositional Logic (PL) [18], while not a TL, is the foundation for all other TLs discussed in this work. Therefore, the syntax and semantics of PL formulas are provided as both a formal introduction and reference.

Definition 2.2.1 (Propositional Logic Syntax)**.** The structure of a PL formula φ is inductively defined by the following grammar:

$$\varphi ::= \top \mid \alpha \mid \neg\varphi \mid \varphi_1 \wedge \varphi_2 \mid \varphi_1 \vee \varphi_2 \mid \varphi_1 \Rightarrow \varphi_2 \mid \varphi_1 \Leftrightarrow \varphi_2$$

where $\top$ is the boolean constant *true*, α is a *propositional variable* (atom) from the set of atomic propositions AP, $\neg$ is the unary logical connective *negation*, and $\wedge$, $\vee$, $\Rightarrow$, and $\Leftrightarrow$ are the binary logical connectives *conjunction*, *disjunction*, *implication* (material conditional), and *biconditional* (iff), respectively.

It should also be noted that the boolean constant false $\bot$ is equivalent to $\neg\top$ and may be used interchangeably. Furthermore, in Definition 2.2.1, the syntax for all logical operators are provided for completeness. However, the structure of a PL formula can be minimally captured with the following set of connectives: $\{\neg, \wedge\}$. In other words, the inclusion of $\{\neg, \wedge\}$ provides a *functionally complete* set of logical operators to capture all possible PL expressions. This is also true for negation accompanied with either disjunction or implication.

With the syntax provided, the semantics of a PL formula can now be introduced below in Definition 2.2.2.

Definition 2.2.2 (Propositional Logic Semantics)**.** Let AP be the set of atomic propositions, $\mathfrak{M}$ be the model, and $i \in T$ be the index of the moment of the model. The semantics of a PL formula are then inductively defined as follows:

$$\begin{array}{lll}
\langle \mathfrak{M},\, i\rangle \models \alpha & \text{iff} & \alpha \in \text{AP and } \alpha \in \Im(i) \\
\langle \mathfrak{M},\, i\rangle \models \neg\varphi & \text{iff} & \langle \mathfrak{M},\, i\rangle \not\models \varphi \\
\langle \mathfrak{M},\, i\rangle \models \varphi \wedge \psi & \text{iff} & \langle \mathfrak{M},\, i\rangle \models \varphi \text{ and } \langle \mathfrak{M},\, i\rangle \models \psi \\
\langle \mathfrak{M},\, i\rangle \models \varphi \vee \psi & \text{iff} & \langle \mathfrak{M},\, i\rangle \models \varphi \text{ or } \langle \mathfrak{M},\, i\rangle \models \psi \\
\langle \mathfrak{M},\, i\rangle \models \varphi \Rightarrow \psi & \text{iff} & \text{if } \langle \mathfrak{M},\, i\rangle \models \varphi\text{, then } \langle \mathfrak{M},\, i\rangle \models \psi \\
\langle \mathfrak{M},\, i\rangle \models \varphi \Leftrightarrow \psi & \text{iff} & \langle \mathfrak{M},\, i\rangle \models (\varphi \Rightarrow \psi) \text{ and } \langle \mathfrak{M},\, i\rangle \models (\varphi \Rightarrow \psi)
\end{array}$$

2.2.2 Linear Temporal Logic

Linear Temporal Logic (LTL) [72, 62] is the most fundamental TL considered in this work with its introduction of unbounded temporal-based logical connectives. As it is built on-top of PL, it is sometimes referred to as Propositional Temporal Logic (PTL) in other literature. Within this thesis, however, simply TL will refer to the propositional-based logic unless otherwise stated. In this section, the syntax and semantics of LTL formulas are provided.

Definition 2.2.3 (Linear Temporal Logic Syntax)**.** The structure of an LTL formula φ is inductively defined by the following grammar:

$$\begin{aligned}\varphi ::= {} & \top \mid \alpha \mid \neg\varphi \mid \varphi_1 \wedge \varphi_2 \mid \varphi_1 \vee \varphi_2 \mid \varphi_1 \Rightarrow \varphi_2 \mid \varphi_1 \Leftrightarrow \varphi_2 \\ & \mid \bigcirc \varphi \mid \Diamond \varphi \mid \Box \varphi \mid \varphi_1 \,\mathcal{U}\, \varphi_2 \mid \varphi_1 \,\mathcal{R}\, \varphi_2\end{aligned}$$

where the newly introduced $\bigcirc$, $\Diamond$, and $\Box$ are the unary temporal logical connectives *next*, *eventually* (finally), and, *always* (globally), respectively; and $\mathcal{U}$ and $\mathcal{R}$ are the binary temporal logical connectives *strict until* and *release*, respectively. For all other symbols not mentioned, please see Definition 2.2.1.

From the definition above, the full set of logical temporal operators are provided. However, similar to PL, LTL can be completely captured using a minimal set of temporal operators. The set of operators, as from [72] in its original conception, $\{\bigcirc, \mathcal{U}\}$ form a functionally complete representation of the newly introduced future-based TL operators where all other operators may be defined by these two.

With the syntax of an LTL formula provided, the semantics of the newly defined operators are provided below in Definition 2.2.4.

Definition 2.2.4 (Linear Temporal Logic Semantics)**.** Let $\mathfrak{M}$ be the model, and $i \in T$ be the index of the moment of the model. The semantics of an LTL formula are then inductively defined as follows:

$$\begin{array}{rlll}
\langle \mathfrak{M},\, i\rangle \models & \bigcirc\,\varphi & \text{iff} & \langle \mathfrak{M},\, i+1\rangle \models \varphi \\
\langle \mathfrak{M},\, i\rangle \models & \Diamond\,\varphi & \text{iff} & \exists j \text{ s.t. } j \geq i \text{ and } \langle \mathfrak{M},\, j\rangle \models \varphi \\
\langle \mathfrak{M},\, i\rangle \models & \Box\,\varphi & \text{iff} & \forall j \text{ if } j \geq i,\ \text{then } \langle \mathfrak{M},\, j\rangle \models \varphi \\
\langle \mathfrak{M},\, i\rangle \models & \varphi\,\mathcal{U}\,\psi & \text{iff} & \exists j \text{ s.t. } j \geq i \text{ and } \langle \mathfrak{M},\, j\rangle \models \psi \text{ and} \\
 & & & \qquad \forall k \text{ if } i \leq k < j,\ \text{then } \langle \mathfrak{M},\, k\rangle \models \varphi \\
\langle \mathfrak{M},\, i\rangle \models & \varphi\,\mathcal{R}\,\psi & \text{iff} & \forall j \text{ if } j \geq i,\ \text{then } (\langle \mathfrak{M},\, j\rangle \models \psi \text{ or} \\
 & & & \qquad \exists k \text{ s.t. } i \leq k < j \text{ and } \langle \mathfrak{M},\, k\rangle \models \varphi)
\end{array}$$

where the semantics of all other operators not mentioned can be found in Definition 2.2.2 accordingly.

Past-Time Linear Temporal Logic

Furthermore, while the previously introduced temporal logical connectives from Definition 2.2.3 support operating over the present-/future-tense, an additional set of past-time temporal operator counterparts exist to support operating over the past-tense—known as Past-Timed Linear Temporal Logic (pt-LTL) [52, 22]. For $\bigcirc$, its past-time counterpart is $\odot$ (*yesterday* or *previous*); for $\Box$, its past-time counterpart is $\boxdot$ (*historically*); for $\Diamond$, its past-time counterpart is $\Diamond\!\!\cdot$ (*once*); for $\mathcal{U}$, its past-time counterpart is $\mathcal{S}$ (*since*); and lastly, for $\mathcal{R}$, its past-time counterpart is $\mathcal{T}$ (*trigger*). These operators have the same binding strength (i.e., precedence) as their future-based variants. However, the meaning of these operators differ, so the semantics of the newly introduced pt-LTL operators are defined below as follows.

Definition 2.2.5 (Past-Timed Linear Temporal Logic Semantics)**.** Let $\mathfrak{M}$ be the model, and $i \in T$ be the index of the moment of the model. The semantics of a pt-LTL formula is then inductively defined as follows:

$$
\begin{array}{llll}
\langle \mathfrak{M},\, i\rangle \models & \odot\, \varphi & \text{iff} & i > 0 \text{ and } \langle \mathfrak{M},\, i-1\rangle \models \varphi \\
\langle \mathfrak{M},\, i\rangle \models & \diamonddot\, \varphi & \text{iff} & \exists j \text{ s.t. } 0 \leq j < i \text{ and } \langle \mathfrak{M},\, j\rangle \models \varphi \\
\langle \mathfrak{M},\, i\rangle \models & \boxdot\, \varphi & \text{iff} & \forall j \text{ if } 0 \leq j < i, \text{ then } \langle \mathfrak{M},\, j\rangle \models \varphi \\
\langle \mathfrak{M},\, i\rangle \models & \varphi\, \mathcal{S}\, \psi & \text{iff} & \exists j \text{ s.t. } 0 \leq j < i \text{ and } \langle \mathfrak{M},\, j\rangle \models \psi \text{ and} \\
 & & & \quad \forall k \text{ if } j < k \leq i, \text{ then } \langle \mathfrak{M},\, k\rangle \models \varphi \\
\langle \mathfrak{M},\, i\rangle \models & \varphi\, \mathcal{T}\, \psi & \text{iff} & \forall j \text{ if } j \leq i, \text{ then } (\langle \mathfrak{M},\, j\rangle \models \psi \text{ or} \\
 & & & \quad \exists k \text{ s.t. } j < k \leq i \text{ and } \langle \mathfrak{M},\, k\rangle \models \varphi)
\end{array}
$$

where the semantics of all other operators not mentioned can be found in Definition 2.2.4 accordingly.

2.2.3 Metric Temporal Logic

Metric Temporal Logic (MTL) [55] is a TL developed to capture real-time properties of reactive systems and is an extension to LTL. This capability is possible by extending the unbounded TL operators in Definition 2.2.3 to hold metric information about distance of time added to the traditional model $\mathfrak{M}$ of TL.

Definition 2.2.6 (Metric Temporal Logic Syntax)**.** The structure of an MTL formula φ is inductively defined by the following grammar:

$$
\begin{aligned}
\varphi ::= \top \mid \alpha \mid \neg\varphi \mid \varphi_1 \wedge \varphi_2 \mid \varphi_1 \vee \varphi_2 \mid \varphi_1 \Rightarrow \varphi_2 \mid \varphi_1 \Leftrightarrow \varphi_2 \\
\mid \bigcirc\, \varphi \mid \Diamond_{\mathcal{I}}\, \varphi \mid \Box_{\mathcal{I}}\, \varphi \mid \varphi_1\, \mathcal{U}_{\mathcal{I}}\, \varphi_2 \mid \varphi_1\, \mathcal{R}_{\mathcal{I}}\, \varphi_2
\end{aligned}
$$

where $\mathcal{I}$ represents an interval over the metric information. For all other symbols not mentioned, please see Definition 2.2.3.

Remark. While the initial concept of MTL by [55] provided the ability to define TL requirements with a sense of boundedness, it also permitted singletons in the interval. As showcased by [3], this acceptance of singular time intervals such as $\mathcal{I} = 5$ proved to be undecidable. To resolve this issue, a slight restriction on the interval usage solved this undecidability and introduced a fragment of MTL known as Metric Interval Temporal Logic (MITL). In retaining this benefit, intervals of MTL follow the rules proposed by MITL. Therefore, an interval $\mathcal{I}$ may be *open* (a, b), *closed* $[a, b]$, or *half-closed* $[a, b)$ or $(a, b]$; and it must hold that the bounds $a, b \in \mathbb{R}_{\geq 0}$.

With the syntax provided, the semantics of the newly defined operators are provided below in Definition 2.2.7.

Definition 2.2.7 (Metric Temporal Logic Semantics)**.** Let $\mathfrak{M}$ be the model, $i \in T$ be the index of the moment of the model, and $\mathcal{I}$ be a non-empty, non-singular interval in $\mathbb{R}_{\geq 0}$. The semantics of an MTL formula is then inductively defined as follows:

$$\begin{array}{llll}
\langle \mathfrak{M},\, i \rangle \models & \Diamond_{\mathcal{I}}\, \varphi & \text{iff} & \exists j \in i + \mathcal{I} \text{ s.t. } \langle \mathfrak{M},\, j \rangle \models \varphi \\
\langle \mathfrak{M},\, i \rangle \models & \Box_{\mathcal{I}}\, \varphi & \text{iff} & \forall j \in i + \mathcal{I},\ \langle \mathfrak{M},\, j \rangle \models \varphi \\
\langle \mathfrak{M},\, i \rangle \models & \varphi\, \mathcal{U}_{\mathcal{I}}\, \psi & \text{iff} & \exists j \in i + \mathcal{I} \text{ s.t. } \langle \mathfrak{M},\, j \rangle \models \psi \text{ and} \\
 & & & \quad \forall k \in [i, j),\ \langle \mathfrak{M},\, k \rangle \models \varphi \\
\langle \mathfrak{M},\, i \rangle \models & \varphi\, \mathcal{R}_{\mathcal{I}}\, \psi & \text{iff} & \forall j \text{ if } j \geq i, \text{ then } (\langle \mathfrak{M},\, j \rangle \models \psi \text{ or} \\
 & & & \quad \exists k \in i + \mathcal{I} \text{ s.t. } \langle \mathfrak{M},\, k \rangle \models \varphi)
\end{array}$$

where $i + \mathcal{I}$ produces a new interval $\mathcal{I}'$ shifted to the moment i. Furthermore, the semantics of all other operators not mentioned can be found in Definition 2.2.4.

For Past-Timed Metric Temporal Logic (pt-MTL), operator semantics are defined similarly to Definition 2.2.5 with the exception that new intervals are formed by "looking" backwards (i.e., $I' = i - I$) from the current moment i in the model $\mathfrak{M}$. For example, the pt-MTL formula $\boxdot_{[0,5]}\, p$ informally states that proposition $p \in \mathrm{AP}$ must be true for the last five moments of time for $\mathfrak{M}$ to satisfy the formula.

2.2.4 Signal Temporal Logic

Signal Temporal Logic (STL) [61] is a TL derived as a fragment of MITL. Apart from the same operator support as defined in the previous TLs, STL introduces the *predicate*: a new operand used to define constraints directly on a sequence of moments from the model $\mathfrak{M}$.

Definition 2.2.8 (Signal Temporal Logic Syntax)**.** The structure of an STL formula φ is inductively defined by the following grammar:

$$\begin{aligned}\varphi ::= \top \mid \alpha \mid \mu \mid \neg\varphi \mid \varphi_1 \wedge \varphi_2 \mid \varphi_1 \vee \varphi_2 \mid \varphi_1 \Rightarrow \varphi_2 \mid \varphi_1 \Leftrightarrow \varphi_2 \\ \mid \bigcirc\, \varphi \mid \Diamond_{\mathcal{I}}\, \varphi \mid \Box_{\mathcal{I}}\, \varphi \mid \varphi_1\, \mathcal{U}_{\mathcal{I}}\, \varphi_2 \mid \varphi_1\, \mathcal{R}_{\mathcal{I}}\, \varphi_2\end{aligned}$$

where μ represents a *predicate* from the set of predicates U.

A predicate is effectively a function $\mu : \mathbb{R}^n \to \mathbb{B}$ that returns boolean valuation set of models $\mathfrak{M}_j$ for each $\mu_j \in U$. In other words, for every predicate, an equivalent sequence of boolean valuations for every $i \in [0, T]$ is produced by evaluating the the model $\mathfrak{M}$ at each moment with predicate $\mu_j \in U$. With the syntax provided, the semantics of the newly defined operators are provided below in Definition 2.2.9.

Definition 2.2.9 (Signal Temporal Logic Semantics)**.** Let $\mathfrak{M}$ be the model and $i \in T$ be the index of the moment of the model. The semantics of an STL formula is then inductively defined as follows:

$$\langle \mathfrak{M},\, i\rangle \models \quad \mu \qquad \text{iff} \quad \mu \in U \text{ and } \mu(\langle \mathfrak{M},\, i\rangle) = \top$$

where the semantics of all other operators not mentioned can be found in Definition 2.2.7 accordingly.

2.2.5 Timed Propositional Temporal Logic

Timed Propositional Temporal Logic (TPTL) [4] is another TL for formalizing specifications over the real-time domain. However, the *freeze quantifier* is introduced to provide an alternative method to MTL for bounding temporal operators by binding variables to a specific point in time and defining constraints on those variables.

Definition 2.2.10 (Timed Propositional Temporal Logic Syntax)**.** The structure of a TPTL formula φ is inductively defined by the following grammar:

$$\begin{aligned}\varphi ::= {} & \top \mid \alpha \mid \neg\varphi \mid \varphi_1 \wedge \varphi_2 \mid \varphi_1 \vee \varphi_2 \mid \varphi_1 \Rightarrow \varphi_2 \mid \varphi_1 \Leftrightarrow \varphi_2 \mid \\ & \Diamond\, \varphi \mid \Box\, \varphi \mid \bigcirc\, \varphi \mid \varphi_1 \,\mathcal{U}\, \varphi_2 \mid \varphi_1 \,\mathcal{R}\, \varphi_2 \mid \\ & x.\varphi \mid \pi_1 \leq \pi_2 \mid \pi_1 \geq \pi_2 \mid \pi_1 \equiv_d \pi_2 \mid \pi_1 \not\equiv_d \pi_2\end{aligned}$$

$$\pi ::= x + c \mid c$$

where $x.\varphi$ is the *freeze quantifier*; $\leq$ and $\geq$ are the binary non-strict relational operators *less than* and *greater than*, respectively; $\equiv_d$ and $\not\equiv_d$ are the binary relational operators *not equal to* and *equal to* modulo d, respectively; $x \in V$ is a time-bounded variable; and $c, d \in \mathbb{R}$ are constants. For all other symbols not mentioned, please see Definition 2.2.3.

Expressions of the form $\pi_1 \sim \pi_2$ where $\sim \in \{\leq, \geq\}$ in Definition 2.2.10 are referred to as *timing constraints* and permit bounding subformulas—similar usage to that of intervals in STL and MTL. With the introduction of the freeze quantifier and timing constraints, the semantics of each are reviewed below in Definition 2.2.11

Definition 2.2.11 (Timed Propositional Temporal Logic Semantics)**.** Let $\mathfrak{M}$ be the model, $i \in T$ be the index of the moment of the model, and $\mathcal{E} : V \to T$ be the interpretation (*environment*) for the variables. The semantics of the newly introduced structures of a TPTL formula is then inductively defined as follows:

$$\begin{array}{llll}\langle \mathfrak{M},\, i \rangle \models_{\mathcal{E}} & x.\varphi & \text{iff} & \langle \mathfrak{M},\, i \rangle \models_{\mathcal{E}'} \varphi \\ \langle \mathfrak{M},\, i \rangle \models_{\mathcal{E}} & \pi_1 \leq \pi_2 & \text{iff} & \mathcal{E}(\pi_1) \leq \mathcal{E}(\pi_2) \\ \langle \mathfrak{M},\, i \rangle \models_{\mathcal{E}} & \pi_1 \equiv_d \pi_2 & \text{iff} & \mathcal{E}(\pi_1) \equiv_d \mathcal{E}(\pi_2)\end{array}$$

where $\mathcal{E}'$ represents a new environment that agrees with environment $\mathcal{E}$ on all variable mappings except for the bound variable x such that $x \mapsto i$ (i.e., a new scope $\mathcal{E}'$ is created for the subformula φ limiting the usage of the newly bound variable x), and the other operators $\geq$ and $\not\equiv_d$ are defined similarly to their counterparts. Furthermore, the semantics of all other operators not mentioned can be found in Definition 2.2.4 accordingly where $\models$ is replaced with $\models_{\mathcal{E}}$ denoting a contextualized (i.e., scoped) interpretation over the structure of the environment $\mathcal{E}$.

2.2.6 Timed Quality Temporal Logic

Timed Quality Temporal Logic (TQTL) [29, 13] is a TL used to formulate requirements for perception-based systems such as Deep Neural Network (DNN) with computer vision tasks such as object detection.

Definition 2.2.12 (Timed Quality Temporal Logic Syntax)**.** The structure of a

TQTL formula φ is inductively defined by the following grammar:

$$\begin{aligned}
\varphi ::= {} & \top \mid \alpha \mid \neg\varphi \mid \varphi_1 \wedge \varphi_2 \mid \varphi_1 \vee \varphi_2 \mid \varphi_1 \Rightarrow \varphi_2 \mid \varphi_1 \Leftrightarrow \varphi_2 \mid \\
& \Diamond\,\varphi \mid \Box\,\varphi \mid \bigcirc\,\varphi \mid \varphi_1\,\mathcal{U}\,\varphi_2 \mid \varphi_1\,\mathcal{R}\,\varphi_2 \mid \\
& x.\varphi \mid \pi_1 \leq \pi_2 \mid \pi_1 \geq \pi_2 \mid \pi_1 \equiv_d \pi_2 \mid \pi_1 \not\equiv_d \pi_2 \\
& \exists id@x, \varphi \mid \forall id@x, \varphi \\
\\
\pi ::= {} & x + c \mid c
\end{aligned}$$

where $\exists id@x$ and $\forall id@x$ represent the *existential* and *universal* quantification over the set of unique object identifiers id bounded to the frozen environment x. For all other symbols not mentioned, please see Definition 2.2.10.

From Definition 2.2.12, the *freeze quantifier* from TPTL is more commonly referred to as the *freeze frame quantifier* as moments in the model of time $\mathfrak{M}$ are more accurately contextualized as a discrete linear sequence of frames. With the syntax provided, the semantics of the newly defined operators are provided below in Definition 2.2.13.

Definition 2.2.13 (Timed Quality Temporal Logic Semantics)**.** Let $\mathfrak{M}$ be the model, $i \in T$ be the index of the moment of the model, and $\mathcal{E} : V_i \cup V_{id} \to T$ be the interpretation (*environment*) for both the time V_t and object V_o variables mapped to a moment in T. The semantics of the newly introduced structures of a TPTL formula is then inductively defined as follows:

$$\begin{aligned}
\langle \mathfrak{M},\, i\rangle \models_{\mathcal{E}} \quad & \exists id@x, \varphi && \text{iff} \quad \exists id \in \mathcal{O}_{\mathcal{E}'} \text{ s.t. } \mathcal{O}_{\mathcal{E}'} \neq \emptyset \text{ and } \langle \mathfrak{M},\, i\rangle \models_{\mathcal{E}'} \varphi \\
\langle \mathfrak{M},\, i\rangle \models_{\mathcal{E}} \quad & \forall id@x, \varphi && \text{iff} \quad \forall id \in \mathcal{O}_{\mathcal{E}'} \text{ s.t. } \mathcal{O}_{\mathcal{E}'} \neq \emptyset \text{ and } \langle \mathfrak{M},\, i\rangle \models_{\mathcal{E}'} \varphi
\end{aligned}$$

where $\mathcal{O}_{\mathcal{E}'}$ is the collection of object identifiers from a given freeze environment $\mathcal{E}'$ where $x \mapsto i$. Furthermore, the semantics of all other operators not mentioned can be found in Definition 2.2.11.

2.2.7 Spatio-Temporal Perception Logic

Spatio-Temporal Perception Logic (STPL) [49], similarly to TQTL, is a TL used to formalize requirements for perception-based systems. However, compared to TQTL, STPL additionally supports authoring requirements on spatial properties of the system such as object bounding box comparisons from DNNs.

Definition 2.2.14 (Spatio-Temporal Perception Logic Syntax)**.** The structure of a

STPL formula φ is inductively defined by the following grammar:

$$
\begin{aligned}
\varphi ::= \; & \top \mid \neg\varphi \mid \varphi_1 \wedge \varphi_2 \mid \varphi_1 \vee \varphi_2 \mid \varphi_1 \Rightarrow \varphi_2 \mid \varphi_1 \Leftrightarrow \varphi_2 \mid \\
& \Diamond\, \varphi \mid \Box\, \varphi \mid \bigcirc\, \varphi \mid \varphi_1\, \mathcal{U}\, \varphi_2 \mid \varphi_1\, \mathcal{R}\, \varphi_2 \mid \\
& \exists id@x.\varphi \mid \forall id@x.\varphi \\
& \mathcal{T} - x \sim t \mid \mathcal{F} - x \sim n \mid \mathcal{F} - x \,\%\, c \sim n \\
& id \equiv id \mid id \not\equiv id \mid \theta \mid \boxed{\exists}\, \tau \mid \boxed{\forall}\, \tau \mid \pi
\end{aligned}
$$

$$
\begin{aligned}
\tau ::= \; & \mathcal{C}(id) \mid \overline{\tau} \mid \tau_1 \sqcap \tau_2 \mid \tau_1 \sqcup \tau_2 \mid \mathbf{I}\, \tau \mid \mathbf{C}\, \tau \\
& \Diamond^s_{\mathcal{I}}\, \tau \mid \Box^s_{\mathcal{I}}\, \tau \mid \bigcirc^s_{\mathcal{I}}\, \tau \mid \tau_1\, \mathcal{U}^s_{\mathcal{I}}\, \tau_2 \mid \tau_1\, \mathcal{R}^s_{\mathcal{I}}\, \tau_2 \mid
\end{aligned}
$$

$$
\begin{aligned}
\theta ::= \; & \mathbf{dist}(id_1, \kappa_1, id_2, \kappa_2) \sim r \\
& \mathbf{lat}(id, \kappa) \sim r \mid \mathbf{lon}(id, \kappa) \sim r \mid \mathbf{lat}(id_1, \kappa_1) \sim r \times \mathbf{lat}(id_2, \kappa_2) \mid \\
& \mathbf{lon}(id_1, \kappa_1) \sim r \times \mathbf{lon}(id_2, \kappa_2) \mid \mathbf{lat}(id_1, \kappa_1) \sim r \times \mathbf{lon}(id_2, \kappa_2) \\
& \mathbf{area}(id) \sim r \mid \mathbf{area}(id_1) \sim r \times \mathbf{area}(id_2) \\
& \mathbf{class}(id) \equiv c \mid \mathbf{class}(id) \equiv \mathbf{class}(id) \\
& \mathbf{prob}(id) \sim r \mid \mathbf{prob}(id_1) \sim r \times \mathbf{prob}(id_2)
\end{aligned}
$$

$$
\pi ::= \mathbf{area}(\tau) \sim r \mid \mathbf{area}(\tau_1) \sim r \times \mathbf{area}(\tau_2)
$$

$$
\kappa ::= \mathrm{LM} \mid \mathrm{RM} \mid \mathrm{TM} \mid \mathrm{BM} \mid \mathrm{CT}
$$

where $\mathcal{T} - x \sim t$ is a *time constraint* such that $\mathcal{T}$ is the current time, $x \in V_t$ is a freeze quantified time variable, and $t \in T$; $\mathcal{F} - x \sim n$ is a *frame constraint* such that $\mathcal{F}$ is the current frame, $x \in V_t$ is a freeze quantified object variable, and $n \in \mathbb{N}$; $\mathcal{F} - x \,\%\, c \sim n$ is a *frame constraint modulo some constant* $c \in \mathbb{N}$; $id \equiv id$ and $id \not\equiv id$

are relational equivalence operators *equal to* and *not equal to* on the object identifiers $id_1, id_2 \in V_o$, respectively; $\boxed{\exists}$ and $\boxed{\forall}$ are the unary spatial quantifiers *nonempty* and *universe*, respectively; $\mathcal{C} : V_o \rightarrow \Pi$ is a mapping between the set of object identifier variables V_o and spatial terms $\Pi \subseteq \langle \mathbb{U}, \mathbf{I} \rangle$ such that $\mathbb{U}$ is the universe (as a set) of the space, and $\mathbf{I}$ is the *interior* operator; $\overline{\tau}$ is the set theoretic *complement*; $\sqcap$ and $\sqcup$ are the binary set theoretic operators *conjunction* (intersection) and *disjunction* (union), respectively; $\mathbf{C}$ is the *closure* operator; $\Diamond^s_{\mathcal{I}}$, $\Box^s_{\mathcal{I}}$, $\bigcirc^s_{\mathcal{I}}$, $\mathcal{U}^s_{\mathcal{I}}$, and $\mathcal{R}^s_{\mathcal{I}}$ are the spatio-temporal operators *eventually*, *always*, *next*, *until*, *release*, respectively—where $\mathcal{I}$ represents a temporal property and s represents a spatial property; and **dist**, **area**, **lat**, **lon**, **class**, and **prob** represents a function to evaluate distance between two object identifier anchors, area of an object (represented as a set), latitudinal position of an object anchor, longitudinal position of an object anchor, classification (as a number) of an object identifier, and confidence of an object identifier, respectively. For all other symbols not mentioned, please see Definition 2.2.12.

As STPL introduces several new operators over its predecessor TQTL along with additional information annotated to the model $\mathfrak{M}$, the semantics for the spatio-temporal framework are left for review in [49].

2.3 Specification-Based Monitoring

Specification-based monitoring is a popular method in the verification and testing community for evaluating complex or overly large systems where formal methods and techniques are impractical [14]. Informally, the process of monitoring involves accepting a specification and model of the system under testing, and from this determines whether the model provided satisfies (i.e., does not violate) to the specification. A model $\mathfrak{M}$ is said to satisfy a requirement φ if $\langle \mathfrak{M}, 0 \rangle \models \varphi$ (i.e., if all moments of the the system starting from the initial state satisfy the requirement φ). In this work, TL-based specifications are used as the formal requirement to specify system constraints to monitor.

In practice, monitoring takes two forms: *online monitoring* and *offline monitoring* [27]. These two forms may also be referred to as *synchronous* and *asynchronous* monitoring [75]. The difference between the two methods is most distinctly represented by the presence of an infinite or finite representation of the system under test. Online monitoring evaluates events as they arrive and process the satisfaction criterion whereas offline monitors have access to the entire system execution at runtime.

In this thesis, the FoRek library is tested with offline monitors only. However, an extension to the online domain is feasible as the library does not have any assumptions regarding online versus offline usage but is not verified.

2.4 Parsers and Tree Data Structures

As the major contribution of this work is a collection of parsers and translators, a high-level overview of these processes is provided. This section does not serve as a comprehensive guide to parsing and translating techniques but rather is purposed in motivating the necessity of each.[1]

The process of parsing is based in formal *language recognition* techniques [79] and involves determining whether an input (usually in the form of a *string*) is correct. By correct, it is meant that the input string w_i falls into the set of possible strings recognized by some (finite or infinite) language $\mathcal{L} = \{w_1, w_2, w_3, ...\}$.

While language recognition is a fundamental of parsing, parsing not only serves to identify whether some input conforms to the set of strings. In practice, parsing provides a method to extract information from the string and transform it into a more

[1]For those interested in a more formal introduction to parsing, translating, and compiler theory in general, please see the popular "Dragon Book" [1].

easily interpretable structure. At an implementation-level, this procedure of parsing a string often serves as an umbrella term to capture two separate but equally necessary stages of language recognition: (i) *lexical analysis* and (ii) *syntactic analysis.* These two components makeup the complete process of parsing an input and have different responsibilities that enable recognition and transformation of strings.

In the context of this work, an input string may be viewed as a finite sequence of characters (e.g., "{a, b, c}") from the alphabet Σ of the language $\mathcal{L}$. The responsibility of the lexical analyzer (henceforth, *lexer*) is to *tokenize* (i.e., group) sequences of characters into lexical units (henceforth, *tokens*). The syntactic analyzer (henceforth, *parser*) is then responsible to validate the ordering of these tokens to ensure they follow the structure of the defined language—commonly captured as a Context-Free Grammar (CFG) [79]. A visualization of this workflow is shown in Figure 2.2.

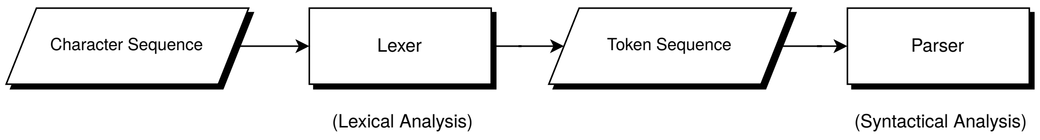


Figure 2.2: Lexer and parser workflow.

As an example, consider the STL requirement from Equation (1.1) below in Example 2.4.1. To effectively interpret this formula, several items must be considered. First, identifying the individual operators and operands to be consolidated into tokens, considering precedence rules for correct evaluation, and ensuring binding strengths of operators for readability.

Example 2.4.1. In a left-to-right fashion, the set of tokens produced from the formula $\Box\ (req\ \Rightarrow\ \Diamond_{[0,10]}\ gnt)$ by the lexer would be '$\Box$' followed by '(' followed by 'req' followed by '$\Rightarrow$' followed by '$\Diamond_{[0,10]}$' followed by 'gnt' followed by ')'. The parser then accepts this stream of tokens and ensures that the ordering of each is correct and possible according to an STL formula as defined in Definition 2.2.8.[1]

A particularly important aspect in the practice of parsing a string is to transform its representation into a more interpretable structure to use for various applications and post-processing. The most common structure that captures important details of a string such as precedence and dependency is the *graph* structure. The design, advantages, and layout of this structure used in this work are provided in Section 2.4.1 as follows below.

2.4.1 Abstract Syntax Tree

An Abstract Syntax Tree (AST) is a graph-based structure popularly used throughout parsing and compiling and found in many language implementation patterns [68]. The advantage of an AST is in its ability to capture formal language constructs as intepretable structures that retain only the needed information. This information includes only relevant operations, precedence, or dependencies and does not contain any irrelevant items such as whitespace, newlines, or comments.

As previously mentioned, ASTs are graphs. Therefore, to provide a baseline representation of the AST leveraged in this work, a formal introduction of the data structure is provided. The formalization of a graph [59], for completeness, is defined as follows below:

Definition 2.4.1 (Graph)**.** A graph G is an ordered pair consisting of a nonempty set of vertices (nodes) V and a nonempty set of edges E and is represented as:

$$G = (V, E)$$

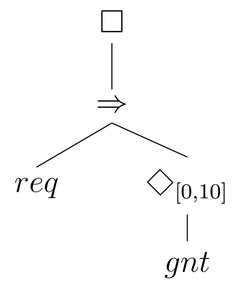
[1]It is recognized that parentheses ‘ (’ and ‘)’ are not formally supported in Definition 2.2.8. However, in practice support for parentheses is a common occurrence and thus used in the examples without major justification needed.

such that each edge $e_i \in E$ is a two-element subset of V indicating a connection between two vertices. In other words, $e_i = \{v_i, v_j\}$ such that $v_i, v_j \in V$.

The definition of a rooted tree used to represent the Intermediate Representation (IR) of a TL formula is specified as an *acyclic connected graph* where one node is labeled the root (i.e., the starting point). Furthermore, within the implementation, this graph structure is a directed graph meaning there is an edge from the parent to the child and not vice versa. With all these restrictions in mind, the final representation of the IR is formalized as a rooted binary tree as defined in [42] which may be viewed as a specialized Directed Acyclic Graph (DAG) where all internal nodes refer to operations from the formal language, and all leaf nodes refer to operands.

In continuing with the Example 2.4.1, the parser is responsible for generating the AST that captures the relevant information, and the lexer is responsible for discarding any irrelevant information that the parser does not care for.

Example 2.4.2. From the set of tokens generated from $\Box\,(req \;\Rightarrow\; \Diamond_{[0,10]}\, gnt)$, the resulting AST is as follows:



From the resulting AST, several properties are apparent. First, the parentheses originally present in the formula are dropped as precedence of operations is captured by the level and positioning of individual nodes (i.e., the subformula $req \;\Rightarrow \Diamond_{[0,10]}\, gnt$ must be resolved before resolving $\Box$ as it is positioned below). Furthermore, as $\Box$ is a unary operation, it has only connecting node representing the subformula $req \;\Rightarrow \Diamond_{[0,10]}\, gnt$

whereas $\Rightarrow$ is a binary operation, so it has two connecting nodes representing the left subformula req and the right subformula $\Diamond_{[0,10]}\ gnt$.

From the example above, the resulting AST structure may then be used for monitoring and testing activities as the formal requirement is clearer to interpret, reason with, and traverse compared to an uncategorized sequence of characters with no inherent meaning. In summary, the lexer identifies the individual components and the parser captures and represents these units in a meaningful manner.

Chapter 3

LITERATURE REVIEW

In this chapter, a review of the reliance on TL as a specification language is performed. This review considers the application of the temporal framework in the context of simulation-based testing and verification—importantly, the methods of integrating a TL specification into its application are highlighted. In addition, a review of TL-based formal requirement frameworks, interfaces, and elicitation tools is performed to highlight the current support for TL requirements authoring.

3.1 Temporal Logic as a Specification Language

TL is a well accepted formalism for describing system properties related to time. The use of this framework is especially prevalent in simulation-based approaches to verifying model satisfaction against a TL specification [53].

Falsification is popular activity from simulation-based approaches that leverage TL requirements. In falsification, a requirement is used to guide the system (i.e., model) under consideration towards violating said requirement [34] (i.e., find an input counterexample). These requirements in implementations are commonly formulated as TL formulae leading to a strict dependence on the framework as the specification language to describe behavior to falsify. Examples of falsification tools for hybrid systems include the following: S-TaLiRo [10] which utilizes MTL requirements, BREACH [31] which utilizes MITL requirements, FalStar [35] which utilizes STL requirements, and more recently Ψ-TaLiRo [80] which utilizes STL requirements. Furthermore, additional falsification tools leveraging TL as the specification language can be found in [63, 81, 2, 83]. While this list of TL-based tools is not exhaustive, the list warrants the impact and use TL has.

The aforementioned tools demonstrate an application of TL. However, these tools primarily provide the searching capabilities to find inputs that falsify a system. To guide the search towards falsifying behavior, a proper metric describing the satisfaction of a TL requirement against a system trace (i.e., cost) must be selected and then evaluated accordingly. This evaluation and metric computation is most commonly performed by monitors [14]. In addition, the responsibility of inferring TL requirements into monitorable requirements is commonly given to the monitors. As there are numerous monitors for various applications, purposes, and contexts, the selection of reviewed tools revolve around their handling of TL requirements and methods of interfacing.

The most primitive form of handling TL requirements is to require the tool/user to author the formula as a custom data structure as defined by the tool. This method is used by tools such as PERCEMON [12], a C++ tool used for monitoring perception algorithms with TQTL specifications, or TLTK [25], a Python tool used for monitoring hybrid systems with STL requirements. This approach provides the advantage that requirements may be type checked (by either the compiler or some static analyzer) and are therefore provably syntactically correct. However, this has several disadvantages from a purely interface perspective: (1) the set of data structures for each tool must be learned regardless if the same TL framework is used, (2) the tool is not easily interoperable with other programming languages, and (3) cannot be easily batched.

An improvement over the previous in terms of interfacing with TL requirements is by introduction of a custom parser to parse and produce a data structure from a TL string. This approach is highly prevalent in the set of monitors from the S-TALIRO toolbox and is a large driving force behind this work. The S-TaLiRo toolbox supports four major monitors: (1) FW-TaLiRo [36] for monitoring MTL requirements using a forward rewriting algorithm, (2) DP-TaLiRo [82] for monitoring MTL requirements using a dynamic programming algorithm, (3) TP-TaLiRo [30] for monitoring TPTL requirements, and (4) STPL-TaLiRo [49] for monitoring STPL requirements. All four monitors are based from the same parsing algorithm and structure that has been manually developed, managed, and extended. The internal parser developed performs an iterative cycle over the input string to group lexical units to be parsed. Furthermore, the BREACH tool [31] parses a BREACH requirement utilizing pattern matching through *regular expressions*. These approaches lower the barrier to writing TL requirements by utilizing parsing a string into the needed data structure, so tools/users can abstract away without concern for data structures used. However, a large limitation of the approach in building a custom parser is in its extensibility, interoperability, and reliability. The tools mentioned perform rudimentary syntax checks and error reporting upon parsing an ill-formed TL requirement leading to longer debugging times upon facing such obstacles.

Lastly, the development of modular monitors attempt to provide TL interfaces that are extensible and customizable through organized parsing frameworks. This approach is best demonstrated in RTAMT [66] which provides an online and offline monitor for evaluating STL formulae. Similarly to this work, the ANTLR tool is used to generate a custom STL parser and lexer. This allows the focus of extensibility to be on the semantic interpretations rather than the syntactic. The approach in RTAMT, however, contains parsing framework that is tightly coupled to the core infrastructure of the tool as it is not intended for general monitoring applications. Therefore, tool interoperability is limited.

In reviewing the use of TL and approaches to writing TL requirements, various methods exist to parse and effectively represent a TL formulae to interpret. This large variety, however, limits the ability of tools to interact and be easily compared. In addition, while supporting a custom parser provides an increased ease-of-use, the maintainability and extensibility begins to become more of a burden. Approaches to modularizing and creating easily customizable parsers and semantic frameworks have been performed, however, have not been intended to be generalized.

3.2 Elicitation of Formal Requirements

The process of capturing system requirements into an understandable, accurate, and expressive medium is an ongoing problem. In working towards a standard to capture said requirements, various formal requirement languages have been proposed to help streamline requirement elicitation and provide a baseline class of properties commonly required for capturing system behavior. Requirement-based languages such as the Property Specification Language (PSL) [51] derived from Sugar [15] or Property Specification Pattern (PSP) [32] were proposed to capture a set of commonly occurring patterns in requirement elicitation [43]. Another approach that takes a more visual approach to defining system specifications is known as *statecharts* [46] that gained much popularity in defining requirements using a sequence of state evolutions depicted visually. Further contributions to visually representing requirements inspired from statecharts include Live Sequence Chart (LSC) [26] and its predecessor Message Sequence Chart (MSC) [76].

An important design principle in creating formal requirement languages and interfaces is in the semantics and computability of the proposed framework—i.e., the requirement language should be able to be automated and represented in a program that can test the system against the requirement accurately and reliably. While there have been various tools and formalizations developed to support these frameworks such as [45, 44] for PSL specifications, [47] for statecharts, and [5] for MSCs, TL-based frameworks has shown a wide variety of computational support, interpretation, and application. Therefore, although the formalism inhabits a higher learning curve that the other proposed solutions attempt to lower, it has prevailed because of its expressive power. In authoring TL-based requirements, approaches from Natural Language Processing (NLP) to Graphical User Interface (GUI) to capture the specifications have been proposed. For the remaining contents of this section, the various approaches to interfacing and eliciting TL requirements are reviewed.

The most recent development in capturing TL specifications has been through the use of NLP [19]. Development of recent tools such as DeepSTL [48] for *semantic parsing* of STL requirements, and [24, 40] that leverage large language models for extracting NL statements into LTL formulae is a newly focused problem. The proposed advantage is in relaxing the bar to entry in formalizing system requirements without losing the formal method assurances of TL frameworks. However, as these approaches utilize a Machine Learning (ML) approach, a confidence threshold is placed on the resulting translation leading to cases of NL statements that do not have a well-formed TL formula, accordingly.

To circumvent the possibility of confidence and assumption of translations between NL to TL, alternatives to retaining NL-based structure and guaranteed translations are with the use of structured NL grammars. Implementations such [37, 56, 11] define a formal grammar from a subset of NL statements that then translate into a valid TL requirement reliably. While this approach provides syntactic sugar over a TL expression as NL statements, the restriction is higher. Therefore, ill-formed NL requirements do not have a valid translation, purposely.

In veering away from NL-based approaches, the use of GUIs to elicit TL requirements have been demonstrated in several applications successfully [58]. In [50], for example, introduced the tool VISPEC along with a usability study showcasing the practicality of GUI-based requirement elicitation tools.

Chapter 4

A FORMAL REQUIREMENTS TOOLKIT

In this chapter, the Formal Requirements Toolkit (FoRek) is presented. While the results from this work is a software library, this thesis reserves code implementation-level details for the library found online at [7] and focuses on the design, justification, and structural aspects of the toolkit. This includes parsing, representing, translating, and interpreting TL requirements—for a review of supported TLs in this work, see Section 2.2 accordingly.

FoRek is a library for recognizing, translating, and interpreting TL formulas. This procedure, as discussed in Section 3.1, is an important and often repeated pattern to effectively utilize TL as a specification language in monitoring, testing, and any other domain having to manage TL formulae. Therefore, this toolkit hopes to merge the current gap of having to self-manage TL recognizers and validators—and doing so efficiently, safely, and reliably. In short, if a TL specification needs parsing or interpreting, FoRek should be used.

As a general overview of the library, FoRek has one input and one output. As input, it accepts a TL formula as a string; and as output, it returns the IR of the formula. An overview the FoRek architectural makeup is highlighted in Figure 4.1. This includes the set of custom TL parser, IR builder, and TL translator procedures along with their associated error check points. These error points produce custom runtime errors associated to the context at which the error was thrown and should be effectively managed by the *external interface.*

From Figure 4.1, there are six process blocks and two input/output blocks. Three

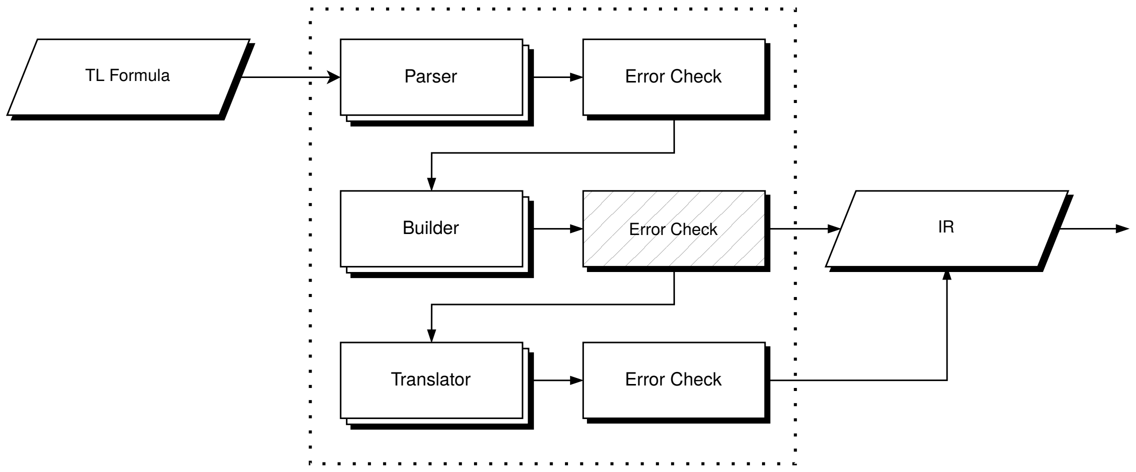


Figure 4.1: Overview of FoRek architecture.

process blocks are responsible for data transformation, and the other three are responsible for runtime error checking. Stacked process blocks are indicative of several separate procedures that perform similar jobs. For example, the stacked parser process block is representative of the set of parsers associated with the various flavors of TL supported—likewise for the builder and translator blocks.

The parser is the first process block in FoRek that accepts the TL formula and parses the structure. The result of this process is sent to the first error checker responsible for checking and throwing parsing-related runtime errors. If no errors are found, then the result is sent to the builder process block to build the IR. This result is then sent to an optional (denoted by hatch lines) error checker that may perform TL language-level assertions and checks such as variable scope rules before returning a valid IR assuming no errors. Additionally, after the builder produces a valid IR, the result may also be translated to an equivalent TL formulae to be interpreted over. This translation between TLs procedure is optional and is not fully supported between all TLs. However, it is useful when implementing a new TL is unreasonable and improves interoperability between tools utilizing different flavors of compatible TL. If used, the resulting translation is also checked for any translation-related runtime errors such as incompatible mappings.

Remark 4.0.1 (Runtime Errors Justification). As shown in Figure 4.1, there are several points where an error must be effectively caught by the external interface without otherwise facing unpredictable runtime termination. While it is not desirable to entrust this reliable behavior cost onto the external interface, parsing TL formulas is not a task that can fail quietly. If ill-formed TL requirements were partially parsed to produce an IR that is not truly representative of the intended specification, then the system under test cannot be confidently deemed robustly safe. Therefore, when parsing a TL formula goes awry, the error must be promptly thrown and left to the external interface on how to proceed. In any case, it is assured that the resulting IR would not be representative of the TL if incorrectly formulated.

For the rest of this chapter, a detailed review of the design for each aforementioned process block is performed. It should also be established that while this work results in the support for six total TLs, the work itself of supporting each is similar in design and implementation. Therefore, each concept, component, and procedure is introduced under the assumption that it applies unanimously to all currently supported TLs unless otherwise stated.

4.1 Recognizing Temporal Logic

Parsing is the first major process in recognizing TL formulae. In order to accurately synthesize the IR of a TL formula, the individual components must be recognized and structured according to its CFG. As a contribution of this work is a TL-based toolkit that is extensible, developing and managing a custom set of parsers and lexers is an impractical investment of time and resources. Therefore, the management of building and maintaining the parser framework is left to ANother Tool for Language Recognition (ANTLR)—details can be found in Appendix A. This selection between an automatically generated parser and a custom parser framework is usually simplified down to two considerations: (1) efficiency and optimization during parse-time, and (2) capabilities of the parser generator to capture the target language. In the context of this work, parsing a TL formula is a single action performed in a one-time pass fashion to produce the needed IR to interpret or translate thereafter. Therefore, while efficiency and optimization is important, it is not a critical consideration in the development of this toolkit in its current version. Furthermore, compared to the competing parser candidates reviewed in Section 3.1, ANTLR generated parsers already support adaptable error listeners, parser recovery methods, and importantly a traditional and reliable language recognition technique without any caveats [69]. In regard to the capabilities of a parser generator, TL is a formal language [21] inductively defined, so ANTLR is able to capture said language without any needed parsing adaptations or obscure methods.

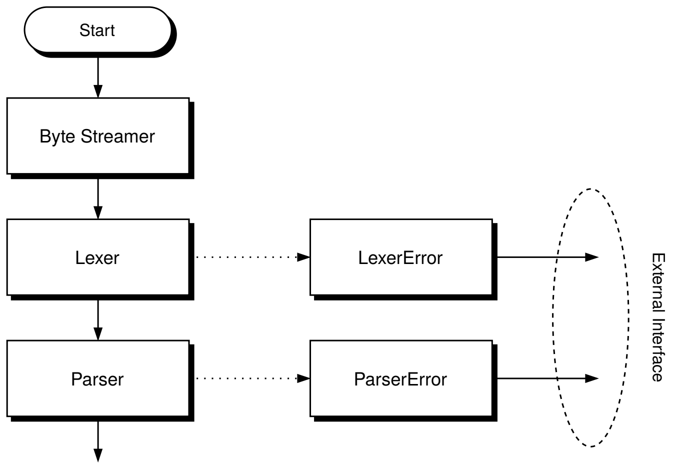


Figure 4.2: Overview of FoRek parsing framework.

In parsing TL formula, the process accepts the TL string as input and returns a *syntax tree*. A detailed overview of the parsing procedure is illustrated below in Figure 4.2. The parsing framework contains three procedures as well as two possible runtime errors that are highly suggested to be caught by the *external interface* without risking program failure or crash.

At the start, the string is first sent through to the byte streamer which encodes each character as an equivalent byte numerical representation. From this sequence of bytes, the lexer performs lexical analysis and generates a stream of tokens. From this stream of tokens, the parser produces a resulting syntax tree. The separation of steps that mimic compiler-based infrastructures allows individual components to be clearly responsible for a single task compared to performing lexical analysis, parsing, and IR creation in one combined step which can be cumbersome to manage, extend, and differentiate.

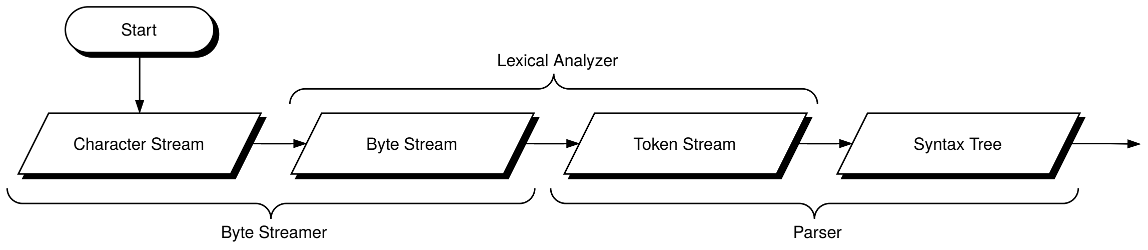


Figure 4.3: Transformation of data in FoRek.

This transformation of data, as illustrated in Figure 4.3 along with the associated process blocks, is handled by ANTLR through its interfaces thus simplifying the task of managing input and output operations, internal data structure representations, and serializable recognizers.

4.1.1 Grammar Organization

As ANTLR is the selected tool to generate parsers capable of capturing the syntax of TL formulas, the syntax must be composed into a readable format by ANTLR. This process of capturing the syntax through ANTLR is done by defining a lexer grammar and a parser grammar in its *meta-language* [69]. While ANTLR supports combined grammars—that is, grammars that define both lexer and parser rules in one file, the selection to separate lexical definitions from syntactic definitions is purposefully done to improve readability, reduce complexity, and encourage reusability.

The organization of the set of TL grammars follows a similar lineage as of Figure 2.1. This design allows for token and parser definitions to be re-introduced without redefinition and is inspired by the natural dependency of the supported TLs. The resulting organization of the set of ANTLR grammars designed is illustrated in Figure 4.4 below where solid fill indicates grammars with a parser and corresponding lexer file, and hatch fill indicates the presence of only a lexer grammar.

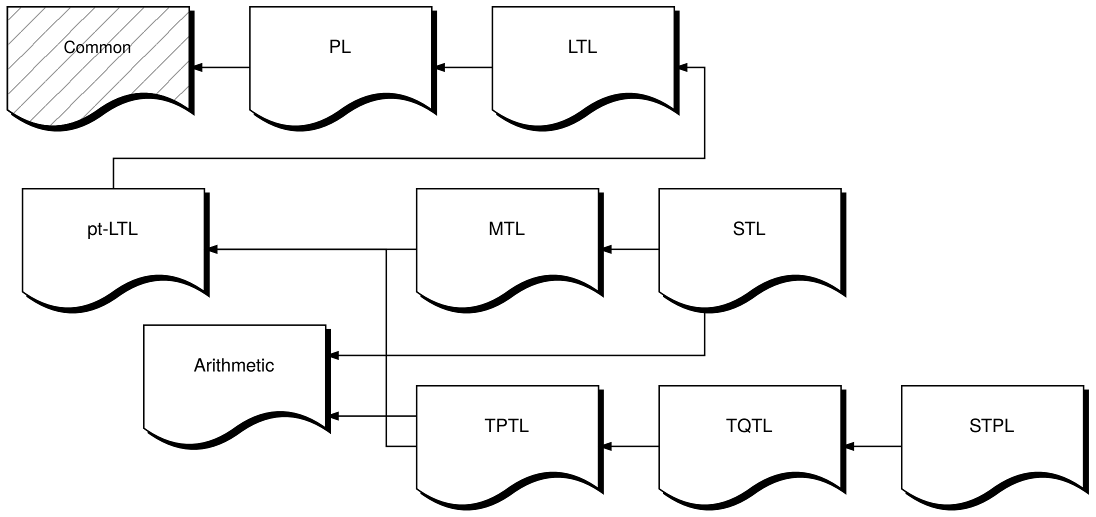


Figure 4.4: Organization of ANTLR grammars in FoRek.

The organization of the currently supported set of TLs is divided into ten logically separate ANTLR grammars. When considering both the parser and grammar, a total of 19 grammar files have been defined to support the syntax of each TL. Unique to this implementation-level structure, there are two grammars not explicitly mentioned in Figure 2.1: the `Common` and `Arithmetic` grammars. These grammars are separated as opposed to implemented in the TL grammars directly as their their dependency is either split between two non-dependent grammars or are logically considered independent enough to not be merged into another grammar.

With this dependency, defining new grammars is as simple as depending on the definitions of a subset grammar and adding new rules as needed. For example, if a new TL were to be developed based on STPL, then importing the STPL grammar to the new corresponding lexer and parser grammars provides all the tokens and parse rules. Furthermore, ANTLR supports importing from multiple targets—as demonstrated from TPTL and STL in Figure 4.4. Therefore, reusing syntax rules from multiple sources is allowed and highly encouraged.

4.1.2 Lexer, Tokens, and Symbols

The first major component in recognizing TL specifications is the lexer. The lexer is responsible for partitioning the stream of bytes into corresponding tokens that the parser processes to develop the structure of the formula. The lexer does not consider the structure of the input but simply categorizes sequences of bytes into discernible token definitions, accordingly. At this stage, tokens that are unused by the parser are discarded. This includes whitespace, newlines, or comments.

As alluded to in Section 4.1.1, there are ten total lexer grammars to define the set of tokens associated with each newly introduced language. Of these lexer grammar files, the `Common` is most elementary grammar file. This grammar file provides token definitions for commonly used tokens throughout all languages and grounds the set of fundamental symbols, so any new language has access and uses the same token names for continuity. As previously mentioned, an important responsibility of the lexer is to discard tokens that are not cared for at later stages. The `Common` grammar provides three definitions in Table 4.1 that are captured and effectively thrown out in all deriving grammars. This set of discardable tokens include whitespace and newlines, line comments, and comment blocks.

As these discarded tokens are defined in the `Common` lexer grammar, all dependent

Token	**Pattern**	**Example**
Whitespace	`[ \t\r\n]+`	
Block Comment	`'/*' .*? '*/'`	`/* a block comment */`
Line Comment	`'//' .*? '\r'? '\n'`	`// a line comment`

Table 4.1: Set of discarded token definitions in FoRek.

grammars also discard these tokens by default. For a complete review of the set of token definitions in `Common`, see Appendix B.

From the `Common` lexer grammar, TL-specific and supporting lexers are defined. The process of creating derived grammars involves two major steps: (1) importing the parent grammar, and (2) introducing new token definitions.

It should be highlighted that a minor issue introduced by the syntaxes in Section 2.2 is the use of symbols and notations for operations are not easily transcribed when considering the American Standard Code for Information Interchange (ASCII) [60]. For example, the symbolic representation for the temporal operator *always* is $\Box$ which is not a supported ASCII character. Therefore, to resolve this issue, ASCII-compliant token definitions are provided for each newly introduced TL operation. For each token definition, a short form and long form are provided. The selection of these token definitions for each are based on the commonly used definitions in other literature such as [22, 10, 66].

From the supported logics as represented in Figure 2.1, six introduce operators that require an ASCII-equivalent notation. This includes PL, LTL, pt-LTL, TPTL, TQTL, and STPL. The branches of logics for STL and STL are supersets of LTL without introducing any new symbols. Therefore, additional ASCII definitions are not required for either as both are captured from the LTL token definitions. For a detailed review of all token definitions and not only the operations, please see Appendix B, accordingly.

Propositional Logic

Within Table 4.2, PL introduces five operators that require equivalent ASCII symbols. This includes the *logical negation*, *conjunction* (logical and), *disjunction* (logical or), *implication*, and *biconditional* operators.

Operator	**Symbol**	**Short Form**	**Long Form**
Logical Negation	$\neg$	`!`	`not`
Logical And	$\wedge$	`&&`	`and`
Logical Or	$\vee$	`||`	`or`
Implication	$\Rightarrow$	`->`	`implies`
Biconditional	$\Leftrightarrow$	`<->`	`iff`

Table 4.2: ASCII token definitions for PL operators.

Linear Temporal Logic

Within Table 4.3, LTL introduces five operators that require equivalent ASCII symbols. This includes the temporal operators *eventually*, *always*, *until*, and *release*.

Operator	**Symbol**	**Short Form**	**Long Form**
Eventually	$\Diamond$	`F`	`eventually`
Always	$\Box$	`G`	`always`
Next	$\bigcirc$	`X`	`next`
Until	$\mathcal{U}$	`U`	`until`
Release	$\mathcal{R}$	`R`	`release`

Table 4.3: ASCII token definitions for LTL operators.

Past-Time Linear Temporal Logic

Within Table 4.4, pt-LTL introduces five operators that require equivalent ASCII symbols. This includes the past-time temporal operator counterparts *once*, *historically*, *previous*, *since*, and *trigger*.

Operator	**Symbol**	**Short Form**	**Long Form**
Once	⟐	`O`	`once`
Historically	$\boxdot$	`H`	`historically`
Previous	$\odot$	`P`	`previous`
Since	$\mathcal{S}$	`S`	`since`
Trigger	$\mathcal{T}$	`T`	`trigger`

Table 4.4: ASCII token definitions for pt-LTL operators.

Timed Propositional Temporal Logic

Within Table 4.5, TPTL introduces a single operator that requires an equivalent ASCII symbol. This includes time-based *freeze quantifier* where `<var>` is replaced with a valid `Identifier` as defined in Appendix B.

Operator	**Symbol**	**Short Form**	**Long Form**
Freeze Time Quantifier	$x.$	`@<var>`	`at <var>`

Table 4.5: ASCII token definitions for TPTL operators.

Timed Quality Temporal Logic

Within Table 4.6, TQTL introduces a two new operators that require equivalent ASCII symbols. This includes frame-based *exists freeze quantifier* and *forall freeze quantifier.* where `<id>` and `<var>` are valid `Identifier` as defined in Appendix B.

Operator	**Symbol**	**Short Form**	**Long Form**	
Exists Freeze Frame	$\exists id@x$	`E(<id>...)@<x>`	`exists(<id>...)`	`at <x>`
Forall Freeze Frame	$\forall id@x$	`A(<id>...)@<x>`	`forall(<id>...)`	`at <x>`

Table 4.6: ASCII token definitions for TQTL operators.

Within Table 4.7, STPL introduces 12 new operators that require equivalent ASCII symbols. This includes *spatial exists*, *spatial forall*, topological *complement*, *intersection*, *union*, *interior*, *closure*, *eventually*, *always*, *next*, *until*, and *release*.

Operator	**Symbol**	**Short Form**	**Long Form**
Spatial Exists	$\boxed{\exists}$	`SE`	`nonempty`
Spatial Forall	$\boxed{\forall}$	`SA`	`universe`
Complement	$\overline{}$	`!`	`not`
Intersection	$\sqcap$	`&&`	`and`
Union	$\sqcup$	`||`	`or`
Interior	$\mathbf{I}$	`I`	`interior`
Closure	$\mathbf{C}$	`C`	`closure`
Spatial Eventually	$\Diamond^s$	`F`	`eventually`
Spatial Always	$\Box^s$	`G`	`always`
Spatial Next	$\bigcirc^s$	`X`	`next`
Spatial Until	$\mathcal{U}^s$	`U`	`until`
Spatial Release	$\mathcal{R}^s$	`R`	`release`

Table 4.7: ASCII token definitions for STPL operators.

From a brief observation of Table 4.7, the tokens definitions associated with the topological set operations are equivalent to those of the logical operations in Table 4.2. However, this is purely a lexical design and does not add any ambiguity to an STPL formula as defined in Definition 2.2.14 and will be further discussed in Section 4.2.

4.1.3 Parser and Operator Precedence

The parser is the last step and most known for realizing the structure of a TL formula. Similar to the lexer, there are nine defined parser grammar rules that capture the syntax of each TL as well as one for arithmetic operations. For the complete syntax for each grammar, see Appendix B.

The structure of a TL parser grammar is similarly designed across all supported languages to easily read and reuse components. Each grammar definition has a `start` rule that expects a `formula` followed by an `EOF` token. The expectation of `EOF` is required to terminate parsing safely. Furthermore, for each TL, the `start` rule must be redefined to match the respective definitions introduced throughout Section 2.2 as this is a limitation of ANTLR that does not allow merging two definitions through import statements. While this limitation does not allow the `formula` definitions of previously defined grammars to be imported in deriving grammars, all other definitions without any name clashes are captured accordingly.

Operator Binding Strength and Precedence

An important behavior to provide full clarity on is in the binding strength and precedence of the connectives associated with the TL operators. Understanding the precedence and binding strength of each operator ensures that formulas are written as intended and do not result in potentially incorrect system evaluations because of unclear precedence rules.

The most notable behavior regarding operator binding is that TL-based operators have a stronger binding over non-temporal operators (i.e., below LTL). To demonstrate this binding behavior, see Example 4.1.1 below.

Example 4.1.1 (Binding Strength of Temporal Logic Operators)**.** Consider the simple LTL formula '`F p && q`' from the syntax in Appendix B that contains one temporal connective '`F`' and one propositional connective '`&&`' with two propositional variables '`p`' and '`q`'. While the intention of this formula may be to eventually capture the conjunction of both '`p`' and '`q`' as the subformula of '`F`' , after parsing the formula following the syntax and binding rules of TL operators, the structure of the resulting syntax tree is as follows below in Figure 4.5.

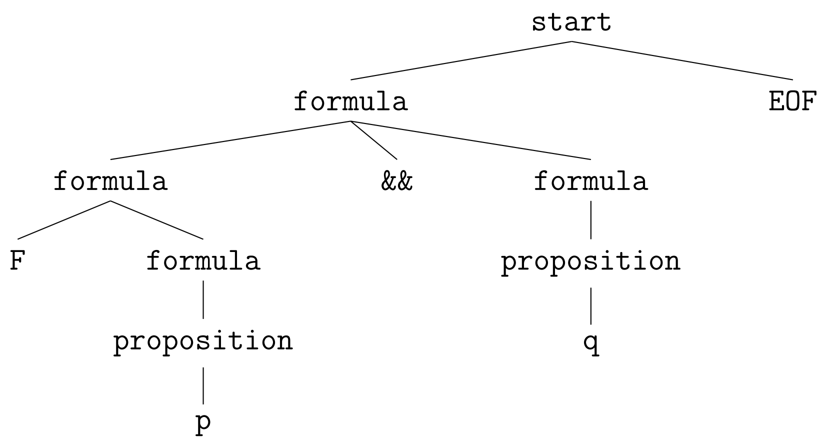


Figure 4.5: Syntax tree for '`F p && q`'.

Notably, since the temporal operator, eventually `F`, has a higher binding power than that of the logical connective, conjunction `&&`, the subformula `p` binds to the temporal operator instead of binding with the logical connective. If the intended behavior is for the temporal operator to bind with '`p && q`' as the subformula, then parentheses around the subformula must be explicitly provided. This newly parenthesized formula '`F (p && q)`' when parsed results in the syntax tree shown in Figure 4.6.

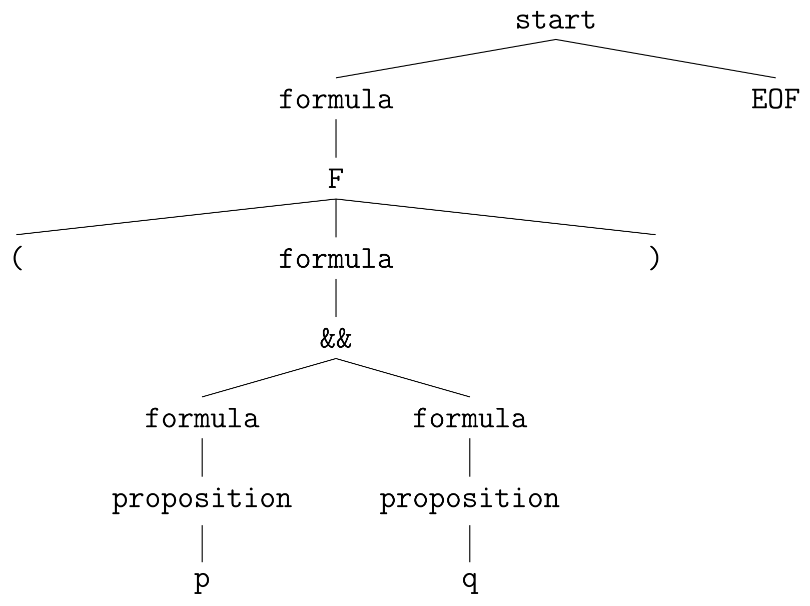


Figure 4.6: Syntax tree for '`F (p && q)`'.

This binding behavior is a common pattern seen when formulating TL requirements. Therefore, considerate placement of parentheses will ensure that formulas behave as expected.

While Example 4.1.1 provided an example with the eventually TL operator, there are many other TL-based operators that share this higher binding strength over non-TL operators and connectives. Furthermore, in the absence of parentheses, the precedence between operators of similar binding power need to be accounted for. This precedence ranking and associativity is represented below in Table 4.8.

Rank	**Operator**	**Associativity**
0	Spatial Exists/Forall	Left
1	Closure	Left
2	Interior	Left
3	Exists/Forall Qualifier	Left
4	Freeze Time Quantifier	Left
5	Eventually	Left
6	Always	Left
7	Next	Left
8	Until	Left
9	Release	Left
10	Logical Not	Right
11	Logical And	Left
12	Logical Or	Left
13	Implication	Left
14	Biconditional	Left

Table 4.8: Operator precedence (lower rank is a higher precedence).

4.2 Representing Temporal Logic Requirements

A significant contribution of this thesis is in the development of representing TL formulas as a modular, configurable, and extensible data structure. In doing so, the ability to evaluate and interpret a correctly parsed formula is simplified. Throughout this section, the transformation from a syntax tree to an interfaceable IR is detailed. This includes the structural components of the IR and the algorithm to construct it.

4.2.1 Intermediate Representation

Referring back to Figure 4.1, constructing a valid IR of a TL formula is performed in the third stage of the FoRek architecture (henceforth, "builder"). The builder accepts a syntax tree and produces a final custom IR to be utilized by external interfaces. Generally, the design of the data structure is a *rooted binary tree* which is a specialized form of a graph as reviewed in Section 2.4.1 and referred to as an AST. The makeup of the tree is composed of internal nodes that reflect operations and leaf nodes that reflect operands. Details regarding the types of nodes supported are provided in Section 4.2.2.

To build the IR, the syntax tree constructed by ANTLR is visited using a Depth-First Search (DFS) algorithm such that new nodes are produced when a meaningful context (i.e., node) of the syntax tree is encountered. For example, a meaningful context of the syntax tree may be a rule containing an '`always`' operator; whereas a non-meaningful context may be a parentheses rule. This decision to selectively construct semantically representative nodes reduces the size of the tree structure and ensures the representation minimally and accurately captures the original input without losing expressive details.

Within Figure 4.7, the syntax tree of the formula '`eventually[0,10] (p -> q)`' is illustrated where internal nodes correspond to rules from the ANTLR grammar and leaf nodes correspond to tokens captured by the lexer.

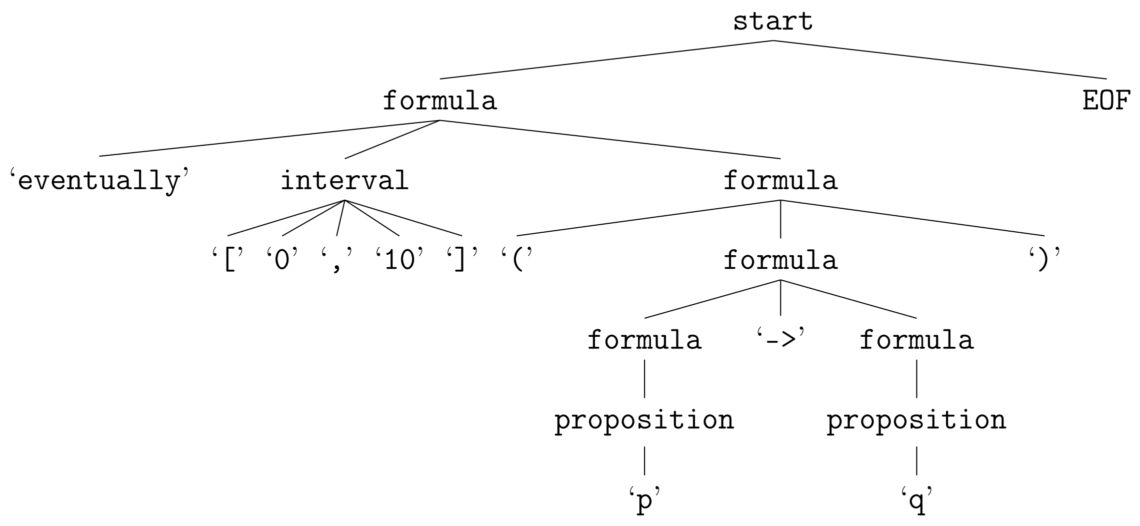


Figure 4.7: Syntax tree of STL formula '`eventually[0,10] (p -> q)`'.

From the syntax tree, the nodes go seven levels deep, and the tree contains additional parsing-related information such as the matching rule, the individual tokens, and the ordering of subformulas to be evaluated. While these aspects are important, not every piece of information populated onto the tree is necessary to capture the intention of the TL formula. As a result, the AST in Figure 4.8 is constructed to solve this issue.

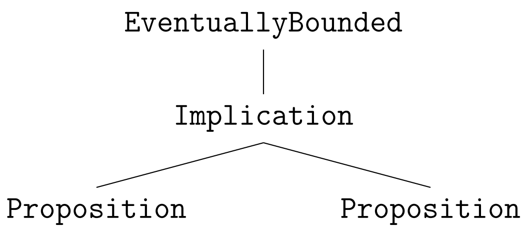


Figure 4.8: Resulting AST from syntax tree in Figure 4.7

The resulting AST in Figure 4.8 reduces the number of nodes from 21 in the syntax tree (Figure 4.7) to four with a max depth of three. With this size reduction, traversing and interpreting TL formulas is simpler and more aligned to the semantics captured in the formal definition of each found in Section 2.2.

4.2.2 Operands and Operations as Nodes

In designing the IR, the AST leverages a polymorphic structure where a base class forms the stem from which all other nodes derive from. In following Object-Oriented Programming (OOP) paradigm design aspects, a template for future operations that may be introduced is provided. From the current set of supported TLs, there are 54 concrete node types—of which 5 are operands. This includes the set of additional grammars needed to define arithmetic expressions.

To begin, the base structure which all TL operands and operations derive from is the `Node`. This the most general structure of a node and does not have any inherent attributes to utilize besides existing as the base class. Furthermore, two important set of classes are derived from which meaning to the tree begins. The first is the `Operand` class, and the second is the `Operation` class. The `Operand` class is equivalent to any tree that does not have children (i.e., *leaf nodes*), and the `Operation` class represents any tree structure that has children (i.e., *internal nodes*). Lastly, within the current FoRek framework, an operation can be categorized into two types: unary or binary. A unary operation has a single operand whereas a binary operation has a left and right operand, accordingly. These two patterns are commonly seen throughout the TL language definitions in Section 2.2, and thus permit a `Unary` and `Binary` operation. A simple Unifed Modeling Language (UML) diagram of these "blueprint" classes (interfaces) is illustrated in Figure 4.9. From these three most derived interfaces already introduced, all other operations and operands from the set of supported TLs are derived, accordingly.

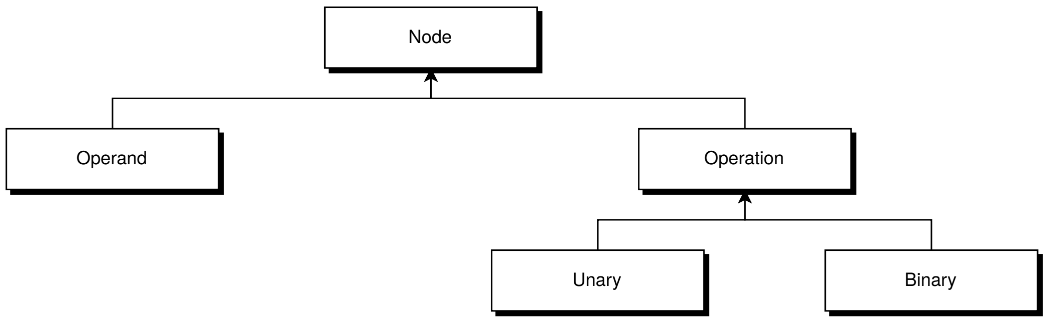


Figure 4.9: Hierarchy of FoRek interface nodes.

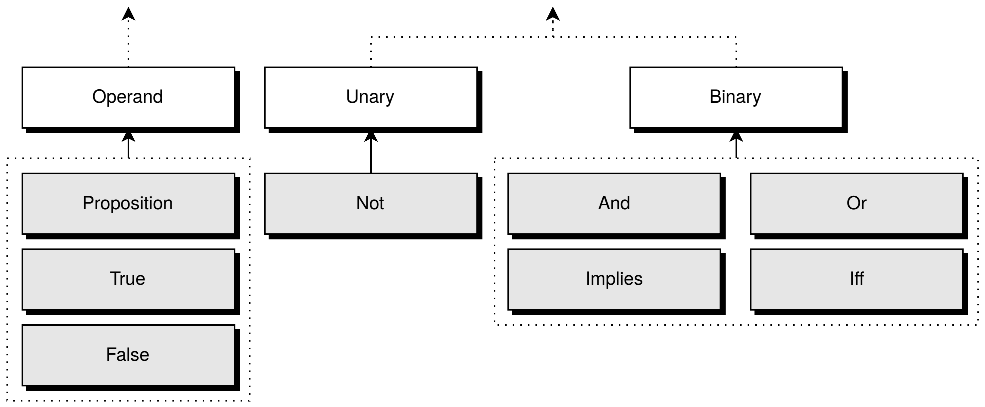


Figure 4.10: Hierarchy of FoRek PL operand and operation nodes.

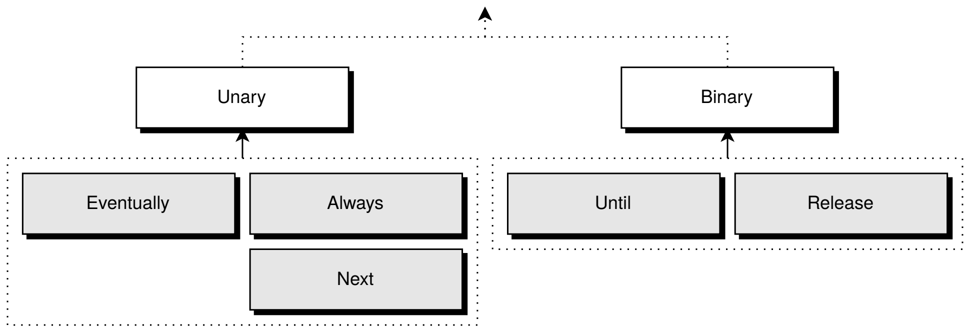


Figure 4.11: Hierarchy of FoRek LTL operation nodes.

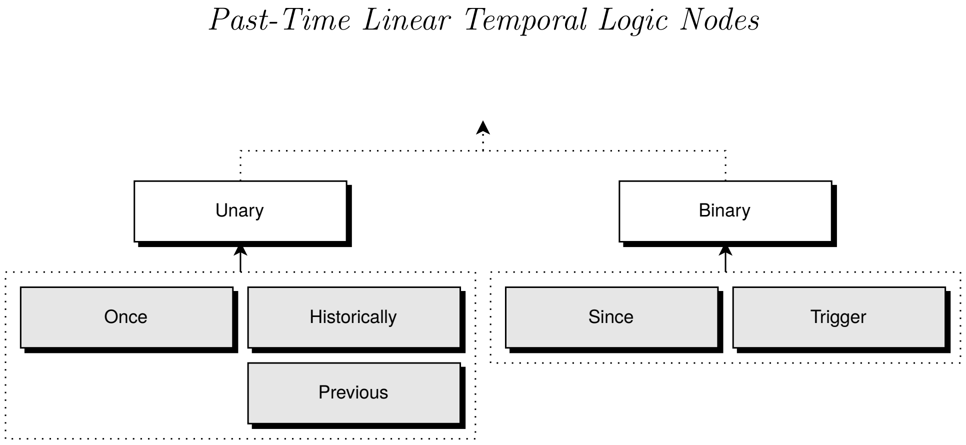


Figure 4.12: Hierarchy of FoRek pt-LTL operation nodes.

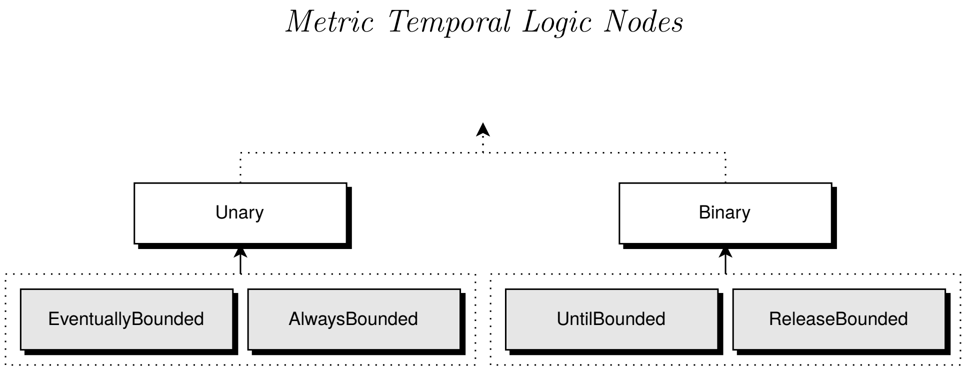


Figure 4.13: Hierarchy of FoRek MTL operation nodes.

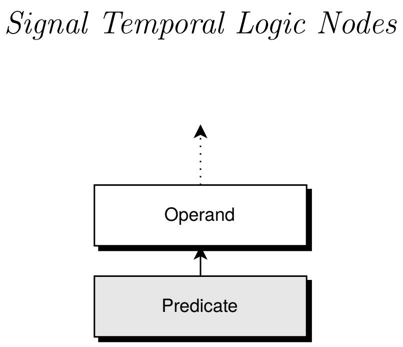


Figure 4.14: Hierarchy of FoRek STL operand node.

Timed Propositional Temporal Logic Nodes

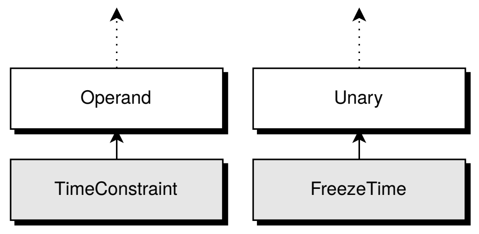


Figure 4.15: Hierarchy of FoRek TPTL operation nodes.

Timed Quality Temporal Logic Nodes

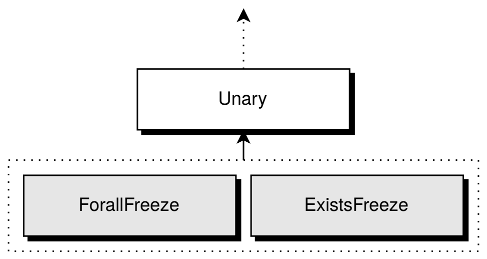


Figure 4.16: Hierarchy of FoRek TQTL operation nodes.

Spatio-Temporal Temporal Logic Nodes

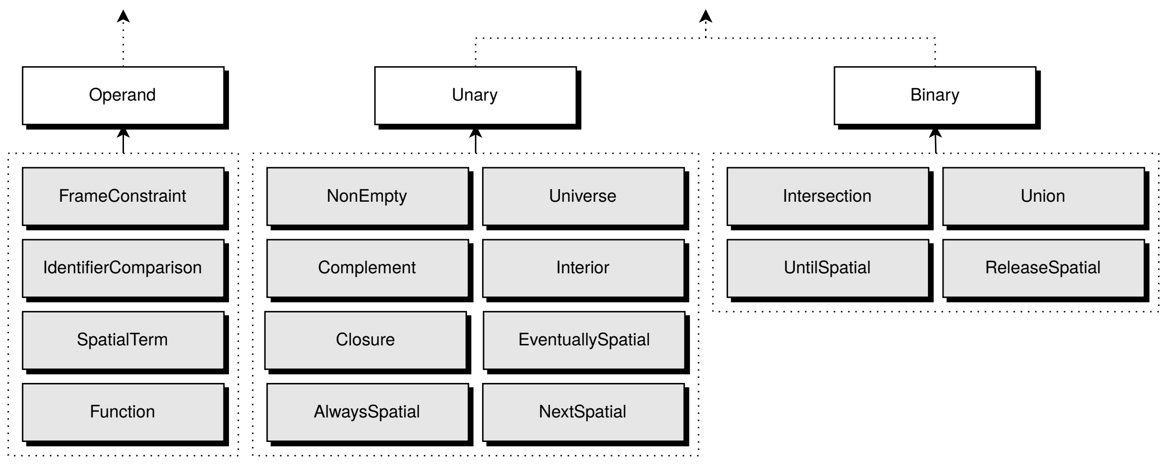


Figure 4.17: Hierarchy of FoRek STPL operation nodes.

4.3 Translating Between Temporal Logics

The third feature provided by FoRek in Figure 4.1 is the ability to translate between TLs. This support is motivated to improve interoperability between interdependent tools utilizing separate TL frameworks. Therefore, FoRek forms a connection between the interface and the underlying monitor without having to modify either directly. A visualization of this scenario is showcased below in Figure 4.18 comprising Psy-TaLiRo and several backends. Instead of modifying either of the tools directly, a unintrusive connection can be formed between the Psy-TaLiRo and the monitoring backends relieving the need for an explicit implementation-level dependency between either.

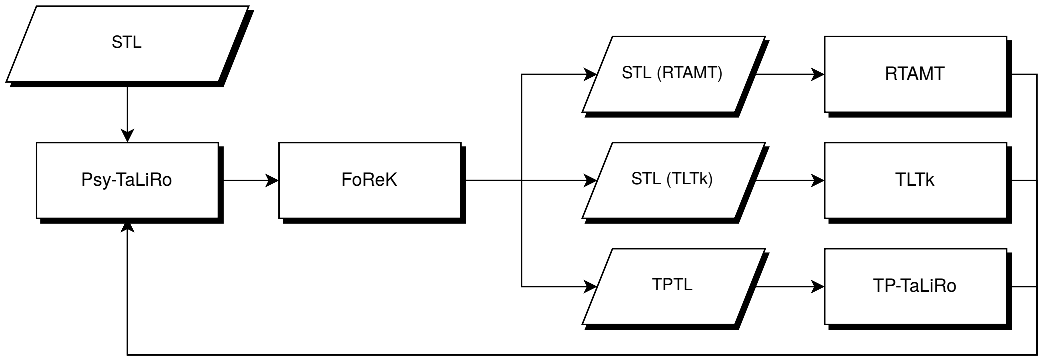


Figure 4.18: FoRek as a translation interface.

Translating between TLs is a step that requires careful consideration to ensure the resulting translation retains the meaning of the previous form as not all TLs can be translated into others. For example, translating between STPL and MTL cannot be fully supported as STPL introduces several semantically new operators that MTL has no knowledge of or other possible interpretation for. While a subset of STPL can be captured in MTL, translating between the two is not exhaustively supported, so a translation layer between these two is not supported. Furthermore, translations between two TLs along the same axis (i.e., lineage) are not translatable.

With these considerations in mind, the current framework of FoRek supports translating between MTL and TPTL formulas. This support derives from the similarity of both TLs to provide temporal-bounded operators with syntactic differences as well as no other new operations or rules that the other may not be able to interpret.

4.3.1 From MTL to TPTL

The translation between MTL and TPTL is supported due to the same functionality written differently. As both languages derive from LTL, a common syntactic source exists, so the translation between the same syntactic elements is trivial. However, in creating an effective translation, the aspects of the languages that are disjunct are highlighted. The translation scheme from MTL to TPTL using the semantic interpretation of TPTL formulas from [28, 17] is shown below in Definition 4.3.1.

Definition 4.3.1 (Metric Temporal Logic to Timed Propositional Temporal Logic). Let $\mathcal{I} = [a, b]$ such that a is the lower bound and b is the upper bound, inclusively. Given a valid MTL formula φ, an equivalently translated TPTL formula φ' is inductively defined below as follows:

$$\begin{aligned}
\bigcirc_{\mathcal{I}}\,\varphi \quad &\Rightarrow \quad x.\bigcirc\big((x \geq a \wedge x \leq b) \wedge \varphi\big)\\
\Diamond_{\mathcal{I}}\,\varphi \quad &\Rightarrow \quad x.\Diamond\big((x \geq a \wedge x \leq b) \wedge \varphi\big)\\
\Box_{\mathcal{I}}\,\varphi \quad &\Rightarrow \quad x.\Box\big((x \geq a \wedge x \leq b) \Rightarrow \varphi\big)\\
\varphi\,\mathcal{U}_{\mathcal{I}}\,\psi \quad &\Rightarrow \quad x.\Big(\varphi\,\mathcal{U}\,\big((x \geq a \wedge x \leq b) \wedge \psi\big)\Big)\\
\varphi\,\mathcal{R}_{\mathcal{I}}\,\psi \quad &\Rightarrow \quad x.\Big(\varphi\,\mathcal{R}\,\big((x \geq a \wedge x \leq b) \Rightarrow \psi\big)\Big)
\end{aligned}$$

where the PL-derived operators and atoms $\top$, α, $\neg$, $\wedge$, $\vee$, $\Rightarrow$, and $\Leftrightarrow$ translate to themselves, accordingly.

From Definition 4.3.1, any valid MTL formula can be synthesized into an equivalent TPTL formula without compromising expressivity allowing tools utilizing MTL to interface with tools utilizing TPTL.

4.4 Interpreting Temporal Logic Requirements

The fourth and final feature provided by the FoRek library is the ability to interpret IRs of the parsed TL formulas. This support is optional and not required to traverse the AST and thus is not included in Figure 4.1. However, this functionality is commonly needed as traversing the tree structure and evaluating subformulas against the trace is often times a common pattern. The general general infrastructure of the interpretation framework is shown below in Figure 4.19.

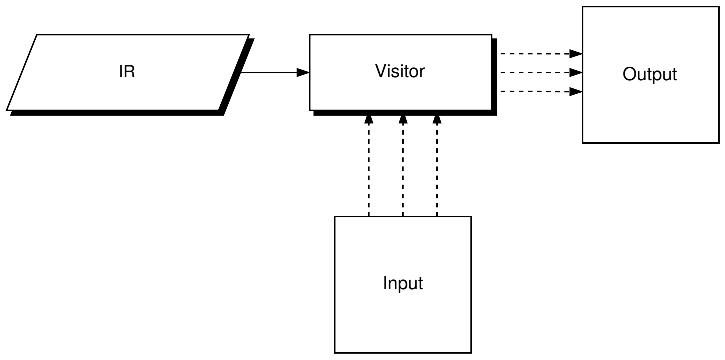


Figure 4.19: FoRek interpretation framework.

The design of this functionality is focused on extensibility and modification through polymorphic behavior. This decision is necessary as the framework, interpretation scheme, and application from which it may be used is unknown ahead a time and strict design decisions may yield a framework too rigid to be of any use. The interpretation of a formula is defined within a `Visitor` interface which is a derived polymorphic interface with base interfaces for each TL that allow an external interface to define the semantics for each unique node supported by the logic, accordingly. Denoted by dashed lines, since the interpretation of the formula cannot be too restrictive, the polymorphic design allows additional inputs, and outputs to be set before and after evaluation to enable a range of versatility and applications.

The design of the interpretation interface follows the *Visitor* design pattern [78, 41, 67] which allows the implementation of behavior to remain separated from the internal code allowing users to modify the semantics of the nodes without touching the internals of FoRek. For each TL, a custom visitor interface is designed which also follows a polymorphic structure to reduce the overhead of extending new TL support. The hierarchy of visitor interfaces is shown below.

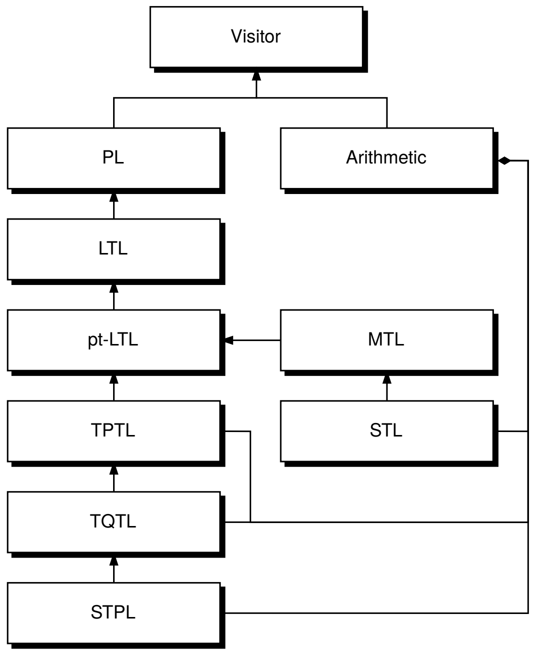


Figure 4.20: Hierarchy of FoRek visitors.

The visitor takes advantage of multiple dispatch that allows the node to control which visitor it should be calling at runtime, and from that call, which visitor method should be called accordingly. As a result, the external interface only needs to define what gets stored into the node as a result of this call.

Chapter 5

PYTHONIC FORMAL REQUIREMENTS LANGUAGE

This chapter presents a brief overview of the Pythonic Formal Requirements Language (PyForeL), and then provides a summary of modifications, features, and improvements over its previous version. Furthermore, a formal definition by induction on the translational units of a PyForeL program is performed to showcase the correctness of the translation procedure.

PyForeL is a Domain-Specific Language (DSL) designed to ease the elicitation of TL requirements. The conception of this work comes from [6, 9] which provided a preliminary method of translating high-level DSL constructs into TL specifications.

5.1 Updates, Features, and Improvements

From its previous iteration, the PyForeL tool has been updated in several areas regarding grammar restructuring, translation procedure, and some changes to the syntactic constructs. In this thesis, the major changes will be reflected here and any minor changes should be consulted with the code found in [8].

5.1.1 Grammar Restructuring

The first major change for the PyForeL tool is in the grammar defined by ANTLR that captures the DSL. The grammar has been reorganized to improve readability, remove rules that do not properly map to TL formulas, and remove all embedded actions. Furthermore, a slight limitation of the syntax previously introduced in PyForeL was the heavy reliance on indentation and dedentation to differentiate scope and subformulas. Consequently, writing inline statements were difficult or required a function call to effectively perform an in-place insertion at translation time. This limitation has been resolved by allowing any valid PyFoReL statements to appear in enclosed braces.

5.1.2 Translation Procedure

In the previous version of PyFoReL, the translation procedure relied on an embedded syntax-directed scheme where upon visiting a rule, the resulting structure would be recursively generated "on-the-fly" [39]. While this approach is functional, it had several drawbacks: (1) the ANTLR grammar was verbose and not easily understood, (2) extending the grammar relied on adding new functions as well, (3) porting the grammar to other target languages was non-obvious and required additional work to support the embedded function calls, and (4) an additional actions grammar had to be supplied to provide additional parsing and lexing logic that is not obvious to the behavior of the DSL.

To resolve this problem, the new translation procedure instead walks the generated parse tree by ANTLR and builds the TL specification recursively. While this still uses the syntax-directed approach, it simplifies the grammar as well as the implementation.

5.1.3 Integration with FoRek

Furthermore, since FoRek is responsible for parsing and interpreting TL formulas, PyFoReL is able to focus and serve just the translation procedure of a PyFoReL program into a TL formula. If further interaction with a PyFoReL program is needed, FoRek may then effectively generate the appropriate IR from the translated PyFoReL program. An overview of this integration is shown below.

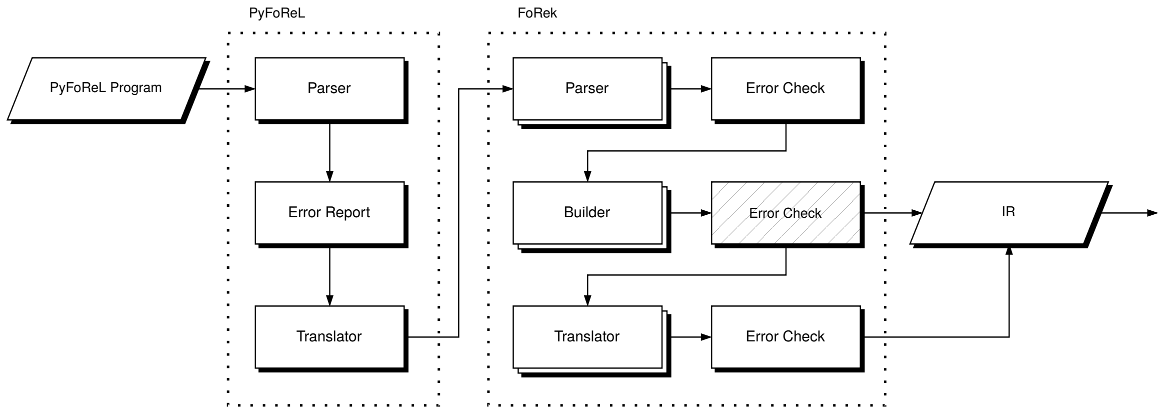


Figure 5.1: Integration of PyFoReL and FoRek framework.

5.2 Structure of PyFoReL Program

To showcase the translation capabilities of the PyFoReL tool, an inductive definition on the core translational units of a PyFoReL program is provided. Notably, this definition does not, however, aim to establish that all TL formulas can be translated into a PyFoReL program but rather that the currently supported set of statements map to a TL formula.

5.2.1 Statements

Before providing the definition, a brief review of the various statements supported within PyFoReL must be performed.

In the current version of PyFoReL (v0.2.0), there are two categories of statements: simple or compound. Simple statements are characterized as statements that fit on a single line and commonly reflect atomic propositions in TL. Compound statements are characterized statements spread across multiple lines. A compound statement commonly reflects TL expressions with subformulas. Within PyFoReL, there are currently four simple statement types and six compound statement types. An overview of each statement and its corresponding categorization is shown below in

Table 5.1.

Statement	**Simple**	**Compound**
Conditional		✓
Definition		✓
False	✓	
Freeze		✓
Function Call	✓	
Proposition	✓	
Qualifier		✓
Temporal		✓
True	✓	
Verbatim		✓

Table 5.1: PyFoReL statements.

In translating a PyFoReL program, the file is read in a top-down fashion populating function definition tables and begins translation upon hitting a proper translational unit. A translational unit is considered a statement that is eventually mapped to a TL formula. For example, the *Definition* statement does not become directly mapped to a TL formula as function definitions are not supported in TL. However, the contents of the statement is a sequence of statements that are considered translational units. Therefore, in providing a clear definition, the translational units are considered and not statements that wrap or support translation indirectly.

5.2.2 Definition

Within this definition, the following statements are not considered in the definition of a PyFoReL program: (1) *Definition*, (2) *Verbatim*.

Furthermore, let $\mathcal{L}(I)$ be the language that recognizes all valid identifiers that begin with any alpha character or underscore followed by zero or more alphanumeric characters or underscores; $\mathcal{L}(K) = \{$ `if`, `else`, `elif`, ... $\}$ be the set of reserved keywords within a PyFoReL program, and $\hookrightarrow$ represent a single indentation.

Definition 5.2.1 (PyFoReL Program)**.** Let $\mathcal{L}(P)$ be the language recognized by a PyFoReL program, and $\mathcal{L}_I(P) = \mathcal{L}(I) \setminus \mathcal{L}(K)$ be the language that recognizes all unreserved identifiers. Given a PyFoReL program p, a unique TL translation φ is inductively defined as follows:

Proposition. The translation of a *Proposition* statement is as follows:

$$v \quad \rightarrow \quad \alpha$$

where $v \in \mathcal{L}_I(P)$ and $\alpha \in \text{AP}$.

True. The translation of a *True* statement is as follows:

$$\texttt{true} \quad \rightarrow \quad \top$$

False. The translation of a *False* statement is as follows:

$$\texttt{false} \quad \rightarrow \quad \bot$$

Conditional I (If). Given the valid PyFoReL programs $A, B \in \mathcal{L}(P)$, the translation of a *Conditional* statement is as follows:

$$\begin{array}{l} \texttt{if} \quad A : \\ \hookrightarrow B \end{array} \quad \rightarrow \quad A \Rightarrow B$$

Conditional II (If-Elif). Given the valid PyFoReL programs $A, B, C, D \in \mathcal{L}(P)$, the translation of a *Conditional* statement is as follows:

$$\begin{array}{l} \texttt{if} \quad A : \\ \hookrightarrow B \\ \texttt{elif} \quad C : \\ \hookrightarrow D \end{array} \quad \rightarrow \quad (A \Rightarrow B) \wedge (\neg A \wedge C \Rightarrow D)$$

Conditional III (If-Else). Given the valid PyForeL programs $A, B, C, D \in \mathcal{L}(P)$, the translation of a *Conditional* statement is as follows:

$$\begin{array}{l} \texttt{if} \quad A \texttt{ :} \\ \hookrightarrow B \\ \texttt{else} \quad C \texttt{ :} \\ \hookrightarrow D \end{array} \quad \rightarrow \quad (A \Rightarrow B) \wedge (\neg A \Rightarrow D)$$

Freeze. Given a valid PyForeL program $A \in \mathcal{L}(P)$, the translation of a *Freeze* statement is as follows:

$$\begin{array}{l} \texttt{at} \quad x \texttt{ :} \\ \hookrightarrow A \end{array} \quad \rightarrow \quad x.A$$

where $x \in V_t$.

Function Call. Given a valid PyForeL program $A \in \mathcal{L}(P)$, the translation of a *Function Call* statement is as follows:

$$f\texttt{()} \quad \rightarrow \quad \mathfrak{F}(f)$$

where $\mathfrak{F} : f \rightarrow A$ such that $f \in \mathcal{L}_I(P)$.

Qualifier. Given a valid PyForeL program $A \in \mathcal{L}(P)$, the translation of a *Qualifier* statement is as follows:

$$\begin{array}{l} \texttt{exists} \quad v_1\texttt{,}\, v_2\texttt{,}\, \ldots\texttt{,}\, v_n \quad \texttt{at} \quad x \texttt{ :} \\ \hookrightarrow A \end{array} \quad \rightarrow \quad \exists v_1 @ x, \exists v_2 @ x, \ldots, \exists v_n @ x, A$$

where $v_i \in V_{id}$ and $x \in V_t$.

Temporal. Given a valid PyForeL program $A \in \mathcal{L}(P)$, the translation of a *Temporal* statement is as follows:

$$\begin{array}{l} \texttt{eventually:} \\ \hookrightarrow A \end{array} \quad \rightarrow \quad \Diamond A$$

Chapter 6

EXPERIMENTS

In this chapter, two different experiments are performed. The first experiment is a demonstration by application of the FoRek library integrated with the offline monitor TP-TaLiRo [30] to replace its current parsing framework with a TPTL parsing interface provided by FoRek. The second experiment showcases the performance metrics of parsing TL formulas with FoRek on a variety with varying levels of complexity and length evaluate its speed and memory consumption.

6.1 Integration with Offline Monitor

Within this demonstration, the FoRek library is integrated into the Timed Propositional Temporal Logic Robustness (TP-TaLiRo) [30] monitor to replace the native TPTL parser with one supplied by FoRek. In integrating FoRek with the monitor, FoRek provides the TL interface to parse TPTL formulas, and TP-TaLiRo supplies the data structures to perform monitoring of these formulas. This approach allows the updated and robust parsers of FoRek to be utilized without compromising the core infrastructure developed by the TP-TaLiRo tool. An overview of this integration is illustrated below in Figure 6.1.

The major change in Figure 6.1 is in the parser block (greyed-out) for TP-TaLiRo being bypassed and replaced with the TPTL parser by FoRek. The FoRek library is responsible for parsing the input TPTL formula and building the appropriate TP-TaLiRo-specific data structure for the monitor to evaluate the model under test.

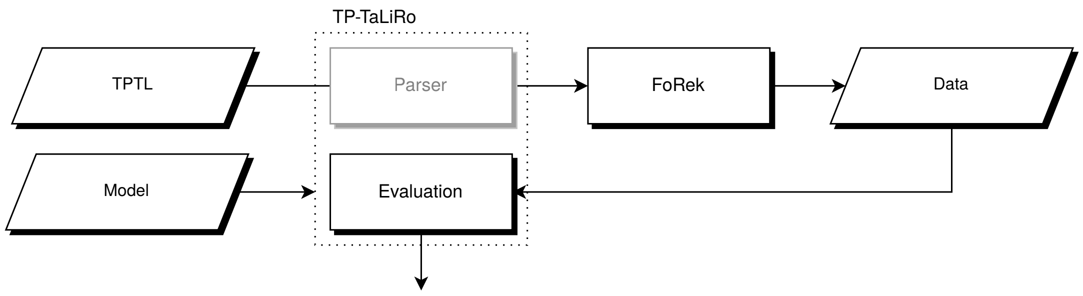


Figure 6.1: Integration approach of TP-TaLiRo with FoRek.

6.2 Performance Benchmarks

Several performance benchmarks were performed to showcase the speed and memory consumption for each of the currently supported parsers within the FoRek library. This included testing the following parsers: (1) PL, (2) LTL, (3) MTL, (4) STL, (5) TPTL, (6) TQTL, and (7) STPL. For each parser, a total of six requirements were written targeting the operations unique to that language. The set of requirements ranged in length (measured by the number of nodes from the resulting AST) from 1 to 3125. Furthermore, for each requirement of each parser, 100 replications were performed and the average was taken. The performance benchmarks of the PL parser are shown below in Figure 6.2.

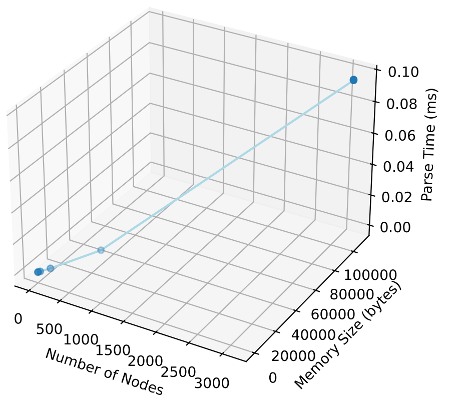
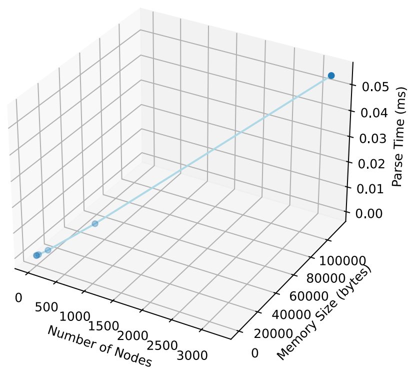


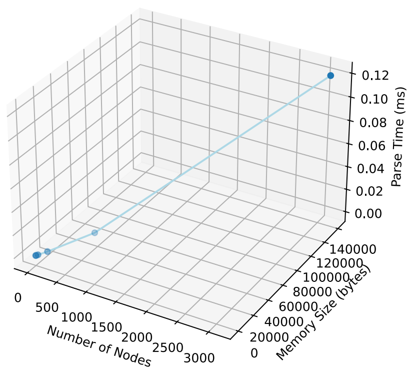
Figure 6.2: PL parser performance.

From observing 6.2, the x-axis ranges in the length of the formula, the y-axis ranges in the memory consumption (i.e., memory size of the resulting AST), and the z-axis showcases the parse-time required to parse and build the resulting IR from the associated formula. The performance benchmarks following the same format for the other six TL-based parsers are shown in Figure 6.3.

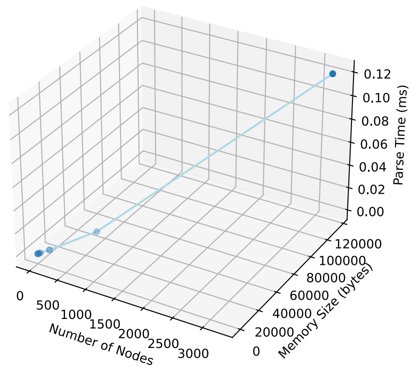
The results of these benchmarks showcase the performance of the parsers to run in average linear time on the length and size of the formula. This result is unsurprising as the time complexity of ANTLR's *ALL(*)* [70] is $O(n^4)$ in theory. However, in practice, it is linear. Furthermore, the time it takes to build the IR of the formula is $O(V + E)$ where V represents the number of nodes and E represents the number of edges. Consequently, as the parsers developed for FoRek are generated from ANTLR using the same parsing method, the results prove true to this statement.



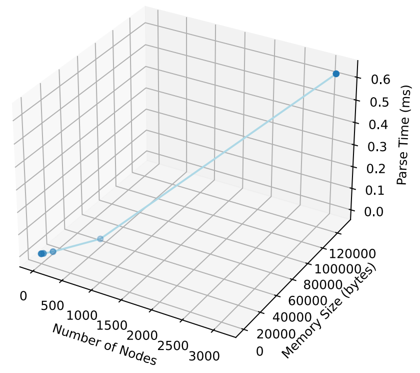
(a) LTL parser performance.



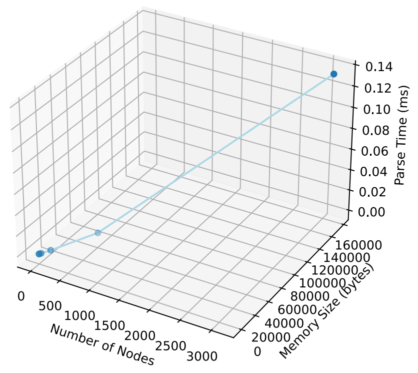
(b) MTL parser performance.



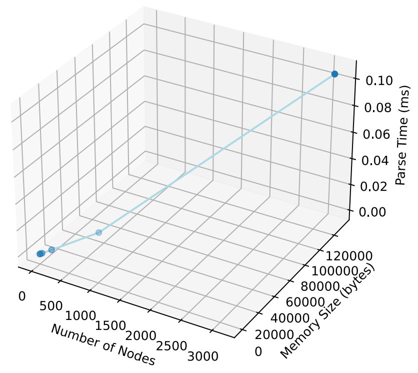
(c) STL parser performance.



(d) TPTL parser performance.



(e) TQTL parser performance.



(f) STPL parser performance.

Figure 6.3: Benchmarks of parse time vs. formula length and size.

Chapter 7

CONCLUSIONS AND FUTURE WORK

In this work, the proposal for a standardization of TL specifications used within testing and monitoring applications was made through the development of a FoRek: a modular, extensible, and modern C++ library. The use of this library was demonstrated through an integration of the library with a currently available offline monitor for evaluating TPTL requirements as well as various performance benchmarks of the collection of parsers. Furthermore, improvements and updates to the PyFoReL tool were made along with a formal definition to showcase the translation mapping of a PyFoReL program to a TL formula.

Future work of the FoRek library entails several possibilities. First, the library does not provide any formula compacting to reduce redundant or otherwise equivalent formulas. Secondly, the tool may be extended to support additional branches of TL such as Computational Tree Logic (CTL) and its variants. Lastly, to support online monitoring applications, a translation procedure from future-bounded TL formulas to past-time equivalents may be supported.

REFERENCES

[1] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. USA: Addison-Wesley Longman Publishing Co., Inc., 2006. ISBN: 0321486811.

[2] Takumi Akazaki, Shuang Liu, Yoriyuki Yamagata, Yihai Duan, and Jianye Hao. "Falsification of cyber-physical systems using deep reinforcement learning". In: *Formal Methods: 22nd International Symposium, FM 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 15-17, 2018, Proceedings 22*. Springer. 2018, pp. 456–465.

[3] Rajeev Alur, Tomás Feder, and Thomas A Henzinger. "The benefits of relaxing punctuality". In: *Journal of the ACM (JACM)* 43.1 (1996), pp. 116–146.

[4] Rajeev Alur and Thomas A Henzinger. "A really temporal logic". In: *Journal of the ACM (JACM)* 41.1 (1994), pp. 181–203.

[5] Rajeev Alur and Mihalis Yannakakis. "Model checking of message sequence charts". In: *CONCUR'99 Concurrency Theory: 10th International Conference Eindhoven, The Netherlands, August 24—27, 1999 Proceedings*. Springer. 2002, pp. 114–129.

[6] Jacob Anderson. "DSL for Spatio-Temporal Perception Logic Specifications". In: (2021).

[7] Jacob Anderson. *Formal Requirements Toolkit - GitLab*. 2023. URL: `https://gitlab.com/sbtg/forek`.

[8] Jacob Anderson. *Pythonic Formal Requirements Language - GitLab*. 2023. URL: `https://gitlab.com/sbtg/pyforel`.

[9] Jacob Anderson, Mohammad Hekmatnejad, and Georgios Fainekos. "PyFoReL: A Domain-Specific Language for Formal Requirements in Temporal Logic". In: *2022 IEEE 30th International Requirements Engineering Conference (RE)*. IEEE. 2022, pp. 266–267.

[10] Yashwanth Annpureddy, Che Liu, Georgios Fainekos, and Sriram Sankaranarayanan. "S-taliro: A tool for temporal logic falsification for hybrid systems". In: *Tools and Algorithms for the Construction and Analysis of Systems: 17th International Conference, TACAS 2011, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2011, Saarbrücken, Germany, March 26–April 3, 2011. Proceedings 17*. Springer. 2011, pp. 254–257.

[11] Marco Autili, Lars Grunske, Markus Lumpe, Patrizio Pelliccione, and Antony Tang. "Aligning qualitative, real-time, and probabilistic property specification

patterns using a structured english grammar". In: *IEEE Transactions on Software Engineering* 41.7 (2015), pp. 620–638.

[12] Anand Balakrishnan, Jyotirmoy Deshmukh, Bardh Hoxha, Tomoya Yamaguchi, and Georgios Fainekos. "PerceMon: online monitoring for perception systems". In: *Runtime Verification: 21st International Conference, RV 2021, Virtual Event, October 11–14, 2021, Proceedings 21*. Springer. 2021, pp. 297–308.

[13] Anand Balakrishnan, Aniruddh G Puranic, Xin Qin, Adel Dokhanchi, Jyotirmoy V Deshmukh, Heni Ben Amor, and Georgios Fainekos. "Specifying and evaluating quality metrics for vision-based perception systems". In: *2019 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE. 2019, pp. 1433–1438.

[14] 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: Introductory and Advanced Topics* (2018), pp. 135–175.

[15] Ilan Beer, Shoham Ben-David, Cindy Eisner, Dana Fisman, Anna Gringauze, and Yoav Rodeh. "The temporal logic Sugar". In: *Computer Aided Verification: 13th International Conference, CAV 2001 Paris, France, July 18–22, 2001 Proceedings 13*. Springer. 2001, pp. 363–367.

[16] Béatrice Bérard, Michel Bidoit, Alain Finkel, François Laroussinie, Antoine Petit, Laure Petrucci, and Philippe Schnoebelen. *Systems and software verification: model-checking techniques and tools*. Springer Science & Business Media, 2013.

[17] Patricia Bouyer, Fabrice Chevalier, and Nicolas Markey. "On the expressiveness of TPTL and MTL". In: *FSTTCS 2005: Foundations of Software Technology and Theoretical Computer Science: 25th International Conference, Hyderabad, India, December 15-18, 2005. Proceedings 25*. Springer. 2005, pp. 432–443.

[18] Hans Kleine Büning and Theodor Lettmann. *Propositional logic: deduction and algorithms*. Vol. 48. Cambridge University Press, 1999.

[19] Igor Buzhinsky. "Formalization of natural language requirements into temporal logics: a survey". In: *2019 IEEE 17th international conference on industrial informatics (INDIN)*. Vol. 1. IEEE. 2019, pp. 400–406.

[20] Hong Chen. "Applications of cyber-physical system: a literature review". In: *Journal of Industrial Integration and Management* 2.03 (2017), p. 1750012.

[21] Noam Chomsky. "On certain formal properties of grammars". In: *Information and control* 2.2 (1959), pp. 137–167.

[22] Alessandro Cimatti, Marco Roveri, and Daniel Sheridan. “Bounded verification of past LTL”. In: *Formal Methods in Computer-Aided Design: 5th International Conference, FMCAD 2004, Austin, Texas, USA, November 15-17, 2004. Proceedings 5*. Springer. 2004, pp. 245–259.

[23] Edmund M Clarke. “Model checking”. In: *Foundations of Software Technology and Theoretical Computer Science: 17th Conference Kharagpur, India, December 18–20, 1997 Proceedings 17*. Springer. 1997, pp. 54–56.

[24] Matthias Cosler, Christopher Hahn, Daniel Mendoza, Frederik Schmitt, and Caroline Trippel. “nl2spec: Interactively Translating Unstructured Natural Language to Temporal Logics with Large Language Models”. In: *arXiv preprint arXiv:2303.04864* (2023).

[25] Joseph Cralley, Ourania Spantidi, Bardh Hoxha, and Georgios Fainekos. “Tltk: A toolbox for parallel robustness computation of temporal logic specifications”. In: *Runtime Verification: 20th International Conference, RV 2020, Los Angeles, CA, USA, October 6–9, 2020, Proceedings 20*. Springer. 2020, pp. 404–416.

[26] Werner Damm and David Harel. “LSC’s: Breathing life into message sequence charts”. In: *Formal Methods for Open Object-Based Distributed Systems: IFIP TC6/WG6. 1 Third International Conference on Formal Methods for Open Object-Based Distributed Systems (FMOODS), February 15–18, 1999, Florence, Italy*. Springer. 1999, pp. 293–311.

[27] Nelly Delgado, Ann Q Gates, and Steve Roach. “A taxonomy and catalog of runtime software-fault monitoring tools”. In: *IEEE Transactions on software Engineering* 30.12 (2004), pp. 859–872.

[28] Adel Dokhanchi. “From formal requirement analysis to testing and monitoring of cyber-physical systems”. PhD thesis. Arizona State University, 2017.

[29] Adel Dokhanchi, Heni Ben Amor, Jyotirmoy V Deshmukh, and Georgios Fainekos. “Evaluating perception systems for autonomous vehicles using quality temporal logic”. In: *Runtime Verification: 18th International Conference, RV 2018, Limassol, Cyprus, November 10–13, 2018, Proceedings 18*. Springer. 2018, pp. 409–416.

[30] Adel Dokhanchi, Bardh Hoxha, Cumhur Erkan Tuncali, and Georgios Fainekos. “An efficient algorithm for monitoring practical TPTL specifications”. In: *2016 ACM/IEEE International Conference on Formal Methods and Models for System Design (MEMOCODE)*. IEEE. 2016, pp. 184–193.

[31] Alexandre Donzé. “Breach, a toolbox for verification and parameter synthesis of hybrid systems.” In: *CAV*. Vol. 10. Springer. 2010, pp. 167–170.

[32] Matthew B Dwyer, George S Avrunin, and James C Corbett. “Property specification patterns for finite-state verification”. In: *Proceedings of the second workshop on Formal methods in software practice*. 1998, pp. 7–15.

[33] E Allen Emerson. “Temporal and modal logic”. In: *Formal Models and Semantics*. Elsevier, 1990, pp. 995–1072.

[34] Gidon Ernst, Paolo Arcaini, Georgios Fainekos, Federico Formica, Jun Inoue, Tanmay Khandait, Mohammad Mahdi Mahboob, Claudio Menghi, Giulia Pedrielli, Masaki Waga, et al. “ARCH-COMP 2022 Category Report: Falsification with Ubounded Resources”. In: *Proceedings of 9th International Workshop on Applied*. Vol. 90. 2022, pp. 204–221.

[35] Gidon Ernst, Sean Sedwards, Zhenya Zhang, and Ichiro Hasuo. “Fast falsification of hybrid systems using probabilistically adaptive input”. In: *Quantitative Evaluation of Systems: 16th International Conference, QEST 2019, Glasgow, UK, September 10–12, 2019, Proceedings 16*. Springer. 2019, pp. 165–181.

[36] Georgios E Fainekos and George J Pappas. “Robustness of temporal logic specifications”. In: *Formal Approaches to Software Testing and Runtime Verification: First Combined International Workshops, FATES 2006 and RV 2006, Seattle, WA, USA, August 15-16, 2006, Revised Selected Papers*. Springer. 2006, pp. 178–192.

[37] Cameron Finucane, Gangyuan Jing, and Hadas Kress-Gazit. “LTLMoP: Experimenting with language, temporal logic and robot control”. In: *2010 IEEE/RSJ International Conference on Intelligent Robots and Systems*. IEEE. 2010, pp. 1988–1993.

[38] Michael Fisher. *An introduction to practical formal methods using temporal logic*. John Wiley & Sons, 2011.

[39] Martin Fowler. *Domain-specific languages*. Pearson Education, 2010.

[40] Francesco Fuggitti and Tathagata Chakraborti. “NL2LTL–A Python Package for Converting Natural Language (NL) Instructions to Linear Temporal Logic (LTL) Formulas”. In.

[41] Erich Gamma, Ralph Johnson, Richard Helm, Ralph E Johnson, and John Vlissides. *Design patterns: elements of reusable object-oriented software*. Pearson Deutschland GmbH, 1995.

[42] Rowan Garnier and John Taylor. *Discrete mathematics: proofs, structures and applications*. CRC press, 2009.

[43] Michael JC Gordon. "Validating the PSL/Sugar semantics using automated reasoning". In: *Formal Aspects of Computing* 15.4 (2003), pp. 406–421.

[44] Mike Gordon. "PSL semantics in higher order logic". In: *Workshop on Designing Correct Circuits (DCC)*. 2004.

[45] Mike Gordon, Joe Hurd, and Konrad Slind. "Executing the formal semantics of the Accellera property specification language by mechanised theorem proving". In: *CHARME*. Springer. 2003, pp. 200–215.

[46] David Harel. "Statecharts: A visual formalism for complex systems". In: *Science of computer programming* 8.3 (1987), pp. 231–274.

[47] David Harel and Amnon Naamad. "The STATEMATE semantics of statecharts". In: *ACM Transactions on Software Engineering and Methodology (TOSEM)* 5.4 (1996), pp. 293–333.

[48] Jie He, Ezio Bartocci, Dejan Ničković, Haris Isakovic, and Radu Grosu. "DeepSTL: from english requirements to signal temporal logic". In: *Proceedings of the 44th International Conference on Software Engineering*. 2022, pp. 610–622.

[49] Mohammad Hekmatnejad, Bardh Hoxha, Jyotirmoy V Deshmukh, Yezhou Yang, and Georgios Fainekos. "Formalizing and Evaluating Requirements of Perception Systems for Automated Vehicles using Spatio-Temporal Perception Logic". In: *arXiv preprint arXiv:2206.14372* (2022).

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

[51] "IEEE Standard for Property Specification Language (PSL)". In: *IEEE Std 1850-2010 (Revision of IEEE Std 1850-2005)* (2010), pp. 1–182.

[52] Johan Anthony Wilem Kamp. *Tense logic and the theory of linear order*. University of California, Los Angeles, 1968.

[53] James Kapinski, Jyotirmoy V Deshmukh, Xiaoqing Jin, Hisahiro Ito, and Ken Butts. "Simulation-based approaches for verification of embedded control systems: An overview of traditional and advanced modeling, testing, and verification techniques". In: *IEEE Control Systems Magazine* 36.6 (2016), pp. 45–64.

[54] John C Knight. "Safety critical systems: challenges and directions". In: *Proceedings of the 24th international conference on software engineering*. 2002, pp. 547–550.

[55] Ron Koymans. “Specifying real-time properties with metric temporal logic”. In: *Real-time systems* 2.4 (1990), pp. 255–299.

[56] Hadas Kress-Gazit, Georgios E Fainekos, and George J Pappas. “Translating structured english to robot controllers”. In: *Advanced Robotics* 22.12 (2008), pp. 1343–1359.

[57] Saul Kripke. “Semantical considerations of the modal logic”. In: *Studia Philosophica* 1 (2007).

[58] Insup Lee and Oleg Sokolsky. “A graphical property specification language”. In: *Proceedings 1997 High-Assurance Engineering Workshop*. IEEE. 1997, pp. 42–47.

[59] Oscar Levin. “Discrete mathematics: An open introduction”. In: (2021).

[60] Charles E Mackenzie. *Coded-Character Sets: History and Development*. Addison-Wesley Longman Publishing Co., Inc., 1980.

[61] Oded Maler and Dejan Nickovic. “Monitoring temporal properties of continuous signals”. In: *Formal Techniques, Modelling and Analysis of Timed and Fault-Tolerant Systems: Joint International Conferences on Formal Modeling and Analysis of Timed Systmes, FORMATS 2004, and Formal Techniques in Real-Time and Fault-Tolerant Systems, FTRTFT 2004, Grenoble, France, September 22-24, 2004. Proceedings*. Springer. 2004, pp. 152–166.

[62] Zohar Manna and Amir Pnueli. *The temporal logic of reactive and concurrent systems: Specification*. Springer Science & Business Media, 2012.

[63] Claudio Menghi, Shiva Nejati, Lionel Briand, and Yago Isasi Parache. “Approximation-refinement testing of compute-intensive cyber-physical models: An approach based on system identification”. In: *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*. 2020, pp. 372–384.

[64] Sayan Mitra. *Verifying Cyber-Physical Systems: A Path to Safe Autonomy*. MIT Press, Feb. 16, 2021. 312 pp. ISBN: 978-0-262-04480-6.

[65] Dejan Nickovic and Oded Maler. “AMT: A property-based monitoring tool for analog systems”. In: *Formal Modeling and Analysis of Timed Systems: 5th International Conference, FORMATS 2007, Salzburg, Austria, October 3-5, 2007. Proceedings 5*. Springer. 2007, pp. 304–319.

[66] Dejan Ničković and Tomoya Yamaguchi. “RTAMT: Online robustness monitors from STL”. In: *Automated Technology for Verification and Analysis: 18th International Symposium, ATVA 2020, Hanoi, Vietnam, October 19–23, 2020, Proceedings*. Springer. 2020, pp. 564–571.

[67] Jens Palsberg and C Barry Jay. "The essence of the visitor pattern". In: *Proceedings. The Twenty-Second Annual International Computer Software and Applications Conference (Compsac'98)(Cat. No. 98CB 36241)*. IEEE. 1998, pp. 9–15.

[68] Terence Parr. "Language implementation patterns: create your own domain-specific and general programming languages". In: *Language Implementation Patterns* (2009), pp. 1–380.

[69] Terence Parr. *The Definitive ANTLR 4 Reference*. 2nd. Pragmatic Bookshelf, 2013. ISBN: 1934356999.

[70] Terence Parr, Sam Harwell, and Kathleen Fisher. "Adaptive LL (*) parsing: the power of dynamic analysis". In: *ACM SIGPLAN Notices* 49.10 (2014), pp. 579–598.

[71] Terence J. Parr and Russell W. Quong. "ANTLR: A predicated-LL (k) parser generator". In: *Software: Practice and Experience* 25.7 (1995), pp. 789–810.

[72] Amir Pnueli. "The temporal logic of programs". In: *18th Annual Symposium on Foundations of Computer Science (sfcs 1977)*. ieee. 1977, pp. 46–57.

[73] Arthur Prior. "Past, present and future". In: (1967).

[74] Arthur N Prior. *Time and modality.* John Locke Lecture, 2003.

[75] Grigore Roşu and Klaus Havelund. "Rewriting-based techniques for runtime verification". In: *Automated Software Engineering* 12.2 (2005), pp. 151–197.

[76] Ekkart Rudolph, Peter Graubmann, and Jens Grabowski. "Tutorial on message sequence charts". In: *Computer networks and ISDN systems* 28.12 (1996), pp. 1629–1641.

[77] Lui Sha, Sathish Gopalakrishnan, Xue Liu, and Qixin Wang. "Cyber-physical systems: A new frontier". In: *2008 IEEE international conference on sensor networks, ubiquitous, and trustworthy computing (sutc 2008)*. IEEE. 2008, pp. 1–9.

[78] Alexander Shvets. "Dive Into Design Patterns". In: *Refactoring. Guru* (2018).

[79] Michael Sipser. *Introduction to the Theory of Computation.* Boston, MA, USA: Cengage Learning, 2013.

[80] Quinn Thibeault, Jacob Anderson, Aniruddh Chandratre, Giulia Pedrielli, and Georgios Fainekos. "Psy-taliro: A Python Toolbox for Search-based Test Generation for Cyber-Physical Systems". In: *Formal Methods for Industrial Critical*

Systems: 26th International Conference, FMICS 2021, Paris, France, August 24–26, 2021, Proceedings 26. Springer. 2021, pp. 223–231.

[81] Masaki Waga. "Falsification of cyber-physical systems with robustness-guided black-box checking". In: *Proceedings of the 23rd International Conference on Hybrid Systems: Computation and Control*. 2020, pp. 1–13.

[82] Hengyi Yang. *Dynamic programming algorithm for computing temporal logic robustness*. Tech. rep. Arizona State University, 2013.

[83] Zhenya Zhang, Deyun Lyu, Paolo Arcaini, Lei Ma, Ichiro Hasuo, and Jianjun Zhao. "Effective hybrid system falsification using monte carlo tree search guided by QB-robustness". In: *Computer Aided Verification: 33rd International Conference, CAV 2021, Virtual Event, July 20–23, 2021, Proceedings, Part I*. Springer. 2021, pp. 595–618.

APPENDIX A
ANOTHER TOOL FOR LANGUAGE RECOGNITION

ANother Tool for Language Recognition (ANTLR) [71] is a *parser generator* responsible for creating custom parsers and lexers from a specified grammar file in the meta-language of ANTLR.

Grammars

When designing parsers and lexers, all grammar files must follow Extended Backus-Naur Form (EBNF) [69]. Furthermore, within FoRek, reusing parser and lexer rules should be prioritized over defining new rules that perform the same function. If the construct is syntactically the same but semantically different, then this semantic difference should be reflected during the builder stage and not at the parser level to reduce duplication of work and management of file dependencies.

Targets

ANTLR supports a multitude of targets (currently, ten total). This lends itself to the natural decision that FoRek takes to separate implementation from grammar to allow the ability to generate the same parser and lexer for the supported language that is preferred.

As FoRek is a C++ library, the resulting parsers and lexer generated by ANTLR target C++. It should be noted that the C++ target has a very fast parsing time relative to other targets. However, does generally suffer from a relatively longer warmup time to begin parsing (i.e., time to load underlying structures to parse the input).

Use in FoRek

All grammars defined within the FoRek library using the ANTLR meta-language are located under the `extras/grammars/` directory and placed under an appropriately named folder. For example, the lexer and parser grammar files for pt-LTL are located under `extras/grammars/ltl/past/`, accordingly.

Furthermore, ANTLR generates various interfaces for walking and visiting the resulting parse tree to simplify implementation. This capability is leveraged and is a necessity to retain separation of implementation and definition for each of the languages. The set of interfaces depended upon and their uses are listed below:

- **Lexer**: Performs tokenization of input stream.
- **Parser**: Performs syntactic analysis of token stream from lexer.
- **Visitor**: Explicitly traverses resulting syntax tree.

APPENDIX B
GRAMMAR REFERENCE FOR FOREK

```
parser grammar PropositionalLogicParser;

start : formula EOF ;

formula : LeftParenthesis formula RightParenthesis  #parentheses

    | True                                           #plTrue
    | False                                          #plFalse

    | NegationOperator formula                       #plNegation
    | formula ConjunctionOperator formula            #plConjunction
    | formula DisjunctionOperator formula            #plDisjunction
    | formula ImplicationOperator formula            #plImplication
    | formula IffOperator formula                    #plIff

    | proposition                                    #plProposition
    ;

proposition : Identifier ;
```

Figure B.1: Propositional Logic parser grammar.

```
parser grammar LinearTemporalLogicParser;

import PropositionalLogicParser;

start : formula EOF ;

formula : LeftParenthesis formula RightParenthesis  #parentheses

    | True                                            #plTrue
    | False                                           #plFalse

    | EventuallyOperator formula                      #ltlEventually
    | AlwaysOperator formula                          #ltlAlways
    | NextOperator formula                            #ltlNext
    | formula UntilOperator formula                   #ltlUntil
    | formula ReleaseOperator formula                 #ltlRelease

    | NegationOperator formula                        #plNegation
    | formula ConjunctionOperator formula             #plConjunction
    | formula DisjunctionOperator formula             #plDisjunction
    | formula ImplicationOperator formula             #plImplication
    | formula IffOperator formula                     #plIff

    | proposition                                     #plProposition
    ;
```

Figure B.2: Linear Temporal Logic parser grammar.

```
parser grammar MetricTemporalLogicParser;

import LinearTemporalLogicParser;

start : formula EOF ;

formula : LeftParenthesis formula RightParenthesis  #parentheses

    | True                                           #plTrue
    | False                                          #plFalse

    | EventuallyOperator (interval)? formula         #ltlEventually
    | AlwaysOperator (interval)? formula             #ltlAlways
    | NextOperator (interval)? formula               #ltlNext
    | formula UntilOperator (interval)? formula     #ltlUntil
    | formula ReleaseOperator (interval)? formula   #ltlRelease

    | NegationOperator formula                       #plNegation
    | formula ConjunctionOperator formula            #plConjunction
    | formula DisjunctionOperator formula            #plDisjunction
    | formula ImplicationOperator formula            #plImplication
    | formula IffOperator formula                    #plIff

    | proposition                                    #plProposition
    ;

/// An interval.
///
/// Examples: `(1.0, 2.0)`, `[1, 10)`, `[100.2, 20)`
interval : (LeftParenthesis | LeftBracket) (Scalar | Infinity) Comma (Scalar |
    ↪ Infinity) (RightParenthesis | RightBracket) ;
```

Figure B.3: Metric Temporal Logic parser grammar.

```
parser grammar SignalTemporalLogicParser;

import MetricTemporalLogicParser, ArithmeticParser;

start : formula EOF ;

formula : LeftParenthesis formula RightParenthesis  #parentheses

    | True                                          #plTrue
    | False                                         #plFalse

    | EventuallyOperator (interval)? formula        #ltlEventually
    | AlwaysOperator (interval)? formula            #ltlAlways
    | NextOperator (interval)? formula              #ltlNext
    | formula UntilOperator (interval)? formula     #ltlUntil
    | formula ReleaseOperator (interval)? formula   #ltlRelease

    | NegationOperator formula                      #plNegation
    | formula ConjunctionOperator formula           #plConjunction
    | formula DisjunctionOperator formula           #plDisjunction
    | formula ImplicationOperator formula           #plImplication
    | formula IffOperator formula                   #plIff

    | predicate                                     #stlPredicate
    | proposition                                   #plProposition
    ;

/// An arithmetic expression.
///
/// Examples: `1 + 2 < 1`, `x + 2 >= 1`, `x == y`, `p != q`.
predicate : expression relationalOperator expression ;

/// The set of relational operators.
///
/// Examples: `<=`, `>=`, `<`, `>`, `==`, `!=`.
relationalOperator : LessThanOrEqualTo
    | GreaterThanOrEqualTo
    | LeftChevron
    | RightChevron
    | EqualTo
    | NotEqualTo
    ;
```

Figure B.4: Signal Temporal Logic parser grammar.

```
parser grammar TimedPropositionalTemporalLogicParser;

import LinearTemporalLogicParser, ArithmeticParser;

start : formula EOF ;

formula : LeftParenthesis formula RightParenthesis  #parentheses

    | True                                          #plTrue
    | False                                         #plFalse

    | EventuallyOperator formula                    #ltlEventually
    | AlwaysOperator formula                        #ltlAlways
    | NextOperator formula                          #ltlNext
    | formula UntilOperator formula                 #ltlUntil
    | formula ReleaseOperator formula               #ltlRelease

    | OnceOperator formula                          #ptltlOnce
    | HistoricallyOperator formula                  #ptltlHistorically
    | PreviousOperator formula                      #ptltlPrevious
    | formula SinceOperator formula                 #ptltlSince
    | formula TriggerOperator formula               #ptltlTrigger

    | FreezeTime Identifier formula                 #tptlFreezeTime

    | NegationOperator formula                      #plNegation
    | formula ConjunctionOperator formula           #plConjunction
    | formula DisjunctionOperator formula           #plDisjunction
    | formula ImplicationOperator formula           #plImplication
    | formula IffOperator formula                   #plIff

    | timeConstraint                                #tptlTimeConstraint
    | proposition                                   #plProposition
    ;

/// A time constraint.
///
/// Examples: ‘x <= 1‘, ‘y >= 2.0‘, ‘x + 1 < 2.0‘.
timeConstraint : expression relationalOperator expression ;

/// The set of relation operators.
///
/// Examples: ‘<=‘, ‘>=‘, ‘<‘, ‘>‘, ‘==‘, ‘!=‘.
relationalOperator : LessThanOrEqualTo
    | GreaterThanOrEqualTo
    | LeftChevron
    | RightChevron
    | EqualTo
    | NotEqualTo
    ;
```

Figure B.5: Timed Propositional Temporal Logic parser grammar.

```
parser grammar TimedQualityTemporalLogicParser;

import TimedPropositionalTemporalLogicParser;

start : formula EOF ;

formula : LeftParenthesis formula RightParenthesis  #parentheses

    | True                                          #plTrue
    | False                                         #plFalse

    | EventuallyOperator formula                    #ltlEventually
    | AlwaysOperator formula                        #ltlAlways
    | NextOperator formula                          #ltlNext
    | formula UntilOperator formula                 #ltlUntil
    | formula ReleaseOperator formula               #ltlRelease

    | objectQualifier formula                       #tqtlObjectQualifier
    | FreezeTime Identifier formula                 #tptlFreezeTime

    | NegationOperator formula                      #plNegation
    | formula ConjunctionOperator formula           #plConjunction
    | formula DisjunctionOperator formula           #plDisjunction
    | formula ImplicationOperator formula           #plImplication
    | formula IffOperator formula                   #plIff

    | timeConstraint                                #tptlTimeConstraint
    | proposition                                   #plProposition
    ;

/// Object qualification.
///
/// Examples: `E(obj)@t`, `A(obj1, obj2)@x`.
objectQualifier : ExistsQuantifier LeftParenthesis argumentList RightParenthesis
    ↪ FreezeTime Identifier  #tqtlExistsQualifier
    | ForallQuantifier LeftParenthesis argumentList RightParenthesis FreezeTime
        ↪ Identifier              #tqtlForallQualifier
    ;

/// An argument list
///
/// Examples: `x, y, z`, `p1, p2, _o2`.
argumentList : Identifier Comma argumentList
    | Identifier
    ;
```

Figure B.6: Timed Quality Temporal Logic parser grammar.

```
parser grammar SpatioTemporalPerceptionLogicParser;

import TimedQualityTemporalLogicParser, MetricTemporalLogicParser, ArithmeticParser;

start : formula EOF ;

formula : LeftParenthesis formula RightParenthesis  #parentheses

    | True                                          #plTrue
    | False                                         #plFalse

    | EventuallyOperator formula                    #ltlEventually
    | AlwaysOperator formula                        #ltlAlways
    | NextOperator formula                          #ltlNext
    | formula UntilOperator formula                 #ltlUntil
    | formula ReleaseOperator formula               #ltlRelease

    | OnceOperator formula                          #ptltlOnce
    | HistoricallyOperator formula                  #ptltlHistorically
    | PreviousOperator formula                      #ptltlPrevious
    | formula SinceOperator formula                 #ptltlSince
    | formula TriggerOperator formula               #ptltlTrigger

    | objectQualifier formula                       #tqtlObjectQualifier

    | FreezeTime Identifier formula                 #tptlFreezeTime

    | SpatialExists spatialFormula                  #stplSpatialExists
    | SpatialForall spatialFormula                  #stplSpatialForall

    | NegationOperator formula                      #plNegation
    | formula ConjunctionOperator formula           #plConjunction
    | formula DisjunctionOperator formula           #plDisjunction
    | formula ImplicationOperator formula           #plImplication
    | formula IffOperator formula                   #plIff

    | fnComparison                                  #stplFunctionComparison

    | timeConstraint                                #tptlTimeConstraint
    | proposition                                   #plProposition
    ;
```

Figure B.7: Spatio-Temporal Perception Logic parser grammar (pt. I).

```
/// A spatial (S4) formula.
///
/// Examples: `...`
spatialFormula : EventuallyOperator (interval)? spatialFormula   #
    ↪ stplSpatialEventually
    | AlwaysOperator (interval)? spatialFormula                  #stplSpatialAlways
    | NextOperator (interval)? spatialFormula                    #stplSpatialNext
    | spatialFormula UntilOperator (interval)? spatialFormula    #stplSpatialUntil
    | spatialFormula ReleaseOperator (interval)? spatialFormula  #stplSpatialRelease

    | NegationOperator spatialFormula                            #stplSpatialNegation
    | spatialFormula ConjunctionOperator spatialFormula          #
       ↪ stplSpatialConjunction
    | spatialFormula DisjunctionOperator spatialFormula          #
       ↪ stplSpatialDisjunction

    | InteriorOperator spatialFormula                            #stplSpatialInterior
    | ClosureOperator spatialFormula                             #stplSpatialClosure

    | spatialTerm                                                #stplSpatialTerm
    ;

/// A bounding box function.
///
/// Examples: `BB(obj)`, `BB(x)`.
spatialTerm : BoundingBoxFunction LeftParenthesis Identifier RightParenthesis ;

/// Function comparisons.
///
/// Examples: `PROB(x) >= (PROB(x) - 0.5)`.
fnComparison : fnExpression relationalOperator fnExpression ;

fnExpression : LeftParenthesis fnExpression RightParenthesis #
    ↪ stplFnExpressionParentheses

    | fnInvocation                                           #
       ↪ stplFnExpressionFnInvocation
    | term                                                   #stplFnExpressionTerm

    | fnExpression MultiplicationOperator fnExpression       #stplFnExpressionTimes
    | fnExpression DivisionOperator fnExpression             #
       ↪ stplFnExpressionDivision
    | fnExpression ModuloOperator fnExpression               #stplFnExpressionModulo
    | fnExpression AdditionOperator fnExpression             #stplFnExpressionPlus
    | fnExpression SubtractionOperator fnExpression          #stplFnExpressionMinus
    ;


/// An invocated function (i.e., function call).
///
/// Examples: `PROB(x)`, `DIST(x, BM, y, TM)`, `CLASS(x)`.
fnInvocation : Identifier LeftParenthesis argumentList RightParenthesis ;
```

Figure B.8: Spatio-Temporal Perception Logic parser grammar (pt. II).

APPENDIX C
PUBLISHED WORKS PERMISSION

Within this thesis, the following works were used and/or referenced to supplement, support, and motivate this work:

1. Quinn Thibeault, Jacob Anderson, Aniruddh Chandratre, Giulia Pedrielli, and Georgios Fainekos. “Psy-taliro: A Python Toolbox for Search-based Test Generation for Cyber-Physical Systems”. In: *Formal Methods for Industrial Critical Systems: 26th International Conference, FMICS 2021, Paris, France, August 24–26, 2021, Proceedings 26*. Springer. 2021, pp. 223–231

To this end, I have received permission and acknowledgement from all listed co-authors to use this previously published work in this thesis as needed.