From Formal Requirement Analysis to Testing and Monitoring of Cyber-Physical Systems

by

Adel Dokhanchi

A Dissertation Presented in Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy

Approved September 2017 by the
Graduate Supervisory Committee:

Georgios Fainekos, Chair
Yann-Hang Lee
Hessam Sarjoughian
Aviral Shrivastava

ARIZONA STATE UNIVERSITY

December 2017

ABSTRACT

Cyber-Physical Systems (CPS) are being used in many safety-critical applications. Due to the important role in virtually every aspect of human life, it is crucial to make sure that a CPS works properly before its deployment. However, formal verification of CPS is a computationally hard problem. Therefore, lightweight verification methods such as testing and monitoring of the CPS are considered in the industry. The formal representation of the CPS requirements is a challenging task. In addition, checking the system outputs with respect to requirements is a computationally complex problem. In this dissertation, these problems for the verification of CPS are addressed. The first method provides a formal requirement analysis framework which can find logical issues in the requirements and help engineers to correct the requirements. Also, a method is provided to detect tests which vacuously satisfy the requirement because of the requirement structure. This method is used to improve the test generation framework for CPS. Finally, two runtime verification algorithms are developed for off-line/on-line monitoring with respect to real-time requirements. These monitoring algorithms are computationally efficient, and they can be used in practical applications for monitoring CPS with low runtime overhead.

To my Mother, Farideh Hodania

Father, Mohammad Reza Dokhanchi

Brother, Ali Dokhanchi

ACKNOWLEDGMENTS

At the beginning, I would like to thank my advisor, and my mentor, Professor Georgios Fainekos, who graciously helped me during the last five years. He trusted me and provided me financial support. Without his patience during my learning process, I could not understand any subject that I am presenting in this dissertation. Without his corrections of my mistakes, I would not be able to finish any of my research ideas. He is one of the most intelligent, determined, and disciplined persons that I have ever seen. This dissertation is the outcome of his research projects that he provided me and he helped me to understand, discuss and work on the problems and to find the best solutions. I am proud to be a student of Professor Fainekos.

I would like to thank the committee members Professor Yann-Hang Lee, Professor Hessam Sarjoughian, and Professor Aviral Shrivastava for consulting me since my first semester at ASU. During the first two years at ASU, they encouraged me to be patient and to be positive until I found the best research lab and the right place to continue my Ph.D. Their feedback about my research and education helped me to always think out of the box and to prepare not only for my Ph.D., but also to plan for my future.

The CPS lab that I spend most of my time is one of the key reasons for my success. When I first met the CPS lab members, I found that this is the place that I can build my future. The people in the CPS lab were the best people that I have ever met, and I thank all of them. Bardh Hoxha, Kangjin Kim, Haussam Abbas, Cumhur Erkan Tuncali, Shakiba Yaghoubi, Mohammad Hekmatnejad and Joe Campbell helped me every day for several years. Without their help, I could not finish any of my works. I should thank my best friend, Moslem Didehban for supporting me, and being available anytime that I needed any kind of help in the last four years. He helped me to manage my life in and out of the CPS lab.

My family supported me during my whole life and especially in the last seven years.

Because of them, the financial and emotional supports were the least concerns of mine. This gave me the freedom to follow my instincts and desires during my life at ASU. They always motivated me to work harder and to push my limits to achieve my goals. Without my family, I could not be at this position and write this dissertation. I am blessed to be a member of a family that I can always count on.

I thank the Arizona State University for giving me the admission and providing me an environment to study and research. I always dreamed of study for Ph.D. at a high ranking university, and ASU helped my dreams to become true. I hope in the future, I can help ASU in return. I also thank National Science Foundation (NSF) for the several grants that provided me to fund my research and education. This dissertation is completed because my work was partially supported by the following NFS awards: CNS-1116136, CNS-1350420, IIP-1361926 and the NSF I/UCRC Center for Embedded Systems.

TABLE OF CONTENTS

CHAPTER Page

LIST OF TABLES

LIST OF FIGURES

Figure Page

Chapter 1

INTRODUCTION

Recently, most systems in industrial domains are extremely complex control systems. In order to improve the flexibility, performance, reliability, and safety of these systems, designers use software and algorithms to control physical and industrial systems. Such systems are known as Cyber-Physical Systems (CPS). Medical devices, modern airplanes, automobiles, and smart buildings are some examples of CPS, where the safety critical components of these systems are controlled by embedded computers which interact with the physical environment. Due to the safety critical applications of CPS, there exist strict requirements on system behavior and functional safety. Hence, it is very important to guarantee that a CPS meets the safety requirements, and to validate the correctness of its behavior during designing or prototyping. This process is usually referred to as the CPS verification problem.

In order to simplify design and analysis of CPS, Model Based Design (MBD) and automatic code generation are used for CPS prototypes. Using MBD, we can simplify the design, focus on the critical components of the system, and simulate the model to find the bugs as soon as possible. In order to make it possible to apply formal or semi-formal verification methods on MBD, we need to use mathematical formalization to specify the physical and cyber models. As a result, *hybrid automata* is suggested as a mathematical model to uniquely model both continuous system dynamics of physical models, and the discrete components of the system such as control modes and switching components [59]. However, the verification problem for hybrid automata with respect to safety requirements is undecidable, in general [6, 60]. Hence, a lot of effort has been invested on bounded-time model checking (reachability analysis) and falsification methods (for an overview see [68]).

In the testing method, test team compares the outputs against the requirements. The requirements usually come from the requirements team. In order to provide the safety requirements, the most convenient way is to use natural language. Engineers usually prefer natural language over formal requirements. However, natural language is not accurate enough to provide requirements for safety critical systems. This is because the natural language is ambiguous and it may have more than one interpretation. To have a unique interpretation we need to use formal logics to specify the requirements.

In order to be able to verify real-time requirements, we need to use a formal logic that considers time in its syntax and semantics [10, 9]. Metric Temporal Logic (MTL) was introduced to verify a real-time specifications [71]. Since its introduction, MTL and its variations have been used in the verification of real-time systems [83]. Formal specifications in MTL have been used for testing and verification of CPS with tools such as S-TaLiRo [13] and Breach [41]. As a result, it is very important to provide CPS requirements in MTL. However, providing correct MTL requirements needs a background in formal logic, which many engineers do not have. Even if someone is an expert in MTL, it is very hard to translate natural language requirements for real-time systems into MTL. Therefore, it takes a lot of experience to be an expert on translating requirements into MTL.

To facilitate creating MTL specifications, [64] provided a graphical formalism (ViSpec tool) that can be used for the elicitation of requirements for a subset of MTL. This subset of MTL is expressive enough to define practical requirements for CPS. In [66], the ViSpec tool was evaluated through a usability study which showed that both expert and non-expert users were able to use the tool to elicit formal specifications. The usability study results also indicated that in many cases the developed specifications were incorrect. In an on-line survey[1], they also tested how well formal method experts can translate natural

[1]The on-line survey is available through: `http://goo.gl/forms/YW0reiDtgi` (the results are reported in [66])

language requirements into Metric Interval Temporal Logic (MITL) [7]. The preliminary results indicate that even experts can make errors in their specifications, which indicates that specification correctness is a major issue in the testing and verification since effort can be wasted in checking incorrect requirements, or even worse, the system can pass the incorrect requirements. Therefore, before verification we must analyze the specification and make sure it does not have any logical issues.

I develop new theories and tools for the analysis and monitoring of real time requirements in the context of CPS testing and verification. The high level architecture of CPS testing framework is provided in Figure 1.1. In this figure, the input generator creates initial conditions and inputs to the system under test. An example of a test generation technology that implements the cyclic architecture in Figure 1.1 (the blue cycle with three white boxes) is presented in [1]. The system executes or simulates to generate an output trace. Then, a monitor checks the trace with respect to the specification and reports to the user whether the system trace satisfies or falsifies the specification (for example [49, 78]). The components that we consider in this dissertation are provided in different colors and will be discussed in the following sections:

1.1 Specification Analysis

I first provide a framework that helps the users detect specification errors, where the requirement issues can be corrected before any test and verification process is initiated. This component is provided in Figure 1.1 in Green and is explained in detail in Chapter 4. The Formal Requirement Analysis block checks for erroneous or incomplete temporal logic specifications, without considering the system under test. In other words, the specification issues are caught independent of the system. For example, in the on-line survey from [66], they asked formal method experts to translate the natural language specification *"At some time in the first 30 seconds, the vehicle speed (v) will go over 100 and stay above 100 for*

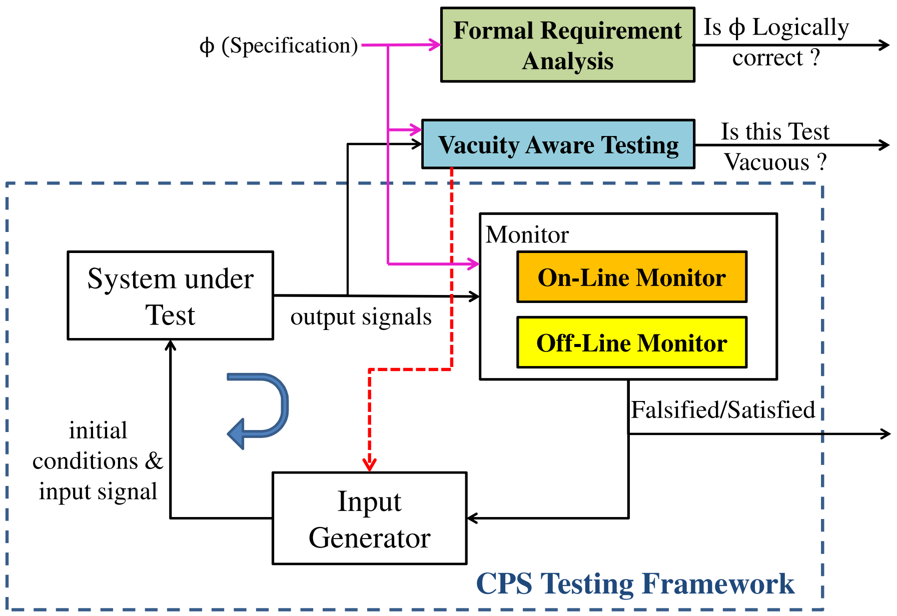

Figure 1.1: CPS Testing Framework with Provided Components.

20 seconds" to MTL. The MTL specification $\varphi = \Diamond_{[0,30]}((v > 100) \rightarrow \Box_{[0,20]}(v > 100))$ was provided as an answer by an expert user. Here, $\Diamond_{[0,30]}$ stands for "*eventually within 30 time units*" and $\Box_{[0,20]}$ for "*always from 0 to 20 time units*". However, the specification φ is a *tautology*, i.e., it evaluates to true no matter what the system behavior is and, thus, the requirement φ is invalid.

Some specification issues cannot be detected unless we consider the system, and test the system behaviors with respect to the specification. Consider the MITL specification $\varphi = \Box_{[0,5]}(req \rightarrow \Diamond_{[0,10]}ack)$. This formula is interpreted as "if at any time within the first 5 seconds, a *request* happens, then from that moment on within the next 10 seconds, an *acknowledge* must happen". The formula φ will successfully pass the Formal Requirement Analysis in Figure 1.1. However, any output signal (See Figure 1.1) that does not satisfy req at any point in time during the test will trivially satisfy φ. We call this issue *vacuous satisfaction* and we call the signals that vacuously satisfy the specification as *vacuous signals*. Any test case that uses vacuous input signals is not a valid test case and the user should be aware of it. We call such bogus tests as *vacuous tests*. We first provide a framework that finds vacuous tests in Chapter 5. Then we use a signal vacuity detection method to improve the S-TaLiRo falsification framework in Chapter 6. The detection and prevention of signal vacuity to happen during testing is called *Vacuity Aware Testing* and it is provided in blue in Figure 1.1.

1.2 Monitoring of Cyber-Physical Systems

Runtime verification is one of the well known methods for checking the correctness of a CPS [92, 75]. In runtime verification a monitoring program checks an execution trace of the CPS and compares it to the desired requirement in a formal logic like MTL. Model checking [30, 16] is not applicable for the verification of CPS, because state space explosion is a limiting factor in model checking. As a result, even when we consider a discrete CPS where the model is finite and the model checking is decidable, the size of the system reduces the efficiency of model checking [31]. In contrast, runtime verification is independent of the system. Runtime verification algorithms' worst case execution time depends on the size of the requirement and/or traces, but not the model. Therefore, it is used for CPS verification in the industry.

There exist two types of runtime verification, off-line monitoring and on-line monitoring. In off-line monitoring, the execution trace is finite and it is saved after running the system for a limited amount of time [49, 78]. The off-line monitoring component is represented in Yellow in Figure 1.1, and it is the subject of Chapter 8. Off-line monitoring has many applications such as system simulation, implementation, testing, and debugging. But, there are some applications where off-line monitoring cannot be used. On-line monitoring is an alternative of the off-line monitoring which can help us for the applications that need the monitor to run simultaneously with the system. In safety critical embedded software, the monitor has to be on-line because of the physical components. Therefore, physical deployment of CPS monitoring cannot happen unless we have an on-line monitor [35, 61].

The on-line monitor component is represented in orange in Figure 1.1 (see Chapter 7).

1.3 Summary of contributions and publications

The summary of contributions and the list of my publication is provided as follows:

1.3.1 Specification Analysis

1. Adel Dokhanchi, Bardh Hoxha and Georgios Fainekos, "Metric interval temporal logic specification elicitation and debugging" ACM/IEEE International Conference on Formal Methods and Models for Codesign, 2015 [36].
   In this paper, I provided the debugging algorithm to check logical issues in real-time specifications. This framework detects validity, redundancy and vacuity issues in formal specifications developed in a fragment of Metric Interval Temporal Logic (MITL) [8] and Signal Temporal Logic (STL) [78]. Our experimental results on the specification collected during the usability study of [66] show that we can benefit from this framework to correct requirements created by the ViSpec software [64]. In the work of [36], I used the MITL satisfiability checker by [23] for finding logical errors in the requirements. In addition, I proved that for the requirements that have only one type of temporal operator, the LTL [84] satisfiability is related to the MITL satisfiability. I also developed a prototype tool for specification debugging (see Chapter 4).

2. Adel Dokhanchi, Bardh Hoxha, and Georgios Fainekos, "Formal Requirement Elicitation and Debugging for Testing and Verification of Cyber-Physical Systems" Accepted for the publication on the ACM Transactions on Embedded Computing Systems journal [37].
   This paper is a journal extension of the previous paper [36]. In this work, I im-

proved the runtime overhead of MITL requirement analysis with using LTL satisfiability solver instead of MITL satisfiability solver. Our experimental results show that the runtime overhead of using the LTL satisfiability solver is negligible. In addition, I provided the framework that signal vacuity can be detected during testing. In Request-Response requirements, the system traces (signals) that vacuously satisfy the requirements are considered vacuous signals and should not be considered as useful tests for example in coverage metrics. After detecting vacuous signals, the user will be notified for further investigation. I also developed a toolbox, that can check the signal vacuity of Matlab/Simulink traces (see Chapter 5).

3. Adel Dokhanchi, Shakiba Yaghoubi, Bardh Hoxha and Georgios Fainekos, "Vacuity Aware Falsification for MTL Request-Response Specifications"[39]
   Vacuous signals cause the system testers to have a false sense of system correctness. Therefore, it is dangerous for CPS verifiers to not identify vacuous signals during testing. On the other hand, Request-Response requirements are typically used in CPS requirements. Falsification of Request-Response requirements is more challenging than other types of requirements. This is because Request-Response requirements have "if-then-else" components and if a signal does not satisfy the "if" part of the requirements, it will vacuously satisfy the "if-then-else" specification. In this paper, we modified the S-TaLiRo testing framework so that it first satisfies the "if" statement and then, falsify the "if-then-else" specification. We showed that the proposed solution can drastically improve the falsification framework for some of the benchmarks (see Chapter 6).

1.3.2 Monitoring of Cyber-Physical Systems

4. Adel Dokhanchi, Bardh Hoxha, and Georgios Fainekos "On-Line Monitoring for Temporal Logic Robustness" International Conference on Runtime Verification, 2014

[35][2].

In this paper, I provided an on-line monitoring algorithm for computing the robustness degree [48] of requirements in bounded future MTL with unbounded past temporal operators $\mathrm{MTL}_{+pt}^{<+\infty}$. My monitoring algorithm was based on the Dynamic Programming method by [87] with polynomial time complexity with respect to the temporal horizon of the $\mathrm{MTL}_{+pt}^{<+\infty}$ formula [53]. I showed that in practical applications with short temporal horizon, the runtime overhead of my algorithm is negligible. I also developed a monitoring block for the library browser of the Matlab/Simulink that users can add to any Simulink diagram and they will be able to monitor on-line any simulating model in Matlab/Simulink (see Chapter 7).

5. Adel Dokhanchi, Bardh Hoxh, Cumhur Erkan Tuncali, and Georgios Fainekos "An efficient algorithm for monitoring practical TPTL specifications" ACM/IEEE International Conference on Formal Methods and Models for Codesign, 2016 [38]

   Monitoring of MTL requirements has been considered for several years [94], and there are many efficient off-line monitoring algorithms for MTL [49, 42]. On the other hand, Timed Propositional Temporal Logic (TPTL) [11] is a real-time requirement representation that is more expressive than MTL and its monitoring is a challenging problem. In this paper, I provided an efficient dynamic programming algorithm for monitoring a fragment of TPTL which is strictly more expressive than MTL. We proved the polynomial time complexity of off-line monitoring for any formula with arbitrary size considering our TPTL fragment (see Chapter 8).

1.3.3 Other CPS related publications (not considered for this dissertation)

4. Adel Dokhanchi, Aditya Zutshi, Rahul T. Sriniva, Sriram Sankaranarayanan and Georgios Fainekos "Requirements driven falsification with coverage metrics" Inter-

[2]An extended version of this paper with proofs is provided in `http://arxiv.org/abs/1408.0045`.

national Conference on Embedded Software, 2015 [40].

MTL is used in S-TaLiRo to falsify safety critical properties of CPS. S-TaLiRo searches for counterexamples to MTL properties for non-linear hybrid systems through global minimization of a robustness metric [48]. The robustness of an MTL formula φ is represented as $[\![\varphi]\!]$, and it is a value that measures how far is the trace from the satisfaction/falsification of φ. This measure is an extension of boolean values $(\top/\bot)$ for representing satisfaction or falsification which is used in conventional monitoring. A positive robustness value means that the trace satisfies the property and a negative robustness means that the property is not satisfied. The stochastic search then returns the simulation trace with the smallest robustness value that was found. The S-TaLiRo testing framework [1, 64] is implemented as a Matlab toolbox that uses stochastic optimization techniques to search for the simulation inputs which falsify the safety requirements presented in MTL [1]. Falsification based approaches for CPS can help us find subtle bugs in industrial size control systems [67].

Coverage guided falsification extends the S-TaLiRo framework with utilizing coverage metrics on the state space of hybrid systems in order to improve the performance and coverage guarantees of the falsification methods. As the search process evolves, coverage statistics are collected for the finite (logical) space of the output in addition to the original output space. If the falsification process fails, then the search algorithm switches to the coverage guided search, where we modify our specification to bias the search towards the desired hybrid locations. In particular, if the original specification is φ and the least visited (or notvisited at all) set of hybrid locations is Q_{Des} , then we define a new MTL formula φ' to incorporate the following requirement "never visit Q_{Des}". The formula φ' now becomes the target for the falsification process. As the falsification algorithm tries to minimize the robustness metric, it indirectly pushes the system to go to the locations provided in φ'. This is due to the fact that the robustness value will be minimized if the hybrid location becomes Q_{Des}. For this work, I developed the testing metrics and I implemented the coverage based test generation method in S-TaLiRo.

5. Bardh Hoxha, Adel Dokhanchi and Georgios Fainekos "Mining Parametric Temporal Logic Properties in Model Based Design for Cyber-Physical Systems" International Journal on Software Tools for Technology Transfer, 2017 [65].
   Parametric Signal Temporal Logic (PSTL) [15] is an STL formula where it has a set of unknown space/time parameters (inside predicates/intervals) of the specification. The problem of parameter mining is as follows: "Given a PSTL formula and a set of parameters, and the range for each parameter's values, find parameters' sub-ranges such that the given system will falsify the PSTL formula with any values in those sub-ranges." In this work, we implemented the parameter estimation as an optimization problem. My main contribution to this paper is the implementation of the algorithm to find the monotonicity of PSTL formulas with respect to the parameters as explained in [15].

6. Bardh Hoxha, Hoang Bach, Houssam Abbas, Adel Dokhanchi, Yoshihiro Kobayashi and Georgios Fainekos "Towards Formal Specification Visualization for Testing and Monitoring of Cyber-Physical Systems" International Workshop on Design and Implementation of Formal Tools and Systems, 2014 [64].
   This workshop paper is an overview of the latest additions to S-TaLiRo.

Chapter 2

HYBRID SYSTEMS

In this dissertation, we consider models of hybrid systems as models for Cyber Physical Systems (CPS) developed within a Model Based Development (MBD) language such as Ptolemy [45] or Matlab Simulink/Stateflow. MBD helps us to define a mathematical representation of the system and to facilitate the analysis and verification of CPS.

2.1 Formal System Representation

In this dissertation, we assume that $\mathbb{R}$ is the set of real numbers, $\mathbb{R}_+$ is the set of non-negative real numbers, and $\overline{\mathbb{R}} = \mathbb{R} \cup \{\pm\infty\}$. Also, $\mathbb{N}$ is the set of natural numbers including 0 and $\mathbb{Z}$ is the set of integers. In addition, $\mathbb{Q}$ is the set of rational numbers, $\mathbb{Q}_+$ is the set of non-negative rational numbers. Given two sets A and B, B^A is the set of all functions from A to B, i.e., for any $f \in B^A$ we have $f : A \to B$. We define $\mathcal{P}(A)$ to be the power set of the set A. We also consider 2^A as the power set of a finite set A. Since we are considering CPS testing or simulation, we fix $T \in \mathbb{R}_+$ to be the maximum simulation time (or similarly maximum time of the signals). A "discrete variable" is a variable that takes value in a countable set such as $\mathbb{N}$, and a "continuous variable" is a variable that takes value in an uncountable set such as $\mathbb{R}$.

Formally, we view a system Σ with states of $\mathcal{X}$ as a mapping from initial conditions $\mathcal{X}_0$, system parameters P and input signals U^R to output signals Y^R. Here, R represents an abstract time domain. For example, $R = \mathbb{N} \times \mathbb{R}$ when we want to talk about the trajectories of hybrid systems [77]. However, in the following, we will just assume that $R = [0, T]$ to avoid many technical issues. In detail, $[0, T]$ can be thought as a dense physical time domain for testing or simulation. Also, U is the set of input values (input space) and Y is the set of output values (output space).

The following three restrictions on the system are critical in order to be algorithmically searchable over an infinite space:

1. The input signals $u \in U^{[0,T]}$ (if any) must be piecewise continuous defined over a finite number of intervals over $[0, T]$. This assumption is necessary in order to be able to parameterize the input signal space over a finite set of parameters. Thus, in the following we assume that any $u \in U^{[0,T]}$ of interest can be represented by a vector of parameter variables p taking values from a set P_U.

2. The output space Y must be equipped with a non-trivial metric. For example, the discrete metric does not provide any useful quantitative information.

3. The system Σ must be deterministic[1]. That is, for a specific initial condition x_0 and input signal u, there must exist a unique output signal $\mathbf{y}$.

The previous restrictions render the system Σ to be a function $\Delta_\Sigma : \mathcal{X}_0 \times P \times P_U \to Y^{[0,T]}$ which takes as input an initial condition vector $x_0 \in \mathcal{X}_0$ and two parameter vectors $p \in P$ and $p' \in P_U$, and produces as output a signal $\mathbf{y} : [0, T] \to Y$.

2.2 Hybrid Automata

Hybrid Automata is a formal representation of a mixed discrete-continuous system [59]. Hybrid Automata is a well known mathematical model to help us formally analyze the behavior of the CPS. This is because Hybrid Automata contains discrete variables to capture the modes of the system as well as continuous variables to capture the physical dynamics of the system.

[1]We remark that this assumption can also be relaxed [3].

Definition 2.2.1 (Hybrid Automata [2]) *Hybrid Automata $\mathcal{H}$ is the following tuple*

$$\mathcal{H} = (\mathcal{X}, L, E, Inv, Flow, Guard, Re)$$

where

- $\mathcal{X} \subseteq \mathbb{R}^n$ is the 'continuous' state of the system and n is the dimension of the system. The continuous state is usually denoted by variable x.
- $L \subset \mathbb{N}$ is the set of control locations or discrete modes of the system.
- $E \subseteq L \times L$ is the set of control switches or location transitions.
- $Inv : L \to \mathcal{P}(\mathcal{X})$ assigns an invariant set to each location.
- $Flow : L \times \mathcal{X} \to \mathbb{R}^n$ defines the time derivative of the continuous dynamics of the system in each location.
- $Guard : E \to \mathcal{P}(\mathcal{X})$ is the guard condition that controls the switch $e = (s, d) \in E$, i.e. the location transition from s to d is enabled when $x \in Guard(e)$.
- $Re : \mathcal{X} \times E \to L \times \mathcal{X}$ is a reset map. That reset the x and the location l to the specified values in the map.

Finally, we set $H = L \times \mathcal{X}$ to denote the state space of a hybrid automaton $\mathcal{H}$, and the initial conditions are denoted as $H_0 \subseteq H$. For a more detailed review of the syntax and semantics of hybrid automata refer to [5, 93]. Now we consider the following simple example:

Example 2.2.1 *[40] Consider the hybrid automata Σ_1 given in Figure 2.1. Briefly, if the initial state $x_0 = (x_{01}, x_{02})$ is in the set S (yellow box in Figure 2.2), then Σ_1 follows the dynamics in location l_2, while if the initial system state x_0 is in $[1, 1]^2 \backslash S$ (green box in Figure 2.2), then Σ_1 follows the dynamics in location l_1. Moreover, if the system is operating*

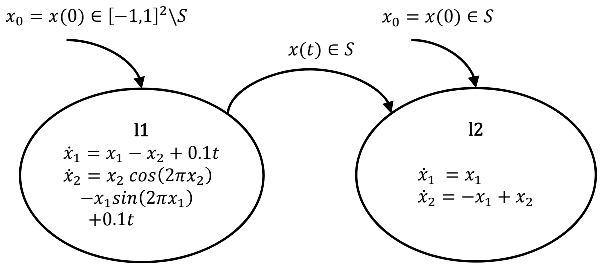

Figure 2.1: The Simple Hybrid Automaton Σ_1 of Example 2.2.1. In This Example, $S = [0.85, 0.95]^2$ and t Is the Time Variable.

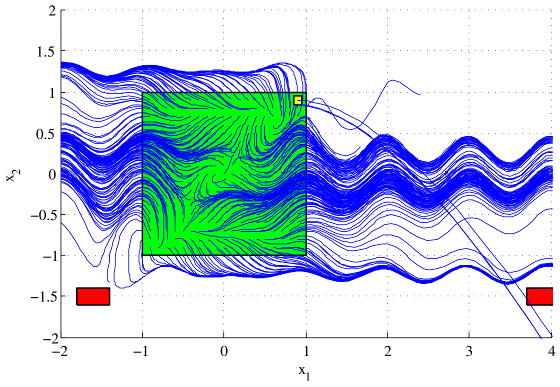

Figure 2.2: The Trajectories of the Hybrid Automaton From Figure 2.1.

under the dynamics in location l_1 and the state of the system enters the set S, then the system switches to location l_2. Sample trajectories with initial conditions over a grid of 0.05 intervals in each dimension over the set of initial conditions $[-1, 1]^2$ *are presented in Figure 2.2.*

2.3 Automatic Transmission (AT)

The Automatic Transmission model is the running example that we will use in most of our experiments throughout this dissertation. The Automatic Transmission (AT) system is provided by Mathworks as a Simulink demo [80]. We introduced a few modifications to the model to make it compatible with the S-TaLiRo framework, which are explained in [63]. This is a model of an automatic transmission controller that contains 69 blocks. AT has two inputs of Throttle and Brake (see Figure 2.3). The throttle and break can take any value

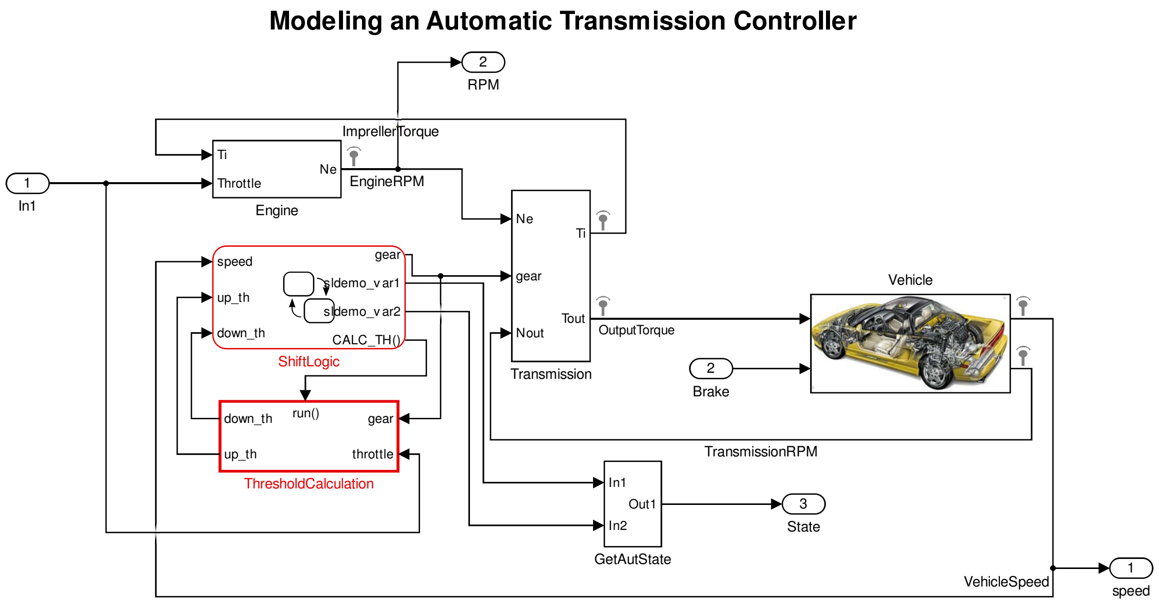

Figure 2.3: Matlab Automatic Transmission Simulink Model [80].

between 0% to 100%, at each point in time. AT contains two continuous state variables: the rotational speed of the engine ω and the speed of the vehicle v.

In addition, AT contains a Stateflow chart of two concurrently executing Finite State Machines (FSM) with 4 and 3 states (see Figure 2.4). Thus, AT is a Simulink model that exhibits both continuous and discrete behavior. The Simulink/Stateflow of AT has the following state space [1]:

$$H = \{\text{first, second, third, fourth}\} \times \{\text{steady state, upshifting, downshifting}\} \times \mathbb{R}^2$$

Since we only consider the `gear_state` as the discrete location of the system that is observed in our requirements, the location state is as follows $L = \{\text{first, second, third, fourth}\} \times \{\text{steady state, upshifting, downshifting}\}$ and $\mathcal{X} = \mathbb{R}^2$.

Figure 2.5 shows a falsifying trajectories which automatically generated by S-TaLiRo [63]. The requirement that the trajectories of Figure 2.5 falsify is as follows:
“The engine speed ω and the vehicle speed v never reach 4500 and 120, respectively”

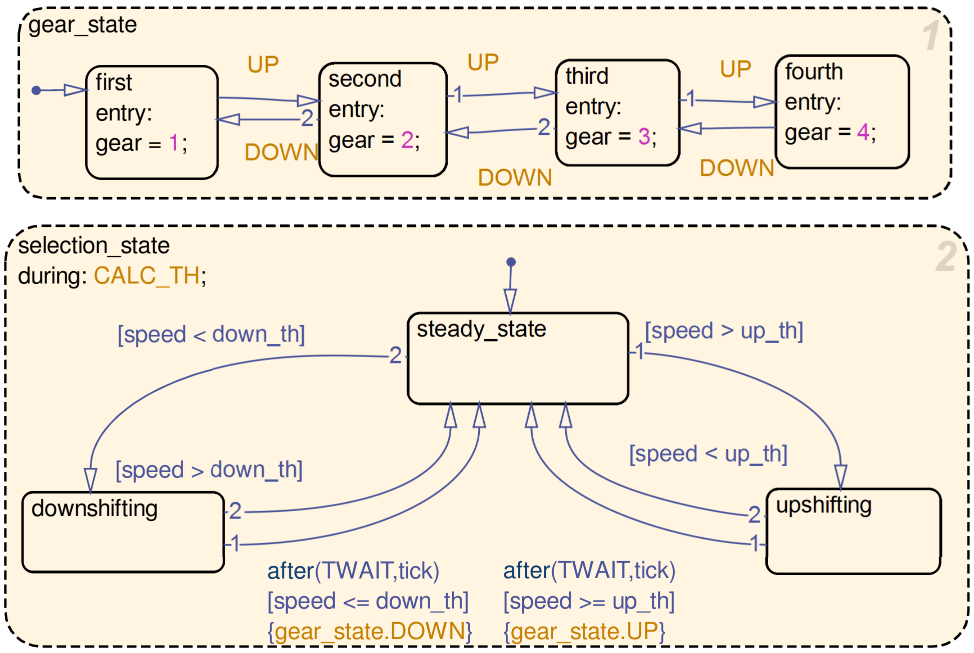

Figure 2.4: The Switching Logic for the Automatic Transmission [80].

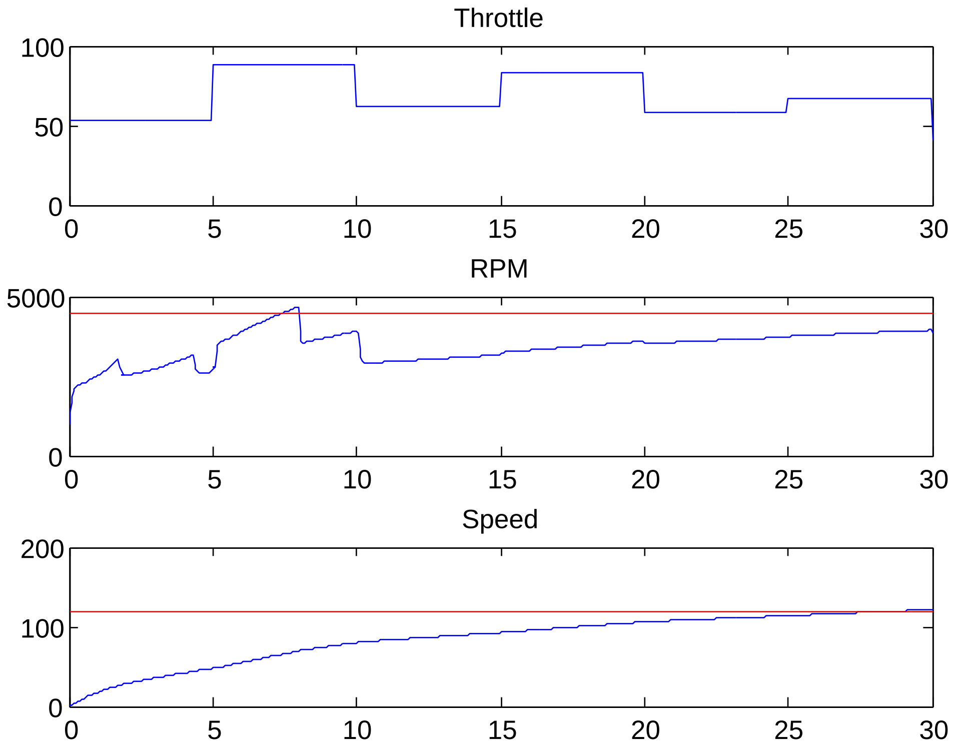

Figure 2.5: Input Throttle (Top) and the Corresponding Outputs of the Automatic Transmission [63].

Chapter 3

FORMAL SPECIFICATIONS

In this chapter, we review the formal representation of system trace and temporal logic in details. This chapter provides the basic formalizations and the background definitions that will be used in the rest of this dissertation. We assume $AP = \{a, b, \cdots\}$ is a set of atomic propositions.

3.1 System Behavior Representation

Each variant of Temporal Logic is interpreted with respect to a different system behavior representation. Therefore, in this section, we consider the different system output representations that will be considered in this dissertation.

3.1.1 Real-Time Signal

We consider system outputs $\mathbf{y} : [0, T] \to Y$, which were introduced in Section 2.1, as real-time signals. Depending on the set Y, $\mathbf{y}$ can be a real-value signal or a state sequence (SS). If Y obtains values $\{\top, \bot\}$ corresponding to existence of an element in a finite set, we consider $\mathbf{y}$ as a state sequence (SS). Metric Interval Temporal Logic (MITL) is interpreted with respect to SS (see Section 3.3.1 Definition 3.3.1). If Y obtains continuous values from $\mathbb{R}$, we consider $\mathbf{y}$ as a real-value signal[1]. Signal Temporal Logic (STL) is interpreted with respect to real-value signals (see Section 3.3.2).

[1]Whether Y is $\mathbb{R}^n$ or a set of atomic propositions is clear from the context.

3.1.2 Timed State Sequence

Since we consider testing and/or simulation, we assume that there exists a sampling function $\tau : \mathbb{N} \to [0, T]$ that returns for each sample i its time stamp $\tau(i)$. In practice, τ is a partial function $\tau : N \to [0, T]$ with $N \subset \mathbb{N}$ and $|N| < \infty$. A *timed state sequence* or *trace* is the pair $\mu = (\mathbf{y} \circ \tau, \tau)$. We will also denote $\mathbf{y} \circ \tau$ by $\tilde{\mathbf{y}}$, where each $\tilde{\mathbf{y}}_k$ contains the values of the state variables of the system at each sampling instance k. The set of all timed state sequences of Σ that correspond to any sampling function τ will be denoted by $\mathcal{L}(\Sigma)$. That is, $\mathcal{L}(\Sigma) = \{(\mathbf{y} \circ \tau, \tau) \mid \exists \tau \in [0, T]^N \,.\, \exists x_0 \in \mathcal{X}_0 \,.\, \exists p \in P \,.\, \exists p' \in P_U \,.\, \mathbf{y} = \Delta_\Sigma(x_0, p, p')\}$. The timed state sequence (TSS) is a widely used model for reasoning about real-time systems [10, 9]. A TSS represents the outcome of a sampling process which is used for digitization of the physical environment. This digitization enables the digital control methods for embedded systems. Using TSS, we assume that system outputs, i.e. signals, satisfy the finite variability assumption. That is, for every finite time interval, the number of times that output changes its value is also finite. The finite variability assumption is an important assumption to have mathematical models which are closely modeling the actual physical world.

3.1.3 Timed State Sequence over Atomic Propositions

Definition 3.1.1 *A state sequence over atomic propositions $\sigma = \sigma_0\sigma_1\sigma_2\cdots$ is an infinite sequence of states $\sigma_i \subseteq AP$, where $i \in \mathbb{N}$. A (sampled) time sequence $\tau = \tau_0\tau_1\tau_2\ldots$ is an infinite sequence of time stamps $\tau_i \in \mathbb{R}_+$, where $i \in \mathbb{N}$.*

A state sequence over atomic propositions σ is a trace of sets of atomic propositions AP [11]. We assume that the time sequence τ is:

1. **Initialized**, which means that the start up time is zero ($\tau_0 = 0$).

2. **Monotonic**, which means that $\tau_i \leq \tau_{i+1}$ for all $i \in \mathbb{N}$.

3. **Progressive**, which means that for all $t \in \mathbb{R}_+$ there is some $i \in \mathbb{N}$ such that $\tau_i > t$.

Definition 3.1.2 (Timed State Sequence over Atomic Propositions (ATSS) [11]) *A Timed State Sequence over Atomic Propositions (ATSS) $\rho = (\sigma, \tau)$ is a pair consisting of a state sequence over atomic propositions σ and a time sequence τ where* $\rho_0\rho_1\rho_2 \cdots = (\sigma_0, \tau_0)(\sigma_1, \tau_1)(\sigma_2, \tau_2) \cdots$.

Given an infinite ATSS ρ, we consider a finite prefix of ρ as a finite ATSS. The symbol $\hat{\rho} = (\hat{\sigma}, \hat{\tau})$ is used to denote a finite ATSS with the size of $|\hat{\rho}| = |\hat{\sigma}| = |\hat{\tau}|$. In Chapter 8, we consider the monitoring of finite ATSS with the size of $|\hat{\rho}|$ which is equal to the number of simulation/execution samples.

3.2 Temporal Logic with Point-Based Semantics

We assume a sampled representation of a system behavior with an ATSS as the output of the system. This is conventional in CPS since we use a digital clock in the system.

3.2.1 Linear Temporal Logic with Past (PLTL)

Linear Temporal Logic (LTL) was first introduced by Pnueli for formal verification of concurrent programs [84]. In this section, we provide the syntax and semantics of Linear Temporal Logic with Past (PLTL) [74].

Definition 3.2.1 (Syntax for $PLTL$**)** *The set of PLTL formulas ϕ over a finite set of atomic propositions (AP) is inductively defined according to the following grammar:*

$$\phi \ ::= \ \top \mid a \mid \neg\phi \mid \phi_1 \wedge \phi_2 \mid \phi_1 \vee \phi_2 \mid \bigcirc \phi \mid \phi_1 U \phi_2 \mid \odot \phi \mid \phi_1 S \phi_2$$

where $\top$ is the symbol for "True". Temporal operators are as follows: $\bigcirc$ is "Next", U is "Until", $\odot$ is "Previous", and S is "Since". Note that "False" is represented as $\bot \equiv \neg\top$ and "Implication" is represented as $\varphi_1 \rightarrow \varphi_2 \equiv \neg\varphi_1 \vee \varphi_2$. For any formula φ, $\Diamond\varphi \equiv \top U \varphi$

(Eventually), $\Box\varphi \equiv \neg\Diamond\neg\varphi$ (Always), $\diamondminus\varphi \equiv \top S\varphi$ (Eventually in the past), and $\boxminus\varphi \equiv \neg\diamondminus\neg\varphi$ (Always in the past) operators are defined, respectively.

Definition 3.2.2 (Semantics of $PLTL$ [74]) *Let ρ be an ATSS, $i \in \mathbb{N}$ and ϕ be a PLTL formula, we inductively define the satisfaction relation $(\rho, i) \models \phi$ by:*

$(\rho, i) \models \top$

$(\rho, i) \models a$ iff $a \in \sigma_i$

$(\rho, i) \models \neg\phi$ iff $(\rho, i) \not\models \phi$

$(\rho, i) \models \phi_1 \wedge \phi_2$ iff $(\rho, i) \models \phi_1$ and $(\rho, i) \models \phi_2$

$(\rho, i) \models \phi_1 \vee \phi_2$ iff $(\rho, i) \models \phi_1$ or $(\rho, i) \models \phi_2$

$(\rho, i) \models \bigcirc\phi$ iff $(\rho, i+1) \models \phi$

$(\rho, i) \models \phi_1 U \phi_2$ iff $(\rho, j) \models \phi_2$ for some $j \geq i$ s.t. $(\rho, k) \models \phi_1$ for all $i \leq k < j$

$(\rho, i) \models \odot\phi$ iff $i > 0$ and $(\rho, i-1) \models \phi$

$(\rho, i) \models \phi_1 S \phi_2$ iff $(\rho, j) \models \phi_2$ for some $0 \leq j \leq i$ s.t. $(\rho, k) \models \phi_1$ for all $j < k \leq i$

Intuitively $(\rho, i) \models \phi$ means that ϕ holds at position i in ρ [74]. We say that ρ satisfies ϕ or $\rho \models \phi$ iff $(\rho, 0) \models \phi$.

3.2.2 Metric Temporal Logic with Past (MTL_P)

Metric Temporal Logic (MTL) is the real-time version of LTL where the temporal operators are annotated with Time Intervals that impose time constraints on the temporal operations [71]. In this section, we consider the syntax and semantics of Metric Temporal Logic with Past (MTL_P) [10].

Definition 3.2.3 (Syntax for MTL_P [10]) *The set of MTL_P formulas ϕ over a finite set of atomic propositions (AP) is inductively defined according to the following grammar:*

$$\phi ::= \top \mid a \mid \neg\phi \mid \phi_1 \wedge \phi_2 \mid \bigcirc_{\mathcal{I}} \phi \mid \phi_1 U_{\mathcal{I}} \phi_2 \mid \odot_{\mathcal{I}} \phi \mid \phi_1 S_{\mathcal{I}} \phi_2$$

where $\mathcal{I}$ is an interval over $\mathbb{R}_+ \cup \{+\infty\}$. An interval is defined by the following template $\langle u, r\rangle$ where $\langle$ is (or $\langle$ is [when the interval is left open or close, respectively. Similarly, $\rangle$ is) or $\rangle$ is] when the interval is right open or close, respectively. When we have universal interval $\mathcal{I} = [0, +\infty)$ we simply remove the $\mathcal{I}$ and temporal operator is equivalent to LTL operator (Definition 3.2.1). For each value $r \in \mathbb{R}_+$ and interval $\mathcal{I}$, we define +,− operators as follows $r + \mathcal{I} := \{r + t \mid t \in \mathcal{I}\}$ and $r - \mathcal{I} := \{r - t \mid t \in \mathcal{I}\} \cap \mathbb{R}_+$. For example, if $\mathcal{I}$ is an interval of the form $[l, u)$, the expressions $y \in x + \mathcal{I}$ and $y \in x - \mathcal{I}$ stand for the following timing constraints $x + l \leq y < x + u$ and $x - u < y \leq x - l$, respectively [10]. For any MTL_P formula φ, $\Diamond_{\mathcal{I}}\varphi \equiv \top U_{\mathcal{I}}\varphi$ (Eventually), $\Box_{\mathcal{I}}\varphi \equiv \neg\Diamond_{\mathcal{I}}\neg\varphi$ (Always), $\Diamond\!\!\!\cdot_{\mathcal{I}}\varphi \equiv \top S_{\mathcal{I}}\varphi$ (Eventually in the past), and $\boxdot_{\mathcal{I}}\varphi \equiv \neg\Diamond\!\!\!\cdot_{\mathcal{I}}\neg\varphi$ (Always in the past) operators are defined respectively.

Definition 3.2.4 (Semantics of MTL_P) *Let ρ be an ATSS, $i \in \mathbb{N}$ and ϕ a MTL_P formula, we inductively define the satisfaction relation $(\rho, i) \models \phi$ by:*

$(\rho, i) \models \top$

$(\rho, i) \models a$ iff $a \in \sigma_i$

$(\rho, i) \models \neg\phi$ iff $(\rho, i) \not\models \phi$

$(\rho, i) \models \phi_1 \wedge \phi_2$ iff $(\rho, i) \models \phi_1$ and $(\rho, i) \models \phi_2$

$(\rho, i) \models \bigcirc_{\mathcal{I}}\phi$ iff $\tau_{i+1} \in \tau_i + \mathcal{I}$ and $(\rho, i + 1) \models \phi$

$(\rho, i) \models \phi_1 U_{\mathcal{I}} \phi_2$ iff $(\rho, j) \models \phi_2$ for some $j \geq i$ where $\tau_j \in \tau_i + \mathcal{I}$ s.t. $(\rho, k) \models \phi_1$ for all $i \leq k < j$

$(\rho, i) \models \odot_{\mathcal{I}}\phi$ iff $i > 0$ and $\tau_{i-1} \in \tau_i - \mathcal{I}$ and $(\rho, i-1) \models \phi$

$(\rho, i) \models \phi_1 S_{\mathcal{I}} \phi_2$ iff $(\rho, j) \models \phi_2$ for some $0 \leq j \leq i$ where $\tau_j \in \tau_i - \mathcal{I}$ s.t. $(\rho, k) \models \phi_1$ for all $j < k \leq i$

Intuitively $(\rho, i) \models \phi$ means that ϕ holds at position i in ρ. We say that ρ satisfies ϕ or $\rho \models \phi$ iff $(\rho, 0) \models \phi$.

3.2.3 Timed Propositional Temporal Logic (TPTL)

TPTL is an extension of LTL that enables the formalization of real-time properties by including time variables and a freeze time quantifier [11]. In this dissertation, we consider Raskin's TPTL semantics [85, 27] [2].

Definition 3.2.5 (Syntax for TPTL) *The set of TPTL formulas ϕ over a finite set of atomic propositions (AP) and a finite set of time variables (V) is inductively defined according to the following grammar:*

$$\phi \ ::= \ \top \mid a \mid x \sim r \mid \neg\phi \mid \phi_1 \wedge \phi_2 \mid \phi_1 \vee \phi_2 \mid \bigcirc \phi \mid \phi_1 U \phi_2 \mid x.\phi$$

where $x \in V$, $r \in \mathbb{R}_+$, $a \in AP$, and $\sim \in \{\leq, <, =, >, \geq\}$. The time constraints of TPTL are represented in the form of $x \sim r$. The freeze quantifier $x.$ assigns the current time of the formula's evaluation (at each sampled time τ_i) to the time variable x. A TPTL formula is *closed* if every occurrence of a time variable is within the scope of a freeze quantifier [11]. In the TPTL specifications, we always deal with closed formulas. Since we focus on off-line monitoring (see Chapter 6), we only consider the TPTL semantics for finite traces.

Definition 3.2.6 ($TPTL$ Semantics) *Let $\hat{\rho} = (\hat{\sigma}, \hat{\tau})$ be a finite ATSS and $i \in \mathbb{N}$ where $i < |\hat{\rho}|$ is the index of the current sample, $a \in AP$, $\phi \in TPTL$, and an environment $\varepsilon : V \to \mathbb{R}_+$. The satisfaction relation $(\hat{\rho}, i, \varepsilon) \models \phi$ is defined recursively as follows:*

[2] We will explain in Section 8.2 why we chose Raskin's semantics.

$(\hat{\rho}, i, \varepsilon) \models \top$

$(\hat{\rho}, i, \varepsilon) \models a$ iff $a \in \sigma_i$

$(\hat{\rho}, i, \varepsilon) \models \neg\phi$ iff $(\hat{\rho}, i, \varepsilon) \not\models \phi$

$(\hat{\rho}, i, \varepsilon) \models \phi_1 \wedge \phi_2$ iff $(\hat{\rho}, i, \varepsilon) \models \phi_1$ and $(\hat{\rho}, i, \varepsilon) \models \phi_2$

$(\hat{\rho}, i, \varepsilon) \models \bigcirc\phi$ iff $(\hat{\rho}, i + 1, \varepsilon) \models \phi$ and $i < (|\hat{\rho}| - 1)$

$(\hat{\rho}, i, \varepsilon) \models \phi_1 U \phi_2$ iff $\exists j, i \leq j < |\hat{\rho}|$ s.t. $(\hat{\rho}, j, \varepsilon) \models \phi_2$ and $\forall k, i \leq k < j$ it holds that $(\hat{\rho}, k, \varepsilon) \models \phi_1$

$(\hat{\rho}, i, \varepsilon) \models x \sim r$ iff $(\tau_i - \varepsilon(x)) \sim r$ i.e. (current time stamp) $- \varepsilon(x) \sim r$

$(\hat{\rho}, i, \varepsilon) \models x.\phi$ iff $(\hat{\rho}, i, \varepsilon[x := \tau_i]) \models \phi$

TPTL semantics are defined over an evaluation function $\varepsilon : V \to \mathbb{R}_+$ which is an environment for the time variables. Assume $x = r$ where $x \in V$, and $r \in \mathbb{R}_+$, then we have $\varepsilon(x) = r$. Given a variable $x \in V$ and $q \in \mathbb{R}_+$, we denote the environment with $\varepsilon' = \varepsilon[x := q]$ which is equivalent to the environment ε on all time variables in V except variable x. The assignment operation $x := q$ changes the environment ε to the new environment ε'. Formally, $\varepsilon'(y) = \varepsilon(y)$ for all $y \neq x$ and $\varepsilon'(x) = q$. We write $\mathbf{0}$ for the (**zero**) environment such that $\mathbf{0}(x) = 0$ for all $x \in V$. We say that $\hat{\rho}$ satisfies ϕ ($\hat{\rho} \models \phi$) iff $(\hat{\rho}, 0, \mathbf{0}) \models \phi$. A variable "$x$" that is bounded by a corresponding *freeze quantifier* "x." saves the local temporal context τ_i (now) in "x". Assume $\varphi(x)$ is a formula with a free variable x. The ATSS $\hat{\rho}$ satisfies $x.\varphi(x)$ if it satisfies $\varphi(\tau_0 = 0)$, where $\varphi(0)$ is obtained from $\varphi(x)$ by replacing all the free occurrences of the variable x with constant 0 [11].

3.3 Temporal Logic with Continuous Semantics

In this section we consider the variants of Temporal Logic that are interpreted with respect to continuous signals (Section 3.1.1).

3.3.1 Metric Interval Temporal Logic (MITL)

Metric Interval Temporal Logic (MITL) is MTL where the timing constraints are not allowed to be singleton sets [8]. The MITL formulas are interpreted with respect to State Sequence (SS):

Definition 3.3.1 (State Sequence) *A State Sequence (SS) is a mapping from the bounded real line to sets of atomic propositions* ($\mathbf{y} : [0, T] \to 2^{AP}$).

We assume that the signals satisfy the finite variability condition (non-Zeno condition) as it is standard in the academia[3].

In the corresponding Chapter 4, we restrict our focus to a fragment of MITL called Bounded-MITL($\Diamond$,$\Box$) where the only temporal operators allowed are *Eventually* ($\Diamond$) and *Always* ($\Box$) operators with timing intervals. Formally, the syntax of Bounded-MITL($\Diamond$,$\Box$) is defined by the following grammar:

Definition 3.3.2 (Bounded-MITL($\Diamond$,$\Box$) syntax)

$$\phi \ ::= \ \top \mid \bot \mid a \mid \neg a \mid \phi_1 \wedge \phi_2 \mid \phi_1 \vee \phi_2 \mid \Diamond_{\mathcal{I}} \phi_1 \mid \Box_{\mathcal{I}} \phi_1$$

where $a \in AP$, $\mathcal{I}$ is a nonsingular interval over $\mathbb{Q}_+$ with defined end-points. The interval $\mathcal{I}$ is right-closed. We assume that Bounded-MITL($\Diamond$,$\Box$) formulas are in Negation Normal Form (NNF) where the negation operation is only applied on atomic propositions. NNF is

[3]The satisfiability tools for MITL that we use in Section 4.5.1, assume that the traces satisfy the finite variability condition [25].

easily obtainable by iteratively applying the following transformation: $\neg\Diamond_{\mathcal{I}}\varphi \equiv \Box_{\mathcal{I}}\neg\varphi$ and $\neg\Box_{\mathcal{I}}\varphi \equiv \Diamond_{\mathcal{I}}\neg\varphi$. NNF formulas only contain the following boolean operators of $(\wedge, \vee)$.

Definition 3.3.3 (Bounded-MITL($\Diamond$,$\Box$) semantics in NNF) *Given an SS* $\mathbf{y} : [0, T] \rightarrow 2^{AP}$ *and* $t, t' \in [0, T]$*, and an MITL formula* ϕ*, the satisfaction relation* $(\mathbf{y}, t) \models \phi$ *is inductively defined as:*

$(\mathbf{y}, t) \models \top$

$(\mathbf{y}, t) \models a$ *iff* $a \in \mathbf{y}(t)$

$(\mathbf{y}, t) \models \neg a$ *iff* $a \notin \mathbf{y}(t)$

$(\mathbf{y}, t) \models \phi_1 \wedge \phi_2$ *iff* $(\mathbf{y}, t) \models \phi_1$ *and* $(\mathbf{y}, t) \models \phi_2$

$(\mathbf{y}, t) \models \phi_1 \vee \phi_2$ *iff* $(\mathbf{y}, t) \models \phi_1$ *or* $(\mathbf{y}, t) \models \phi_2$

$(\mathbf{y}, t) \models \Diamond_{\mathcal{I}}\phi_1$ *iff* $\exists t' \in (t + \mathcal{I}) \cap [0, T]$ *s.t* $(\mathbf{y}, t') \models \phi_1$.

$(\mathbf{y}, t) \models \Box_{\mathcal{I}}\phi_1$ *iff* $\forall t' \in (t + \mathcal{I}) \cap [0, T]$, $(\mathbf{y}, t') \models \phi_1$.

An SS $\mathbf{y}$ satisfies a Bounded-MITL($\Diamond$,$\Box$) formula ϕ (denoted by $\mathbf{y} \models \phi$), iff $(\mathbf{y}, 0) \models \phi$. For simplifying the presentation, when we mention MITL, we mean Bounded-MITL($\Diamond$,$\Box$). Given MITL formulas φ and ψ, φ satisfies ψ, denoted by $\varphi \models \psi$ iff $\forall \mathbf{y}.\mathbf{y} \models \varphi \Rightarrow \mathbf{y} \models \psi$. We use $\varphi \in \psi$ to denote that φ is a subformula of ψ.

3.3.2 Signal Temporal Logic (STL)

The logic and semantics of MITL can be extended to real-valued signals through Signal Temporal Logic (STL) [78].

Definition 3.3.4 (Signal Temporal Logic [78]) *Let* $s : [0, T] \rightarrow \mathbb{R}^m$ *be a real-valued signal, and* $\Pi = \{\pi_1, ..., \pi_n\}$ *be a collection of predicates or boolean functions of the form* $\pi_i : \mathbb{R}^m \rightarrow \mathbb{B}$ *where* $\mathbb{B} = \{\top, \bot\}$ *is a boolean value.*

For any STL formula Φ_{STL} over predicates Π, we can define a corresponding MITL formula Φ_{MITL} over some atomic propositions AP as follows:

1. Define a set AP such that for each $\pi \in \Pi$, there exist some $a_\pi \in AP$
2. For each real-valued signal s we define a $\mathbf{y}$ such that $\forall t.a_\pi \in \mathbf{y}(t)$ iff $\pi(s(t)) = \top$
3. $\forall t.(s, t) \models \Phi_{STL}$ iff $(\mathbf{y}, t) \models \Phi_{MITL}$

The traces resulting from abstractions through predicates of signals from physical systems satisfy the finite variability assumption. For practical applications, the finite variability assumption is satisfied. Since we focus on CPS in this dissertation, with a slight abuse of terminology, we may use the term signal to refer to both a TSS μ and a signal $\mathbf{y}$.

3.4 Temporal Logic with Robustness Semantics

Intuitively, robustness semantics is an extension of the Boolean semantics of temporal logic. Robust semantics evaluate to positive values if the trajectory satisfies the specification and it evaluates to negative values if the trajectory violates the specification. Moreover, the magnitude of the robust evaluation indicates how robustly the behavior satisfies or violates the specification.

Temporal Logic (TL) can capture many system requirements by defining a set of atomic propositions AP which labels subsets of the output space Y. We define those subsets through an observation map $\mathcal{O} : AP \rightarrow \mathcal{P}(Y)$ where each $\pi \in AP$ is mapped to a set $\mathcal{O}(\pi) \subset Y$.

3.4.1 Euclidean Distance Metric

Euclidean distance metric is used to capture the continuous system behavior for the atomic propositions of TL. Using a *metric* $\mathbf{d}$ [90], we can define a distance function that captures how far away a point $y \in Y$ is from a set $S \subseteq Y$. Intuitively, the distance function assigns positive values when y is in the set S and negative values when y is outside the set S. The metric $\mathbf{d}$ must be at least a generalized quasi-metric as described in [1] which also includes the case where $\mathbf{d}$ is a metric as it was introduced in [48].

Definition 3.4.1 (Signed Distance) *Let $y \in Y$ be a point, $S \subseteq Y$ be a set and $\mathbf{d}$ be a metric. Then, we define the Signed Distance from y to S to be*

$$\mathbf{Dist_d}(y, S) := \begin{cases} -\mathbf{dist_d}(y, S) & \text{if } y \notin S \\ \mathbf{dist_d}(y, Y \backslash S)\} & \text{if } y \in S \end{cases}$$

where

$$\mathbf{dist_d}(y, S) := \inf\{\mathbf{d}(y, y') \mid y' \in S\}$$

and inf *is the infimum.*

We should point out that we use the extended definition of supremum ($\sqcup$) and infimum ($\sqcap$). In other words, the supremum of the empty set is defined to be bottom element of the domain, while the infimum of the empty set is defined to be the top element of the domain. For example, $\sup \emptyset := -\infty$ and $\inf \emptyset := +\infty$.

3.4.2 Hybrid Distance Metric

S-TaLiRo [13] originally supported the following metrics. When $Y = \mathbb{R}^n$, then we use the Euclidean metric $d(y_1, y_2) = \|y_1 - y_2\|$. When Y is a hybrid space, we assume that the output space Y of the system Σ comprises of the original output space Y_Σ of the system (equipped with the nontrivial metric), and a finite space Y_F. That is, $Y = Y_\Sigma \times Y_F$ with $Y_\Sigma = \mathbb{R}^n$ and $Y_F = Q$, where Q is the set of states of a Hybrid Automata $\mathcal{H}$ (Section 2.2), then we use the following generalized quasi-metric [82, 1] $\mathbf{d_h} : Y \times Y \to \left\langle \mathbb{N} \cup \infty, \overline{\mathbb{R}}_+ \right\rangle$ with definition:

$$\mathbf{d_h}(\langle x, q\rangle, \langle x', q'\rangle) = \begin{cases} \langle 0, d(x, x')\rangle & \text{if } q = q' \\ \left\langle \pi(q, q'), \min\limits_{q \to q'' \rightsquigarrow q'} \mathbf{dist_d}(x, Guard(q, q'')) \right\rangle & \text{otherwise} \end{cases}$$

where $\pi(q, q')$ is the size of the shortest path between q and q' on the graph of $\mathcal{H}$. The distance metric $\mathbf{d}$ is on $\mathbb{R}^n$, and $Guard(q, q'')$ denotes that the switching guard that activates the transition from state q to state q'' in the transitions set $E \in \mathcal{H}$ (see Definition 2.2.1). State q'' is chosen from all the neighbors of q that are one step closer to q' in the shortest path (with the size equal to $\pi(q, q')$). We assume that $Guard(q, q'') \subseteq Y_\Sigma$. Finally, the "min" operator quantifies over all neighboring states q'' of q that are on a shortest path from q to q'. A more detailed discussion can be found in [1]. When the two points $\langle x, q\rangle, \langle x', q'\rangle$ are in the same locations, then the distance computation reduces to the distance computation between the points in the continuous state space. When the two points $\langle x, q\rangle, \langle x', q'\rangle$ are in different locations, then the distance is the path distance between the two locations "weighted" by the distance to the closest guard that will enable the transition to the next locations (q'') that reduces the path distance. Essentially, the last condition is a heuristic that chooses one of the shortest paths.

The generalized distance function $\mathbf{d_h}$ is computationally complex. On the other hand, the *Guard* may not be defined in some $\mathcal{H}$s. Therefore, we define the following simplified generalized quasi-metric [82, 1] $\mathbf{d_h^0} : Y \times Y \to \left\langle \mathbb{N} \cup \infty, \overline{\mathbb{R}}_+ \right\rangle$:

$$\mathbf{d_h^0}(\langle x, q\rangle, \langle x', q'\rangle) = \begin{cases} \langle 0, d(x, x')\rangle & \text{if } q = q' \\ \langle \pi(q, q'), 0\rangle & \text{if } q \neq q' \text{ and } \pi(q, q') < +\infty \\ \langle +\infty, +\infty\rangle & \text{otherwise} \end{cases}$$

The distance function $\mathbf{d_h^0}$ ignores the guards and considers distance metric $\mathbf{d}$ only when the two points are in the same location in $\mathcal{H}$ [1].

3.4.3 Multiple Hybrid Distance Metric

When dealing with industrial size models, it is unrealistic to assume that there is going to be a single hybrid automata $\mathcal{H}$ or that it is going to be efficient to flatten all the different hybrid automata into a single one $\mathcal{H}'$. On the other hand, the hybrid system modeled as a Simulink or Ptolemy [45] model may have various discrete blocks such as statechart, switch, saturation or if-then-else block. In this case, the hybrid metric needs to resolve the distance metric of all the discrete blocks [40]. Therefore, we have modified S-TaLiRo to handle multiple finite state components as follows: We extend the metric in a natural way using the maximum pairwise hybrid distance. Namely, when $Y = Y_\Sigma \times Y_F$, where $Y_F = Q_1 \times \ldots \times Q_m$, and $Q_1, \ldots, Q_m$ are the sets of states of m different $\mathcal{H}$s, we use the following metric:

$$\mathbf{d}_{\mathbf{h}}^{\max}(\langle x, q_1, \ldots, q_m\rangle, \langle x', q'_1, \ldots, q'_m\rangle) = \max\{\mathbf{d}_{\mathbf{h}}(\langle x, q_1\rangle, \langle x', q'_1\rangle), \ldots, \mathbf{d}_{\mathbf{h}}(\langle x, q_m\rangle, \langle x', q'_m\rangle).\}$$

The multiple hybrid distance metric is used to define different coverage metrics for addressing the coverage guided falsification of hybrid systems [40].

3.4.4 Robustness Semantics for MTL_P

The formulas in MTL_P state requirements over the observable trajectories of a CPS. In order to capture these requirements, each predicate $p \in AP$ is mapped to a subset of the metric space. Note that, we use an observation map $\mathcal{O}$ to interpret each predicate $p \in AP$, where the observation map is defined as $\mathcal{O} : AP \to \mathcal{P}(Y)$ such that for each $p \in AP$ the corresponding set is $\mathcal{O}(p)$. We define the robust valuation of an MTL_P formula φ over a TSS $\tilde{\mathbf{y}}$ as follows [47].

Definition 3.4.2 (*MTL_P* Robustness Semantics) *Let μ be a timed state sequence as defined in Section 3.1.2, and $\mathcal{O}$ be an observation map $\mathcal{O} : AP \to \mathcal{P}(Y)$, then the robust semantics of any formula $\varphi \in MTL_P$ with respect to μ is recursively defined as:*

$[\![\top]\!](\mu, i) := \bigsqcup Range(\mathbf{d})$

$[\![p]\!](\mu, i) := \mathbf{Dist}_{\mathbf{d}}(\tilde{\mathbf{y}}_i, \mathcal{O}(p))$ *see Definition 3.4.1*

$[\![\neg\varphi]\!](\mu, i) := -[\![\varphi]\!](\mu, i)$

$$[\![\psi \vee \varphi]\!](\mu, i) := [\![\psi]\!](\mu, i) \sqcup [\![\varphi]\!](\mu, i)$$

$$[\![\psi U_{\mathcal{I}} \varphi]\!](\mu, i) := \bigsqcup_{j \in \tau^{-1}(\tau(i)+\mathcal{I})} \left([\![\varphi]\!](\mu, j) \sqcap \bigsqcap_{i \le k < j} [\![\psi]\!](\mu, k) \right)$$

$$[\![\psi S_{\mathcal{I}} \varphi]\!](\mu, i) := \bigsqcup_{j \in \tau^{-1}(\tau(i)-\mathcal{I})} \left([\![\varphi]\!](\mu, j) \sqcap \bigsqcap_{i \le k < j} [\![\psi]\!](\mu, k) \right)$$

where τ^{-1} is the inverse function of τ, and $-$ is an unary operator defining the "negative" values of the range of $\mathbf{d}$, i.e., *Range*($\mathbf{d}$). A trace μ satisfies an MTL_P formula ϕ (denoted by $\mu \models \phi$), if $[\![\phi]\!]_{\mathbf{d}}(\mu, 0) > 0$. On the other hand, a trace μ' falsifies an MTL_P formula ϕ (denoted by $\mu' \not\models \phi$), if $[\![\phi]\!]_{\mathbf{d}}(\mu', 0) < 0$.

3.4.5 Syntax and Semantics for MTL$^{<+\infty}_{+pt}$

The set of formulas of MTL$^{<+\infty}_{+pt}$ is a subset of MTL_P. We refine MTL_P into MTL$^{<+\infty}_{+pt}$ to make a feasible and efficient solution for the on-line monitoring problem (see Chapter 7). We must assume a fixed sampling period for the system, where there exists a fixed time period between consecutive time stamps. Using the fixed time period $\Delta t > 0$, for all $i \geq 0$, we have $\tau_{i+1} - \tau_i = \Delta t$. As a result, we can simply compute each time stamp τ_i knowing the trace index (or simulation step) i by this multiplication $\tau_i = i\Delta t$. Therefore, in MTL$^{<+\infty}_{+pt}$, we use the trace index (simulation step i) as the reference of time.

Definition 3.4.3 (MTL$^{<+\infty}_{+pt}$ Syntax) *Let AP be the set of predicates and $\mathcal{I}$ be any non-empty interval of $\mathbb{N}$, and $\overline{\mathcal{I}}$ be any non-empty interval of $\mathbb{N} \cup \{+\infty\}$. The set MTL$^{<+\infty}_{+pt}$ formulas is inductively defined as $\varphi ::= \top \mid p \mid \neg\varphi \mid \psi \vee \varphi \mid \psi U_{\mathcal{I}} \varphi \mid \psi S_{\overline{\mathcal{I}}} \varphi$ where $p \in AP$ and $\top$ stands for* true.

Note that in MTL$^{<+\infty}_{+pt}$ syntax, we use the number of samples to represent the time interval constraints of temporal operators. For example, assume that $\Delta t = 0.1$, then the MTL formula $\Diamond_{[0,0.5]} a$ where the timing constraints are over time is instead represented by $\Diamond_{[0,5]} a$ in MTL$^{<+\infty}_{+pt}$. All bounded future temporal operators can be syntactically defined using

Until ($U_{\mathcal{I}}$), where $\bigcirc$ (Next), $\Diamond$ (Eventually), and $\Box$ (Always) are defined as $\bigcirc\varphi \equiv \top U_{[1,1]}\varphi$, $\Diamond_{\mathcal{I}}\varphi \equiv \top U_{\mathcal{I}}\varphi$, and $\Box_{\mathcal{I}}\varphi \equiv \neg\Diamond_{\mathcal{I}}\neg\varphi$ respectively. The intuitive meaning of the $\psi U_{[a,b]}\varphi$ operator at sampling time i is a follows: ψ has to hold at least until φ becomes true within the time interval of $[i + a, i + b]$ in the future. Similarly, all other bounded/unbounded past temporal operators can be defined using Since ($S_{\overline{\mathcal{I}}}$), where $\odot$ (Previous), $\diamondsuit$ (Eventually in the past), and $\boxdot$ (Always in the past) are defined as $\odot\varphi \equiv \top S_{[1,1]}\varphi$, $\diamondsuit_{\overline{\mathcal{I}}}\varphi \equiv \top S_{\overline{\mathcal{I}}}\varphi$, and $\boxdot_{\overline{\mathcal{I}}}\varphi \equiv \neg\diamondsuit_{\overline{\mathcal{I}}}\neg\varphi$. The intuitive meaning of the $\psi S_{[a,b]}\varphi$ operator at sampling time i is as follows: since φ becomes true within the interval $[i - b, i - a]$ in the past, ψ must hold till now (current time i). We define the robust valuation of an $\text{MTL}_{+pt}^{<+\infty}$ formula φ over a trace $\tilde{\mathbf{y}}$ as follows [47].

Definition 3.4.4 ($\text{MTL}_{+pt}^{<+\infty}$ Robustness Semantics) *Let $\tilde{\mathbf{y}}$ be a sampled output as defined in Section 3.1.2, and $\mathcal{O}$ be an observation map $\mathcal{O} : AP \to P(Y)$, then the robust semantics of any formula $\varphi \in MTL_{+pt}^{<+\infty}$ with respect to $\tilde{\mathbf{y}}$ is recursively defined as:*

$$[\![\top]\!](\tilde{\mathbf{y}}, i) := +\infty$$

$$[\![p]\!](\tilde{\mathbf{y}}, i) := \mathbf{Dist_d}(\tilde{\mathbf{y}}_i, \mathcal{O}(p)) \text{ \textit{see Definition 3.4.1}}$$

$$[\![\neg\varphi]\!](\tilde{\mathbf{y}}, i) := -[\![\varphi]\!](\tilde{\mathbf{y}}, i)$$

$$[\![\psi \vee \varphi]\!](\tilde{\mathbf{y}}, i) := [\![\psi]\!](\tilde{\mathbf{y}}, i) \sqcup [\![\varphi]\!](\tilde{\mathbf{y}}, i)$$

$$[\![\psi U_{[l,u]}\varphi]\!](\tilde{\mathbf{y}}, i) := \bigsqcup_{j=i+l}^{i+u} \left([\![\varphi]\!](\tilde{\mathbf{y}}, j) \sqcap \bigsqcap_{k=i}^{j-1} [\![\psi]\!](\tilde{\mathbf{y}}, k)\right)$$

$$[\![\psi S_{[l',u'\rangle}\varphi]\!](\tilde{\mathbf{y}}, i) := \bigsqcup_{j=max\{0,i-u'\}}^{i-l'} \left([\![\varphi]\!](\tilde{\mathbf{y}}, j) \sqcap \bigsqcap_{k=j+1}^{i} [\![\psi]\!](\tilde{\mathbf{y}}, k)\right)$$

where $\sqcup$ stands for max, $\sqcap$ stands for min, $p \in AP$, $l, u, l' \in \mathbb{N}$ and $u' \in \mathbb{N} \cup \{\infty\}$. Furthermore, the symbol $\rangle$ in $\mathcal{S}_{[l',u'\rangle}$ will be) when $u' = +\infty$ and] when $u' \neq +\infty$.

Chapter 4

SYSTEM INDEPENDENT SPECIFICATION DEBUGGING

In this chapter, I provide a specification analysis framework that would enable the debugging of specifications. The specification debugging algorithm identifies invalid and incorrect specifications. This method will then be applied to find critical issues in the system when we consider Request-Response requirements (Chapters 5, 6).

4.1 Related Works

Let us consider the model checking with respect to LTL formulas [30, 16]. It is possible that the model satisfies the specification but not in the intended way. This may hide actual problems in the model. These satisfactions are called *vacuous* satisfactions. Antecedent failure was the first problem that raised the vacuity as a serious issue in verification [20, 22].

Vacuity can be addressed with respect to a model [21, 14, 73] or without a model [54, 29]. A formula which has a subformula that does not affect the overall satisfaction of the formula is a vacuous formula. It has been proven in [54] that a specification φ is satisfied vacuously in all systems that satisfy it iff φ is equivalent to some mutations of it. In [29], they provide an algorithmic approach to detect vacuity and redundancy in LTL specifications. Vacuity with respect to testing was considered in [18], where they considered vacuity in model checking as a *strong vacuity*. In contrast, [18] defined *weak vacuity* for test suites that vacuously pass LTL monitors, e.g. [58]. Our work extends [29] and it is applied to a fragment of MITL. For the overview of MITL fragment, consider Section 3.3.1.

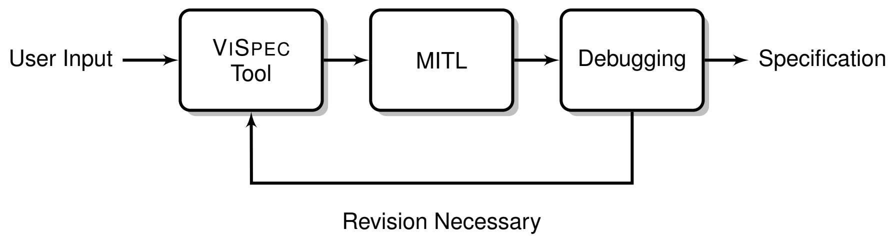

Figure 4.1: Specification Elicitation Framework

4.2 MITL Elicitation Framework

The framework for elicitation of MITL specifications is presented in [66]. Once a specification is developed using VISPEC [66], it is translated into STL (see Figure 4.1). Then, we create the corresponding MITL formula from STL. Next, the MITL specification is analyzed by the debugging algorithm which returns an alert to the user if the specification has inconsistency or correctness issues. The debugging process is explained in details in the next section.

To enable the debugging of specifications, we must first project the STL predicate expressions (functions) into atomic propositions with independent truth valuations. This is very important because the atomic propositions ($a \in AP$) in MITL are assumed to be independent of each other. However, when we project predicates to the atomic propositions, the dependency between the predicates restricts the possible combinations of truth valuations of the atomic propositions. This notion of predicate dependency is illustrated using the following example. Consider the real-valued signal $Speed$ in Figure 4.2. The boolean abstraction a (resp. b) over the $Speed$ signal is true when the $Speed$ is above 100 (resp. 80). The predicates a and b are related to each other because it is always the case that if $Speed > 100$ then also $Speed > 80$. In Figure 4.2, the boolean signals for predicates a and b are represented in black solid and dotted lines, respectively. It can be seen that solid and dotted lines are overlapping which shows the dependency between them. However, this dependency is not captured if we naively substitute each predicate with a unique atomic proposition. If we lose information about the intrinsic logical depen-

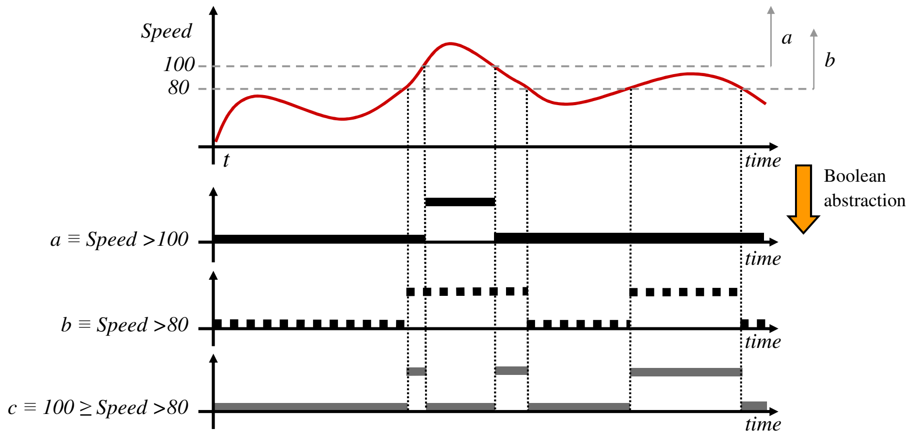

Figure 4.2: The Real-valued $Speed$ Signal and Its Three Boolean Abstractions: $a \equiv Speed > 100$ (Solid Black Line), $b \equiv Speed > 80$ (Dotted Line), and $c \equiv 100 \geq Speed > 80$ (Gray Line).

dency between a and b, then the debugging algorithm will not find possible specication issues. For analysis of STL formulas within our MITL debugging process, we must replace the original predicate with non-overlapping (mutual exclusive) predicates. For the example illustrated in Figure 4.2, we create a new atomic proposition c which corresponds to $100 \geq speed > 80$ and the corresponding boolean signal is represented in gray. In addition, we replace the atomic proposition b with the propositional formula $a \vee c$ since $speed > 80 \equiv (speed > 100 \vee 100 \geq speed > 80)$. Now, the dependency between $Speed > 100$ and $Speed > 80$ can be preserved because it is always the case that if a ($Speed > 100$), then $a \vee c$ ($Speed > 80$). It can be seen in Figure 4.2 that the signal b (dotted line) is the disjunction of the solid black (a) and gray (c) signals, where a and c cannot be simultaneously true.

The projection of STL to MITL with independent atomic propositions is conducted using a brute-force approach that runs through all the combinations of predicate expressions to find overlapping parts. The high level overview of Algorithm 1 is as follows: given the set of predicates $\Pi = \{\pi_1, ..., \pi_n\}$, the algorithm iteratively calls Algorithm 2 (DecPred) in order to identify predicates whose corresponding sets have non-empty intersections. For each predicate π_i, we assume there exists a corresponding set $\mathcal{S}_i$ such that $\mathcal{S}_i = \{x \mid x \in$

Algorithm 1 Generate Mutually Exclusive Predicates

Input: Set of predicates $\Pi = \{\pi_1, ..., \pi_n\}$

Output: Mutually exclusive predicates Ψ

```
1: Initially, update $\Pi$ with Disjunction of $\Psi$
2: termCond $\leftarrow$ 0;$\Delta \leftarrow \Pi$
3: while termCond = 0 do
4:     $\Psi \leftarrow$ DecPred($\Delta$)
5:     if $\Psi \neq \emptyset$ then
6:         $\Delta \leftarrow \Psi$
7:     else
8:         termCond $\leftarrow$ 1
9:         $\Psi \leftarrow \Delta$
10:    end if
11: end while
12: $\Pi \leftarrow$CreateDisjunction($\Pi, \Psi$)
13: return $\Pi$,$\Psi$
```

$\mathbb{R}^m, \pi_i(x) = \top\}$. The set $\mathcal{S}_i$ represents part of space $\mathbb{R}^m$ where predicate function π_i evaluates to $\top$. When no non-empty intersection is found, the Algorithm 1 terminates.

Algorithm 1 creates a temporary copy of Π in a new set Δ. Then in a while loop Algorithm 2 is called in Line 3 to find overlapping predicates. Algorithm DecPred checks all the combination of predicates in Δ until it finds overlapping sets (see Line 3). The DecPred partitions two overlapping predicates π_i, π_j into three mutually exclusive predicates $\overline{\pi}_{ij1}, \overline{\pi}_{ij2}, \overline{\pi}_{ij3}$ in Lines 4-6. Then π_i, π_j are removed from Δ in Line 7 and new predicates $\overline{\pi}_{ij1}, \overline{\pi}_{ij2}, \overline{\pi}_{ij3}$ are appended to Δ (Line 8). If no overlapping predicates are found, then DecPred returns $\emptyset$ in Line 13 and termCond gets value 1 in Line 7 of Algorithm 1. Now the

Algorithm 2 DecPred: Decompose Two Predicates

Input: Set of predicates $\Pi = \{\pi_1, ..., \pi_n\}$

Output: Set of updated predicates Π

```
1:  for $i = 1$ to size of $\Pi$ do
2:      for $j = i + 1$ to size of $\Pi$ do
3:          if $\mathcal{S}_i \cap \mathcal{S}_j \neq \emptyset$ then
4:              $\overline{\pi}_{ij1} \leftarrow \mathcal{S}_i \cap \mathcal{S}_j$
5:              $\overline{\pi}_{ij2} \leftarrow \mathcal{S}_i \setminus \mathcal{S}_j$
6:              $\overline{\pi}_{ij3} \leftarrow \mathcal{S}_j \setminus \mathcal{S}_i$
7:              Remove($\Pi, \{\pi_i, \pi_j\}$)
8:              Aappend($\Pi, \{\overline{\pi}_{ij1}, \overline{\pi}_{ij2}, \overline{\pi}_{ij3}\}$)
9:              return $\Pi$
10:         end if
11:     end for
12: end for
13: return $\emptyset$
```

while loop will terminate and Ψ contains all non-overlapping predicates. We must rewrite the predicates in Π with a disjunction operation on the new the predicates in Ψ. This operation takes place in Line 11 of Algorithm 1. Since the CreateDisjunction function is trivial, we omit its pseudo code. The runtime overhead of Algorithm 1 and the size of the resulting set Ψ can be exponential to $|\Pi| = n$, because we can have 2^n possible combinations of predicate evaluations.

4.3 Problem Formulation

In this section, we present the framework which checks the requirements for erroneous or incomplete MITL specifications without considering the system (system independent).

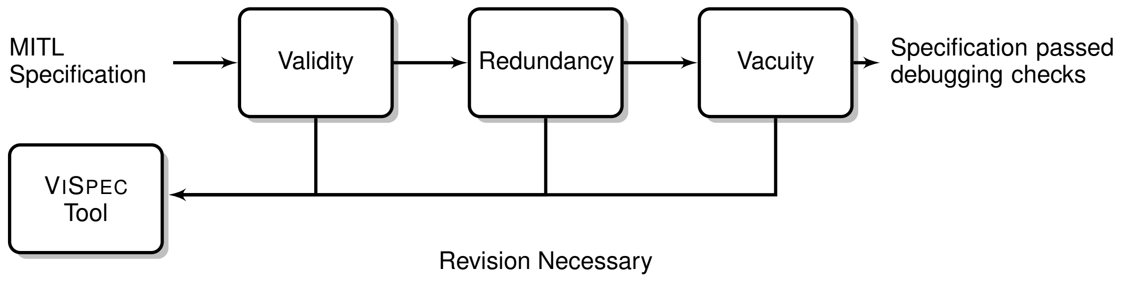

Figure 4.3: Specification Debugging Framework [36]

Then, in Chapter 5, we will provide the method to analyze reactive requirements with respect to system test traces to find more subtle errors in specifications or systems. This framework can help the users to detect the specification errors, where the requirement issues can be corrected before any test and verification process is initiated.

Problem 1 (System Independent MITL Analysis) *Given an MITL formula φ, find whether φ has any of the following logical issues:*

1. Validity: the specification is unsatisfiable or a tautology.
2. Redundancy: the formula has redundant conjuncts.
3. Vacuity: some subformulas do not contribute to the satisfiability of the formula.

In particular, issues 2 and 3 usually indicate some misunderstanding in the requirements. The overview of our proposed solution to Problem 1 is provided in Figure 4.3. This framework appeared in [36], and it can solve the specification correctness problem for VISPEC [66] requirements. The user of VISPEC can benefit from our feed-back and fix any reported issue.

4.4 MITL Specification Debugging

In the following, we present algorithms that can detect inconsistency and correctness issues in specifications. This will help the user in the elicitation of correct specifications.

Our specification debugging process conducts the following checks in this order: 1) Validity, 2) Redundancy, and 3) Vacuity. In brief, validity checking determines whether the specification is unsatisfiable or a tautology. Namely, if the specification is unsatisfiable no system can satisfy it and if it is a tautology every system can trivially satisfy it. For example, $p \vee \neg p$ is a tautology. If an MITL formula passes the validity checking, this means that the MITL is satisfiable but not a tautology.

Redundancy checking determines whether the specification has any redundant conjunct when the specification is a conjunction of MITL formulas. For example, in the specification $p \wedge \Box_{[0,10]} p$, the first conjunct is redundant. Sometimes redundancy is related to incomplete or erroneous requirements where the user may have wanted to specify something else. Therefore, the user should be notified.

Vacuity checking determines whether the specification has a subformula that does not affect on the satisfaction of the specification. For example, $\varphi = p \vee \Diamond_{[0,10]} p$ is vacuous since the first occurrence of p does not have any affect the satisfaction of φ. This is a logical issue because a part of the specification is overshadowed by the other components.

The debugging process is presented in Figure 4.3. The feedback (Revision Necessary) to the user is a textual description about the detail of each issue. First, given a specification, a validity check is conducted. If a formula does not pass the validity check then it means that there is a major problem in the specification and the formula is returned for revision. Therefore, redundancy and vacuity checks are not relevant at that point and the user is notified that the specification is either unsatisfiable or is a tautology. Similarly, if the specification is redundant it means that it has a conjunct that does not have any effect on the satisfaction of the specification and we return the redundant conjunct to the user for revision. Lastly, if the specification is vacuous it is returned with the issue for revision by the user. When vacuity is detected, we return to the user the simplified formula which is equivalent to the original MITL.

4.4.1 Redundancy Checking

Recall that a specification has a redundancy issue if one of its conjuncts can be removed without affecting the models of the specification. Before we formally present what redundant requirements are, we have to introduce some notation. We consider specification Φ as a conjunction of MITL subformulas (φ_j):

$$\Phi = \bigwedge_{j=1}^{k} \varphi_j \tag{4.1}$$

To simplify discussion, we will abuse notation and we will associate a conjunctive formula with the set of its conjuncts. That is:

$$\Phi = \{\varphi_j \mid j = 1, ..., k\} = \varphi_1 \cup \varphi_2 \cup \cdots \cup \varphi_k \tag{4.2}$$

Similarly, $\{\Phi \backslash \varphi_i\}$ represents the specification Φ where the conjunct φ_i is removed:

$$\{\Phi \backslash \varphi_i\} = \{\varphi_j \mid j = 1, ..., i-1, i+1, ..., k\} = \bigwedge_{j=1}^{i-1} \varphi_j \wedge \bigwedge_{j=i+1}^{k} \varphi_j \tag{4.3}$$

Therefore $\{\Phi \backslash \varphi_i\}$ represents a conjunctive formula. Redundancy in specifications can appear in practice due to the incremental additive approach that system engineers take in the development of specifications. Redundancy should be avoided in formal specification because it increases the overhead of the testing and verification processes. In addition, redundancy can be the result of incorrect translation from natural language requirements. In the following, we consider the redundancy removal algorithm provided in [29] for LTL formulas and we extend it to support MITL formulas.

Definition 4.4.1 (Redundancy of Specification) *A conjunct φ_i is redundant with respect to Φ if*

$$\bigwedge_{\psi \in \{\Phi \backslash \varphi_i\}} \psi \models \varphi_i$$

To reformulate, φ_i is redundant with respect to Φ if $\{\Phi \backslash \varphi_i\} \models \varphi_i$. For example, in $\Phi = \Diamond_{[0,10]}(p \wedge q) \wedge \Diamond_{[0,10]} p \wedge \Box_{[0,10]} q$, the conjunct $\Diamond_{[0,10]}(p \wedge q)$ is redundant with respect to $\Diamond_{[0,10]} p \wedge \Box_{[0,10]} q$ since $\Diamond_{[0,10]} p \wedge \Box_{[0,10]} q \models \Diamond_{[0,10]}(p \wedge q)$. In addition, $\Diamond_{[0,10]} p$ is redundant with respect to $\Diamond_{[0,10]}(p \wedge q) \wedge \Box_{[0,10]} q$ since $\Diamond_{[0,10]}(p \wedge q) \wedge \Box_{[0,10]} q \models \Diamond_{[0,10]} p$. This method can detect both the issues and report them to the user. Algorithm 3 finds redundant conjuncts in the conjunction operation of the following levels:

1. Conjunction as the root formula (top level).

2. Conjunction in the nested subformulas (lower levels).

In the top level, it provides the list of subformulas that are redundant with respect to the original MITL Φ. In the lower levels, if a specification has nested conjunctive subformulas ($\phi_i \in \Phi$), it will return the conjunctive subformula ϕ_i as well as its redundant conjunct $\psi_j \in \phi_i$. For example, if $\Phi = \Diamond_{[0,10]}(p \wedge \Box_{[0,10]} p)$ is checked by Algorithm 3, then it will return the pair of $(p, p \wedge \Box_{[0,10]} p)$ to represent that p is redundant in $p \wedge \Box_{[0,10]} p$. In Line 5, the pair of (ψ_j, ϕ_i) is interpreted as follows: ψ_j is redundant in ϕ_i.

4.4.2 Specification Vacuity Checking

Vacuity detection is used to ensure that all the subformulas of the specification contribute to the satisfaction of the specification. In other words, vacuity check enables the detection of irrelevant subformulas in the specifications [29]. For example, consider the STL specification $\phi_{stl} = \Diamond_{[0,10]}((speed > 100) \vee \Diamond_{[0,10]}(speed > 80))$. In this case, the subformula $(speed > 100)$ does not affect the satisfaction of the specification. This indicates that ϕ_{stl} is a vacuous specification. We need to create correct atomic propositions for the predicate expressions of ϕ_{stl} to be able to detect such vacuity issues in MITL formulas. If we naively replace the predicate expressions $speed > 100$ and $speed > 80$ with the atomic propositions a and b, respectively, then the resulting MITL formula

Algorithm 3 Redundancy Checking

Input: Φ ($MITL$ Specification)

Output: RL_φ (redundant conjuncts w.r.t conjunctions)

```
1: $RL_\varphi \leftarrow \emptyset$
2: for each conjunctive subformula $\phi_i \in \Phi$ do
3:     for each conjunct $\psi_j \in \phi_i$ do
4:         if $\{\phi_i \backslash \psi_j\} \models \psi_j$ then
5:             $RL_\varphi \leftarrow RL_\varphi \cup (\psi_j, \phi_i)$
6:         end if
7:     end for
8: end for
9: return $RL_\varphi$
```

will be $\phi_{mitl} = \Diamond_{[0,10]}(a \vee \Diamond_{[0,10]}b)$. However, ϕ_{mitl} is not vacuous. Therefore, we must extract non-overlapping predicates as explained in Section 4.2. The new specification $\phi'_{mitl} = \Diamond_{[0,10]}(a \vee \Diamond_{[0,10]}(a \vee c))$ where a corresponds to $speed > 100$ and c corresponds to $100 \geq speed > 80$ is the correct MITL formula corresponding to ϕ_{stl}, and it is vacuous. In the following, we provide the definition of MITL vacuity with respect to a signal:

Definition 4.4.2 (MITL Vacuity with respect to timed trace) *Given a timed trace μ and an MITL formula φ. A subformula ψ of φ does not affect the satisfiability of φ with respect to μ if and only if ψ can be replaced with any subformula θ without changing the satisfiability of φ on μ. A specification φ is satisfied vacuously by μ, denoted by $\mu \models_V \varphi$, if there exists a subformula ψ which does not affect the satisfiability of φ on μ.*

In the following, we extend the framework presented in [29] to support MITL specifications. Let φ be a formula in NNF where only predicates can be in the negated form. A *literal* is defined as a predicate or its negation. For a formula φ, the set of literals of

Algorithm 4 Vacuity Checking

Input: Φ ($MITL$ Specification)

Output: VL_{φ} (vacuous formulas)

```
1: $VL_{\varphi} \leftarrow \emptyset$
2: for each formula $\varphi_i \in \Phi$ do
3:     for each $l \in litOccur(\varphi_i)$ do
4:         if $\Phi \models \varphi_i[l \leftarrow \perp]$ then
5:             $VL_{\varphi} \leftarrow VL_{\varphi} \cup \{\Phi \backslash \varphi_i\} \wedge \varphi_i[l \leftarrow \perp]$
6:         end if
7:     end for
8: end for
9: return $VL_{\varphi}$
```

φ is denoted by *literal*(φ) and contains all the literals appearing in φ. For example, if $\varphi = (\neg p \wedge q) \vee \Diamond_{[0,10]} p \vee \Box_{[0,10]} q$, then *literal*($\varphi$) $= \{\neg p, q, p\}$. Literal occurrences, denoted by *litOccur*(φ), is a multi-set of literals appearing in some order in φ, e.g., by traversal of the parse tree. For the given example *litOccur*(φ) $= \{\neg p, q, p, q\}$. For each $l \in$ *litOccur*(φ), we create the mutation of φ by substituting the occurrence of l with $\perp$. We denote the mutated formula as $\varphi[l \leftarrow \perp]$.

Definition 4.4.3 ($MITL$ Vacuity w.r.t. literal occurrence) *Given a timed trace μ and an MITL formula φ in NNF. Specification φ is vacuously satisfied by μ if there exists a literal occurrence $l \in$ litOccur(φ) such that μ satisfies the mutated formula $\varphi[l \leftarrow \perp]$. Formally, $\mu \models_V \varphi$ if $\exists l \in$ litOccur(φ) s.t. $\mu \models \varphi[l \leftarrow \perp]$.*

Theorem 4.4.1 ($MITL$ Vacuity with respect to Specification) *Assume that the specification Φ is a conjunction of MITL formulas. If $\exists \varphi_i \in \Phi$ and $\exists l \in$ litOccur(φ_i), such that $\Phi \models \varphi_i[l \leftarrow \perp]$, then Φ is inherently vacuous.*

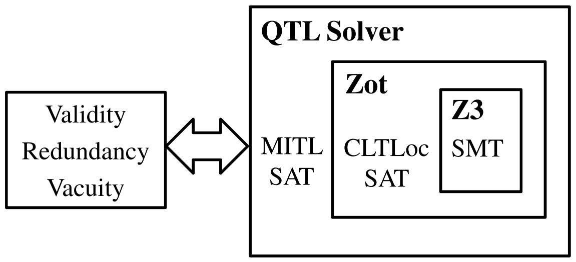

Figure 4.4: The MITL SAT Solver from [25] Is Used for Debugging Specifications.

A specification Φ is *inherently vacuous* if it is equivalent to its simplified mutation, which means that Φ is vacuous independent of any signal or system. Inherent vacuity of LTL formulas is addressed in [54, 29]. The proof of Theorem 4.4.1 is straightforward modification of the proofs given in [29, 73]. We provide the proof in Appendix A. When we do not have a root-level conjunction in the specification ($\Phi = \varphi$[1]), we check the vacuity of the formula with respect to itself. In other words, we check whether the specification satisfies its mutation ($\varphi \models \varphi[l \leftarrow \perp]$ or $\varphi \models_V \varphi$). Technically, Algorithm 4 as presented, returns a list of all the mutated formulas that are equivalent to the original MITL.

4.5 Experiments

All the thee level correctness analysis of MITL specifications need satisfiability checking as the underlying tool [26]. In validity checking, we simply check whether the specification and its negation are satisfiable. In general, in order to check whether $\varphi \models \psi$, we should check whether $\varphi \rightarrow \psi$ is a tautology, that is $\forall \mu, \mu \models \varphi \rightarrow \psi$. This can be verified by checking whether $\neg(\varphi \rightarrow \psi)$ is unsatisfiable. Recall that $\varphi \rightarrow \psi$ is equivalent to $\neg\varphi \vee \psi$. So we have to check whether $\varphi \wedge \neg\psi$ is unsatisfiable to conclude that $\varphi \models \psi$. We use the above reasoning for redundancy checking as well as for vacuity checking. For redundancy checking of conjuncts at the root level, $\{\Phi \backslash \varphi_i\} \wedge \neg\varphi_i$ should be unsatisfiable, in order to conclude that $\{\Phi \backslash \varphi_i\} \models \varphi_i$. For vacuity checking, $\Phi \wedge \neg(\varphi_i[l \leftarrow \perp])$ should be unsatisfiable,

[1]In this case, we assume $\{\Phi \backslash \varphi_i\} \equiv \top$ in Line 5 of the Algorithm 4.

in order to prove that $\Phi \models \varphi_i[l \leftarrow \perp]$.

4.5.1 MITL Satisfiability

The satisfiability problem of MITL is EXPSPACE-complete [8]. In order to check whether an MITL formula is satisfiable we use two publicly available tools: qtlsolver[2] and zot[3]. The qtlsolver that we used translates MITL formulas into CLTL-over-clocks [25, 26]. Constraint LTL (CLTL) is an extension of LTL where predicates are allowed to be assertions on the values of non-Boolean variables [32]. That is, in CLTL, we are allowed to define predicates using relational operators for variables over domains like $\mathbb{N}$ and $\mathbb{Z}$. Although satisfiability of CLTL in general is not decidable, some variants of it are decidable [32].

CLTLoc (CLTL-over-clocks) is a variant of CLTL where the clock variables are the only arithmetic variables that are considered in the atomic constraints. It has been proven in [24] that CLTLoc is equivalent to timed automata [30]. Moreover, it can be polynomially reduced to decidable Satisfiability Modulo Theories which are solvable by many SMT solvers such as Z3[4]. The satisfiability of CLTLoc is PSPACE-complete [26] and the translation from MITL to CLTLoc in the worst case can be exponential [25]. Some restrictions must be imposed on the MITL formulas in order to use the qtlsolver [25]. That is, the lower bound and upper bound for the intervals of MITL formulas should be integer values and intervals are left/right closed. Therefore, we expect the values to be integer when we analyse MITL formulas. The high level architecture of the MITL SAT solver, which we use to check the three issues, is provided in Figure 4.4.

[2]qtlsolver: A solver for checking satisfiability of Quantitative / Metric Interval Temporal Logic (MITL/QTL) over Reals. Available from `https://code.google.com/p/qtlsolver/`

[3]The zot bounded model/satisfiability checker. Available from `https://code.google.com/p/zot/`

[4]Microsoft Research, Z3: An efficient SMT solver. Available from `http://research.microsoft.com/en-us/um/redmond/projects/z3/`

4.5.2 Specification Debugging Results

We utilize the debugging algorithm on a set of specifications developed as part of a usability study for the evaluation of the VISPEC tool [66]. The usability study was conducted on two groups:

1. Group A: These are users who declared that they have little to no experience in working with requirements. The Group A cohort consists of twenty subjects from the academic community at Arizona State University. Most of the subjects have an engineering background.

2. Group B: These are users who declared that they have experience working with system requirements. Note that they do not necessarily have experience in writing requirements using formal logics. The Group B subject cohort was comprised of ten subjects from industry in the Phoenix metro area.

Each subject received a task list to complete. The list contained ten tasks related to automotive system specifications. Each task asked the subject to formalize a natural language specification through VISPEC and generate an STL specification. The task list is presented in Table 4.1. A detailed report on the accuracy of the users response to each natural language requirement is provided in [66]. Note that the specifications were preprocessed and transformed from the original STL formulas to MITL in order to run the debugging algorithm. For example, specification ϕ_3 in Table 4.2 originally in STL was $\phi_{3_{STL}} = \Diamond_{[0,40]}(((speed > 80) \rightarrow \Diamond_{[0,20]}(rpm > 4000)) \wedge \Box_{[0,30]}(speed > 100))$. The STL predicate expressions $(speed > 80), (rpm > 4000), (speed > 100)$ are mapped into atomic propositions with non-overlapping predicates (Boolean functions) p_1, p_2, p_3. The predicates p_1, p_2, p_3 correspond to the following STL representations: $p_1 \equiv speed > 100$, $p_2 \equiv rpm > 4000$, and $p_3 \equiv 100 \geq speed > 80$. In Table 4.2, we present common issues with the elicited specifications that our debugging algorithm would have detected and alerted each subject if the tool were available at the time of the study. Note that validity, redundancy and vacuity issues are present in the specifications listed. It should be noted that for specification ϕ_3, although finding the error takes a significant amount of time, our algorithm can be used off-line.

In Figure 4.5, we present the runtime overhead of the three stage debugging algorithm over specifications collected in the usability study. In the first stage, 87 specifications go through validity checking. Five specifications fail the test and therefore they are immediately returned to the user. As a result, 82 specifications go through redundancy checking of conjunction in the root level [5], where 9 fail the test. Lastly, 73 specifications go through vacuity checking where 5 specifications have vacuity issues. The remaining 68 specifications passed the tests. Note that in the figure, two outlier data points are omitted from the vacuity sub-figure for presentation purposes. The two cases were timed at 39,618sec and 17,421sec. In both cases, the runtime overhead was mainly because the zot software took hours to determine that the modified specification is unsatisfiable (both specifications were vacuous). The overall runtime of ϕ_3 in Table 4.2 is 39,645sec which includes the runtime of validity and redundancy checking. The runtime overhead of vacuity checking of ϕ_3 can be reduced by half because, originally, in vacuity checking we run MITL satisfiability checking for all literal occurrences. In particular, ϕ_3 has four literal occurrences where for two cases the zot took more than 19,500sec to determine that the modified specification is unsatisfiable. We can provide an option for early detection: stop and report as soon as an issue is found (the first unsatisfiability).

The circles in Figure 4.5 represent the timing performance in each test categorized by the number of literal occurrences and temporal operators. The asterisks represent the mean values and the dashed line is the linear interpolation between them. In general, we observe

[5]In these experiments, we did not consider conjunctions in the lower level subformulas for redundancy checking.

an increase in the average computation time as the number of literal occurrences and temporal operators increases. All the experimental results in Section 4.5 were performed on an Intel Xeon X5647 (2.993GHz) with 12 GB RAM.

Table 4.1: Task List with Automotive System Specifications Presented in Natural Language

Task	Natural Language Specification
1. Safety	In the first 40 seconds, vehicle speed should always be less than 160.
2. Reachability	In the first 30 seconds, vehicle speed should go over 120.
3. Stabilization	At some point in time in the first 30 seconds, vehicle speed will go over 100 and stay above for 20 seconds.
4. Oscillation	At every point in time in the first 40 seconds, vehicle speed will go over 100 in the next 10 seconds.
5. Oscillation	It is not the case that, for up to 40 seconds, the vehicle speed will go over 100 in every 10 second period.
6. Implication	If, within 40 seconds, vehicle speed is above 100 then within 30 seconds from time 0, engine speed should be over 3000.
7. Request-Response	If, at some point in time in the first 40 seconds, vehicle speed goes over 80 then from that point on, for the next 30 seconds, engine speed should be over 4000.
8. Conjunction	In the first 40 seconds, vehicle speed should be less than 100 and engine speed should be under 4000.
9. Non-strict sequencing	At some point in time in the first 40 seconds, vehicle speed should go over 80 and then from that point on, for the next 30 seconds, engine speed should be over 4000.
10. Long Sequence	If, at some point in time in the first 40 seconds, vehicle speed goes over 80 then from that point on, if within the next 20 seconds the engine speed goes over 4000, then, for the next 30 seconds, the vehicle speed should be over 100.

Table 4.2: Incorrect Specifications from the Usability Study In [66], Error Reported to the User by the Debugging Algorithm, and Algorithm Runtime. Formulas Have Been Translated from STL to MITL.

ϕ	#	MITL Specification created by VISPEC users	Reporting the errors	Sec.
ϕ_1	3	$\Diamond_{[0,30]} p_1 \wedge \Diamond_{[0,20]} p_1$	$\Diamond_{[0,30]} p_1$ is redundant	14
ϕ_2	3	$\Diamond_{[0,30]}(p_1 \rightarrow \Box_{[0,20]} p_1)$	φ is a tautology	7
ϕ_3	10	$\Diamond_{[0,40]}(((p_1 \vee p_3) \rightarrow \Diamond_{[0,20]} p_2) \wedge \Box_{[0,30]} p_1)$	φ is vacuous: $\varphi \models \varphi[p_3 \leftarrow \perp]$	39645
ϕ_4	4	$\Box_{[0,40]} p_1 \wedge \Box_{[0,40]} \Diamond_{[0,10]} p_1$	$\Box_{[0,40]} \Diamond_{[0,10]} p_1$ is redundant	29
ϕ_5	10	$\Diamond_{[0,40]}(p_1 \vee p_3) \wedge \Diamond_{[0,40]} p_2 \wedge \Diamond_{[0,40]} \Box_{[0,30]} p_1$	$\Diamond_{[0,40]}(p_1 \vee p_3)$ is redundant	126

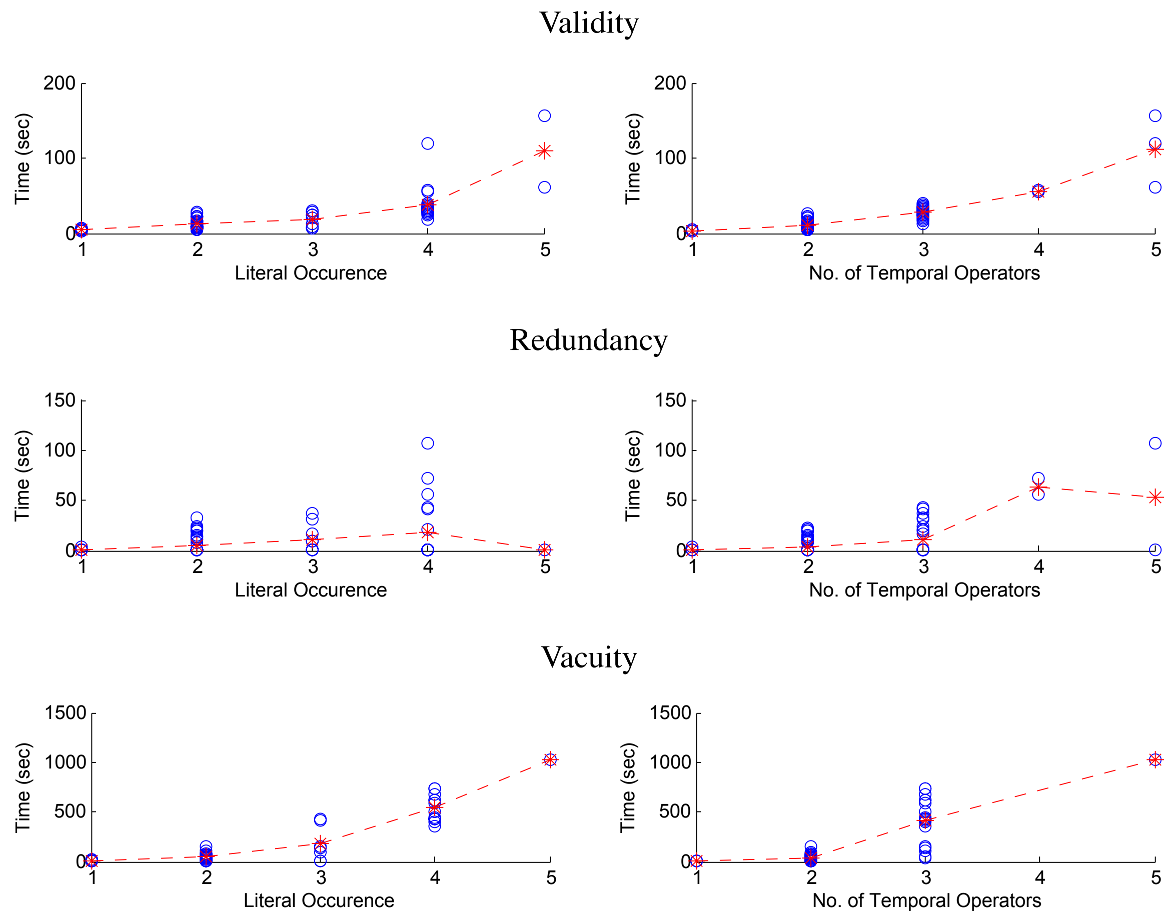

Figure 4.5: Runtime Overhead of the Three Stages of the Debugging Algorithm over User-submitted Specifications. Timing Results Are Presented over the Number of Literal Occurrences and the Number of Temporal Operators.

4.5.3 LTL Satisfiability

In the previous section, we mentioned that MITL satisfiability problem is a computationally hard problem. However, in practice, we know that LTL satisfiability is solvable faster than MITL satisfiability [76]. In this section, we consider how we can use the satisfiability of LTL formulas to decide about the satisfiability of MITL formulas. Consider the following fragments of MITL and LTL in NNF:

MITL($\Box$): $\varphi ::= \top \mid \bot \mid p \mid \neg p \mid \varphi_1 \wedge \varphi_2 \mid \varphi_1 \vee \varphi_2 \mid \Box_{\mathcal{I}}\varphi_1$

MITL($\Diamond$): $\varphi ::= \top \mid \bot \mid p \mid \neg p \mid \varphi_1 \wedge \varphi_2 \mid \varphi_1 \vee \varphi_2 \mid \Diamond_{\mathcal{I}}\varphi_1$

LTL($\Box$): $\varphi ::= \top \mid \bot \mid p \mid \neg p \mid \varphi_1 \wedge \varphi_2 \mid \varphi_1 \vee \varphi_2 \mid \Box\varphi_1$

LTL($\Diamond$): $\varphi ::= \top \mid \bot \mid p \mid \neg p \mid \varphi_1 \wedge \varphi_2 \mid \varphi_1 \vee \varphi_2 \mid \Diamond\varphi_1$

In Appendix A, we prove that the satisfaction of a formula $\phi_M \in$ MITL($\Diamond$) in NNF is related to the satisfaction of an LTL version of ϕ_M called $\phi_L \in$ LTL($\Diamond$) where ϕ_L is identical to ϕ_M except that every interval $\mathcal{I}$ in ϕ_M is removed. For example, if $\phi_M = \Diamond_{[0,10]}(p \wedge q) \wedge \Diamond_{[0,10]}p$ then $\phi_L = \Diamond(p \wedge q) \wedge \Diamond p$. In essence, if ϕ_M is satisfiable, then ϕ_L is also satisfiable. Therefore, if ϕ_L is unsatisfiable, then ϕ_M is also unsatisfiable.

For the always ($\Box$) operator, satisfiability is the dual of the eventually operator ($\Diamond$). Assume that $\phi'_M \in$ MITL($\Box$) contains only the $\Box$ operator and $\phi'_L \in$ LTL($\Box$) is the LTL version of ϕ'_M. If ϕ'_L is satisfiable, then ϕ'_M will also be satisfiable.

Based on the above discussion, if the specification that we intend to test/debug belongs to either category (fragment) of MITL($\Diamond$) or MITL($\Box$), then we can check the satisfiability of its LTL version (ϕ_L) and decide according to the following:

Theorem 4.5.1 *For any formula $\phi_M \in$ MITL($\Diamond$) and $\phi'_M \in$ MITL($\Box$) then*

If $\phi_L \in$ LTL($\Diamond$) is unsatisfiable, then ϕ_M is unsatisfiable.

If $\phi'_L \in$ LTL($\Box$) is satisfiable, then ϕ'_M is satisfiable.

In these two cases, we do not need to run MITL SAT, if otherwise, we must apply MITL SAT which means that we wasted effort by checking LTL SAT. However, since the runtime of LTL SAT is negligible, it will not drastically reduce the performance. As a result, LTL satisfiability checking is useful for validity testing. For redundancy checks, it may also be useful. For example, if we have a formula $\phi = \Diamond_{[0,10]}p \wedge \Box_{[0,20]}p$ we should check the satisfiability of $\phi' = \Box_{[0,10]}\neg p \wedge \Box_{[0,20]}p$ and $\phi'' = \Diamond_{[0,10]}p \wedge \Diamond_{[0,20]}\neg p$ for redundancy. Although the original formula ϕ does not belong to either MITL($\Diamond$) or MITL($\Box$), its modified NNF version will fit in these fragments and we may benefit by the usually faster LTL satisfiability for ϕ' and/or ϕ''. For vacuity checking, we can use LTL satisfiability if after manipulating/simplifying the original specification and creating the NNF version, we can categorize the resulting formula into the MITL($\Diamond$) or the MITL($\Box$) fragments (see Table 4.3).

We can check LTL satisfiability of the modified MITL specifications using existing methods and tools [88]. In our case, we used the NuSMV[6] tool with a similar encoding of LTL formulas as in [88]. In Table 4.3, we compare the runtime overhead of MITL and LTL satisfiability checking. For the results of the usability study in [66], we conduct validity and vacuity checking with the LTL satisfiability solver. We remark that in our results in Table 4.3, all the formulas belong to the MITL($\Box$) fragment. Since we did not find any MITL($\Diamond$) formula in our experiments where its LTL version is not satisfiable, we did not provide MITL($\Diamond$) formulas in Table 4.3.

The first column of Table 4.3 provides the debugging test phase where we used the satisfiability checkers. The second column represents the MITL formulas that we tested using the SAT solver. We omit the LTL formulas from Table 4.3, since they are identical to MITL but do not contain timing intervals. The atomic propositions p_1, p_2, p_3, p_4, p_5 of the MITL formulas in Table 4.3 correspond to the following STL predicates: $p_1 \equiv speed > 100$, $p_2 \equiv rpm > 4000$, $p_3 \equiv 100 \geq speed > 80$, $p_4 \equiv rpm > 3000$, and $p_5 \equiv speed > 80$. The third and fourth columns represent the runtime overhead of satisfiability checking for MITL specifications and their corresponding LTL version. The last column represents the speedup of the LTL approach over the MITL approach. It can be seen that the LTL SAT solver (NuSMV) is about 30-300 times faster than the MITL SAT solver (zot). These results confirm that, when applicable, LTL SAT solvers outperform MITL SAT solvers in checking vacuity and validity issues in specifications. As a result, it is worth running LTL SAT before MITL SAT when it is possible.

[6] NuSMV Version 2.6.0. Available from `http://nusmv.fbk.eu/`

Table 4.3: Comparing the Runtime Overhead of MITL Satisfiability and LTL Satisfiability (in Seconds) for Some of the Specifications from VISPEC's Usability Study.

Phase	MITL Specification	MITL	LTL	MITL/LTL
Validity	$\square_{[0,40]}(p_1 \rightarrow \square_{[0,10]}(p_1))$	4.154	0.047	88
Validity	$\square_{[0,30]}(\neg p_1) \vee \square_{[0,20]}(\neg p_1)$	3.418	0.0538	63
Validity	$\square_{[0,40]}((\neg p_1 \wedge \neg p_3) \vee \square_{[0,20]}\neg p_2 \vee \square_{[0,30]}p_1))$	10.85	0.045	240
Validity	$\square_{[0,40]}((p_1 \vee p_3) \rightarrow \square_{[0,20]}(p_2 \rightarrow \square_{[0,30]}p_1))$	15.406	0.0463	333
Vacuity	$\square_{[0,40]}(p_1)$	1.71	0.0473	36
Vacuity	$\square_{[0,40]}(p_1 \wedge \square_{[0,10]}(p_1))$	3.727	0.044	84
Vacuity	$\square_{[0,40]}p_1 \wedge \square_{[0,30]}(p_4)$	5.77	0.0456	126
Vacuity	$\square_{[0,40]}p_5 \wedge \square_{[0,70]}(p_5)$	8.599	0.044	194

4.6 Conclusions and Future Work

In this chapter, we have presented a specification debugging framework that helps expert and non-expert users to produce correct formal specifications. The debugging algorithm enables the detection of logical inconsistencies in MITL specifications. Our algorithm improves the elicitation process by providing feedback to the users on validity, redundancy and vacuity issues. The specification debugging framework will be integrated in the VISPEC tool to simplify MITL specification development for verification of CPS.

Chapter 5

SYSTEM DEPENDENT VACUITY CHECKING

In this chapter, we provide a new definition of vacuity with respect to real-time signals. We devise algorithms to detect specification issues with respect to signals which helps the CPS developers find more in depth specification issues during system testing.

5.1 Related Works

In this chapter, we mainly adopt the method provide in [18] for CPS. For the overview of vacuity issues in the model checking, consider Section 4.1. For each implication subformula ($\phi \rightarrow \psi$), the left operand (ϕ) is the precondition (antecedent) of the implication. In the previous chapter, we addressed specification vacuity without considering the system. However, in many cases specification vacuity depends on the system. For example, consider the LTL specification $\varphi = \Box(req \rightarrow \Diamond ack)$. The specification φ does not have an inherent vacuity issue [54]. However, if *req* never happens in any of the behaviors of the system, then the specification φ is vacuously satisfied on this specific system. In this example, *ack* does not affect the satisfaction of φ in any system with no *request* [18]. As a result, it has been argued that it is important to add vacuity detection in the model checking process [73]. We encounter the same issue when we test signals and systems with respect to Request-Response STL/MITL specifications. We refer to signals that do not satisfy the antecedent (precondition) of the subformula as vacuous signals. The following is the formal definition of the signal vacuity problem.

Problem 2 (Signal Vacuity and Coverage Checking) *Given an MITL formula φ, and signal* $\mathbf{y}$*, check whether* $\mathbf{y}$ *satisfies a mutation of φ.*

5.2 Vacuous Signals

Consider the MITL specification $\varphi = \Box_{[0,5]}(req \rightarrow \Diamond_{[0,10]}ack)$. This formula will pass the MITL Specification Debugging method presented in Section 4.4. However, any signal $\mathbf{y}$ that does not satisfy req at any point in time during the test will vacuously satisfy φ. We refer to timed traces that do not satisfy the antecedent (precondition) of the subformula as vacuous timed traces. Similarly, these issues follow for STL formulas over signals as well. Consider Task 6 in Table 4.1 with the specification $\psi = \Diamond_{[0,40]}(speed > 100) \rightarrow \Box_{[0,30]}(rpm > 3000)$. Any real-time signal $\mathbf{y}$ that does not satisfy $\Diamond_{[0,40]}(speed > 100)$ will vacuously satisfy ψ. Finding such signals is important in testing and monitoring, since if a signal $\mathbf{y}$ does not satisfy the precondition of an STL/MITL specification, then there is no point in considering $\mathbf{y}$ as a useful test.

Definition 5.2.1 (Vacuous Signal) *Given an MITL (STL) formula φ, a timed trace μ (signal s) is vacuous if it satisfies the Antecedent Failure mutation of φ.*

Antecedent Failure is one of the main sources of vacuity. Antecedent Failure occurs in a Request-Response specification such as $\varphi_{RR} = \Box_{[0,5]}(req \Rightarrow \Diamond_{[0,10]}ack)$. We provide a formula mutation that can detect signal vacuity in Request-Response specifications [62]:

Definition 5.2.2 (Request-Response MITL) *A Request-Response MITL formula φ_{RR} is an MITL formula that has one or more implication ($\rightarrow$) operations in positive polarity (without any negation). In addition, for each implication operation the consequent must have a temporal operator at the top-level.*

In the Request-Response (RR) specifications, we define sequential events in a specific order (by using the implication operator). Many practical specification patterns based on the Request-Response format are provided for system properties [44, 70]. Therefore, we can define a chain of events that the system must respond/react to. In an RR-specification such as $\varphi_{RR} = \Box_{[0,5]}(req \rightarrow \Diamond_{[0,10]}ack)$, the temporal operator for the consequent $\Diamond_{[0,10]}ack$ is necessary, unless the system does not have any time to acknowledge the *req*. For any trace μ in which *req* never happens, we can substitute *ack* by any formula and the specification is still satisfied by μ. Therefore, if the antecedent is failed by a trace μ, then φ_{RR} is vacuously satisfied by μ.

For each implication subformula $(\varphi \rightarrow \psi)$, the left operand (φ) is the precondition (antecedent) of the implication. An antecedent failure mutation is a new formula that is created with the assertion that the precondition (φ) never happens. Note that RR-specifications should not be translated into NNF For each precondition φ, we create an antecedent failure mutation $\Box_{\mathcal{I}_\varphi}(\neg\varphi)$ where $\mathcal{I}_\varphi$ is called the *effective interval* of φ.

Definition 5.2.3 (Effective Interval) *The effective interval of a subformula is the time interval when the subformula can have an impact on the truth value of the whole MITL(STL) specification.*

Each subformula is evaluated only in the time window that is provided by the *effective interval*. For example, for the MITL specification $\varphi \wedge \psi$, the effective interval for both φ and ψ is [0,0], because φ and ψ can change the value of $\varphi \wedge \psi$ only within the interval [0,0]. Similarly, for the MITL specification $\Box_{[0,10]}\varphi$, the effective interval of φ is [0,10], since the truth value of φ is observed in the time window of [0,10] for evaluating $\Box_{[0,10]}\varphi$. The effective interval is important for the creation of an accurate antecedent failure mutation. This is because the antecedent can affect the truth value of the MITL formula only if it is evaluated in the effective interval. The effective interval is like a timing window to make the antecedent observable for an outside observer the way it is observed by the MITL specification.

The effective interval of MITL formulas can be computed recursively using Algorithm 5. To run Algorithm 5, we must process the MITL formula parse tree[1]. The algorithm must be initialized with the interval of [0,0] for the top node of the MITL formula, namely, EIU(φ,[0,0]). This is because, according to the semantics of MITL, the value of the whole MITL formula is only important at time zero. In Line 8 of Algorithm 5, the operator $\oplus$ is used to add two intervals as follows:

Definition 5.2.4 ($\oplus$) *Given intervals $\mathcal{I} = [l, u]$ and $\mathcal{I}' = [l', u']$, we define $\mathcal{I}'' \leftarrow \mathcal{I} \oplus \mathcal{I}'$ where $\mathcal{I}'' = [l'', u'']$ such that $l'' = l + l'$ and $u'' = u + u'$. If u or u' is $+\infty$ then $\mathcal{I}'' = [l'', +\infty)$*

If either $\mathcal{I}$ or $\mathcal{I}'$ is left open (resp. right open), then $\mathcal{I}''$ will be left open (resp. right open)[2]. In Line 1 of Algorithm 5, the input interval $\mathcal{I}$ is assigned to the effective interval of φ, namely $\varphi.EI$. If the top operation of φ is a propositional operation ($\neg, \vee, \wedge, \rightarrow$) then the $\mathcal{I}$ will be propagated to subformulas of φ (see Lines 2-6). If the top operation of φ is a temporal operator ($\Box_{\mathcal{I}'}, \Diamond_{\mathcal{I}'}$), then the effective interval is modified according to Definition 5.2.4 and interval $\mathcal{I}'' \leftarrow \mathcal{I} \oplus \mathcal{I}'$ is propagated to the subformulas of φ.

For example, assume that the MITL specification is $\varphi_{RR} = \Box_{[1,2]}(\Diamond_{[3,5]}b \rightarrow (\Box_{[4,6]}(c \rightarrow d)))$. The specification φ has two antecedents, $\alpha_1 = \Diamond_{[3,5]}b$ and $\alpha_2 = c$. The effective intervals of α_1 and α_2 are $\mathcal{I}_{\alpha_1} = [0, 0] \oplus [1, 2] = [1, 2]$ and $\mathcal{I}_{\alpha_2} = [0, 0] \oplus [1, 2] \oplus [4, 6] = [5, 8]$, respectively. As a result, the corresponding antecedent failure mutations are $\Box_{[1,2]}(\neg\Diamond_{[3,5]}b)$ and $\Box_{[5,8]}(\neg c)$, respectively. Algorithm 6 returns the list of antecedent failures AF_φ, namely all the implication subformulas that are vacuously satisfied by $\mathbf{y}$. If the AF_φ list is empty,

[1]We assume that the MITL specification is saved in a binary tree data structure where each node is a formula with the left/right child as the left/right corresponding subformula. of φ. In addition, we assume that the nodes of φ's tree contain a field called EI where we annotate the effective interval of φ in EI, namely, $\varphi.EI \leftarrow \mathcal{I}_\varphi$.

[2]Although we assume in Definition 3.3.2 that intervals are right-closed, Algorithm 5 can be applied to right-open intervals as well.

Algorithm 5 Effective Interval Update EIU(φ,$\mathcal{I}$)

Input: φ (Parse Tree of the MITL formula), $\mathcal{I}$ (Effective Interval)

Output: φ (Updated formula with subformulas annotated with effective intervals)

```
 1: $\varphi.EI \leftarrow \mathcal{I}$
 2: if $\varphi \equiv \neg\varphi_m$ then
 3:     EIU($\varphi_m$,$\mathcal{I}$)
 4: else if $\varphi \equiv \varphi_m \vee \varphi_n$ OR $\varphi \equiv \varphi_m \wedge \varphi_n$ OR $\varphi \equiv \varphi_m \rightarrow \varphi_n$ then
 5:     EIU($\varphi_m$,$\mathcal{I}$)
 6:     EIU($\varphi_n$,$\mathcal{I}$)
 7: else if $\varphi \equiv \Box_{\mathcal{I}'}\varphi_m$ OR $\varphi \equiv \Diamond_{\mathcal{I}'}\varphi_m$ then
 8:     $\mathcal{I}'' \leftarrow \mathcal{I}' \oplus \mathcal{I}$
 9:     EIU($\varphi_m$,$\mathcal{I}''$)
10: end if
11: return $\varphi$
```

then the signal **y** is not vacuous. To check whether the signal s satisfies φ's mutations in Algorithm 6 (Line 3), we should use an off-line monitor such as [49].

5.3 Vacuity Detection in Testing and Falsification

Detecting vacuous satisfaction of specifications is usually applied on top of model checking tools for finite state systems [21, 73]. However, in general, the verification problem for hybrid automata (a mathematical model of CPS) is undecidable [6]. Therefore, a formal guarantee about the correctness of CPS modeling and design is impossible, in general. CPS are usually safety critical systems and the verification and validation of these systems is necessary. One approach is to use Model Based Design (MBD) with a mathematical model of the CPS to facilitate the system analysis and implementation [1]. Thus,

Algorithm 6 Antecedent Failure

```
Input: $\varphi$,y (Specification, Signal)
Output: $AF_\varphi$ a list of failed antecedents
1: $AF_\varphi \leftarrow \emptyset$
2: EIU($\varphi$,[0,0])
3: for each implication $(\varphi_i \rightarrow \psi_i) \in \varphi$ do
4:     $I\varphi_i \leftarrow \varphi_i.EI$
5:     if $\mathbf{y} \models \Box_{\mathcal{I}\varphi_i}(\neg\varphi_i)$ then
6:         $AF_\varphi \leftarrow AF_\varphi \cup \{\varphi_i \rightarrow \psi_i\}$
7:     end if
8: end for
9: return $AF_\varphi$
```

semi-formal verification methods are gaining popularity [69]. Although we cannot solve the correctness problem with testing and monitoring, we can detect possible errors with respect to STL requirements.

In order to use vacuity detection (Problem 2) in CPS testing, we provide our framework in Figure 5.1. The input generator creates initial conditions and inputs to the system under test. The system under test can be a Model, Process in the Loop, Hardware in the Loop or a real system. Then, a monitor checks the trace with respect to the specification [94, 78]. For each falsification, we will report to the user the falsifying trajectory to investigate the system for this error. Falsification can help us find counter-examples in the CPS (see Section 6.2). If after using stochastic-based testing and numerical analysis we could not find any issue, then we are more confident that the system works correctly.

However, it will be concerning if the numerical analysis is mostly based on vacuous signals. This is because vacuous signals satisfy the specification for reasons other than

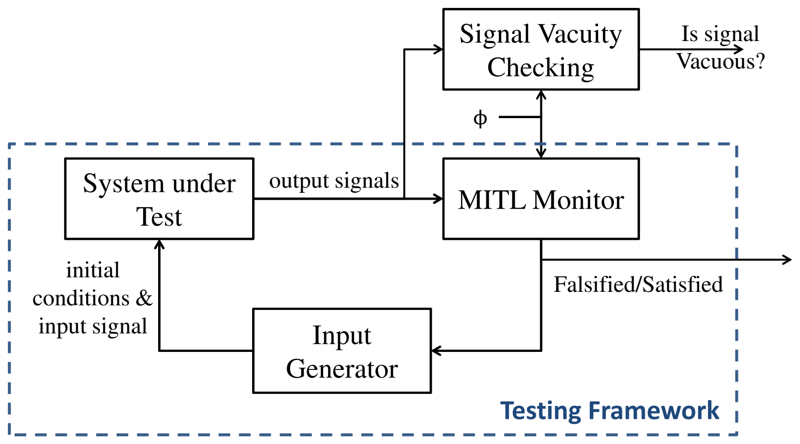

Figure 5.1: Using Signal Vacuity Checking to Improve the Confidence of an Automatic Test Generation Framework.

what was originally intended. Signal vacuity checking is conducted in Fig. 5.1 using Algorithm 6, and vacuous signals are reported to the user for further inspection. This will help users to focus their analysis on the part of the system that generates vacuous signals to prevent vacuous test generation. It should be noted that signal vacuity checking in the S-TaLiRo tool is computationally efficient (PTIME). The time complexity is proportional to the number of implication operations, the size of the formula and to the size of the signal [49].

5.4 Detecting Partially Covering Signals

A problem closely related to vacuity detection is the partial coverage problem. In this section, we show that Literal Occurrence Removal can be used for determining partially covering signals. Partially covering signals are the signals that not only satisfy the specification but also they satisfy Literal Occurrence Removal mutation:

Definition 5.4.1 (Partially Covering Timed Trace (Signal)) *Given an MITL (STL) formula φ in NNF, a timed trace μ (signal s) is partially covering if it satisfies the Literal Occurrence Removal mutation of φ.*

This mutation is generated by repeatedly substituting the occurrences of literals with $\bot$, which is denoted by $\varphi[l \leftarrow \bot]$ (see the Definition 4.4.3). In Algorithm 7, we check whether the signal will satisfy the mutated specification ($\varphi_i[l \leftarrow \bot]$). In the following, we prove that all satisfying signals are also partially covering signals:

Theorem 5.4.1 *For all MITL formula $\varphi \in \Phi$ in NNF, if there exists a disjunction subformula in φ, then for all μ such that $\mu \models \varphi$, it is always the case that there exists a literal $l \in litOccur(\varphi)$ s.t. $\mu \models \varphi[l \leftarrow \bot]$.*

The proof of Theorem 5.4.1 is provided in Appendix B. For any MITL (STL) specification φ, which contains one or more disjunction operators ($\vee$) in NNF, any timed trace (signal) that satisfies φ will also satisfy a mutation $\varphi[l \leftarrow \bot]$ for some literal occurrence l. Further, any specification which lacks a disjunction operator ($\vee$) in its NNF will not satisfy $\varphi[l \leftarrow \bot]$ for any literal occurrence l. That is, for formulas without any disjunction operator in NNF, we have $\varphi[l \leftarrow \bot] \equiv \bot$ since for any MITL formula φ, we have $\varphi \wedge \bot \equiv \bot$. If all satisfying signals are partially covering signals, what is the benefit of Algorithm 7? There are two applications for Algorithm 7 as follows.

First, Algorithm 7 can find which (how many) disjuncts are satisfied by the partially covering signal. This information can be used by the falsification technique to find the disjuncts/predicates that cause the formula satisfaction. Therefore, the falsification method will target the system behaviors corresponding to those predicates. As a result, Algorithm 7 can be used to improve the falsification method.

Second, Algorithm 7 can also be used for coverage analysis when falsification occurs. This is because with a slight modification, the dual of Algorithm 7 can help us to find the source of the falsification. According to Corollary B.0.2, for any φ in NNF, where φ has a conjunctive subformula of $\psi = \psi_1 \wedge \psi_2$, if $\mu \not\models \varphi$ then $\exists l \in litOccur(\psi)$ s.t. $\mu \not\models \varphi[l \leftarrow \top]$. Now, we can identify which conjunct of ψ contributes towards the falsification

Algorithm 7 Literal Occurrence Removal

Input: Φ,**y** (Specification, Signal)

Output: MF_φ a list of mutated formulas

```
1: $MF_\varphi \leftarrow \emptyset$
2: for each formula $\varphi_i \in \Phi$ do
3:     for each $l \in litOccur(\varphi_i)$ do
4:         if $\mathbf{y} \models \varphi_i[l \leftarrow \bot]$ then
5:             $MF_\varphi \leftarrow MF_\varphi \cup \{\varphi_i[l \leftarrow \bot]\}$
6:         end if
7:     end for
8: end for
9: return $MF_\varphi$
```

by substituting it by iteratively applying $\varphi[l \leftarrow \top]$. This can be better explained in the following example:

Example 5.4.1 *Assume $\varphi = \Diamond_{I_1}(a \wedge \Diamond_{I_2} b)$ and a falsifying trace $\mu \not\models \varphi$ exists. Formula φ contains conjunction of $\psi = a \wedge \Diamond_{I_2} b$ and $litOccur(\psi) = \{a, b\}$. We can substitute a and b with $\top$ to find the main source of falsification of φ as follows:*

• *If $\mu \models \varphi[a \leftarrow \top]$ then $\mu \models \Diamond_{I_1}(\Diamond_{I_2} b)$, so a is the source of the problem.*

• *If $\mu \models \varphi[b \leftarrow \top]$ then $\mu \models \Diamond_{I_1}(a)$, so b is the source of the problem.*

As a result, the dual of Algorithm 7 can be used to debug a trace when the counter example is created using falsification methodologies [1].

5.5 Experiments

To apply signal vacuity checking we use the S-TaLiRo testing framework [1, 64]. S-TaLiRo is a MatLab toolbox that uses stochastic optimization techniques to search for

Table 5.1: Automatic Transmission Requirements Expressed in Natural Language and MITL from [63].

Req.	**Natural Language**	MITL Formula
ϕ_1^{AT}	There should be no transition from gear two to gear one and back to gear two in less than 2.5 sec.	$\Box_{[0,27.5]}((g_2 \wedge \Diamond_{(0,0.04]} g_1) \rightarrow \Box_{[0,2.5]} \neg g_2)$
ϕ_2^{AT}	After shifting into gear one, there should be no shift from gear one to any other gear within 2.5 sec.	$\Box_{[0,27.5]}((\neg g_1 \wedge \Diamond_{(0,0.04]} g_1) \rightarrow \Box_{[0,2.5]} g_1)$
ϕ_3^{AT}	If the ω is always less than 4500, then the v can not exceed 85 in less than 10 sec.	$\Box_{[0,30]}(\omega \leq 4500) \rightarrow \Box_{[0,10]}(v \leq 85)$
ϕ_4^{AT}	Within 10 sec. v is less than 80 and from that point on, ω is always less than 4500.	$\Diamond_{[0,10]}((v \leq 80) \rightarrow \Box_{[0,30]}(\omega \leq 4500))$

system inputs for Simulink models which falsify the safety requirements presented in MTL [1]. If we report vacuous signals to S-TaLiRo users, then they will be aware of the vacuity issue. For example, users should find the system inputs that activate the antecedent in case of antecedent failure.

In the following, we illustrate the vacuous signal detection process by using the Automatic Transmission (AT) model provided in Section 2.3. S-TaLiRo calls the AT Simulink model in order to generate the output trajectories. The outputs contain two continuous-time real-valued signals: the speed of the engine ω (RPM) and the speed of the vehicle v. In addition, the outputs contain one continuous-time discrete-valued signal *gear* with four possible values (*gear* = 1, ..., *gear* = 4) which indicates the current gear in the auto-transmission controller. S-TaLiRo then monitors system trajectories with respect to the

Table 5.2: Reporting Signal Vacuity Issue for Each Mutated Formula.

Requirement	**Antecedent Failure**	Vacuous Signals / All Signals
ϕ_1^{AT}	$\Box_{[0,27.5]}\neg(g_2 \wedge \Diamond_{(0,0.04]}g_1)$	1989 / 2000
ϕ_2^{AT}	$\Box_{[0,27.5]}\neg(\neg g_1 \wedge \Diamond_{(0,0.04]}g_1)$	1994 / 2000
ϕ_3^{AT}	$\neg\Box_{[0,30]}(\omega \leq 4500)$	60 / 214
ϕ_4^{AT}	$\Box_{[0,10]}\neg(v \leq 80)$	1996 / 2000

requirements provided in Table 5.1. There, in the MITL formulas, we use the shorthand g_i to indicate the gear value, i.e. $(gear = i) \equiv g_i$. The simulation time for the system is set to 30 seconds; therefore, we can use bounded MITL formulas for the requirements.

After testing the AT with S-TaLiRo, we collected all the system trajectories. Then, we utilized the antecedent failure mutation on the specification to check signal vacuity (Algorithm 6) for each of the formulas that are provided in Table 5.1. We provide the antecedent failure specifications and the number of signals that satisfy them in Table 5.2. It can be seen in Table 5.2 that most of the system traces are vacuous signals where the antecedent is not satisfied. This helps the users to consider these issues and identify interesting test cases that can be used to initialize the system tester so that the antecedent is always satisfied.

5.6 Conclusions and Future Work

In this chapter, we considered vacuity detection with respect to signals. Our method reports the user, the number of vacuous signals during system testing. The techniques presented in this chapter can also be used for specification coverage analysis when falsification occurs. Namely, in a conjunctive requirement, we can identify which of the conjuncts contributed towards the falsification. This can be achieved by a straightforward modification of Algorithm 7 using Corollary B.0.2. As a result, users can debug the system when the specification is falsified.

Chapter 6

VACUITY AWARE FALSIFICATION

Falsification methods try to find unsafe behaviors with respect to safety specifications [1]. These methods are used to debug the CPS design during model based development (through simulations), implementation (through software-in-the-loop testing), and prototyping (through hardware-in-the-loop testing). *Request-Response* requirements are very important in safety critical systems where the CPS must react to a critical event. Request-Response requirements specify that every request should be followed by some response usually within some bounded time. For example, one such specification is "Every time the engine shifts from 1st to 2nd gear, then it does not shift back to 1st gear within 2.5 sec" [50]. In this case, the request is the event of shifting from 1st to 2nd gear, while the response is that the engine should not shift back to 1st gear for a bounded amount of time.

Falsification of Request-Response specifications is particularly difficult since the falsification method must first satisfy the antecedent and, then, falsify the consequent. Hence, it can be the case that computational effort is wasted because the generated test cases do not satisfy the request part of the specification (see for example the discussion in Section 5.5).

In this chapter, we propose a method to improve automatic test case generation for falsification of CPS with respect to Request-Response requirements. We consider the application of utilizing signal vacuity checking (Chapter 5) to improve the counter-example generation process. Vacuity detection is the problem of determining whether a temporal logic specification is vacuously satisfied with respect to a signal or system (see Chapter 5 for more details). One of the main sources of vacuity in system testing and verification is the antecedent failure in Request-Response requirements (see Section 5.2). Request-Response requirements contain at least one implication operation ($\varphi \rightarrow \psi$) which consists of an antecedent (φ) and a consequent (ψ). The system trajectories that fail to satisfy the antecedent (φ) will trivially satisfy the implication ($\rightarrow$). We refer to these system trajectories (behaviors) that trivially (vacuously) satisfy the specifications as *vacuous signals* (see Section 5.2). We have developed a framework to discover and focus the falsification process on non-vacuous signals in order to improve the counter-example generation for CPS. We call the framework *Vacuity Aware Falsification* (VAF). We have implemented our results on top of S-TaLiRo [13]. Our experimental results demonstrate that VAF achieves better falsification outcomes.

6.1 Related Works

The most related work is by Akazaki [4]. Akazaki applied Gaussian Process Regression (GPR) [19] to improve the probability of antecedent satisfaction during the falsification process using the robust semantics of Signal Temporal Logic (STL) formulas [43, 48]. The work in [4] focuses the search on the antecedent satisfaction by applying GPR to estimate the input region that most likely leads the system to satisfy the antecedent.

Our work is based on the results of our earlier work [36, 37] (see Chapters 5,4). We generalize the concept of antecedent failure as a subset of the signal vacuity issue [37], and we utilize the signal vacuity detection to provide an alternative solution to this problem using a two stage falsification process. Hence, our solution can benefit from various stochastic optimization techniques as we report in the experiments. Furthermore, our framework can also be applied to the systems where the robust semantics do not provide any guidance to the falsification process. For instance, this can be the case when the request in the specification is over the Boolean values $\{\top, \bot\}$. Finally, our approach can utilize the GPR method of [4] in order to improve the probability of antecedent satisfaction in our framework (Section 6.4, Stage 1).

A thorough recent review on search based falsification methods can be found in [68]. In our prior work [36, 37], we studied the problems of vacuous requirements and the impact of vacuous signals to the efficiency of the falsification process. However in [37] (see Chapter 5), we did not discuss how to improve the falsification process which is the focus of this chapter.

6.2 Falsification Framework

The falsification problem is the process of finding a witness trace/signal of the system Σ which does not satisfy a formal specification. To formalize the falsification problem:

Problem 3 (MTL Falsification) *Given a system Σ and an MTL specification ϕ, the falsification problem consists of finding a trace μ of the system Σ starting from an initial state x_0, parameter p, and an input signal u such that $\mu = \Sigma(x_0, p, u)$ and $\mu \not\models \phi$.*

Thus, falsification method tries to generate tests that reduce the robustness of the system behavior with respect to the specification and, eventually, become negative. The robust semantics (see Section 3.4.4) can help us to guide the search for MTL falsification [43, 48]. In order to falsify the specification, we use the temporal logic robustness as a cost function which we attempt to minimize. Therefore, we converted the falsification problem into an optimization problem. This is essentially the application of a range of stochastic or deterministic optimization algorithms such as Simulated Annealing [1], Cross-Entropy [89], Ant Colony Optimization [12], and Nelder-Mead [41] to the falsification problem.

The high level overview of the solution of the Temporal Logic Falsification (TLF) problem appears in Figure 6.1. The optimization algorithm generates initial conditions, and input signals. Then, the system under test produces the output signal for which the specification robustness is evaluated by an MTL monitor [50]. The process is repeated until a maximum number of tests is reached or a falsifying behavior is detected. The framework of Figure 6.1 can be implemented as a MATLAB toolbox, i.e., S-TaLiRo [13] or Breach

[41].

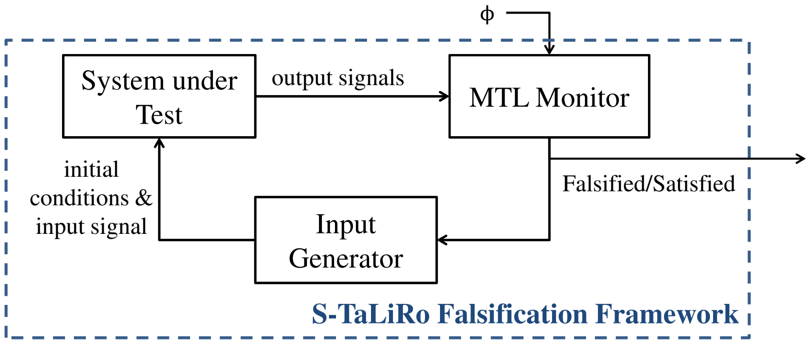

Figure 6.1: Overview of S-TaLiRo Testing Framework for the Metric Temporal Logic (MTL) Falsification Problem.

6.3 Problem Formulation

To simplify the presentation, we assume that the MTL specification has only one implication operation. In order to falsify the implication operation $\phi = \psi \rightarrow \varphi \equiv \neg\psi \vee \varphi$, we need to satisfy the antecedent ψ first. Therefore, we need to satisfy the antecedent before any attempt to falsify ϕ.

Problem 4 (Vacuity Aware Falsification) *Given a system Σ and a Request-Response MTL specification ϕ with an implication $\psi \rightarrow \varphi$ subformula in a positive form*[1], *find a trace μ of the system Σ starting from an initial state x_0, a fixed parameter p and an input signal $u = \overline{u}\underline{u}$ such that the prefix of the trace $\overline{\mu} = \Sigma(x_0, p, \overline{u})$ satisfies the antecedent $\overline{\mu} \models \psi$*[2] *and the whole trace $\mu = \Sigma(x_0, p, \overline{u}\underline{u})$ falsifies the main MTL formula $\mu \not\models \phi$.*

6.4 Vacuity Aware Falsification Framework

Our strategy for Vacuity Aware Falsification is a two stage solution: 1) The falsification process should first satisfy the antecedent. 2) For the traces that satisfy the antecedent,

[1]Without any negation in the parent nodes of $\psi \rightarrow \varphi$ in ϕ's parse tree.

[2]Traces that satisfy the antecedent are called non-vacuous traces [37].

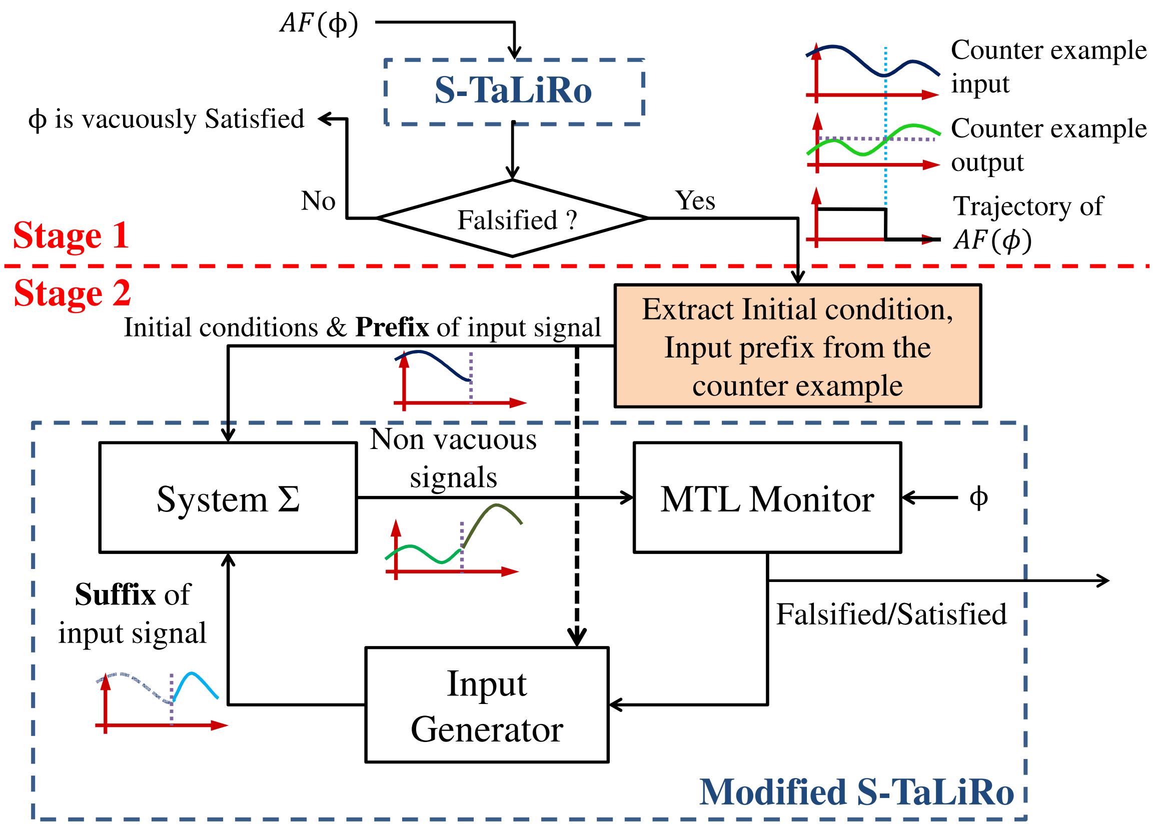

Figure 6.2: Proposed Flow for Vacuity Aware Falsification.

the process will guide the system toward falsifying the consequent. The proposed flow is provided in Fig. 6.2. To address the first stage, we create the Antecedent Failure (AF) formula $AF(\phi)$ where its falsification is interpreted as satisfaction of the antecedent. The AF in $AF(\phi)$ is a function that when given the formula $\phi = \Box_I(\psi \rightarrow \varphi)$, it returns $\Box_I(\neg\psi)$. The algorithm that extracts $AF(\phi)$ from a more complex MTL formula ϕ is provided in [37]. The $AF(\phi) = \Box_I(\neg\psi)$ formula asserts that the antecedent ψ would never happen in the time window of I, see [37] for more details. If S-TaLiRo falsifies $AF(\phi)$, it means that the antecedent ψ has eventually been satisfied. According to the architecture in Fig. 6.2, our proposed flow runs the testing framework in two stages:

Stage 1: We try to falsify $AF(\phi)$ using the falsification framework. If the $AF(\phi)$ is falsified during Stage 1, it means that the antecedent has eventually been satisfied by $\overline{\mu}$. Thus, we can proceed to Stage 2 to falsify the main formula ϕ. Otherwise, ϕ is vacuously satisfied in this run.

Stage 2: Since $AF(\phi)$ is falsified (in Stage 1), the counter example is the system input u that leads the system to create trajectories that satisfy the antecedent of the specification. Now, we should extract the shortest prefix of the input (denoted as $\overline{u}$) that leads the system to just falsify $AF(\phi)$ and immediately stop the simulation at the falsification point. The input prefix $\overline{u}$ leads the system to create the $\overline{\mu}$ trace. We should choose the input prefix $\overline{u}$ as short as possible to increase the search space to help the input generator to find the best suffix (input $\underline{u}$) that may lead the system to falsify ϕ. Now, we can explain why the system Σ should be deterministic. This is because in Stage 2 we expect to create the same satisfying output $\overline{\mu}$ for the same input prefix $\overline{u}$ that is extracted from Stage 1.

In Stage 2, we fix the initial condition x_0, parameter p and input prefix $\overline{u}$ which forces all the new testing trajectories to become *non-vacuous signals*. Recall that non-vacuous signals are the signals that satisfy the antecedent. As a result, the input generator will search over the suffix of the input $\underline{u}$ for the system to find a non-vacuous signal $\mu = \Sigma(x_0, p, \overline{u}\underline{u})$ that will eventually falsify the main formula.

The high-level pseudo code of the algorithm that corresponds to Fig. 6.2 is provided in Algorithm 8, where *opt* and *opt′* are the optimizers of choice, and N_{MAX} is the upper-limit for the number of optimizer's iterations. In Line 2, we run S-TaLiRo to falsify $\phi_{AF} = AF(\phi)$ (Stage 1). S-TaLiRo returns x_0, p, u correspond to the minimum robustness. If the search is successful, we move to Stage 2, unless we report that ϕ is vacuously satisfied (Line 15). In Stage 2, we extract the input prefix $\overline{u}$ in Line 6 and run S-TaLiRo to find the falsifying suffix (Line 7). In Line 7, $(\overline{u}, U)$ is the input space with fixed prefix $\overline{u}$. S-TaLiRo in Stage 2 searches over the suffixes of the input signal to find the trajectory μ' that falsifies the specification until the number of tests of *opt′* reaches to N_ϕ.

Finally, we report the falsification results in Lines 10 and 12. Here, we need to remark that $[\![\phi_{AF}]\!]_{\mathbf{d}}(\overline{\mu}, 0) < 0$ does not guarantee that there exists a μ such that $[\![\phi]\!]_{\mathbf{d}}(\mu, 0) < 0$.

Algorithm 8 Vacuity Aware Falsification

```
Input: $\Sigma, P, \mathcal{X}_0, U, \phi, opt, opt', N_{MAX}$;
Output: Message about Falsification Report;
Procedure VAF($\Sigma, P, \mathcal{X}_0, U, \phi, opt, N_{MAX}$)
 1: $\phi_{AF} \leftarrow AF(\phi)$
 2: $[x_0, p, u, N_{AF}] \leftarrow$ S-TaLiRo($\Sigma, P, \mathcal{X}_0, U, \phi_{AF}, opt, N_{MAX}$)
 3: $\mu \leftarrow \Sigma(x_0, p, u)$ ; $N_\phi \leftarrow N_{MAX} - N_{AF}$
 4: if $[\![\phi_{AF}]\!]_{\mathbf{d}}(\mu, 0) < 0$ then
 5:     Extract $\overline{\mu} \subset \mu$ such that $[\![\phi_{AF}]\!]_{\mathbf{d}}(\overline{\mu}, 0) < 0$
 6:     Extract $\overline{u} \subset u$ such that $\overline{\mu} = \Sigma(x_0, p, \overline{u})$
 7:     $[x_0, p, u', N_f] \leftarrow$ S-TaLiRo($\Sigma, p, x_0, (\overline{u}, U), \phi, opt', N_\phi$)
 8:     $\mu' \leftarrow \Sigma(x_0, p, u')$
 9:     if $[\![\phi]\!]_{\mathbf{d}}(\mu', 0) < 0$ then
10:         return "$\phi$ is falsified"
11:     else
12:         return "$\phi$ is NOT falsified"
13:     end if
14: else
15:     return "$\phi$ is vacuously satisfied!"
16: end if
```

6.4.1 Input Prefix-Suffix Example

An example for extracting the input prefix $\overline{u}$ from input u is depicted in Fig. 6.3. Consider the following specification $\phi = \Box_{[0,t_1]}(a \rightarrow \Diamond_{[0,t_2]} b)$ where $a \equiv v > 80$ and $b \equiv v < 60$, which formalizes the following natural language requirement:

"Always during the simulation time up to t_1 seconds, if the speed (v) goes above 80,

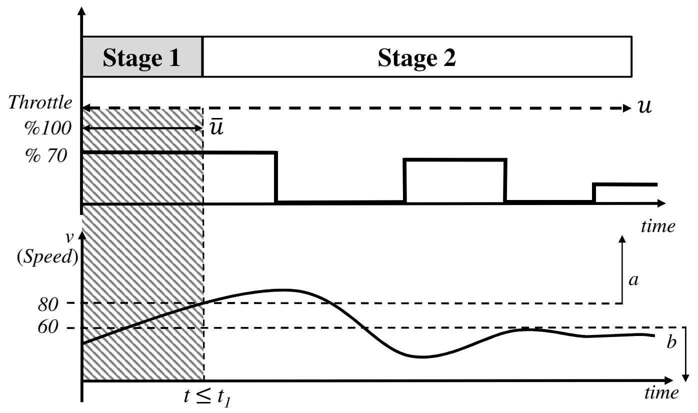

Figure 6.3: Stage 1 (Gray) and Stage 2 (White) of the Vacuity Aware Falsification.

then it must eventually drop below 60 in t_2 seconds"

Figure 6.3 represents the system input and trajectory corresponding to the formula ϕ. In Fig. 6.3, the system input u (Throttle) and the system output v (Speed) are presented. Any system trace μ that falsifies ϕ must first satisfy the precondition of ϕ. In other words, its prefix $\overline{\mu}$ must falsify the antecedent failure, namely $AF(\phi) = \Box_{[0,t_1]}\neg(a) = \Box_{[0,t_1]}(v \leq 80)$. The system trajectory in Fig. 6.3 is a falsifying signal for the antecedent failure $\Box_{[0,t_1]}(v \leq 80)$. Therefore, the trajectory in Fig. 6.3 is a non-vacuous signal since $v > 80$. The entire duration of input signal u is represented by a dashed line which contains the whole throttle schedule. The shortest prefix of the input signal $\overline{u}$ that leads the system to $v > 80$ is represented with a hashed box.

6.5 Experiments

In this section, we consider the application of our proposed method to improve the performance of the falsification method. Our experiments were conducted on a 64-bit Intel Xeon CPU (2.5GHz) with 64-GB RAM and Windows Server 2012. We used MATLAB 2015a to run the falsification toolbox S-TaLiRo[3] and to implement our method (Fig. 6.2 and Algorithm 8). For our experiments, we used the following stochastic optimization meth-

[3]S-TaLiRo: `https://sites.google.com/a/asu.edu/s-taliro/`

ods: Simulated Annealing (SA) [1], Cross-Entropy (CE) optimization [89], and Uniform Random (UR) sampling. We remark that all the experiments were performed with the default parameters for each optimization method. All the benchmark problems are available with the S-TaLiRo distribution or from the ARCH workshop repository [4].

6.5.1 Navigation Benchmark with Inputs

We consider a version of the Navigation Benchmark proposed by Fehnker and Ivancic [51] with a few modifications. The Navigation Benchmark is a four continuous-state autonomous affine hybrid automaton. Formal description of hybrid automata is provided in Definition 2.2.1. The primary modification is that now we allow for external inputs to the system (beyond the constant affine term in the original model). Even though affine hybrid systems can now be efficiently solved using reachability tools for hybrid systems, it still remains a challenge to verify Request-Response requirements as expressed in MTL. In addition, we remark that for this benchmark, the affine dynamics in each mode could be changed to complex smooth non-linear dynamics without any impact to the applicability of the proposed methodology.

The benchmark studies a hybrid automaton $\mathcal{H}$ with a variable number of discrete locations and 4 continuous state variables x_1, x_2, x_3 and x_4 that form the state vector $x = [x_1 \;\; x_2 \;\; x_3 \;\; x_4]^T$. The structure of the hybrid automaton can be better visualized in Fig. 6.4. The hybrid automaton has a number of modes (16 in the example of Fig. 6.4) where in each mode, the dynamics of the system are different.

In detail, in each location i of the hybrid automaton, the system evolves under the differential equation

$$\dot{x} = Ax - Bv(i) + Cu \tag{6.1}$$

[4]Applied Verification for Continuous and Hybrid Systems (ARCH):
`https://cps-vo.org/group/ARCH`

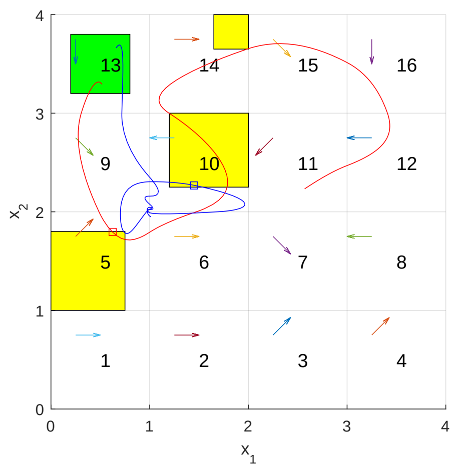

Figure 6.4: Modified Navigation Benchmark with 16 Locations (Modes): Two Trajectories Falsifying the Requirements ϕ_{NB_1}, and ϕ_{NB_2}.
where u is a 2 dimensional external continuous input to the system (in this benchmark for all time t, $u(t) \in [-5, 5]^2$), the matrices A, B and C are defined as

$$A = \begin{bmatrix} 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & -1.2 & 0.1 \\ 0 & 0 & 0.1 & -1.2 \end{bmatrix}, \; B = \begin{bmatrix} 0 & 0 \\ 0 & 0 \\ -1.2 & 0.1 \\ 0.1 & -1.2 \end{bmatrix} \text{ and } C = \begin{bmatrix} 0 & 0 \\ 0 & 0 \\ 1 & 0.5 \\ 0 & 1 \end{bmatrix}$$

and the constant vector term in each location is

$$v(i) = [\sin(\pi D(i)/4) \;\; \cos(\pi D(i)/4)]^T.$$

The array D is one of the parameters of the hybrid automaton that the user can control in order to define different benchmarks. It defines the input vector in each discrete location (see arrows in Fig. 6.4).

The invariant set of every location (mode) is a 1×1 box that constraints the "position" of the system (x_1, x_2), while the velocity (x_3, x_4) can flow unconstrained. The guards in each location are the edges and the vertices that are common among the neighboring locations. When a guard is reached, the system switches between system dynamics. The set of initial conditions is the set to $H_0 = \{(m, x) \mid m = 13, x \in [0.2\ 0.8] \times [3.2\ 3.8] \times [-0.4\ 0.4]^2\}$ (green box in Fig. 6.4).

Sample trajectories (under some input signal) of the system appear in Fig. 6.4 for initial conditions $(0.6821, 3.6558, 0.0685, -0.1790)$ for the blue trajectory and

$(0.4136, 3.2076, -0.3705, 0.3474)$ for the red trajectory.

We evaluated the following Request-Response requirements on the system (the sets which correspond to each predicate in the formulas are highlighted as yellow boxes in Fig. 6.4):

$$\begin{aligned}\phi_{NB_1} =& \Box((i = 10 \wedge x_1 \geq 1.2 \wedge x_2 \geq 2.25) \\ & \rightarrow \Box\neg(i = 5 \wedge x_1 \leq 0.75 \wedge x_2 \leq 1.8)) \\ \phi_{NB_2} =& \Box((i = 5 \wedge x_1 \leq 0.75 \wedge x_2 \leq 1.8) \\ & \rightarrow \Box\neg(i = 14 \wedge x_1 \geq 1.65 \wedge x_2 \geq 3.65))\end{aligned}$$

Both specifications state "if a set X is visited, then from that point on a set Y should not be visited". Variations of these requirements with timing constraints can be easily constructed. Since the predicates in ϕ_{NB_i} represent hybrid space (discrete locations with continuous state variables) we need to use hybrid distance semantics for the robustness semantics (see the generalized distance function $\mathbf{d_h}$ in in Section 3.4.2). Finally, we set $N_{MAX} = 200$ for Algorithm 8.

The results for both formulas are presented in Table 6.1 (ϕ_{NB} rows). All the experiments are conducted with the same number of optimization's tests (N_{MAX}) for both VAF and S-TaLiRo. The following observations can be made. First and foremost, using VAF uniformly improves the falsification outcomes independently of what the underlying method is. In all cases, by utilizing VAF, the rate of detecting falsifying behaviors is at least doubled. Second, on harder problem instances, i.e., for specification ϕ_{NB_2}, the VAF method outperforms the methods without VAF by an order of magnitude. In general, the difficulty of a benchmark can be assessed by how easily it is falsified using uniform random sampling.

Table 6.1: Comparing Vacuity Aware Falsification (VAF) with Temporal Logic Falsification (TLF) for the Falsification of ϕ_{NB}, ϕ_{AT}.

	Vacuity Aware Falsification (VAF)			S-TaLiRo	
Spec.	Opt.	AF(Spec.) is falsified (S1)	Spec. is falsified (S2)	Opt.	falsified
ϕ_{NB_1}	UR	100/100	88/100	UR	32/100
ϕ_{NB_1}	SA	91/100	59/91	SA	21/100
ϕ_{NB_1}	CE	100/100	67/100	CE	26/100
ϕ_{NB_2}	UR	100/100	10/100	UR	1/100
ϕ_{NB_2}	SA	92/100	23/92	SA	1/100
ϕ_{NB_2}	CE	100/100	24/100	CE	2/100
ϕ_{AT_1}	UR	97/100	97/97	UR	20/100
ϕ_{AT_2}	UR+SA	95/100 (UR)	95/95 (SA)	SA	17/100
ϕ_{AT_2}	UR+CE	91/100 (UR)	91/91 (CE)	CE	8/100

6.5.2 Automatic Transmission

The Automatic Transmission (AT) model is provided in Section 2.3. In order to evaluate the improvement of S-TaLiRo framework by using VAF, we considered the following safety Request-Response requirements:

1. "After shifting down into gear one, there should be no shift from gear one to any other gear within 2.5 sec."

2. "After shifting down into gear one, the engine speed ω should always stay below 3000 RPM within 2.5 sec."

The simulation time for the system is set to 30 seconds. Therefore, we can use bounded MTL formulas for the requirement such that the horizon of MTL formula equals to the simulation time (30 seconds). We formalize the above requirements as the follows:

$$\phi_{AT_1} = \Box_{[0,27.5]}((\neg g_1 \wedge \bigcirc g_1) \rightarrow \Box_{(0,2.5]} g_1)$$

$$\phi_{AT_2} = \Box_{[0,27.5]}((\neg g_1 \wedge \bigcirc g_1) \rightarrow \Box_{(0,2.5]} r_1)$$

where $g_1 \equiv \{gear = 1\}$ and $r_1 \equiv \{\omega \leq 3000\}$.

For the AT experiments, we set the number of optimization's tests to be 1000 ($N_{MAX} = 1000$). In addition, for VAF, we create the antecedent failure of ϕ_{AT_1} and ϕ_{AT_2} as follows:

$$AF(\phi_{AT}) = \Box_{[0,27.5]}(\neg(\neg g_1 \wedge \bigcirc g_1))$$

For evaluating VAF, we first setup the S-TaLiRo to falsify the $AF(\phi_{AT})$ which is the execution of Stage 1 in Fig. 6.2. Then, we run the second stage of VAF if Stage 1 was successful (see Fig. 6.2 Stage 2).

The falsification results of our proposed method are provided in Table 6.1 (ϕ_{AT_i} rows). It can be observed that for ϕ_{AT_1} the original UR method can successfully falsify only 20 out of 100 runs. However, our method successfully falsified the antecedent failure in 97 out of 100 runs in Stage 1, and among the runs that successfully falsify $AF(\phi_{AT})$, all of them would ultimately falsify the original specification in Stage 2.

For the rows corresponding to ϕ_{AT_2}, we choose UR for the falsification at Stage 1. This is due to the fact that the hybrid robustness value of g_1 is equivalent to $\top$ when $gear = 1$, and $\bot$ when $gear \neq 1$ with no intermediate values between them (see the generalized distance function $\mathbf{d_h^0}$ in Section 3.4.2). Therefore, the cost function of the stochastic optimizer does not decrease towards the falsification. In this case, since g_1 behaves like a Boolean event, UR is the preferred optimization algorithm in Stage 1. This demonstrates the flexibility of our method in that we can choose different optimizations for different Stages of VAF.

For the second stage of ϕ_{AT_2}, we used SA and CE. The VAF method with using UR+SA improves the falsification for SA-TaLiRo as follows: The original SA-TaLiRo successfully falsifies 17 out of 100 runs. We used UR-TaLiRo in Stage 1 to falsify antecedent failure in 95 out of 100 runs and SA-TaLiRo used those signal prefixes to falsify all of the runs in Stage 2. CE-TaLiRo improves the results in a similar way. Our experiments on AT show that VAF with UR-TaLiRo in Stage 1, can drastically improve the falsification process.

6.6 Conclusions and Future Work

We have introduced a new framework for Vacuity Aware Falsification (VAF) for Cyber-Physical Systems (CPS). Our experimental results demonstrate improvements for different S-TaLiRo optimization methods when we apply our new VAF framework. In the future, this method will be applied to more complex Request-Response requirements with more than one implication operations.

Chapter 7

ON-LINE MONITORING FOR BOUNDED MTL WITH PAST

Formal specifications in MTL [71] have been used for testing and semi-formal verification of CPS with tools such as S-TaLiRo [13] and Breach [41]. These tools use off-line and on-line monitoring algorithms to check whether the execution trace of a CPS satisfies/falsifies an MTL formula. In off-line monitoring, the execution trace is finite and generated by running the system for a bounded amount of time. Then, the off-line monitor checks whether the execution trace satisfies the specification. On the other hand, an on-line monitor runs simultaneously with the system. On-line monitoring is more challenging than off-line monitoring. This is because on-line monitoring is used for surveillance of the real-time events for safety critical applications and it may have catastrophic outcomes due to failure. In this chapter, we provide an efficient algorithm for on-line monitoring of a subset of MTL, which is called $\text{MTL}_{+pt}^{<+\infty}$. Syntax and Semantics of $\text{MTL}_{+pt}^{<+\infty}$ formulas are provided in Section 3.4.5.

7.1 Related Works

Monitoring of CPS is the problem of finding whether an execution trace satisfies the requirements. Monitoring of MTL has been considered in [94]. This algorithm is modified for monitoring of real-value signals considering robustness metric [48]. Since a rewriting based approach for monitoring is not efficient [57, 94], the MTL robustness computation method has the same runtime over head issue [48]. Therefore, in order to improve the monitoring time of S-TaLiRo [13], a dynamic programming method for computing the MTL robustness is provided in [49]. The algorithm in DP-TaLiRo [49] uses the method that is provided for LTL formulas of boolean traces [87] and it is applied to compute the

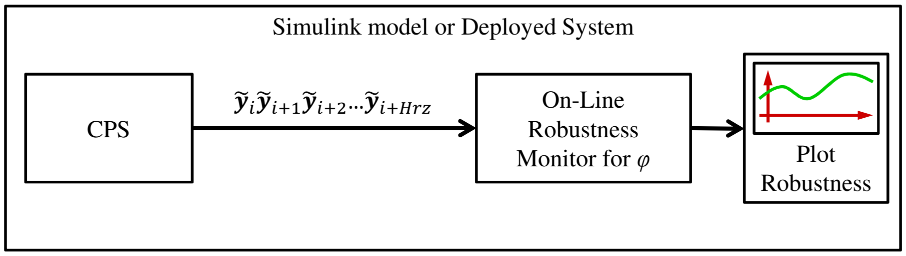
Simulink model or Deployed System
CPS
$\tilde{y}_i\tilde{y}_{i+1}\tilde{y}_{i+2}\cdots\tilde{y}_{i+Hrz}$
On-Line Robustness Monitor for φ
Plot Robustness

Figure 7.1: Overview of the Solution of the MTL$_{+pt}^{<+\infty}$ On-Line Monitoring Problem. The Monitored Robustness Values Could Be Used as Feedback to the CPS or It Could Be Plotted to Be Observed by a Human Supervisor If Needed.

robustness of MTL formulas with respect to real-value traces.

Another alternative to our S-TaLiRo testing framework is Breach [41]. They defined the Signal Temporal Logic (STL) to specify the real-time requirements of real value signals [78]. They provided a robustness concept of STL requirements [43] based on the definition of robustness in S-TaLiRo [48]. In Breach, they provide a sliding window algorithm for computing the robustness of finite traces [42]. The work that I will present in this chapter is followed by [33] and they applied their sliding window algorithm in [42] for on-line monitoring of real value signals. In [33], their on-line monitor does not consider the specifications that contain both past and future temporal operators; therefore, they monitor a subset of requirements that we can monitor in [35].

7.2 Problem Formulation

In the following, we represent the set of natural numbers including zero by $\mathbb{N}$ and the finite interval of $\mathbb{N}$ up to m by $\mathbb{N}_m = \{0, 1, \dots, m\}$. We assume that we have access to some discrete time execution or simulation traces of the CPS. We view *(execution or simulation) traces* as timed state sequences $\mu = \mu_0\mu_1\mu_2 \dots \mu_m = (\tilde{\mathbf{y}}_0, \tau_0)\ (\tilde{\mathbf{y}}_1, \tau_1)\ (\tilde{\mathbf{y}}_2, \tau_2)\ \dots\ (\tilde{\mathbf{y}}_m, \tau_m)$ where for each $k \in \mathbb{N}_m$, $\tau_k \in \mathbb{R}_{\geq 0}$ is a time stamp.

Throughout this chapter, the variable i, which ranges over $\mathbb{N}$, is used to represent the current simulation step or the current index of the sampling process. Since in Section 3.4.5

we assumed constant sampling rate for syntax and semantics of $\mathrm{MTL}_{+pt}^{<+\infty}$, when we mention time, we are actually referring to the corresponding sampling index i. More specifically, at each time i, we would like to monitor safety requirements represented as $\mathrm{MTL}_{+pt}^{<+\infty}$ formulas. These formulas capture safety properties of the system, such as bounded reactivity, which can be periodically analyzed for violation.

In our formulation, we use the robust (quantitative) semantics [48] that quantify the distance between a given execution trace of a CPS and all the execution traces that violate the property. The robustness of a formula $[\![\varphi]\!]$ with respect to a trace $\tilde{\mathbf{y}}$ time i is a value that measures how far is the trace from the satisfaction/falsification. This measure is an extension of boolean values representing satisfaction or falsification which is used in conventional monitoring. A positive robustness value means that the trace satisfies the property and a negative robustness means that the specification is not satisfied.

We assume that at each time i, the CPS outputs its current state $\tilde{\mathbf{y}}_i$ along with a finite prediction $\tilde{\mathbf{y}}_{i+1}$, $\tilde{\mathbf{y}}_{i+2}$, ..., $\tilde{\mathbf{y}}_{i+Hrz}$ of horizon length $Hrz \in \mathbb{N}$ (see Figure 7.1). The horizon length Hrz will be formally defined in Section 7.3; informally, Hrz is the required number of samples after time i so that any future requirements in the MTL specification ϕ are resolved, i.e., the horizon depends on the structure of the formula ϕ, $Hrz = hrz(\phi)$. When dealing with CPS, there exist numerous methods by which such a prediction horizon (forecasting) can be computed [46, 17, 81]. Now, we can formally define the on-line monitoring problem.

Problem 5 ($\mathrm{MTL}_{+pt}^{<+\infty}$ Robustness Monitoring) *Given an $MTL_{+pt}^{<+\infty}$ specification φ, a sampling instance i and an execution trace $\tilde{\mathbf{y}} = \tilde{\mathbf{y}}_0\tilde{\mathbf{y}}_1 \ldots \tilde{\mathbf{y}}_m$ such that $m = i + hrz(\varphi)$, compute the current robustness estimate $[\![\varphi]\!](\tilde{\mathbf{y}}, i)$ at time $\tau_i = i\Delta t$.*

Intuitively, φ represents a system invariant that must hold at every point in the system execution. This can also be viewed as testing for the specification robustness $[\![\Box\varphi]\!](\tilde{\mathbf{y}}, 0)$, where φ is an arbitrary $\text{MTL}_{+pt}^{<+\infty}$ specification. However, instead of caring about the satisfaction of the formula at the beginning of the time, we care about the potential of violating φ for which we design an on-line monitor.

7.3 Solution Overview

We provide an on-line monitoring approach for computing the robustness of an $\text{MTL}_{+pt}^{<+\infty}$ formula with respect to execution traces of a CPS. An overview of the solution for the $\text{MTL}_{+pt}^{<+\infty}$ on-line monitoring problem appears in Figure 7.1. Our method monitors the behavior of a CPS as it executes. Our toolbox is also useful for applications where Simulink models are actually used for process monitoring (and not simulation). In addition, it can also be used for code generation for general $\text{MTL}_{+pt}^{<+\infty}$ monitors for deployment on actual systems. Our method computes the robustness of invariants $[\![\varphi]\!](\tilde{\mathbf{y}}, i)$ by storing previous specification robustness values – if needed – and by only utilizing a bounded number of pairs of the execution trace $\tilde{\mathbf{y}}_{Hst}, \ldots, \tilde{\mathbf{y}}_{Hrz}$ where $Hst \in \mathbb{N}_i$ and it will be formally defined in Sec. 7.3. Our monitor uses bounded memory and, in the worst case, it has quadratic time complexity that depends on the magnitude of $Hrz - Hst$. In principle, our solution for robustness monitoring is inspired by the boolean temporal logic monitoring algorithm in [53].

7.4 Finite horizon and history of $\text{MTL}_{+pt}^{<+\infty}$

For each $\text{MTL}_{+pt}^{<+\infty}$ formula ψ we define the finite horizon $hrz(\psi)$ as the number of samples we need to consider in the future. In MTL, the satisfaction of the formula depends on what will happen in the future. In bounded MTL, the finite horizon $hrz(\psi)$ is the number of steps (samples) which we need to consider in the future in order to evaluate the formula ψ at the current time i. In other words, $hrz(\psi)$ is the number of steps into the future for which the truth value of the sub-formula ψ depends on [53]. Similarly, we define the finite history

$hst(\psi)$ of ψ as the number of samples we need to look into the past. That is, the number of steps in the past for which the truth value of the sub-formula ψ depends on. Intuitively, the $hst(\psi)$ is the size of the history we need to consider in order to keep track of what happened in the past to evaluate the formula ψ at present time. The finite horizon and the history can be defined recursively. We define $hrz(\psi)$ (similar to $h(\psi)$ in [53]) and we add the recursive definition of $hst(\psi)$ in the following:

$$hrz(p) = 0 \qquad hst(p) = 0$$

$$hrz(\neg\psi) = hrz(\psi) \qquad hst(\neg\psi) = hst(\psi)$$

$$hrz(\psi \mathbf{\ OP\ } \varphi) = max\{hrz(\psi), hrz(\varphi)\} \qquad hst(\psi \mathbf{\ OP\ } \varphi) = max\{hst(\psi), hst(\varphi)\}$$

$$hrz(\psi \mathcal{U}_{[l,u]} \varphi) = max\{hrz(\psi) + u - 1, hrz(\varphi) + u\} \qquad hst(\psi \mathcal{U}_{[l,u]} \varphi) = max\{hst(\psi), hst(\varphi)\}$$

$$hrz(\psi \mathcal{S}_{[l',u'\rangle} \varphi) = max\{hrz(\psi), hrz(\varphi)\}$$

$$hst(\psi \mathcal{S}_{[l',u'\rangle} \varphi) = \begin{cases} max\{hst(\psi) + u' - 1, hst(\varphi) + u'\} & \text{if } u' \neq +\infty \\ max\{hst(\psi) + l' - 1, hst(\varphi) + l'\} & \text{if } u' = +\infty \end{cases}$$

where $p \in AP$. Here, $\mathbf{OP}$ is any binary operator in propositional logic, and ψ, φ are $\text{MTL}_{+pt}^{<+\infty}$ formulas. For the unbounded $\mathcal{S}_{[0,+\infty)}$ operator, the computation of finite history is more involved and needs more explanation. Namely, we need to restate the dynamic programming algorithm for monitoring a sub-formula $\psi \mathcal{S}_{[0,+\infty)} \varphi$ based on the following works [87, 58]. According to the robustness semantics, the robustness of $\psi \mathcal{S}_{[0,+\infty)} \varphi$ at time i is as follows:

$$[\![\psi \mathcal{S}_{[0,+\infty)} \varphi]\!](\tilde{\mathbf{y}}, i) = \bigsqcup_{j=0}^{i} \left([\![\varphi]\!](\tilde{\mathbf{y}}, j) \sqcap \bigsqcap_{k=j+1}^{i} [\![\psi]\!](\tilde{\mathbf{y}}, k) \right)$$

also robustness of $\psi \mathcal{S}_{[0,+\infty)} \varphi$ at time $i - 1$ is

$$[\![\psi \mathcal{S}_{[0,+\infty)} \varphi]\!](\tilde{\mathbf{y}}, i-1) = \bigsqcup_{j=0}^{i-1} \left([\![\varphi]\!](\tilde{\mathbf{y}}, j) \sqcap \bigsqcap_{k=j+1}^{i-1} [\![\psi]\!](\tilde{\mathbf{y}}, k) \right)$$

Thus, we can rewrite $[\![\psi \mathcal{S}_{[0,+\infty)} \varphi]\!](\tilde{\mathbf{y}}, i)$ as

$$[\![\psi \mathcal{S}_{[0,+\infty)}\varphi]\!](\tilde{\mathbf{y}}, i) = [\![\varphi]\!](\tilde{\mathbf{y}}, i) \sqcup \left([\![\psi]\!](\tilde{\mathbf{y}}, i) \sqcap \left(\bigsqcup_{j=0}^{i-1}\left([\![\varphi]\!](\tilde{\mathbf{y}}, j) \sqcap \bigsqcap_{k=j+1}^{i-1}[\![\psi]\!](\tilde{\mathbf{y}}, k)\right)\right)\right) =$$
$$= [\![\varphi]\!](\tilde{\mathbf{y}}, i) \sqcup \left([\![\psi]\!](\tilde{\mathbf{y}}, i) \sqcap \left([\![\psi \mathcal{S}_{[0,+\infty)}\varphi]\!](\tilde{\mathbf{y}}, i-1)\right)\right)$$

Therefore, similar to [58] we recursively update the robustness of $\psi \mathcal{S}_{[0,+\infty)}\varphi$ at the current time i and save it in a variable called "*Pre*" to reuse it for the computation of the next time step (see [58] for more details). As a result, when we have an unbounded past time operator, we do not need the full history table. However, if the formula contains a nested future time operator, we need to extend the history to be long enough to contain the actual values. In other words, although for unbounded past time operators we do not need the whole history table, we should still extend the history to be able to store the actual simulation values (not the predicted values) in "*Pre*".

7.5 Monitoring Algorithm

For each $\text{MTL}_{+pt}^{<+\infty}$ formula φ, we construct a table called **Robustness Table** with width of $Hst + 1 + Hrz$, where $Hrz = hrz(\varphi)$ is the finite horizon of the specification formula φ, and, $Hst = Hrz + hst(\varphi)$, where $hst(\varphi)$ is the finite history of the specification φ. Hst is extended conservatively due to the fact that "*Pre*" value can only store the robustness values corresponding to the actual simulation. The height of the robustness table is the size of the formula φ ($|\varphi|$), where $|\varphi|$ is the number of sub-formulas of φ including itself. For example, assume we have a formula $\varphi = \boxdot_{[0,+\infty)}p \wedge \Box_{[1,2]}q$ and we intend to compute $[\![\varphi]\!](\mathcal{T}, i)$ at each time i. In formula φ, $Hst = 2$ and $Hrz = 2$. Since φ has unbounded past-time operators, it needs the *Pre* vector as well as the Robustness Table. The *Pre* vector appended to the Robustness Table is presented in Table 7.1. In particular, the *Pre* vector contains the value of past sub-formulas from the beginning of the time up to the current time.

Table 7.1: Pre Vector and Robustness Table.

Pre$[k]$	$T_{k,j}$	col. $j \Rightarrow$	-2	-1	0	1	2
	$k \Downarrow$	Time(i)	$i-2$	$i-1$	i	$i+1$	$i+2$
	$\psi_1 = \varphi$	$\psi_2 \wedge \psi_3$	$[\![\varphi]\!](\tilde{\mathbf{y}}, i-2)$	$[\![\varphi]\!](\tilde{\mathbf{y}}, i-1)$	$[\![\varphi]\!](\tilde{\mathbf{y}}, i)$	$[\![\varphi]\!](\tilde{\mathbf{y}}, i+1)$	$[\![\varphi]\!](\tilde{\mathbf{y}}, i+2)$
	ψ_2	$\Box_{[1,2]}q$	$[\![\psi_2]\!](\tilde{\mathbf{y}}, i-2)$	$[\![\psi_2]\!](\tilde{\mathbf{y}}, i-1)$	$[\![\psi_2]\!](\tilde{\mathbf{y}}, i)$	$[\![\psi_2]\!](\tilde{\mathbf{y}}, i+1)$	$[\![\psi_2]\!](\tilde{\mathbf{y}}, i+2)$
$[\![\psi_3]\!](\tilde{\mathbf{y}}, i-3)$	ψ_3	$\boxdot_{[0,+\infty)}p$	$[\![\psi_3]\!](\tilde{\mathbf{y}}, i-2)$	$[\![\psi_3]\!](\tilde{\mathbf{y}}, i-1)$	$[\![\psi_3]\!](\tilde{\mathbf{y}}, i)$	$[\![\psi_3]\!](\tilde{\mathbf{y}}, i+1)$	$[\![\psi_3]\!](\tilde{\mathbf{y}}, i+2)$
	ψ_4	p	$[\![\psi_4]\!](\tilde{\mathbf{y}}, i-2)$	$[\![\psi_4]\!](\tilde{\mathbf{y}}, i-1)$	$[\![\psi_4]\!](\tilde{\mathbf{y}}, i)$	$[\![\psi_4]\!](\tilde{\mathbf{y}}, i+1)$	$[\![\psi_4]\!](\tilde{\mathbf{y}}, i+2)$
	ψ_5	q	$[\![\psi_5]\!](\tilde{\mathbf{y}}, i-2)$	$[\![\psi_5]\!](\tilde{\mathbf{y}}, i-1)$	$[\![\psi_5]\!](\tilde{\mathbf{y}}, i)$	$[\![\psi_5]\!](\tilde{\mathbf{y}}, i+1)$	$[\![\psi_5]\!](\tilde{\mathbf{y}}, i+2)$

Table 7.2: Robustness Computation of Each Table Entries (Gray Cells Are Unused).

$T_{k,j}$	$i-2$	$i-1$	i	$i+1$	$i+2$
$k \Downarrow, j \Rightarrow$	$j=-2$	$j=-1$	$j=0$	$j=1$	$j=2$
Pre[1]	$T_{2,-2} \sqcap T_{3,-2}$	$T_{2,-1} \sqcap T_{3,-1}$	$T_{2,0} \sqcap T_{3,0}$	$T_{2,1} \sqcap T_{3,1}$	$T_{2,2} \sqcap T_{3,2}$
Pre[2]	$T_{5,-1} \sqcap T_{5,0}$	$T_{5,0} \sqcap T_{5,1}$	$T_{5,1} \sqcap T_{5,2}$	$T_{5,2}$	$+\infty$
Pre[3]	Pre[3]$\sqcap T_{4,-2}$	$T_{3,-2} \sqcap T_{4,-1}$	$T_{3,-1} \sqcap T_{4,0}$	$T_{3,0} \sqcap T_{4,1}$	$T_{3,1} \sqcap T_{4,2}$
Pre[4]	$\mathbf{Dist_d}(\tilde{\mathbf{y}}_{i-2}, \mathcal{O}(p))$	$\mathbf{Dist_d}(\tilde{\mathbf{y}}_{i-1}, \mathcal{O}(p))$	$\mathbf{Dist_d}(\tilde{\mathbf{y}}_{i}, \mathcal{O}(p))$	$\mathbf{Dist_d}(\tilde{\mathbf{y}}_{i+1}, \mathcal{O}(p))$	$\mathbf{Dist_d}(\tilde{\mathbf{y}}_{i+2}, \mathcal{O}(p))$
Pre[5]	$\mathbf{Dist_d}(\tilde{\mathbf{y}}_{i-2}, \mathcal{O}(q))$	$\mathbf{Dist_d}(\tilde{\mathbf{y}}_{i-1}, \mathcal{O}(q))$	$\mathbf{Dist_d}(\tilde{\mathbf{y}}_{i}, \mathcal{O}(q))$	$\mathbf{Dist_d}(\tilde{\mathbf{y}}_{i+1}, \mathcal{O}(q))$	$\mathbf{Dist_d}(\tilde{\mathbf{y}}_{i+2}, \mathcal{O}(q))$

In the following, we explain how the values of Table 7.2, the robustness table, are computed using Algorithms 9 and 10. In order to make our algorithms more readable, we used a vector to show the CPS output $\tilde{\mathbf{y}}_i, \tilde{\mathbf{y}}_{i+1}, \ldots, \tilde{\mathbf{y}}_{i+Hrz}$ to the monitoring (see Figure 7.1). We define a vector $\tilde{\mathbf{y}}'_i = \tilde{\mathbf{y}}_i\tilde{\mathbf{y}}_{i+1}\ldots\tilde{\mathbf{y}}_{i+Hrz}$ which appends current state $\tilde{\mathbf{y}}_i$ with predictions $\tilde{\mathbf{y}}_{i+1}$, $\tilde{\mathbf{y}}_{i+2}, \ldots, \tilde{\mathbf{y}}_{i+Hrz}$. In Table 7.1, i is the current simulation step which corresponds to column 0. At each simulation step i, for each unbounded past time sub-formula ϕ, we first save the values of the column $-Hst + hst(\phi)$ in the *Pre* vector (Algorithm 9 lines 1-3) since the column $-Hst + hst(\phi)$ contains the robustness value of ϕ from the beginning of the simulation. We need the *Pre* vector to compute the robustness of ϕ at the next sampling time using the dynamic programming method. In the above example, for $\boxdot_{[0,+\infty)}p$ the value at column -2 is saved in *Pre* to be used during robustness computation. Then, we shift all the robustness table entries of the predicates by one position to the left (Algorithm 9, lines 4-10). Then the loop (Algorithm 9, lines 11-21) recursively calls Algorithm 10 to fill the robustness table for each sub-formula from bottom to top.

Each call of Algorithm 10 (CR) computes each table entry $T_{k,j}$ (see tables 7.1,7.2) where column j is the horizon/history index and row k is the sub-formula index. For past sub-formulas the table entries are computed from left to right (Algorithm 9, lines 13-15), and for future sub-formulas the table entries are computed from right to left (Algorithm 9, lines 17-19). New values for predicates (according to execution traces) will be placed in column 0 and the predicted values of the predicates will be saved in columns 1 to Hrz (Algorithm 10, lines 2-5). Table 7.2 shows the updates of predicate values in rows 4, and 5 which correspond to Algorithm 10, line 4.

Algorithm 9 On-Line Monitor

Input: φ, $\tilde{\mathbf{y}}'_i = \tilde{\mathbf{y}}_i\tilde{\mathbf{y}}_{i+1}\ldots\tilde{\mathbf{y}}_{i+Hrz}$, $\mathbf{d}$, $\mathcal{O}$; **Global variables:** T, Pre; **Output**: $T_{1,0}$(*robustness value*). **procedure** MONITOR($\varphi, \tilde{\mathbf{y}}'_i, \mathbf{d}, \mathcal{O}$)

```
1:  for $k \leftarrow 1$ to $|\varphi|$ do
2:      $Pre(k) \leftarrow T_{k,(-Hst+hst(\varphi_k))}$
3:  end for
4:  for $j \leftarrow 1 - Hst$ to $Hrz$ do
5:      for $k \leftarrow 1$ to $|\varphi|$ do
6:          if $\varphi_k = p \in AP$ then
7:              $T_{k,j-1} \leftarrow T_{k,j}$
8:          end if
9:      end for
10: end for
11: for $k \leftarrow |\varphi|$ down to 1 do
12:     if $\varphi_k = \varphi_m \mathcal{S}_{[l',u')} \varphi_n$ then
13:         for $j \leftarrow -Hst + hst(\varphi_k)$ to $Hrz$ do
14:             $T_{k,j} \leftarrow CR(\varphi_k, j, \tilde{\mathbf{y}}'_i, \mathbf{d}, \mathcal{O})$
15:         end for
16:     else
17:         for $j \leftarrow Hrz$ down to $-Hst + hst(\varphi_k)$ do
18:             $T_{k,j} \leftarrow CR(\varphi_k, j, \tilde{\mathbf{y}}'_i, \mathbf{d}, \mathcal{O})$
19:         end for
20:     end if
21: end for
22: return $T_{1,0}$
end procedure
```

Algorithm 10 Robustness Computation (CR)

Input: φ_k, j, $\tilde{\mathbf{y}}'_i = \tilde{\mathbf{y}}_i\tilde{\mathbf{y}}_{i+1}\ldots\tilde{\mathbf{y}}_{i+Hrz}$, $\mathbf{d}$, $\mathcal{O}$; **Global variables:** T, Pre; **Output**: $T_{k,j}$.

```
procedure CR($\varphi_k, j, \tilde{\mathbf{y}}'_i, \mathbf{d}, \mathcal{O}$)
1:  if $\varphi_k = \top$ then $T_{k,j} \leftarrow +\infty$
2:  else if $\varphi_k = p \in AP$ then
3:      if $j >= 0$ then
4:          $T_{k,j} \leftarrow \mathbf{Dist_d}(\tilde{\mathbf{y}}_{i+j}, \mathcal{O}(p))$
5:      end if
6:  else if $\varphi_k = \neg\varphi_m$ then
7:      $T_{k,j} \leftarrow -T_{m,j}$
8:  else if $\varphi_k = \varphi_m \vee \varphi_n$ then
9:      $T_{k,j} \leftarrow T_{m,j} \sqcup T_{n,j}$
10: else if $\varphi_m \mathcal{U}_{[l,u]} \varphi_n$ then
11:     if $j + l \leq Hrz$ then
12:         $tmp_{min} \leftarrow \bigsqcap_{j \leq j' < j+l} T_{m,j'}$
13:         $T_{k,j} \leftarrow -\infty$
14:         $mins \leftarrow min\{Hrz, j+u\}$
15:         for $j' \leftarrow j + l$ to $mins$ do
16:             $T_{k,j} \leftarrow T_{k,j} \sqcup (tmp_{min} \sqcap T_{n,j'})$
17:             $tmp_{min} \leftarrow tmp_{min} \sqcap T_{m,j'}$
18:         end for
19:     else
20:         $T_{k,j} \leftarrow -\infty$
21:     end if
22: else if $\varphi_m \mathcal{S}_{[l',u')} \varphi_n$ then
23:     $tmp_{min} \leftarrow \bigsqcap_{j-l' < j' \leq j} T_{m,j'}$
24:     if $u' \neq +\infty$ then
25:         $T_{k,j} \leftarrow -\infty$
26:         for $j' \leftarrow j - l'$ down to $j - u'$ do
27:             $T_{k,j} \leftarrow T_{k,j} \sqcup (tmp_{min} \sqcap T_{n,j'})$
28:             $tmp_{min} \leftarrow tmp_{min} \sqcap T_{m,j'}$
29:         end for
30:     else
31:         if $j = -Hst + hst(\varphi_k)$ then
32:             $tmp_{\mathcal{S}} \leftarrow Pre[k] \sqcap T_{m,j}$
33:         else
34:             $tmp_{\mathcal{S}} \leftarrow T_{k,j-1} \sqcap T_{m,j}$
35:         end if
36:         $T_{k,j} \leftarrow (T_{n,j-l'} \sqcap tmp_{min}) \sqcup tmp_{\mathcal{S}}$
37:     end if
38: end if
39: return $T_{k,j}$
end procedure
```

In the following, we explain how the CR Algorithm 10 computes the MTL robustness values for three different cases of MTL:

Case 1 (Lines 10-20): The robustness of bounded future temporal sub-formulas with interval $[l, u]$ at each column j is computed given the values of its operands for columns j up-to $min\{j + u, Hrz\}$ (Line 14). For example, this case is used in Table 7.2 to compute the robustness of sub-formula $\psi_2 = \boxdot_{[1,2]}q$ from right to left. Case 1 in CR Algorithm is similar to the DP-TALIRO algorithm [50].

Case 2 (Lines 23-28): The robustness of bounded past temporal sub-formulas with interval $[l', u']$ at each column j is computed given the values of its operands for columns j down-to $j - u'$ (Line 25).

Case 3 (Lines 30-36): The robustness of unbounded past temporal sub-formulas with interval $[l', +\infty)$ for column j is computed using the stored value in column $j - 1$ in dynamic programming fashion (Line 33) and using the Pre vector (Line 31). For example, Case 3 is used to compute the robustness of $\psi_3 = \boxdot_{[0,+\infty]}p$ using $Pre[3]$ from left to right in Table 7.2.

Finally, we update table entries for the top row which corresponds to $\psi_1 = \varphi$. Since its corresponding operator $\wedge$ is propositional (Algorithm 10 Lines 6-9), we can update its value from any direction. The high level explanation of Algorithm 9 is described as follows:

1. Store values of column $-Hst + hst(\phi_k)$ for each unbounded past sub-formula ϕ_k in $Pre[k]$ and shift the table entries of predicates one to the left (Lines 1-10).

2. For each row i from $|\varphi|$ to 1 compute the robustness values according to:

   (a) If φ_i is a future temporal operator, for each column j from Hrz down to $-Hst + hst(\varphi_i)$, update table entry $T_{i,j}$ using Algorithm 10.

   (b) If φ_i is a past temporal operator, for each column j from $-Hst + hst(\varphi_i)$ up to Hrz update table entry $T_{i,j}$ using Algorithm 10.

3. Return the robustness ($T_{1,0}$).

We provided the proof of this section in Appendix C.

7.6 Experimental Results

We measured the overhead of the proposed monitoring framework on the Automatic Transmission (AT) model provided in Section 2.3. We introduce our $\mathrm{MTL}_{+pt}^{<+\infty}$ monitoring block in the AT model and test the performance over a set of specifications. In order to test the runtime overhead of our work, we artificially generate 30 different $\mathrm{MTL}_{+pt}^{<+\infty}$ formulas based on typical critical safety formulas to show that the runtime overhead depends on both of the size of the formula and the horizon/history. We test our method for 100 runs of monitoring algorithm for each specification (formula), and for each run we use 100 simulation steps. Then, we compute the mean and variance of the overhead for each simulation step which is the execution time of Algorithm 9 (in Table 8.4). In Table 8.4, the overhead is measured on specifications that contain either nested Until operators (U columns) or nested Eventually operators (E columns).

We generate 30 formulas according to the following templates:

- **E formulas:** $\phi_n(H) = p_j \longrightarrow \psi_n(H/n)$

  where $H \in \mathbb{N}$ is the finite horizon of the formula. In Table 8.4, we used 1,000, 2,000 and 10,000 for the size of the horizon. Here, p_j is an arbitrary predicate and $\psi_n(H/n)$ is defined recursively as follows:

  $\psi_1(h) = \Diamond_{[0,h]} p_k$ and $\psi_n(h) = \Diamond_{[0,h]}(p_l \wedge \psi_{n-1}(h))$, for $1 < n \leq 10$

  where $h = H/n$, i.e., the finite horizon H divided by the number of nested sub-formulas n and p_k, p_l are arbitrary predicates.

- **U formulas:** $\phi_n(H) = p_j \longrightarrow \psi_n(H/n)$

  where $H \in \mathbb{N}$ is the finite horizon of the formula. In Table 8.4, we used 1,000, 2,000 and 10,000 for the size of the horizon of H. Here, p_j is an arbitrary predicate and

Table 7.3: The Overhead on Each Simulation Step on the Automatic Transmission Model with Specifications of Increasing Length. Table Entries Are in Milliseconds.

#	H=1,000				H=2,000				H=10,000			
	E		U		E		U		E		U	
	Mean	Var.	Mean	Var.	Mean	Var.	Mean	Var.	Mean	Var.	Mean	Var.
$\phi_1(H)$	2.39	0.00	4.83	0.00	8.03	0.00	15.8	0.001	188.8	0.001	358.5	0.036
$\phi_3(H)$	4.24	0.00	7.5	0.001	12.7	0.00	25.09	0.005	314.4	0.01	599	0.665
$\phi_5(H)$	4.66	0.00	8.36	0.001	14.01	0.00	27.8	0.005	309.2	0.077	650	0.014
$\phi_7(H)$	4.95	0.00	8.94	0.00	14.83	0.00	29.33	0.006	311	0.013	674.2	0.033
$\phi_9(H)$	5.23	0.00	9.46	0.001	15.4	0.001	30.56	0.007	317.5	0.011	683.5	0.698

$\psi_n(H/n)$ is defined recursively as follows:

$\psi_1(h) = p_k U_{[0,h]} p_l$ and $\psi_n(h) = p_m U_{[0,h]}(p_n \wedge \psi_{n-1}(h))$, for $1 < n \leq 10$

where $h = H/n$ and p_k, p_l, p_m, p_m are arbitrary predicates.

As illustrated in Table 8.4, the computational complexity of the monitoring algorithm is closely related to the horizon and history size. Since the algorithm's complexity is of order $O(n^2)$ where n is the horizon/history, the added overhead (in worst case execution) is quadratic in terms of the size of the horizon for some formulas in table 8.4 (like $\phi_1(H)$). Moreover, in most cases, the impact of the number of nested temporal operators is not significant compared to the size of horizon/history windows. From Table 8.4, we notice that when the horizon and history size is less than 2,000, the overhead for each simulation step is negligible with our prototype implementation. Furthermore, for most practical reactivity requirements, it is quite unlikely that even a window size of 2,000 sampling points is necessary.

7.7 Case Study

In the following, we utilize the monitoring method on an industrial size high-fidelity engine model. The model is part of the SimuQuest Enginuity [91] Matlab/Simulink tool package. The Enginuity tool package includes a library of modules for engine component blocks. It also includes pre-assembled models for standard engine configurations. In this work, we use the Port Fuel Injected (PFI) spark ignition, 4 cylinder inline engine configuration. It models the effects of combustion from first physics principles on a cylinder-by-cylinder basis, while also including regression models for particularly complex physical phenomena. The model includes a tire-model, brake system model, and a drive train model (including final drive, torque converter and transmission). The input to the system is the throttle schedule. The output is the normalized air-to-fuel(A/F) ratio. Simulink reports that this is a 56 state model. Note that this number represents only the visible states. It is possible that more states are present in the blackbox s-functions which are not accessible. This is a high dimensional non-linear system for which reachability analysis is very difficult. It also includes lookup tables, non-linear components, and inputs that affect the switching guards.

A specification of practical interest for an engine is the settling time for the A/F ratio, which is the quotient between the air mass and fuel mass flow. Ideally, the normalized A/F ratio λ should always be 1, indicating that the ratio of the air and fuel flow is the same as the stoichiometric ratio. Under engine operating conditions, this output fluctuates $\pm\%10$. We add the on-line monitoring block to the Simulink model as presented in Figure 7.2.

Our goal is to monitor the engine while allowing temporary fluctuations to λ. We formally define the specification as follows:

$$\phi_{pt} = (\lambda \text{ out of bounds}) \rightarrow \Diamond_{[0,1]} \boxdot_{[0,1]} \neg(\lambda \text{ out of bounds})$$

Here, the formal specification states that if the A/F ratio exceeds the allowed bounds,

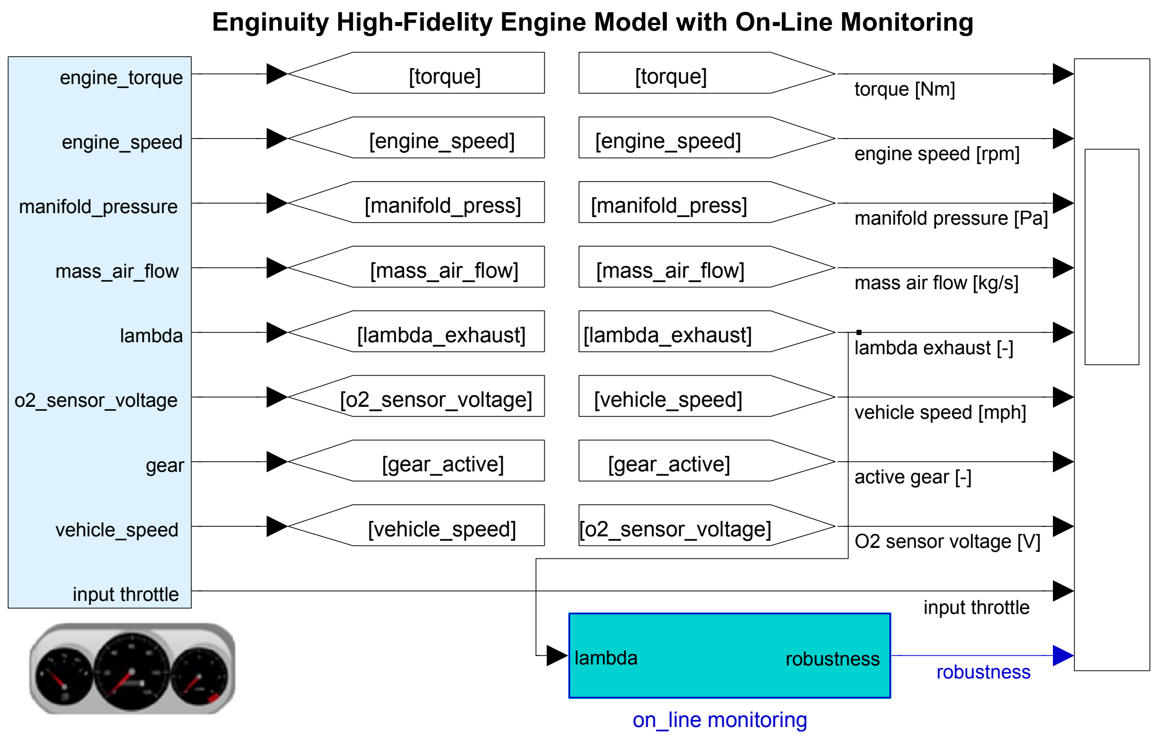

Figure 7.2: SimuQuest [91] Enginuity Matlab Simulink Engine Model with the On-Line Monitoring Block.

then the ratio should have been settled for at least one second within the last two seconds.

Notice that an alternative presentation of the formula would be to use the future eventually and always operators, i.e. the formula would be defined as follows:

$$\phi_{ft} = (\lambda \text{ out of bounds}) \rightarrow \Diamond_{[0,1]}\Box_{[0,1]}\neg(\lambda \text{ out of bounds})$$

In this case, the specification states that always, if the A/F ratio output exceeds the allowed bounds, then within one second it should settle inside the bounds and stay there for a second.

Clearly, both ϕ_{pt} and ϕ_{ft} are equivalent in terms of the set of traces that satisfy/falsify each specification[1]. However, in real-time robustness monitoring, there is an important distinction between the two. When the specification requires future information, either a predictor is put in place or the semantics will handle only the current information. In

[1]Formally, this is the case if we ignore the first 2 seconds of the execution trace as well as the last 2 seconds – if the execution trace is finite.

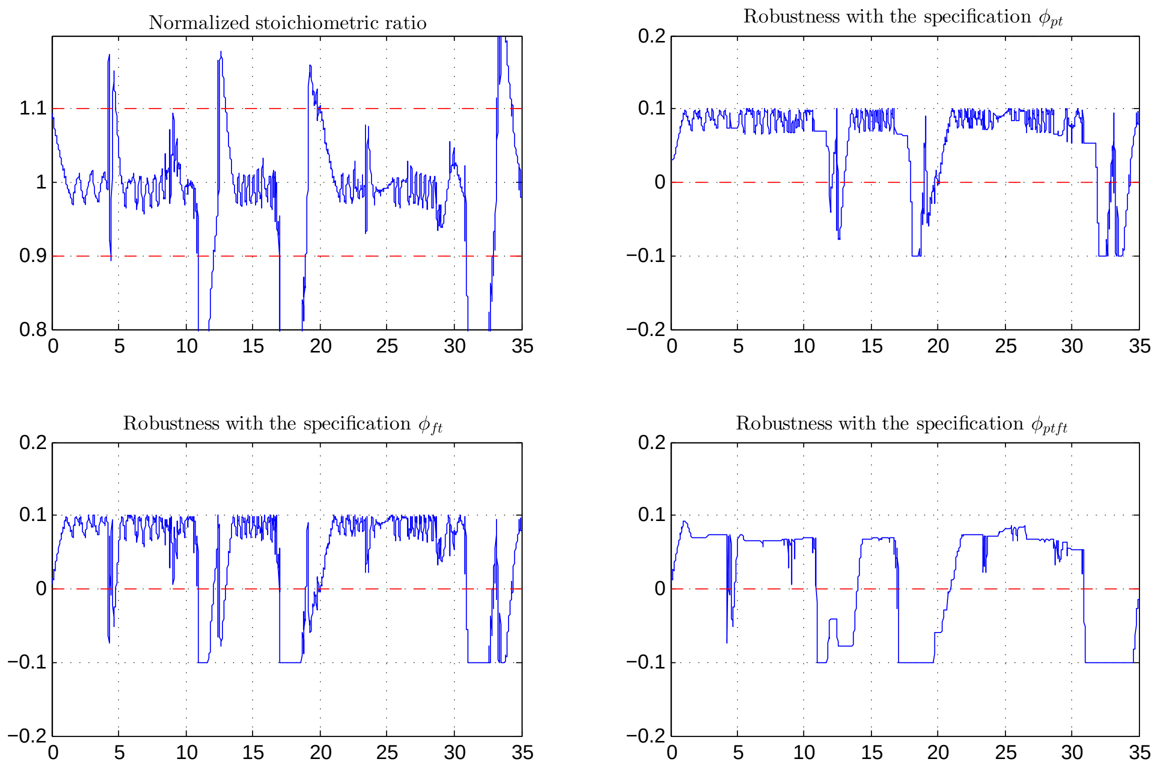

Figure 7.3: Runtime Monitoring of Specifications ϕ_{pt}, ϕ_{ft} and ϕ_{ptft} On the High-Fidelity Engine Model. The Figure Presents a Normalized Stoichiometric Ratio, and the Corresponding Robustness Values for Specifications ϕ_{pt}, ϕ_{ft} and ϕ_{ptft}. Note That No Predictor Is Utilized When Computing the Robustness Values.

this case, without a predictor, the future time formula reduces to the propositional formula ϕ_{ft} = (λ out of bounds) $\rightarrow \neg$(λ out of bounds) $\equiv$ (λ out of bounds). Therefore, past time operators should be used. Recall that when monitoring robustness, our goal is to provide early warning on when the specification may fail by approaching dangerously an undesired threshold. In other words, the past formula allows us to reason about the robustness of the actual system observations, while the future formula in collaboration with a forecast model would allow us to estimate the likely robustness. This is in contrast to many boolean monitoring algorithms which issue an "*undecided until further notice*" verdict that does not provide any actionable information.

A third alternative monitoring specification is the following formula:

$$\phi_{ptft} = \boxdot_{[0,2]}((\lambda \text{ out of bounds}) \rightarrow \Diamond_{[0,1]}\Box_{[0,1]}\neg(\lambda \text{ out of bounds}))$$

Table 7.4: Simulation Runtime Statistics for the High-Fidelity Engine Model Running for 35 Seconds with Simulation Step Size of 0.01s. The Results Include the Confidence Intervals for the Mean Simulation Runtime.

Simulation runtime(sec.)	Est. Mean	Est. Std. Dev	95%		99%	
			LB	UB	LB	UB
Without monitor	10.811	0.090	10.778	10.844	10.766	10.857
With monitor	10.987	0.086	10.955	11.019	10.944	11.030

This specification states that at some point in the last two seconds, when λ is out of bounds then within the next second, λ will not be out of bounds and stay there for one second. This alternative seems to be the balance between the ϕ_{pt} and ϕ_{ft} formulas. Where ϕ_{pt} purely relies on past information, and ϕ_{ft} relies on information from a predictor, ϕ_{ptft} has the advantage that it utilizes both the information from the past but also it could include information from the predictor.

An example of real-time monitoring on the high-fidelity engine model is presented in Figure 7.3. The figure illustrates the significance of using past time operators when defining specifications. Due to the lack of predictor information, the ϕ_{ft} monitor falsely returns falsification at about 4 seconds whereas the ϕ_{pt} monitor does not.

In the following, we analyze the overhead of the monitoring algorithm for this case study. Since the runtime is influenced by numerous sources of nondeterminism, we apply the central limit theorem to form confidence intervals for the mean simulation runtime when running the simulations with and without the monitor. To generate the results in Table 7.4, we collected 30 samples with 100 simulation run-times in each sample. We note that the difference between the estimated mean simulation runtime when adding the monitor is 0.97%. The experimental results were generated on an Intel Xeon X5647 (2.993GHz, 8 CPUs) machine with 12 GB RAM, Windows 7, and Matlab 2012a.

7.8 Conclusions and Future Work

We have presented an algorithm for monitoring the robustness of combined past and future MTL specifications. Our framework can incorporate predicted or estimated data as provided by a model predictive component. We have created a Simulink toolbox for MTL robustness monitoring which is distributed with the S-TaLiRo tools [13]. Our experiments indicate that the toolbox adds minimal overhead to the simulation time of Simulink models and it can be used for both runtime analysis of the models and for off-line testing. The current version of the tool allows reasoning over timed state sequences generated under a constant sampling rate. In the future, we would like to relax this constraint so that we allow arbitrary sampling functions. In addition, we would like to investigate the possibility of porting our monitor on FPGA platforms similar to [53, 86].

Chapter 8

OFF-LINE MONITORING FOR TPTL

In this chapter, we consider off-line monitoring of TPTL specifications. The time complexity of the off-line monitoring for MTL is linear to the size of a finite system trace and linear to the size of MTL formula. Several algorithms using dynamic programming [49] or sliding windows [42] have been proposed for MTL monitoring of CPS. Therefore, the off-line monitoring of MTL specifications are not challenging. Instead, in this chapter, we consider off-line monitoring of TPTL specifications which are more expressive than MTL specifications [27]. Syntax and Semantics of TPTL formulas are provided in Section 3.2.3.

8.1 Related Works

TPTL is an extension of Linear Temporal Logic (LTL) with freeze quantifiers represented as "$x.$". A freeze quantifier $x.$ assigns to time variable x the "current" time stamp when the corresponding subformula $x.\varphi(x)$ is evaluated [11]. Then, the time value (stored in x) can be evaluated inside time constraints which are linear inequalities over the time variables.

Since its introduction, two semantics where considered for TPTL [11, 27]. Alur's semantics [11] allows two time variables in time constraints (for example $x + 1 \leq y + 4$). In contrast, Raskin's semantics allows only one time variable in the time constraint ($x \leq 4$) and implicitly considers the current time as the second time variable [27, 85]. Since the latter semantics was first considered by Jean-Franois Raskin in [85], we will refer to it as "Raskin's TPTL semantics" in this chapter. Raskin's TPTL semantics was mentioned with alternative terms such as "Timed LTL" in [72]. In another line of work, in [34], the authors augmented Alur's time constraints with more complex temporal-special predicates to define the closeness property of two different CPS trajectories. However, the authors in [34] did not provide a TPTL monitoring algorithm.

Since TPTL subsumes MTL, it is expected that the monitoring problem of TPTL is computationally more complex [52]. It has been proven that monitoring of a finite trace with respect to Alur's TPTL specification is PSPACE-hard [79]. In [79], the authors transform a Quantified Boolean Formula (QBF), which is PSPACE-hard, into a TPTL formula with real value time variables. A similar complexity result (PSPACE-hard) for Raskin's TPTL semantics is obtained for integer time variables in [52]. It is mentioned in [52] that in order to obtain a polynomial time algorithm for TPTL monitoring (path checking), we need to fix the number of time variables. In other words, if the number of time variables is bounded then the finite trace monitoring will be polynomial to the size of the TPTL formula. However, in [52], the authors did not provide any applicable algorithm for TPTL monitoring and they focused only on the complexity class.

In this work, we move one step further from [52], and allow the number of time variables to be arbitrary, but they must be independent to each other[1]. For this fragment of TPTL, we provide an efficient TPTL monitoring algorithm which has time complexity quadratic in the length of the finite trace. In addition, the runtime of the algorithm is proportional to the number of time variables in TPTL.

A rewriting based algorithm for TPTL has been provided in [28]. In [28], the authors did not evaluate the time complexity of their proposed algorithm. The rewriting technique was used for on-line monitoring of TPTL specifications in [56]. The authors used the relativization of TPTL formula with respect to the sequence of observed states [56], and it was reported that the time complexity is exponential to the size of TPTL formula [56].

[1]In Section 8.2, Definition 8.2.1, we introduce independent time variables.

8.2 Problem Formulation

In this section, we introduce a TPTL fragment for which we have developed a monitoring algorithm. This restriction is crucial for obtaining the polynomial runtime of the algorithm.

Definition 8.2.1 (Independent Time Variable) *A time variable x is independent if it is in the scope of only one freeze quantifier $x.$ and no other time variable is in the scope of the corresponding freeze quantifier $(x.)$.*

For example in $x.(\psi(x) \vee \Diamond y.\varphi(x, y))$, neither x nor y is independent. This is because x is within the scope of the freeze time quantifiers $x.$ in $x.(\psi(x) \vee \Diamond y.\varphi(x, y))$ and $y.$ in $y.\varphi(x, y)$. Similarly, y is not the only time variable that is within the scope of $y.$ in $y.\varphi(x, y)$. However, both x and y are independent in $x.(\psi(x) \vee \Diamond y.\varphi(y))$.

Now we explain why we focus on Raskin's semantics in our monitoring algorithm. In Raskin's semantics, each time constraint contains a single time variable (see Definition 3.2.5). However, in Alur's semantics each time constraint contains two time variables [11]. In Alur's semantics, time variables in the same constraint are dependent to each other. As a result, in order to benefit from independent time variables, we should consider Raskin's semantics.

Definition 8.2.2 (Encapsulated TPTL formula) *Encapsulated TPTL formulas are TPTL formulas where all the time variables are independent.*

In other words, an encapsulated formula is a closed formula in which every sub-formula has at most one free time variable.

Definition 8.2.3 (Frozen Subformula) *Given an encapsulated TPTL formula Φ, a frozen subformula ϕ of Φ is a subformula which is bounded by a freeze quantifier corresponding to (an independent) time variable.*

In encapsulated formulas, all the closed subformulas are frozen. For example the formula $x.(\psi(x) \vee \Diamond y.\varphi(x,y))$ is not an "encapsulated" formula because $y.\varphi(x,y)$ is not frozen since x, y are not independent. Here are two TPTL formulas φ_1,φ_2 that look similar but only one of them is encapsulated.

- $\varphi_1 = \Box x.\Diamond(a \wedge x \leq 10 \wedge y.\Box(\mathbf{y} \leq \mathbf{2} \wedge y \geq 1 \wedge b))$
- $\varphi_2 = \Box x.\Diamond(a \wedge x \leq 10 \wedge y.\Box(\mathbf{x} \leq \mathbf{2} \wedge y \geq 1 \wedge b))$

In the above, φ_1 is encapsulated, but φ_2 is not encapsulated since $y.\Box(x \leq 2 \wedge y \geq 1 \wedge b)$ where $x \leq 2$ is inside the scope of "$y.$".

Lemma 8.2.1 *Any MTL formula can be represented by an "encapsulated" TPTL formula.*

Problem 6 *Each time interval of an MTL temporal operator can be represented with a unique time variable which is independent of the rest of time variables. The syntactic modification works as follows: every MTL formula of the form $\varphi = \psi U_{[l,u]}\phi$ can be recursively represented as the following TPTL formula $\varphi = x.(\psi U(x \geq l \wedge x \leq u \wedge \phi))$. The resulting TPTL formula is encapsulated.*

Lemma 8.2.2 *MTL is less expressive than "encapsulated" TPTL formulas.*

Proof 8.2.1 *It is proven in [27] that the following TPTL formula, which is evidently encapsulated, cannot be expressed by any MTL formula [27]: $\psi = x.\Diamond(a \wedge x \leq 1 \wedge \Box(x \leq 1 \rightarrow \neg b))$*

In the rest of the chapter, we focus on the following problem:

Problem 7 *Given a finite ATSS $\hat{\rho}$ and an "encapsulated" TPTL formula φ, check whether $\hat{\rho}$ satisfies φ ($\hat{\rho} \models \varphi$).*

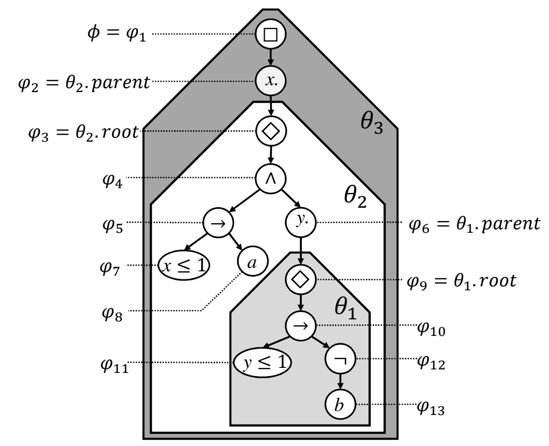

Figure 8.1: Binary Tree of Example 8.3.1 (ϕ) with Three Subtrees Corresponding to Sets of Subformulas $\theta_1, \theta_2, \theta_3$.

8.3 Solution Overview

In the following, we will describe the data structure that will be utilized to capture the solution for the TPTL monitoring problem. We store each TPTL formula in a binary tree data structure. Consider the following example:

Example 8.3.1 *Assume $AP = \{a, b\}$ and let*

$\phi = \Box x.\Diamond((x \leq 1 \rightarrow a) \wedge y.\Diamond(y \leq 1 \rightarrow \neg b))$

$\phi \equiv \Box x.\Diamond((x \leq 1 \rightarrow a) \wedge y.\psi_1(y)) \equiv \Box x.\psi_2(x)$

where we use ψ_1 and ψ_2 to simplify the presentation:

$\psi_1(y) \equiv \Diamond(y \leq 1 \rightarrow \neg b)$

$\psi_2(x) \equiv \Diamond((x \leq 1 \rightarrow a) \wedge y.\psi_1(y))$

In this example, we have two independent time variables x and y. The binary tree of Example 8.3.1 is depicted in Figure 8.1. There, the thirteen nodes correspond to thirteen subformulas.

In Figure 8.1, each subformula φ_i has a node corresponding to the highest operator for φ_i. In addition, for each subformula φ_i we assign an index i. The order of indexes is generated according to a topological sort where parents have lower index values than children. Therefore, the original subformula ϕ obtains the index 1 because it is the first visited. To evaluate each node's $\top/\bot$ value we need to evaluate its children's $\top/\bot$ value before, this is because of the TPTL recursive semantics (see Definition 3.2.6). If we evaluate the nodes in the decreasing order of indexes, we would be able to evaluate all the children before their parents.

Now, we must partition the formula tree into subtrees rooted by the freeze time operators. Since in Example 8.3.1, we have two independent time variables, we created 2+1 subtrees (two for time variables and one for the original formula). Each subtree contains a set of subformulas. These subformulas and their corresponding subtrees $\theta_1, \theta_2, \theta_3$ are shown in Figure 8.1 with different colors:

The set θ_1 contains subformulas rooted at node φ_9 represented in the **light-gray** subtree. The set θ_1 contains the subformulas of $y.\psi_1(y)$ as follows $\theta_1 = \{\Diamond(y \leq 1 \rightarrow \neg b), y \leq 1 \rightarrow \neg b, y \leq 1, \neg b, b\} = \{\varphi_9, \varphi_{10}, \varphi_{11}, \varphi_{12}, \varphi_{13}\}$.

The set θ_2 contains subformulas rooted at node φ_3 represented in the **white** subtree. The set θ_2 contains the subformulas of $x.\psi_2(x)$ as follows $\theta_2 = \{\Diamond((x \leq 1 \rightarrow a) \wedge y.\psi_1(y)), (x \leq 1 \rightarrow a) \wedge y.\psi_1(y), (x \leq 1 \rightarrow a), y.\psi_1(y), x \leq 1, a\} = \{\varphi_3, \varphi_4, \varphi_5, \varphi_6, \varphi_7, \varphi_8\}$.

The set θ_3 contains subformulas rooted at node φ_1 represented in **dark-gray** subtree. The set θ_3 contains the subformulas of $\theta_3 = \{\Box x.\psi_2(x)\ , x.\psi_2(x)\} = \{\varphi_1, \varphi_2\}$.

Each of the subtrees θ_1 and θ_2 have distinguished fields referencing to (the index of) *parent* and *root* nodes which are represented in Figure 8.1 as follows:

1) $\theta_1.parent = 6$ and $\theta_1.root = 9$.

2) $\theta_2.parent = 2$ and $\theta_2.root = 3$.

Note that θ_1 is subformula of θ_2, and θ_2 is subformula of θ_3. This ordering is very important for our algorithm. We created these subtrees because each frozen subformula can be separately evaluated. Therefore, we can guarantee the polynomial runtime. The method will be described in details in Section 8.5.

8.4 Monitoring Table

We assume that the sampled system output is mapped (projected) on a *finite* ATSS $\hat{\rho}$; therefore, we can evaluate the system output using our off-line monitor. If the specification does not have a freeze time operator, then the formula is an LTL formula for which the existing monitoring algorithms will be utilized [87]. If the specification has a freeze time operator, we first "instantiate" the time variable with the time label of the current sample before formula evaluation. Then, we compute $\perp/\top$ values of the corresponding time constraints. When time constraints are evaluated, they will be resolved to $\perp/\top$, and then, the frozen subformula ($x.\varphi(x)$) is converted into an LTL formula. Hence, we can apply dynamic programming method [87] to compute the Boolean value of the frozen subformula.

For each frozen subformula ($x.\varphi(x)$) at each time instance τ_i, we must first precompute the Boolean ($\perp/\top$) value of the corresponding time constraints to transform this frozen subformula into an LTL. A two-dimensional matrix $M_{|\phi|\times|\hat{\rho}|}$ with height (number of rows) $|\phi|$, and width (number of columns) $|\hat{\rho}|$ is created. Here $|\phi|$ denotes the number of subformulas in ϕ, and $|\hat{\rho}|$ is the number of samples. Note that row indexing starts from 1 ($\phi \equiv \varphi_1$) up to $|\phi|$ and column indexing starts from 0 (ρ_0) up to $|\hat{\rho}| - 1$.

The monitoring table of Example 8.3.1 is presented in Table 8.1. At the beginning, the system outputs corresponding to atomic propositions ($AP = \{a, b\}$) are stored in the rows which belong to the propositions a (row φ_8) and b (row φ_{13}) in Table 8.1. In Figure 8.1, the subformula $\psi_2(x)$ is depicted inside the **white** subtree and $\psi_1(y)$ is depicted inside the **light-gray** subtree. In the following, we explain the other rows of Table 8.1 and provide a high level overview of the monitoring of ϕ:

1st Run) We first instantiate time variable y at each sample i with the corresponding timed instance τ_i to evaluate the Boolean values for the corresponding time constraint $y \leq 1$ (row φ_{11}). The instantiation transforms $y.\psi_1(y)$ into an LTL formula. Then we compute the

Boolean values of $\psi_1(\tau_0)$, $\psi_1(\tau_1)$, $\psi_1(\tau_2)$, $\ldots$, $\psi_1(\tau_6)$ from left to right. Now the Boolean value of $y.\psi_1(y)$ for each time stamp τ_i is available for the higher level subtree of the Table 8.1. Therefore, the Boolean values should be copied from row φ_9 to row φ_6.

2nd Run) Given the $\perp/\top$ values of $y.\psi_1(y)$, we can instantiate x at each time stamp τ_i and modify formula $x.\psi_2(x)$ into an LTL formula. Then we compute the Boolean values of $\psi_2(\tau_0)$, $\psi_2(\tau_1)$, $\psi_2(\tau_2)$, $\ldots$, $\psi_2(\tau_6)$ from left to right. Now the Boolean values of $x.\psi_2(x)$ are available for each time stamp τ_i for the higher subtree. As a result, the $\perp/\top$ values should be copied from row φ_3 to row φ_2.

3rd Run) The Boolean value of $\Box x.\psi_2(x)$ is computed given the Boolean values of $\psi_2(\tau_i)$ according to the semantics of Always ($\Box$) operator:

$$\phi \equiv \bigwedge_{i=0}^{6} \psi_2(\tau_i)$$

Table 8.1: The Monitoring Table of Formula ϕ of Example 8.3.1 (Figure 8.1)

φ_i(OP)	τ_0	τ_1	τ_2	τ_3	τ_4	τ_5	τ_6
$\varphi_1(\Box)$	$\{\bot/\top\}$						
$\varphi_2(x.)$	$\psi_2(0)$	$\psi_2(\tau_1)$	$\psi_2(\tau_2)$	$\psi_2(\tau_3)$	$\psi_2(\tau_4)$	$\psi_2(\tau_5)$	$\psi_2(\tau_6)$
$\varphi_3(\Diamond)$	$\psi_2(0)$	$\psi_2(\tau_1)$	$\psi_2(\tau_2)$	$\psi_2(\tau_3)$	$\psi_2(\tau_4)$	$\psi_2(\tau_5)$	$\psi_2(\tau_6)$
$\varphi_4(\wedge)$							
$\varphi_5(\rightarrow)$							
$\varphi_6(y.)$	$\psi_1(0)$	$\psi_1(\tau_1)$	$\psi_1(\tau_2)$	$\psi_1(\tau_3)$	$\psi_1(\tau_4)$	$\psi_1(\tau_5)$	$\psi_1(\tau_6)$
$\varphi_7(x \leq 1)$							
$\varphi_8(a)$							
$\varphi_9(\Diamond)$	$\psi_1(0)$	$\psi_1(\tau_1)$	$\psi_1(\tau_2)$	$\psi_1(\tau_3)$	$\psi_1(\tau_4)$	$\psi_1(\tau_5)$	$\psi_1(\tau_6)$
$\varphi_{10}(\rightarrow)$							
$\varphi_{11}(y \leq 1)$							
$\varphi_{12}(\neg)$							
$\varphi_{13}(b)$							

8.5 TPTL Monitoring Algorithm

The algorithms has the main following steps.

1. For each time variable (frozen subformula) and for each time stamp.
2. Resolve the time constraints into $\perp/\top$ values (This step converts the corresponding frozen subformula into an LTL formula).
3. Compute $\perp/\top$ value of the resulting LTL formula using the dynamic programming algorithm.
4. These $\perp/\top$ values of frozen subformula are used to evaluate the higher level subformulas.

In the following, a detailed description and pseudo code of the proposed algorithm for TPTL monitoring will be explained.

8.5.1 TPTL to LTL Transformation

The pseudo code of the monitoring algorithm is provided in Algorithm 11 and its main loop has $|V| + 1$ iterations where $|V|$ is the number of freeze time variables. Algorithm 11 calls Algorithm 12 for computing the Boolean value of LTL subformulas. The first line of Algorithm 11 sets the monitoring table entries of the corresponding atomic propositions, namely the Boolean value of each $p \in AP$ is extracted from the finite state sequence $\hat{\mathbf{y}}$. In addition, Line 1 sets the monitoring table entries for constant boolean values $\perp/\top$. For each time variable v_k (in Line 2), we need to compute the $\perp/\top$ value of the subtree θ_k. The order of k is in such away that the inner most subtree (θ_1) is evaluated first then θ_2, and finally, θ_3 (See Fig 8.1 for Example 8.3.1). This order is crucial for the correctness of the algorithm, because higher level subformulas consider the lower level frozen subformulas as $\perp/\top$.

To transform the frozen formula into LTL for each sample time t between 0 to $|\hat{\rho}| - 1$ (see Line 3), we must first instantiate the time variable v_k to the corresponding time stamp τ_t, then compute the Boolean value of the corresponding time constraint $v_k \sim r$. The instantiation evaluates the whole constraint row into $\bot/\top$ in Lines 4-13 of Algorithm 11. The environment is updated based on the time stamp τ_t and the formula translated into an LTL formula. Now we use a dynamic programming algorithm based on [87] to compute the $\bot/\top$ value of the frozen subformula in Lines 14-18. In Line 15 of Algorithm 11, $\theta_k.max$ ($\theta_k.min$) is the maximum (minimum) index of subformulas in the subtree θ_k. In Example 8.3.1:

1) $\theta_1.min = 9$ and $\theta_1.max = 13$

2) $\theta_2.min = 3$ and $\theta_2.max = 8$

When the Boolean value of the frozen subformula of $v_k.\psi(v_k)$ ($\theta_k.root$) at time stamp $v_k = \tau_t$ is resolved, this Boolean value is copied to the parent of θ_k ($\theta_k.parent$) to be used by higher level subformulas (see Line 19 of Algorithm 11). The loop of Line 3-20 continues for the other time stamps ($\tau_1 \ldots \tau_{|\hat{\rho}|-1}$) and computes the $\bot/\top$ value of the frozen subformula for each instantiation of v_k to the time stamps $\tau_1 \ldots \tau_{|\hat{\rho}|-1}$ in this order. Now we resolved the $\bot/\top$ value of the frozen subformula of $v_k.\psi(v_k)$ for all time stamps. We continue this process for other time variables (Lines 2-21).

When the Boolean values of the frozen subformulas are resolved for each time variable $v_1 \ldots v_k \ldots v_{|V|}$ in this order, we have an LTL formula for the highest level subformula where it corresponds to subtree $\theta_{|V|+1}$. To compute the $\bot/\top$ value of the highest set of subformulas we run Lines 22-26 of Algorithm 11. Note that Lines 22-26 are almost identical to Lines 14-18 because the highest set of subformulas is in LTL. The final value that corresponds to the monitoring trace is stored in table entry $M[1, 0]$ and it will be returned to the user. The table entry $M[1, 0]$ contains the Boolean value of the TPTL specification (φ_1) at sampled index 0.

8.5.2 LTL Monitoring

Now we explain how to compute the Boolean values of the LTL subtree. Algorithm 12 is based on [87], and follows Definition 3.2.6. Algorithm 11 calls Algorithm 12 at each sample u. Algorithm 12 has the following 5 cases to compute the Boolean values of the corresponding LTL operators:

1. Lines 1-2 for the NOT operation ($\neg$).
2. Lines 3-4 for the AND operation ($\wedge$).
3. Lines 5-6 for the OR operation ($\vee$).
4. Lines 7-12 for the NEXT operation ($\bigcirc$).
5. Lines 13-19 for the UNTIL operation (U).

Note that Algorithm 12 (ComputeLTL) is $O(1)$ complexity. Since we can evaluate each frozen subformula ($x.\varphi(x)$) separately because of independent time variables, the time complexity of the algorithm is proportional to the number of time variables and the size of the subformula. On the other hand, for each time sample we instantiate each time variable to convert the TPTL subformula into an LTL subformula in $O(|\hat{\rho}|)$ then run the LTL monitoring algorithm in $O(|\hat{\rho}|)$. As a result, the upper bound on the time complexity of Algorithm 11 is $O(|V| \times |\varphi| \times |\hat{\rho}|^2)$, where $|V|$ is the number of time variables, $|\varphi|$ is the number of subformulas, and $|\hat{\rho}|$ is the number of ATSS samples. Both algorithms' correctness proofs are provided in Appendix D.

8.5.3 Running example

In this section, we utilize our monitoring algorithm to compute the solution for Example 8.3.1. First step of the algorithm is the $\top/\bot$ computation of the frozen subformula $y.\psi_1(y)$

which corresponds to subtree θ_1 and is represented in **light-gray** rows of Tables 8.1 and 8.2. In Table 8.2, when the time value of y is instantiated to 0, then the value of the time constraint $y \leq 1$ will be resolved for all the samples of i between 0 to 6 according to the following inequality $\tau_i - 0 \leq 1$. Now $\psi_1(0)$ is transformed into LTL and $\psi_1(0)$ is evaluated, i.e., $\psi_1(0) \equiv \top$ (see row φ_9 column τ_0). Then, the time value of y is instantiated to $\tau_1 = 0.3$ and the value of the time constraint $y \leq 1$ will be resolved for all the samples of i between 1 to 6 according to the following inequality $\tau_i - 0.3 \leq 1$. Similarly, $\psi_1(0.3)$ is transformed into LTL and $\psi_1(0.3)$ can be computed, i.e., $\psi_1(0.3) \equiv \top$ (see row φ_9 column τ_1). We continue the computation of $\psi_1(\tau_i)$ with the following instantiation $\tau_2 = 0.7, \ldots, \tau_6 = 1.9$ similar to τ_0. Now $\perp/\top$ values of the frozen subformula $y.\psi_1(y)$ for each time stamp τ_i are available in row φ_9 of Table 8.2.

The Boolean values of subtree θ_1 should be available for higher level subformulas. Therefore, the row φ_9 will be copied to row φ_6 (in Table 8.2 both rows have the same color). Now we can continue the second run of the algorithm. The $\top/\perp$ computation of the frozen subformula $x.\psi_2(x)$ which corresponds to subtree θ_2 is represented in **white** rows of Table 8.1 and 8.2. In Table 8.2, the time value of x is instantiated to 0, then the value of $\psi_2(0)$ is computed, i.e., $\psi_2(0) \equiv \top$ (see row φ_3 column τ_0). Now, the time value of x is instantiated to $\tau_1 = 0.3$ and the value of $\psi_2(0.3)$ is computed $\psi_2(0.3) \equiv \top$ (see row φ_3 column τ_1). We continue the computation of $\psi_2(\tau_i)$ similarly with $\tau_2 = 0.7 \ldots \tau_6 = 1.9$. Now the $\perp/\top$ values of the frozen subformula $x.\psi_2(x)$ for each time stamp τ_i are available in row φ_3 of Table 8.2. Since the Boolean values of subtree θ_2 should be available for higher level subformulas, the row φ_3 is copied to row φ_2. Finally, we compute $\phi = \Box x.\psi_2(x)$ using Lines 22-26 of Algorithm 11 which corresponds to following: $\phi = \bigwedge_{i=0}^{6} \psi_2(\tau_i) \equiv \perp$

Algorithm 11 TPTL Monitor

Input: $\varphi, \hat{\rho} = (\tilde{\mathbf{y}}_0, \tau_0)(\tilde{\mathbf{y}}_1, \tau_1) \cdots (\tilde{\mathbf{y}}_T, \tau_T)$; **Global variables:** $M_{|\varphi| \times |\hat{\rho}|}$; **Output**: $M[1, 0]$.

```
procedure TPTLMonitor($\varphi, \hat{\rho}$)
1:  Init rows of $M_{|\varphi| \times |\hat{\rho}|}$ correspond to predicates $\varphi_j \equiv p \in AP$ with $\top/\bot$ according to $\hat{\tilde{\mathbf{y}}}$.
2:  for $k \leftarrow 1$ to $|V|$ do
3:      for $t \leftarrow 0$ to $|\hat{\rho}| - 1$ do
4:          for $u \leftarrow t$ to $|\hat{\rho}| - 1$ do
5:              for each $\varphi_j \equiv v_k \sim r \in \theta_k$ where
6:                  $j$ is the index of $v_k \sim r$ in $M$ do
7:                  if $(\tau_u - \tau_t) \sim r$ then
8:                      $M[j, u] \leftarrow \top$
9:                  else
10:                     $M[j, u] \leftarrow \bot$
11:                 end if
12:             end for
13:         end for
14:         for $u \leftarrow |\hat{\rho}| - 1$ down to $t$ do
15:             for $j \leftarrow \theta_k.max$ down to $\theta_k.min$ do
16:                 $M[j, u] \leftarrow ComputeLTL(\varphi_j, u, M_{|\varphi| \times |\hat{\rho}|})$
17:             end for
18:         end for
19:         $M[\theta_k.parent, t] \leftarrow M[\theta_k.root, t]$
20:     end for
21: end for
22: for $u \leftarrow |\hat{\rho}| - 1$ down to 0 do
23:     for $j \leftarrow \theta_{|V|+1}.max$ down to $\theta_{|V|+1}.min$ do
24:         $M[j, u] \leftarrow ComputeLTL(\varphi_j, u, M_{|\varphi| \times |\hat{\rho}|})$
25:     end for
26: end for
27: return $M[1, 0]$ // Return the value of the first cell/row in $M_{|\varphi| \times |\hat{\rho}|}$ table
end procedure
```

Algorithm 12 LTL Monitor

Input: $\varphi_j, u, M_{|\varphi|\times|\hat{\rho}|}$; **Output**: $M[j, u]$.

```
procedure ComputeLTL($\varphi_j, u, M_{|\varphi|\times|\hat{\rho}|}$)
1:  if $\varphi_j \equiv \neg\varphi_m$ then
2:      return $\neg M[m, u]$
3:  else if $\varphi_j \equiv \varphi_m \wedge \varphi_n$ then
4:      return $M[m, u] \wedge M[n, u]$
5:  else if $\varphi_j \equiv \varphi_m \vee \varphi_n$ then
6:      return $M[m, u] \vee M[n, u]$
7:  else if $\varphi_j \equiv \bigcirc\varphi_m$ then
8:      if $u = |\hat{\rho}| - 1$ then
9:          return $\bot$
10:     else
11:         return $M[m, u + 1]$
12:     end if
13: else if $\varphi_j \equiv \varphi_m U \varphi_n$ then
14:     if $u = |\hat{\rho}| - 1$ then
15:         return $M[n, u]$
16:     else
17:         return $M[n, u] \vee (M[m, u] \wedge M[j, u + 1])$
18:     end if
19: end if
end procedure
```

Table 8.2: Computing the Boolean Values for $\phi = \Box x.\psi_2(x)$. Boolean Values Correspond to the Final Snapshot of Monitoring Table.

φ_i(OP)	$\tau_0 = 0$	$\tau_1 = 0.3$	$\tau_2 = 0.7$	$\tau_3 = 1.0$	$\tau_4 = 1.1$	$\tau_5 = 1.5$	$\tau_6 = 1.9$
$\varphi_1(\Box)$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$
$\varphi_2(x.)$	$\psi_2(0) \equiv \top$	$\psi_2(\tau_1) \equiv \top$	$\psi_2(\tau_2) \equiv \top$	$\psi_2(\tau_3) \equiv \top$	$\psi_2(\tau_4) \equiv \bot$	$\psi_2(\tau_5) \equiv \bot$	$\psi_2(\tau_6) \equiv \bot$
$\varphi_3(\Diamond)$	$\top$	$\top$	$\top$	$\top$	$\bot$	$\bot$	$\bot$
$\varphi_4(\wedge)$	$\bot$	$\bot$	$\top$	$\top$	$\bot$	$\bot$	$\bot$
$\varphi_5(\rightarrow)$	$\bot$	$\bot$	$\top$	$\top$	$\top$	$\bot$	$\bot$
$\varphi_6(y.)$	$\psi_1(0) \equiv \top$	$\psi_1(\tau_1) \equiv \top$	$\psi_1(\tau_2) \equiv \top$	$\psi_1(\tau_3) \equiv \top$	$\psi_1(\tau_4) \equiv \bot$	$\psi_1(\tau_5) \equiv \bot$	$\psi_1(\tau_6) \equiv \bot$
$\varphi_7(x \leq 1)$	$\top$	$\top$	$\top$	$\top$	$\top$	$\top$	$\top$
$\varphi_8(a)$	$\bot$	$\bot$	$\top$	$\top$	$\top$	$\bot$	$\bot$
$\varphi_9(\Diamond)$	$\top$	$\top$	$\top$	$\top$	$\bot$	$\bot$	$\bot$
$\varphi_{10}(\rightarrow)$	$\top$	$\top$	$\bot$	$\top$	$\bot$	$\bot$	$\bot$
$\varphi_{11}(y \leq 1)$	$\top$	$\top$	$\top$	$\top$	$\top$	$\top$	$\top$
$\varphi_{12}(\neg)$	$\top$	$\top$	$\bot$	$\top$	$\bot$	$\bot$	$\bot$
$\varphi_{13}(b)$	$\bot$	$\bot$	$\top$	$\bot$	$\top$	$\top$	$\top$

8.6 Experimental Results

An implementation of our TPTL monitoring algorithm is provided in the S-TaLiRo testing framework [64]. S-TaLiRo is a Matlab toolbox that uses stochastic techniques to find initial states and inputs to Simulink models which result in trajectories that falsify MTL formulas. With our TPTL off-line monitoring algorithm, S-TaLiRo can evaluate specifications that are more expressive than MTL. We measured the runtime of our TPTL monitoring algorithm using the S-TaLiRo toolbox. The system under test was the Automatic Transmission (AT) model provided in Section 2.3. We introduced a few modifications to the model to make it compatible with the S-TaLiRo framework, which are explained in [63]. AT has two inputs of Throttle and Brake. The outputs contain two real-valued traces: the rotational speed of the engine ω and the speed of the vehicle v. In addition, the outputs contain one discrete-valued trace *gear* with four possible values.

To provide TPTL specifications, we defined four atomic propositions corresponding to the following predicates:

1) $a_1 \equiv (\omega \geq 4500)$: "rotational speed of the engine $\geq$ 4500"

2) $a_2 \equiv (\omega \leq 1500)$: "rotational speed of the engine $\leq$ 1500"

3) $a_3 \equiv (v \geq 40)$: "speed of the vehicle $\geq$ 40"

4) $a_4 \equiv (v \leq 120)$: "speed of the vehicle $\leq$ 120"

Note that these predicates are chosen to be non-trivial and have meaning in the CPS context. The TPTL formulas are generated based on typical safety Request-Response specifications. We generated these TPTL formula patterns to check the runtime with respect to: 1) Size of system trace 2) Number of temporal operators 3) Number of time variables.

We created 18 TPTL formulas that cannot be expressed in MTL. All the specifications have the Request-Response pattern: $\Box(a_1 \rightarrow \psi)$ where ψ is categorized in two groups:

1. EA group (ψ_{EA}): contains Eventually/Always specifications with 2, 4 and 8 temporal

operators.

2. UR group (ψ_{UR}): contains Until/Release specifications with 2, 4 and 8 temporal operators.

We first chose a ψ specification in LTL from Table 8.3 column (LTL template). In Table 8.3, column (#) represents the number of temporal operators for each LTL template. Then, we added time variables to create a TPTL specification. The last column in Table 8.3 represents the number of TPTL formulas that we created by adding time constraints on ψ. The time variables that we add to ψ correspond to individual temporal operators. In this case, for ψ_{EA2} we create two TPTL formulas with one and two time variables respectively given as ϕ_1 and ϕ_2:

EA $\phi_1 = \Box(a_1 \rightarrow x.\Diamond(a_2 \wedge \Box(a_3 \vee a_4 \wedge C_x)))$

EA $\phi_2 = \Box(a_1 \rightarrow x.\Diamond(a_2 \wedge C_x \wedge y.\Box(a_3 \vee a_4 \wedge C_y)))$

where C_x and C_y are the corresponding time constraints for x and y. Similarly for ψ_{UR2} we created two TPTL formulas with one and two time variables respectively given as ϕ_1 and ϕ_2:

UR $\phi_1 = \Box(a_1 \rightarrow x.(a_2 U(a_3 R(a_4 \wedge C_x))))$

UR $\phi_2 = \Box(a_1 \rightarrow x.(a_2 U a_4 \wedge C_x \wedge y.(a_3 R(a_4 \wedge C_y)))$

We used a similar method to generate ϕ_3 with one time variable, ϕ_4 with two time variables, and ϕ_5 with four time variables based on ψ_{EA4} and ψ_{UR4} with the total number of six TPTL formulas. Finally, we create eight TPTL formulas based on ψ_{EA8} and ψ_{UR8}. These formulas are ϕ_6, ϕ_7, ϕ_8, ϕ_9 and they are represented in Table 8.4. Our experiments were conducted on a 64-bit Intel Xeon CPU (2.5GHz) with 64-GB RAM and Windows Server 2012. We used

Matlab 2015a and Microsoft Visual C++ 2013 Professional to compile our algorithms' code (in C) using the Matlab mex compiler.

The runtime is provided in Table 8.4. Each row considers two TPTL formulas in EA or UR configuration. For example, the first column ϕ_1 represents $\Box(a_1 \rightarrow x.\Diamond(a_2 \wedge \Box(a_3 \vee a_4 \wedge C_x)))$ and $\Box(a_1 \rightarrow x.(a_2 U(a_3 R(a_4 \wedge C_x))))$ in EA and UR configurations, respectively. In Table 8.4 the second column (#) represents the number of temporal operators in the corresponding frozen subformula, namely, the number of of temporal operators in $\psi_{EA\#}$ or $\psi_{UR\#}$. The third column ($|V|$) in Table 8.4 represents the number of time variables in $\psi_{EA\#}$ or $\psi_{UR\#}$.

We tested our algorithm with the execution traces of the length 1000, 2000, and 10000. For each TPTL formula, we tested our algorithm 100 times where the AT's throttle input is provided by random signal generator (without brake). We reported the mean value (in **Bold**) and variance of the algorithm's runtime in Table 8.4. It can be seen that when the length of the trace doubles from $|\hat{\rho}|$=1,000 to $|\hat{\rho}|$=2,000 , the runtime quadruples (see **Mean** values in Table 8.4). Similarly, when the length of trace increases ten times from $|\hat{\rho}|$=1,000 to $|\hat{\rho}|$=10,000 the runtime increased 100 times (see **Mean** values in Table 8.4). Now, consider the mean values of ϕ_1 and ϕ_2. The number of time variables in ϕ_1 is one and in ϕ_2 is two. It can be seen that mean values of ϕ_2 are twice as those of ϕ_1. Similarly, comparing ϕ_3 and ϕ_4 and ϕ_5 shows that the runtime is proportional to the number of time variables. Finally, comparing rows ϕ_1 and ϕ_3 and ϕ_6 shows that the runtime relates to the number of temporal operators. The experimental results indicate that the runtime behaves as expected, considering that our algorithm is in $O(|V| \times |\varphi| \times |\hat{\rho}|^2)$.

Table 8.3: Specifications of ψ Before Adding Time Variables.

LTL	#	LTL template	TPTLs
ψ_{EA2}	2	$\Diamond(a_2 \wedge \Box(a_3 \vee a_4)$	2
ψ_{EA4}	4	$\Diamond(a_2 \wedge \Box(a_3 \vee a_4 \wedge \psi_{EA2})$	3
ψ_{EA8}	8	$\Diamond(a_2 \wedge \Box(a_3 \vee a_4 \wedge \Diamond(a_2 \wedge \Box(a_3 \vee a_4 \wedge \psi_{EA4}))))$	4
ψ_{UR2}	2	$a_2 U(a_3 R a_4)$	2
ψ_{UR4}	4	$a_2 U(a_3 R(a_4 \wedge \psi_{UR2}))$	3
ψ_{UR8}	8	$a_2 U(a_3 R(a_4 \wedge (a_2 U(a_3 R(a_4 \wedge \psi_{UR4})))))$	4

Table 8.4: The Runtime of Monitoring Algorithm for 18 TPTL Formulas. All the Values Are in Seconds.

			$\lvert\hat{\rho}\rvert$=1,000				$\lvert\hat{\rho}\rvert$=2,000				$\lvert\hat{\rho}\rvert$=10,000			
			EA ($\psi_{EA\#}$)		UR ($\psi_{UR\#}$)		EA ($\psi_{EA\#}$)		UR ($\psi_{UR\#}$)		EA ($\psi_{EA\#}$)		UR ($\psi_{UR\#}$)	
ϕ	#	$\lvert V\rvert$	Mean	Var.	Mean	Var.	Mean	Var.	Mean	Var.	Mean	Var.	**Mean**	Var.
ϕ_1	2	1	0.077	0.0002	0.064	0.000	0.326	0.001	0.250	0.0013	8.512	0.066	6.427	0.068
ϕ_2	2	2	0.151	0.0005	0.137	0.0003	0.5887	0.0018	0.551	0.002	14.31	0.191	13.67	0.175
ϕ_3	4	1	0.142	0.0003	0.097	0.0001	0.5885	0.002	0.382	0.002	15.33	0.232	10.46	0.154
ϕ_4	4	2	0.205	0.0003	0.15	0.0002	0.871	0.0032	0.604	0.002	22.9	0.344	16.35	0.24
ϕ_5	4	4	0.417	0.0012	0.38	0.0004	1.721	0.0058	1.558	0.007	46.25	7.08	41.2	1.077
ϕ_6	8	1	0.227	0.0001	0.154	0.0002	0.948	0.005	0.552	0.0046	30.27	9.708	17.01	2.184
ϕ_7	8	2	0.367	0.025	0.235	0.0011	1.474	0.0078	1.023	0.0137	41.59	2.17	26.95	2.204
ϕ_8	8	4	0.533	0.0042	0.437	0.0013	2.26	0.024	1.751	0.0115	66.13	34.36	48.95	8.857
ϕ_9	8	8	1.145	0.025	1.093	0.0066	4.9	0.0391	4.346	0.1413	137	220	124.6	184

8.7 Case Study

In this section, we consider CPS requirements which are impossible to formalize in MTL [27], but we formalize them in TPTL, very easily. The ultimate goal is to run the testing algorithm on these requirements. Our TPTL monitoring algorithm is provided as add-on to the S-TaLiRo testing framework. S-TaLiRo searches for counterexamples to MTL properties through global minimization of a robustness metric [48]. The robustness of an MTL formula φ is a value that measures how far is the trace from the satisfaction/falsification of φ. This measure is an extension of Boolean values ($\top/\bot$) for representing satisfaction or falsification. A positive robustness value means that the trace satisfies the property and a negative value means that the property is not satisfied. The stochastic search then returns the simulation trace with the smallest robustness value that was found.

To falsify safety requirements in TPTL which are more expressive than MTL, we should use our proposed TPTL monitor that can handle those specifications. Now let us consider the Automatic Transmission (AT) system. It contains the discrete output *gear* signal with four possible values ($gear = 1$, ..., $gear = 4$) which indicate the current gear in the auto-transmission controller. We use four atomic propositions g_1, g_2, g_3, g_4 for each possible gear value, where $(gear = i) \equiv g_i$. Then we define three up-shifting events as follows:
1) $e_1 = g_1 \wedge \bigcirc g_2$ means shift from gear one to gear two.
2) $e_2 = g_2 \wedge \bigcirc g_3$ means shift from gear two to gear three.
3) $e_2 = g_3 \wedge \bigcirc g_4$ means shift from gear three to gear four.

In CPS, it is possible that we need to specify the safety requirement about three or more events in sequence, but the time difference between the first and last event happening should be of importance. In general, these types of specification are impossible to represent in MTL. We provide two very succinct TPTL specifications that can formalize these chal-

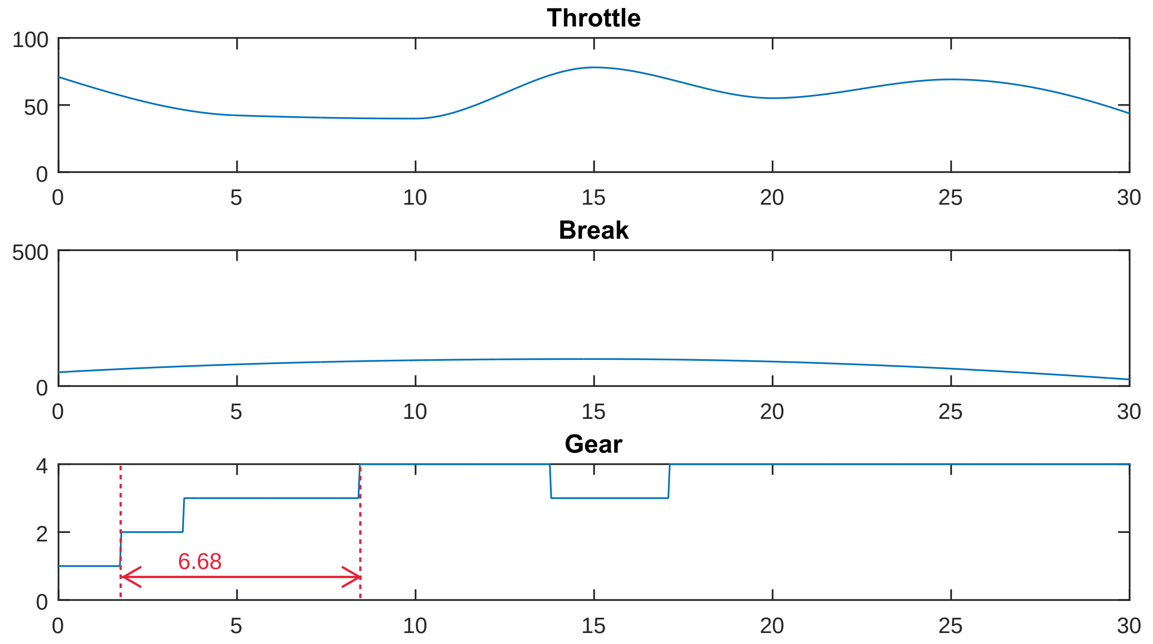

Figure 8.2: Falsification of Φ_1 Using S-TaLiRo. The Duration Between e_1 and e_3 Is Less than 8 Seconds.

lenging requirements.

The first requirement is as follows:

"*Always if e_1 happens, then if e_2 happens in future and if e_3 happens in future after e_2, then the duration between e_1 and e_3 should be equal or more than 8.*"

This specification is formalized in the following formula:

$$\Phi_1 = \Box z.(e_1 \rightarrow \Box(e_2 \rightarrow \Box(e_3 \rightarrow z \geq 8)))$$

S-TaLiRo successfully falsified Φ_1 which is represented in Figure 8.2. In Figure 8.2 the Throttle, Break, and Gear trajectory of the corresponding falsification is presented. It can be seen that the duration between e_1 and e_3 is less that 8. Its actual value is $8.4 - 1.72 = 6.68 < 8$.

The second requirement is as follows:

"*Always if e_1 happens, then e_2 should happen in future, and e_3 should happen in future after e_2, and the duration between e_1 and e_3 should be equal or less than 12.*"

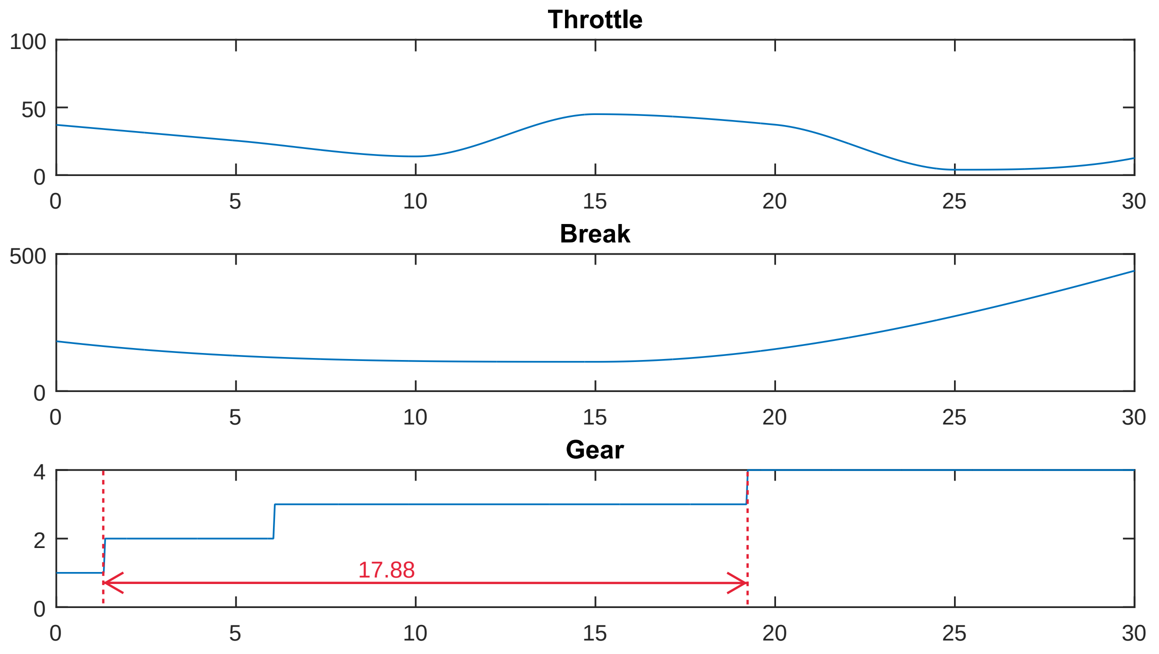

Figure 8.3: Falsification of Φ_2 Using S-TaLiRo. The Duration Between e_1 and e_3 Is More than 12 Seconds.

This specification is formalized by the following formula:

$$\Phi_2 = \Box z.(e_1 \rightarrow \Diamond(e_2 \wedge \Diamond(e_3 \wedge z \leq 12)))$$

In Figure 8.3 the Throttle, Break, and Gear trajectories of the falsification of Φ_2 are represented. It can be seen that the duration between e_1 and e_3 is more than 12, its actual value is $19.2 - 1.32 = 17.88 > 12$. This case study shows that S-TaLiRo can be used for the falsification problem of challenging TPTL requirements. The method we propose in this work opens the possibility for CPS off-line monitoring of very complex specifications in TPTL using an efficient algorithm.

8.8 Conclusions and Future Work

In this chapter, we provide an efficient polynomial time algorithm for a practical subset of TPTL specifications. We show that very complex specifications can be succinctly represented in this TPTL subset. Our method can help CPS developers to efficiently test requirements that cannot be expressed in MTL. In the future, we can combine full TPTL with a bounded number of time variables with our suggested algorithm to test the specifications that have an arbitrary number of independent time variables and full TPTL with limited number of time variables.

Chapter 9

CONCLUSIONS AND FUTURE WORKS

Nowadays, most CPS are safety-critical systems. Therefore, safety is the top priority during design, implementation and deployment of CPS. It is important to guarantee that a CPS satisfies the safety requirements. Formal verification is the best way to prove the correctness of a system with respect to its requirements. However, in general, the verification problem for CPS is an undecidable problem. To face the undecidability issues, research and development teams have worked on a number of semi-formal problems. Testing is one of the most important methods for verification and validation of CPS.

In this dissertation, we provided a number of contributions to improve testing of CPS. We first provided a debugging framework which could find the most common logical issues in formal specifications represented in MITL. Checking the logical inconsistency helps the testing teams catch fundamental issues in MITL requirements and fix them before doing any testing and monitoring on the system. Our contributions on this topic are reported in Chapter 4. In the future, our specification debugging framework can be integrated into the specification elicitation tool ViSpec as a single system. This can improve the specification elicitation framework by debugging MITL formulas on the fly and by interacting with non-expert users in a better way.

Some of the specification issues can only be detected when considering both the system and the specification. In Chapter 5, we considered signal vacuity checking for improving CPS testing. This enables improved and deeper analysis since we consider both the system and the specification. We applied this method to improve the S-TaLiRo falsification framework in Chapter 6. We showed that non-vacuous signals can improve the falsification of Request-Response requirements for the benchmarks that we tested. We considered the Request-Response specifications with only one implication. In the future, our vacuity aware falsification framework can be applied to Request-Response specifications with more than one implication operators to improve the falsification framework for more complex requirements.

One of the main parts of any CPS testing flow is the monitoring component. In the monitor section, we checked whether the system trace satisfies the requirement or not. Since monitoring of a formal specification has theoretical and practical challenges, we devoted two chapters of this dissertation to CPS monitoring. We provided an on-line monitoring component in Chapter 7, and an off-line monitoring method in Chapter 8. Both our monitoring algorithms are efficient and they are used in S-TaLiRo for testing industrial size models. In the future, we will consider the embedded implementation of our motoring components for real-time applications on physical platforms (as opposed to models).

REFERENCES

[1] H. Abbas, G. Fainekos, S. Sankaranarayanan, F. Ivančić, and A. Gupta. Probabilistic temporal logic falsification of cyber-physical systems. *ACM Trans. Embed. Comput. Syst.*, 12(2s):95:1–95:30, May 2013.

[2] H. Abbas and G. E. Fainekos. Linear hybrid system falsification through descent. *CoRR*, abs/1105.1733, 2011.

[3] H. Abbas, B. Hoxha, G. Fainekos, and K. Ueda. Robustness-guided temporal logic testing and verification for stochastic cyber-physical systems. In *Proc. of IEEE International Conference on CYBER Technology in Automation, Control, and Intelligent Systems*, 2014.

[4] T. Akazaki. Falsification of conditional safety properties for cyber-physical systems with gaussian process regression. In *Runtime Verification - 16th International Conference, RV 2016, Madrid, Spain, September 23-30, 2016, Proceedings*, pages 439–446, 2016.

[5] R. Alur. *Principles of Cyber-Physical Systems*. MIT Press, 2015.

[6] R. Alur, C. Courcoubetis, N. Halbwachs, T. A. Henzinger, P.-H. Ho, X. Nicollin, A. Olivero, J. Sifakis, and S. Yovine. The algorithmic analysis of hybrid systems. *Theoretical Computer Science*, 138(1):3–34, 1995.

[7] R. Alur, T. Feder, and T. A. Henzinger. The benefits of relaxing punctuality. In *Symposium on Principles of Distributed Computing*, pages 139–152, 1991.

[8] R. Alur, T. Feder, and T. A. Henzinger. The benefits of relaxing punctuality. *J. ACM*, 43(1):116–146, 1996.

[9] R. Alur and T. A. Henzinger. *Real-Time: Theory in Practice: REX Workshop Mook, The Netherlands, June 3–7, 1991 Proceedings*, chapter Logics and models of real time: A survey, pages 74–106. Springer Berlin Heidelberg, Berlin, Heidelberg, 1992.

[10] R. Alur and T. A. Henzinger. Real-time logics: Complexity and expressiveness. *Inf. Comput.*, 104(1):35–77, 1993.

[11] R. Alur and T. A. Henzinger. A really temporal logic. *Journal of the ACM*, 41(1):181–204, 1994.

[12] Y. S. R. Annapureddy and G. E. Fainekos. Ant colonies for temporal logic falsification of hybrid systems. In *Proceedings of the 36th Annual Conference of IEEE Industrial Electronics*, pages 91–96, 2010.

[13] Y. S. R. Annapureddy, C. Liu, G. E. Fainekos, and S. Sankaranarayanan. S-taliro: A tool for temporal logic falsification for hybrid systems. In *Tools and algorithms for the construction and analysis of systems*, volume 6605 of *LNCS*, pages 254–257. Springer, 2011.

[14] R. Armoni, L. Fix, A. Flaisher, O. Grumberg, N. Piterman, A. Tiemeyer, and M. Y. Vardi. Enhanced vacuity detection in linear temporal logic. In *Computer Aided Verification, 15th International Conference, CAV 2003, Boulder, CO, USA, July 8-12, 2003, Proceedings*, pages 368–380, 2003.

[15] E. Asarin, A. Donzé, O. Maler, and D. Nickovic. Parametric identification of temporal properties. In *Runtime Verification - Second International Conference, RV 2011, San Francisco, CA, USA, September 27-30, 2011, Revised Selected Papers*, pages 147–160, 2011.

[16] C. Baier and J. Katoen. *Principles of model checking*. MIT Press, 2008.

[17] A. Bakirtzis, V. Petridis, S. Kiartzis, M. Alexiadis, and A. Maissis. A neural network short term load forecasting model for the greek power system. *IEEE Transactions on Power Systems*, 11(2):858–863, May 1996.

[18] T. Ball and O. Kupferman. Vacuity in testing. In *Tests and Proofs, Second International Conference, TAP 2008, Prato, Italy, April 9-11, 2008. Proceedings*, pages 4–17, 2008.

[19] E. Bartocci, L. Bortolussi, L. Nenzi, and G. Sanguinetti. System design of stochastic models using robustness of temporal properties. *Theor. Comput. Sci.*, 587:3–25, 2015.

[20] D. L. Beatty and R. E. Bryant. Formally verifying a microprocessor using a simulation methodology. In *DAC*, pages 596–602, 1994.

[21] I. Beer, S. Ben-David, C. Eisner, and Y. Rodeh. Efficient detection of vacuity in temporal model checking. *Formal Methods in System Design*, 18(2):141–163, 2001.

[22] S. Ben-David, D. Fisman, and S. Ruah. Temporal antecedent failure: Refining vacuity. In *CONCUR 2007 - Concurrency Theory, 18th International Conference, CONCUR 2007, Lisbon, Portugal, September 3-8, 2007, Proceedings*, pages 492–506, 2007.

[23] M. Bersani, M. Rossi, and P. San Pietro. A tool for deciding the satisfiability of continuous-time metric temporal logic. *Acta Informatica*, pages 1–36, 2015.

[24] M. M. Bersani, M. Rossi, and P. S. Pietro. A logical characterization of timed (non-)regular languages. In *Mathematical Foundations of Computer Science 2014 - 39th International Symposium, MFCS 2014, Budapest, Hungary, August 25-29, 2014. Proceedings, Part I*, pages 75–86, 2014.

[25] M. M. Bersani, M. Rossi, and P. S. Pietro. A tool for deciding the satisfiability of continuous-time metric temporal logic. *Acta Inf.*, 53(2):171–206, 2016.

[26] M. M. Bersani, M. Rossi, and P. San Pietro. Deciding the satisfiability of mitl specifications. In Fourth International Symposium on *Games, Automata, Logics and Formal Verification,*, volume 119 of *EPTCS*, pages 64–78. Open Publishing Association, 2013.

[27] P. Bouyer, F. Chevalier, and N. Markey. On the expressiveness of TPTL and MTL. *Inf. Comput.*, 208(2):97–116, 2010.

[28] M. Chai and H. Schlingloff. A rewriting based monitoring algorithm for TPTL. In *Proceedings of the 22nd International Workshop on Concurrency, Specification and Programming, Warsaw, Poland*, pages 61–72, 2013.

[29] H. Chockler and O. Strichman. Before and after vacuity. *Form. Methods Syst. Des.*, 34(1):37–58, Feb. 2009.

[30] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. MIT Press, Cambridge, Massachusetts, 1999.

[31] E. M. Clarke, W. Klieber, M. Novácek, and P. Zuliani. Model checking and the state explosion problem. In *Tools for Practical Software Verification, LASER, International Summer School 2011, Elba Island, Italy, Revised Tutorial Lectures*, pages 1–30, 2011.

[32] S. Demri and D. D'Souza. An automata-theoretic approach to constraint LTL. *Inf. Comput.*, 205(3):380–415, 2007.

[33] J. V. Deshmukh, A. Donzé, S. Ghosh, X. Jin, G. Juniwal, and S. A. Seshia. Robust online monitoring of signal temporal logic. In *Runtime Verification - 6th International Conference, RV 2015 Vienna, Austria, September 22-25, 2015. Proceedings*, pages 55–70, 2015.

[34] J. V. Deshmukh, R. Majumdar, and V. S. Prabhu. Quantifying conformance using the skorokhod metric. In *Computer Aided Verification - 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part II*, pages 234–250, 2015.

[35] A. Dokhanchi, B. Hoxha, and G. E. Fainekos. On-line monitoring for temporal logic robustness. In *Runtime Verification - 5th International Conference, RV 2014, Toronto, ON, Canada, September 22-25, 2014. Proceedings*, pages 231–246, 2014.

[36] A. Dokhanchi, B. Hoxha, and G. E. Fainekos. Metric interval temporal logic specification elicitation and debugging. In *13. ACM/IEEE International Conference on Formal Methods and Models for Codesign, MEMOCODE 2015, Austin, TX, USA, September 21-23, 2015*, pages 70–79, 2015.

[37] A. Dokhanchi, B. Hoxha, and G. E. Fainekos. Formal requirement elicitation and debugging for testing and verification of cyber-physical systems. *CoRR*, abs/1607.02549, 2016.

[38] A. Dokhanchi, B. Hoxha, C. E. Tuncali, and G. Fainekos. An efficient algorithm for monitoring practical TPTL specifications. In *2016 ACM/IEEE International Conference on Formal Methods and Models for System Design, MEMOCODE 2016, Kanpur, India, November 18-20, 2016*, pages 184–193, 2016.

[39] A. Dokhanchi, S. Yaghoubi, B. Hoxha, and G. Fainekos. Vacuity aware falsification for MTL request-response specifications. In *Proceedings of the 13th international Conference on Automation Science and Engineering*. IEEE, 2017.

[40] A. Dokhanchi, A. Zutshi, R. T. Sriniva, S. Sankaranarayanan, and G. E. Fainekos. Requirements driven falsification with coverage metrics. In *2015 International Conference on Embedded Software, EMSOFT 2015, Amsterdam, Netherlands, October 4-9, 2015*, pages 31–40, 2015.

[41] A. Donze. Breach, a toolbox for verification and parameter synthesis of hybrid systems. In *Computer Aided Verification*, volume 6174 of *LNCS*, pages 167–170. Springer, 2010.

[42] A. Donzé, T. Ferrère, and O. Maler. Efficient robust monitoring for STL. In *Computer Aided Verification - 25th International Conference, CAV 2013, Saint Petersburg, Russia, July 13-19, 2013. Proceedings*, pages 264–279, 2013.

[43] A. Donze and O. Maler. Robust satisfaction of temporal logic over real-valued signals. In *Formal Modelling and Analysis of Timed Systems*, volume 6246 of *LNCS*. Springer, 2010.

[44] M. B. Dwyer, G. S. Avrunin, and J. C. Corbett. Property specification patterns for finite-state verification. In *Proceedings of the Second Workshop on Formal Methods in Software Practice*, FMSP '98, pages 7–15. ACM, 1998.

[45] J. Eker, J. Janneck, E. A. Lee, J. Liu, X. Liu, J. Ludvig, S. Sachs, and Y. Xiong. Taming heterogeneity - the ptolemy approach. *Proceedings of the IEEE*, 91(1):127–144, Jan. 2003.

[46] J. M. Eklund, J. Sprinkle, and S. Sastry. Implementing and testing a nonlinear model predictive tracking controller for aerial pursuit/evasion games on a fixed wing aircraft. In *American Control Conference*, 2005.

[47] G. Fainekos and G. J. Pappas. Robustness of temporal logic specifications. In *Formal Approaches to Testing and Runtime Verification*, volume 4262 of *LNCS*, pages 178–192. Springer, 2006.

[48] G. Fainekos and G. J. Pappas. Robustness of temporal logic specifications for continuous-time signals. *Theor. Comput. Sci.*, 410(42):4262–4291, 2009.

[49] G. Fainekos, S. Sankaranarayanan, K. Ueda, and H. Yazarel. Verification of automotive control applications using s-taliro. In *Proceedings of the American Control Conference*, 2012.

[50] G. Fainekos, S. Sankaranarayanan, K. Ueda, and H. Yazarel. Verification of automotive control applications using s-taliro. In *Proceedings of the American Control Conference*, 2012.

[51] A. Fehnker and F. Ivancic. Benchmarks for hybrid systems verification. In *Hybrid Systems: Computation and Control*, volume 2993 of *LNCS*, pages 326–341. Springer, 2004.

[52] S. Feng, M. Lohrey, and K. Quaas. Path checking for MTL and TPTL over data words. In *Developments in Language Theory - 19th International Conference, DLT 2015, Liverpool, UK, July 27-30, 2015, Proceedings.*, pages 326–339, 2015.

[53] B. Finkbeiner and L. Kuhtz. Monitor circuits for ltl with bounded and unbounded future. In *Runtime Verification*, volume 5779 of *LNCS*, pages 60–75. Springer, 2009.

[54] D. Fisman, O. Kupferman, S. Sheinvald-Faragy, and M. Y. Vardi. A framework for inherent vacuity. In *Hardware and Software: Verification and Testing, 4th International Haifa Verification Conference, HVC 2008, Haifa, Israel, October 27-30, 2008. Proceedings*, pages 7–22, 2008.

[55] R. Gerth, D. Peled, M. Y. Vardi, and P. Wolper. Simple on-the-fly automatic verification of linear temporal logic. In *Proceedings of the Fifteenth IFIP WG6.1 International Symposium on Protocol Specification, Testing and Verification XV*, pages 3–18, London, UK, UK, 1996.

[56] J. Håkansson, B. Jonsson, and O. Lundqvist. Generating online test oracles from temporal logic specifications. *STTT*, 4(4):456–471, 2003.

[57] K. Havelund and G. Rosu. Monitoring programs using rewriting. In *Proceedings of the 16th IEEE international conference on Automated software engineering*, 2001.

[58] K. Havelund and G. Rosu. Efficient monitoring of safety properties. *STTT*, 6(2):158–173, 2004.

[59] T. A. Henzinger. The theory of hybrid automata. In *Proceedings, 11th Annual IEEE Symposium on Logic in Computer Science, New Brunswick, New Jersey, USA, July 27-30, 1996*, pages 278–292, 1996.

[60] T. A. Henzinger, P. W. Kopke, A. Puri, and P. Varaiya. What's decidable about hybrid automata? *J. Comput. Syst. Sci.*, 57(1):94–124, 1998.

[61] H. Ho, J. Ouaknine, and J. Worrell. Online monitoring of metric temporal logic. In *Runtime Verification - 5th International Conference, RV 2014, Toronto, ON, Canada, September 22-25, 2014. Proceedings*, pages 178–192, 2014.

[62] F. Horn, W. Thomas, N. Wallmeier, and M. Zimmermann. Optimal strategy synthesis for request-response games. *RAIRO - Theor. Inf. and Applic.*, 49(3):179–203, 2015.

[63] B. Hoxha, H. Abbas, and G. Fainekos. Benchmarks for temporal logic requirements for automotive systems. In *Proc. of Applied Verification for Continuous and Hybrid Systems*, 2014.

[64] B. Hoxha, H. Bach, H. Abbas, A. Dokhanchi, Y. Kobayashi, and G. Fainekos. Towards formal specification visualization for testing and monitoring of cyber-physical systems. In *Int. Workshop on Design and Implementation of Formal Tools and Systems*. October 2014.

[65] B. Hoxha, A. Dokhanchi, and G. Fainekos. Mining parametric temporal logic properties in model-based design for cyber-physical systems. *International Journal on Software Tools for Technology Transfer*, pages 1–15, 2017.

[66] B. Hoxha, N. Mavridis, and G. Fainekos. ViSpec: a graphical tool for easy elicitation of MTL requirements. In *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems, Hamburg, Germany*, September 2015.

[67] X. Jin, A. Donze, J. Deshmukh, and S. Seshia. Mining requirements from closed-loop control models. In *Hybrid Systems: Computation and Control*. ACM Press, 2013.

[68] J. Kapinski, J. V. Deshmukh, X. Jin, H. Ito, and K. Butts. Simulation-based approaches for verification of embedded control systems: An overview of traditional and advanced modeling, testing, and verification techniques. *IEEE Control Systems Magazine*, 36(6):45–64, 2016.

[69] J. Kapinski, J. V. Deshmukh, X. Jin, H. Ito, and K. R. Butts. Simulation-guided approaches for verification of automotive powertrain control systems. In *American Control Conference, ACC 2015, Chicago, IL, USA, July 1-3, 2015*, pages 4086–4095, 2015.

[70] S. Konrad and B. H. C. Cheng. Real-time specification patterns. In *Proceedings of the 27th International Conference on Software Engineering*, ICSE '05, pages 372–381. ACM, 2005.

[71] R. Koymans. Specifying real-time properties with metric temporal logic. *Real-Time Systems*, 2(4):255–299, 1990.

[72] K. J. Kristoffersen, C. Pedersen, and H. R. Andersen. Runtime verification of timed LTL using disjunctive normalized equation systems. In *Proceedings of the 3rd Workshop on Run-time Verification*, volume 89 of *ENTCS*, pages 1–16, 2003.

[73] O. Kupferman and M. Y. Vardi. Vacuity detection in temporal model checking. *STTT*, 4(2):224–233, 2003.

[74] F. Laroussinie, N. Markey, and P. Schnoebelen. Temporal logic with forgettable past. In *17th IEEE Symposium on Logic in Computer Science (LICS 2002), 22-25 July 2002, Copenhagen, Denmark, Proceedings*, pages 383–392, 2002.

[75] M. Leucker and C. Schallhart. A brief account of runtime verification. *J. Log. Algebr. Program.*, 78(5):293–303, 2009.

[76] J. Li, L. Zhang, G. Pu, M. Y. Vardi, and J. He. LTL satisfiability checking revisited. In *2013 20th International Symposium on Temporal Representation and Reasoning, Pensacola, FL, USA, September 26-28, 2013*, pages 91–98, 2013.

[77] J. Lygeros, K. H. Johansson, S. N. Simic, J. Zhang, and S. Sastry. Dynamical properties of hybrid automata. *IEEE Transactions on Automatic Control*, 48:2–17, 2003.

[78] O. Maler and D. Nickovic. Monitoring temporal properties of continuous signals. In *Proceedings of FORMATS-FTRTFT*, volume 3253 of *LNCS*, pages 152–166, 2004.

[79] N. Markey and J.-F. Raskin. Model checking restricted sets of timed paths. *Theor. Comput. Sci.*, 358(2):273–292, Aug. 2006.

[80] MathWorks. Modeling an automatic transmission controller, available at: `http://www.mathworks.com/help/simulink/examples/modeling-an-automatic-transmission-controller.html`, 1998.

[81] C. Monteiro, R. Bessa, V. Miranda, A. Botterud, J. Wang, and G. Conzelmann. Wind power forecasting: State-of-the-art 2009. Technical Report ANL/DIS-10-1, Argonne National Laboratory, 2009.

[82] T. Nghiem, S. Sankaranarayanan, G. E. Fainekos, F. Ivancic, A. Gupta, and G. J. Pappas. Monte-carlo techniques for falsification of temporal properties of non-linear hybrid systems. In *Proceedings of the 13th ACM International Conference on Hybrid Systems: Computation and Control, HSCC 2010, Stockholm, Sweden, April 12-15, 2010*, pages 211–220, 2010.

[83] J. Ouaknine and J. Worrell. Some recent results in metric temporal logic. In *Formal Modeling and Analysis of Timed Systems, 6th International Conference, FORMATS 2008, Saint Malo, France, September 15-17, 2008. Proceedings*, pages 1–13, 2008.

[84] A. Pnueli. The temporal logic of programs. In *Proceedings of the 18th IEEE Symposium Foundations of Computer Science*, pages 46–57, 1977.

[85] J.-F. Raskin. Logics, automata and classical theories for deciding real-time. *Ph.D. Thesis, University of Namur, Belgium*, 1999.

[86] T. Reinbacher, K. Y. Rozier, and J. Schumann. Temporal-logic based runtime observer pairs for system health management of real-time systems. In *Proceedings of the 20th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, volume 8413 of *LNCS*, pages 357–372. Springer, 2014.

[87] G. Rosu and K. Havelund. Synthesizing dynamic programming algorithms from linear temporal logic formulae. Technical report, Research Institute for Advanced Computer Science (RIACS), 2001.

[88] K. Y. Rozier and M. Y. Vardi. LTL satisfiability checking. *STTT*, 12(2):123–137, 2010.

[89] S. Sankaranarayanan and G. Fainekos. Falsification of temporal properties of hybrid systems using the cross-entropy method. In *Proceedings of the 15th ACM International Conference on Hybrid Systems: Computation and Control*, HSCC '12, pages 125–134, New York, NY, USA, 2012. ACM.

[90] A. K. Seda and P. Hitzler. Generalized distance functions in the theory of computation. *The Computer Journal*, 53(4):bxm108443–464, 2008.

[91] Simuquest. Enginuity. `http://www.simuquest.com/products/enginuity`, 2013. Accessed: 2013-10-14.

[92] O. Sokolsky, K. Havelund, and I. Lee. Introduction to the special section on runtime verification. *STTT*, 14(3):243–247, 2012.

[93] P. Tabuada. *Verification and Control of Hybrid Systems: A Symbolic Approach*. Springer, 2009.

[94] P. Thati and G. Rosu. Monitoring algorithms for metric temporal logic specifications. In *Runtime Verification*, volume 113 of *ENTCS*, pages 145–162. Elsevier, 2005.

APPENDIX A

PROOFS OF CHAPTER 4

Proof of Theorem 4.4.1

Proof A.0.1 *In order to show that φ_i is satisfied vacuously with respect to Φ, we must show that if $\Phi \models \varphi_i[l \leftarrow \perp]$, then the mutated specification is equivalent to the original specification. In other words, we should show that if $\Phi \models \varphi_i[l \leftarrow \perp]$, then $(\{\Phi \backslash \varphi_i\} \cup \varphi_i[l \leftarrow \perp]) \equiv \Phi$. If the mutated specification is equivalent to the original specification, then the original specification is vacuously satisfiable in any system. That is, the specification is inherently vacuous [54, 29]. We already know that if $\Phi \models \varphi_i[l \leftarrow \perp]$, then $\Phi \implies \varphi_i[l \leftarrow \perp]$ and trivially $\Phi \implies \varphi_i[l \leftarrow \perp] \cup \{\Phi \backslash \varphi_i\}$. Now we just need to prove the other direction. We need to prove that when φ_i is in NNF, then $\varphi_i[l \leftarrow \perp] \implies \varphi_i$. Since we replace only one specific literal occurrence of φ with $\perp$, the rest of the formula remains the same. Therefore, it should be noted that $\varphi_i[l \leftarrow \perp]$ does not modify any $l' \in litOccur(\varphi_i)$ where $l' \neq l$.*

Base Case: *$\varphi_i = l$ or $\varphi_i = l' \neq l$*

We know that $\perp \implies l$ and $l' \implies l'$. Therefore $\varphi_i[l \leftarrow \perp] \implies \varphi_i$.

Induction Hypothesis: $\forall \varphi_j, \varphi_j[l \leftarrow \perp] \implies \varphi_j$

Induction Step: *We will separate the case into unary and binary operators.*

Before providing the cases we should review the positively monotonic operators [73]. According to MITL semantics, $f \in \{\Box_{\mathcal{I}}, \Diamond_{\mathcal{I}}\}$ and $g \in \{\wedge, \vee\}$ are positively monotonic, i.e. for every MITL formulas φ_1 and φ_2 in NNF with $\varphi_1 \implies \varphi_2$, we have $f(\varphi_1) \implies f(\varphi_2)$. Also, for all MITL formulas φ' in NNF, we have $g(\varphi_1, \varphi') \implies g(\varphi_2, \varphi')$ and $g(\varphi', \varphi_1) \implies g(\varphi', \varphi_2)$.

Case 1: *$\varphi_i = f(\varphi_j)$ where $f \in \{\Box_{\mathcal{I}}, \Diamond_{\mathcal{I}}\}$. Since f is positively monotonic, we have that $\varphi_j[l \leftarrow \perp] \implies \varphi_j$ implies $f(\varphi_j[l \leftarrow \perp]) \implies f(\varphi_j)$. Thus,*

$f(\varphi_j)[l \leftarrow \perp] = f(\varphi_j[l \leftarrow \perp]) \implies f(\varphi_j) = \varphi_i$. As a result $\varphi_i[l \leftarrow \perp] \implies \varphi_i$.

Case 2: *$\varphi_i = g(\varphi_{j_1}, \varphi_{j_2})$ where $g \in \{\wedge, \vee\}$ Since g is positively monotonic, we have that $\varphi_{j_1}[l \leftarrow \perp] \implies \varphi_{j_1}$, and $\varphi_{j_2}[l \leftarrow \perp] \implies \varphi_{j2}$ implies*

$g(\varphi_{j_1}[l \leftarrow\perp], \varphi_{j_2}[l \leftarrow\perp]) \implies g(\varphi_{j_1}, \varphi_{j_2})$. *Thus,* $g(\varphi_{j_1}, \varphi_{j_2})[l \leftarrow\perp] = g(\varphi_{j_1}[l \leftarrow\perp], \varphi_{j_2}[l \leftarrow\perp]) \implies g(\varphi_{j_1}, \varphi_{j_2}) = \varphi_i$. *As a result* $\varphi_i[l \leftarrow\perp] \implies \varphi_i$.

Since $\varphi_i[l \leftarrow\perp] \implies \varphi_i$ *we can have:*

$\{\Phi \backslash \varphi_i\} \cup \varphi_i[l \leftarrow\perp] \implies \{\Phi \backslash \varphi_i\} \cup \varphi_i$ *which is equivalent to*

$\{\Phi \backslash \varphi_i\} \cup \varphi_i[l \leftarrow\perp] \implies \Phi$

Proof of Theorem 4.5.1

We consider two MITL($\Diamond$,$\Box$) fragments, denoted MITL($\Box$), and MITL($\Diamond$). In this proof we assume that all formulas are in NNF. We also consider LTL($\Diamond$,$\Box$) as the set of LTL formulas (with continuous semantics) that contains only $\Diamond$ and $\Box$ as temporal operators. In the following we provide the continuous semantics of LTL($\Diamond$,$\Box$) over traces with bounded duration. Semantics of LTL($\Diamond$,$\Box$) over bounded timed traces can be defined as follows:

Definition A.0.1 (LTL($\Diamond$,$\Box$) continuous semantics) *Given a timed trace $\mu : [0,T] \rightarrow 2^{AP}$ and $t, t' \in \mathbb{R}$, and an LTL($\Diamond$,$\Box$) formula ϕ, the satisfaction relation $(\mu, t) \vDash \phi$ for temporal operators is inductively defined:*

$(\mu, t) \vDash \Diamond\phi_1$ *iff* $\exists t' \in [t, T]$ *s.t* $(\mu, t') \vDash \phi_1$.

$(\mu, t) \vDash \Box\phi_1$ *iff* $\forall t' \in [t, T]$, $(\mu, t') \vDash \phi_1$.

We will consider two LTL($\Diamond$,$\Box$) fragments denoted LTL($\Box$), and LTL($\Diamond$). The syntax of MITL and LTL fragments are as presented in Section 4.5.3. We define the operator $[\phi]_{LTL}$ which can be applied to any MITL($\Diamond$,$\Box$) formula and removes its interval constraints to create a new formula in LTL($\Diamond$,$\Box$). For example if $\phi = \Diamond_{[0,10]}(p \wedge q) \wedge \Diamond_{[0,10]}p \wedge \Box_{[0,10]}q$, then $[\phi]_{LTL} = \Diamond(p \wedge q) \wedge \Diamond p \wedge \Box q$. As a result, for any $\phi \in$ MITL($\Diamond, \Box$) there exists a $\psi \in$ LTL($\Diamond, \Box$) where $\psi = [\phi]_{LTL}$. For each MITL($\Diamond, \Box$) formula ϕ, the language of ϕ denoted $L(\phi)$ is the set of all timed traces that satisfy ϕ: $\mu \vDash \phi$ iff $\mu \in L(\phi)$. Similarly, for any $\psi \in$ LTL($\Diamond, \Box$), the language of ψ denoted $L(\psi)$ is the set of all timed traces that satisfy ψ: $\mu' \vDash \psi$ iff $\mu' \in L(\psi)$. Based on set theory, it is trivial to prove that $A \subseteq B$ and $C \subseteq D$ implies $A \cup C \subseteq B \cup D$ and $A \cap C \subseteq B \cap D$.

Theorem A.0.1 *For any formula $\varphi \in$ MITL($\Diamond$), and $t \in [0, T]$ we have $L_t(\varphi) \subseteq L_t([\varphi]_{LTL})$ where $L_t(\varphi) = \{\mu \mid (\mu, t) \vDash \varphi\}$. In other words for every timed trace μ we have $(\mu, t) \vDash \varphi$ implies $(\mu, t) \vDash [\varphi]_{LTL}$.*

Proof A.0.2 *We use structural induction to prove that $L_t(\varphi) \subseteq L_t([\varphi]_{LTL})$*

Base Case: *if $\varphi = \top, \bot, p, \neg p$, then $[\varphi]_{LTL} = \varphi$ and $L_t(\varphi) \subseteq L_t([\varphi]_{LTL})$*

Induction Hypothesis: *We assume that there exist $\varphi_1, \varphi_2 \in$ MITL($\Diamond$) where for all $t \in [0, T]$, $L_t(\varphi_1) \subseteq L_t([\varphi_1]_{LTL})$ and $L_t(\varphi_2) \subseteq L_t([\varphi_2]_{LTL})$*

Case 1: *For Binary operators $\wedge, \vee$ we can use the union and intersection properties. In essence, for all formulas φ_1, φ_2 we have $L_t(\varphi_1 \vee \varphi_2) = L_t(\varphi_1) \cup L_t(\varphi_2)$ and $L_t(\varphi_1 \wedge \varphi_2) = L_t(\varphi_1) \cap L_t(\varphi_2)$. According to the IH $L_t(\varphi_1) \subseteq L_t([\varphi_1]_{LTL})$ and $L_t(\varphi_2) \subseteq L_t([\varphi_2]_{LTL})$; therefore, $L_t(\varphi_1) \cap L_t(\varphi_2) \subseteq L_t([\varphi_1]_{LTL}) \cap L_t([\varphi_2]_{LTL})$ and $L_t(\varphi_1) \cup L_t(\varphi_2) \subseteq L_t([\varphi_1]_{LTL}) \cup L_t([\varphi_2]_{LTL})$. As a result, $L_t(\varphi_1 \wedge \varphi_2) \subseteq L_t([\varphi_1]_{LTL} \wedge [\varphi_2]_{LTL}) = L_t([\varphi_1 \wedge \varphi_2]_{LTL})$, and $L_t(\varphi_1 \vee \varphi_2) \subseteq L_t([\varphi_1]_{LTL} \vee [\varphi_2]_{LTL}) = L_t([\varphi_1 \vee \varphi_2]_{LTL})$.*

Case 2: *For the temporal operator $\Diamond$, we need to compare the semantics of MITL($\Diamond$) and LTL($\Diamond$). Recall that*

$(\mu, t) \models \Diamond_{\mathcal{I}}\varphi_1$ iff $\exists t' \in (t + \mathcal{I}) \cap [0, T]$ s.t $(\mu, t') \models \varphi_1$.

$(\mu, t) \models \Diamond\varphi_1$ iff $\exists t' \in [t, T]$ s.t $(\mu, t') \models \varphi_1$.

Recall that $t'' \in (t + \mathcal{I}) \cap [0, T]$ implies $t'' \in [t, T]$ since the left bound of $\mathcal{I}$ is nonnegative.

According to the semantics, $\forall\mu.(\mu, t) \models \Diamond_{\mathcal{I}}\varphi_1$ implies

$\exists t' \in (t + \mathcal{I}) \cap [0, T]$ s.t $(\mu, t') \models \varphi_1$ implies

$\exists t' \in (t + \mathcal{I}) \cap [0, T]$ s.t $(\mu, t') \models [\varphi_1]_{LTL}$ according to IH ($L_{t'}(\varphi_1) \subseteq L_{t'}([\varphi_1]_{LTL})$).

If $\exists t' \in (t + \mathcal{I}) \cap [0, T]$ s.t $(\mu, t') \models [\varphi_1]_{LTL}$ then

$\exists t' \in [t, T]$ s.t $(\mu, t') \models [\varphi_1]_{LTL}$ since $t' \in (t + \mathcal{I}) \cap [0, T]$ implies $t' \in [t, T]$.

Moreover, $(\mu, t') \models [\varphi_1]_{LTL}$ implies that $(\mu, t) \models \Diamond[\varphi_1]_{LTL} \equiv [\Diamond\varphi_1]_{LTL}$.

As a result, $\forall\mu.\ (\mu, t) \models \Diamond_{\mathcal{I}}\varphi_1 \implies (\mu, t) \models [\Diamond\varphi_1]_{LTL}$ so $L_t(\Diamond_{\mathcal{I}}\varphi_1) \subseteq L_t([\Diamond\varphi_1]_{LTL})$.

If $\varphi \in$ MITL($\Diamond$) then $\overline{L_t([\varphi]_{LTL})} \subseteq \overline{L_t(\varphi)}$ (immediate from set theory). Thus, for all timed traces μ, $\mu \not\models [\varphi]_{LTL}$ implies that $\mu \not\models \varphi$.

Corollary A.0.1 *For any $\varphi \in$ MITL($\Diamond$), if $[\varphi]_{LTL} \in$ LTL($\Diamond$) is unsatisfiable, then φ is un-*

satisfiable.

Theorem A.0.2 *For any formula $\varphi \in$ MITL($\Box$), and $t \in [0, T]$, we have $L_t([\varphi]_{LTL}) \subseteq L_t(\varphi)$, where $L_t(\varphi) = \{\mu | (\mu, t) \models \varphi\}$. In other words. $\forall \mu.(\mu, t) \models [\varphi]_{LTL} \implies (\mu, t) \models \varphi$*

Proof A.0.3 *Similar to Theorem A.0.1, we can apply structural induction for the proof of Theorem A.0.2.*

APPENDIX B

PROOFS OF CHAPTER 5

Proof of Theorem 5.4.1

In this section, we will prove that any MITL (STL) Φ which contains a disjunction operation ($\vee$) in NNF can be be satisfied by partially covering signals. In other words, we will prove that any timed trace (signal) which satisfies Φ will be considered as partially covering timed trace (signal) according to Algorithm 7. Without loss of generality we assume that both operands of disjunction are not constant. This is because if one of the operands be equivalent to $\top$ or $\bot$, this will remove the disjunction semantically as follows $\psi \vee \top \equiv \top$ or $\psi \vee \bot \equiv \bot$ for any MITL(STL) ψ.

Lets consider partially covering timed trace (signal) returned by Algorithm 7. If there exist $\varphi_i \in \Phi$ and $l \in litOccur(\varphi_i)$ such that the timed trace μ satisfies $\varphi_i[l \leftarrow \bot]$, then μ will be reported as partially covering timed trace (signal). Recall that we assume that Φ is a conjunction of MITL specifications φ_j according to Equation (4.1). We also assume that the conjunct $\varphi_k \in \Phi$ is the subformula that contains the disjunction operation. Namely, that $\psi = \psi_1 \vee \psi_2$ is a subformula of φ_k.

Theorem B.0.3 *Any timed trace μ that satisfies φ_k will satisfy $\varphi_k[l \leftarrow \bot]$ for some $l \in litOccur(\varphi_k)$.*

Proof B.0.4 *We have two cases for $\mu \models \varphi_k$*

1. *$\mu \not\models \psi$: In this case ψ does not affect the satisfaction of $\mu \models \varphi_k$. If we choose $l' \in litOccur(\psi)$, then due to monotonicity of φ_k, $\psi[l' \leftarrow \bot]$ does not affect the satisfaction of φ_k. As a result, $\mu \models \varphi_k[l' \leftarrow \bot]$ since φ_k is in NNF.*

2. *$\mu \models \psi$: In this case ψ affects the satisfaction of $\mu \models \varphi_k$. So either $\mu \models \psi_1$ or $\mu \models \psi_2$. If $\mu \models \psi_1$ then we can choose $l' \in litOccur(\psi_2)$ and we have $\mu \models \psi[l' \leftarrow \bot]$. Similarly, if $\mu \models \psi_2$ then we can choose $l'' \in litOccur(\psi_1)$ and we have $\mu \models \psi[l'' \leftarrow \bot]$. As*

a result, there exists some $l \in litOccur(\psi)$ where $\mu \models \psi[l \leftarrow \bot]$ and accordingly $\mu \models \varphi_k[l \leftarrow \bot]$.

Finally, if $\psi = \psi_1 \vee \psi_2$ is a subformula of φ_k then there exists some $l \in litOccur(\psi)$ where $\mu \models \varphi_k[l \leftarrow \bot]$. Which means that μ is a partially covering timed trace (signal).

Corollary B.0.2 *Assume that the conjunct $\varphi_j \in \Phi$ is the subformula that contains the conjunction operation in NNF. Namely, that $\psi = \psi_1 \wedge \psi_2$ is a subformula of φ_j. Any timed trace μ that falsifies φ_j will falsify $\varphi_j[l \leftarrow \top]$ for some $l \in litOccur(\varphi_j)$.*

APPENDIX C

PROOFS OF CHAPTER 7

We prove by induction the correctness of Algorithms 9 and 10. We need to prove that at each simulation step i, the returning value of the CR algorithm is the same as the robustness value. Without loss of generality, assume $i \geq Hst$; therefore, the values in the table columns $-Hst$ to 0 contain the robustness values based on the actual simulation. When $i < Hst$ then the proof is immediate by the semantics of temporal logic. We must show that, for each sub-formula φ_k the value stored in column j of robustness table $T_{k,j}$ should be correctly computed according to the semantics

$$[\![\varphi_k]\!](\tilde{\mathbf{y}}, i+j) = T_{k,j} = CR(\varphi_k, j, \tilde{\mathbf{y}}'_i, \mathbf{d}, \mathcal{O})$$

given matrix T and vector Pre

Base case:

We will show that for each $\mathrm{MTL}_{+pt}^{<+\infty}$ sub-formula in the form of a predicate, the value which is returned by the CR algorithm (Algorithm 10) is equal to the semantics of the sub-formula. Assume the sub-formula is a predicate $p = \varphi_k$, for each simulation time $i + j$, the corresponding robustness value is stored in the column j of robustness table as follows:

$$\forall j, -Hst \leq j \leq Hrz, [\![p]\!](\tilde{\mathbf{y}}, i+j) = \mathbf{Dist_d}(\tilde{\mathbf{y}}_{i+j}, \mathcal{O}(p)) =$$

$$T_{k,j} = CR(p, j, \tilde{\mathbf{y}}'_i, \mathbf{d}, \mathcal{O})$$

Therefore, for each predicate the algorithm "CR" computes the correct robustness value.

Induction Hypothesis:

For each temporal sub-formulas φ_k, $Hst - hst(\varphi_k) \geq Hrz$ because of the fact that $Hst = Hrz + hst(\varphi) \geq Hrz + hst(\varphi_k)$; therefore $-Hst + hst(\varphi_k) \leq -Hrz$.

As a result, the values at the columns from $-Hst$ up to $-Hst + hst(\varphi_k)$ will only depend on the actual simulation values, i.e., the predicates from column $-Hst$ up to column 0 which will not change in next simulation steps. These values are shown in gray color cells of Table C.1. As a result, all the table entries from $-Hst$ up to $-Hst + hst(\varphi_k)$ will not change (in next run) and the re-computation is not needed. Therefore, we shift all the values of predicates one column to the left and we ignore columns $-Hst$ to $-Hst + hst(\varphi_k) - 1$ in our current run of Algorithm 9 (Lines 13 and 18). Therefore, it is not necessary to include the columns $-Hst$ to $-Hst + hst(\varphi_k) - 1$ in proof and Induction Hypothesis.

For Induction Hypothesis, we assume that the value stored in the robustness table is the semantically correct robustness value for each sub-formula φ_k:

$$\forall j, -Hst + hst(\varphi_k) \leq j \leq Hrz, [\![\varphi_k]\!](\tilde{\mathbf{y}}, i + j) = T_{k,j} = CR(\varphi_k, j, \tilde{\mathbf{y}}'_i, \mathbf{d}, O)$$

And if there exists unbounded past operator sub-formula like $\varphi_k = \psi \mathcal{S}_{[l',+\infty)}\varphi$, we assume the $Pre(k) = [\![\psi \mathcal{S}_{[l',+\infty)}\varphi]\!](\tilde{\mathbf{y}}, i - 1 - Hst + hst(\varphi_k)))$ because it belongs to the previous run of $i - 1$, i.e we store the value $T_{k,(-Hst+hst(\varphi_k))}$ in $Pre(k)$ before processing the current run (i) (see Algorithm 9 line 2).

Table C.1: Robustness Table (Unchangeable Values in next Runs Are in Gray Color)

$column(j) \Rightarrow$	$-Hst$	...	$-Hst + hst(\varphi_k)$	...	$-Hrz = -Hst + hst(\varphi)$	...	-1	0	1	...	Hrz
$index(time) \Rightarrow$	$i - Hst$	...	$i - Hst + hst(\varphi_k)$	...	$i - Hrz$	...	$i - 1$	i	$i + 1$	...	$i + Hrz$
Pre[1]	▨	▨	▨	▨	▨						
Pre[...]											
Pre[k]	▨	▨	▨								
Pre[...]											
Pre[$|\varphi|$]	▨	▨	▨	▨	▨	▨	▨	▨			

Induction Step:

- Negation:

  $\forall j, -Hst + hst(\varphi_k) \leq j \leq Hrz : [\![\varphi_k]\!](\tilde{\mathbf{y}}, i+j) = [\![\neg\varphi_m]\!](\tilde{\mathbf{y}}, i+j) =$

  $-[\![\varphi_m]\!](\tilde{\mathbf{y}}, i+j) = -T_{m,j} = CR(\neg\varphi_m, j, \tilde{\mathbf{y}}'_i, \mathbf{d}, O)$

- Disjunction:

  $\forall j, -Hst + hst(\varphi_k) \leq j \leq Hrz : [\![\varphi_k]\!](\tilde{\mathbf{y}}, i+j) = [\![\varphi_m \vee \varphi_n]\!](\tilde{\mathbf{y}}, i+j) =$

  $[\![\varphi_m]\!](\tilde{\mathbf{y}}, i+j) \sqcup [\![\varphi_n]\!](\tilde{\mathbf{y}}, i+j) = T_{m,j} \sqcup T_{n,j} = CR(\varphi_m \vee \varphi_n, j, \tilde{\mathbf{y}}'_i, \mathbf{d}, O)$

- Until:

  For sub-formulas of the form $\varphi_k = \varphi_m \mathcal{U}_{[l,u]} \varphi_n$, either the corresponding robustness values are correctly saved in robustness matrix for φ_m, φ_n or the semantics will satisfy the correctness if the corresponding values belong to columns beyond the Hrz:

  $\forall j, -Hst + hst(\varphi_k) \leq j \leq Hrz : [\![\varphi_k]\!] = [\![\varphi_m \mathcal{U}_{[l,u]} \varphi_n]\!](\tilde{\mathbf{y}}, i+j) =$
  $\bigsqcup_{h=i+j+l}^{i+j+u}([\![\varphi_n]\!](\tilde{\mathbf{y}}, h) \sqcap \bigsqcap_{r=i+j}^{h-1}[\![\varphi_m]\!](\tilde{\mathbf{y}}, r)) =$
  $\bigsqcup_{h\in[j+l,j+u]\cap[-Hst,Hrz]}(T_{n,h} \sqcap \bigsqcap_{r=j}^{h-1} T_{m,r}) = CR(\varphi_m \mathcal{U}_{[l,u]} \varphi_n, j, \tilde{\mathbf{y}}'_i, \mathbf{d}, O)$

- Bounded Since:

  For bounded sub-formula $\varphi_k = \varphi_m \mathcal{S}_{[l,u]} \varphi_n$, the robustness is defined as follows:

  $[\![\varphi_k]\!](\tilde{\mathbf{y}}, i+j) = [\![\varphi_m \mathcal{S}_{[l,u]} \varphi_n]\!](\tilde{\mathbf{y}}, i+j) = \bigsqcup_{h=i+j-u}^{i+j-l}([\![\varphi_n]\!](\tilde{\mathbf{y}}, h) \sqcap \bigsqcap_{r=h+1}^{i+j}[\![\varphi_m]\!](\tilde{\mathbf{y}}, r))$

  Based on IH we know that $j \geq -Hst + hst(\varphi_k)$. We must show that the values of $T_{n,p}$ for $j-u \leq p \leq j-l$ satisfy $T_{n,p} = [\![\varphi_n]\!](\tilde{\mathbf{y}}, i+p)$ i.e. $-Hst+hst(\varphi_n) \leq j-u$ and also we need to show that the values of $T_{m,q}$ for $j-u+1 \leq q \leq j$ satisfy $T_{m,q} = [\![\varphi_m]\!](\tilde{\mathbf{y}}, i+q)$

i.e. $-Hst + hst(\varphi_m) \leq j - u + 1$.

We have two cases for $hst(\varphi_k)$:

Case 1: $hst(\varphi_k) = hst(\varphi_n) + u = max\{hst(\varphi_n) + u, hst(\varphi_m) + u - 1\}$
According to IH, $j \geq -Hst + hst(\varphi_k)$, then $j \geq -Hst + hst(\varphi_n) + u$. Thus $j - u \geq -Hst + hst(\varphi_n)$ which satisfies the fact that $T_{n,p} = [\![\varphi_n]\!](\tilde{\mathbf{y}}, i + p)$ for $j - u \leq p \leq j - l$. On the other hand, in this case: $hst(\varphi_n) + u \geq hst(\varphi_m) + u - 1$
According to IH, $j + Hst \geq hst(\varphi_k) \geq hst(\varphi_m) + u - 1$, i.e., $j + Hst \geq hst(\varphi_m) + u - 1$. Thus $j - u + 1 \geq -Hst + hst(\varphi_m)$, which satisfies the fact that $T_{m,q} = [\![\varphi_m]\!](\tilde{\mathbf{y}}, i + q)$ for $j - u + 1 \leq q \leq j$.

Case 2: $hst(\varphi_k) = hst(\varphi_m) + u - 1 = max\{hst(\varphi_n) + u, hst(\varphi_m) + u - 1\}$
According to IH, $j \geq -Hst + hst(\varphi_k)$ then $j \geq -Hst + hst(\varphi_m) + u - 1$. Thus $j - u + 1 \geq -Hst + hst(\varphi_m)$ which satisfies the fact that $T_{m,q} = [\![\varphi_m]\!](\tilde{\mathbf{y}}, i + q)$ for $j - u + 1 \leq q \leq j$. On the other hand, in this case: $hst(\varphi_m) + u - 1 \geq hst(\varphi_n) + u$.
According to IH, $j + Hst \geq hst(\varphi_k) \geq hst(\varphi_n) + u$ i.e $j + Hst \geq hst(\varphi_n) + u$. Thus $j - u \geq -Hst + hst(\varphi_n)$ which satisfies the fact that $T_{n,p} = [\![\varphi_n]\!](\tilde{\mathbf{y}}, i + p)$ for $j - u \leq p \leq j - l$.

As a result:
$\forall j, -Hst + hst(\varphi_k) \leq j \leq Hrz : [\![\varphi_k]\!](\tilde{\mathbf{y}}, i + j) = [\![\varphi_m \mathcal{S}_{[l,u]} \varphi_n]\!](\tilde{\mathbf{y}}, i + j) =$
$\bigsqcup_{h=i+j-u}^{i+j-l}([\![\varphi_n]\!](\tilde{\mathbf{y}}, h) \sqcap \bigsqcap_{r=h+1}^{i+j}[\![\varphi_m]\!](\tilde{\mathbf{y}}, r)) =$
$\bigsqcup_{h=j-u}^{j-l}(T_{n,h} \sqcap \bigsqcap_{r=h+1}^{j} T_{m,r}) = CR(\varphi_m \mathcal{S}_{[l,u]} \varphi_n, j, \tilde{\mathbf{y}}'_i, \mathbf{d}, O)$

- Unbounded Since:

  For unbounded sub-formula $\varphi_k = \varphi_m \mathcal{S}_{[l,+\infty)} \varphi_n$, according to Induction Hypothesis:

$$Pre(k) = [\![\varphi_m \mathcal{S}_{[l,+\infty)} \varphi_n]\!](\tilde{\mathbf{y}}, i - 1 - Hst + hst(\varphi_k))$$

  In dynamic programming we recursively update the value

  $[\![\varphi_m \mathcal{S}_{[l,+\infty)} \varphi_n]\!](\tilde{\mathbf{y}}, i - Hst + hst(\varphi_k) + x)$

  given the previous robustness value in the table

  $[\![\varphi_m \mathcal{S}_{[l,+\infty)} \varphi_n]\!](\tilde{\mathbf{y}}, i - Hst + hst(\varphi_k) + x - 1)$

  (where $x = 0$ when we use the $Pre(k)$)

  According to Def. 3 the robustness semantics at time $i + j$:

  $[\![\varphi_m \mathcal{S}_{[l,+\infty)} \varphi_n]\!](\tilde{\mathbf{y}}, i + j) = \bigsqcup_{h=0}^{i+j-l} \left([\![\varphi_n]\!](\tilde{\mathbf{y}}, h) \sqcap \bigsqcap_{r=h+1}^{i+j} [\![\varphi_m]\!](\tilde{\mathbf{y}}, r) \right)$

  and robustness for previous time $i + j - 1$:

  $[\![\varphi_m \mathcal{S}_{[l,+\infty)} \varphi_n]\!](\tilde{\mathbf{y}}, i + j - 1) = \bigsqcup_{h=0}^{i+j-l-1} \left([\![\varphi_n]\!](\tilde{\mathbf{y}}, h) \sqcap \bigsqcap_{r=h+1}^{i+j-1} [\![\varphi_m]\!](\tilde{\mathbf{y}}, r) \right)$

  We can define robustness value at time $i + j$ given the value at time $i + j - 1$:

$$[\![\varphi_m \mathcal{S}_{[l,+\infty)} \varphi_n]\!](\tilde{\mathbf{y}}, i + j) = \bigsqcup_{h=0}^{i+j-l} \left([\![\varphi_n]\!](\tilde{\mathbf{y}}, h) \sqcap \bigsqcap_{r=h+1}^{i+j} [\![\varphi_m]\!](\tilde{\mathbf{y}}, r) \right) =$$

$$= \left(\bigsqcup_{h=0}^{i+j-l-1} \left([\![\varphi_n]\!](\tilde{\mathbf{y}}, h) \sqcap \bigsqcap_{r=h+1}^{i+j-1} [\![\varphi_m]\!](\tilde{\mathbf{y}}, r) \right) \sqcap [\![\varphi_m]\!](\tilde{\mathbf{y}}, i + j) \right) \sqcup$$
$$\left([\![\varphi_n]\!](\tilde{\mathbf{y}}, i + j - l) \sqcap \bigsqcap_{r=i+j-l+1}^{i+j} [\![\varphi_m]\!](\tilde{\mathbf{y}}, r) \right) =$$

$$= \left([\![\varphi_m \mathcal{S}_{[l,+\infty)} \varphi_n]\!](\tilde{\mathbf{y}}, i + j - 1) \sqcap [\![\varphi_m]\!](\tilde{\mathbf{y}}, i + j) \right) \sqcup$$
$$\left([\![\varphi_n]\!](\tilde{\mathbf{y}}, i + j - l) \sqcap \bigsqcap_{r=i+j-l+1}^{i+j} [\![\varphi_m]\!](\tilde{\mathbf{y}}, r) \right)$$

  Based on IH, we know that $j \geq -Hst + hst(\varphi_k)$. We must show that the value of $T_{n,j-l} = [\![\varphi_n]\!](\tilde{\mathbf{y}}, i + j - l)$, i.e., $-Hst + hst(\varphi_n) \leq j - l$ and also the values of $T_{m,q}$ for

$j - l + 1 \leq q \leq j$ satisfy $T_{m,q} = [\![\varphi_m]\!](\tilde{\mathbf{y}}, i + q)$, i.e., $-Hst + hst(\varphi_m) \leq j - l + 1$.

We have two cases for $hst(\varphi_k)$:

Case 1: $hst(\varphi_k) = hst(\varphi_n) + l = max\{hst(\varphi_n) + l, hst(\varphi_m) + l - 1\}$
According to IH, $j \geq -Hst + hst(\varphi_k)$; therefore, $j \geq -Hst + hst(\varphi_n) + l$ and $j - l \geq -Hst + hst(\varphi_n)$ which satisfies $T_{n,j-l} = [\![\varphi_n]\!](\tilde{\mathbf{y}}, i + j - l)$. On the other hand in this case: $hst(\varphi_n) + l \geq hst(\varphi_m) + l - 1$.
According to IH, $j + Hst \geq hst(\varphi_k) \geq hst(\varphi_m) + l - 1$, i.e., $j + Hst \geq hst(\varphi_m) + l - 1$. Thus $j - l + 1 \geq -Hst + hst(\varphi_m)$ which satisfies the fact that $T_{m,q} = [\![\varphi_m]\!](\tilde{\mathbf{y}}, i + q)$ for $j - l + 1 \leq q \leq j$.

Case 2: $hst(\varphi_k) = hst(\varphi_m) + l - 1 = max\{hst(\varphi_n) + l, hst(\varphi_m) + l - 1\}$
According to IH, $j \geq -Hst + hst(\varphi_k)$; therefore, $j \geq -Hst + hst(\varphi_m) + l - 1$ where $j - l + 1 \geq -Hst + hst(\varphi_m)$ which satisfies the fact that $T_{m,q} = [\![\varphi_m]\!](\tilde{\mathbf{y}}, i + q)$ for $j - l + 1 \leq q \leq j$. On the other hand in this case: $hst(\varphi_m) + l - 1 \geq hst(\varphi_n) + l$.
According to IH, $j + Hst \geq hst(\varphi_k) \geq hst(\varphi_n) + l$ i.e. $j + Hst \geq hst(\varphi_n) + l$.
thus $j - l \geq -Hst + hst(\varphi_n)$ which satisfies $T_{n,j-l} = [\![\varphi_n]\!](\tilde{\mathbf{y}}, i + j - l)$

As a result:
$\forall j, -Hst + hst(\varphi_k) \leq j \leq Hrz : [\![\varphi_k]\!](\tilde{\mathbf{y}}, i + j) = [\![\varphi_m \mathcal{S}_{[l,+\infty)} \varphi_n]\!](\tilde{\mathbf{y}}, i + j) =$
$= \Big([\![\varphi_m \mathcal{S}_{[l,+\infty)} \varphi_n]\!](\tilde{\mathbf{y}}, i + j - 1) \sqcap [\![\varphi_m]\!](\tilde{\mathbf{y}}, i + j)\Big) \bigsqcup$
$\Big([\![\varphi_n]\!](\tilde{\mathbf{y}}, i + j - l) \sqcap \bigsqcap_{r=i+j-l+1}^{i+j} [\![\varphi_m]\!](\tilde{\mathbf{y}}, r)\Big) =$

$$= \left(\begin{Bmatrix} T_{k,j-1} & \text{if } j > -Hst + hst(\varphi_k) \\ Pre[k] & \text{if } j = -Hst + hst(\varphi_k) \end{Bmatrix} \sqcap T_{m,j} \right) \bigsqcup \Big(T_{n,j-l} \sqcap \bigsqcap_{r=j-l+1}^{j} T_{m,r} \Big) =$$

$$\left(tmp_{\mathcal{S}}\right) \bigsqcup \left(T_{n,j-l} \sqcap tmp_{min}\right) = CR(\varphi_m \mathcal{S}_{[l,+\infty)} \varphi_n,\, j, \tilde{\mathbf{y}}'_i, \mathbf{d}, O).$$

APPENDIX D

PROOFS OF CHAPTER 8

In this section, we will prove the correctness of Algorithms 11 and 12. Our method first transforms the TPTL formula into LTL formula using Algorithm 11. Then it uses the dynamic programming method for monitoring LTL using Algorithm 12.

Proof of the correctness of Algorithm 11

Theorem D.0.4 *Given an encapsulated TPTL formula φ, and a finite ATSS $\hat{\rho}$, after the execution of Algorithm 11 the returned value is:*

$$M[1,0] = \top \textit{ iff } (\hat{\rho}, 0, \mathbf{0}) \models \varphi$$

To prove this theorem, we must show that the Boolean value of the subformulas that are computed using Algorithm 11, follows the TPTL semantics in Definition 3.2.6. Since Algorithm 11 does not evaluate propositional and temporal operators, their corresponding proof will be provided in Section D.

According to the TPTL semantics in Definition 3.2.6, for each freeze time operation $x.\varphi(x)$, and for each time stamp τ_i we must instantiate the time variable x with the value of τ_i. This instantiation enables us to evaluate time constraints and transform TPTL to LTL. The loop of Lines 2-21 is the main loop of Algorithm 11 which instantiates each variable v_k with each time sample τ_t in Line 3.

Lemma D.0.1 *The loop invariant of Algorithm 11 is as follows:*

$$\forall j, k, t \textit{ where } \varphi_j \equiv v_k.\varphi_i, 0 \leq t < |\hat{\rho}| :$$

$$M[j,t] = \top \textit{ iff } (\hat{\rho}, t, \varepsilon) \models v_k.\varphi_i$$

We use induction to prove the loop invariant of Algorithm 11.

Base Case: If $|V| = 0$, then formula is in LTL and algorithm does not enter the to loop of Lines 2-21 (only executes Lines 22-26). The proof of LTL is provided in Section D.

Induction Hypothesis: We assume for all v_l, where $l < k$ the invariant holds. In other words

$$\forall j, l < k, t \text{ where } \varphi_j \equiv v_l.\varphi_i, 0 \le t < |\hat{\rho}| :$$

$$M[j,t] = M[\theta_l.parent, t] = \top \text{ iff } (\hat{\rho}, t, \varepsilon) \models v_l.\varphi_i$$

Induction Step: To show the correctness for the case of v_k, we prove that Algorithm 11 correctly transform TPTL into LTL. Then we apply the correctness of LTL (See Section D) to establish the correctness of invariant considering v_k. Thus, we consider two cases that instantiate and evaluate v_k and show that Algorithm 11 follows the semantics in Definition 3.2.6. According to I.H. and since time variables are independent, we can correctly consider frozen subformulas of φ_i as $\top/\bot$. As a result, we will conclude that φ_i is in LTL.

Case of $v_k.\varphi_i$:

Consider the semantics of the freeze operator in Definition 3.2.6:

$$(\hat{\rho}, t, \varepsilon) \models v_k.\varphi_i \text{ iff } (\hat{\rho}, t, \varepsilon[v_k := \tau_t]) \models \varphi_i$$

According to this semantics, the freeze operation "v_k." first assigns a new value to the variable ($v_k := \tau_t$). Then the $\top/\bot$ value of $v_k.\varphi_i \equiv \varphi_j$ will be resolved to the same $\top/\bot$ value of φ_i (with the new environment update). Therefore, for each variable assignment ($v_k := \tau_t$), we first update the environment variables (Algorithm 11, Line 3), and then copy the φ_i's $\top/\bot$ value into $v_k.\varphi_i$'s corresponding row (Algorithm 11, Line 19).

Since each time variable v_k is independent, we create the subtree (set) θ_k corresponding to the subformulas of $v_k.\varphi_i(v_k)$ (see Section 8.3). To evaluate $v_k.\varphi_i(v_k)$, we must first instantiate variable v_k for each time stamp $\tau_0 \ldots \tau_{|\hat{\rho}|-1}$. This instantiation is considered in Line 2 of Algorithm 11 for time variable v_k and for each sample of time $0 \ldots (|\hat{\rho}| - 1)$ in Line 3 of Algorithm 11. Now we must copy the resulting $\top/\bot$ value from φ_i back to $v_k.\varphi_i$. The row corresponding to $\theta_k.root$ contains the $\top/\bot$ value of φ_i which is the root of θ_k subtree.

This values must be copied to the row $\theta_k.parent$ which is the parent of subtree θ_k and it corresponds to φ_j (Algorithm 11, Line 19).

Case of $v_k \sim r$:

Consider the semantics of time constraints in Definition 3.2.6:

$$(\hat{\rho}, u, \varepsilon) \models v_k \sim r \text{ iff } (\tau_u - \varepsilon(v_k)) \sim r$$

In the above semantics, $\varepsilon(v_k)$ corresponds to the frozen value of the time variable v_k (environment of v_k). In the previous case for $v_k.\varphi_i$, we mentioned that we should instantiate v_k at each time stamp $\tau_0 \ldots \tau_{|\hat{\rho}|-1}$. According to semantics in Definition 3.2.6, each freeze operator assigns the environment variable for the current and future samples of time t:

$$(\hat{\rho}, t, \varepsilon) \models v_k.\varphi_i \text{ iff } (\hat{\rho}, t, \varepsilon[v_k := \tau_t]) \models \varphi_i$$

Which means that the environment updates $\varepsilon[x := \tau_t]$ are observable for the current and the future samples ($t \leq u$). Therefore, after we instantiated variable v_k at each time stamp τ_t, the environment update will affect all the samples u between $t \leq u \leq |\hat{\rho}| - 1$. As a result, the time constraint $v_k \sim r$ must be updated for all future samples of $t \leq u \leq |\hat{\rho}| - 1$ for $\varepsilon[v_k := \tau_t]$ instantiation.

Lines 4-13 of Algorithm 11 follow the above discussion. Namely, for time variable v_k, we instantiate each time stamp τ_t (Line 3), the time constraints of current/future samples are evaluated according to the frozen time stamp τ_t. Actual evaluation happens in the Line 7 of Algorithm 11, where $(\tau_u - \tau_t) \sim r$ follows the semantic $(\tau_u - \varepsilon(v_k)) \sim r$ for each environment assignment of $\varepsilon[v_k := \tau_t]$. Lines 14-18 of Algorithm 11 will evaluate the LTL formula $\varphi_i(\tau_t)$.

So far, we transformed TPTL $v_k.\varphi_i(v_k)$ into LTL $\varphi_i(\tau_t)$ for each time stamp τ_t. Now we can prove that the loop invariant of Algorithm 11 holds for v_k.

Proof D.0.5 *We will prove the Induction Step by assuming the correctness of LTL formula φ_i according to Section D:*

$$\forall i, t, \varepsilon \textit{ where } \varphi_i \subset LTL, 0 \leq t < |\hat{\rho}|$$

$$M[i, t] = \top \textit{ iff } (\hat{\rho}, t, \varepsilon) \models \varphi_i$$

Since for each θ_k, $i = \theta_k.root$ is the index of the highest LTL, $M[\theta_k.root, t]$ will also contain the correct $\top/\bot$ value, therefore
$M[i, t] = M[\theta_k.root, t] = \top$ *iff* $(\hat{\rho}, t, \varepsilon) \models \varphi_i(v_k = \tau_t)$ *iff* $(\hat{\rho}, t, \varepsilon[v_k := \tau_t]) \models \varphi_i$

Since in Line 19 $M[\theta_k.parent, t] \leftarrow M[\theta_k.root, t]$
and $j = \theta_k.parent$ we have
$M[j, t] \leftarrow M[i, t]$*, as a result*

$$M[j, t] = M[\theta_k.parent, t] = \top \textit{ iff } (\hat{\rho}, t, \varepsilon) \models v_k.\varphi_i \equiv \varphi_j$$

Proof of the correctness of Algorithm 12

LTL formulas consider only propositional and temporal operators; therefore, the time variables' environment (ε) is not affected by Algorithm 12. Since time variables do not change during Algorithm 12, we assume that Algorithm 12 considers time constraints as $\top/\bot$ values since they are already evaluated in Algorithm 11. In this section, we prove that the output of Algorithm 12 corresponds to the correct evaluation of the LTL subformula φ_j at sample instance u based on Definition 3.2.6.

In essence, we will prove $M[j, u] = \top$ if $(\hat{\rho}, u, \varepsilon) \models \varphi_j$ and similarly $M[j, u] = \bot$ if $(\hat{\rho}, u, \varepsilon) \not\models \varphi_j$. For the proof of Algorithm 12, we use induction:

Base Case: In Section 8.5, we mentioned that in Line 1 of Algorithm 11 the corresponding values for atomic propositions are stored in the monitoring table. In essence, for each $a \in AP$, and for each time stamp τ_u, we save the following values in the monitoring table entry $M[a_{index}, u]$, where a_{index} is the index of atomic proposition a in the monitoring table $M_{|\varphi|\times|\hat{\rho}|}$:

1. $M[a_{index}, u] \leftarrow \top$ if $a \in \tilde{\mathbf{y}}_u$ if $(\hat{\rho}, u, \varepsilon) \models a$
2. $M[a_{index}, u] \leftarrow \bot$ if $a \notin \tilde{\mathbf{y}}_u$ if $(\hat{\rho}, u, \varepsilon) \not\models a$

Since evaluation of predicates is independent of the time variables' environment (ε) the above cases are always satisfied for all sample instances u and all environments ε. As a result, every table entry corresponding to a predicate, correctly reflects the satisfaction of the predicate with respect to the state trace $\hat{\tilde{\mathbf{y}}}$ and the environment ε. Similarly, the table entries for constant Boolean values ($\top/\bot$) are trivially correct.

Induction Hypothesis: Algorithm 11 updates the values of Table from right to left, i.e., for the samples with indexes $|\hat{\rho}| - 1$ down to 0. This is because we resolve temporal operators looking into the future. Namely, if the Boolean value in the next samples of time are resolved, then we can resolve the Boolean evaluation for the current sample of time. For the Induction Hypothesis, we assume the table entries for the proper subformulas of φ_j at the same or future samples contain the correct $\top/\bot$, i.e, we assume that

$$\forall \varphi_k \subset \varphi_j, \forall v \geq u, M[k, v] = \top \text{ iff } (\hat{\rho}, v, \varepsilon) \models \varphi_k$$

And also for the same subformula (φ_j), we assume the table entries for all the future samples contain the correct $\top/\bot$ values as follows:

$$\forall v > u, M[j, v] = \top \text{ iff } (\hat{\rho}, v, \varepsilon) \models \varphi_j$$

Induction Step: For the induction step we consider five cases of φ_j:

Case 1: $\varphi_j \equiv \neg\varphi_m$:

Consider $M[j,u] \leftarrow \neg M[m,u]$ (Algorithm 12, Line 2).

According to Definition 3.2.6: $(\hat{\rho},u,\varepsilon) \models \neg\varphi_m$ iff $(\hat{\rho},u,\varepsilon) \not\models \varphi_m$

Based on IH: $M[m,u] = \bot$ iff $(\hat{\rho},u,\varepsilon) \not\models \varphi_m$ iff (based on Def. 3.2.6) $(\hat{\rho},u,\varepsilon) \models \neg\varphi_m \equiv \varphi_j$

Therefore, $M[j,u] = \neg M[m,u] = \neg\bot$ iff $(\hat{\rho},u,\varepsilon) \not\models \varphi_m$ iff $(\hat{\rho},u,\varepsilon) \models \neg\varphi_m \equiv \varphi_j$

As a result $M[j,u] = \top$ iff $(\hat{\rho},u,\varepsilon) \models \varphi_j$

Case 2: $\varphi_j \equiv \varphi_m \wedge \varphi_n$:

Consider $M[j,u] \leftarrow M[m,u] \wedge M[n,u]$ (Algorithm 12, Line 4).

According to Definition 3.2.6: $(\hat{\rho},u,\varepsilon) \models \varphi_m \wedge \varphi_n$ iff $(\hat{\rho},u,\varepsilon) \models \varphi_m$ and $(\hat{\rho},u,\varepsilon) \models \varphi_n$

Based on IH: $M[m,u] = \top$ iff $(\hat{\rho},u,\varepsilon) \models \varphi_m$ and $M[n,u] = \top$ iff $(\hat{\rho},u,\varepsilon) \models \varphi_n$

We know that, $M[m,u] \wedge M[n,u] = \top$ iff $M[m,u] = \top$ and $M[n,u] = \top$

Thus, $M[m,u] \wedge M[n,u] = \top$ iff $(\hat{\rho},u,\varepsilon) \models \varphi_m$ and $(\hat{\rho},u,\varepsilon) \models \varphi_n$

Therefore, $M[m,u] \wedge M[n,u] = \top$ iff $(\hat{\rho},u,\varepsilon) \models \varphi_m \wedge \varphi_n \equiv \varphi_j$

As a result $M[j,u] = \top$ iff $(\hat{\rho},u,\varepsilon) \models \varphi_j$

Case 3: $\varphi_j \equiv \varphi_m \vee \varphi_n$:

Consider $M[j,u] \leftarrow M[m,u] \vee M[n,u]$ (Algorithm 12, Line 6).

According to Definition 3.2.6: $(\hat{\rho},u,\varepsilon) \models \varphi_m \vee \varphi_n$ iff $(\hat{\rho},u,\varepsilon) \models \varphi_m$ or $(\hat{\rho},u,\varepsilon) \models \varphi_n$

Based on IH: $M[m,u] = \top$ iff $(\hat{\rho},u,\varepsilon) \models \varphi_m$ and $M[n,u] = \top$ iff $(\hat{\rho},u,\varepsilon) \models \varphi_n$

We know that, $M[m,u] \vee M[n,u] = \top$ iff $M[m,u] = \top$ or $M[n,u] = \top$

Thus, $M[m,u] \vee M[n,u] = \top$ iff $(\hat{\rho},u,\varepsilon) \models \varphi_m$ or $(\hat{\rho},u,\varepsilon) \models \varphi_n$

Therefore, $M[m,u] \vee M[n,u] = \top$ iff $(\hat{\rho},u,\varepsilon) \models \varphi_m \vee \varphi_n \equiv \varphi_j$

As a result $M[j,u] = \top$ iff $(\hat{\rho},u,\varepsilon) \models \varphi_j$

Case 4: $\varphi_j \equiv \bigcirc\varphi_m$

Consider $M[j,u] \leftarrow M[m,u+1]$ if $u < |\hat{\rho}| - 1$ (Line 11) and $M[j,u] \leftarrow \bot$ otherwise (Line

9 of Algorithm 12).

According to Definition 3.2.6 we have two cases:

Case 4.1) $u < (|\hat{\rho}| - 1)$:

$(\hat{\rho}, u, \varepsilon) \models \bigcirc\varphi_m$ iff $(\hat{\rho}, u+1, \varepsilon) \models \varphi_m$

Based on IH: $M[m, u+1] = \top$ iff $(\hat{\rho}, u+1, \varepsilon) \models \varphi_m$ iff $(\hat{\rho}, u, \varepsilon) \models \bigcirc\varphi_m \equiv \varphi_j$

As a result $M[j, u] = M[m, u+1] = \top$ iff $(\hat{\rho}, u, \varepsilon) \models \varphi_j$

Case 4.2) $u = |\hat{\rho}| - 1$:

by Definition 3.2.6, $(\hat{\rho}, u, \varepsilon) \not\models \bot$

Line 9 of Algorithm 12 similarly assigns $M[j, u] \leftarrow \bot$

Case 5: $\varphi_j \equiv \varphi_m U \varphi_n$

According to [55], Until operation can be simplified according to following equivalence relation:

$$\phi U \psi \equiv \psi \vee (\phi \wedge \bigcirc(\phi U \psi))$$

In other words, we need to consider current value of $\bigcirc(\phi U \psi)$ (future value of $\phi U \psi$ at the next sample) and use the current values of ϕ and ψ to resolve and evaluate $\phi U \psi$ at the current sample using equation $\psi \vee (\phi \wedge \bigcirc(\phi U \psi))$. Algorithm 12 considers two case for $\varphi_j \equiv \varphi_m U \varphi_n \equiv \varphi_n \vee (\varphi_m \wedge \bigcirc(\varphi_m U \varphi_n))$:

Case 5.1) $u < (|\hat{\rho}| - 1)$:

Now consider the update of $M[j, u] \leftarrow M[n, u] \vee (M[m, u] \wedge M[j, u+1])$ according to Line 17 of Algorithm 12.

Based on IH: $M[n, u] = \top$ iff $(\hat{\rho}, u, \varepsilon) \models \varphi_n$ and $M[m, u] = \top$ iff $(\hat{\rho}, u, \varepsilon) \models \varphi_m$ and

$M[j, u+1] = \top$ iff $(\hat{\rho}, u+1, \varepsilon) \models \varphi_j$ iff $(\hat{\rho}, u, \varepsilon) \models \bigcirc\varphi_j$

According to Case 2 (Conjunction) $M[m, u] \wedge M[j, u+1] = \top$ iff $(\hat{\rho}, u, \varepsilon) \models \varphi_m$ and $(\hat{\rho}, u, \varepsilon) \models \bigcirc\varphi_j$

Therefore, $M[m, u] \wedge M[j, u+1] = \top$ iff $(\hat{\rho}, u, \varepsilon) \models \varphi_m \wedge \bigcirc\varphi_j$

We know that, $M[j, u] = \top$ iff $M[n, u] = \top$ or $M[m, u] \wedge M[j, u + 1] = \top$

According to Case 3 (Disjunction) $M[j, u] = \top$ iff $(\hat{\rho}, u, \varepsilon) \models \varphi_n$ or $(\hat{\rho}, u, \varepsilon) \models \varphi_m \wedge \bigcirc\varphi_j$

As a result, $M[j, u] = \top$ iff $(\hat{\rho}, u, \varepsilon) \models \varphi_n \vee (\varphi_m \wedge \bigcirc\varphi_j)$

Case 5.2) $u = |\hat{\rho}| - 1$:

According to Case 4.2 for Next operator: $(\hat{\rho}, u, \varepsilon) \not\models \bot$

This implies that $\varphi_j \equiv \varphi_n \vee (\varphi_m \wedge \bot) \equiv \varphi_n \vee \bot \equiv \varphi_n$

Now consider the update of $M[j, u] \leftarrow M[n, u]$ according to Line 15 of Algorithm 12.

Based on IH: $M[n, u] = \top$ iff $(\hat{\rho}, u, \varepsilon) \models \varphi_n$

Therefore after the assignment, $M[j, u] = \top$ iff $(\hat{\rho}, u, \varepsilon) \models \varphi_j$