On the Equivalence between Must-Test Specifications and the Linear Temporal Logic

by

Atefeh Farhadi

A thesis submitted to the
Department of Computer Science
in conformity with the requirements for
the degree of Master of Science

Bishop's University
Canada
July 2025

Abstract

This paper explores the connections between two formal verification techniques: must testing and Linear Temporal Logic (LTL). Both are vital for validating system behaviors and ensuring proper operation. Must testing captures dynamic behaviors and ensures system requirements are met under all execution scenarios, while LTL specifies and verifies temporal properties within systems. Concretely, we explore the equivalence between must testing and LTL. We develop a practical (algorithmic) framework to translate must-test specifications into equivalent LTL formulas. While we thus go only half-way toward establishing an equivalence, we believe that the conversion the other way around is possible as well. On a practical note, we also note that model checking (the algorithmic LTL verification framework) is a very mature technology widely uses in practice, while must testing is deployed to a much lesser extent. We therefore argue that our conversion is much more useful for practical applications. This being said, a conversion of LTL formulae into equivalent must tests is a necessary future expansion of this study to establish formally the equivalence of the two frameworks.

Acknowledgment

First and foremost, I want to thank the Computer Science Department at Bishop's University for giving me the opportunity to pursue a Master's degree.

I am deeply grateful to Dr. Stefan Bruda for his guidance, patience, and support throughout this research. His insights and encouragement have helped shape this work, and I truly appreciate everything he has done.

Huge thanks go to my husband and my family, especially my mom and dad, who may be far away but have always been with me in spirit. Their love, support and belief in me have been my greatest motivation, pushing me forward even when things got tough.

Contents

Chapter 1

Introduction

Ensuring the reliability of complex systems is crucial, especially as they become increasingly concurrent and distributed. Formal verification provides the mathematical rigor needed to guarantee system correctness, with must testing and Linear Temporal Logic (LTL) being two powerful tools in this domain. Must testing, rooted in process algebra, evaluates system behavior across all potential environments, while LTL is extensively used to specify and verify temporal properties in reactive systems.

Despite their strengths, the connection between must testing and LTL has not been fully explored. Uncovering their formal equivalence could lead to more integrated and robust verification frameworks. This proposal aims to address this gap by developing a framework that establishes the equivalence between must-testing and LTL, emphasizing practical applications in system verification.

Building on recent progress in formal verification, especially in the context of concurrent systems, this research seeks to contribute a new approach that unifies testing with formal verification, ultimately improving the reliability of complex systems.

Systems need to be reliable because failures can cause serious financial damage or even put people at risk. Formal verification helps catch problems before they turn into major issues, making systems more dependable. One challenge is that testing and formal verification often feel like separate worlds, even though they both aim to check if a system behaves correctly. My research focuses on connecting these two by finding a way to convert test structures into equivalent LTL formulas while preserving must-testing semantics. This makes it easier to use formal methods in real-world verification. Formal verification helps identify potential issues before they become major problems [4].

Formal verification uses mathematical models and logic to validate system behavior and can be approached from either an algebraic or a logical perspective. Algebraic approaches include model-based testing [2, 16], while the most common logical approach is model checking, which operates on a specification formulated in some form of temporal logic [5]. Algebraic models capture dynamic behavior, while model checking verifies temporal properties.

Ultimately however both these approaches accomplish the same thing namely, the specification and verification of computing systems. As a consequence, an interesting research question is whether algorithmic equivalence between algebraic and logical specifications exist. Previous research established an equivalence between failure trace testing and CTL [4, 18]. We now consider two different frameworks and investigate the equivalence between must testing and LTL. We aim to develop methods that can be used to effectively convert must-test specifications into equivalent LTL formulas. Our big research question is how to develop a practical framework that formally establishes the equivalence between must testing and Linear Temporal Logic (LTL).

In this research, we develop a method to systematically translate test structures into LTL formulas, ensuring that the conversion respects must-testing semantics. We prove key equivalence results and demonstrated how this approach strengthens formal verification. This work bridges the gap between testing and verification, making it easier to reason about system correctness using LTL. Our approach defines an algorithmic conversion process from must testing to LTL formulas. As a consequence it becomes now possible to formulate must-test scenarios, converting them into LTL, and validate the results against real-world system failures using model checking. Further research is needed to establish the same kind of algorithmic conversion the other way around (from LTL formulae to tests under the must-testing model).

System failures can severely impact sectors like healthcare, finance, and transportation. Formal verification helps analyze and mitigate these risks. Must testing and LTL model checking both aim to ensure system reliability. They both have advantages and disadvantages. Finding common ground between these techniques offers new research and practical opportunities [4, 14].

Chapter 2

Preliminaries

In this chapter, we provide a comprehensive overview of fundamental concepts necessary for understanding our research. We begin with an introduction to model checking and model-based testing, followed by a formal discussion on temporal logic, labeled transition systems, and satisfaction operators. Furthermore, we elaborate on must testing, its formal foundations, and its application in the verification of system properties.

2.1 Temporal Logic

Temporal logic is a formal language used to reason about time and temporal relationships within systems. *Linear Temporal Logic (LTL)* extends propositional logic with temporal operators to specify properties of systems over time [12]. The main temporal operators in LTL are:

- $X\varphi$: Next φ
- $F\varphi$: Eventually φ
- $G\varphi$: Globally φ
- $\varphi U \psi$: φ Until ψ

LTL is typically interpreted over *Kripke structures* which provide the semantic basis for model checking [5, 8]. In many formal verification settings, particularly in model checking, we use *Kripke structures* to represent system behavior. A Kripke structure is defined as $M = (S, S_0, R, L)$ where:

- S is a finite set of states,

- $S_0 \subseteq S$ is the set of initial states,
- $R \subseteq S \times S$ is the transition relation (note in particular that transitions are not labeled in any way).
- $L : S \to 2^{AP}$ is a labeling function that assigns to each state the set of atomic propositionsfrom the set AP that are true in that state.

Kripke structures provide the semantic foundation for temporal logics like LTL, as they let us formally specify which properties should hold at each state along an execution path.

We will also explore LTL semantics over *Labeled Transition Systems (LTS)*. An LTS is a formalism used in model-based testing to represent state transitions labeled with actions. An LTS is defined as $LTS = (S, L, T)$ where:

- S is a set of states representing the different configurations that the system can be in,
- A is a set of labels (or actions) that drive the transitions between states.
- $T \subseteq S \times A \times S$ is the transition relation, meaning that if $(s, a, s') \in T$, then the system can move from state s to state s' by executing action a.

Note in passing that a Kripke structure is similar to an LTS but includes additional information about which atomic propositions hold in each state [5].

In what follows we will use the natural concept of reachability over pairs of Kripke structure or LTS states. For the purpose we define the relation $\rightsquigarrow$ such that $s \rightsquigarrow s'$ means that state s' can be reached from state s by following a series of transitions [6].

2.1.1 Satisfaction over Kripke Structures

The satisfaction operator is crucial for verifying the properties of systems modeled using formal methods. It allows formal specifications (expressed in temporal logic or other formalisms) to be checked against system models [5, 8]. Evaluating whether specific states satisfy desired properties enables the detection of errors or the verification of correctness in system behaviors.

In a Kripke structure, we use the satisfaction operator to check if certain properties hold true at a specific state that $M = (S, S_0, R, L)$:

- **Atomic Proposition**: If the basic fact p is true in state s, then s satisfies p.

$$M, s \models p \quad \text{if} \quad p \in L(s)$$

- **Negation**: If φ is not true at state s, then s satisfies $\neg\varphi$.

$$M, s \models \neg\varphi \quad \text{if} \quad M, s \not\models \varphi$$

- **Conjunction**: If both φ and ψ are true at state s, then s satisfies $\varphi \wedge \psi$.

$$M, s \models \varphi \wedge \psi \quad \text{if} \quad M, s \models \varphi \text{ and } M, s \models \psi$$

- **Next**: If the next state s' reachable from s satisfies φ, then s satisfies $X\varphi$.

$$M, s \models X\varphi \quad \text{if there exists } s' \text{ such that } (s, s') \in R \text{ and } M, s' \models \varphi$$

- **Eventually**: If you can eventually reach a state s' where φ is true, starting from s, then s satisfies $F\varphi$.

$$M, s \models F\varphi \quad \text{if there exists } s' \text{ such that } s \rightsquigarrow s' \text{ and } M, s' \models \varphi$$

- **Globally**: If φ is true in s and all states reachable from s, then s satisfies $G\varphi$.

$$M, s \models G\varphi \quad \text{if for all } s' \text{ such that } s \rightsquigarrow s',\ M, s' \models \varphi$$

- **Until**: If there is a state s' reachable from s where ψ is true, and φ is true in all states from s up to s', then s satisfies $\varphi U \psi$.

$$\begin{aligned} M, s \models \varphi U \psi \quad &\text{if there exists } s' \text{ such that } s \rightsquigarrow s' \text{ and } M, s' \models \psi, \\ &\text{and for all } s'' \text{ such that } s \rightsquigarrow s'' \text{ and } s'' \neq s',\ M, s'' \models \varphi \end{aligned}$$

2.1.2 Satisfaction over Labeled Transition Systems

In a Labeled Transition System (LTS), we also use the satisfaction operator [6] to check properties, but we focus on actions that cause transitions between states that $LTS = (S, L, T)$:

- **Atomic Proposition**: If the basic fact p is true in state s, then s satisfies p.

$$LTS, s \models p \quad \text{if} \quad p \in L(s)$$

- **Negation**: If φ is not true at state s, then s satisfies $\neg\varphi$.

$$LTS, s \models \neg\varphi \quad \text{if} \quad LTS, s \not\models \varphi$$

- **Conjunction**: If both φ and ψ are true at state s, then s satisfies $\varphi \wedge \psi$.

$$LTS, s \models \varphi \wedge \psi \quad \text{if} \quad LTS, s \models \varphi \text{ and } LTS, s \models \psi$$

- **Next**: If the next state s' reachable from s by an action a satisfies φ, then s satisfies $X\varphi$.

$$LTS, s \models X\varphi \quad \text{if there exists a transition } (s, a, s') \in T \text{ such that } LTS, s' \models \varphi$$

- **Eventually**: If from state s, you can eventually reach a state s' where φ is true, then s satisfies $F\varphi$.

$$LTS, s \models F\varphi \quad \text{if there exists a state } s' \text{ such that } s \rightsquigarrow s' \text{ and } LTS, s' \models \varphi$$

- **Globally**: If φ is true in s and all states reachable from s, then s satisfies $G\varphi$.

$$LTS, s \models G\varphi \quad \text{if for all states } s' \text{ such that } s \rightsquigarrow s',\ LTS, s' \models \varphi$$

- **Until**: If there is a state s' reachable from s where ψ is true, and φ is true in all states from s up to s', then s satisfies $\varphi U \psi$.

$$LTS, s \models \varphi U \psi \quad \text{if there exists a state } s' \text{ such that } s \rightsquigarrow s' \text{ and } LTS, s' \models \psi, \text{ and for all states } s'' \text{ such that } s \rightsquigarrow s'' \text{ and } s'' \neq s',\ LTS, s'' \models \varphi$$

2.2 Model-Based Testing

Model-based testing uses formal models of system behavior to generate test cases. These models abstract the system's behavior, allowing systematic and efficient test case creation. Techniques used in model-based testing include finite state machines and labeled transition systems. These techniques help represent system behavior formally, leading to comprehensive test coverage and efficient bug detection [2].

A finite state machine is a particular case of LTS. A FSM is defined as a tuple $M = (S, S_0, I, T, F)$ where:

- S is a finite set of states.
- $S_0 \subseteq S$ is the set of initial states.
- I is the set of inputs.
- $T : S \times I \rightarrow S$ is the transition function.
- F is the set of final states.

2.2.1 Must Testing

Must testing is a formal verification technique used to ensure that a system satisfies specified requirements under all possible environmental conditions. Let S represent the system under test and let M denote a set of requirements or properties that S must fulfill [10, 14].

Formally, must-testing evaluates whether all possible environments E interacting with S, the composed system $S \parallel E$ satisfies M. This is expressed as:

$$\forall E.(S \parallel E \models M)$$

where $S \parallel E$ denotes the system S running concurrently with environment E, and $\models$ denotes satisfaction of the requirements M.

The objective of must testing is to ensure that S behaves correctly and meets its essential specifications regardless of the external conditions it encounters during execution. This verification process is crucial for validating the reliability and robustness of systems across various operational scenarios.

In academic research, such as that by Uddin et al. [14], the formal definition of must-testing serves as a foundational concept for subsequent theoretical developments and empirical investigations. It provides a rigorous basis for analyzing system behavior and verifying compliance with critical requirements.

Must-testing is defined inductively. This method involves defining tests step-by-step, starting from basic elements and progressing to more complex scenarios.

The basic test cases are atomic actions as well as the basic tests *STOP*(deadlock) and *SUCCESS*(successful test termination). An atomic action test a can be defined as:

$$\text{test}(a) = \{a\}$$

This test observes whether the system can act a.

More complex tests can observe sequences of actions. If α and β are sequences of actions, then their concatenation $\alpha \cdot \beta$ forms a new test:

$$\text{test}(\alpha \cdot \beta) = \text{test}(\alpha) \cup \text{test}(\beta)$$

This test observes whether the system can perform the sequence α followed by β.

To handle incomplete observations, we consider the prefix closure of tests. If α is a test, then the prefix closure $\text{prefix}(\alpha)$ includes all prefixes of α:

$$\text{prefix}(\alpha) = \{\beta \mid \beta \text{ is a prefix of } \alpha\}$$

Tests may include non-deterministic choices, represented by the union of multiple tests. If α and β are tests, then their non-deterministic choice is:

$$\text{test}(\alpha \cup \beta) = \text{test}(\alpha) \cup \text{test}(\beta)$$

Tests can also observe concurrent actions. If α and β are tests, then their concurrent execution is:

$$\text{test}(\alpha \parallel \beta) = \{(\alpha', \beta') \mid \alpha' \in \text{test}(\alpha), \beta' \in \text{test}(\beta)\}$$

Must testing defines test processes, which interact with the system under test (SUT) to check for specific behaviors. A test process specifies the expected sequences of events and interactions. The SUT must refine the test process for the test to be considered successful. This means the behavior of the SUT must be a superset of the behavior defined by the test process. A must-test is successful if, for every possible interaction specified by the test process, the SUT can exhibit the expected behavior. If the SUT can pass all such interactions, it satisfies the must-test.

2.2.2 Process Algebraic Definition of Must-Testing

A process algebra such as CSP [9] is commonly used to specify tests and systems under test. Here, we present the minimal set of CSP operators essential for understanding the remainder of this manuscript and in particular needed for the definition of must testing. Readers interested in the full CSP definition are encouraged to consult other resources [13, 15].

We define a finite set of actions Σ and two special actions τ (internal action) and ω (success), where $\tau, \omega \notin \Sigma$. The basic processes include

STOP, which does not perform any action and thus has no semantics, and *SUCCESS*, which performs the special action ω:

$$\frac{}{SUCCESS \xrightarrow{\omega} STOP}$$

For any CSP process P and action a, $a \to P$ denotes a CSP process with the following semantics:

$$\frac{}{(a \to P) \xrightarrow{a} P}$$

The prefix choice $x : A \to P(x)$, where $A \subseteq \Sigma$ and $P(a)$ are CSP processes for all $a \in A$, generalizes the prefix operator above and has the following semantics:

$$\frac{}{(x : A \to P(x)) \xrightarrow{a} P(a)} \ [a \in A]$$

Prefix choices are more generally represented using the external choice operators $P \,\Box\, Q$ and $\Box_{i \in I} P_i$ with the following semantics:

$$\frac{P \xrightarrow{a} P'}{\begin{array}{l} P \,\Box\, Q \xrightarrow{a} P' \\ Q \,\Box\, P \xrightarrow{a} P' \end{array}} \ [a \neq \tau]$$

$$\frac{P \xrightarrow{\tau} P'}{\begin{array}{l} P \,\Box\, Q \xrightarrow{\tau} P' \,\Box\, Q \\ Q \,\Box\, P \xrightarrow{\tau} Q \,\Box\, P' \end{array}}$$

$$\frac{P_j \xrightarrow{a} P'}{\Box_{i \in I} P_i \xrightarrow{a} P'} \ [j \in I, a \neq \tau]$$

$$\frac{P_j \xrightarrow{\tau} P'_j}{\Box_{i \in I} Q \xrightarrow{\tau} \Box_{i \in I} P'_i \text{ with } i \neq j \implies P'_i = P_i} \ [j \in I]$$

As opposed to external choice, the internal choice $P \sqcap Q$ is resolved before interaction with the environment, and can be easily defined using internal actions:

$$\frac{}{\begin{array}{l} P \sqcap Q \xrightarrow{\tau} P \\ P \sqcap Q \xrightarrow{\tau} Q \end{array}}$$

$$\frac{}{\sqcap_{i \in I} P_i \xrightarrow{\tau} P_j} \ [j \in I]$$

Processes P and Q can be run in parallel using the alphabetized parallel operator ${}_A\|_B$, which restricts the interfaces of P and Q to A and B, respectively, and synchronizes P and Q over common actions (in $A \cap B$):

$$\frac{\begin{array}{c} P \xrightarrow{a} P' \\ \hline P \;{}_A\|_B\; Q \xrightarrow{a} P' \;{}_A\|_B\; Q \\ Q \;{}_B\|_A\; P \xrightarrow{a} Q \;{}_B\|_A\; P' \end{array}}{} \left[a \in A \setminus B \cup \{\tau\}\right]$$

$$\frac{\begin{array}{c} P \xrightarrow{a} P' \\ Q \xrightarrow{a} Q' \end{array}}{P \;{}_A\|_B\; Q \xrightarrow{a} P' \;{}_A\|_B\; Q'} \left[a \in A \cap B\right]$$

The shortcut $\|$ is common and represents ${}_{\alpha(P)}\|_{\alpha(Q)}$, where $\alpha(X)$ is the interface of X, i.e., the set of actions that can be performed by process X. We note in passing that CSP features a few more parallel composition operators, but all of them are either particular instances of, of slight generalizations of the alphabetized parallel operators.

Finally, the hiding operator $P \setminus A$ hides in P all the occurrences of any action $a \in A$ by converting them into internal actions τ. It has the following semantics:

$$\frac{P \xrightarrow{a} P'}{P \setminus A \xrightarrow{\tau} P' \setminus A} \left[a \in A\right]$$

$$\frac{P \xrightarrow{a} P'}{P \setminus A \xrightarrow{a} P' \setminus A} \left[a \notin A\right]$$

Formulating must testing in CSP A system under test P is tested against the test T by considering all maximal executions of the process $(P \parallel Q) \setminus \Sigma$. The hiding of all the actions means that we are interested only in the outcome of the test (success or failure). The process P passes the must test T (denoted as P must T) if all maximal executions of $(P \parallel Q) \setminus \Sigma$ result in ω, indicating successful execution [15].

For example, consider the processes:

$$\begin{aligned} P_1 &= a \rightarrow STOP \,\Box\, b \rightarrow STOP \\ P_2 &= a \rightarrow STOP \,\sqcap\, b \rightarrow STOP \end{aligned}$$

Differentiated by the test $T = b \rightarrow SUCCESS$, P_1 must T as it guarantees successful execution based on the test, whereas $\neg(P_2 \text{ must } T)$ because P_2's internal choice may deadlock before reaching *SUCCESS* [13, 15].

Chapter 3

Previous Work

Understanding the relationship between failure trace testing and Computation Tree Logic (CTL) has been a crucial area of research in formal verification. Zuo's foundational work established key connections between failure trace testing and CTL [18], demonstrating their semantic links [4]. This background provides the context for further exploration of related topics, including the connections between Linear Temporal Logic (LTL) and must testing, which are central to this thesis.

Recent studies have built on these foundations, exploring the equivalence between different temporal logics and their applications in system verification. For example, Bruda et al. examined how failure trace testing aligns with CTL, focusing on modeling and verifying failure scenarios using temporal logic [4]. This work highlights the ongoing effort to unify different approaches to temporal logic and testing, thereby providing a more robust framework for system verification.

3.1 Previous Efforts on LTL and Must Testing

In the domain of Linear Temporal Logic (LTL) and must testing, several research efforts have contributed to the current understanding. Büchi automata have played a significant role in relating LTL to testing preorders, particularly in defining semantic equivalences. Early work by DeNicola and Hennessy [10] used Büchi automata to establish a unified semantic theory that connected LTL with Labelled Transition Systems (LTS) through trace inclusion. This approach laid the groundwork for constructing Büchi processes that could be used to represent the behaviors specified by LTL formulas.

The transition from Büchi automata to practical system verification, especially in the real-time domain[7], has been a significant area of focus. This progression has influenced the methodologies used in converting must-tests into equivalent LTL formulas, which is a central theme of this thesis [1].

3.2 Ongoing Effort

Recent advancements have significantly contributed to the conversion of must-tests to Linear Temporal Logic (LTL) formulas. Notably, De Nicola and Hennessy's foundational work on testing equivalences provides essential methodologies for relating various testing strategies to temporal logic formulas [11]. Building upon this, Bruda et al. have developed new algorithms that facilitate the conversion between different testing strategies and temporal logic representations [4]. These developments offer valuable insights and methodologies directly applicable to the conversion process discussed in this thesis. By integrating these recent advancements, this research not only builds upon the existing body of knowledge but also contributes to the latest progress in the field.

Additionally, the relationship between CTL and algebraic specifications has seen limited but noteworthy investigation. Early research in this area, which introduced constructive conversions of LTS into equivalent Kripke structures, serves as a precursor to modern efforts in the field [11]. This work, alongside DeNicola's algorithmic conversions involving "dummy" elementary propositions, provides a historical backdrop for understanding the challenges and opportunities in aligning must testing with LTL [10].

This thesis aims to integrate these historical perspectives with recent advancements, demonstrating how contemporary methodologies can enhance the practical application of LTL in must-testing scenarios. By synthesizing these efforts, this research seeks to establish a comprehensive framework for translating must-test specifications into equivalent LTL formulas, thereby contributing to the broader field of formal verification.

Chapter 4

Converting Must-Tests to LTL Formulas

This chapter presents a method for converting a test structure into an equivalent Linear Temporal Logic (LTL) formula. The goal of this conversion is to align the test structure with must-testing semantics, ensuring that the test applies to all possible executions of a system. In other words, we aim to express the conditions of the test in a temporal logic framework that can verify whether all potential executions of the system meet these conditions. This approach is crucial in the verification of concurrent systems, where there are multiple potential execution paths that must all be considered in the test.

To begin, we establish a formal semantics for the behavior of a system, drawing inspiration from Bruda et al. This foundational model provides a way to relate the structure of the test to the possible behaviors of the system. Following this, we present a proof of equivalence using structural induction to show that any must test T is logically equivalent to its corresponding LTL formula $R(T)$. This proof is key to ensuring that the conversion maintains the integrity of the original test conditions. Finally, we validate the approach through several practical examples and counterexamples, demonstrating the correctness and applicability of the conversion method.

4.1 Principles

To ensure the correctness of the conversion, we define a function $R(T)$ that maps each must test T to an equivalent LTL formula. The conversion follows a set of principles designed to preserve the intent of the test as follows:

- If T is a simple action test, where the test expects a specific action to occur, the corresponding LTL formula $R(T)$ is simply the atomic LTL proposition that represents that action.
- If T consists of sequential actions $T_1; T_2$, where T_1 must be followed by T_2, the LTL formula becomes $R(T) = R(T_1) \wedge XR(T_2)$. The operator X represents the "next" operator, indicating that after T_1 is satisfied, T_2 must occur.
- If T allows for a choice between actions T_1 and T_2, the LTL formula is $R(T) = R(T_1) \vee R(T_2)$. This reflects that either T_1 or T_2 can be satisfied, capturing the disjunction between the choices.
- If T involves looping behavior, the LTL formula uses temporal operators like G (globally) and F (eventually) to express repetition. For instance, if T requires an action to repeat infinitely, the LTL formula might include G to indicate that the action must occur at every point in the future.

The use of structural induction on T allows us to prove that this transformation preserves logical equivalence with the must-testing semantics, ensuring that the resulting LTL formula accurately reflects the test's intent.

4.1.1 Examples and Counterexamples

Example 1: Simple Action Test Consider a simple test T that requires action a to always occur. The corresponding LTL formula is:

$$R(T) = Ga$$

This formula guarantees that action a will occur in every execution trace of the system, ensuring the test's conditions are satisfied across all potential executions.

Example 2: Choice Between Actions Suppose T allows for the choice between action a and action b. The equivalent LTL formula is:

$$R(T) = Fa \vee Fb$$

This formula ensures that at least one of the actions, either a or b, will eventually be executed, satisfying the test's conditions.

Example 3: Sequential Execution Consider a test T that requires action a to occur first, followed by action b. The corresponding LTL formula is:

$$R(T) = a \wedge X(Fb)$$

This formula ensures that after a occurs, b must eventually occur, capturing the sequential nature of the test.

Example 4: Looping Behavior Consider a test where action a must repeatedly occur. The corresponding LTL formula is:

$$R(T) = G(a \Rightarrow Xa)$$

This formula ensures that whenever action a occurs, it will be followed by another occurrence of a in every execution trace, capturing the looping behavior of the test.

Counterexample: Failing Conversion Consider a test T that expects action a to occur exactly once. A naive conversion might suggest the LTL formula:

$$R(T) = Fa \wedge \neg Ga$$

However, this formula fails to correctly capture cases where a occurs multiple times. For instance, if a occurs twice, this formula would incorrectly allow the test to pass, highlighting the importance of carefully considering the exact semantics of the test during conversion.

This example demonstrates the need for precise conversion strategies to ensure that the resulting LTL formula accurately reflects the test's intent and constraints.

4.1.2 Role of the Internal Action

Internal actions τ are transitions that occur within the system but are not observable from the outside. Despite their invisibility, τ-transitions are crucial for modeling the internal behavior of concurrent systems. These actions help to connect states that are logically related without causing observable changes in the system's external behavior.

To properly handle τ-transitions, we need to introduce two concepts:

- **τ-equivalence** and **τ-closure**:We say that two states s_0 and s_1 are τ-equivalent, written $s_0 \approx_\tau s_1$, if there exists a sequence of internal

transitions (τ) from s_0 to s_1 or from s_1 to s_0 — that is, they are observationally equivalent being differentiated only via τ-transitions. This relation is reflexive, transitive, and symmetric, and allows us to treat such states as semantically equivalent in our analysis. This equivalence allows us to treat states that are reachable by τ-transitions as if they are part of the same execution path. The τ-closure $\tau(s)$ of a state s is the reflexive and transitive closure of $\{s\}$ under $\approx_\tau$.

- **Absorbing τ-States**: In some cases, states that only have τ-transitions may be absorbed into the next observable state. This allows us to simplify the LTS by removing unnecessary τ-states and focusing on the observable behaviors.

These concepts are critical when converting must tests to LTL formulas, ensuring that the internal behavior of the system is captured without introducing unnecessary complexity.

Since τ-transitions are invisible, any states connected by them should satisfy the same test conditions. This is crucial for ensuring that a test covers all possible behaviors of the system, including those that involve internal actions. The presence of τ-transitions means that certain states can be reached via silent internal actions, without any observable behavior, but these states should still conform to the same test expectations.

When converting test structures to LTL formulas, we must carefully consider τ-closure to align with must-testing semantics. The τ-closure ensures that the formula remains accurate and captures all reachable behaviors, even those that involve invisible τ-transitions. By properly handling τ-transitions, we ensure that the resulting LTL formula correctly reflects the test's intent and accounts for all possible execution paths within the system.

The critical point here is that, although τ-transitions do not directly affect the system's observable behavior, they are essential for correctly modeling the state space and ensuring comprehensive test coverage. Without considering these invisible transitions, the LTL formula may miss valid behaviors, potentially leading to false conclusions during verification.

4.2 Conversion Process

The conversion of a test structure into an equivalent LTL formula $R(T)$ is a key step to ensure that must-testing semantics are preserved. We proceed with the construction by structural induction on the form of the test.

Use of τ-closure. Throughout this construction, we interpret each test structure T as potentially including a sequence of internal (τ) transitions before any observable behavior. Therefore, we define $R(s)$ for every $s \in \tau(T)$, i.e., every state reachable from T by zero or more τ-transitions. This ensures that our translation to LTL captures the full behavior of the test, including the effect of internal transitions.

Base Cases

***STOP*.** The process *STOP* represents a state where the system halts and no further actions can be performed. Since the system may reach a *STOP* state through internal transitions, we define:

$$R(s) = \text{false} \quad \text{for every } s \in \tau(STOP)$$

In general, whenever we evaluate the satisfaction of a subformula in the conversion process, we assume that the system may take zero or more internal (τ) transitions before reaching a state where the formula applies. That is, the next state after an observable action is considered to be any state in the τ-closure of the resulting state. This ensures that our LTL formulas correctly account for silent internal transitions.

***SUCCESS*.** The *SUCCESS* state represents a successful test completion, meaning the system has satisfied all required conditions. Again accounting for internal transitions, we define:

$$R(s) = \text{true} \quad \text{for every } s \in \tau(\text{SUCCESS})$$

This means that reaching any state in the τ-closure of *SUCCESS* satisfies the test.

Inductive Case

Let the test have the form:

$$T = \Box_{a \in A}(a \rightarrow P_a)$$

To capture this structure correctly, we define $R(s)$ for all $s \in \tau(T)$. For each such state s, we assume it must match one of the summands in the choice

(since s is reached via τ-transitions from a test that begins with a visible action). Thus, we define:

$$R(s) = \bigvee_{a \in A} (a \wedge XR(P_a))$$

We justify this inductive step as follows:

$$\begin{aligned} P \text{ must } T &\Leftrightarrow \forall a \in A, (P \text{ must } (a \rightarrow P_a)) \\ &\Leftrightarrow \forall a \in A, (P \models a \wedge XR(P_a)) \quad \text{(inductive assumption)} \\ &\Leftrightarrow P \models \bigvee_{a \in A} (a \wedge XR(P_a)) \end{aligned}$$

That is, P must T iff $P \models R(T)$, as desired. This ensures that after each action a, the corresponding sub-test P_a must be satisfied. The LTL formula $R(T)$ guarantees that the system satisfies all required conditions after executing each action, and by applying it to every state in $\tau(T)$, we capture all possible internal transitions that may occur prior to the observable behavior.

4.3 Examples and Counterexamples

We start with a follow-up to the simple example discussed earlier (in Section 2.2.2). Recall that the two processes

$$\begin{aligned} P_1 &= a \rightarrow STOP \,\Box\, b \rightarrow STOP \\ P_2 &= a \rightarrow STOP \,\sqcap\, b \rightarrow STOP \end{aligned}$$

are differentiated by the test $T = b \rightarrow SUCCESS$, as P_1 must T but $\neg(P_2 \text{ must } T)$. The reason for the failure of T on P_2 is that the internal choice in P_2 is resolved before any interaction with T, and if the result is $a \rightarrow STOP$ then $P_2 \parallel T$ deadlock before T can become $SUCCESS$.

The inclusion of τ-closure in the conversion process apparently alters this reasoning: $\tau(P_2)$ includes states where both ‘a‘ and ‘b‘ are available as first actions. Since the τ-closure considers all possible outcomes of internal nondeterminism, we must check whether each possible outcome satisfies the formula $R(T) = b \wedge X \text{ true}$. Since P_2 can internally resolve to the ‘a‘ branch (in which case the formula fails), we conclude that $P_2 \not\models R(T)$. The claim holds because, while τ-closure reveals both ‘a‘ and ‘b‘ as potential starting actions, the formula requires ‘b‘ to be the first action. Thus, not all behaviors of P_2 satisfy the formula, and therefore, $P_2 \not\models R(T)$. On the other

hand it is quite obvious that $P_1 \models R(T)$ and so the processes continue to be differentiated by $R(T)$.

We now continue with concrete examples and counterexamples to show why it is critical to handle internal (τ) transitions correctly when converting must-tests into LTL formulas. We also explain what we mean by a "state" in our system, as it plays a central role in our definitions.

Recall that in our model, a *state* represents a specific configuration of the system at a given moment. In both Labeled Transition Systems (LTS) and Kripke structures, states capture all the information about the system that is relevant to its behavior. When converting tests, we must ensure that the behavior observed in these states—both observable actions and unobservable τ-transitions—is accurately reflected in the resulting LTL formula.

Example: Validation with τ To illustrate why it is necessary to properly handle τ-transitions, consider an LTS with states $\{s_0, s_1, s_2, s_3\}$ and transitions:

$$s_0 \xrightarrow{a} s_1, \quad s_1 \xrightarrow{\tau} s_2, \quad s_2 \xrightarrow{b} s_3.$$

Here, s_0, s_1, s_2, and s_3 are the states of our system, each representing a distinct configuration. The action a is observable, whereas the transition labeled τ is an internal (silent) transition that we do not see externally.

Now, suppose we have a test

$$T = a \rightarrow (b \rightarrow \text{SUCCESS}),$$

which requires that once action a occurs, the system must eventually perform action b for the test to pass.

We introduce the notation τ^* to represent zero or more consecutive internal transitions. This concept helps us account for silent steps that occur between observable actions.

While τ^* is not an LTL operator, we use it here informally to express that the system may move through a sequence of internal states before reaching a state that satisfies the next part of the formula. A more precise conversion of the test T to an LTL formula would involve checking the satisfaction of the next subformula in all states reachable via τ-transitions after action a.

Thus, instead of writing an invalid LTL formula like

$$R(T) = a \wedge X(\tau^* \wedge b \wedge X\,\text{true}),$$

we clarify that the intended meaning is: after executing a, the system may undergo a sequence of τ-transitions, and eventually reach a state where b

occurs, followed by success. That is, $b \rightarrow \text{SUCCESS}$ must hold in *some* state within the τ-closure following the execution of a.

This approach ensures that our conversion accurately captures the behavior of systems with silent transitions and preserves the intended must-testing semantics.

Counterexample: Omitting τ To further emphasize the importance of internal actions, consider an incorrect conversion where τ-transitions are ignored entirely. Suppose we have the same system from the previous example, with transitions:

$$s_0 \xrightarrow{a} s_1, \quad s_1 \xrightarrow{\tau} s_2, \quad s_2 \xrightarrow{b} s_3.$$

Consider the test:

$$T = a \rightarrow (b \rightarrow \text{SUCCESS}),$$

which expects the system to perform a, then eventually b, to pass the test.

Now, consider a naive LTL conversion that simply treats this as:

$$R'(T) = a \wedge X(b \wedge X\,\text{true}),$$

This formula assumes that after executing a, the system must immediately reach a state where b holds. However, in our system, action b is only reachable after an internal τ-transition — so $R'(T)$ would incorrectly consider the system as failing the test, even though it actually satisfies it under must-testing semantics.

To address this, we recall our earlier notation τ^*. Thus, the correct interpretation of $R(T)$ is not a direct LTL formula involving τ^*, but rather a semantic condition: after executing a, the formula $b \rightarrow \text{SUCCESS}$ must hold in some state within the τ-closure of the resulting state. This preserves the intended semantics of the must test and avoids incorrectly rejecting valid system behavior.

This counterexample highlights why it is essential to incorporate τ-closure into our conversion process. Ignoring internal transitions may lead to LTL formulas that miss valid behaviors, resulting in incorrect verification outcomes.

4.4 Compact LTL Formulas for Tests with Loops

In the previous sections, we explored the process of converting test structures into equivalent LTL formulas, focusing on the direct translation of actions

and conditions into temporal logic expressions. However, a significant challenge arises when dealing with test structures that contain loops[17]. Such loops can lead to infinitely repeating behavior in the resulting LTL formulas, making them complex, inefficient, and difficult to analyze.

To address this challenge, we propose a method for compacting LTL formulas by recognizing and optimizing loops, similar to the approach used in CTL formula compaction for failure trace tests [3]. By identifying loops within the test structure and encoding them more succinctly, we aim to produce LTL formulas that are both compact and easier to interpret, while still preserving their intended semantics.

Before we proceed, we recall our use of the notation τ^* to represent zero or more consecutive internal transitions. While τ^* is not part of standard LTL syntax, we use it as an informal semantic shorthand to indicate that the next observable action or condition is satisfied in a state reachable through such internal transitions.

For example, consider a test structure that requires the observable action a to be followed by observable action b, even when interleaved with silent transitions. The correct interpretation of the corresponding LTL requirement is:

> After performing a, the system should eventually perform b, possibly after a sequence of internal τ-transitions.

To express this informally, we may write:

$$\text{Intended meaning of } R(T): \quad a \wedge X(\tau^* \Rightarrow (b \wedge X\,\text{true}))$$

Here, the τ^* component indicates that the system may undergo a sequence of silent transitions before reaching a state where b holds. We emphasize that this is not a valid LTL formula in the formal sense, but a descriptive notation meant to convey the use of τ-closure during satisfaction evaluation.

By contrast, omitting τ^* from the reasoning—as in writing $R(T) = a \wedge X(b \wedge X\,\text{true})$—would incorrectly assume that b must be observed immediately after a, ignoring the presence of internal transitions. Such an omission leads to a failure to capture valid behaviors and would result in an incorrect interpretation of the test's semantics.

Similarly, as discussed in the counterexample earlier, omitting τ^* causes the resulting LTL expression to misrepresent the system's actual execution paths. Internal transitions can delay the observation of required actions,

and failure to account for them can cause a system that truly satisfies the test to appear as failing.

Therefore, any compacted LTL formula or reasoning about system behavior must integrate the concept of τ^* appropriately in order to maintain semantic fidelity with the original must-testing structure. This ensures that the resulting LTL formulas accurately reflect both observable and silent behaviors, allowing for efficient and correct model checking even in the presence of loops and internal nondeterminism.

4.4.1 Detecting and Compacting Loops in Test Structures

The Problem of Infinite Repetition When translating a test structure that contains loops, the corresponding LTL formula may inadvertently become infinitely repetitive. This redundancy not only inflates the formula but also makes model checking and other verification techniques impossible. Instead of explicitly encoding an infinite sequence, we aim to represent loops in a way that captures their behavior without unnecessary repetition.

Consider a simple test structure where a sequence of actions repeats indefinitely:

$$a_1 \longrightarrow a_2 \longrightarrow a_3 \longrightarrow a_1 \longrightarrow a_2 \longrightarrow a_3 \longrightarrow \dots$$

A straightforward LTL representation might be:

$$\varphi = (a_1 \wedge a_2 \wedge a_3 \wedge \dots) \wedge \mathsf{X}(\varphi)$$

This formulation leads to an infinitely expanding sequence of actions. To manage this, we introduce the concept of a *loop entry action*, which serves as a marker for when the loop begins. Instead of unrolling the loop indefinitely, we construct a compact representation by referring back to this entry action.

Loop Compacting Our method for compacting loops in LTL formulas consists of three key steps:

1. **Identifying the Loop**: A loop is detected when a sequence of actions repeats within the test structure. For example, if a sequence $a_1 \longrightarrow a_2 \longrightarrow a_3$ recurs, we treat it as a loop [3].

2. **Marking the Loop Entry**: The action where the loop begins (e.g., a_1) is designated as the loop entry action.

3. **Compacting the Loo**p: Instead of writing an infinite sequence, we replace it with a succinct formula that captures the repetitive nature of the loop:
$$\mathsf{F}(\text{start}(a_1)) \rightarrow \mathsf{X}(\varphi) \wedge \mathsf{XX}(\varphi) \wedge \ldots$$
This ensures that starting from a_1, the sequence of actions repeats indefinitely, without explicitly unrolling them in the formula.

4.4.2 Inductive Construction of Compact LTL Formulas for Nested Loops

Many test structures feature nested loops, where one loop is contained within another. To efficiently construct LTL formulas for such cases, we adopt an inductive approach, starting with the innermost loop and building outward.

We proceed along the following example. Consider a test structure with two nested loops:

$$\begin{aligned} a_1 \rightarrow a_2 \rightarrow a_3 \rightarrow a_1 \quad & \text{(Inner loop)} \\ b_1 \rightarrow b_2 \rightarrow (a_1 \rightarrow a_2 \rightarrow a_3 \rightarrow a_1) \rightarrow b_3 \quad & \text{(Outer loop)} \end{aligned}$$

We construct the LTL formula as follows: The inner loop formula becomes:

$$\mathsf{F}(a_1) \rightarrow \mathsf{X}(a_2) \wedge \mathsf{X}(a_3) \wedge \mathsf{XX}(a_1)$$

and then we integrate the inner loop into the outer loop as follows:

$$\mathsf{F}(b_1) \rightarrow \mathsf{X}(b_2) \wedge \mathsf{X}(\text{Innermost Loop Formula}) \wedge \mathsf{X}(b_3)$$

This ensures correctness while keeping the formula concise.

4.4.3 Algorithm for Handling Deadlocks and Infinite Loops in LTL Conversion

In this section, I present a general-purpose algorithm designed to address potential deadlocks and infinite loops when converting must-test structures into LTL formulas. The goal is to ensure that for every loop in the test structure, the resulting LTL formula enforces an eventual exit, thus preventing the system from getting stuck in non-progressing states.

Overview: The algorithm begins by obtaining the initial LTL formula from a given test structure using the conversion function $R(T)$. It then

iterates over each detected loop in the test structure. For each loop, the algorithm identifies any exit transitions—that is, actions that lead to a state outside the loop. If no such exit exists, a default exit condition is defined. Finally, the algorithm augments the LTL formula with a condition that, eventually, an exit action or the default exit condition must occur. This additional constraint guarantees that the system will eventually progress, even in the presence of infinite looping behavior.

Algorithm 1 Handling Deadlocks and Infinite Loops in LTL Conversion

procedure HANDLEDEADLOCKS(TestStructure T)
  $\phi \leftarrow R(T)$ ▷ Convert the test structure T into its initial LTL formula
  for all loops L in T **do**
    ExitActions $\leftarrow \{a \in L : \exists\,(s, a, s') \text{ with } s' \notin L\}$
    if ExitActions $= \emptyset$ **then** ▷ No exit transition detected for loop L
      Define a default exit condition ExitCond(L) for L
      $\phi \leftarrow \phi \wedge F(\text{ExitCond}(L))$
    else
      $\phi \leftarrow \phi \wedge F\Big(\bigvee_{a \in \text{ExitActions}} a\Big)$
    end if
  end for
  return ϕ
end procedure

Detailed Explanation:

1. **Initialization:** We start by applying our conversion function $R(T)$ to the test structure T, which produces an initial LTL formula ϕ. This formula encodes the basic behavior of T without yet addressing the issues of deadlocks or infinite loops.

   The challenge in this step is that the presence of loops with internal transitions could in theory create infinite scenarios. There are multiple ways to avoiding such loops, one of the most straightforward solution being to limit the search depth to a maximum level d. We ensure a complete formula ϕ as long as d exceeds the number of states in the test T.

2. **Loop Analysis:** The algorithm then iterates through every loop L detected in the test structure. A loop is defined as a segment of T where a sequence of actions repeats indefinitely.

3. **Exit Transition Identification:** For each loop L, we create a set called ExitActions that contains all actions a for which there exists a transition (s, a, s') with s' not belonging to L. These actions represent valid exits from the loop.

4. **Handling the No-Exit Case:** If no exit transitions are found (ExitActions is empty), the algorithm defines a default exit condition ExitCond(L). This condition should capture any observable behavior that indicates progress, even if it is not explicitly defined as an exit in the loop. We then enforce that eventually (F) this default condition must hold.

5. **Augmenting the LTL Formula:** If exit actions are detected, the algorithm augments the LTL formula by enforcing that eventually at least one of these exit actions occurs. This is done by appending the condition $F\big(\bigvee_{a \in \text{ExitActions}} a\big)$ to ϕ.

6. **Final Output:** After processing all loops, the modified LTL formula ϕ is returned. This final formula now guarantees that the system cannot remain trapped in a loop indefinitely, as there is always an enforced eventual transition out of the loop.

4.4.4 Computational Complexity and Implementation Details

The complexity of the algorithm that converts must-tests into compact LTL formula is quadratic in the number of states of the test being converted, as follows. We henceforth use n to denote the number of states in the test structure T.

- **Computing the Initial LTL Formula $R(T)$ (line 2 of Algorithm 1):** The first step involves creating the initial LTL formula $R(T)$ from the test structure T. This includes figuring out which states can be reached through internal τ-transitions. This step requires $O(n \times d)$ time, where d is the depth limit mentioned in the previous section. Indeed, we might need to check up to d transitions for each of the n states.

  As argued earlier, we ensure a complete conversion as long as d exceeds n e.g., for $d = n + 1$. In this case the worst-case run time is in $O(n \times (n + 1)) = O(n^2)$.

- **Main Loop (Lines 3—11 of Algorithm 1):** Next the algorithm goes through each state in the structure, for $O(n)$ iterations. During each iteration we might need to detect loops, which in the worst case involves looking at every state once more which adds up to $O(n)$ time. The overall complexity is thus $O(n^2)$.

Both steps have quadratic complexity, for an overall running time of $O(n^2)$.

Practical Implementation Details. To tackle practical challenges like infinite loops, we implemented a few strategies:

1. **Limited-depth Exploration of Internal Transitions (τ-closure):** To ensure the algorithm always finishes, even in the presence of loops, we limit how deep the search can go when exploring internal transitions. A large enough limit ensure completeness, but in practice the limit d can re reduced to improve the running time.

2. **Efficient Loop Detection:** We use standard graph traversal techniques such as depth-first search (DFS) to quickly and effectively identify loops. We mark states as visited and recognize loops immediately when a state is revisited.

3. **Compact Formula Representation:** To keep the resulting LTL formulas concise, especially when loops occur, we need to apply techniques to compactly represent repeating patterns.

While these compaction techniques help significantly, the resulting formulas can still become large if the test structures have many states or deeply nested loops. In practice the combination of depth limitations and compaction methods generally keeps things manageable. For extremely complicated cases however further optimizations might be necessary.

Practical Testing and Validation. To see how the algorithm performs in practice, we suggest running experiments with test structures of varying complexity:

- **Simple Cases:** Small-scale tests to verify basic functionality and efficiency.

- **Moderate Cases:** Tests with some loops and internal transitions to observe how well the algorithm scales.

- **Complex Cases:** Larger, more intricate tests with many states and nested loops to identify potential performance bottlenecks and optimization opportunities.

These practical experiments will help demonstrate the algorithm's performance and guide future improvements.

Our experiments are relatively restricted. This being said, we did perform a few experiments using NuSMV, which provided clear insights into the algorithm's performance and the feasibility of verification. In these experiments, we modeled various test structures and generated the corresponding LTL formulas, then used NuSMV to check their correctness and performance. The practical runs confirmed that the resulting formulas were effectively manageable in typical verification scenarios. Additionally, performance metrics such as runtime and memory usage indicated good scalability for moderately complex test structures.

Overall, these practical evaluations demonstrate that the developed algorithm can reliably produce efficient and verifiable LTL formulas for a wide range of must-test scenarios, guiding future enhancements and optimizations. We did not notice anything out of ordinary so we did not see the needs to experiment further.

Chapter 5

Conclusion

This thesis explores the relationship between must-testing and Linear Temporal Logic (LTL), focusing on the systematic conversion of must-test conditions into equivalent LTL formulas. The goal was to enhance formal verification techniques by leveraging LTL expressive power to ensure that systems satisfy their required properties across all possible executions, including those involving internal (τ) transitions.

The proposed conversion process is built on formal semantics and structural induction, ensuring correctness while preserving must-testing semantics. A key challenge in this conversion was handling τ-transitions, which are critical in concurrent systems where unobservable actions influence state evolution. By incorporating τ-closure, we ensured that states connected by internal transitions remain semantically equivalent, maintaining the intended test behavior.

To validate the correctness of our approach, we provided both examples and counterexamples, demonstrating how an incomplete translation could lead to incorrect conclusions. Additionally, we addressed the issue of LTL formula complexity when dealing with test structures that contain loops. By adapting techniques from CTL formula compaction, we proposed a method to reduce redundancy in LTL representations, making them more efficient for model checking without sacrificing expressiveness.

The findings in this thesis contribute to formal verification by bridging the gap between must-testing and temporal logic-based verification. We need to mention however that we only went one way (from must tests to LTL) so a complete equivalence is not established yet. We believe that such an equivalence exists, and so developing the same kind of algorithmic conversion from LTL formulae to must-tests is an obvious immediate extension of our research.

Looking farther ahead, there are several promising directions for future research. One avenue is optimizing LTL formula generation using automata-based techniques to improve efficiency. Additionally, extending this approach to richer temporal logics, such as CTL* or HyperLTL, could offer deeper insights into verification strategies for concurrent and distributed systems.

By systematically translating must tests into LTL formulas, this work provides a structured framework for verifying system behaviors. Formal methods have been split into logical and algebraic approaches since forever, with both approaches having their advantages and disadvantages. Allowing a mixed, part algebraic and part logical specification offers obvious advantages, combining the advantages of both frameworks without introducing any major disadvantage. A system engineer can specify parts of the system logically and other parts algebraically, whichever kind of specification is more convenient for the particular sub-system under scrutiny. An algorithm can then simply take over and construct an overall specification that can then be used in a model checker.

Bibliography

[1] Anderson, R., & Davis, S. (2004). Büchi automata in real-time verification. *Real-Time Systems*, 29(2), 157–180.

[2] Broy, M., & Jonsson, B. (Eds.). (2005). *Model-Based Testing of Reactive Systems*. Lecture Notes in Computer Science, vol. 3472. Springer.

[3] Bruda, S. D., & Zhang, Z. (2009). *Model Checking is Refinement: Computation Tree Logic is Equivalent to Failure Trace Testing*. Technical Report 2009-002, Department of Computer Science, Bishop's University, August 2009.

[4] Bruda, S. D., Singh, S., Uddin, A. F. M. N., Zhang, Z., & Zuo, R. (2019). A constructive equivalence between Computation Tree Logic and failure trace testing. *arXiv preprint arXiv:1901.10925*.

[5] Clarke, E., Grumberg, O., & Peled, D. (1999). *Model Checking*. MIT Press.

[6] Cleaveland, R., & Lüttgen, G. (2000). A semantic theory for heterogeneous system design. In *Proceedings of FSTTCS 2000: Foundations of Software Technology and Theoretical Computer Science (LNCS 1974)*, pp. 312–324. Springer.

[7] Dai, C., & Bruda, S. D. (2008). A testing framework for real-time specifications. In *Proceedings of the 9th IASTED International Conference on Software Engineering and Applications (SEA 2008)*, Orlando, FL, November 2008, pp. 1–8.

[8] Emerson, E. A., & Clarke, E. M. (1981). Using branching time temporal logic to synthesize synchronization skeletons. *Science of Computer Programming*, 2(3), 241–266.

[9] Hoare, C. A. R. (1978). Communicating sequential processes. *Communications of the ACM*, 21 (8), 666–677.

[10] De Nicola, R., & Hennessy, M. (1985). Testing equivalences for processes. *Theoretical Computer Science*, 34(1), 83–133.

[11] De Nicola, R. (1996). Algebraic specifications and temporal logic: A constructive approach. *Journal of Symbolic Computation*, 21(4), 381–398.

[12] Pnueli, A. (1977). The temporal logic of programs. In *Proceedings of the 18th Annual Symposium on Foundations of Computer Science (FOCS 1977)*, pp. 46–57. IEEE.

[13] Roscoe, A. W. (2005). The Theory and Practice of Concurrency. Prentice-Hall.

[14] Uddin, A. F. M. N. (2015). *Computation Tree Logic is Equivalent to Failure Trace Testing*. (Master's thesis, Department of Computer Science, Bishop's University, Canada). Available at: https://part.bruda.ca/_media/part/Uddin20150720.pdf.

[15] Schneider, S. (2000). *Concurrent and Real-Time Systems: The CSP Approach*. Wiley.

[16] Utting, M., & Legeard, B. (2006). *Practical Model-Based Testing: A Tools Approach*. Morgan Kaufmann.

[17] Xiao, X., Li, S., Xie, T., & Tillmann, N. (2013). Characteristic studies of loop problems for structural test generation via symbolic execution. In *Proceedings of the 28th IEEE/ACM International Conference on Automated Software Engineering (ASE 2013)*, pp. 246–256. IEEE.

[18] Zuo, R. (2018). *The Equivalence of Computation Tree Logic and Failure Trace Testing under Multiple Conversion Algorithms*. (Master's thesis, Department of Computer Science, Bishop's University, Canada). Available at: https://part.bruda.ca/_media/part/Zuo20180424.pdf.