Algebraic/Logical Verification for Real-Time Reactive Systems

by

Christian Onyeukwu

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
December 2024

Abstract

We investigate the possibility of using mixed, logical and algebraic approaches to the verification of systems. Constructive equivalence ensures that different models of a system exhibit identical behaviors with respect to specified properties, crucial for maintaining system integrity and safety. We show that it is possible to integrate the formal frameworks of timed temporal logic (namely, TCTL) and timed process algebra (namely, TCCS). For this purpose we show that we can algorithmically determine whether a given TCTL formula and a given TCCS specification are equivalent.

While we effectively show that conversion algorithms between the two frameworks exist, we fall short of providing such algorithms. However, we use relatively simple examples such as traffic light control, railroad crossing, and elevator control systems to suggest a way forward toward these algorithms.

Our investigation opens the study of mixed, algebraic and logical specifications of large real-time systems. Such an approach will greatly improve the scalability of real-time formal methods, but to the best of our knowledge has never been tried before.

Acknowledgements

My supervisor, Prof. Stefan D. Bruda made me start researching the theoretical computing on formal techniques and for that I would be forever thankful to him. I am grateful to him for all his help and encouragement. He provided me with lots of useful references and helped me greatly throughout the entire writing process of my thesis. This study would not have been performed in its present form without his consistent guidance. My appreciation also extends to the Department of Computer Science and Management and as well my course professors for their support during my coursework.

Big thank you to my amazing family, wife, Mrs. Faith Onyeukwu and our kids Jett and Alex Nduka-Onyeukwu for sacrificing a lot of sleep with me most nights keeping alert so I would work through this thesis. Well, I should not miss to thank my friends (classmates), who did all help they could for solving out the problems during challenging times.

Extensive thanks to everyone that assisted me over the several months while this project was in progress.

Contents

Chapter 1

Introduction

1.1 Motivation

Computers now narrowly practically every aspect of our daily lives. The globe has become a global village thanks to the Internet. Recently, computer system applications have been used in every aspect of daily life, including production, communication, entertainment, education, and health care. Artificial Intelligence (AI) is trending now. It is certain that human life will speed up, behaviors will alter, and businesses will undergo significant transformation by AI powered technologies in the nearest future. Acknowledging the accuracy of system behaviors and security is the obstacle that computer scientists and stakeholders must overcome. Consequently, system verification is required.

For ages, there has been a manner for evaluating computer systems. It has been established that the oldest testing method is empirical system verification [23, 71, 81]. Empirical testing is gathering information from the testing procedure and utilizing it to inform choices on the functionality and quality of the product. A continuous feedback loop is frequently used, with modifications made in response to outcomes as they are seen. It might incorporate both white-box (unit testing) and black-box testing (user testing) methodologies. The emphasis is on obtaining information and coming to conclusions using empirical evidence. This non-formal approach feeds a system with input, watches the output, and confirms that the output matches the input's predicted value. Such testing can never prove accuracy because it is unable to verify every potential combination of inputs. However, it can refute correctness. The next verification technique to be developed historically is deductive verification [46, 78]. Deductive verification is a process in software engineering and formal methods where the correctness of a system or software is proven through deductive reasoning and mathematical methods. The goal is to formally demonstrate that a program or system meets its specified requirements, ensuring that it behaves correctly under all possible conditions. It entails manually

producing proofs of program correctness using a set of axioms and inference guidelines. Program proofs are a time-consuming and highly skilled expert-dependent method of providing authoritative proof of correctness.

Numerous methods have been devised to carry out program verification automatically in a way that is like deductive reasoning but more automated. The broad category of verification techniques is known as formal techniques.Formal techniques are mathematical methods used to rigorously specify, develop, and verify software and hardware systems.These techniques provide a foundation for proving system properties and ensuring correctness without exhaustive testing [22]. The key formal techniques include, model checking,theorem proving and equivalence checking [10, 25, 14, 1]. The general method involves automatically comparing a system to a formal specification. Model checking and model-based testing are the two formal techniques approaches that gained traction. Their origins are in deductive reasoning and simulation, respectively. On the other hand, these formal techniques are reliable, comprehensive, and mostly automatic. They have shown their worth over time and are presently widely utilized in the computer sector.

Algebraic and logical are the two primary categories into which formal system specifications [22] and implementations fall. The first is in favor of refinement when a system's definition and implementation are represented by a single algebraic formalism that has a refinement relation attached [85, 84]. A valid implementation is one that improves upon its specification. Traditional refinement relations are either behavioural equivalences or preorders [21, 35], with process algebra [49], labeled transition systems [22], and finite automata [61] being frequently utilized because they frequently describe the system transitionally. Model-based testing is one common example [87]. In the second approach to conformance testing, assertive constructions are preferred; the attributes of the system requirements and implementations are described using various formalisms [22, 23]. While implementations are typically stated in a logical language, specifications

Model-based testing (MBT) is a formal technique used in software testing. It involves creating a model that represents the expected behavior of a system and then generating test cases based on this model. The primary goal is to systematically derive test cases from the model to ensure comprehensive testing of the system. MBT is considered a formal technique because it relies on well-defined models and mathematical concepts to guide the testing process. A well-known black box testing method for creating test cases is MBT [41, 90]. In MBT, some model types often referred to as test models are created or taken from previous stages of the software lifecycle (e.g., requirements or design) for generation of test cases. Since a few decades ago, a great deal of work on MBT has also been done by researchers in the field of formal methods. See [21, 22, 67, 72, 86]. For instance, in [86], a formal method notation called Labelled Transition Systems (LTS) was used in an MBT approach. In [42], a method for determining finite-state machines (FSMs) was introduced; these FSMs can then be utilized in MBT. In MBT, a system's specification is provided

algebraically, and the underlying semantics are provided in an operational way as LTS, or occasionally as a finite automaton (a special, limited type of LTS). Usually, an abstract description of the system's intended behavior is included in such a specification. The same formalism either finite or infinite LTS is used to model the system that is being tested. After that, tests are derived from the specification in a methodical and formal manner and applied to the system that is being tested. Soundness and completeness are guaranteed by the manner the tests are created.

In contrast, the system specification in model checking is provided in a temporal logic format [23, 28, 29, 74]. Consequently, the specification is a (logical) description of the intended system characteristics. A formal specification language called TCTL (Timed Computation Tree Logic) is used to specify temporal features of systems with a time concept. In model checking a formal verification method used to determine whether a system model meets a certain set of properties TCTL is very well-liked. Computation Tree Logic (CTL) is extended by TCTL, which adds temporal operators, especially for timed system reasoning.

The study of process algebra, on the other hand, as the basis for the semantics of concurrent computation has proven to be a fruitful endeavor, yielding nearly constant discoveries over the past ten years regarding the mathematical structure and useful advancement of concurrent processes, whether mechanical or not [66]. Milner's Calculus of Communicating Systems (CCS) of [64] and lately [63] was one of the first process algebra approaches examined as a mathematical model of concurrency, and this algebra is still being researched and expanded upon in numerous ways. ACP (Algebra of Communicating Processes) by [13], Boudol's Meije calculus of [16], Hennessy's process language [46], and Hoare's Communicating Sequential Processes (CSP) of [50] and [20] are other significant methods in process algebra. Algebra is a form of mathematics that simplifies difficult problems by using symbols to represent variables, calculus, and their relations. Algebra enables complicated problems to be expressed and investigated in a formal and rigorous process. The process algebra is a set of formal notations and rules for describing algebraic relations of software processes. Wang and his colleagues found that the existing work on process algebra and their timed variations.[77, 15, 79, 92],can be extended to a new form of expressive mathematics, now the Real-Time Process Algebra(RTPA)

In this research, we will focus on the timed calculus of communication system (CCS) as RTPA model. The Calculus of Communication Systems (CCS) is an extenstion of the traditional process algebra developed by Robin Milner in the late 1970s and early 1980s [64, 63]. It's a formal method used to model and analyze the behavior of real-time systems. In CCS a software system is perceived and described mathematically as a set of coherent processes. A computational action that modifies a system's inputs, outputs, and/or internal variables to change it from one state to another is referred to as a process in RTPA. A process might be as simple as a single meta-process or as complicated as a multi-process that builds upon the RTPA's process relations formula. We consider applying CCS notations to the formal design

of real-time systems. Our aim is to establish a constructive equivalence in real time systems between algebraic and logical frameworks and to demonstrate a translation between TCCS processes and TCTL formulas such that the behavior of a process in TCCS corresponds to the satisfaction of TCTL formula and vice versa.

1.2 Contribution

Equivalence proving in formal methods involves demonstrating that two different models $\models_1$ and $\models_2$ or descriptions ϕ_1 and ϕ_2 of a system are behaviorally identical. This is crucial for ensuring that different representations, such as an abstract model and its concrete implementation, conform to the same specifications. Equivalence can be checked in various contexts, including algebraic models (Timed Calculus of Communicating Systems - TCCS) and logical models (Timed Computation Tree Logic - TCTL). The process loops from defining the system using two different formalisms such algebraic modeling and logical specification(modeling), Specifying the properties or behaviour that the system should exhibit in both models (specification), determining the criteria for equivalence such as bisumation, trace equivalence, or logical equivalence (methodology), and finally using formal methods to prove that the models meet the equivalence criteria (verification). The process aims to ensure that both models consistently represent the same system behavior. This consistency is crucial for verifying the system properties and using different formal methods. Demonstrating equivalence increases confidence that the system meets its specifications across different levels of abstraction. It also makes models interoperable by facilitating the use of different tools and techniques for verification, leveraging the strength of each formalism. By understanding and applying these formal techniques, system designers and verifiers can ensure that different models of a system are behaviorally equivalent, leading to more robust and reliable systems.

Chapter 2

Preliminaries

In this chapter, we will provide the background knowledge required for temporal logic. We shall focus on the algebraic and logical frameworks we are using in this thesis.

2.1 Algebraic Theories and Specifications

We seize this space to look into the structure of CCS notation and specification. A fundamental problem identified in real-time system software is specification and refinement and the CCS approach for solving the problem are described.

2.1.1 CCS Operational Syntax and Semantics

CCS introduces a limited collection of operators for creating system descriptions from subsystem specifications [64]. Actions are the fundamental building elements of these descriptions and system definitions in all process algebras currently in use. Actions intuitively signify discrete, uninterrupted operations that systems could carry out; certain actions signify internal functioning, while others indicate possible exchanges between the system and its surroundings. CCS is based on a sequential, synchronous model of process communication, and this design choice is reflected in the way the collection of activities is organized. Actions might be internal computing steps or inputs/outputs on ports. Because actions on ports necessitate interaction with the environment in order to occur, they are sometimes referred to as external.

Let Λ represent a countably infinite set of labels, or ports, that do not include the unique symbol τ in order to formalize these intuitions. Then, a CCS action can take one of the three following shapes.

- α, where $\alpha \in \Lambda$, represents the act of receiving a signal on port τ.
- $\overline{\alpha}$, where $\alpha \in \Lambda$. represents the act of emitting a signal on port α.

- τ represents an internal computation step.

In that manner,A_{CCS} is used to stand for the set of all CCS actions; that is,

$$A_{CCS} = \Lambda \cup \{\overline{\alpha} | \alpha \in \Lambda\} \cup \{\tau\}.$$

We refer to the actions α and $\overline{\alpha}$, where $\alpha \in \Lambda$, as complementary, as they represent an input and output action on the same channel. The set $A_{CCS} - \{\tau\}$ then contains the set of external, or visible, actions; the only internal action is τ .

Now that the set A_{CCS} of CCS actions has been established, we may present the operators that the process algebra offers for system construction. In the following, we assume a countably infinite set $\mathscr{C}$ of process variables and that p , p_1, and p_2 represent previously created descriptions of CCS systems. The following constructors are offered by CCS.

- nil represent the terminated process that has finished execution.
- Given $a \in A_{CCS}$, the prefixing operator a. allows an action to be "prepended" onto an existing system description. Intuitively, a.p is capable first of an a and then behaves like p.
- + represents a choice construct. The system $p_1 + p_2$ offers the potential of behaving like either p_1 or p_2, depending on the interactions enabled by the environment.
- | denotes parallel composition. The system $p_1|p_2$ interleaves the execution of p_1 and p_2 while also permitting complementary actions of p1 and p2 to synchronize; in this case, the resulting composite action is a τ .
- If $L \subseteq A_{CCS} - \{\tau\}$ then the restriction operator $\backslash L$ permits actions to be localized within a system. Intuitively, $p \setminus L$ behaves like p except that it is disallowed from interacting with its environment using actions mentioned in L. Note that τ can never be restricted.
- The operator [f] allows actions in a process to be renamed. Here f is a function from A_{CCS} to A_{CCS} that is required to satisfy the following two restrictions
  - $f(\tau) = [\tau]$
  - $f(\overline{a}) = \overline{f(a)}$

  When this is the case, f is called a renaming. The system $p[f]$ behaves exactly like p except that f is applied to each action that p may engage in.
- If $C \in \mathscr{C}$, then C represents a valid system provided that a defining equation of the form $C \triangleq p$ has been given. Intuitively, C represents an "invocation" that behaves like p. This construct allows systems to be defined recursively.

System descriptions created with the use of the operators above are frequently referred to as terms or processes in process-algebraic terminology. To represent the collection of all CCS processes, we employ $\mathscr{P}_{CCS}$.

CCS has an operational semantics built in to accurately describe the execution stages that processes are allowed to take in order to clarify their meanings. A ternary relation, $\rightarrow$, is typically used to specify this semantics; intuitively, $p \xrightarrow{a} p'$ holds if system p is able to perform action a and subsequently behave like p. A set of inference rules for each operator is usually used by process algebras like CCS to define $\rightarrow$ inductively. These guidelines are formatted as follows.

$$\frac{Premises}{Conclusion}(SideCondition)$$

A rule states that, if one has established the premises, and the side condition holds,then one may infer the conclusion. This presentation style for operational semantics is often called SOS, for Structural Operational Semantics, see [73]. For instances;

$$\frac{}{a.p \xrightarrow{a} p}$$

This rule concludes that processes of the form a.p may participate in a and then act like p. The rule has no premises. Take note that the side condition is missing; in these instances, it is taken to be "true." Option. There are two symmetric principles for the choice operator.

$$\frac{p \xrightarrow{a} p'}{p+q \xrightarrow{a} p'} \quad \frac{q \xrightarrow{a} q'}{q+p \xrightarrow{a} q'}$$

These rules basically say that a system of the form p + q "inherits" the transitions that occur in its p and q subsystems.The parallel composition operator has three rules, the first two of which are symmetric.

$$\frac{p \xrightarrow{a} p'}{p|q \xrightarrow{a} p'|q} \quad \frac{q \xrightarrow{a} q'}{q|p \xrightarrow{a} q'|p}$$

These rules indicate that | interleaves the transitions of its subsystems. The next rule allows processes connected by | to interact.

$$\frac{p \xrightarrow{a} p', q \xrightarrow{\bar{a}} q'}{p|q \xrightarrow{\tau} p'|q'}$$

According to this rule, subsystems may synchronize on complementary actions (i.e., inputs and outputs on the same port). Note that the action produced as the result of the synchronization is a τ ; since τ is undefined, this ensures that synchronizations involve only two partners. Restriction. The restriction operator has one rule.

$$\frac{p \xrightarrow{a} p'}{p \diagdown L \xrightarrow{a} p' \diagdown L}(a, \overline{a} \notin L)$$

This rule, which includes a side condition, only allows actions not mentioned in L (or whose complements are not in L) to be performed by $p \diagdown L$. Restriction in effect "localizes" actions in L, since the operator forbids the system's environment from interacting with the system using them. Relabeling. The relabeling operation has one rule.

$$\frac{p \xrightarrow{a} p'}{p[f] \xrightarrow{f(a)} p'[f]}$$

As the intuitive account above suggests, $p[f]$ engages in the same transitions as p, the difference being that the actions are relabeled via f . Process Variables. The behavior of process variables is given by one rule.

$$\frac{p \xrightarrow{a} p'}{C \xrightarrow{a} p'}(C \triangleq p)$$

This rule states that a system C behaves like the body, p, of its definition $C \triangleq p$.

The combination rules of meta-processes in RTPA are governed by a set of algebraic process relations. For instances, $P \parallel Q$ is the operational semantics of parallel that denote relations between system architectural concepts that are functionally parallel or equivalent. $P \rightarrow Q$ implies a sequential relation, $P \circlearrowleft P$ means a recursion which process relation in which a process P calls itself.

2.1.2 Labeled Transition Systems

Given the notion of $\rightarrow$, CCS processes can be understood as a particular kind of state machine. Firstly, we demonstrate how CCS may be understood as a structure known as a labeled transition system, which is made up of a variety of potential system states and transitions.

A labeled transition system (LTS) [21] is a tuple $M = (S, A, \rightarrow, s_0)$ where S is a countable, non empty set of states, $s_0 \in S$ is the initial state, and A is a countable set of actions. The actions in A are called visible (or observable), by contrast with the special, unobservable action $\tau \notin A$ (also called internal action). The relation $\rightarrow \subseteq S \times (A \cup \{\tau\}) \times S$ is the transition relation; we use $p \xrightarrow{a} q$ instead of $(p, a, q) \in \rightarrow$.A transition $p \xrightarrow{a} q$ means that state p becomes state q after performing the (visible or internal) action a.

A path (or run) π starting from state p' is a sequence $p' = p_0 \xrightarrow{a_1} p_1 \xrightarrow{a_2} \ldots p_{k-1} \xrightarrow{a_k} p_k$ with $k \in \mathcal{N} \cup \{\omega\}$ such that $p_{i-1} \xrightarrow{a_i} p_i$ for all $0 < i \leq k$. We use $|\pi|$ to refer to k, the length of π. If $|\pi| \in \mathcal{N}$, then we say that π is finite. The trace of π is the sequence $trace(\pi) = (a_i)_{0<i\leq|\pi|, a_i \neq \tau} \in A^*$ of all the visible actions that occur in the run listed in their order of occurrence and including duplicates. Note in particular that internal

actions do not appear in traces. The set of finite traces of a process p is defined as $Fin(p) = \{tr \in traces(p) : |tr| \in N\}$. If we are not interested in the intermediate states of a run then we use the notation $p \xrightarrow{\omega} q$ to state that there exists a run π starting from state p and ending at state q such that trace $(\pi) = \omega$. We also use $p \xrightarrow{\omega}$ instead of $\exists p' : p \xrightarrow{\omega} p'$. A process p that has no outgoing internal action cannot make any progress unless it performs a visible action. We say that such a process is stable [80]. We write $p \downarrow$ whenever we want to say that process p is stable. Formally, $p \downarrow = \neg(\exists p' \neq p : p \xrightarrow{\in} p')$. A stable process p responds predictably to any set of actions $X \subseteq A$, in the sense that its response depends exclusively on its outgoing transitions. Whenever there is no action $a \in X$ such that $p \xrightarrow{a}$ we say that p refuses the set X. Only stable processes are able to refuse actions; unstable processes refuse actions "by proxy": they refuse a set X whenever they can internally become a stable process that refuses X. Formally, p refuses X (written p ref X) if and only if $\forall a \in X : \neg(\exists p' : (p \xrightarrow{\in} p') \wedge p' \downarrow \wedge p' \xrightarrow{a})$

There are additional definitions of LTS that specify a start state.Known as rooted labeled transition systems, these labeled transition systems are made up of quadruples of the type $Q, A, \rightarrow, qS$, where $qS \in Q$ denotes the start state. However, this chapter's definitions demonstrate that CCS can be thought of as a single LTS. Remember that $\mathscr{P}_{CCS}$ stands for the (infinite) set of syntactically valid CCS system definitions. The transition relation specified in the preceding subsection is represented by $\rightarrow CCS$. Therefore, $\mathscr{P}_{CCS}$, $\mathscr{A}_{CCS}$, and $\rightarrow CCS$ meet the requirements for being an LTS. There are two implications to this discovery. First, by defining some definitions with respect to LTSs, language-independent definitions for things like behavioral equivalencies and refinement orderings can be provided. The potential conversion of individual system descriptions into rooted LTSs is the second effect. A rooted LTS is mathematically represented as the quadruple $\mathscr{P}_{CCS}$,$\mathscr{A}_{CCS}$,$\rightarrow CCS, p$ for every CCS system.

2.1.3 Timed Automata

If we introduce a real-time model, as an extension of finite state automata with real-valued variables for measuring time. The abstraction can be known as Timed automata. [6] introduced timed automata in the early 1990s as finite-state automata equipped with real-valued variables for measuring time between transitions in the automaton.

Timed automata have proven very convenient for modeling and reasoning about real-time systems: they combine a powerful formalism with advanced expressiveness and efficient algorithmic and tool support, and have become a model of choice in the framework of verification of embedded systems.

The timed-automata formalism is now routinely applied to the analysis of real-time control programs [17, 37, 62] and timing analysis of software and asynchronous

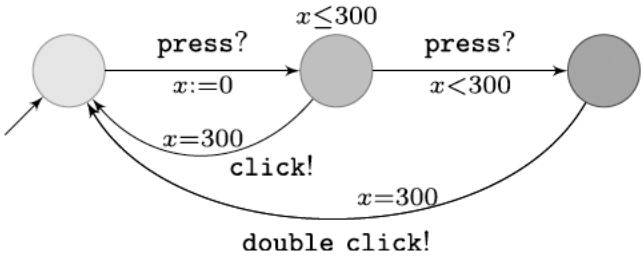

Figure 2.1: Timed automata of simplified mouse clicks.

circuits [18, 17, 88, 89]. Similarly, numerous real-time communication protocols have been analysed using timed automata technology, often with inconsistencies being revealed [43, 82]. During the last few years, timed-automata-based schedulability and response-time analysis of multitasking applications running under real-time operating systems have received substantial research effort [19, 32, 33, 53, 93]. Also, for optimal planning and scheduling, (priced) timed automata technology has been shown to provide competitive and complementary performances to classical approaches [2, 3, 11, 38, 44, 52, 58]. Finally, controller synthesis from timed games has been applied to several industrial case studies[4, 24, 54].

Our model in Figure 2.1 is an example of a (simplified) computer mouse: this automaton receives press events, corresponding to an action of the user on the button of the mouse. When two such events are close enough (less than 300 milliseconds apart), this is translated into a double-click event. Because clock variables are real-valued, timed automata are infinite-state models, where a configuration is given by the location of the automaton and a valuation of the clocks. Timed automata have two kinds of transitions: action transitions correspond to firing a transition of the automaton, and delay transitions correspond to letting time elapse in the current location of the automaton.

Let Σ be a finite set of actions in a set of time domain $\mathscr{R}_{\geq 0}$. A time sequence is a finite or infinite non-decreasing sequence of non-negative reals. A timed word is a finite or infinite sequence of pairs $(a_1, t_1) \ldots (a_p, t_p) \ldots$ such that $a_i \in \Sigma$ for every i,and $(t_i)i \geq 1$ is a time sequence. An infinite timed word is converging if its time sequence is bounded above (or, equivalently, converges).

If we consider a finite set C of variables, called clocks. A (clock) valuation over C is a mapping $v : C \to \mathscr{R}_{\geq 0}$ which assigns to each clock a real value. The set of all clock valuations over C is denoted $\mathscr{R}^C_{\geq 0}$, and 0_C denotes the valuation assigning 0 to every clock $x \in C$.

Let $v \in \mathscr{R}^C_{\geq 0}$ be a valuation and $t \in \mathscr{R}_{\geq 0}$; the valuation $v + t$ is defined by $(v + t)(x) = v(x) + t$ for every $x \in C$. For a subset r of C, we denote by v[r] the valuation obtained from v by resetting clocks in r; formally, for every $x \in r$,

$v[r](x) = 0$ and for every $x \in C\ r, v[r](x) = v(x)$.

The set $\phi(C)$ of clock constraints over C is defined by the grammar

$$\phi(C) \ni \varphi ::= x - y \bowtie k | \phi_1 \wedge \phi_2 (x \in C, k \in \mathscr{Z} and \bowtie \in <, \leq, =, \geq, >)$$

A Timed Automaton [6] is a tuple $(L, \mathscr{L}_0, C, \Sigma, I, E)$ consisting of a finite set L of locations with initial location $\mathscr{L}_0 \in L$, a finite set C of clocks, an invariant1 mapping $I : L \rightarrow \phi(C)$, a finite alphabet Σ and a set $E \subseteq L \times \phi(C) \times \Sigma \times 2C \times L$ of edges. We shall write $l \xrightarrow{\varphi, a, r} \mathscr{L}'$ for an edge $(\varphi, a, r, \mathscr{L}') \in E$.

The operational semantics of a timed automaton A = $(L, \mathscr{L}_0, C, \Sigma, I, E)$ is the (infinite-state) timed transition system $[[A]] = (S, s_0, R_{\geq 0} \times \Sigma, T)$ given as follow: $S = \{(\mathscr{L}, v) \in L \times \mathscr{R}^C_{\geq 0} | v \models I(\mathscr{L})\}$, $s_0 = (\mathscr{L}_0, 0_C)$, T = $\{(\mathscr{L}, v) \xrightarrow{d, a} (\mathscr{L}', v') | \forall d' \in [0, d] : v + d' \models I(\mathscr{L}) \wedge \exists \mathscr{L} \xrightarrow{\varphi, a, r} \in E : v + d \models \varphi \wedge v' = (v + d)[r]\}$.

2.1.4 Event Clock Automata

Event Clock Automata (ECA) is a type of timed automaton designed specifically for the verification of timed properties using event clocks. Unlike standard timed automata, where clocks may be reset at any transition, ECA restricts the usage of clocks by associating each clock with specific events. This allows for efficient model checking of Timed Computation Tree Logic (TCTL) properties, as the event clocks reset only when specific events occur, simplifying the tracking of time and reducing the state space.

Formally, ECA can be defined as a turple $\mathcal{A} = (Q, \Sigma, C, I, q, F)$ where Q is finite set of states, Σ is a finite alphabet of events (or actions), C is finite set of event clocks. Each clock $c \in C$ is associated with an event in Σ and is reset every time that event occurs.

The essential power of nondeterminism in timed automata lies in its ability to reset clocks nondeterministically. An Event Clock Automaton (ECA) is a special type of timed automaton designed to track and reason about the timing of events within a system. Unlike regular timed automata, ECAs specifically use event clocks, which are reset not based on the passage of time but on the occurrence of certain events in the system [7]. This makes them well-suited for systems where the timing of certain events and the relationships between those events need to be monitored and verified. Event Clock Automata are primarily used in model checking and formal verification to reason about the timing of actions and events in real-time systems, particularly when verifying systems against timed temporal logic properties, such as TCTL.

An event clock automaton $\mathcal{A}$ can be deterministically transformed with relative ease. Initially, we can convert $\mathcal{A}$ into an automaton $\mathcal{B}$ such that the set of guards $\mathcal{G}$ utilized on the transitions is minimal. This means that for any pair of guards g and g' within $\mathcal{G}$, it is impossible to find a clock valuation that satisfies both simultaneously.

Following this, a subset construction can be applied to this automaton. Let us denote $B = \langle V, V^0, V^F, X, E\rangle$. Subsequently, we can create a deterministic event recording automaton $C = \langle 2^V, V^0, F, X, E^{'}\rangle$, where for every subset $S \subseteq V$, an input symbol $a \in \Sigma$, and a guard $g \in \mathcal{G}$, the relation $(S, a, g, S^{'}) \in E^{'}$ holds true if $S^{'} = v^{'} \in V | \forall v \in S.(v, a, g, v^{'}) \in E$. The collection $\mathcal{F}$ consists of sets $S \subseteq V$ such that their intersection with the final states V^F is non-empty ($S \cap V^F \neq \emptyset$). It is evident that C operates deterministically and recognizes the same language as B. However, it is crucial to note that a similar construction would not succeed for timed automata due to the possibility of having two states within a set §, namely v and $v^{'}$, connected by edges (v, g, λ, v_1) and $(v^{'}, g^{'}, \lambda^{'}, v_1^{'})$, where the outputs differ ($\lambda \neq \lambda^{'}$).

Theorem 2.1. [7] *Event Clock Automata are effectively closed under complementation. Additionally, a given timed automaton $\mathcal{A}$ and an event clock automaton $\mathcal{B}$, the challlenge of checking wether $L(\mathcal{A}) \subseteq L(\mathcal{B})$ is PSPACE-complete.*

2.1.5 (Bi)simulation Checking

The operational semantics of timed automata is provided in terms of timed transition systems, which are actually equivalent to standard labeled transition systems with labels (d, a) consisting of a delay and a letter, as explained in Sect. 2.1.3. Therefore, any ordering and behavioral equivalency defined on labelled transition systems may be understood across timed automata. The following idea of timed (bi)simulation is specifically derived from the classical concepts of simulation and bisimulation [63, 70]:

Definition 2.1. *Let $A = (L, \mathscr{L}_0, C, \Sigma, I, E)$ be a timed automaton. A relation $R \subseteq L \times \mathcal{R}_{\geq 0}^C \times L \times \mathcal{R}_{\geq 0}^C$ is a timed simulation provided that for all $(\mathscr{L}_1, v_1)R(\mathscr{L}_2, v_2)$, for all $(\mathscr{L}_1, v_1) \xrightarrow{d,a} (\mathscr{L}_1^{'}, v_1^{'})$ with $d \in \mathcal{R}_{\geq 0}$ and $a \in \Sigma$, there exists some $(\mathscr{L}_2^{'}, v_2^{'})$ such that $(\mathscr{L}_1^{'}, v_1^{'})R(\mathscr{L}_2^{'}, v_2^{'})$.*

A timed bisimulation is a timed simulation which is also symmetric, and two states $(\mathscr{L}_1, v_1), (\mathscr{L}_2, v_2) \in [[A]]$ are said to be timed bisimilar, written $(\mathscr{L}_1, v_1) \sim (\mathscr{L}_2, v_2)$, if there exists a timed bisimulation R for which $(\mathscr{L}_1, v_1)R(\mathscr{L}_2, v_2)$.

Note that $\sim$ is itself a timed bisimulation on A, which is easily shown to be an equivalence relation and hence transitive, reflexive, and symmetric. Also as usual timed bisimilarity may be lifted to an equivalence between two timed automata A and B by relating their initial states.

Consider the four automata A, X, U and D in Figure 2.2 (identifying the automata with the names of their initial locations). Here (U, v) and (D, v) are timed bisimilar as any transition $(U, v) \xrightarrow{d,a} (V, v^{'})$ may be matched by either $(D, v) \xrightarrow{a} (G, v^{'})$ or $(D, v) \xrightarrow{a} (E, v^{'})$ depending on whether $v(y) > 2$ or not after delay d. Infact, it may easily be seen that U and D are the only locations of Figure 2.1.3 that are timed bisimilar (when coupled with the same valuation of y). E.g., A and X are not timed

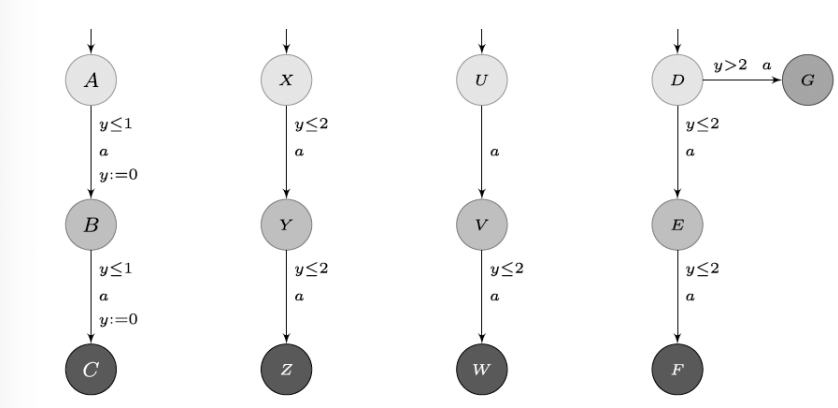

Figure 2.2: Autamata A,X, U, and D.

bisimilar since the transition $(X, 0) \xrightarrow{1.5,a} (Y, 1.5)$ cannot be matched by $(A, 0)$ by a transition with exactly the same duration. Instead A and X are related by weaker notion of timed-abstracted bisimulation, which does not require equality of delays. It may be seen that A and X are both time-abstracted simulated by U and D but not time-abstracted bisimilar to U and D. Also, U and D are time-abstracted bisimilar, which follows from the following easy fact:

Theorem 2.2. *Any two automata being timed bisimilar are also time-abstracted bisimilar.*

2.2 Logical Theories and Specification

2.2.1 Kripke Structure

A Kripke structure $\mathscr{K}$, named after the logician Saul Kripke, is a mathematical structure used in modal logic and model checking to represent possible worlds or states and the relationships between them. It serves as a formal semantics for modal logics, providing a way to interpret and reason about model formulas. The structure $\mathscr{K}$ is a model of the formula ϕ. If $\mathscr{K} \nvDash \phi$, then the model checker outputs a counterexample that witnesses the violation of ϕ by $\mathscr{K}$. The generation of counterexamples means that, in practice, falsification (the detection of bugs) can often be faster than verification (the proof of their absence). If every one of the specified initial states of each structure $\mathscr{K}$ satisfies the logic formula, the system then meets the specification [23].

Kripke structures [53] are finite directed graphs whose vertices are labeled with sets of atomic propositions. The vertices and edges of the graph are called "states" and "transitions," respectively. In our context, they are used to represent the possible configurations and configuration changes of a discrete dynamical system.

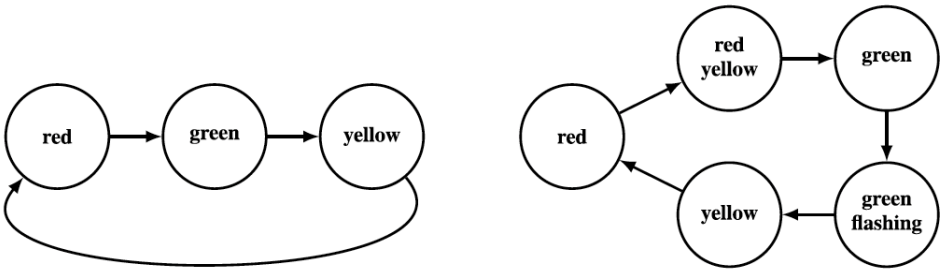

Figure 2.3: Canadian and Austrian traffic lights as Kripke structures.

Formally, a Kripke [29] $\mathscr{K}$ over a set AP of atomic proposition is a tuple $(S, S_0, \rightarrow, L)$, where S is a set of states, $S_0 \subseteq S$ is the set of initial states, $\rightarrow \subseteq S \times S$ is the transition relation, and $L : S \rightarrow 2^{AP}$ is a function that assigns to each state exactly all the atomic propositions that are true in that state. As usual we write $s \rightarrow t$ instead of $(s, t) \in \rightarrow$. It usually assumed [29] that $\rightarrow$ is total, meaning that for every state, $s \in S$ there exists a state $t \in S$ such that $s \rightarrow t$. Such requirement can however, be easily established by creating a "sink" state that has an atomic proposition assigned to it, is the target of all the transitions from states with no other outgoing transitions, and has one outgoing "self-loop" transition back to itself. The system's dynamic behavior represented by a Kripke structure corresponds to a path through the graph.

A path π in Kripke structure is sequence $s_0 \rightarrow s_1 \rightarrow s_2 \rightarrow \cdots$ such that s_{i+1} for all $i \geq 0$. The path originates at state s_0. Numerous paths can begin in any state. It follows that all the pathways start from a particular state s_0. Any state could be the beginning of several paths. As a result, any path that begins at a certain state, s_0, can be represented collectively as a computation tree with nodes with state labels. Such a tree has its root at s_0, and its edges (s, t) are present if and only if $s ß t$. While some temporal logics consider computation pathways separately, others consider computation trees.

2.2.2 TCTL

Timed Computation Tree Logic(TCTL) is the real-time extension of Computation tree logic (CTL). CTL was introduced by Emerson and Clark [26] as a specification language for finite-state systems. If we briefly review it's syntax and semantics, we would come up with the following presentation as follows;

Let AP be a set of atomic propositions. The formulas of CTL are inductively defined as

$$\phi := p | false | \phi_1 \rightarrow \phi_2 | \exists \bigcirc \phi_1 | \exists (\phi_1 \bigcup \phi_2) | \forall (\phi_1 \bigcup \phi_2),$$

where $p \in AP$, and ϕ_1 and ϕ_2, are CTL-formula $\exists \bigcirc \phi$ intuitively means that there is an immediate successor state, reachable by extending one step, in which ϕ holds. $\exists(\phi_1 \bigcup \phi_2)$ means that for some computation path, there exists an initial prefix of the path such that ϕ_2 holds at last state of the prefix and ϕ_1 holds at all the intermediate states. $\forall(\phi_1 \bigcup \phi_2)$ means that for every computation path the above property holds.

Some of the commonly used abbreviations are $\exists \diamond \phi$ for $\exists(true \bigcup \phi)$,$\forall \diamond \phi$ for $\forall(true \bigcup \phi)$,$\exists \Box \phi$,for $\neg \forall \diamond \neg \phi$, and $\forall \Box \phi$, for $\neg \exists \diamond \neg \phi$

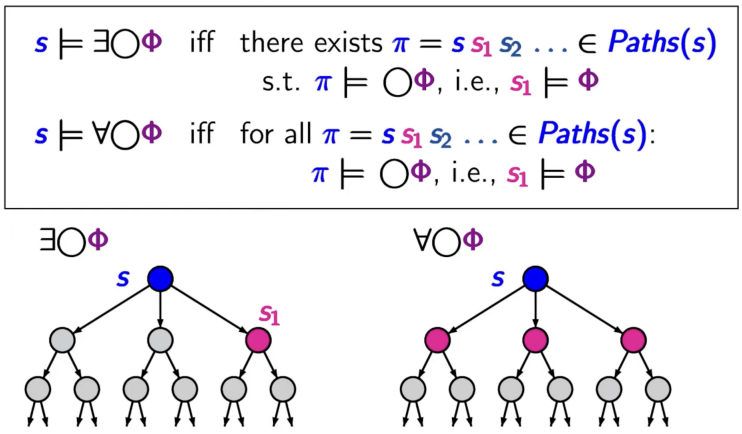

Figure 2.4: The **next** operator semantics.

Formally, the semantics of CTL can be defined to a Kripke structure $\mathscr{M} = (\mathscr{S}, \mu \rightarrow, E)$, where $\mathscr{S}$ is a countable set of states, $\mu : \mathscr{S} \rightarrow 2^{AP}$ gives an assignment of truth values to propositions in each state, and E is a binary relation over $\mathscr{S}$ given the possible transitions. A path is an infinite sequence of states $(S_1, S_2, \cdots)$ such that $\langle S_i, S_{i+1} \rangle \in E$ for all $i \geq 0$.

Given a CTL-formula ϕ and a state $S \in \mathscr{S}$, the satisfaction relation $(\mathscr{M}, S) \models \phi$ (meaning ϕ is true in $\mathscr{M}$ at S) is defined inductively as follows. (since the structure is fixed, we can abbreviate $(\mathscr{M}, s) \models \phi$ to $S \models \phi$):

- $S \models p$ iff $p \in \mu(S)$
- $S \not\models$ false
- $S \models \phi_1 \rightarrow \phi_2$ iff $S \not\models \phi_1$ or $S \models \phi_2$
- $S \models \exists \bigcirc \phi$ iff $t \models \phi$ for some state $t \in \mathscr{S}$ such that $\langle s, t \rangle \in E$
- $S \models \exists(\phi_1 \bigcup \phi_2)$ iff for some path $(S_0, S_1, \cdots)$ with $S = S_0$ for some $i \geq 0$, $S_i \models \phi_2$ and $S_j \models \phi_1$ for $0 \leq j < i$

- $S \models \forall(\phi_1 \bigcup \phi_2)$ iff for every path $(S_0, S_1, \cdots)$ with $S = S_0$, for all some $i \geq 0, S_i \models \phi_2$ and $S_j \models \phi_1$ for $0 \leq j < i$

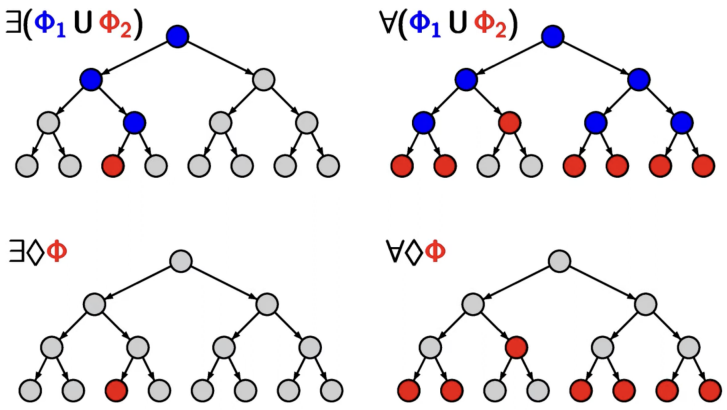

Figure 2.5: The **until** and **eventually** operator semantics.

A CTL-formula ϕ is called satisfiable iff there are Kripke structure $\mathscr{M}$ and a state S of it such that $(\mathscr{M}, S) \models \phi$. However, the Timed Computation Tree Logic(TCTL) is the real-time extension of (CTL) [61] with time constraints on modalities.

In CTL, if we write a formula $\exists \diamond p$, which says along some computation path, p eventually becomes true. CTL does not provide a way to put a bound on the time at which p will become true. A natural and straightforward extension is to put subscripts on the temporal operators to limit their scope in time [56, 39]. For instance, we can write $\exists \diamond_{<5} p$ to say along the computation path p become true *5-time units*. This approach is then used to introduce the explicit time in the syntax

$$\phi ::= p|\neg\phi|\phi_1 \wedge \phi_2|\phi_1 \vee \phi_2|AF_{\leq t}\phi|EF_{\leq t}\phi|AG_{\leq t}\phi|EG_{\leq t}\phi|A(\phi_1 \cup_{\leq t} \phi_2)|E(\phi_1 \cup_{\leq t} \phi_2)$$

where the temporal operators with timing constraints imply:

- $AF_{\leq t\phi}$: "In all future paths,ϕ holds within time t"
- $EF_{\leq t\phi}$: "In some future paths,ϕ holds within time t"
- $AG_{\leq\phi}$: " Globally in all path, ϕ" holds at all time within t
- $EG_{\leq\phi}$: " Globally in some path, ϕ holds at all time within t"
- $A(\phi_1 \bigcup_{\leq t} \phi_2)$: "In all paths, ϕ_1 holds until, ϕ_2 holds within time t"
- $E(\phi_1 \bigcup_{\leq t\phi_2} \phi_2)$: "In some paths, ϕ_1 holds until, ϕ_2 holds within time t"

A path is simply a ω-sequence of states. If we assume that along any particular computation path, there is a unique state at every instant domain R to states of the system.

In TCTL, constraints are typically imposed on either states or paths, which are sequences of states. The timing is monitored using clock variables, enabling the specification of conditions such as "within 10 seconds" or "at least 5 seconds have elapsed." The operators are categorized into two groups: state-based operators (such as AX and EX) and path-based operators (including AG, EG, AF, EF, AU, and EU). Their meaning is summarized in Table 2.1.

Real-time constraints and TCTL operators enable the articulation of intricate timing requirements in the formal verification of systems, thereby ensuring safety, liveness, and fairness within stringent timing parameters. The integration of logical operators with timing constraints in TCTL creates a robust framework for modeling and verifying the temporal behaviors essential for real-time systems. All the operators are thus reactive to real-time constraints because of the time intervals [a,b]. This is important for the verification of temporal properties in real-time systems. They represent concrete lower and upper time bounds in which a specific property must be satisfied or an event must happen, making them particularly useful for real-time systems with hard deadlines and safety-critical tasks.

Integration of these timing constraints in turn helps better manage fulfilment properties for a real-time system.

2.2.3 Model Checking

Model checking is a computer-aided method for the analysis of dynamical systems that can be modeled by state-transition systems.In mathematical logic, programming languages, hardware design, and theoretical computer science, model checking is now widely used for the verification of hardware and software in industry[27]. Timed Computation Tree Logic (TCTL) is a popular formalism used in model checking. The temporal ordering of events within the system is specified by these attributes, which also serve as constraints. Temporal logic, like Timed Computation Tree Logic (TCTL) or Linear Temporal Logic (LTL), can be used to define temporal features in an abstract model. These logics make it possible to formally specify the system's restrictions and temporal linkages. These logics allow for the formal specification of temporal relationships and constraints within the system.

Operator	Interpretation	Operational Semantics
$AX_{[a,b]}$	For all paths, in the next state, a property ϕ holds within the time interval [a,b].	In real-time, suppose we have a system where if a process is active in a current state, then in all possible next states, a certain safety condition must hold within 3 to 5 seconds. If the safety condition doesn't hold in even one next state, $AX_{[3,5]}$(Safety) would be false
$EX_{[a,b]}$	There exists at least one path where, in the next state, a property ϕ holds within the time interval [a,b]	In a real-time network communication protocol, we might want to confirm that there's at least one transition to a recovery state within 2 to 4 seconds after an error occurs. This is written as $EX_{[2,4]}$(Recovery).
$AG_{[a,b]}$	For all paths, a property ϕ holds in every state (globally) at all times within the interval [a,b].	If we want a temperature sensor to stay within safe operating limits at all times winthin 0 to 10 second, we would need $AG_{[0,10]}$(SafeTemperature) to hold,ensuring that every path maintains the temperature constraint throughout the interval.
$EG_{[a,b]}$	There exists a path where a property ϕ holds in every state (globally) in the time interval [a,b].	In a system power management setting, $EG_{[0,5]}$ (NoOverload) would mean that at least one path exists where the system remains free from overloads for up to 5 seconds
$AF_{[a,b]}$	Along all paths, a property ϕ eventually holds within the time interval [a,b].	In real-time, if a request is made for a shared resource, $AF_{[2,6]}$(AccessGranted) would mean that every path guarantees resource access within 2 to 6 seconds.
$EF_{[a,b]}$	There exists at least one path where a property ϕ eventually holds within the time interval [a,b].	In real-time, in an emergency response system, $EF_{[1,3]}$(Evacuate) would indicate that there's at least one path where evacuation is initiated between 1 and 3 seconds after an alarm.
$AU_{[a,b]}$	For all paths, a property ϕ_1 holds until ϕ_2 holds within the time interval [a,b].	In real time situation, for a robotic arm, $AU_{[2,7]}$(ArmEngagedUTargetReached) would mean that on all paths, the arm stays engaged until it reaches the target within 2 to 7 seconds.
$EU_{[a,b]}$	There exists a path where a property ϕ_1 holds until ϕ_2 holds within the time interval [a,b].	In an automated assembly line, $EU_{[5,10]}$ (AssemblyRunningUTaskComplete) means there is a path where the assembly process continues running until the task completes within 5 to 10 seconds

Table 2.1: Temporal operators and their meaning.

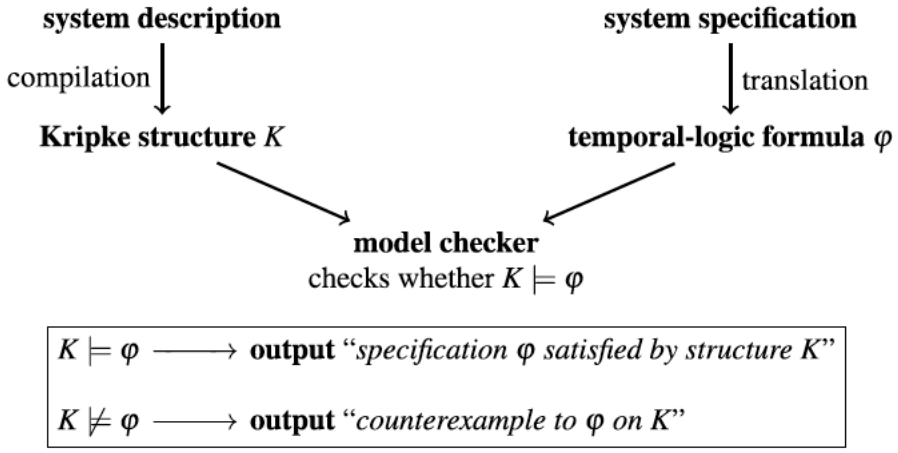

Figure 2.6: The basic model-checking methodology.

Chapter 3

Literature Review

Constructive equivalence is a fundamental concept in real-time system verification that ensures different models of a system act consistently with respect to predetermined attributes. This review of the literature examines the use of process algebra and model checking to demonstrate constructive equivalence in real-time systems. This overview covers fundamental theories, significant applications, significant methodologies, and current advancements in the field.

3.1 Fundamental Theories

The concept of timed automata was introduced by [6] and provides a mathematical framework for the analysis and modeling of real-time systems. Process algebra is relevant to this idea. Temporal restrictions on states and transitions can be set using timed automata, which are finite automata with clocks added to them. A formal vocabulary for explaining and debating concurrent systems is provided by process algebra. CSP (Communicating Sequential Processes) and CCS (Calculus of Communicating Systems) are two instances of this. CCS was introduced by [47] and is the foundation for several process algebra approaches used in real-time systems.

Equivalency and Bisimulation The equivalence of many models can be demonstrated with the understanding of bisimulation. Two systems are bisimilar if they are able to gradually imitate each other's behavior. [70] established bisimulation as a connection.

3.2 Methodologies

Concurrent systems with finite states can be verified automatically using a method called model checking. An algorithmic method to ascertain whether a system model fits a given specification is called model checking, and it is commonly described in

temporal logic [29]. A popular tool for model testing timed automata, UPPAAL was created by [60] and serves as a useful foundation for verifying real-time systems.

The real-time capabilities of process algebra have been enhanced. Two techniques are utilized to introduce temporal restrictions into the process algebra framework: Timed Calculus of Communicating Systems (TCCS) and Timed CSP [80]. These advances make it possible to check timing details and simulate time-dependent behavior using algebraic tools. Examples of real-time system models that illustrate real-world application use cases are the traffic light control system, rail crossing system, and elevator control system [48, 38, 5]. These systems do not exhibit dynamic behavior in the sense of unpredictable or learning-based changes. However, they are considered real-time systems because their operation depends critically on timing constraints and deterministic responses to external events.

3.3 Previous Work

We have not gone too far in exploring the connections between algebraic and logical frameworks of formal definition and verification. Parameterized verification has gained popularity recently and deals with verifying systems with an arbitrary number of components. In [40], methods for parameterized model checking of timed systems were created, allowing the verification of systems with various numbers of processes.

A semantic model for reasoning about real-time system specifications that combines timed processes and formulas in linear-timed temporal logic with a time constant (TLTL) was proposed in Chun Dai's work [30] on testing framework for real-time systems. Negar Nourollahi [68] also conducted a great deal of work on inductive conversion problems in dense time, as well as verification and TCTL model testing of real-time systems on timed automata and timed Kripke structure. The basis of algebraic software system specification is provided by CCS [64], CSP [50], and ACP [13], however the only comprehensive study of the topic is based on computation tree logic (CTL) and its relationship in [85].

3.3.1 Abstraction and Equivalence

There are two different approaches to system verification[68]. Equivalency checking is the first method, and it seeks to determine a semantic equivalent between two systems, one of which is the implementation of the specification provided by the other. The second strategy, known as model checking, seeks to determine whether a particular system meets a condition that is often provided in a temporal or modal logic [12]. Developing techniques like abstraction and bisimulation equivalency try to replace a huge structure with a smaller structure that meets the same features in order to prevent the state explosion problem. By providing a mapping between a small set of abstract data values and the actual data values in the system, the

abstraction is made possible. Finding transition systems that can simulate one another step-by-step because they have the same branching structure is the aim of bisimulation equivalency.

3.3.2 Linear Time and Branching Time Concurrency Semantics

Up to 12 semantics can be defined on uniform concurrency, as shown in Figure 3.1. Since trace semantics only considers the (partial) trace, it is regarded as the coarsest [49]. Miller in [65] found that the most sophisticated semantics were found in bisimulation. Semantics of bisimulation is the norm for the process algebra CCS. The benefits of bisimulation equivalency over observational equivalency were demonstrated in [45]. On the domain of concrete sequential processes that branch finitely, both equivalencies are consistent. The semantics established in [34] complement bisimulation semantics in this field as well.

There are ten semantics in between. Firstly, by using entire traces instead of partial ones, several kinds of trace semantics can be achieved. In [51], failures semantics is put out and used to build a process algebra model [49]. Compared to comprehensive trace semantics, it is finer. [35] established that the failure semantics on the domain of finitely branching, concrete, sequential processes coincides with testing equivalency as introduced in [67], as done in [55] and [31]. Readiness semantics, which is marginally finer than failures semantics, is introduced in [69]. As independently proposed in [75], ready trace semantics is identified between bisimulation semantics and readiness. (there called barbed semantics)[9] and [76] (under the name exhibited behaviour semantics).

3.3.3 Compact Kripke Structure Equivalent with an LTS

We will demonstrate in this thesis that it is feasible to actualize equivalence between Timed Calculus of Computation System(TCCS) and Time Computation tree logic(TCTL). Previous work by Nourollahi in [68] has established algorithm equivalence between Timed automata (TA) and Timed Kripke (TK) structure, for instance, as a conversion between (TA)and timed (TK) in the dense time domain. This has been answered positively in the untimed domain, where different constructive equivalence relations under CTL between labeled transition system (LTS) and Kripke structures in the untimed domain have been developed. The equivalences are inductive and algorithmic.

Method 1: Function $\mathcal{K}$ Converts an LTS into an Equivakent Kripke Structure

Definition 3.1. *Equivalence between Kripke Structurs and LTS: Given a Kripkee structure K and a set of states Q of K, the pair K, Q is equivalent to a process p, written $K, Q \simeq p$ (or $p^{'} K, Q$), if and only if for any CTL* formula $K, Q \models f$ if and only if $p \models f$.*

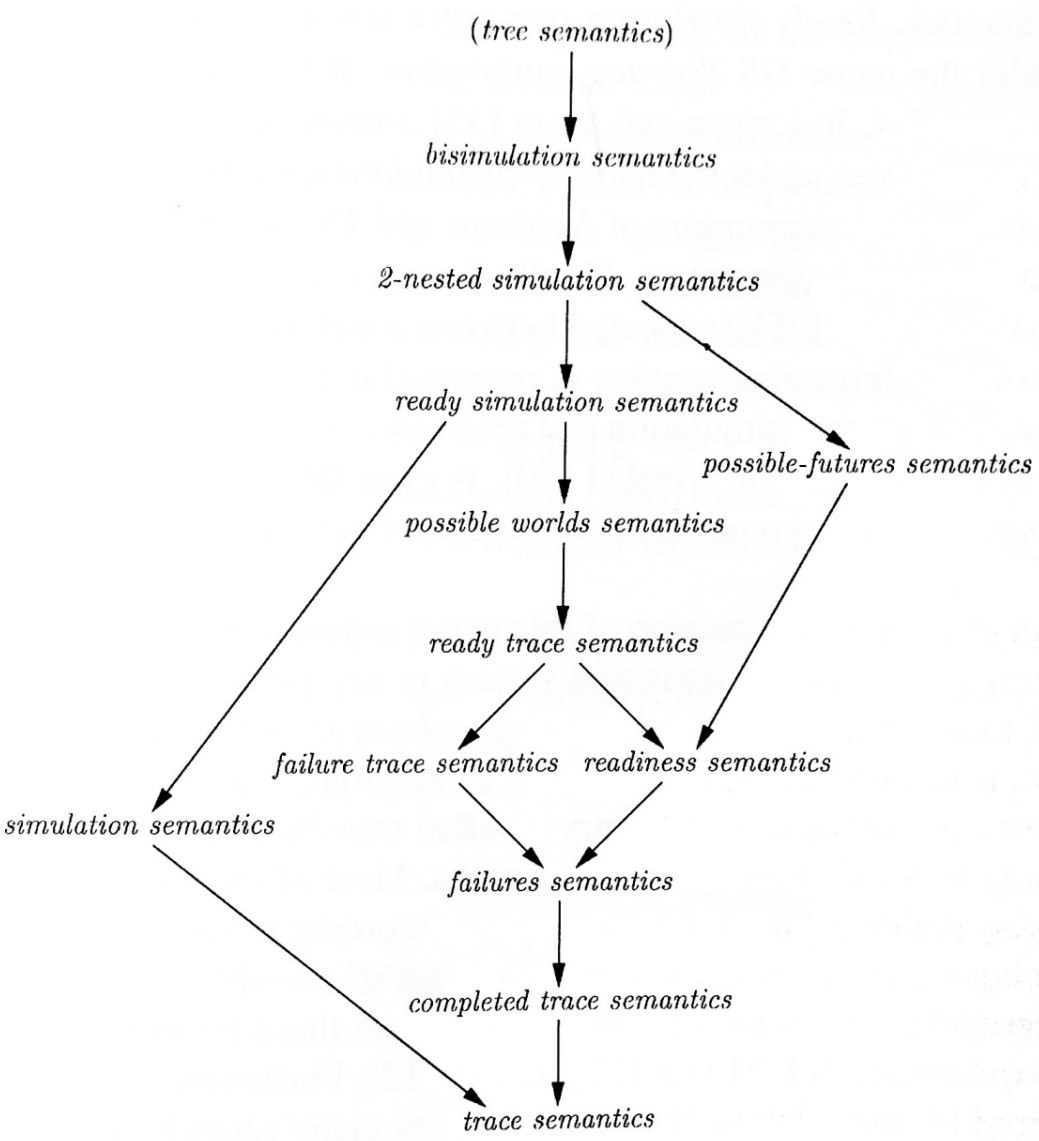

Figure 3.1: Linear time-branching spectrum [68].

Theorem 3.1. *There exists an algorithmic function $\mathcal{K}$ which converts a labeled transition system p into a Kripke structure K and a set of states Q such that $p \simeq (K, Q)$.*

Specifically, for any labeled transition system $p = (S, A, \rightarrow, S_0)$, its equivalent Kripke structure $k = K(p)$ is defined as $k = (S_0, Q, R', L')$ where

1. $S' = \{\langle s, x\rangle : s \in S, x \subseteq init(a)\}$
2. $Q = \{\langle s_0, x\rangle \in S'\}$

3. $\mathcal{R}$ contains exactly all the transitions $(\langle a, N\rangle, \langle t, O\rangle)$ such that $\langle a, N\rangle, \langle t, O\rangle \in S'$ and
   - for any $n \in N, s \xrightarrow{n} t$
   - for some $q \in S$ and for any $o \in O, t \xrightarrow{0} q$, and item if $N = \phi$ then $O = \phi$ and $t = s$ (these loop ensure that the relation R' is complete
   - $L' : S' \rightarrow 2^{AP}$ such that $L'(s, x) = x$ where $AP = A$

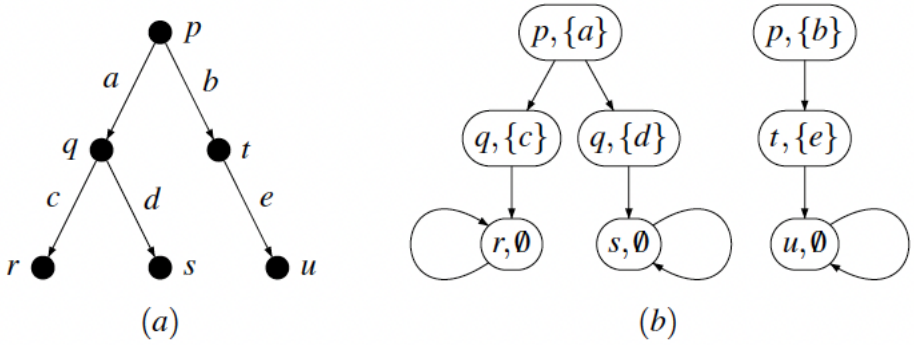

Figure 3.2: A conversion of an LTS (a) to an equivalent Kripke structure (b) [23].

By utilizing function K conversion method, the semantics of CTL^* formulae with respect to a process rather than Kripke structure can be definedd. The resulting Kripke structure is very compact, but a new satisfaction operator for sets of Kripke states is needed [23] (since one state of a process can generate multiple initial Kripke states).

Method 2: Function $\mathcal{X}$ Converts an LTS into an Equivakent Kripke Structure The need of a supplementary satisfaction operator can be eliminated using a different conversion function [36], at the expense of a considerably larger Kripke structure.

Theorem 3.2. . *There exists an algorithmic function $\mathcal{X}$ which converts a labeled transition system into an equivalent Kripke structure.*

The function $\mathcal{X}$ is defined as follows: with $\triangle$ a fresh symbol not in A, given an LTS $p = (S, A, \rightarrow, s_0)$ the Kripke structure $\mathcal{X}(p) = (S', Q, R', L)$ is given by:

1. $AP = \mathcal{A} \uplus \triangle$
2. $S' \cup \{(r, a, s) : a \in \mathcal{A} \ \ and \ \ r \xrightarrow{a} s\}$
3. $Q = \{s_0\}$

4. $\mathcal{R}' = \{(r,s) : r \xrightarrow{\tau} s\} \cup \{r,(r,a,s)) : r \xrightarrow{a} s\} \cup \{((r,a,s),s) : r \xrightarrow{a} s\}$
5. For $r,s \in S$ and $a \in \mathcal{A} : \mathcal{L}((r,a,s)) = \{a\}$

Then $p \simeq \mathcal{X}(p)$.

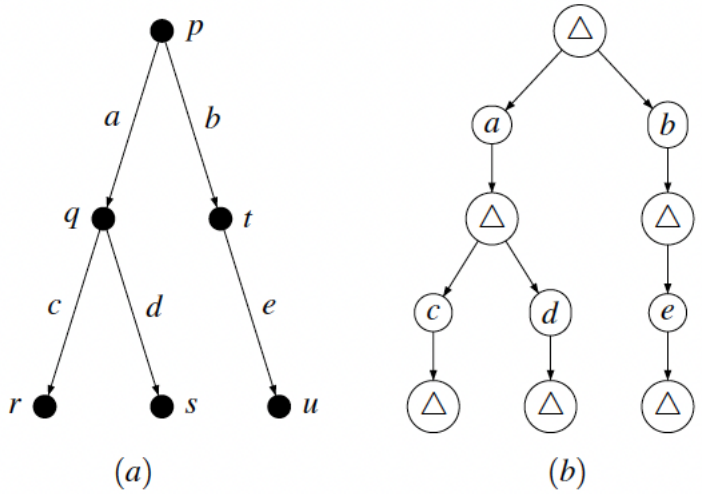

Figure 3.3: Conversion of an LTS (a) to its equivalent Kripke structure (b) [23].

In the resulting Kripkle structure, instead of combining each state with its corresponding actions in the LTS (and thus possibly splitting the LTS state into multiple Kripke structure states), the new symbol △ is used to stand for the original LTS states. Every △ state of the Kripke structure is the LTS state, and all the other states in the Kripke structure are the actions in the LTS. This ensures that all states in the Kripke structure corresponding to actions that are outgoing from a single LTS state have all the same parent. This inturn eliminates the need for the weaker satisfaction operator over sets of states. However, a relatively straightforward modification to the CTL satisfaction operator is needed (to "jump over" △ states).

3.4 Compositional Verification

Compositional verification breaks down the verification of complex systems into smaller, more manageable components. [59] proposed compositional model checking techniques for timed systems, enhancing scalability and reducing the complexity of verification.

Overall, a vibrant and developing subject is constructive equivalency in real-time systems employing process algebra and model checking. Rigorous verification methods are based on foundational theories like bisimulation and timed

automata. Real-time system practical verification is made possible by time computation tree logic and process algebra extensions, which have important applications in traffic control, train safety, and elevator systems. Modern developments in compositional methods, probabilistic models, and parameterized verification keep these approaches more capable and scalable while guaranteeing the security and dependability of ever-more complicated real-time systems.

Chapter 4

Equivalence Checking of Real-Time Systems

Because real-time systems are intricate and involved, they also have more stringent demands [22]. For instance, when verifying real-time systems with formal methods it is very important to guarantee that multiple models (algebraic view versus logical view) still describe the same behavior. Equivalence checking is one of a class of techniques to determine whether two models are, in some sense not extrapolated here, equivalent. In particular, we will investigate formal equivalences including those between Timed Calculus of Communicating Systems (TCCS), a process algebra for the modelling timed systems and Timed Computation Tree Logic (TCTL) which can specify properties of real-time aspects.

Constructive equivalence implies that descriptions of system behaviors in TCCS can be translated to checks in TCTL, and vice versa: the distinct ways of describing a system parse; [57, 8]. This is useful in defining the system behaviour with respect to the rule that has been mentioned, both using TCCS and asking questions about it with TCTL. Consider a job like we have some new robot and it knows what to do such as move forward, turn left or right, stop etc. TCCS gives us a way of describing them and how long they take. But TCTL would be suitable checking that certain things happen in the system at a particular time or higher level checks. You're literally asking questions regarding what the robot does: "does robot takes a turn in 5 seconds"? or "Does the robot always wait at 2 seconds before moving again". By constructive equivalence we mean that to every method of encoding system actions in TCCS gives rise a question in CTL asking if the encoded behaviour is satisfied and vice versa.

Example The road crossing is a well-known concurrent system, similar to the railroad crossing, where every part must coordinate to timely alert people and vehicles to stay off the railway track while a train is approaching. Usually this consist of a

train, crossing gate, vehicles (cars) and may even include pedestrians all synchronically working together to prevent any accidents via exercises in cooperation. The problem at railroad crossings is a common example of safety-critical real-time systems — where the train and gate (controlled by different parties) must react within real-time constraints to prevent an accident.

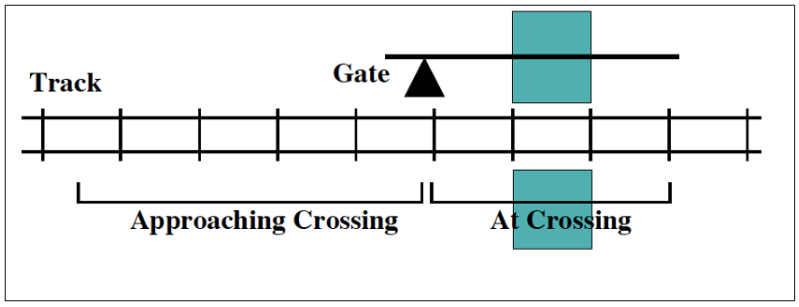

Figure 4.1: The Rail Road Crossing Problem [83].

TCCS, an extension of CCS (Calculus of Communicating Systems) with timing information. TCCS reports on process behaviors over time, which interpreters can then use to define delays, timeouts and timed transitions. However, there are a number of features specific to simulation. Transitions is associated with time delays while behavioural equivalence between processes which can be defined using bisimulation (methodology), which is extended to timing information in CCS. CCS model equation is given by $P \xrightarrow{\tau} Q$: Process P can transition to process Q with delay τ.

Similarly, TCTL is an extension of Computation Tree Logic (CTL) that includes timing constraints, allowing the expression of temporal properties with explicit timing requirements. The key features include; temporal operators with timing Constraints such as $AF_{\leq t}, EF_{\leq t, AG_{\leq t}}, etc$ and path qualifiers. TCTL formulas describe properties over paths, considering all possible future executions or specific ones.

We can represent the scenerio with a directed graph (KripKe structure) or using timed directed graph (Timed Automata).

4.1 Equivalence Verification Algorithm

In a very broad sense, equivalence checking in formal verification is the process of deciding if two models (an algebraic model and an logical one) behave identically. In this section we develop an algorithm for checking the equivalence between models in Timed CCS and sets of properties written in TCTL.

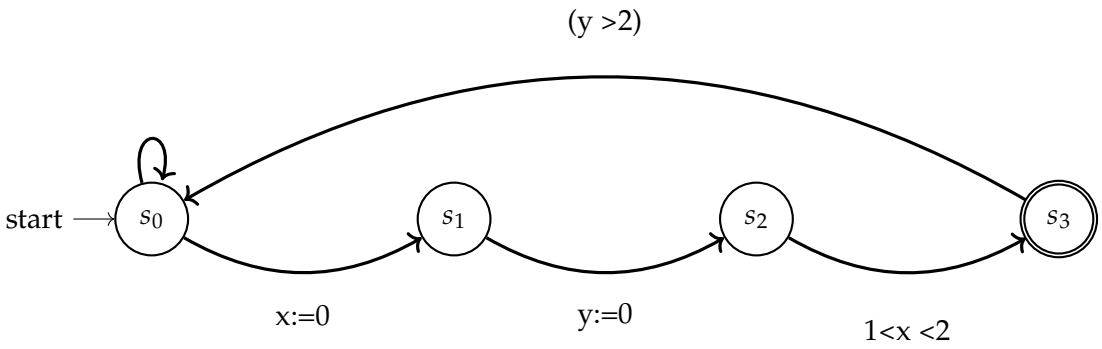

Figure 4.2: The Rail Road Crossing timed graph.

1. Define the system utilizing TCCS, outlining states, actions, and transitions along with their corresponding timing limitations.

   Let $\mathcal{M}_{TCCS} = (\mathcal{S}, Act, \rightarrow, I)$, where $\mathcal{S}$ is the set of states, Act is the set of actions, $\rightarrow$ is the transition relation, and I is the timing interpretation function.

2. Define the TCTL property ϕ_{TCTL} that need to be verified.

3. Construct the state space of the $TCCS$ model by exploring all possible states and the transitions.

   We thus generate the transition System $\mathcal{T} = (S, \rightarrow)$, where S is the set of states and $\rightarrow$ is the set of transitions (i.e. a given finite Automaton)

4. Translate ϕ_{TCTL} into an equivalent event clock automaton. This involves the construction of an event clock automaton that accepts all paths satisfying the $TCTL$ formula.

5. Synchronous Product, here we construct the $TCCS$ model $\mathcal{M}_{TCCS}$ and the automaton representing ϕ_{TCTL}. The product automaton will represent possible behaviours of the $TCCS$ model that might satisfy the $TCTL$ formula.

   Synchronous Product is an operation to merge a process model like Timed Calculus of Communicating systems (TCCS) with a specification model such as Timed Automaton which can be a representation of TCTL(Timed Computation Tree Logic) formula. It is from this method that we can use property specifications in checking the behaviour of a system at certain temporal properties.

6. Bisimulation Checking [63, 70, 10]: This involves conducting bisimulation equivalence between the $TCCS$ model and $TCTL$ automaton to ensure that

for every behaviour (path) in the $TCCS$ model, there is a corresponding path in the $TCTL$ automaton and vice versa

7. Model Checking [29, 10]: Perform model checking to verify whether the $TCCS$ model satisfies the $TCTL$ formula. We can use a model checker tool to explore all paths in the $TCCS$ model and check if the ϕ_{TCTL} holds.

   However, Performing model checking to verify whether a TCCS (Timed Calculus of Communicating Systems) model satisfies a TCTL (Timed Computation Tree Logic) formula involves a systematic process to explore all possible states and transitions of the TCCS model to ensure that the temporal logic properties expressed by the TCTL formula hold. Below are the detailed steps and considerations for this verification.If we formally define the TCCS model that represents the behavior of the real-time system. This includes defining the processes, actions, communication events, and timing constraints.

8. Equivalence Decision: If the bisimulation holds and the model checker confirms that the $TCCS$ model satisfies ϕ_{TCTL}, conclude that model are equivalent. Otherwise, if there is exception(a path in the TCCS model does not satisfy ϕ_{TCTL}, the models are not equivalent.

4.1.1 Translation of a TCTL Formula into an Equivalent Event Clock Automaton

Step 4 of the algorithm above needs now to be refined. Recall that the input of this step is a TCTL formula ϕ and the output will be an equivalent event clock automaton $\mathcal{A}_\phi$. We proceed by structural induction.

For the base case we handle atomic propositions as follows:

1. Atomic propositions: We create states and transitions in the ECA that correspond to the satisfaction or violation of each atomic proposition p in ϕ. Add a transition from state S to the new state S_p, labeled with p, if p holds at state S.

We then provide the following constructions for the Boolean operators:

2. Conjunction: Given the automata $\mathcal{A}_{\phi_1}$ and $\mathcal{A}_{\phi_2}$ for two formulae ϕ_1 and ϕ_2, we construct an automaton for $\phi_1 \wedge \phi_2$ that accepts a path if both $\mathcal{A}_{\phi_1}$ and $\mathcal{A}_{\phi_2}$ do.

   Let $\mathcal{S}_{\phi_1}$ and $\mathcal{S}_{\phi_1}$ be the set of all states representing different stages satisfying ϕ_1 and ϕ_2, respectively. The product Automaton $\mathcal{A}_{\phi_1 \wedge \phi_2}$ involves combining the states and transitions of $\mathcal{A}_{\phi_1}$ and $\mathcal{A}_{\phi_2}$. The set of states of product automaton is then the Cartesian product of the states of $\mathcal{A}_{\phi_1}$ and $\mathcal{A}_{\phi_2}$: $\mathcal{S}_{\phi_1 \wedge \phi_2} = \mathcal{S}_{\phi_1} \times \mathcal{S}_{\phi_2}$, with the initial state of $\mathcal{A}_{\phi_1 \wedge \phi_2}$ being $(S0_{\phi_1}, S0_{\phi_2})$, where

$S0_{\phi_1}$ and $S0_{\phi_2}$ are the initial state of $\mathcal{A}_{\phi_1}$ and $\mathcal{A}_{\phi_2}$ respectively. Similarly, the accepting states of $\mathcal{A}_{\phi_1 \wedge \phi_2}$ is the product of the accepting states of $\mathcal{A}_{\phi_1}$ and $\mathcal{A}_{\phi_2}$ that is, $\mathcal{F}_{\phi_1 \wedge \phi_2} = \mathcal{F}_{\phi_1} \times \mathcal{F}_{\phi_2}$. A state (s_1, s_2) is accepting if both s_1 is accepting state in $\mathcal{A}_{\phi_1}$ and s_2 is an accepting state in $\mathcal{A}_{\phi_2}$.

We now define transition for $\mathcal{A}_{\phi_1 \wedge \phi_2}$ based on transition of $\mathcal{A}_{\phi_1}$ and $\mathcal{A}_{\phi_2}$ as follows: A transition $((s_1, s_2), a, (s_1^{'}, s_2^{'}))$ exists in $\mathcal{A}_{\phi_1 \wedge \phi_2}$ whenever $(s_1, a, s_1^{'})$ in $\mathcal{A}_{\phi_1}$ and $(s_2, a, s_2^{'})$ in $\mathcal{A}_{\phi_2}$. This implies that both automata simultaneouly perform the same action a. We also ensure to synchronize clocks, meaning that clocks from both $\mathcal{A}_{\phi_1}$ and $\mathcal{A}_{\phi_2}$ are combined. Constraints must hold for both automata for a transition to be valid.

To construct the timing constraints for the automaton $\mathcal{A}_{\phi_1 \wedge \phi_2}$ from the individual constraints of $\mathcal{A}_{\phi_1}$ and $\mathcal{A}_{\phi_2}$, we need to combine the constraints and condition of both automata into a single automaton that satisfies both subformulas ϕ_1 and ϕ_2 simultaneously. If the clocks used for $\mathcal{A}_{\phi_1}$ are defined as $\{x_1, x_2, \ldots, x_n\}$ and those of automaton $\mathcal{A}_{\phi_2}$ are $\{y_1, y_2, \ldots, y_m\}$ then $\mathcal{A}_{\phi_1 \wedge \phi_2}$ will feature the clocks $\{x_1, x_2, \ldots, x_n, y_1, y_2, \ldots, y_m\}$, the disjoint union of the two sets of clocks. All clocks are reset according to the transitions of their respective automata. The disjoint union ensure that the reset of clocks on one automaton does not interfere with the clocks in the other automaton.

To construct the timing constraints for the automaton $\mathcal{A}_{\phi_1 \wedge \phi_2}$ (conjunction of two subformulas ϕ_1 and ϕ_2) out of the timing constraints of $\mathcal{A}_{\phi_1}$ and $\mathcal{A}_{\phi_2}$ we simply take the conjunction of the existing constraints. That is, with the clock constraints for $\mathcal{A}_{\phi_1}$ and $\mathcal{A}_{\phi_2}$ as K_1 and K_2, respectively, the timing constraints for the new automaton $\mathcal{A}_{\phi_1 \wedge \phi_2}$ will be $K_{\phi_1 \wedge \phi_2} = K_1 \wedge K_2$. This means that a transition is allowed in $\mathcal{A}_{\phi_1 \wedge \phi_2}$ only if both $\mathcal{A}_{\phi_1}$ and $\mathcal{A}_{\phi_2}$ allow the transition (i.e., their individual timing constraints are satisfied). That is, if $(S_1 \xrightarrow{a, k_1} S_1^{'})$ is a transition in $\mathcal{A}_{\phi_1}$ and $(S_2 \xrightarrow{a, k_2} S_2^{'})$ is a transition in $\mathcal{A}_{\phi_1}$, then the combined transition in $\mathcal{A}_{\phi_1 \wedge \phi_2}$ is: $S_1, S_2) \xrightarrow{a, k_1 \wedge k_2} (S_1^{'}, S_2^{'})$.

3. Disjunction: Following the same conventions we construct the combined automaton $\mathcal{A}_{\phi_1 \vee \phi_2}$ that accepts a path if either $\mathcal{A}_{\phi_1}$ or $\mathcal{A}_{\phi_2}$ does. For this purpose we create a disjoint union of the two automata that is, $\mathcal{S}_{\phi_1 \vee \phi_2} = \mathcal{S}_{\phi_1} \cup \mathcal{S}_{\phi_2}$. We then introduce a new initial state $S0$ with transition to both $S0_{\phi_1}$ and $S0_{\phi_2}$: $S_0 \xrightarrow{\epsilon} S_{0\phi_1}$ and $S_0 \xrightarrow{\epsilon} S_{0\phi_2}$. These epsilon transition indicates that the system can non-deterministically start in either $\mathcal{A}_{\phi_1}$ or $\mathcal{A}_{\phi_2}$. It follows that the set accepting states for $\mathcal{A}_{\phi_1 \vee \phi_2}$ is the union of the accepting states of $\mathcal{A}_{\phi_1}$ and $\mathcal{A}_{\phi_2}$. $\mathcal{F}_{\phi_1 \vee \phi_2} = \mathcal{F}_{\phi_1} \cup \mathcal{F}_{\phi_2}$.

4. Negation follows immediately from Theorem 2.1. The process involves the conversion of the given automaton into a deterministic version and then the

flip of all the accepting states into non-accepting states and the other way around.

For the temporal operators we find convenient to consider the operators themselves first and then consider path quantifiers separately.

5. For $F_I\phi_1$ we construct an automaton that tracks the time until ϕ_1 is satisfied within interval I by creating a clock x and reset it upon entering a state where ϕ_1 holds. The clock resetting to 0 upon each entry into a state $s' \in S_{\phi_1}$ is event-driven (satisfying EVA criteria). This can be done by introducing a clock x that will measure the time elapsed since the automaton entered a specific state where ϕ_1 holds. This means that whenever the automaton enters a state in $\mathcal{S}_{\phi_1}$, x is set to 0. Formally, for any transition (s, a, s') where $s' \in \mathcal{S}_{\phi_1}$: $s \xrightarrow{a,x:=0} s'$. Then we add transitions that move to an accepting state if x satisfies the interval I.This can be done by introducing transitions that move to an accepting state based on the value of clock x. Let $\mathcal{I}$ be the time interval that defines the time constraint for accepting a path. These transitions should check whether x falls within the interval $\mathcal{I}$ and move to an accepting state if the condition is satisfied. Formally, for each state $s' \in \mathcal{S}_{\phi_1}$ we add: $s' \xrightarrow{a,x\in\mathcal{I}} \mathcal{S}_{accept}$.

6. For $G_I\phi_1$ we construct an automaton that ensures ϕ_1 holds continuously within interval I that is, we maintain the same clock x and add transitions that remain in the current state as long as ϕ_1 holds and x is within I.

   This can be done by for each state $s \in \mathcal{S}_{\phi_1}$, add a self-loop transition that allows the system to remain in the same state as long as ϕ_1 holds and x is within the interval $\mathcal{I}$. Formally, for each state $s \in \mathcal{S}$ we put $s \xrightarrow{\varepsilon,x\in\mathcal{I}} s$. The clock condition $x \in \mathcal{I}$ ensures that this self-loop is valid only if x is within the specified time interval $\mathcal{I}$. However, the clock constraints are applied on the transitions, not on maintaining a condition within states. This aligns with the requirement that event clock automata should not impose time constraints on states directly.

7. To convert $\phi_1 \bigcup_I \phi_2$ we construct an automaton that stays in a where ϕ_1 holds until ϕ_2 is satisfied, ensuring that ϕ_1 holds up until that point, and ϕ_2 is satisfied within interval I. This can be achieved by ensuring a temporal property where ϕ_1 must hold continuously until ϕ_2 is satisfied within a timing constraint.

   Concretely, let $\mathcal{A}_{\phi_1}$ and $\mathcal{A}_{\phi_2}$ be the automata equivalent to ϕ_1 and ϕ_2, respectively. A new clock x is reset on every transition outgoing from the start state s_{start} of $\mathcal{A}_{\phi_1}$, and we add the time constraint $x \in I$ for all the other transitions of $\mathcal{A}_{\phi_1}$. This ensures that we can only stay in $\mathcal{A}_{\phi_1}$ within the time interval

I. We then add a transition $s_f \xrightarrow{\varepsilon, x \in I} s_{start}$ for each accepting state s_f of $\mathcal{A}_{\phi_1}$, which allows us to stay within this automaton for as long as necessary. Finally, we also add the transitions $s_f \xrightarrow{\varepsilon, x \in I} s'_{start}$, where s'_{start} is the start state of $\mathcal{A}_{\phi_2}$. This allows the resulting automaton to transition from ϕ_1 to ϕ_2 at any time within I, without the possibility of coming back. If $\mathcal{A}_{\phi_2}$ completes its run successfully then ϕ_2 accepts and thus releases ϕ_1 from its obligation. If on the other hand the run is rejecting, then the input will be rejected unless there exists another, successful run. This observes all the properties of the $\bigcup$ operator.

All the timing conditions remain unchanged, except for the clock x which is added and observes the restrictions of event clock automata requirements.

Finally we introduce the path quantifiers A and E. We will construct automata corresponding to the formulae $A\phi_I$ and $E\phi_I$ assuming by inductive hypothesis the existence of the automaton $\mathcal{A}_{\phi_I}$ equivalent to ϕ_I. Note that ϕ_I can have one of the forms $X_I\phi$, $F_I\phi$, $G_I\phi$, and $AU_I\phi$.

8. The automaton $\mathcal{A}\phi_I$ is already equivalent to $E\phi_I$. Indeed, the automaton accepts iff there exists a successful run. A single such a run suffices, thus meeting the requirements of the existential quantifier.

9. On the other hand the automaton equivalent to $E\phi_I$ must accept only if all the runs are accepting. This can be handled by constructing a variant of the deterministic version of $\mathcal{A}_{\phi_I}$. Let S and F be the set of states and the set of accepting states of $\mathcal{A}_{\phi_I}$, respectively. We proceed with the usual construction in establishing the states and transition of the deterministic automaton [7] (also see Theorem 2.1), thus obtaining $\mathcal{A}^d_{\phi_I}$ with the set of stated $S^d \subseteq 2^S$ and the set of accepting states F^d. Recall in particular that $F^d = \{f \in S^d : F \cap f \neq \emptyset\}$: A run of $\mathcal{A}^d_{\phi_I}$ ends in a state s whenever all the runs of $\mathcal{A}_{\phi_I}$ end in one of the states in s; if one of these states is accepting then the respective run is accepting and so the input is accepted and the other, possibly rejecting runs become irrelevant. When we introduce the universal quantifier we want *all* the runs to be accepting in $\mathcal{A}_{\phi_I}$ for $\mathcal{A}^d_{\phi_I}$ to accept the input. This is easily accomplished by setting $F^d = 2^F$. In other words, a state in $\mathcal{A}^d_{\phi_I}$ is accepting if *all* the component states (from $\mathcal{A}_{\phi_I}$) are accepting that is, if *all* the runs $\mathcal{A}_{\phi_I}$ are accepting.

   No new clocks or time constraints are added, so $\mathcal{A}^d_{\phi_I}$ is an event clock automaton under the inductive hypothesis that $\mathcal{A}_{\phi_I}$ is an event clock automaton.

Theorem 4.1. *There exists an algorithm that determines whether a given TCTL formula and a given TCCS process are equivalent. The algorithm presented in Section 4.1 accepts the input TCTL formula and TCCS process if and only if they are equivalent.*

Proof. The following are the constructions used in the algorithm.

(a) Timed Automaton Construction: The first step in the algorithm involves constructing a timed automaton from the TCTL formula. This construction is formalized and proven in Section 4.1.1. Specifically, the automaton encodes all the timing constraints and logical structures present in the TCTL formula, such that the automaton accepts exactly the same set of timed traces (timed words) that satisfy the formula. Thus, this step guarantees the correctness of the automaton representing the TCTL formula.

(b) Cross-Product Construction: Once both the timed automaton (representing the TCTL formula) and the Timed Transition System (TTS) (representing the TCCS process) are constructed, the algorithm proceeds to take the cross-product of the two systems.

The cross-product construction combines the state spaces and transitions of the TTS and the timed automaton into a new automaton. The result of this construction represents the joint behavior of the two systems. By doing this, the algorithm creates a unified framework to compare the behavior of both the TCTL formula (through the timed automaton) and the TCCS process (through the TTS).

This step ensures that we can analyze whether the behaviors (timed traces) of the TTS and the timed automaton align. In this context, the cross-product is not yet sufficient to establish equivalence, but it prepares the two systems for further analysis.

(c) Bisimulation Check: Next, the algorithm performs a bisimulation check between the TTS and the timed automaton from the cross-product. Bisimulation is a formal equivalence relation that checks whether two systems simulate each other step by step. In the context of timed systems, the bisimulation takes into account both state transitions and timing constraints.

If the bisimulation check succeeds, it means that the TTS and the timed automaton (representing the TCTL formula) exhibit equivalent behavior in all relevant states and time evolutions. This establishes that the TCCS process and the TCTL formula describe the same timed behavior. If the bisimulation check fails, the two systems are not behaviorally equivalent, and hence the TCCS process does not satisfy the TCTL formula. Thus, the bisimulation check is a necessary step for confirming whether the process and formula are observationally indistinguishable in terms of timed behaviors.

(d) Model Checking: The next step is performing model checking on the TTS against the TCTL formula. Model checking systematically explores the state space of the TTS to verify whether all possible executions (timed traces) satisfy the TCTL formula.

If the model checking succeeds, the TTS satisfies the TCTL formula, meaning that the process described by the TCCS model adheres to the logical and timing requirements specified by the formula. If the model checking fails, there exists

a counterexample trace where the TTS violates the TCTL formula, thus proving non-equivalence between the process and the formula. The model checking step provides a concrete verification of whether the specific TCCS process adheres to the formal specification described by the TCTL formula.

Note that both bisimulation and model checking are required because they establish equivalence from different perspectives: Bisimulation guarantees behavioral equivalence between the TCCS process and the timed automaton representing the TCTL formula. It checks whether the process and formula behave equivalently under all possible scenarios. Model checking verifies whether the specific instance of the TCCS process satisfies the logical properties encoded in the TCTL formula. It provides a direct method for ensuring that all behaviors of the process are valid with respect to the specification.

In other words bisimulation ensures that the two systems are behaviorally equivalent, while model checking guarantees that all behaviors of the process are allowed by the formula. Thus, both steps together establish the full equivalence between the TCCS process and the TCTL formula, ensuring that the algorithm correctly accepts the input if and only if they are equivalent. □

4.2 Examples

4.2.1 TCTL Formulae and Equivalent Automata

Consider the following TCTL formula:

$$\phi = A[G(p \rightarrow F_{[0,5]q})]$$

The equivalent automaton is define d as follows:

- States: $\mathcal{S} = \{S_0, S_1, S_2\}$
- Clock: $C = x$.
- Transition relation: $S_0 \xrightarrow{p, x:=0} S_1$, $S_1 \xrightarrow{\neg q, x<5} S_1$, $S_1 \xrightarrow{q, 0 \leq x \leq 5} S_2$
- Accepting states: $\mathcal{F} = \{S_2\}$.

For path $\pi = (S_0, S_1, S_2)$, the automaton will accept the path if it reaches the state S_2, because $S_2 \in \mathcal{F}$. The automaton is shown in Figure 4.3.

Consider now the following TCTL formula

$$\phi = E[pU_{[2,5]}]q$$

which reads as follows: There exists at least one path where p holds continuously until q becomes true, and q occurs after at least 2 times but no more than 5 time units from the beginning. The equivalent automaton is shown in Figure 4.4, where:

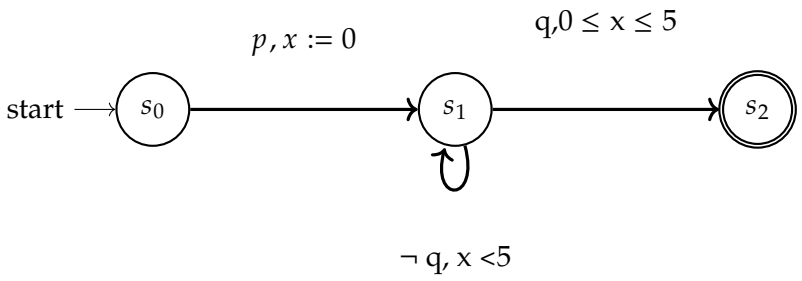

Figure 4.3: Automata illustrating the temporal operators A, G, F

- States: $\mathcal{S} = \{S_i, S_0, S_1\}$,
- Clock: $\mathcal{C} = x$.
- Transition relation: $S_i \xrightarrow{p,x:=0} S_0$, $S_0 \xrightarrow{p,x\leq 5} S_0$, $S_0 \xrightarrow{q,2\leq x\leq 5} S_1$,
- Accepting states: $\mathcal{F} = \{s_1\}$.

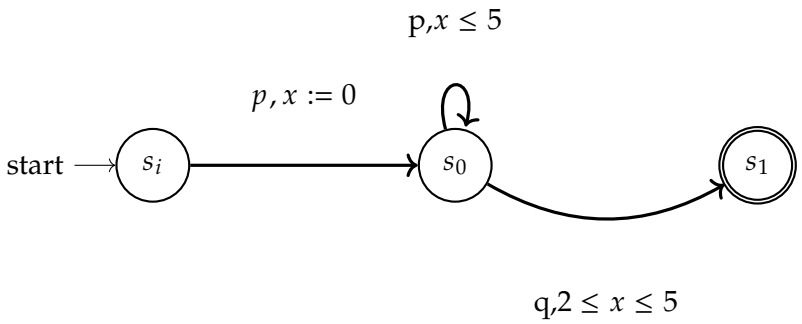

Figure 4.4: Automaton illustrating the temporal operators E and U.

Finally we consider a more complex formula as follows:

$$\phi = A[G(p \rightarrow E[qU_{[1,4]}F_{[2,6]}r])]$$

This formula specifies that for all paths, globally, if p holds, then there exist at least one path where q holds continuously until r becomes true, and r occurs within the

time interval [1,4] time units. This implies that r eventually holds between 2 and 6 time units.

The equivalent automaton is shown in Figure 4.5, where:

- States: $\mathcal{S} = \{S_0, S_1, S_2, S_3\}$.
- Clock: $\mathcal{C} = x$.
- Transition relation: $S_0 \xrightarrow{p,x:=0} S_1$, $S_1 \xrightarrow{q,1\leq x\leq 4]} S_2$, $S_2 \xrightarrow{r,2\leq x\leq 6} S_3$.
- Accepting states: $\mathcal{F} = \{S_1, S_3\}$. Note that S_1 corresponds to the continuous satisfaction of q, and S_3 corresponds to r being eventually satisfied within the interval [2,6].

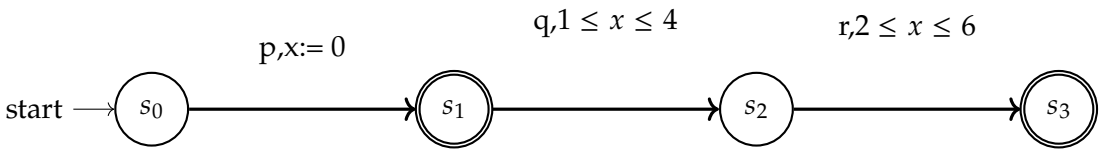

Figure 4.5: An Automaton illustrating the temporal operators A, E, F, G, U.

4.2.2 Synchronous Product Example

Let us assume a simple TCCS model of a train crossing system where a train can approach and cross a gate.

1. Definition of the TCCS Model say M_{TCCS}
   processes:
   $Train = approach.T$
   $T = cross.G$
   $Gate = close.g.open.Gate$
   [gate closes, wait for the signal to open , and the opens]
   System: Representation of the entire system
   $M_{TCCS} = (Train|Gate)\setminus\{close, open\}$
   where | donates a parallel composition and $\setminus\{close, open\}$ donate the hiding of actions

2. Definition of TCTL formula say ϕ_{TCTL}
   $\phi_{TCTL} = A[(cross \rightarrow F_{[0,3]}open)]$

3. Construction of time Automaton
   $S_0 \xrightarrow{cross,x:=0} S_2$

$S_1 \xrightarrow{open, x \leq 3} S_2$
$S_1 \xrightarrow{x>3}$ Violation : if 3 times units pass without the gate opening, trainsition to a violation state

4. Synchronize M_{TCCS} and the Timed Automaton
   The synchronous product combines the states and transitions of M_{TCCS} and the automaton, ensuring that the TCCS system behaves according to the timing constraints specified in ϕ_{TCTL}.
   - Initial state:$(Train, Gate, S_0)$
   - Transition 1:$(Train, Gate, S_0) \xrightarrow{approach} (T, close.g.open.Gate, S_0)$
   - Transition 2:$(T, close.g.open.Gate, S_0) \xrightarrow{cross, x:=0} (G, Gate, S_1)$
   - Transition 3:$(G, Gate, S_1) \xrightarrow{open, x \leq 3} (Train, Gate, S_2)$

Our illustration demonstrates the process of creating a TCCS model along with its associated automaton to represent a TCTL formula. The synchronous product guarantees that the system model M_{TCCS} fulfills the temporal requirements outlined by ϕ_{TCTL}.

4.2.3 Automated Model Checking Implementation

Consider a TCCS model of a simple train crossing system:

$$\begin{array}{rcl} \text{Train} & = & \text{(approach -> cross -> leave -> Train)} \\ \text{Signal} & = & \text{(signalOn -> wait -> signalOff -> Signal)} \end{array}$$

We define a temporal formula that satisfies the TCCS model (M_{TCCS}) as :

$$\phi_{TCTL} = A[G(signalOn \rightarrow F_{[0,3]}cross)]$$

If TCCS model is converted to a transition system that captures the states and transitions. This is very necessary before model checking because it allows the system's possible behaviours to be translated in terms of states and actions, making the system suitable for model checker. Each translation between states is associated with an action such as approcah, signalOn, cross, etc.

Once the transition system is defined, it is provided to a model checker tool as input. Popular model checkers for real-time systems include: UPPAAL(Supports timed automata and TCTL-based model checking), NuSMV(a symbolic model checker for finite state systems that can be used for real time verification) and PRISM(a tool for model checking probabilistic real-time systems.

After inputting the TCCS model and the TCTL formula into the model checker, the tool will explore all possible states and transitions of the model to check if the

formula holds across all paths. It generates all possible states reachable from initial state of the TCCS model and check if the timing constraints and actions specified in the TCTL formula are satisfied in every state and transition. For the given TCTL formula:

$$A[G(signalOn \rightarrow F_{[0,3]}cross)]$$

The model checker verifies whether for all path, whether the signalOn event occurs, the cross event happens within 3 times units.

4.2.4 Equivalence Verification with a Clock Automaton

Let the overall $TCCS$ model $\mathcal{T}_{System}$ be a parallel composition of $\mathcal{T}_{train}$ and $\mathcal{T}_{gate}$:

$$\mathcal{T}_{System} = \mathcal{T}_{gate} || \mathcal{T}_{train}$$

And the Safety property in $TCTL$ is the primary safety property we want to verify given as :

$$\phi_{safety} = AG(train \neq crossing \rightarrow gate = open)$$

Similarly, if we specify that gate closes within 2 times units of the train's arrival and open's only after the train has crossed.

$$\phi_{timing} = AG(arrive \rightarrow AF_{\leq 2}close) \quad \wedge \quad AG(cross \rightarrow AF_{\leq 1}open)$$

Then using bisimulation equivalence verification we have:

stateSpace $\mathcal{S} = \{idle, waiting, crossing, done\}$

transitions = $\{arrive, close, cross, open\}$

TCTLTimedAutomaton $\phi = \{\phi_{Safety}, \phi_{Timing}\}$

synchronousProduct $\mathcal{P} = pair(\mathcal{T}_{System}, \phi)$

The timed automaton will accept all behaviour where:

- The gate is closed before the train crosses,
- The gate closes within 2 times unites of the trains arrival, and
- The gate opens only when the train has crossed.

We recall from Definition 2.1 that two states s_1 and s_2 are bisimilar if whenever s_1 can transition to $s_1^{'}$ in one system, s_2 can transition to $s_2^{'}$ in the other system, and $s_1^{'}$ and $s_2^{'}$ are bisimilar. The checking process is as follows:

1. For each state $pair(\mathcal{S}_{TCCS}, \mathcal{S}_{Automation})$
2. Ensure that timing constraints are preserved in both models

3. If all state pairs satisfy bisimulation, the models are equivalent

Finally, we use model checker to confirm that the $TCCS$ model $\mathcal{T}_{system}$ satisfies the $TCTL$ properties ϕ_{safety} and ϕ_{timing}.

In the formal example, we verify an equivalence between a TCCS model and a TCTL specification by using bisimulation to check that real-time system behaviors satisfy safe timing properties. For example, each step from the TCCS description should exactly match with an existing TCTL checks so that both algebraic and logical descriptions of Railroad crossing problem fits together. This makes TCCS process verifiable wrt the behavior and demonstrates its constructive equivalent.

4.2.5 Yet Another Constructive Equivalence Checking

Let a constructive equivalence can be formally represented as $\mathcal{A} \equiv \mathcal{B}$, with $\mathcal{A} = (\Sigma, Q, q_0, \Omega, F)$ and $\mathcal{B} = (\Sigma, Q^{'}, q_0^{'}, \Omega^{'}, F^{'})$ where:

- Σ is sets of inputs,
- Q and $Q^{'}$ are sets of states,
- q_0 and $q_0^{'}$ are sets of initial states,
- F and $F^{'}$ are sets of accepting states.

We need to show the following:

$$\forall q \in Q, \exists q^{'} \in Q^{'} : \text{property}(q) = \text{property}(q^{'})$$

In our case,

$$\forall q \in \{S_0, S_1, S_2, S_3\}, \exists q^{'} \in \{S_0^{'}, S_1^{'}, S_2^{'}, S_3^{'}\} : \neg(S_1 \wedge S_2) \equiv \neg(S_1^{'} \wedge S_2^{'})$$

Therefore, by guaranteeing that the safety property is upheld, the equivalency is demonstrated across both models. This same procedure can also be applied to other examples of real time systems such as the elevator system, the traffic light system, etc.

4.3 Addressing Non-Constructive Equivalence

Equivalence checking is a hard and complex problem, especially for large or intricate real-time systems. State space explosion is a significant challenge if found to be non-equivalent. This often results in additional simplification or re-design of the system under question with respect to its intended specifications. Non-constructive equivalence in real system verification between an algebraic model, such as Timed Calculus of Communicating Systems (TCCS), and a logical model, e.g. Time Computation Tree Logic(TCTL), means that the behaviors or properties expressible in

one formalism cannot be entirely captured by another[91, 6, 5]. That difference causes the verification to be incomplete and decreases reliability of system analysis.

For example, Algebraic model (TCCS): Train Behavior= $\{approach.T_1, cross.T_2, leave, .T\}$ and the Gate Behavior $=\{open.G_1, close.G_2, open.G\}$ and the system composition can be expressed as follows:

$$System = (T||G)\{approach, cross, leave, open, close\}$$

then the logical Model (TCTL) should be able to close within 3 times units after train leaves this can be given as :

$$\phi_1 = A(approach \rightarrow \exists \leq 3 \ \ close)$$

Also in another situation, the gate must open within 2 time units after the train leaves:

$$\phi_2 = A(leave \rightarrow \exists \leq 2 \ \ open)$$

TCCS has the capability to represent intricate nested timing constraints that TCTL struggles to capture fully, especially in cases like "the train crosses only if the gate closed within 3 time units of approach." As a result, the consequence of this lack of comprehensive equivalence leads to:

- Incomplete Verification: TCTL cannot verify some critical nested timing constraints, potentially overlooking important system behaviors.
- Reduced Reliability: The inability to check all timing constraints reduces the assurance that the system meets all safety and performance requirements.
- Increased Complexity: Bridging the gap between models may require extensions or more expressive logics, complicating the verification process.

Chapter 5

Conclusion

In this thesis we examined the verification of constructive equivalence across real-time systems using a hybrid approach that combines process algebra (TCCS) and temporal logic (TCTL) techniques such as bisimulation and model checking. The need for a scalable, automated verification framework to verify the stability of safety-critical systems with strict real-time limitations motivated this research.

We began by looking at the fundamental ideas of real-time systems, summarizing existing verification tools, and emphasizing the ongoing advances in formal verification methods (see Chapter 1). This contextual awareness provided the foundation for our strategy. Chapters 2 and 3 provided the theoretical foundation for our research, including the syntax and semantics of TCCS and TCTL. We conducted a thorough literature analysis, discussing prior efforts on formal verification and finding gaps that this thesis seeks to fill.

In Chapter 4, we used algebraic and logical approaches to create formal models that represented and tested real-time system behavior. The main contribution of our study is the fact that TCCS and TCTL are equivalent. We thus extended to some degree an earlier result establishing the equivalence between failure trace testing and CTL [23] to the timed domain. We effectively guarantee consistency between algebraically modeled and logically verified system behaviors. We made it possible to address the compositional complexity of big systems in a more structured manner by using process algebra for modular verification.

Our findings advance the subject of formal verification by showcasing a scalable method for combining logical and algebraic methodologies. Constructive equivalence can improve the resilience of system verification, according to our results, especially in fields like medical devices, aircraft, and transportation where timing and safety are crucial.

As opposed to the results available for the untimed domain [23], we fall short of actually providing algorithms for converting logical specifications into algebraic specifications and the other way around. This limitation restricts the immediate practical applicability of our framework and highlights an area requiring further

investigation. Additionally, we only focused on deterministic real-time systems, and stochastic behaviors were not addressed.

To build on the findings of this thesis, immediate future work should focus on developing algorithms for bidirectional conversion between logical and algebraic specifications. Other promising avenues for future research include extending the verification framework to handle more complex real-time systems with stochastic behaviors, incorporating machine learning techniques to optimize model-checking processes and enhance automation, and conducting extensive industrial case studies to validate the scalability and effectiveness of the proposed framework in practical, large-scale systems.

Limitations and future directions notwithstanding, this work lays the foundation for more robust, scalable, and widely applicable approaches to real-time system verification.

Bibliography

[1] M. Aagaard, R. B. Jones, and C.-J. H. Seger, *Formal verifivcation of loop pipeline designs.*, Design Automation Conference, 1998.

[2] Y. Abdeddaïm, E. Asarin, and O. Maler, *Scheduling with timed automata*, Theor. Comput. Sci., 2006.

[3] Y. Abdeddaïm and O. Maler, *ob-shop scheduling using timed automata*, In: Berry, G., Comon, H., Finkel, A. (eds.) Intl. Conf. on Computer-Aided Verification (CAV). LNCS, vol. 2102,pp. 478–492. Springer, Heidelberg, 2001.

[4] K. Altisen, G. Gößler, A. Pnueli, J. Sifakis, S. Tripakis, and S. Yovine, *A framework for scheduler synthesis*, In: Symposium on Real-Time Systems (RTSS), pp. 154–163. IEEE,Piscataway, 1999.

[5] R. Alur, C. Courcoubetis, and D. Dill, *Model-checking for real-time systems*, In Proceedings of the Fifth Annual Symposium on Logic in Computer Science (LICS), 1996.

[6] R. Alur and D. Dill, *A theory of timed automata.*, Theor. Comput. Sci. 126(2), 183–235, 1994.

[7] R. Alur, L. Fix, and T. Henzinger, *Event Clock Automata, a determinizable class of time Automata*, Theoritical Computer Science, 211:253-273, 1999.

[8] R. Alur and T. A. Henzinger, *Logics and Models of real time:*, A survey in Real-time:Theory in Practice, 1992.

[9] J. C. Baeten, J. A. Bergstra, and J. W. Klop, *Ready-trace semantics for concrete process algebra with the priority operator*, The Computer Journal, 30, 498-506, 1987.

[10] C. Baier and J. P. Katoen, *Principles of model checking*, 2008.

[11] G. Behrmann, A. Fehnker, T. Hune, K. Larsen, P. Pettersson, and J. Romijn, *Efficient guiding towards cost-optimality in Uppaal*, In: Margaria, T., Yi, W. (eds.) Intl. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS). LNCS, vol. 2031, pp. 174–188. Springer, Heidelberg, 2001.

[12] J. Bergstra, S. Smolka, and A. Ponse, *Handbook of Process Algebra*, 2001.

[13] J. A. Bergstra and J. W. Klop, *Algebra for Communicating Processes with Abstraction.*, Journal of Theoretical Computer Science, 1985.

[14] Y. Bertot and P. Casteran, *Interactive Theorem Proving and Program Development:CoqArt: The Calculus of Interactive Constructions*, Springer, 2004.

[15] A. Boucher and G. R., *A Timed Model for Extended Communicating Sequential Processes*, In Proceedings of ICALP'87, Lecture Notes in Computer Science, Vol. 267, Springer., 1987.

[16] G. Boudol, *Notes on Algebraic Calculi of Processes, Logics and Models of Concurrent Systems.*, NATO ASI Series f13 (K. Apt, ed)., 1985.

[17] P. Bouyer, N. Markey, U. Fahrenberg, J. Larsen, K. G.and Ouaknine, and J. Worrel, *µodel Checking real time systems*, Springer International, 2018.

[18] M. Bozga, H. Jianmin, O. Maler, and S. Yovine, *Verification of asynchronous circuits using timed automata*, In: Asarin, E., Maler, O., Yovine, S. (eds.) Workshop on Theory and Practice of Timed Systems (TPTS). Electronic Notes in Theoretical Computer Science, vol. 65, pp. 47–59. Elsevier, Amsterdam, 2002.

[19] A. Brekling, M. Hansen, and J. Madsen, *Models and formal verification of multiprocessorsystem-on-chips*, J. Log. Algebraic Program. 77(1–2), 1–19, 2008.

[20] S. D. Brookes, C. A. Hoare, and A. W. Roscoe, *A Theory of Communicating Sequential Processes*, Journal of ACM, 31, 1984.

[21] M. Broy, B. Johnson, J. P. Katoen, M. Leucker, and A. Pretschner, *Model-Based Testing of Reactive Systems:Advanced Lectures*, 3472 of Lecture Notes in Computer Science, 2005.

[22] S. D. Bruda and C. Dai, *A Testing Theory for real-time Systems.*, 2009.

[23] S. D. Bruda, S. Singh, A. N. Uddin, Z. Zhang, and R. Zuo, *A Constructive Equivalence between Computation Tree Logic and Failure Trace Testing*, 2019.

[24] F. Cassez, J. Jensen, K. Larsen, J.-F. Raskin, and P. A. Reynier, *Automatic synthesis of robust and optimal controllers—an industrial case study*, In: Majumdar, R., Tabuada, P. (eds.) Int. Workshop on Hybrid Systems: Computation and Control (HSCC). LNCS, vol. 5469, pp. 90–104. Springer, Heidelberg, 2009.

[25] E. M. Clarke, *Model checking and the state explosion problem*, Software engineering, 1993.

[26] E. M. Clarke and E. A. Emerson, *Design and synthesis of synchronization skeletons using branching time temporal logic*, in Works in Logic of Programs, 52 -71, 1982.

[27] E. M. Clarke, T. A. Henziger, and V. H., *Introduction to Model Checking*, Springer International Publishing AG, part of Springer Nature 2018.

[28] E. Cleake, O. Grumberg, and D. Long, *Model Checking and abstraction*, ACM transactions on Programming Languages and Systems. 1512-1542, 1994.

[29] E. M. Clerke, O. Grumberg, and D. A. Pelled, *Model Checking*, Model Checking, 1999.

[30] C. Dai, *Testing Framework for Real-Time Specifications*, TMasters thesis, Bishop's University, Canada., 2008.

[31] P. Darondeau, *An enlarged definition and complete axiomatization of observational congruence of finite processes, in International Symposium on Programming*, Springer, pp. 47–62., 1982.

[32] A. David, K. Larsen, A. Legay, M. Miku ˇcionis, and Z. Wang, *Time for statistical model checking of real-time systems*, In: Gopalakrishnan, G., Qadeer, S. (eds.) Intl. Conf. on Computer-Aided Verification (CAV). LNCS, vol. 6806, pp. 349–355. Springer, Heidelberg, 2012.

[33] A. David, K. Larsen, J. Rasmussen, and A. Skou, *Model-Based Design for Embedded Systems*, In: Nicolescu, G., Mosterman, P.J. (eds.) Model-Based Design for Embedded Systems. Computational Analysis, Synthesis, and Design of Dynamic Systems. CRC Press, Boca Raton, 2009.

[34] J. W. De Bakker and J. I. Zucker, *Processes and the denotational semantics of concurrency*, Information and control, 54, pp. 70-120, 1982.

[35] R. De Nicola, *Extensional equivalences for transition systems*, Acta Informatica, 24, pp. 211–237, 1987.

[36] R. De Nicola and F. Vaandrager, *Action versus state based logic for transition systems*, in LITP Spring School on Theoritical Computer science,Springer, pp.407-419, 1990.

[37] H. Dierks, *PLC-automata: a new class of implementable real-time automata*, Theor. Comput. Sci. 253(1), 61–93, 2001.

[38] ——, *Timed automata for modelling and verifying real-time control systems*, Formal Aspects of Computing, 2005.

[39] E. A. Emerson, A. Mok, A. P. Sistla, and J. Srinivasan, *Quantitative temporal reasoning, presented at First Workshop on Computer aided Verification, Grenoble, France*, 1989.

[40] J. Esparza and H. Knoop, *Parameterized model checking of timed systems*, Information and Computation, 2014.

[41] V. Garousi, A. B. Keles, Y. Balaman, Z. O. Guler, and A. Arcur, *Model-based testing in practice: An experience report from the web applications domain .*

[42] R. Groz, A. Simao, A. Petrenko, and C. Oriat, *Inferring finite state machines without reset using state identification sequences, in IFIP International Conference on Testing Software and Systems: 161-177*, Springer, 2015.

[43] K. Havelund, A. Skou, K. Larsen, and K. Lund, *Formal modelling and analysis of an audio/video protocol: an industrial case study using Uppaal.*, n: Symposium on Real-Time Systems (RTSS), pp. 2–13. IEEE, Piscataway, 1997.

[44] M. Hendriks, B. van den Nieuwelaar, and F. Vaandrager, *Model checker aided design of a controller for a wafer scanner*, Int. J. Softw. Tools Technol. Transf. 8(6), 633–647, 2006.

[45] M. Hennessy and R. Milner, *On observing nondeterminism and concurrency, inInternational Colloquium on Automata, Languages, and Programming*, Springer, pp. 299-309, 1980.

[46] M. C. Hennessy, *Algebraic Theory of Processes*, MIT Press., 1988.

[47] R. Hennessy, M.and Milner, *Algebraic laws for nondeterminism and concurrency*, Journal of the ACM (JACM), 1985.

[48] T. A. Henzinger and Z. Manna, *On-the-fly model checking of real-time systems*, Journal of Real-Time Systems, 1991.

[49] C. Hoare, *Communicating Sequential processes*, Prentice Hall, 1985.

[50] C. A. Hoare, *Communicating Sequential Processes. Communications of ACM, 21.*, 1978.

[51] C. A. R. Hoare, S. D. Brookes, and A. W. Roscoe, *A theory of communicating Sequential processes*, Oxford University Computing Laboratory, Programming Research Group, 1981.

[52] G. Igna, V. Kannan, Y. Yang, T. Basten, M. Geilen, F. Vaandrager, M. Voorhoeve, S. de Smet, and L. Somers, *Formal modeling and scheduling of datapaths of digital document printers*, In: Cassez, F., Jard, C. (eds.) International Conference on Formal Modelling and Analysis of Timed Systems (FORMATS). LNCS, vol. 5215, pp. 170–187. Springer, Heidelberg, 2008.

[53] M. Jaghoori, F. de Boer, T. Chothia, and M. Sirjani, *Schedulability of asynchronous realtime concurrent objects*, J. Log. Algebraic Program. 78(5), 402–416, 2009.

[54] J. Jensen, J. Rasmussen, K. Larsen, and A. David, *Guided controller synthesis for climate controller using UPPAAL Tiga*, In: Raskin, J.-F., Thiagarajan, P.S. (eds.) International Conference on Formal Modelling and Analysis of Timed Systems (FORMATS). LNCS, vol. 4763, pp. 227–240. Springer, Heidelberg, 2007.

[55] J. R. Kennaway, *Formal semantics of nondeterminism and parallelism*, PhD thesis, University of Oxford, 1981.

[56] R. Koymans, *Specifying real-time properties with metric temporal logic*, J. Real-Time Systems 2, 255-299, 1990.

[57] Larsen, G. Kim, and Y. Wang, *Timed-abstracted bisimulation:Implicit specifications and decidability*, Information and Computation, 1994.

[58] K. Larsen, G. Behrmann, E. Brinksma, A. Fehnker, T. Hune, P. Pettersson, and J. Romijn, *As cheap as possible: efficient cost-optimal reachability for priced timed automata*, In: Berry, G., Comon, H., Finkel, A. (eds.) Intl. Conf. on Computer-Aided Verification (CAV). LNCS, vol. 2102, pp. 493–505. Springer, Heidelberg, 2001.

[59] K. G. Larsen and U. Nyman, *Compositional model checking of real-time systems*, Formal Methods in System Design, 2006.

[60] K. G. Larsen, P. Pettersson, and W. Yi, *UPPAAL in a nutshell*, International Journal on Software Tools for Technology Transfer (STTT), 1997.

[61] N. Lynch and H. Attiya, *Using mapping to prove timing properties, in "Proceeding of the Ninth ACM Symposium on Principles of Distributed Computing."*, 1990.

[62] A. Mader and H. Wupper, *Timed automaton models for simple programmable logic controllers*, In: Euromicro Conference on Real-Time Systems (ECRTS), pp. 106–113. IEEE, Piscataway, 1999.

[63] R. Milner, *Communication and Concurrency. Prentice Hall International*.

[64] ——, *A Calculus of Communicating Systems, Lecture Notes in Computer Science 92*, Springer-Verlag., 1980.

[65] ——, *Calculi for synchrony and Asynchrony*, Theoritical Computer Science 25, pp.267-310, 1983.

[66] F. Moller, *Process Algebra as a Tool for Real Time Analysis.*, British Computer Society., 1991.

[67] R. D. Nicola and M. C. Hennessy, *Testing equivalence for processes, Theoritical Computer Science. 34, 19-32.*

[68] N. Nourollahi, *Verification and TCTL Model Checking of Real-Time Systems on Timed Automata and Timed Kripke Structures, and Inductive Conversion Issues in Dense Time*, Masters thesis, Bishop's University, Canada., 2019.

[69] E.-R. Olderog and C. A. R. Hoare, *Specification-oriented semantics for communicating processes*, Acta Informatica, 23, pp. 9–66., 1986.

[70] D. Park, *Concurrency and automata on infinite sequences*, Theoretical Computer Science, 1981.

[71] K. Pawlikowski, *Steady-state simulation of queueing processs: Survey of problem and solutions*, Survey of problem and solutions. ACM computing survey, 22, 123-170., 1990.

[72] A. Petrenko, A. Simao, and J. C. Maldonado, *Model-based testing of software and systems: recent advances and challenges*, Springer, 2012.

[73] G. Plotkin, *A structural approach to operational semantics.*, Tech. Rep. DAIMI FN-19, Computer Science Department, Aarhus University, 1981.

[74] A. Pnueli, *A temporal logic of concurrent of concurrnt programs, Theoritical Computer Science, 13. 45 -60.*, 1981.

[75] ——, *Linear and branching structures in the semantics and logics of reactive systems,in International Colloquium on Automata, Languages, and Programming*, Springer, pp. 15–32, 1985.

[76] L. Pomello, *Some equivalence notions for concurrent systems. an overview, in European Workshop on Applications and Theory in Petri Nets*, Springer, pp. 381–400., 1985.

[77] G. Reed and A. Roscoe, *A Timed Model for Communicating Sequential Processes*, In Proceedings of ICALP'86, Lecture Notes in Computer Science, Vol. 226, Springer, Berlin, 1986.

[78] H. Saidi, *The invariant checker:Automated deductive verification of reactive systems in proceedings of Computer Aided verification (CAV 97)*, 1254 of Lecture Notes In computer Science., 1997.

[79] S. Schneider, *An Operational Semantics for Timed CSP*, Programming Research Group Technical Report TR-1-91, Oxford University, 1991.

[80] S. Schneider, *Concurrent and Real-time Systems*, The CSP Approach. Wiley., 2000.

[81] T. J. Schriber, J. Banks, A. F. Seila, I. Stahl, A. M. Law, and R. G. Born, *Simulation textboks-old and new, panel, in Winter simulation conference. 1952-1963.*

[82] M. Schuts, Y. Zhu, F. Heidarian, and F. Vaandrager, *Modelling clock synchronization in the Chess gMACWSN protocol*, In: Andova, S., McIver, A.K., D'Argenio, P.R., Cuijpers, P.J.L., Markovski, J., Morgan, C.C., Núñez, M. (eds.) Workshop on Quantitative Formal Methods: Theory and Applications (QFM). Electronic Proceedings in Theoretical Computer Science, vol. 13, pp. 41–54, 2009.

[83] L. Steggles, *pecifying and Veryfying Real-time Systems using second order Algebraic Methods: A case study of the Railroad Crossing Controller.*

[84] L. J. Steggles, *Extensions of Higher-Order Algebra: Fundamental Theory and Case Studies*, Ph.D. Thesis, University of Wales, Swansea., 1995.

[85] W. Thomas, *Automata on infinite objects:A Handbook of theoritical Computer Science, J. van Leauwen, ed, Vol B. 133-191.*

[86] J. Tretmans, *onformance testing with labelled transition systems: Implementation relations and test generation.*, Computer Networks and ISDN Systems, 29, 49-79., 1996.

[87] ——, *Model based testing with labelled transition systems, in Formal methods and testing.*, Springer, 1-38., 2008.

[88] S. Tripakis, *Description and schedulability analysis of the software architecture of an automated vehicle control system*, In: Sangiovani-Vincentelli, A.L., Sifakis, J. (eds.) Int. Conf. on Embedded Software (EMSOFT). LNCS, vol. 2491, pp. 123–137. Springer, Heidelberg, 2002.

[89] S. Tripakis and S. Yovine, *Timing analysis and code generation of vehicle control software using Taxys*, In: Havelund, K., Ro¸su, G. (eds.) International Workshop on Runtime Verification (RV). Electronic Notes in Theoretical Computer Science, vol. 55, pp. 277–286. Elsevier, Amsterdam, 2001.

[90] M. Utting and B. Legeard, *Practical model-based testing: a tools approach*.

[91] F. Wang, *Formal Verification of Timed Systems:*, A survey and perspective, proceedings of the IEEE, 2012.

[92] Y. Wang, *Formal Description of the UML Architecture and Extendibility*, The International Journal of the Object 6, 4, 469–488, 2001.

[93] L. Waszniowski and Z. Hanzálek, *Formal verification of multitasking applications based on timed automata model.*, Real-Time Syst. 38(1), 39–65, 2008.