Formal Methods For Concurrent And Real Time Systems

Christian Rossi

Academic Year 2024-2025

Abstract

The goal of this course is to develop the ability to analyze, design, and verify critical systems, with a particular focus on real-time aspects, using formal methods. Key topics covered include Hoare's method for program specification and verification, specification languages for real-time systems, and case studies based on industrial projects. The course aims to provide a solid foundation in applying formal methods to ensure the reliability and correctness of systems, particularly in time-sensitive contexts.

Contents

1 Introduction **1**

1.1 Formal methods . . . 1

1.2 Concurrent systems . . . 1

1.2.1 Time formalization . . . 2

1.3 Critycal systems . . . 3

1.3.1 Formal verification . . . 3

1.3.2 Model checking . . . 3

2 Transition systems **4**

2.1 Introduction . . . 4

2.1.1 Determinism . . . 4

2.1.2 Run . . . 5

2.2 Program graphs . . . 5

2.3 Concurrency . . . 6

2.3.1 Handshaking . . . 6

2.3.2 Channel system . . . 7

2.4 Nano Promela . . . 8

2.4.1 Syntax . . . 8

2.4.2 Features . . . 9

2.5 Linear time properties . . . 9

2.5.1 Linear time property in Transition System . . . 9

2.5.2 Linear time property taxonomy . . . 9

2.5.3 Decomposition theorem . . . 10

2.6 Fairness . . . 10

2.6.1 Fairness formalization . . . 11

2.6.2 Fairness and safety . . . 12

3 Model checking **13**

3.1 Safety property . . . 13

3.1.1 Invariant checking . . . 13

3.1.2 Algorithm . . . 14

3.2 Liveness property . . . 14

3.2.1 Regular languages over infinite words . . . 14

3.2.2 Nondeterministic Büchi automata . . . 15

3.2.3 Regular property model checking . . . 15

3.3 Temporal logic . . . 16

3.4 Linear Time Logic . . . 16

3.4.1 Syntax . . . 17
3.4.2 Semantics . . . 17
3.4.3 Fainrness . . . 18
3.4.4 Positive Normal Form . . . 18
3.4.5 Automata model checking . . . 19
3.5 Computation Tree Logic . . . 21
3.5.1 Syntax . . . 21
3.5.2 Semantic . . . 22
3.5.3 Fairness . . . 22
3.5.4 Model checking . . . 23
3.6 Symbolic model checking . . . 24
3.7 Bounded model checking . . . 25
3.7.1 Back loops and transition relation . . . 26
3.7.2 Temporal logic properties . . . 26
3.7.3 Model checking . . . 27
3.8 Probabilistic model checking . . . 28
3.8.1 Markov chains . . . 28
3.8.2 Probabilistic Computation Tree Logic . . . 29
3.8.3 Markov decision processes . . . 30

4 Properties 31
4.1 Equivalence . . . 31
4.1.1 Trace equivalence . . . 31
4.1.2 Bisimulation . . . 31
4.1.3 Quotient Transition System . . . 33
4.1.4 Computation Tree Logic equivalence . . . 33
4.2 Transition systems refinement . . . 33
4.2.1 Simulation . . . 34
4.2.2 Abstraction . . . 35
4.2.3 Safety property . . . 35
4.2.4 Path fragments . . . 35
4.2.5 Simulation quotient . . . 36
4.2.6 Atomic proposition determinism . . . 36
4.2.7 Action based bisimulation . . . 36

5 Timed automata and languages 37
5.1 Introduction . . . 37
5.2 Timed Automata . . . 37
5.2.1 Clocks . . . 38
5.2.2 Transition systems as Timed Automata . . . 39
5.2.3 Clock valuation . . . 39
5.2.4 Transitions . . . 40
5.2.5 Definitions and diagonal constraints . . . 41
5.3 Timed Automata analysis . . . 42
5.3.1 Clock regions . . . 42
5.3.2 Region based automata . . . 43
5.3.3 Time abstracted bisimulation . . . 43
5.4 Verification . . . 44
5.4.1 Timed Computation Tree Logic . . . 44

5.4.2 Model checking . . . 46
5.5 Timed languages . . . 47
5.5.1 Timed Automata . . . 48
5.5.2 Timed regular languages . . . 48
5.5.3 Timed Automata properties . . . 49
5.5.4 Model Checking Timed Automata for Linear Temporal Logic . . . 49
5.6 Uppaal . . . 49
5.6.1 Query language . . . 50

6 Hoare logic **51**
6.1 Introduction . . . 51
6.1.1 Correctness . . . 51
6.2 Syntax and semantic . . . 52
6.3 Axioms . . . 52
6.3.1 Loop correctness . . . 53
6.4 Structural rules . . . 53

CHAPTER 1

Introduction

1.1 Formal methods

Informal methods often suffer from several major issues:

- *Lack of precision*: ambiguous definitions and specifications can lead to misunderstandings and errors in interpretation.
- *Unreliable verification*: traditional testing methods have well-known limitations, making it difficult to ensure correctness.
- *Safety and security risks*: if a flawed program were part of a critical system, it could result in serious consequences.
- *Economic impact*: errors in software can lead to financial losses.
- *Limited generality and reusability*: informal approaches often produce software that is difficult to reuse, adapt, or port to different environments.
- *Overall poor quality*: the lack of rigorous foundations can lead to unreliable and suboptimal software.

Formal methods offer a structured, mathematical approach to software and system development. Ideally, they provide a comprehensive formalization (every aspect of the system is modeled mathematically), and mathematical reasoning and verification (analysis is performed using formal proofs and supported by specialized tools). By applying formal methods, we can achieve greater precision, reliability, and confidence in complex systems.

1.2 Concurrent systems

When transitioning from sequential to concurrent or parallel systems, fundamental shifts occur in how we define and model computation:

- Usually, the traditional problem formulation changes significantly.
- The rise of networked and interactive systems demands new models focused on interactions rather than just algorithmic transformations.

- Many modern systems do not have a clear beginning and end but instead involve continuous, ongoing computations. This requires us to consider infinite sequences (infinite words), leading to a whole branch of formal language theory designed for such systems.
- We must account for interleaved signals flowing through different channels.

Definition (*System*)**.** A system is a collection of abstract machines, often referred to as processes.

In some cases, we can construct a global state by combining the local states of individual processes. However, with concurrent systems, this is often inconvenient or even impossible:

- Each process evolves independently, synchronizing only occasionally.
- Asynchronous systems do not have a globally synchronized state.
- Finite State Machines capture interleaving semantics but differ fundamentally from asynchronous models.

In distributed systems, components are physically separated and communicate via signals. As system components operate at speeds approaching the speed of light, it becomes meaningless to assume a well-defined global state at any given moment.

1.2.1 Time formalization

When time becomes a factor in computation, things become significantly more complex. Unlike traditional engineering disciplines computer science often abstracts away from time, treating it separately in areas like complexity analysis and performance evaluation.

While this abstraction is sufficient for many applications, it is inadequate for real-time systems, where correctness explicitly depends on time behavior. In such systems, we must consider:

1. The occurrence and order of events.
2. The duration of actions and states.
3. Interdependencies between time and data.

Over the years, time has been integrated into formal models in various ways.

Operational formalism These approaches incorporate time directly into system execution models: timed transitions, timed Petri networks, and time as a system variable.

Descriptive formalism These approaches focus on reasoning about time without explicitly simulating execution: temporal logic (treats time as an abstract concept, focusing on event ordering rather than durations), and metric temporal logics (extensions of temporal logic introduce time constraints).

1.3 Critycal systems

In critical applications, precision and rigor are essential. One way to achieve this is through formal techniques, which rely on mathematical models of the system being designed.

By using formal models, we can (at least in principle) verify system properties with a high degree of confidence. In many cases, this verification can be automated, reducing the risk of human error.

1.3.1 Formal verification

When developing a critical system, we define:

- Specification (S): a high-level formal model of the system.
- Requirement (R): a property we want the system to satisfy.

Requirements are typically divided into two main categories:

1. *Functional requirements*: define expected input/output behaviors.
2. *Non-functional requirements*: covers aspects such as ordering constraints, metric constraints, probabilistic guarantees, and real-time probabilistic constraints

Once we have formalized R and S, we aim to verify that R holds given S. This is denoted as:

$$R \models S$$

Which means that property R holds for specification S. The ultimate goal of formal verification is to determine whether this statement is true or false.

1.3.2 Model checking

Definition (*Model checking*)**.** Model checking is an automated technique that, given a finite-state model of a system and a formal property, systematically checks whether this property holds in the model.

In model checking, the system is typically represented as a finite-state automaton or a similar formal model. The properties to be verified are expressed in temporal logic, which allows reasoning about sequences of events over time. The fundamental idea is to explore all possible system states to determine whether the given property holds.

If verification succeeds, it provides strong assurance that the system behaves as expected. However, if the verification fails, the model checker generates a counterexample, which serves as a concrete illustration of a scenario where the property does not hold. This counterexample is invaluable for debugging and refining the system.

Advantages One of the greatest advantages of model checking is its high degree of automation. Once the system model and properties are specified, the verification process becomes essentially a push-button task.

Drawbacks A major issue is state space explosion, where the number of possible states grows exponentially with system complexity, making verification computationally expensive. Additionally, certain complex system behaviors may be difficult to express within the formalism, limiting the technique's applicability in some cases.

CHAPTER 2

Transition systems

2.1 Introduction

A Transition System is a fundamental model used to describe the behavior of dynamic systems. It consists of a set of states and transitions, which define how the system evolves in response to actions.

Definition. A Transition System is a tuple $\mathrm{TS} = \langle S, \mathrm{Act}, \rightarrow, I, \mathrm{AP}, L \rangle$, where:

- S is a set of states.
- Act is a set of input symbols (also called actions).
- $\rightarrow \subseteq \times \mathrm{Act} \times S$ is a transition relation defining how states evolve.
- $I \subseteq S$ is a nonempty set of initial states.
- AP is a set of atomic propositions, used to label states.
- $L : S \rightarrow 2^{\mathrm{AP}}$ is a labeling function, assigning each state a subset of atomic propositions.

The sets of states, actions, and atomic propositions may be finite or infinite. Additionally, a special action, denoted τ, represents an internal (silent) event.

2.1.1 Determinism

A Transition System can be either deterministic or nondeterministic, depending on how transitions are defined.

Definition (*Deterministic Transition System*)**.** A Transition System is deterministic if, for every state s and input i, there is at most one state s' such that $\langle s, i, s' \rangle \in \rightarrow$.

If multiple successor states exist for the same state and input, the system is nondeterministic.

2.1.2 Run

The execution of a Transition System is captured through runs, which describe sequences of state transitions in response to input actions.

Definition (*Run*)**.** Given a (possibly infinite) sequence $\sigma = i_1 i_2 i_3 \dots$ of input symbols from Act, a run r_σ of a Transition System $\langle S, \text{Act}, \rightarrow, I, \text{AP}, L\rangle$ is a sequence:

$$s_0 i_1 s_1 i_2 s_2 \dots$$

Here, $s_0 \in I$, each $s_j \in S$ and for all $k \geq 0$, the transition $\langle s_k, i_{k+1}, s_{k+1}\rangle \in\rightarrow$ holds.

If the Transition System is nondeterministic, multiple runs may exist for the same input sequence.

Definition (*Reachable state*)**.** A state s' is reachable if there exists an input sequence $\sigma = i_1 i_2 \dots i_k$ and a finite run $r_\sigma = s_0 i_1 s_1 i_2 s_2 \dots i_k s'$.

A key aspect of Transition Systems is the trace, which records the sequence of state labels encountered during a run.

Definition. Given a run r_σ, its trace is the sequence of atomic proposition subsets:

$$L(s_0)L(s_1)L(s_2)\dots$$

Sometimes, the term trace is also used to refer to the input sequence σ that generates a run r_σ, in which case it is called an input trace.

A run may be finite if it reaches a terminal state (a state with no outgoing transitions). However, many systems, particularly reactive systems, are modeled using infinite runs, as they are designed to operate indefinitely rather than terminate.

2.2 Program graphs

A common transformation in system modeling is moving external inputs into state labels. This approach simplifies definitions and system analysis by leaving only internal communications as actual inputs.

When dealing with variables, Transition Systems are referred to as Program Graphs. A Program Graph consists of:

- A set of variables, where each variable has a value assigned in every state by an evaluation function.
- Transitions that may include conditions based on variable values.
- An effect function, which describes how inputs modify variable values.
- States, which are typically called locations in the context of Program Graphs.

Transformation Program graphs can always be converted into a (potentially infinite) Transition System. However, Transition System do not inherently include guards or variables. Instead:

- Guards can be represented as symbols in a set of atomic propositions.
- The atomic proposition set must also include all locations from the Program Graph.
- While this transformation results in a very large atomic proposition set, in practice, only a small portion is usually relevant for analyzing system properties.

2.3 Concurrency

Given two Transition Systems:

$$\text{TS}_1 = \langle S_1, \text{Act}_1, \rightarrow_1, I_1, \text{AP}_1, L_1 \rangle \qquad \text{TS}_2 = \langle S_2, \text{Act}_2, \rightarrow_2, I_2, \text{AP}_2, L_2 \rangle$$

Their interliving is defined as:

$$\text{TS}_1 \mid\mid\mid \text{TS}_2 = \langle S_1 \times S_2, \text{Act}_1 \cup \text{Act}_2, \rightarrow_1, I_1 \times I_2, \text{AP}_1 \cup \text{AP}_2, L \rangle$$

Here, $L(\langle s_1, s_2 \rangle) = L(s_1) \cup L(s_2)$ and the transition relation $\rightarrow$ is:

$$\frac{s_1 \xrightarrow{\alpha} s_1'}{\langle s_1, s_2 \rangle \xrightarrow{\alpha} \langle s_1', s_2 \rangle} \wedge \frac{s_2 \xrightarrow{\alpha} s_2'}{\langle s_1, s_2 \rangle \xrightarrow{\alpha} \langle s_1, s_2' \rangle}$$

In practice, the two Transition System proceed independently (alternating nondeterministically), but only one at a time is considered to be "active". Similar to non-synchronizing threads: all possible interleavings are allowed.

No shared variables When two Program Graphs, PG_1 and PG_2, do not share variables, their interleaving can be naturally defined as:

$$\text{TS}_1(\text{PG}_1) \mid\mid\mid \text{TS}_2(\text{PG}_2)$$

This straightforward composition allows both Transition Systems to operate independently.

Shared variables If PG_1 and PG_2 share variables, the simple interleaving:

$$\text{TS}_1(\text{PG}_1) \mid\mid\mid \text{TS}_2(\text{PG}_2)$$

May not be valid, as some locations might become inconsistent. This happens because both Program Graphs access shared critical variables, leading to potential conflicts.

Constraint synchronization To ensure consistency, components must coordinate by imposing constraints on shared variables. Execution progresses only when the conditions are satisfied in both Transition Systems. This synchronization mechanism ensures that shared variables remain valid across all transitions.

2.3.1 Handshaking

In parallel composition with handshaking, two Transition Systems synchronize on a set of shared actions H, which is a subset of their common actions:

$$\text{TS}_1 \parallel_H \text{TS}_2$$

They evolve independently (interleaving) for actions outside H. This is similar to firing a transition in Petri nets. To synchronize, processes must shake hands, a concept also known as Synchronous Message Passing.

If there are no shared actions $\text{Act}_1 \cap \text{Act}_2$, handshaking reduces to standard interleaving:

$$\text{TS}_1 \parallel_\varnothing \text{TS}_2 = \text{TS}_1 \mid\mid\mid \text{TS}_2$$

If H includes all common actions, we simply write:

$$\text{TS}_1 \parallel \text{TS}_2$$

Given two Transition Systems:

$$\text{TS}_1 = \langle S_1, \text{Act}_1, \rightarrow_1, I_1, \text{AP}_1, L_1 \rangle \qquad \text{TS}_2 = \langle S_2, \text{Act}_2, \rightarrow_2, I_2, \text{AP}_2, L_2 \rangle$$

Their handshaking synchronization is defined as:

$$\frac{s_1 \xrightarrow{\alpha}_1 s_1' \land s_2 \xrightarrow{\alpha}_2 s_2'}{\langle s_1, s_2 \rangle \xrightarrow{\alpha} \langle s_1', s_2' \rangle}$$

This means that both systems must simultaneously perform the shared action α to transition together.

Broadcasting If a fixed set of handshake actions H exists such that:

$$\text{Act}_1 \cap \text{Act}_2 \cdots \cap \text{Act}_n$$

Then all processes can synchronize on these actions. In this case, the handshaking operator $\parallel_H$ is associative, meaning we can compose multiple Transition Systems as:

$$\text{TS} = \text{TS}_1 \parallel_H \text{TS}_2 \parallel_H \cdots \parallel_H \text{TS}_n$$

This allows for synchronized execution across multiple processes.

2.3.2 Channel system

Handshaking synchronization does not inherently introduce a direction for message exchange. In other words, it lacks a cause-effect relationship between components during synchronization.

However, in many real-world scenarios, directionality is natural (one component sends a message, and another receives it). To model this, we use first in first out channels, which explicitly define the direction of communication. Now, transitions in a system include:

$$\rightarrow \subseteq S \times (\text{Act} \cup C! \cup C?) \times S$$

Here, $C!$ represents sending operations, with messages of the form $c!x$ (sending x through channel c) and $C?$ represents receiving operations, with messages of the form $c?x$ (receiving x from channel c).

Channel capacity The capacity of a first in first out channel determines how many events (messages) can be stored in its buffer at a time:

- If capacity(c) $= 0$, the sender and receiver must synchronize instantly (just like standard handshaking), but with a different syntax.
- If capacity(c) > 0, the sender can execute $c!x$ without waiting for a receiver, as long as the buffer isn't full. If the channel is full, the sender is blocked until space becomes available. A receiver performing $c?x$ is blocked until x reaches the front of the queue.

2.4 Nano Promela

Transition Systems provide a mathematical foundation for modeling and verifying reactive systems. However, in practice, we need more user-friendly specification languages.

One such language is Promela, designed for the SPIN model checker to describe Transition Systems. We will focus on a simplified subset of Promela called Nano-Promela.

2.4.1 Syntax

A Promela program consists of a set of interleaving processes that communicate either synchronously or through finite first in first out channels. The syntax of statements in Nano-Promela is as follows:

```
stmt ::= skip | x := expr | c?x | c!expr |
         stmt1; stmt2 | atomic{assignments} |
         if :: g1 => stmt1 ... :: gn => stmtn fi |
         do :: g1 => stmt1 ... :: gn => stmtn do
```

Here:

- `expr` represents an expression.
- `skip` represents a process that terminates in one step, without modifying any variables or channels.
- `stmt1; stmt2` denotes sequential execution: `stmt1` runs first, followed by `stmt2`.
- `atomicassignments` defines an atomic region, meaning `stmt` executes as a single, indivisible step. This prevents interference from other processes and helps reduce verification complexity by avoiding unnecessary interleavings.

Conditional statement The conditional statement is expressed as:

```
if :: g1 => stmt1 ... :: gn => stmtn fi
```

This represents a nondeterministic choice between multiple guarded statements. The system chooses one of the `stmti` for which `gi` holds in the current state. The selection and the first execution step are performed atomically, meaning no other process can interfere. If none of the guards hold, the process blocks. However, other processes may unblock it by changing shared variables, causing one of the guards to become true.

Loop The loop is expressed as:

```
do :: g1 => stmt1 ... :: gn => stmtn do
```

This represents a loop that repeatedly executes a nondeterministic choice among the guarded statements. If a guard `gi` holds, the corresponding `stmti` executes. Unlike `if-fi`, `do-od` does not block when all guards fail; instead, the loop simply terminates.

2.4.2 Features

Nano-Promela can be formally defined using Program Graphs, but full Promela provides additional powerful features, including: more complex atomic regions (beyond just assignments), a rrays and richer data types, and dynamic process creation.

2.5 Linear time properties

A linear time property specifies a desired behavior of a system. Unlike a formula, it is a set of infinite words over the alphabet 2^{AP}, where AP represents a set of atomic propositions. We denote:

- The set of infinite words over alphabet A as A^ω.
- The set of finite words over alphabet A as A^*.
- A linear time property over AP as a subset of $\left(2^{\mathrm{AP}}\right)^\omega$.

2.5.1 Linear time property in Transition System

Definition (*Linear time property*)**.** A Transition System $\mathrm{TS} = (S, \mathrm{Act}, \to, I, \mathrm{AP}, l)$ satisfies a linear time property P if and only if:

$$\mathrm{TS} \models P \Leftrightarrow \mathrm{traces}(\mathrm{TS}) \subseteq P$$

Definition (*Transition equivalence*)**.** Two Transition Systems TS_1 and TS_2 are trace equivalent with respect to AP if:

$$\mathrm{traces}_{\mathrm{AP}}(\mathrm{TS}_1) = \mathrm{traces}_{\mathrm{AP}}(\mathrm{TS}_2)$$

Corollary 2.5.0.1. *If TS_1 and TS_2 are Transition Systems without terminal states and share the same atomic propositions, then:*

$$traces(TS_1) = traces(TS_2)$$

if and only if TS_1 and TS_2 satisfy the same linear time properties.

Thus, no linear time property can distinguish between trace equivalent Transition Systems. To prove that two Transition Systems are not trace-equivalent, it suffices to find a linear time property that holds for one but not the other.

2.5.2 Linear time property taxonomy

Linear time properties for Transition Systems are often expressed using regular properties and finite state automata. They are typically classified as invariants, safety properties, and liveness properties.

2.5.2.1 Invariant

Definition (*Invariant*)**.** An invariant is a linear time property where a propositional logic formula Φ over AP holds at every step:

$$P = \{A_0 A_1 A_2 \cdots \mid A_j \text{ satisfies } \Phi \text{ for all } j \geq 0\}$$

Verifying an invariant involves checking Φ in all states reachable from an initial state. Standard graph traversal algorithms, such as depth-first search or breadth-first search, can efficiently perform this check in linear time relative to the number of states.

2.5.2.2 Safety property

A safety property ensures that a bad event never occurs. If an infinite word σ violates a safety property, then it must contain a bad prefix σ', meaning that any infinite word starting with σ' also violates the property.

Definition (*Safety property*)**.** A safety property P_{safe} is a linear time property over AP if:

$$P_{\text{safe}} \cap \left\{\sigma' \in \left(2^{\text{AP}}\right)^{\omega} \mid \hat{\sigma} \text{ is a finite prefix of } \sigma'\right\} = \varnothing$$

2.5.2.3 Liveness property

A liveness property ensures that something good eventually happens. Unlike safety properties, finite traces provide no information about whether a liveness property holds. Instead, every finite prefix must be extendable to an infinite trace that satisfies the property.

Definition (*Liveness*)**.** A liveness propert P_{live} over AP satisfies:

$$\text{pref}(P_{\text{live}}) = \left(2^{\text{AP}}\right)^{*}$$

This means that every finite word w can be extended to an infinite word σ such that $w\sigma \in P$.

2.5.3 Decomposition theorem

Safety and liveness properties are disjoint. The only linear time property that is both a safety and a liveness property is $\left(2^{\text{AP}}\right)^{\omega}$.

Theorem 2.5.1. *Every linear time property P over AP can be decomposed into a safety property P_{safe} and a liveness property P_{live} such that:*

$$P = P_{safe} \cap P_{live}$$

2.6 Fairness

To ensure realistic system behavior, fairness constraints must be introduced. These constraints prevent unrealistic execution patterns by guaranteeing that processes are given a fair chance to execute. Fairness is especially relevant in concurrent systems where multiple processes compete for execution.

Fairness can be classified into three main types:

- *Unconditional fairness*: every process gets a chance to execute infinitely often, regardless of other conditions.
- *Strong fairness*: if a process is enabled infinitely often, it must eventually execute infinitely often.
- *Weak fairness*: if a process remains continuously enabled from a certain point onward, it must eventually execute infinitely often.

These fairness levels follow a logical hierarchy:

$$\text{unconditional fairness} \implies \text{strong fairness} \implies \text{weak fairness}$$

Definition (*Fairness constraint*)**.** A fairness constraint defines a set of actions that must occur under a given fairness assumption (unconditional, strong, or weak).

These constraints play a crucial role in ensuring liveness properties, which guarantee that something will eventually happen. Fairness constraints can be efficiently expressed using Büchi automata or Linear Temporal Logic. However, incorporating fairness into Transition Systems requires careful handling to ensure correctness.

2.6.1 Fairness formalization

Definition (*Fairness*)**.** Let $\mathrm{TS} = \langle S, \mathrm{Act}, \rightarrow, I, \mathrm{AP}, L\rangle$. The enabled actions at a state s are given by $\mathrm{Act}(s) = \left\{\alpha \in \mathrm{Act} \mid \exists s' \in s, s \xrightarrow{\alpha} s'\right\}$ For an infinite execution fragment $\rho = s_0 \xrightarrow{\alpha_0} s_1 \xrightarrow{\alpha_1} \dots$ we define the fairness conditions:

- *Unconditional fairness*: ρ is unconditionally A-fair if there exists infinitely many j for all $\alpha_i \in A$.
- *Strong fairness*: ρ is strongly A-fair if:

$$(\exists \text{ infinitely many } j \mid \mathrm{Act}(s_j) \cap A \neq \varnothing) \implies (\exists \text{ infinitely many } j \mid \alpha_j \in A)$$

- *Weak fairness*: ρ is weakly A-fair if:

$$(\forall \text{ sufficiently large } j \mid \mathrm{Act}(s_j) \cap A \neq \varnothing) \implies (\exists \text{ infinitely many } j \mid \alpha_j \in A)$$

Definition (*Fairness assumption*)**.** A fairness assumption $\mathcal{F}$ is defined as a triple:

$$\mathcal{F} = \langle \mathcal{F}_{\text{uncond}}, \mathcal{F}_{\text{strong}}, \mathcal{F}_{\text{weak}} \rangle$$

Here, $\mathcal{F}_{\text{uncond}}, \mathcal{F}_{\text{strong}}, \mathcal{F}_{\text{weak}} \subseteq 2^{\mathrm{Act}}$.

An execution ρ is $\mathcal{F}$-fair if:

1. It is unconditionally A-fair for all $A \in \mathcal{F}_{\text{uncond}}$.
2. It is strongly A-fair for all $A \in \mathcal{F}_{\text{strong}}$.
3. It is weakly A-fair for all $A \in \mathcal{F}_{\text{weak}}$.

Fair traces Traces that satisfy a fairness constraint F are called Fair-Traces.

Definition (*Fair traces*)**.** Let P be a linear time property over AP and $\mathcal{F}$ a fairness assumption over Act. A Transition System TS fairly satisfies P, denoted $\mathrm{TS} \models_{\mathcal{F}} P$ if and only if:

$$\mathrm{fairTraces}_{\mathcal{F}}(\mathrm{TS}) \subseteq P$$

This follows the hierarchy:

$$\mathrm{TS} \models_{\mathcal{F}_{\text{weak}}} P \implies \mathrm{TS} \models_{\mathcal{F}_{\text{strong}}} P \implies \mathrm{TS} \models_{\mathcal{F}_{\text{uncond}}} P$$

2.6.2 Fairness and safety

Definition (*Realizable fairness assumption*)**.** A fairness assumption $\mathcal{F}$ for a Transition System TS is called realizable if every reachable state s satisfies:

$$\text{fairPaths}_{\mathcal{F}}(s) \neq \varnothing$$

Theorem 2.6.1. *Let TS be a Transition System with set of propositions AP, $\mathcal{F}$ a realizable fairness assumption, and P_{safe} a safety property. Then:*

$$TS \models P_{safe} \Leftrightarrow TS \models_{\mathcal{F}} P_{safe}$$

This theorem establishes that safety properties are independent of fairness assumptions, meaning that if a system satisfies a safety property, it does so regardless of fairness constraints.

CHAPTER 3

Model checking

3.1 Safety property

Safety properties ensure that a system never reaches an undesirable state. Regular safety properties can be characterized using a nondeterministic finite automaton that recognizes finite words over the power set of atomic propositions, denoted as $\left(2^{\text{AP}}\right)^*$.

Definition (*Safety property model checking*)**.** Given a regular safety property P_{safe} over the atomic propositions AP and a finite Transition System TS (without terminal states), model checking verifies whether:

$$\text{TS} \models P_{\text{safe}}$$

To achieve this, we use an nondeterministic finite automaton $\mathcal{A}$ that recognizes the minimal bad prefixes of P_{safe}. This allows us to define an invariant property:

$$P_{\text{inv}(\mathcal{A})} = \bigwedge_{q \in \mathcal{F}} \neg q$$

Here, $\mathcal{A}$ must not reach a final state.

3.1.1 Invariant checking

Verification of the safety property can be reduced to checking an invariant by following these steps:

1. Construct the product of the Transition System and the nondeterministic finite automaton TS$\otimes\mathcal{A}$. This operation is similar to the synchronous composition of two Nondeterministic Finite Automata.

2. The following conditions are equivalent:

    - The Transition System satisfies the safety property:

    $$\text{TS} \models P_{\text{safe}}$$

    - The set of finite traces of the Transition System does not intersect the language of:

    $$\mathcal{A} : \text{traces}_{\text{fin}}(\text{TS}) \cap L(\mathcal{A}) = \varnothing$$

- The product system satisfies the invariant:

$$\mathrm{TS} \otimes \mathcal{A} \models P_{\mathrm{inv}(\mathcal{A})}$$

Thus, checking a safety property reduces to verifying an invariant in the product system.

3.1.2 Algorithm

Given a finite Transition System TS and a regular safety property P_{safe}, the algorithm returns either true ($\mathrm{TS} \models P_{\mathrm{safe}}$) or false ($\mathrm{TS} \not\models P_{\mathrm{safe}}$), with a counterexample.

Algorithm 1 Safety property model checking

```
1: Let nondeterministic finite automaton $\mathcal{A}$ (with accept states $F$) be such that $\mathcal{L}(\mathcal{A})$ are the
   bad prefixes of $P_{\mathrm{safe}}$
2: Construct the product Transition System $\mathrm{TS} \otimes \mathcal{A}$
3: Check the invariant $P_{\mathrm{inv}(\mathcal{A})}$ with proposition $\neg F = \wedge_{q \in F} \neg q$ on $\mathrm{TS} \otimes \mathcal{A}$
4: if $\mathrm{TS} \otimes \mathcal{A}$ then
5:     return true
6: else
7:     Determine an initial path fragment $\langle s_0, q_1 \rangle, \ldots, \langle s_n, q_{n+1} \rangle$ of $\mathrm{TS} \otimes \mathcal{A}$ with $q_{n+1} \in F$
8:     return (false, $s_0 s_1 \ldots s_n$)
9: end if
```

The time and space complexity of this approach is:

$$\mathcal{O}(|\mathrm{TS}| \cdot |\mathcal{A}|)$$

Here, $|\mathrm{TS}|$ and $|\mathcal{A}|$ denote the number of states and transitions in the Transition System and the nondeterministic finite automaton, respectively.

3.2 Liveness property

Liveness properties ensure that certain desired behaviors eventually occur in a system. Unlike safety properties, liveness properties cannot be verified on a finite prefix of a run. Instead, they require reasoning about infinite sequences, necessitating a framework to handle infinite words.

3.2.1 Regular languages over infinite words

An infinite word over an alphabet Σ is an infinite sequence $A_0 A_1 A_2 \ldots$ where each symbol $A_i \in \Sigma$. The set of all infinite words over Σ is denoted by Σ^ω. Any subset of Σ^ω is called an ω-language.

Regular languages over infinite words, known as ω-regular languages, can be defined using automata or generalized regular expressions. While ω-regular expressions provide an intuitive understanding, automata-based definitions are more practical for verification purposes.

3.2.2 Nondeterministic Büchi automata

A nondeterministic Büchi automaton is a variation of a nondeterministic finite automaton that accepts infinite words instead of finite ones.

Definition (*Nondeterministic Büchi automaton*)**.** A nondeterministic Büchi automaton $\mathcal{A}$ is formally defined as a tuple $\mathcal{A} = \langle Q, \Sigma, \delta, Q_0, F \rangle$, where:

- Q is a finite set of states.
- Σ is an input alphabet.
- $\delta : Q \times \Sigma \to 2^Q$ is the transition function.
- $Q_0 \subseteq Q$ is a set of initial states.
- $F \subseteq Q$ is a set of final (accepting) states.

3.2.2.1 Acceptance condition

A run of $\mathcal{A}$ on an infinite word $\sigma = A_0 A_1 A_2 \cdots \in \Sigma^\omega$ is an infinite sequence of states $q_0 q_1 q_2 \ldots$ such that:

- $q_0 \in Q_0$ (initial state).
- $q_i \xrightarrow{A_i} q_{i+1}$ for all $i \geq 0$.

The run is accepted if it visits a state in F infinitely often. The language recognized by $\mathcal{A}$ is:

$$\mathcal{L}_\omega(\mathcal{A}) = \{\sigma \in \Sigma^\omega \mid \text{there exists an accepting run for } \sigma \text{ in } \mathcal{A}\}$$

3.2.2.2 Deterministic and nondeterministic comparison

Nondeterministic Büchi automata are strictly more powerful than deterministic Büchi automata.

Theorem 3.2.1. *There does not exists a deterministic Büchi automata $\mathcal{A}$ such that $\mathcal{L}_\omega(\mathcal{A}) = \mathcal{L}_\omega((A+B)^* B^\omega)$*

3.2.3 Regular property model checking

Given a finite Transition System TS without terminal states and an ω-regular P, the goal is to verify whether TS $\models P$. This is equivalent to checking whether the traces of TS intersect with the complement of P, recognized by an nondeterministic Büchi automata $\mathcal{A}$:

$$\text{traces(TS)} \cap \mathcal{L}_\omega(\mathcal{A}) \neq \varnothing$$

3.2.3.1 Algorithm

The verification process consists of the following steps:

1. *Construct the product automaton*: compute the product $\mathrm{TS} \otimes \mathcal{A}$, which combines paths in TS with runs in $\mathcal{A}$.
2. *Graph analysis*: check if there exists a path in $\mathrm{TS} \otimes \mathcal{A}$ that visits an accepting state infinitely often.
3. *Counterexample detection*: if such a path exists, it serves as a counterexample, proving that TS does not satisfy P. Otherwise, all runs corresponding to traces in Transition System are non-accepting, meaning $\mathrm{TS} \models P$.

The core of the algorithm is detecting cycles in the product automaton that include accepting states. This can be achieved using depth-first search. With optimizations like nested depth-first search, the algorithm runs in linear time concerning the size of the product graph. However, complementing an Nondeterministic Büchi automaton recognizing P is computationally expensive, requiring up to $(0.76n)^n$ time, making direct complementation impractical.

3.3 Temporal logic

Temporal logic is widely used to specify and verify properties of Transition Systems. It allows expressing conditions over time, ensuring that a system behaves as expected in different scenarios.

Taxonomy Temporal logics can be classified based on different criteria:

- *Time representation*: discrete-time or ontinuous-time.
- *Metric constraints*: metric or non-metric.
- *Computation structure*: linear or branching.

Among these, two primary temporal logics are commonly used in system verification:

- *Linear Temporal Logic*: focuses on linear sequences of states.
- *Computation Tree Logic*: explores branching structures of state transitions.

3.4 Linear Time Logic

Linear Temporal Logic is a formalism used to describe temporal properties of systems. It is widely used in model checking, verification, and automated reasoning. Linear Temporal Logic operates over sequences of states, allowing the specification of temporal constraints using logical operators.

3.4.1 Syntax

Linear Time Logic formulas are constructed using the following grammar:

$$\phi ::= \text{true} \mid a \mid \phi_1 \wedge \phi_2 \mid \neg\phi \mid \circ\phi \mid \phi_1 \cup \phi_2$$

Here, a is an atomic proposition, $\circ\phi$ (next) asserts that ϕ holds in the next state, $\phi_1 \cup \phi_2$ (until) states that ϕ_2 will eventually hold, and until then, ϕ_1 must hold. From this, we can derive the eventually ($\Diamond\phi = \text{true} \cup \phi$) and always ($\Box\phi = \neg\Diamond\neg\phi$) operators.

3.4.2 Semantics

TL formulas are interpreted over infinite sequences of states $\sigma = A_0 A_1 \ldots$, where $A_i \subseteq 2^{\text{AP}}$. The satisfaction relation $\sigma \models \phi$ is defined as:

- $\sigma \models \text{true}$
- $\sigma \models a \Leftrightarrow a \in A_0$
- $\sigma \models \phi_1 \wedge \phi_2 \Leftrightarrow \sigma \models \phi_1 \wedge \sigma \models \phi_2$
- $\sigma \models \neg\phi \Leftrightarrow \sigma \not\models \phi$
- $\sigma \models \circ\phi \Leftrightarrow \sigma[1\ldots] = A_1 A_2 A_3 \ldots \models \phi$
- $\sigma \models \phi_1 \cup \phi_2 \Leftrightarrow \exists j \geq 0 \quad \sigma[j\ldots] \models \phi_2 \wedge \sigma[1\ldots] \models \phi_1 \forall 0 \leq i < j$
- $\sigma \models \Diamond\phi \Leftrightarrow \exists j \geq 0 \quad \sigma[j\ldots] \models \phi$
- $\sigma \models \Box\phi \Leftrightarrow \forall j \geq 0 \quad \sigma[j\ldots] \models \phi$

The timed interpretation of the operators is as follows:

- $\circ^k\phi$ means ϕ holds after exactly k steps.
- $\Diamond^{\leq k}\phi$ means ϕ will hold within at most k steps.
- $\Box^{\leq k}\phi$ means ϕ holds now and for the next k steps.

Past operators Although Linear Temporal Logic primarily focuses on the future, past operators can be introduced without increasing expressive power:

- *Previous*: $\bullet\phi$ holds if ϕ was true in the previous state.
- *Since*: $\phi S \psi$ means ψ held at some past time, and ϕ was true until then.
- *Eventually in the past*: $\blacklozenge\phi$ is true $S\phi$.
- *Always in the past*: $\blacksquare\phi = \neg\blacklozenge\neg\phi$.

Metric operators Metric extensions allow explicit bounds on temporal operators:

- $\phi\mathcal{U}_{\sim\sqcup}\psi$ means ψ holds within time t while ϕ holds until then.
- $\phi\mathcal{S}_{\sim\sqcup}\psi$ means ψ holds within time t while ϕ holds until then.

3.4.3 Fainrness

Linear Temporal Logic is defined over atomic propositions, not directly over actions. Therefore, fairness constraints in Linear Temporal Logic are expressed in terms of states, rather than actions.

Action-based fairness is more intuitive and straightforward but Linear Temporal Logic fairness is equally expressive. Action-based fairness assumptions can be translated into Linear Temporal Logic fairness assumptions. One way to achieve this is by making a copy of each non-initial state s and recording which action led to this state. For each possible action a, a copy of the state is created to indicate that state s was reached through action a. This copied state, denoted $\langle s, a\rangle$, indicates that the state s has been reached via action a.

In Linear Temporal Logic, fairness can be expressed using different types of fairness constraints:

1. *Unconditional fairness*: $\square\blacklozenge\psi$.
2. *Strong fairness*: $\square\blacklozenge\phi \rightarrow \square\blacklozenge\psi$.
3. *Weak fairness*: $\blacklozenge\square\phi \rightarrow \blacklozenge\square\psi$.

Assumptions To combine fairness assumptions, we use:

$$\text{fair} = \text{unconditionally fair} \wedge \text{strongly fair} \wedge \text{weakly fair}$$

This leads to the following:

- Fair paths from state s:

$$\text{fairPaths}(s) = \{\pi \in \text{paths}(s) \mid \pi \models \text{fair}\}$$

- Fair satisfaction of a formula ϕ:

$$s \models_{\text{fair}} \phi \Leftrightarrow \forall \pi \in \text{fairPaths}(s) \quad \pi \models \phi$$

- Fair satisfaction in the Transition System:

$$\text{TS} \models_{\text{fair}} \phi \Leftrightarrow \forall s_0 \in I \quad s_0 \models_{\text{fair}} \phi$$

Theorem 3.4.1. *For a Transition System TS without terminal states, an Linear Temporal Logic formula ϕ, and fairness assumption fair:*

$$TS \models_{fair} \Leftrightarrow TS \models (fair \rightarrow \phi)$$

3.4.4 Positive Normal Form

Positive Normal Form is a canonical form where negations appear only adjacent to atomic propositions. This is similar to disjunctive and conjunctive normal forms in propositional logic. Every Linear Temporal Logic formula can be transformed into Positive Normal Form, but this transformation requires new dual operators.

Operator	**Dual**	**Formula**
OR	AND	$\vee \to \wedge$
Next	Next	$\neg \circ \phi \to \circ\neg\phi$
Until	Weak until	$\neg(\phi \cup \psi) \to (\phi \wedge \neg\psi) W (\neg\phi \wedge \neg\psi)$

The syntax of Linear Temporal Logic in weak until Positive Normal Form is:

$$\phi ::= \text{true} \mid \text{false} \mid a \mid \neg a \mid \phi_1 \wedge \phi_2 \mid \circ\phi \mid \phi_1 \cup \phi_2 \mid \phi_1 W \phi_2$$

Theorem 3.4.2. *For any Linear Temporal Logic formula ϕ, there exists an equivalent Linear Temporal Logic formula in weak until Positive Normal Form.*

However, the size of the resulting formula may be exponential in the size of the original formula.

Release operator The release operator R is defined as:

$$\phi R \psi = \neg(\neg\phi \cup \neg\psi)$$

The semantycs is $\sigma \models \phi R \psi$ if $\forall j \geq 0, \sigma[j \dots] \models \psi$ or $\exists i \geq 0 \quad (\sigma[i \dots] \models \phi) \wedge \forall k \leq i \quad \sigma[k \dots] \models \psi$. Intuitively, the formula $\phi R \psi$ holds if ψ is always true unless ϕ becomes true, at which point the requirement for ψ is released. The syntax for Linear Temporal Logic with the release operator is:

$$\phi ::= \text{true} \mid \text{false} \mid a \mid \neg a \mid \phi_1 \wedge \phi_2 \mid \circ\phi \mid \phi_1 \cup \phi_2 \mid \phi_1 R \phi_2$$

Rewriting rules for Linear Temporal Logic formulas with the release operator:

- $\neg\text{true} \rightsquigarrow \text{false}$
- $\neg\neg\phi \rightsquigarrow \phi$
- $\neg(\phi \wedge \psi) \rightsquigarrow \neg\phi \vee \neg\psi$
- $\neg \circ \phi \rightsquigarrow \circ\neg\phi$
- $\neg(\phi \cup \psi) \rightsquigarrow \neg\phi R \neg\psi$

For any Linear Temporal Logic formula ϕ, there exists an equivalent formula ϕ' in release Positive Normal Form with the same size.

3.4.5 Automata model checking

An important observation is that every Linear Temporal Logic formula ϕ can be represented by a nondeterministic Büchi automaton Let words(ϕ) be the set of ω-words satisfying an Linear Temporal Logic formula ϕ. The model checking condition states that TS $\models \phi$ if and only if:

- traces(TS) $\subseteq$ words(ϕ)
- traces(TS) $\cap \left(2^{\text{AP}}\right)^{\omega} \setminus$ words(ϕ) $= \varnothing$
- traces(TS) $\cap$ words($\neg\phi$) $= \varnothing$

For a nondeterministic Büchi automaton $\mathcal{A}$ with language $\mathcal{L}_\omega(\mathcal{A}) = \text{words}(\neg\phi)$, we have:

$$\text{TS} \models \phi \Leftrightarrow \text{traces(TS)} \cap \mathcal{L}_\omega(\mathcal{A}) = \varnothing$$

Instead of building a Büchi automaton equivalent to the negation of the formula and complementing it, it's more efficient to complement the formula first and then construct the equivalent Büchi automaton.

3.4.5.1 Linear Time Logic to generalized Büchi automaton

The construction first builds a generalized Büchi automaton and then converts it to a nondeterministic Büchi automaton. A generalized Büchi automaton has a set $\mathcal{F} \subseteq 2^Q$ of acceptance sets, where the accepted language consists of all ω-words that have an infinite run $q_0 q_1 q_2 \ldots$ such that for each acceptance set $F \in \mathcal{F}$, there are infinitely many indices i with $q_i \in F$.

For any Linear Temporal Logic formula ϕ, there exists a corresponding generalized Büchi automaton $\mathcal{G}_\phi$. For any Linear Temporal Logic formula ϕ, there exists a nondeterministic Büchi automaton $\mathcal{A}_\phi$ such that words$(\phi) = \mathcal{L}_\omega(\mathcal{G}_\phi)$. This can be constructed in time and space $2^{\mathcal{O}(|\phi|)}$.

Until operator To model the semantics of the until operator ($\cup$), an acceptance set F_ψ is introduced for each subformula $\psi = \phi_1 \cup \phi_2$ of ϕ. The semantics of the until operator ensures that for a word σ to satisfy $\psi = \phi_1 \cup \phi_2$, the condition is met only if ϕ_2 eventually becomes true, while ϕ_1 must hold until ϕ_2 becomes true This is enforced by the acceptance set $F_{\phi_1 \cup \phi_2}$, defined as:

$$F_{\phi_1 \cup \phi_2} = \{B \in Q \mid \phi_1 \cup \phi_2 \notin B \vee \phi_2 \in B\}$$

The complete set of acceptance sets $\mathcal{F}$ for a given Linear Temporal Logic formula is:

$$\mathcal{F} = \{F_{\phi_1 \cup \phi_2} \mid \phi_1 \cup \phi_2 \in \text{closure}(\phi)\}$$

Theorem 3.4.3. *For any Linear Temporal Logic formula ϕ (over a set of atomic propositions AP) there exists a nondeterministic Büchi automaton $\mathcal{A}_\phi$ such that words$(\phi) = \mathcal{L}_\omega(\mathcal{A}_\phi)$ which can be constructed in time and space $2^{\mathcal{O}(|\phi|)}$.*

Theorem 3.4.4. *There exists a family of Linear Temporal Logic formulas ϕ_n such that every nondeterministic Büchi automaton has at least 2^n states.*

While the nondeterministic Büchi automaton construction can have an exponential number of states, these automata are more expressive than Linear Temporal Logic formulas themselves.

Complexity The time and space complexity of the Linear Temporal Logic model-checking algorithm is PSPACE-complete, with the complexity of checking a formula ϕ against a Transition System TS given by:

$$\mathcal{O}(|\text{TS}|\, 2^{|\phi|})$$

However, in practice, the performance can be quite good due to optimizations, such as on-the-fly model checking. This approach, which constructs the nondeterministic Büchi automaton for the negation of ϕ during the process of checking the system, can help avoid constructing the entire automaton upfront.

3.4.5.2 Algorithm

The satisfiability and validity of Linear Temporal Logic formulas can be determined by checking the emptiness of the corresponding nondeterministic Büchi automaton. This check can be performed using a nested depth-first search that looks for a reachable cycle containing an accepting state. Both satisfiability and validity checking are PSPACE-complete problems.

Given a Linear Temporal Logic formula ϕ over the atomic propositions AP, the algorithm returns true if ϕ is satisfiable or false otherwise.

Algorithm 2 Linear Time Logic model cheking

```
1: Construct a nondeterministic Büchi automaton $\mathcal{A} = \langle Q, 2^{\mathrm{AP}}, \delta, Q_0, F \rangle$ with $\mathcal{L}_\omega(\mathcal{A}) =$ words($\phi$)
2: if $\mathcal{L}_\omega(\mathcal{A}) = \varnothing$ then
3:     return false
4: end if
5: repeat
6:     Perform a nested depth first search
7:     if there exists a state $q \in F$ reachable from $q_0 \in Q_0$ and that lies on a cycle then
8:         return true
9:     end if
10: until all nodes are explored
11: return false
```

3.5 Computation Tree Logic

In branching-time temporal logics, time is represented as a tree-like structure where each state may have multiple successor states. This framework allows for reasoning about all or some possible futures, making it useful for verifying concurrent systems.

Computation Tree Logic is a branching-time temporal logic used to specify properties of computation trees, which represent all possible executions of a Transition System. Unlike Linear Temporal Logic, which interprets formulas in terms of single execution paths, Computation Tree Logic allows reasoning about multiple possible futures.

3.5.1 Syntax

The syntax of Computation Tree Logic formulas is based on two main type of formulas, based on states and paths.

State formulas In state formulas we have assertions about properties of individual states and their branching structure:

- $E\phi$: there exists a path from the current state along which ϕ holds.
- $A\phi$: for all paths from the current state, ϕ holds.

Therefore, the complete syntax for the state formulas is:

$$\Phi ::= \mathit{true} \mid a \mid \neg\Phi \mid \Phi_1 \cup \Phi_2 \mid E\phi \mid A\phi$$

Here, a represents atomic propositions.

Path formulas In path formulas we describe temporal properties along paths:

- $X\phi$: in the next state along the current path, ϕ holds.
- $F\phi$: there exists a future state along the current path where ϕ holds.
- $G\phi$: ϕ holds in all future states along the current path.

- $\phi_1 \cup \phi_2$: ϕ_2 holds at some future state, and ϕ_1 holds until then.

Therefore, the complete syntax for the path formulas is:

$$\Phi ::= X\Phi \mid F\Phi \mid G\Phi \mid \Phi_1 \cup \Phi_2$$

3.5.2 Semantic

Given a Transition System $\text{TS} = \langle S, \text{Act}, \rightarrow, I, \text{AP}, L \rangle$, we have:

- $s \models a$ if and only if $a \in L(s)$.
- $s \models \neg\Phi$ if and only if $s \not\models \Phi$.
- $s \models \Phi_1 \cup \Phi_2$ if and only if $s \models \Phi_1$ and $s \models \Phi_1$.
- $s \models E\phi$ if and only if there exists a path from s where ϕ holds.
- $s \models A\phi$ if and only if for all paths from s, ϕ holds.

For path formulas, given a path σ (a sequence of states $s_0 s_1 s_2 \ldots$ of a run) in TS:

- $\sigma \models X\Phi$ if and only if $s_1 \models \Phi$.
- $\sigma \models \Phi_1 \cup \Phi_2$ if and only if there exists $j \geq 0$ such that $s_j \models \Phi_2$ and for all $0 \leq i < j$, $s_i \models \Phi_1$.

A Computation Tree Logic formula Φ and an Linear Temporal Logic formula ϕ are equivalent, denoted as $\Phi \equiv \phi$, if they hold for the same Transition Systems:

$$\text{TS} \models \Phi \Leftrightarrow \text{TS} \models \phi$$

Theorem 3.5.1. *Computation Tree Logic and Linear Temporal Logic are incomparable.*

Theorem 3.5.2. *Let Φ be a Computation Tree Logic formula, and let ϕ be the Linear Temporal Logic formula obtained by eliminating all path quantifiers in Φ. Then, one of the following holds:*

- *$\Phi \equiv \phi$, meaning Φ can be fully expressed in Linear Temporal Logic.*
- *There does not exist any Linear Temporal Logic formula that is equivalent to the Computation Tree Logic formula Φ.*

3.5.3 Fairness

airness constraints are essential for verifying concurrent systems. A fair path is one where each fairness condition is satisfied infinitely often. However, Computation Tree Logic alone cannot express fairness conditions directly within its path formulas. To address this limitation, Fair Computation Tree Logic is introduced:

- Fair Computation Tree Logic has the same syntax as Computation Tree Logic but is interpreted over fair paths.
- Fair paths are defined using a Computation Tree Logic formula, interpreted as if it were Linear Temporal Logic.

3.5.4 Model checking

Normal form Similar to Linear Temporal Logic, normal forms can be defined for Computation Tree Logic:

- *Positive normal form*: defined analogously to Linear Temporal Logic.
- *Existential normal form*: a form where only the modalities exists next, exists until, and exists globally appear. In this form, negation cannot be pushed to atomic propositions. However, model-checking algorithms can handle universal operators by duality.

Model checking Given a Transition System TS and a Computation Tree Logic formula Φ, the problem is to determine whether TS $\models \Phi$. The approach follows these steps:

1. For each state subformula ψ of Φ, explore the state space of TS to determine the set of states where ψ holds.
2. Start from atomic propositions (temporal depth 0) and incrementally evaluate subformulas with increasing temporal depth.
3. The formula Φ itself has the highest nesting depth among its subformulas.
4. Assume formulas are given in existential normal form. The only allowed temporal subformulas are: exists until ($E(\phi_1 \cup \phi_2)$), exists next ($EX\phi$), and exists globally ($EG\phi$).

Satisfaction set computation Given a finite Transition System TS with set state S and Computation Tree Logic formula Φ in existential normal form, the algorithm computes $\text{sat}(\Phi) = \{s \in S \mid s \models \Phi\}$.

Algorithm 3 Satisfaction set computation

```
1: repeat
2:     switch $\Phi$ do
3:         case true
4:             return $S$
5:         case $a$
6:             return $\{s \in S \mid a \in L(s)\}$
7:         case $\Phi_1 \wedge \Phi_2$
8:             return $\text{sat}(\Phi_1) \cap \text{sat}(\Phi_2)$
9:         case $\neg\psi$
10:            return $S \setminus \text{sat}(\psi)$
11:        case $\exists \circ \psi$
12:            return $\{s \in S \mid \text{post}(s) \cap \text{sat}(\psi) \neq \varnothing\}$
13:        case $\exists(\Phi_1 \cup \Phi_2)$
14:            $T = \text{sat}(\Phi_2)$
15:            while $s \in \{s \in \text{sat}(\Phi_1) \setminus T \mid \text{post}(s) \cap T \neq \varnothing\} \neq \varnothing$ do
16:                $s = \{s \in \text{sat}(\Phi_1) \setminus T \mid \text{post}(s) \cap T \neq \varnothing\}$
17:                $T = T \cup \{s\}$
18:            end while
19:            return $T$
20:        case $\exists\Box\phi$
21:            $T = \text{sat}(\Phi)$
22:            while $s \in \{s \in T \mid \text{post}(s) \cap T = \varnothing\} \neq \varnothing$ do
23:                $s = \{s \in T \mid \text{post}(s) \cap T = \varnothing\}$
24:                $T = T \setminus \{s\}$
25:            end while
26:            return $T$
27: until all subformulas $\psi$ of $\Phi$ are evaluated
```

For a Transition System TS with N states and K transitions, and Computation Tree Logic formula Φ, the complexity of Computation Tree Logic model checking is $\mathcal{O}((N + K) \cdot |\Phi|)$. his suggests that Computation Tree Logic model checking is computationally more efficient than Linear Temporal Logic model checking. However, there is an important caveat:

- TL formulas can be exponentially more compact than their Computation Tree Logic equivalents (if they exist).
- If $\mathcal{P} \neq \mathcal{NP}$, there exist Linear Temporal Logic formulas ϕ_n of polynomial length n, where any equivalent Computation Tree Logic formula must have non-polynomial length.
- Additionally, for properties expressible in both Computation Tree Logic and Linear Temporal Logic, Linear Temporal Logic model checking can also be made linear in time complexity.

3.6 Symbolic model checking

When dealing with systems with an extremely large number of states, explicitly representing each state becomes impractical. Instead, symbolic model checking reformulates the process by representing entire sets of states and transitions rather than individual elements. This shift allows for more efficient computation, as operations on states are replaced by set operations, which can often be computed much more effectively.

A common approach to symbolic model checking relies on boolean encoding of the state space. Each state is mapped to a binary representation, where a characteristic function defines any subset of states. The transition relation, instead of being stored explicitly, is represented as a boolean function that maps a pair of states to a truth value. By encoding these functions efficiently, the complexity of the model-checking procedure can be significantly reduced.

One of the primary advantages of symbolic model checking is its scalability. Systems with an astronomical number of states can still be analyzed using symbolic techniques. The use of ordered binary decision diagrams and similar structures enables operations on entire sets of states, rather than iterating through them one by one. However, efficiency heavily depends on choosing an appropriate variable ordering, which can greatly impact the size of the ordered binary decision diagram representation. Despite these optimizations, the worst-case complexity of model checking remains unchanged, meaning that certain problems remain computationally difficult even with symbolic methods.

Symbolic model checking has been particularly successful in hardware verification, where state spaces tend to be structured in a way that benefits from ordered binary decision diagram-based representations. In contrast, software verification often favors Bounded Model Checking, which leverages SAT solvers and can sometimes be more effective in handling complex program structures. While symbolic techniques have led to significant advancements, their effectiveness ultimately depends on the nature of the system being analyzed and the efficiency of the underlying boolean function representation.

3.7 Bounded model checking

Bounded Model Checking is a technique used to verify properties of Transition Systems by checking the satisfiability of Boolean formulas. SAT solvers, which work on Boolean formulas in Conjunctive Normal Form, play a crucial role in this approach. Although SAT is an $\mathcal{NP}$-complete problem with no known sub-exponential worst-case algorithm, recent advancements in SAT solvers have led to efficient solutions for many practical cases. These solvers use the standard Dimacs Conjunctive Normal Form format and can routinely handle formulas with tens of thousands of variables and millions of constraints.

Most modern solvers implement sophisticated variations of the Davis-Putnam-Logemann-Loveland algorithm, which is based on backtracking. The Davis-Putnam-Logemann-Loveland algorithm is the foundation of many SAT solvers. Given a formula F in conjunctive normal form a set of true literals T (initially empty), the procedure is as follows:

Algorithm 4 Davis-Putnam-Logemann-Loveland

```
1: function DPLL($F$, $T$)
2:     if $F$ evaluted over $T$ is true then
3:         return true
4:     else if $F$ evaluted over $T$ is false then
5:         return false
6:     else if a clause of $F$ has only one literal $L$ then
7:         return DPLL($F(L = \text{true})$, $T \cup \{L\}$)
8:     else if a literal $L$ appears only as $L = x$ or $L = \neg x$ but not both then
9:         return DPLL($F(L = \text{true})$, $T$)
10:    end if
11:    Choose a literal $L$
12:    return DPLL($F(L = \text{true})$, $T \cup \{L\}$) or DPLL($F(L = \text{false})$, $T \cup \{\neg L\}$)
13: end function
```

Counterexamples A fundamental aspect of Bounded Model Checking is the concept of counterexamples for Linear Temporal Logic properties in a Transition System. Since the number of states in a system is finite, any counterexample must have a finite length. If a cycle occurs within a path, it must do so within a bounded length of at most the number of states plus one. This bound, referred to as the diameter or completeness threshold, allows us to encode the unfolding of the transition relation up to k steps into a Boolean formula. By adding constraints based on the property being verified and cycle detection, we construct a formula ϕ_k such that ϕ_k is satisfiable if and only if there exists a counterexample to the property of length at most k.

3.7.1 Back loops and transition relation

A finite prefix of length k can represent an infinite path if a back loop exists from the last state of the prefix to any of the previous states. If no such back loop is found, the prefix does not provide information about the infinite behavior of the path. The k-times unfolding of a Transition System's transition relation is represented as a propositional formula $|[M]|_k$, where states are encoded as bit vectors. This unfolding captures all finite paths of length k:

$$|[M_S]|_k \leftrightarrow I(S_0) \bigwedge_{0 \leq i \leq k1} T(S_i, S_{i+1})$$

New loop selector variables are introduced to identify possible loops, ensuring that at most one loop exists and that if a loop occurs at position h, then the state atS_{h-1} must be identical to S_k. Additional Boolean variables indicate whether a state is within a loop and whether a loop exists in the structure.

3.7.2 Temporal logic properties

The semantics of a Linear Temporal Logic formula Φ in positive normal form is expressed through Boolean constraints over new formula variables. Each subformula of Φ is associated with a propositional variable at every time step.

Propositional constraint Propositional constraints define how these variables interact with system states. Given a formula ϕ and a index $0 \leq i \leq k$, we have the following:

- $p \rightarrow |[p]|_i \leftrightarrow p \in S_i$
- $\neg p \rightarrow |[\neg p]|_i \leftrightarrow p \notin S_i$
- $\phi_1 \wedge \phi_2 \rightarrow |[\phi_1 \wedge \phi_2]|_i \leftrightarrow |[\phi_1]|_i \wedge |[\phi_2]|_i$
- $\phi_1 \vee \phi_2 \rightarrow |[\phi_1 \vee \phi_2]|_i \leftrightarrow |[\phi_1]|_i \vee |[\phi_2]|_i$

Temporal Operators Temporal operators such as weak until ($\mathcal{W}$) and strong until ($\mathcal{U}$) require additional handling, particularly when eventualities appear inside loops. Special propositional variables track eventualities to ensure that strong properties are correctly encoded. Given a formula ϕ and a index $0 \leq i \leq k$, we have the following:

- $\circ\phi_1 \rightarrow |[\circ\phi_1]|_i \leftrightarrow |[\phi_1]|_{i+1}$
- $\phi_1\mathcal{U}\phi_2 \rightarrow |[\phi_1\mathcal{U}\phi_2]|_i \leftrightarrow |[\phi_2]|_i \vee \left(|[\phi_1]|_i \wedge |[\phi_1\mathcal{U}\phi_2]|_{i+1}\right)$
- $\phi_1\mathcal{R}\phi_2 \rightarrow |[\phi_1\mathcal{R}\phi_2]|_i \leftrightarrow |[\phi_2]|_i \wedge \left(|[\phi_1]|_i \vee |[\phi_1\mathcal{R}\phi_2]|_{i+1}\right)$

Complete encoding The complete encoding Φ_k results in a Boolean formula that logically conjoins all components, including loops, propositional connectives, temporal operators, and eventualities. The formula is evaluated at instant 0, ensuring that the verification process correctly captures the required behaviors.

3.7.3 Model checking

Bounded Model Checking translates the verification problem into a propositional SAT problem, leveraging SAT solvers for efficiency. The transformation procedure is as follows:

1. Unfolding the transition relation: the system's transition relation is unfolded k times, producing a propositional formula $|[M]|_k$.
2. Loop constraints: additional variables and constraints define the presence and position of loops (if any).
3. Encoding Linear Temporal Logic semantics: a Linear Temporal Logic formula F, in release-positive normal form, is translated into Boolean constraints over newly introduced formula variables.
4. For each subformula and each instant $0 \leq i \leq k + 1$, a propositional variable $|[j]|_i$ is introduced.

Then, we follow this procedure:

1. Choose a k value that is large enough but not excessive.
2. Construct the formula Φ_k.
3. Use a SAT solver to check Φ_k.

4. If Φ_k is satisfiable: the solver returns a valid counterexample of length less or equal than k.
5. If Φ_k is unsatisfiable: the property may hold, but longer counterexamples may still exist. Increment k and repeat the process.
6. Completeness threshold: if $k \geq \mathrm{CT}$ and Φ_k remains unsatisfiable, the property P holds. However, CT is often unknown and computationally challenging to determine.

Bounded Model Checking is significantly faster than traditional model checking in finding counterexamples due to its reliance on SAT solvers. It is particularly useful for debugging models early in development. However, an unsatisfiable result for $k < \mathrm{CT}$ does not rule out violations at larger k, necessitating further exploration.

3.8 Probabilistic model checking

Many systems operate in environments influenced by randomness, making it difficult to guarantee absolute correctness. Instead of relying solely on nondeterminism, we often seek probabilistic guarantees. To analyze such scenarios, we extend traditional models to include probabilities, utilizing structures like Markov chains and Markov Decision Processes. This allows for verifying properties such as:

- *Qualitative properties*: ensuring that a good event happens with probability 1, or that a bad event has probability 0.
- *Quantitative properties*: checking if a desired event occurs with at least 95% probability, or if an undesired event happens with less than 5% probability.

3.8.1 Markov chains

Markov chains are widely used to evaluate the performance and reliability of information-processing systems. They extend traditional Transition Systems by associating probabilities with state transitions rather than relying on nondeterministic choices.

Definition (*Discrete time Markov chain*)**.** A discrete time Markov chain $\mathcal{M}$ is defined as a tuple $\mathcal{M} = \langle S, \mathrm{Pr}, \ell_{\mathrm{init}}, \mathrm{AP}, L \rangle$ where:

- S is a countable, nonempty set of states.
- $\mathrm{Pr} : S \times S \to [0, 1]$ defines transition probabilities, ensuring that for every state s: $\sum_{s' \in S} \mathrm{Pr}(s, s') = 1$.
- $\ell_{\mathrm{init}} : S \to [0, 1]$ is the initial probability distribution, such that $\sum_{s \in S} \ell_{\mathrm{init}}(s) = 1$.
- AP is a set of atomic propositions.
- $L : S \to 2^{\mathrm{AP}}$ labels each state with relevant propositions.

Since discrete time Markov chains lack nondeterminism, they cannot model interleaving behavior in concurrent systems.

3.8.1.1 Probabilistic logic for Markov chains

Unlike traditional model-checking techniques, where infinite paths might lead to unrealistic behaviors, probability-based logics help us analyze realistic system behaviors.

Given a linear time logic formula ϕ, the probability of ϕ holding in state s is:

$$\Pr(s \models \phi) = \Pr_s \{\pi \in \text{paths}(s) \mid \pi \models \phi\}$$

Here, $\Pr_s$ is the total probability of all paths starting at s where ϕ holds.

3.8.2 Probabilistic Computation Tree Logic

Probabilistic Computation Tree Logic extends Computation Tree Logic by incorporating probability bounds, allowing for the formal verification of probabilistic systems.

3.8.2.1 Syntax

The syntax of Probabilistic Computation Tree Logic consists of state and path formulas. State formulas describe properties of individual states:

$$\Phi ::= \text{true} \mid a \mid \Phi_1 \wedge \Phi_2 \mid \neg\Phi \mid \mathbb{P}_J(\phi)$$

Here, $a \in \mathit{atomicproposition}$ is an atomic proposition, ϕ is a path formula, and $J \subseteq [0,1]$ is a probability interval.

Path formulas define properties over execution paths:

$$\phi ::= \circ\Phi \mid \Phi_1 \cup \Phi_2 \mid \Phi_1 \cup^{\leq n} \Phi_2$$

Here, $\circ$ represents the next operator, $\cup$ denotes the until operator, and $\cup^{\leq n}$ expresses bounded until with a maximum of n steps.

3.8.2.2 Semantic

Probabilistic Computation Tree Logic is interpreted over a Markov chain, where the semantics of the non-probabilistic fragment follow those of Computation Tree Logic. The probability operator is defined as:

$$s \models \mathbb{P}_j(\phi) \Leftrightarrow \Pr(s \models \phi) \in J$$

Here, $\Pr(s \models \phi)$ represents the probability that paths originating from state s satisfy ϕ. The following rules define how Probabilistic Computation Tree Logic formulas are evaluated:

- $s \models a \Leftrightarrow a \in L(s)$
- $s \models \neg\Phi \Leftrightarrow s \not\models \Phi$
- $s \models \Phi \wedge \psi \Leftrightarrow s \models \Phi \wedge s \models \psi$
- $\pi \models \circ\Phi \Leftrightarrow \pi[1] \models \Phi$
- $\pi \models \Phi \cup \psi \Leftrightarrow \exists j \geq 0 \quad (\pi[j] \models \psi \wedge (\forall 0 \leq k < j.\pi[k] \models \Phi))$
- $\pi \models \Phi \cup^{\leq n} \psi \Leftrightarrow \exists 0 \leq j \leq n \quad (\pi[j] \models \psi \wedge (\forall 0 \leq k < j.\pi[k] \models \Phi))$

Here, for a path $\pi = s_0 s_1 s_2 \ldots$ and $\pi[i]$ denotes the $(i+1)$-th state of π.

3.8.2.3 Model checking

The Probabilistic Computation Tree Logic model checking problem involves determining if a given state in a Markov chain satisfies a Probabilistic Computation Tree Logic formula. Given a finite Markov chain $\mathcal{M}$, a state s in $\mathcal{M}$, and a Probabilistic Computation Tree Logic state formula Φ, the problem is to determine whether $s \models \Phi$. This is achieved by computing the satisfaction set sat(Φ) using a bottom-up traversal of the formula's parse tree.

Theorem 3.8.1. *For finite Markov chain $\mathcal{M}$ and Probabilistic Computation Tree Logic formula Φ, the model checking problem $\mathcal{M} \models \Phi$ can be solved in time:*

$$\mathcal{O}\left(poly\left(size\left(\mathcal{M}\right) \cdot n_{\max} \cdot |\Phi|\right)\right)$$

Here, $n_{\max}$ is the maximal step bound appearing in a bounded until subformula $\Psi_1 \cup^{\leq n} \Psi_2$ of Φ, and $n_{\max} = 1$ if Φ contains no bounded until operators.

Restricting probability bounds to greater than zero, equal to one, equal to zero or less than one results in a qualitative fragment of Probabilistic Computation Tree Logic, which allows reasoning without explicit probability values and improves model checking efficiency.

Property 3.8.1. There is no Computation Tree Logic formula that is equivalent to $\mathbb{P}_{=1}(\Diamond a)$.

Property 3.8.2. There is no Computation Tree Logic formula that is equivalent to $\mathbb{P}_{>0}(\Box a)$.

Property 3.8.3. There is no qualitative Probabilistic Computation Tree Logic formula that is equivalent to $\forall\Diamond a$.

Property 3.8.4. There is no qualitative Probabilistic Computation Tree Logic formula that is equivalent to $\exists\Box a$.

3.8.3 Markov decision processes

A Markov Decision Process extends Markov chains by introducing nondeterminism, making them useful for modeling concurrent systems.

Theorem 3.8.2. *For a finite Markov Decision Process $\mathcal{M}$ and a Probabilistic Computation Tree Logic formula Φ, the model checking problem $\mathcal{M} \models \Phi$ can be solved in time:*

$$\mathcal{O}\left(poly\left(size\left(\mathcal{M}\right) \cdot n_{\max} \cdot |\Phi|\right)\right)$$

Here, $n_{\max}$ is the maximal step bound appearing in a bounded until subformula $\Psi_1 \cup^{\leq n} \Psi_2$ of Φ, and $n_{\max} = 1$ if Φ contains no bounded until operators.

CHAPTER 4

Properties

4.1 Equivalence

In the context of Transition Systems, trace equivalence and bisimulation are two ways to compare systems based on their behaviors and properties.

4.1.1 Trace equivalence

Two Transition Systems, TS_1 and TS_2, are considered trace equivalent with respect to a set of atomic propositions if they generate the same set of traces over AP. This is denoted as:

$$\mathrm{traces}_{\mathrm{AP}}(\mathrm{TS}_1) = \mathrm{traces}_{\mathrm{AP}}(\mathrm{TS}_2)$$

In simple terms, two systems are trace equivalent if they can produce the same sequences of observable actions (traces), even if their internal structures differ.

Theorem 4.1.1. *For a linear-time property P over AP, If $traces_{AP}(TS_1) \subseteq traces_{AP}(TS_2)$, then:*

$$TS_1 \models P \implies TS_2 \models P$$

Corollary 4.1.1.1. *Trace-equivalent Transition Systems satisfy the same linear-time properties.*

Trace equivalence may not always be sufficient for reactive or concurrent systems, as it ignores the internal branching structure of the system. While this is useful for parsers and compilers where we care about the observable behavior (language equivalence), reactive systems may require a more nuanced comparison.

4.1.2 Bisimulation

Bisimulation provides a more detailed notion of equivalence, focusing on the internal structure of the systems. A Transition System TS_2 is said to simulate TS_1 if every transition in TS_1 can be matched by one or more transitions in TS_2.

Definition (*Bisimulation equivalent*)**.** Let $\mathrm{TS}_1 = \langle S_1, \mathrm{Act}_1, \rightarrow_1, I_1, \mathrm{AP}, L_1 \rangle$ and $\mathrm{TS}_2 = \langle S_2, \mathrm{Act}_2, \rightarrow_2, I_2, \mathrm{AP}, L_2 \rangle$ be two Transition Systems. A bisimulation between them is a binary relation $\mathcal{R} \subseteq S_1 \times S_2$ satisfying:

1. For each initial state $s_1 \in I_1$, there exists an initial state $s_2 \in I_2$ such that $(s_1, s_2) \in \mathcal{R}$, and viceversa.
2. For all pairs $(s_1, s_2) \in \mathcal{R}$, the following conditions hold:
   (a) $L_1(s_1) = L_2(s_2)$ (the labeling of the states must be identical).
   (b) If $s_1' \in \text{post}(s_1)$, then there exists $s_2' \in \text{post}(s_2)$ such that $(s_1', s_2\prime) \in \mathcal{R}$.
   (c) If $s_2' \in \text{post}(s_2)$, then there exists $s_1' \in \text{post}(s_1)$ such that $(s_1', s_2\prime) \in \mathcal{R}$.

If such a bisimulation exists, we say that TS_1 and TS_2 are bisimulation equivalent ($\text{TS}_1 \sim \text{TS}_2$).

Properties The key properties of bisimulation equivalence are:

- *Bisimulation implies trace equivalence*: if two systems are bisimulation equivalent, they are also trace equivalent.
- *Symmetry, transitivity, and reflexivity*: the bisimulation relation $\sim$ is an equivalence relation, meaning it satisfies these properties.
- *Minimization*: bisimulation allows us to minimize the number of states needed to represent a system while preserving its behavior. If $\text{TS}_1 \sim \text{TS}_2$, and TS_2 is smaller than TS_1, we can verify properties on the reduced system TS_2 rather than the larger system TS_1, which can be especially useful for infinite-state systems.

Bisimulation relation Bisimulation can be considered as a relation between states of a Transition System. The goal is to minimize the number of states required to prove a certain property.

Definition (*Bisimulation equivalent*)**.** Let $\text{TS} = \langle S, \text{Act}, \rightarrow, I, \text{AP}, L\rangle$ be a Transition System. A bisimulation for TS is a binary relation on $\mathcal{R} \subseteq S \times S$ such that:

1. $L_1(s_1) = L_2(s_2)$ for all $(\text{TS}_1, \text{TS}_2) \in \mathcal{R}$:
2. If $s_1' \in \text{post}(s_1)$, then there exists $s_2' \in \text{post}(s_2)$ such that $(s_1', s_2\prime) \in \mathcal{R}$.
3. If $s_2' \in \text{post}(s_2)$, then there exists $s_1' \in \text{post}(s_1)$ such that $(s_1', s_2\prime) \in \mathcal{R}$.

States s_1 and s_2 are said to be bisimulation equivalent (denoted$s_1 \sim_{\text{TS}} s_2$) if there exists a bisimulation $\mathcal{R}$ for TS such that $(\text{TS}_1, \text{TS}_2) \in \mathcal{R}$.

The properties of the bisimulation relation are:

1. $\sim_{\text{TS}}$ is an equivalence relation on the set of states S.
2. $\sim_{\text{TS}}$ is a bisimulation for the Transition System TS.
3. $\sim_{\text{TS}}$ is the coarsest bisimulation for TS.

4.1.3 Quotient Transition System

A quotient Transition System is defined using an equivalence relation. Specifically, if we have a bisimulation relation $\sim_{\text{TS}}$ on a Transition System TS, we can define a quotient system $\text{TS}\backslash\sim_{\text{TS}}$, which groups states that are bisimulation equivalent into equivalence classes.

Definition (*Quotient transition*)**.** For Transition System $\text{TS} = \langle S, \text{Act}, \rightarrow, I, \text{AP}, L\rangle$ and a bisimulation $\sim_{\text{TS}}$, the quotient Transition System $\text{TS}\backslash\sim_{\text{TS}}$ is defined as:

$$\text{TS}\backslash\sim_{\text{TS}} = \langle S\backslash\sim_{\text{TS}}, \{\tau\}, \rightarrow', I', \text{AP}, L'\rangle$$

Here, $I' = \{[s]_\sim \mid s \in I\}$ is the set of equivalence classes of initial state, $\rightarrow'$ is defined by $\frac{s \xrightarrow{\alpha} s'}{[s]_\sim \xrightarrow{\tau} [s']_\sim}$, and $L'([s]_\sim) = L(s)$.

Theorem 4.1.2. *For any Transition System TS, it holds that $TS \sim TS\backslash\sim$.*

This means that we can prove properties in the quotient system rather than the original one.

4.1.4 Computation Tree Logic equivalence

Computation Tree Logic equivalence is a way to compare Transition Systems based on the set of Computation Tree Logic formulas they satisfy. States in a Transition System are Computation Tree Logic-equivalent if they satisfy the same Computation Tree Logic formulas.

Definition (*Computation Tree Logic state equivalence*)**.** States s_1 and s_2 in a Transition System TS Computation Tree Logic-equivalent, denoted $s_1 \equiv_{\text{CTL}} s_2$, if for all Computation Tree Logic formulas ϕ over AP, $s_1 \models \phi$ if and only if $s_2 \models \phi$.

Definition (*Computation Tree Logic system equivalence*)**.** Transition systems TS_1 and TS_2 are Computation Tree Logic-equivalent, denoted $s_1 \equiv_{\text{CTL}} s_2$, if for all Computation Tree Logic formulas ϕ, $\text{TS}_1 \models \phi$ if and only if $\text{TS}_2 \models \phi$.

For finite Transition Systems without terminal states, bisimulation equivalence and Computation Tree Logic equivalence are equivalent:

$$\sim_{\text{TS}} = \equiv_{\text{CTL}}$$

For infinite-state systems, bisimulation equivalence implies Computation Tree Logic equivalence, allowing us to prove Computation Tree Logic properties on the quotient system.

Theorem 4.1.3. *The bisimulation quotient of a finite Transition System TS can be computed in time:*

$$\mathcal{O}\left(|S| \cdot |AP| + M \log |S|\right)$$

Here, M denotes the number of edges in the state graph.

4.2 Transition systems refinement

Transition systems can represent software or hardware at various levels of abstraction, where the level of detail included in the model depends on the purpose of the analysis. At lower levels of abstraction, more implementation details are included, while higher levels intentionally omit those details, focusing on the core behaviors of the system. This section discusses the processes of abstraction and refinement in Transition Systems and their role in preserving certain properties while simplifying or detailing the model.

The process of abstraction involves starting with a detailed Transition System and creating a more abstract version, which may be easier to manage or reason about, while still preserving key properties of interest. This is often necessary for high-level analysis or verification tasks. On the other hand, refinement is the opposite process, where a more abstract model is made more detailed to reflect implementation specifics, without losing the essential properties.

Both abstraction and refinement are concerned with maintaining the correctness of a system while adjusting the level of detail, and this can be formally described using implementation relations. These relations connect two Transition Systems at different levels of abstraction.

Definition (*Implementation relation*)**.** An implementation relation is a binary relation between two Transition Systems at different abstraction levels.

When two Transition Systems, TS_1 and TS_2, are related by an implementation relation, one system is said to be refined by the other. The system that is more abstract is seen as an abstraction of the more detailed system. If the implementation relation is an equivalence, then the two systems are indistinguishable at the relevant level of abstraction, as they exhibit the same observable properties.

The set of atomic propositions plays a critical role in comparing Transition Systems. In bisimulation, AP represents the relevant atomic propositions. Other propositions can be ignored if they do not affect the properties being studied.

When verifying the satisfaction of a temporal logic formula ϕ, we can restrict the atomic proposition to those propositions that are relevant to the formula. This reduces the scope of the comparison and allows the verification of the formula in a more manageable way.

4.2.1 Simulation

While bisimulation relations require that two states exhibit identical stepwise behavior, simulation relations relax this requirement. A simulation relation only mandates that if a state s' simulates state s, then s' must mimic all transitions of s, but not necessarily vice versa. In other words, for each successor of s, there must be a corresponding successor of s', but not all successors of s' need to correspond to those of s.

Simulation relations are useful when we want to show that one system correctly implements another, more abstract system, or when we want to find a smaller abstract model that still preserves important properties.

Definition (*Simulation*)**.** Let $\mathrm{TS}_1 = \langle S_1, \mathrm{Act}_1, \rightarrow_1, I_1, \mathrm{AP}, L_1\rangle$ and
$\mathrm{TS}_2 = \langle S_2, \mathrm{Act}_2, \rightarrow_2, I_2, \mathrm{AP}, L_2\rangle$ be Transition Systems over AP. A simulation for $(\mathrm{TS}_1, \mathrm{TS}_2)$ is a binary relation $\mathcal{R} \subseteq S_1 \times S_2$ such that:

1. For every initial state $\forall s_1 \in I_1$, $s_2 \in I_2$ such that $(s_1, s_2) \in \mathcal{R}$.
2. For all $(s_1, s_2) \in \mathcal{R}$, the following hold:
   (a) $L_1(s_1) = L_2(s_2)$
   (b) If $s_1' \in \mathrm{post}(s_1)$, then there exists $s_2' \in \mathrm{post}(s_2)$ such that $(s_1', s_2\prime) \in \mathcal{R}$.

We say that TS_1 is simulated by TS_2, denoted $\mathrm{TS}_1 \preceq \mathrm{TS}_2$, if there exists a bisimulation $\mathcal{R}$ for $(\mathrm{TS}_1, \mathrm{TS}_2)$.

The first condition ensures that every initial state in TS_1 is related to an initial state in TS_2, although there might be initial states in TS_2 that are not related to any initial state in TS_1. The second condition is similar to that of bisimulation, but the symmetric counterpart of the condition is not required. In other words, while TS_1 may have some successors not matched by TS_2, every successor of TS_1 must have a corresponding successor in TS_2.

4.2.2 Abstraction

Definition (*Refinement*)**.** If TS_1 is obtained from TS_2 by deleting transitions, then TS_1 is simulated by TS_2. This makes TS_1 a refinement of TS_2, as the nondeterminism in TS_2 is resolved in TS_1.

Definition (*Abstraction*)**.** Conversely, if TS_2 is obtained from TS_1 by some form of abstraction, then TS_1 is simulated by TS_2. In this case, TS_2 abstracts away some of the details of TS_1.

An abstraction of TS_1 to TS_2 requires:

- A common set of atomic propositions.
- The states of TS_2 are "abstract states".
- An abstraction function f that associates each concrete state s of TS_1 with the abstract state $f(s)$ of TS_2, respecting the labels in AP.

Abstractions may vary in terms of the choice of abstract states, the abstraction function f, and the relevant propositions in AP.

4.2.3 Safety property

A safety property is one that asserts that something bad will never happen. A key property of simulation relations is that they preserve safety properties.

Property 4.2.1. Let P_{safe} be a safety linear-time property, and let TS_1 and TS_2 be Transition Systems. Then:

$$\mathrm{TS}_1 \preceq \mathrm{TS}_2 \wedge \mathrm{TS}_2 \models P_{\text{safe}} \implies \mathrm{TS}_1 \models P_{\text{safe}}$$

This means that if TS_2 satisfies a safety property and TS_1 is simulated by TS_2, then TS_1 will also satisfy the safety property.

4.2.4 Path fragments

Definition. A path fragment is a finite part of a path in the Transition System.

The simulation relation preserves the set of finite path fragments, but it does not preserve entire paths that end in terminal states. If a path fragment p_1 from state s_1 is simulated by a path fragment p_2 from state s_2, and p_1 ends in a terminal state, but p_2 does not, then: p_1 is a valid path in TS_1, but p_2 is not a valid path in TS_2.

Simulation preserves the finite traces of the system, which correspond to the finite path fragments, but it does not preserve traces that involve terminal states. If there are no terminal states in the Transition System, simulation preserves all traces (including infinite traces) and not just the safety properties.

4.2.5 Simulation quotient

A simulation quotient is a reduced version of a Transition System where simulation equivalence is used to merge equivalent states. Given a Transition System TS and a simulation equivalence $\cong_{\text{TS}}$, we can define a quotient Transition System $TS_{\setminus\cong}$, where equivalent states are merged. The quotient system will have the same behavior as TS with reduced state space.

4.2.6 Atomic proposition determinism

Definition (*Atomic proposition determinism*)**.** An Atomic proposition deterministic Transition System $\text{TS} = \langle S, \text{Act}, \rightarrow, I, \text{AP}, L \rangle$ has the following properties:

1. For any set $A \subseteq \text{AP}$, there is at most one initial state in TS with label A.
2. If a state s can transition to two distinct states s' and s'' via the same action and with the same label, then $s' = s''$.

Theorem 4.2.1. *IF TS_1 and TS_2 are atomic proposition-determinstic, then $TS_1 \sim TS_2$ if and only if $TS_1 \simeq TS_2$.*

This means that for atomic proposition-deterministic systems, simulation equivalence implies bisimulation equivalence, and vice versa.

4.2.7 Action based bisimulation

While traditional bisimulation focuses on state labels, action-based bisimulation also takes into account the actions triggering transitions between states.

Definition (*Action based bisimulation equivalence*)**.** Let $\text{TS}_1 = \langle S_1, \text{Act}_1, \rightarrow_1, I_1, \text{AP}, L_1 \rangle$ and $\text{TS}_2 = \langle S_2, \text{Act}_2, \rightarrow_2, I_2, \text{AP}, L_2 \rangle$ be Transition Systems over a set of actions Act. An action-based bisimulation for $(\text{TS}_1, \text{TS}_2)$ is a binary relation $\mathcal{R} \subseteq S_1 \times S_2$ such that:

1. For each $s_1 \in I_1$, there exists $s_2 \in I_2$ such that $(s_1, s_2) \in \mathcal{R}$, and viceversa.
2. For each pair (s_1, s_2), the following hold:
   (a) If $s_1 \xrightarrow{\alpha} s_1'$, then $s_2 \xrightarrow{\alpha} s_2'$ with $(s_1', s_2') \in \mathcal{R}$ for some $s_2' \in S_2$.
   (b) If $s_2 \xrightarrow{\alpha} s_2'$, then $s_1 \xrightarrow{\alpha} s_1'$ with $(s_1', s_2') \in \mathcal{R}$ for some $s_1' \in S_1$.

TS_1 and TS_2 are action-based bisimulation equivalent, denoted $\text{TS}_1 \sim^{\text{Act}} \text{TS}_2$, if there exists an action-based bisimulation $\mathcal{R}$ for $(\text{TS}_1, \text{TS}_2)$.

CHAPTER 5

Timed automata and languages

5.1 Introduction

In many systems, timing is crucial, and they must operate within strict time constraints. These time-critical systems require careful handling of time to ensure proper functionality.

In classical Finite State Machines, time is often not explicitly modeled. Time is treated as a discrete, implicit metric, with an infinite number of states under a Büchi acceptance condition. This means time is ignored in the traditional sense, where each transition represents one time step, assuming a fixed underlying clock rate.

However, this approach may not be sufficient in more complex systems. When Finite State Machines are composed to form more intricate models, the temporal relationships between events depend heavily on the method of composition. In complex system models, where multiple Finite State Machines are often composed, the rigid notion of one transition is one time step becomes inadequate, especially when some components operate asynchronously.

To address this, we need more advanced mechanisms to capture the progression of time, especially when dealing with asynchronous components. Several alternatives to the traditional Finite State Machine model can better handle time in complex systems:

1. *Transition-based time advancement*: in some models, time advances only when specific transitions are taken or when certain conditions are met (statecharts). This approach remains fundamentally transition-based and is effective for discrete-time systems.

2. *Independent time progression*: in models like Timed and Hybrid Automata, time can advance independently of transitions. Here, transitions may include time-based guards, allowing them to be triggered only when certain timing conditions are met. This allows for the modeling of continuous or dense time, capturing a broader range of behaviors in time-critical systems.

5.2 Timed Automata

Timed Automata are a type of Program Graph where real-valued variables, called clocks, are added to track time. The state of a Timed Automaton is defined by a combination of its current location (the logical state of the automaton) and the values of the clocks. The set of locations in a Timed Automata is finite, but the clocks take values in non-negative real numbers, making the state space infinite. Timed Automatas often do not use atomic propositions but rely on the names of the locations to define the states.

A Timed Automaton can evolve in two main ways:

- *Location jumps*: the automaton can change its location without the passage of time. This happens when a transition occurs, but time does not elapse.
- *Time elapsing*: time progresses while the automaton remains in the same location. This means that the location remains unchanged, but the values of the clocks are updated.

The clocks in a Timed Automaton measure the passage of time according to the differential equation $\dot{x} = 1$, meaning that the value of each clock increases at a constant rate.

5.2.1 Clocks

In Timed Automata, clocks play a crucial role in tracking the passage of time. However, access to clocks is limited: they can only be inspected (checked) or reset to zero during transitions.

Unless explicitly reset, each clock continuously increments as time progresses, at a rate of 1. After d time units have passed, all clocks advance by d. The value of a clock represents the amount of time that has elapsed since its last reset. Clocks can be thought of as independent stopwatches, which can be started and checked independently of one another.

Clock constraints Clock constraints (conditions on the values of the clocks) are used to define enabling conditions, or guards, for transitions. A transition will only be enabled and capable of occurring if the clock constraint is satisfied. If the condition is not met, the transition (and its associated action) is disabled.

Definition (*Clock constraint*)**.** A clock constraint over a set C of clocks is formed according to the following grammar:

$$g ::= x < c \mid x \leq c \mid x > c \mid x \geq c \mid g \wedge g$$

Here, $x \in C$ represents a clock, and $c \in \mathcal{N}$ represents a constant. Let $\mathrm{cc}(C)$ denote the set of clock constraints over C.

A Timed Automaton is a Program Graph with a finite set C of clocks. The edges of this graph are labeled with the tuples $\langle g, \alpha, D\rangle$, where:

- g is the clock constraint (the guard).
- α is an action performed during the transition.
- $D \subseteq C$ is the set of clocks that will be reset to zero during the transition.

The intuitive interpretation of the transition:

$$\ell \overset{g:\alpha,D}{\hookrightarrow} \ell'$$

is that the automaton can move from location ℓ to location ℓ' if the clock constraint g holds. During this transition, the clocks in D are reset to zero, and the action α is executed.

Definition (*Invariant*)**.** A location invariant is a clock constraint associated with a location ℓ.

he invariant specifies the maximum amount of time that can be spent in location ℓ before the system must transition to another location.

5.2.2 Transition systems as Timed Automata

Definition. A Timed Automaton TA can be formally defined as a tuple
$\mathrm{TA} = \langle \mathrm{Loc}, \mathrm{Act}, C, \rightarrow, \mathrm{Loc}_0, \mathrm{Inv}, \mathrm{AP}, L\rangle$, where:

- Loc: a finite set of locations (states).
- $\mathrm{Loc}_0 \subseteq \mathrm{Loc}$: a set of initial locations.
- Act: a finite set of actions.
- C: a finite set of clocks.
- $\rightarrow \subseteq \mathrm{Loc} \times \mathrm{cc}(C) \times \mathrm{Act} \times 2^C \times \mathrm{Loc}$: the transition relation, specifying allowed state changes.
- $\mathrm{Inv} : \mathrm{Loc} \rightarrow \mathrm{cc}(C)$: a function assigning clock constraints (invariants) to locations.
- AP: a finite set of atomic propositions.
- $L : \mathrm{Loc} \rightarrow 2^{\mathrm{AP}}$: a labeling function that maps each location to a set of atomic propositions.

Given two timed automata $\mathrm{TA}_1 = \langle \mathrm{Loc}_1, \mathrm{Act}_1, C_1, \rightarrow_1, \mathrm{Loc}_{0,1}, \mathrm{Inv}_1, \mathrm{AP}_1, L_1\rangle$ and $\mathrm{TA}_2 = \langle \mathrm{Loc}_2, \mathrm{Act}_2, C_2, \rightarrow_2, \mathrm{Loc}_{0,2}, \mathrm{Inv}_2, \mathrm{AP}_2, L_2\rangle$ where $H \subseteq \mathrm{Act}_1 \cap \mathrm{Act}_2$, $C_1 \cap C_2 = \varnothing$ and $\mathrm{AP}_1 \cap \mathrm{AP}_2 = \varnothing$. The parallel composition $\mathrm{TA}_1 \parallel_H \mathrm{TA}_2$ is defined as:

$$\langle \mathrm{Loc}_1 \times \mathrm{Loc}_2, \mathrm{Act}_1 \cup \mathrm{Act}_2, C_1 \cup C_2, \hookrightarrow, \mathrm{Loc}_{0,1} \times \mathrm{Loc}_{0,2}, \mathrm{Inv}, \mathrm{AP}_1 \cup \mathrm{AP}_2, L\rangle$$

Here, $L(\langle \ell_1, \ell_2\rangle) = L(\ell_1) \cup L(\ell_2)$ and $\mathrm{Inv}(\langle \ell_1, \ell_2\rangle) = \mathrm{Inv}(\ell_1) \wedge \mathrm{Inv}(\ell_2)$. The transition relation $\hookrightarrow$ is defined based on synchronization conditions:

- For $\alpha \in H$ (synchronized transition):

$$\frac{\ell_1 \overset{g_1:\alpha,D_1}{\hookrightarrow}_1 \ell_1' \wedge \ell_2 \overset{g_2:\alpha,D_2}{\hookrightarrow}_2 \ell_2'}{\langle \ell_1, \ell_2\rangle \overset{g_1 \wedge g_2:\alpha,D_1 \cup D_2}{\hookrightarrow} \langle \ell_1', \ell_2'\rangle}$$

- For $\alpha \notin H$:

$$\frac{\ell_1 \overset{g:\alpha,D}{\hookrightarrow}_1 \ell_1'}{\langle \ell_1, \ell_2\rangle \overset{g:\alpha,D}{\hookrightarrow} \langle \ell_1', \ell_2'\rangle} \quad \text{and} \quad \frac{\ell_2 \overset{g:\alpha,D}{\hookrightarrow}_2 \ell_2'}{\langle \ell_1, \ell_2\rangle \overset{g:\alpha,D}{\hookrightarrow} \langle \ell_1', \ell_2'\rangle}$$

5.2.3 Clock valuation

A clock valuation is a function:

$$\eta : C \rightarrow \mathcal{R}_{\geq 0}$$

hat assigns a nonnegative real value to each clock $x \in C$. This function represents the amount of time elapsed since the last reset of each clock. The set of all possible clock valuations over C is denoted as $\mathrm{eval}(C)$. A clock valuation η satisfies a clock constraint $g \in \mathrm{cc}$ if and only if the constraint evaluates to true when the clocks take values according to η.

Definition (*Satisfaction relation for clock constraints*)**.** For set C of clocks, $x \in C$, $\eta \in \mathrm{eval}(C)$, $c \in \mathbb{N}$, and $g, g' \in \mathrm{cc}(C)$, let $\models \subseteq \mathrm{eval}(C) \times \mathrm{cc}(C)$ be defined by:

- $\eta \models \mathrm{true}$

- $\eta \models x < c$ if and only if $\eta(x) < c$
- $\eta \models x \leq c$ if and only if $\eta(x) \leq c$
- $\eta \models \neg g$ if and only if $\eta \not\models g$
- $\eta \models g \wedge g'$ if and only if $\eta \models g \wedge \eta \models g'$

For any positive real number d, the notation $\eta + d$ represents a new clock valuation where all clocks in η have their values increased by d.

Definition (*Reset*)**.** A reset operation on a clock x in a clock valuation η results in a new clock valuation where all clocks remain unchanged except for x, which is reset to zero.

5.2.4 Transitions

The behavior of a timed automaton is defined in terms of an underlying Transition System. In this model, a system can evolve in two fundamental ways:

- Taking a discrete transition: moving from one location to another based on clock constraints and reset conditions.
- Letting time progress: remaining in the same location while clocks continue to increase.

Discrete transitions correspond to instantaneous events, meaning they occur with zero duration. Time progression occurs only while staying in a location. Multiple actions can take place at the same time instant since transitions are instantaneous.

Definition (*Timed automaton transition*)**.** A Timed Automaton is defined as
$\text{TA} = \langle \text{Loc}, \text{Act}, C, \rightarrow, \text{Loc}_0, \text{Inv}, \text{AP}, L \rangle$, where:

- Loc is the set of locations (states).
- Act is the set of possible actions.
- C is the set of clocks.
- $\rightarrow$ is the transition relation.
- Loc_0 is the set of initial locations.
- Inv assigns invariants to locations.
- AP is the set of atomic propositions.
- L is a labeling function.

The transition relation defines two types of transitions.

Discrete transition A discrete transition:

$$\langle \ell, \eta \rangle \overset{\alpha}{\hookrightarrow} \langle \ell', \eta' \rangle$$

Occurs if the following conditions hold:

1. There exists a transition $\ell \overset{g:\alpha,D}{\hookrightarrow} \ell'$ in TA.
2. The current clock valuation satisfies the guard: $\eta \models g$.
3. The new clock valuation η' is obtained by resetting all clocks in D
$$\eta' = \text{reset } D \in \eta$$
4. The new clock valuation satisfies the invariant of the destination location:
$$\eta' \models \text{inv}(\ell')$$

Delay transition A delay transition:

$$\langle \ell, \eta \rangle \overset{d}{\hookrightarrow} \langle \ell, \eta + d \rangle$$

For any $d \in \mathbb{R}_{\geq 0}$ is valid if and only if $\eta + d \models \text{inv}(\ell)$, which ensures that time can progress while remaining in ℓ.

5.2.5 Definitions and diagonal constraints

Definition (*Time converfet path*)**.** A time-convergent path is a sequence of transitions where time does not progress beyond a certain bound.

A time-divergent path ensures that time progresses indefinitely toward infinity.

Definition (*Timelock*)**.** A timelock occurs when a state s in TS(TA) has no time-divergent path starting from it.

This means time cannot progress indefinitely from s, making it unrealistic in real-world systems. Avoiding timelocks is essential when designing time-critical systems, ensuring that time always has a way to progress.

Definition (*Zeno path*)**.** A path π in TS(TA) is called Zeno if:

- It is time-convergent (time does not diverge to infinity).
- It contains infinitely many discrete transitions.

Zeno paths are problematic because they imply that an infinite number of actions happen in finite time, which would require infinitely fast processors, making such behavior unrealizable in practice.

Diagonal constraints Diagonal constraints extend standard clock constraints by allowing direct comparisons between two clocks. These constraints require tracking relationships between clocks rather than absolute values. To model these constraints without changing the fundamental structure of timed automata by storing extra information in states about the last reset time of each clock and using this information to encode constraints.

5.3 Timed Automata analysis

To verify properties of a system modeled as a Timed Automaton, we need to explore all possible states of the automaton.

A state in a Timed Automata is represented as $\langle \text{location}, \text{clock evaluation} \rangle$. The set of possible clock evaluations, denoted $\text{eval}(C)$, consists of real-valued clock assignments. Since clock values come from $\mathbb{R}_{\geq 0}$, this results in an infinite state space.

5.3.1 Clock regions

To make exhaustive exploration feasible, we construct a finite abstraction of the state space: clock regions provide a way to group equivalent clock valuations into a finite number of equivalence classes. Two clock valuations η_1 and η_2 belong to the same region if and only if:

1. They agree on the integer parts of all clock values. For each $x \in C$, the integer parts of $\eta_1(x)$ and $\eta_2(x)$ must be identical.
2. They agree on the ordering of fractional parts of all clock values. If two clocks x and y in η_1 will reach their next integer values in a specific order, they must reach them in the same order in η_2.

For each clock $x \in C$, define x_c as the largest integer appearing in constraints: if a constraint $x \leq c$ or $c \leq x$ appears in the system, set $c_x = \max(c)$.

Definition (*Clock valuation equivalence*)**.** Clock valuations η and η' are equivalent (i.e., belong to the same clock region) if and only if:

- If $\eta(x) > c_x$ and $\eta'(x) > c_x$ for every clock x, then they are considered equivalent.
- Otherwise, for all $x, y \in C$ with $\eta(x), \eta'(x) \leq c_x$ and $\eta(y), \eta'(y) \leq c_y$:
  - They have the same integer part:
  $$\lfloor \eta(x) \rfloor = \lfloor \eta'(x) \rfloor$$
  - Their fractional parts agree in ordering:
  $$\text{frac}(\eta(x)) \leq \text{frac}(\eta(y)) \Leftrightarrow \text{frac}(\eta'(x)) \leq \text{frac}(\eta'(y))$$
  - $\text{frac}(\eta(x)) = 0$ if and only if $\text{frac}(\eta'(x)) = 0$

Definition (*Clock region*)**.** A clock region is defined as an equivalence class of clock evaluations under the relation $\cong$.

Since there are only finitely many ways to assign integer values and compare fractional parts, the number of regions is finite. The region abstraction enables efficient verification of properties in timed automata by ensuring a finite number of explored states.

5.3.2 Region based automata

By using clock regions, we can construct a finite representation of all possible states in the timed Transition System TS(TA). state in TS(TA) is represented as $\langle \ell_i, \rangle$ where $\ll_i$ is a location and η_i is a clock valuation. The key idea is to group clock valuations into equivalence classes (regions) to define a finite region-based Transition System. Since there is a maximum constant in the clock constraints, the number of clock regions is finite.

Thus, for a given Timed Automaton, we build a region-based Transition System where states are pairs:

$$\langle \ell_i, \rangle$$

To define transitions in the region-based Transition System, we need to determine time successors of clock regions.

Definition (*Time successor*)**.** A clock region α' is a time-successor of a clock region α if:

$$\forall \eta \in \alpha \qquad \exists t \in \mathbb{R}_{\geq 0} \mid \eta + t \in \alpha'$$

This means that every reachable clock region along a diagonal transition is a time-successor. If two clock valuations are in the same region, the automaton can take the same transitions. However, the exact delays might need to be adjusted to ensure they reach corresponding successor regions.

5.3.3 Time abstracted bisimulation

Clock equivalence can be seen as a special case of time-abstracted bisimulation.

Definition (*Time abstracted bisimulation*)**.** A relation R on the states of a timed automata is a time based bisimulation if $(\ell_1, \upsilon_1)R(\ell_2, \upsilon_2)$ and $(\ell_1, \upsilon_1) \xrightarrow{d_1,a} (\ell_1', \upsilon_1')$ for some $d_1 \in \mathbb{R}_{\geq 0}$ and $a \in \Sigma$ imply $(\ell_2, \upsilon_2) \xrightarrow{d_2,a} (\ell_2', \upsilon_2')$ for some $d_2 \in \mathbb{R}_{\geq 0}$ with $(\ell_1', \upsilon_1\prime)R(\ell_2\prime, \upsilon_2\prime)$ and viceversa.

Theorem 5.3.1. *Clock equivalence is a bisimulation equivalence.*

Clock equivalence defines a bisimulation over atomic propositions. Since clock equivalence is a finite-index relation, we obtain a finite quotient Transition System:

$$\text{RTS(TA)}$$

From this, we can derive that:

1. Every path in the infinite TS(TA) has a corresponding path in the finite RTS(TA).
2. Every path in RTS(TA) corresponds to an infinite set of paths in TS(TA).
3. Computation Tree Logic-style logical properties (used in model checking) are preserved in RTS(TA).

Thus, region-based automata allow efficient analysis of timed systems while maintaining correctness guarantees.

5.4 Verification

The Region Transition System can be leveraged to verify the reachability of specific locations in a Timed Automaton.

Theorem 5.4.1. *Determining whether a location is reachable in a Timed Automaton is PSPACE-complete.*

The complexity arises from the exponential growth of the region Transition System, which depends on both the number of clocks and the binary representation of the constants used for clock comparisons.

Since bisimulation equivalence implies Computation Tree Logic equivalence (even for systems with aninfinite number of states). This ensures that the verification results remain consistent.

However, Computation Tree Logic has limitations when expressing precise timing constraints, as it lacks a metric for time. Despite this, it remains a valuable tool for verification in many cases.

5.4.1 Timed Computation Tree Logic

Timed Computation Tree Logic extends Computation Tree Logic by introducing a metric until operator that allows reasoning about time constraints.

5.4.1.1 Syntax

Timed Computation Tree Logic state formulas follow the usual Computation Tree Logic structure but incorporate atomic clock constraints g (typically defined over the set of clocks in a Timed Automaton):

$$\Phi :== \text{true} \mid a \mid g \mid \Phi \wedge \Phi \mid \neg\phi \mid E\phi \mid \forall\phi$$

Here, $a \in \text{AP}$ is the set of atomic propositions, and $g \in \text{acc}(C)$ are atomic clock constraints over the set of clocks C in a Timed Automaton. In addition, Timed Computation Tree Logic introduces path formulas:

$$\phi :== \Phi U^J \Phi$$

Here, J is an interval with natural number bounds. The derived operators are:

- *Future operator* ($F^J\Phi = \text{true}U^J\Phi$): Φ must hold at some point within interval J.
- *Existential globally operator* ($EG^J\Phi = \neg AF^J\neg\Phi$): there exists a path where Φ holds during J.
- *Universal globally operator* ($AG^J\Phi = \neg EF^J\neg\Phi$): for all paths, Φ holds throughout J.

5.4.1.2 Semantic

Path formulas must hold only on time-divergent paths. If s is a state represented as $s = \langle \ell, \eta \rangle$, then:

- $s \models \text{true}$
- $s \models a$ if and only if $a \in L(\ell)$

- $s \models g$ if and only if $\eta \models g$
- $s \models \neg\Phi$ if and only if $s \not\models \Phi$
- $s \models \Phi \wedge \Psi$ if and only if $s \models \Phi \wedge s \models \Psi$
- $s \models \exists\phi$ if and only if $\pi \models \phi$ for some $\pi \in \text{paths}_{\text{div}}(s)$
- $s \models \forall\phi$ if and only if $\pi \models \phi$ for all $\pi \in \text{paths}_{\text{div}}(s)$

For a time-divergent path $\pi \in s_0 \overset{d_0}{\Longrightarrow} s_1 \overset{d_2}{\Longrightarrow} \dots$, we can define eventuality, globality and bounded until operators.

Eventuality Eventuality operator is denoted as $\pi \models \lozenge^J \Phi$. It holds when there esists some index $\exists i \geq 0$ and delay $d \in [0, d_i]$ such that:

$$s_i + d \models \Phi \qquad \sum_{k=0}^{i-1} d_k + d \in J$$

This operator means that a ϕ-state is reached at some time instant $t \in J$.

Globality Globality operator is denoted as $\pi \models \square^J \Phi$. It holds when for every index $\exists i \geq 0$ and all delay $d \in [0, d_i]$ such that:

$$s_i + d \models \Phi \qquad \sum_{k=0}^{i-1} d_k + d \in J$$

This operator means that all states visited within interval J satisfy ϕ.

Bounded until Bounded until operator is denoted as $\Phi \cup^J \Psi$. Holds when at some point within J, a state satisfying Ψ is reached, and at all previous time instants, Φ or Ψ holds. In standard Linear Temporal Logic and Computation Tree Logic, $\Phi \cup \Psi$ is equivalent to $(\Phi \vee \Psi) \cup \Psi$.

5.4.1.3 Timed Computation Tree Logic Semantics for Timed Automata

Let Timed Automata be a Timed Automaton with: clocks C, locations Loc, initial states Loc_0, and a clock evaluation function eval(C). For a Timed Computation Tree Logic state formula Φ, the satisfaction set is defined as:

$$\text{sat}(\Phi) = \{s \in \text{Loc} \times \text{eval}(C) \mid s \models \Phi\}$$

A Timed Automaton Timed Automata satisfies a Timed Computation Tree Logic formula Φ if and only if Φ holds in all initial states:

$$\text{TA} \models \Phi \Leftrightarrow \forall \ell_0 \in \text{Loc}_0 \quad \langle \ell_0, \eta_0 \rangle \models \Phi$$

Here, $\eta_0(x) = 0$ for all $x \in C$.

5.4.2 Model checking

The construction of the Region Transition System for a Timed Automaton can be modified to incorporate regions introduced by a Timed Computation Tree Logic formula Φ. When defining RTS(TA), we must also consider clock constraints in Φ, specifically the maximum constant c_x for each clock x. The resulting system is denoted as $\text{RTS}(\text{TA}, \Phi)$.

5.4.2.1 Equivalence and clock elimination

Unfortunately, $\text{RTS}(\text{TA}, \Phi)$ is not bisimilar to TS(TA) because certain delay transitions are eliminated. However, this is still valid since $\text{RTS}(\text{TA}, \Phi)$ is stutter-equivalent to TS(TA). A transformation exists to eliminate timing parameters from every Timed Computation Tree Logic subformula by introducing new clocks.

Temporal logic with clocks Instead of using dense time operators, a logic called Timed Computation Tree Logic extends Computation Tree Logic by introducing a new set of formula clocks, atomic constraints, and a new in operator to reset a given clock to zero. A Timed Computation Tree Logic formula Φ can always be rewritten into an equivalent Timed Computation Tree Logic formula Φ'. We can then construct a Region Transition System denoted as $\text{RTS}(\text{TA}, \Phi')$, by introducing the formula clocks and their constraints into $\text{RTS}(A)$.

Timed Computation Tree Logic can be interpreted as a Computation Tree Logic formula over $\text{RRTS}(\text{TA}, \Phi')$, as long as:

- Clock constraints are treated as atomic propositions.
- Non-Zeno behaviors (paths with infinitely many actions in a finite time) are excluded.

Under the non-Zeno assumption:

$$\text{TA} \models \Phi \Leftrightarrow \text{RTS}(\text{TA}, \Phi') \models \Phi'$$

Thus, standard Computation Tree Logic model-checking techniques can be applied. Various optimizations are often introduced in practice.

5.4.2.2 Complexity

Theorem 5.4.2. *For Timed Automaton Timed Automata and Timed Computation Tree Logic formula Φ, the Timed Computation Tree Logic model checking problem $TA \models \Phi$ can be determined in time:*

$$\mathcal{O}((N + K) \times |\Phi|)$$

Here, N and K are the number of states and transitions in the region Transition System $RTS(TS, \Phi)$, respectively.

Since Region Transition System states grow exponentially in the number of clocks and the binary representation of constants, the complexity is significant.

Theorem 5.4.3. *The Timed Computation Tree Logic model checking problem is PSPACE-complete.*

In practice, most model checkers do not support full Timed Computation Tree Logic. Symbolic techniques, which improve verification efficiency, cannot easily extend to full Timed Computation Tree Logic. Model checking safety, reachability, and ω-regular properties, Linear Temporal Logic, and Computation Tree Logic for Timed Automata is PSPACE-complete. Timed Computation Tree Logic satisfiability is undecidable. Model checking Timed Linear Temporal Logic is undecidable.

5.5 Timed languages

A timed ω-word (or simply timed word) is an infinite sequence of states, where each state is associated with a real-valued timestamp. Formally, a timed word is a sequence:

$$(\sigma_1, \tau_1), (\sigma_2, \tau_2), (\sigma_3, \tau_3), \ldots$$

Here, each symbol σ_i is a symbol from a finite alphabet, and τ_i is a real-valued timestamp indicating when σ_i occurs.

Definition (*Timed sequence*)**.** A timed sequence is an infinite sequence of time values:

$$\tau = \tau_1, \tau_2, \tau_3, \ldots$$

Here, each τ_i satisfies:

- *Monotonicity*: $\tau_i \leq \tau_{i+1}$ for all $i \geq 1$. If $\tau_i < \tau_{i+1}$ for all $i \geq 1$ the sequence is strictly monotonic.
- *Progress*: for every real number $t \in \mathbb{R}_{\geq 0}$, there exists some $i \geq 1$ such that $\tau_i > t$.

Definition (*Timed word*)**.** A timed word over an alphabet $\mathcal{A}$ (a finite set of symbols) is a pair:

$$\sigma, \tau$$

Here, $\sigma = \sigma_1\sigma_2\sigma_3 \ldots$ is an ω-word (an infinite sequence of symbols from $\mathcal{A}$), and τ is a timed sequence providing the timestamps associated with each symbol.

Definition (*Timed language*)**.** A timed language over an alphabet $\mathcal{A}$ is a set of timed words over $\mathcal{A}$.

Untime operation The untime operation on a timed language discards the time values associated with symbols.

Definition. For a timed language $\mathcal{L}$ over $\mathcal{A}$:

$$\text{untime}(\mathcal{L}) = \{\sigma \in \mathcal{A}^\omega \mid \text{there exists a } \tau \text{ such that } (\sigma, \tau) \in \mathcal{L}\}$$

untime($\mathcal{L}$) is a set of ω-words.

5.5.1 Timed Automata

A timed automato is defined over an alphabet of actions, called the input alphabet. It is represented as a tuple:

$$\mathcal{A} = \langle L, \Sigma, C, E, l_0, J, F \rangle$$

Here:

- L is a finite set of locations.
- Σ is a finite set of input symbols.
- C is a finite set of clocks.
- $l_0 \in L$ is the initial location.
- $E \subseteq L \times B(C) \times \Sigma \times 2^C \times L$ is a set of edges, where $B(C)$ denotes the set of clock constraints.
- $J : L \to B(C)$ assigns invariants to locations.
- $F \subseteq L$ is the set of final locations.

A timed word is accepted if there exists a path leading infinitely often to a final location.

Definition (*Run*)**.** A run $r_{(\sigma,\tau)}$ over a timed word (σ, τ), where $\sigma = \sigma_1\sigma_2\dots$ and $\tau = \tau_1\tau_2\dots$ is an infinite sequence:

$$r_{(\sigma,\tau)} = \langle l_0, \eta_0 \rangle \xrightarrow{\sigma_1,\tau_1} \langle l_1, \eta_1 \rangle \xrightarrow{\sigma_2,\tau_2} \langle l_2, \eta_2 \rangle \dots$$

Here, $l_i \in L$ and η_i are clock valuations satisfying:

- *Initialization*: $\eta_0(x) = 0$ for all $x \in C$.
- *Transition condition*: for all $i \geq 1$, there exists an edge $e \in E$ such that:

  $$e = \langle l_{i-1}, \gamma_i, \sigma_i, \hat{C}_i, l_i \rangle$$

  Here, $(\eta_{i-1} + (\tau_i - \tau_{i-1}))$ satisfies both γ_i and $J(l_{i-1})$, and the clock update is given by:

  $$\eta_i = [\hat{C}_i \mapsto 0](\eta_{i-1} + (\tau_i - \tau_{i-1}))$$

- η_i satisfies the invariant $J(l_i)$.

The set of locations visited infinitely often in the run is denoted by $\inf(r_{(\sigma,\tau)})$. A timed word (σ, τ) is accepted by a Timed Automaton $\mathcal{A}$ if:

$$\inf(r(\sigma, \tau)) \cap F \neq \emptyset$$

5.5.2 Timed regular languages

Definition (*Timed regular language*)**.** A timed regular language is a language accepted by some Timed Automaton.

If $\mathcal{L} \subseteq \mathcal{A}^\omega$ is an ω-regular language, then $\mathcal{L}_t = \{(\sigma, \tau) \mid \sigma \in \mathcal{L}\}$ is a timed regular language.

Untimed languages Given a Timed Automata that accepts a language $\mathcal{L}$, there exists a Büchi automaton (BA) that accepts untime($\mathcal{L}$). That is, for any timed regular language $\mathcal{L}$, its untimed version untime($\mathcal{L}$) is ω-regular. However, simply removing timing constraints is insufficient.

5.5.3 Timed Automata properties

Timed Automata are closed under union and intersection but not under complementation. Timed Automata cannot be determinized. Non-closure under complement limits automata-based model checking. Language inclusion and equivalence are undecidable. Emptiness is decidable by verifying the emptiness of the Region Automaton (similar to the Region Transition System).

Continuous time semantics Timed Automata semantics can be defined over signals instead of timed words. The standard Transition System semantics is based on timed words. Linear Temporal Logics are more naturally interpreted over signals. Decision problems become harder in continuous-time semantics.

5.5.4 Model Checking Timed Automata for Linear Temporal Logic

Model checking Timed Automata for Linear Temporal Logic (both pointwise and continuous semantics) is PSPACE-complete. The steps are:

1. Construct a Büchi automaton equivalent to the negation of a formula.
2. Build the product of this Büchi automaton with the Timed Automata.
3. Verify the emptiness of the resulting automaton.

Since Timed Automata are closed under intersection, the result is still a Timed Automata. However, most metric extensions of Linear Temporal Logic are undecidable.

5.6 Uppaal

Uppaal is a widely used tool for verifying Timed Automata, though it was not the first of its kind. The original tool for this purpose was Kronos. Uppaal was developed collaboratively by Uppsala University in Sweden and Aalborg University in Denmark, from which its name is derived.

Timed Automata network Uppaal enables users to construct complex models by composing networks of Timed Automata. These models are broken down into multiple interacting components, making them more manageable and modular. The interaction between components is achieved through synchronization channels. One component sends a message through a channel, while another component receives it, allowing for structured communication within the system.

Abstraction To analyze time constraints, Uppaal provides two levels of abstraction:

- *Region abstraction*: a fine-grained approach that can result in a large number of states.
- *Clock zones*: a coarser abstraction where a zone corresponds to a set of clock constraints, covering multiple regions and reducing complexity.

Committed and urgent locations In Uppaal, certain locations have special timing constraints:

- *Urgent locations*: time cannot pass when an automaton is in an urgent location. This can be modeled as resetting a clock upon entering the location and enforcing an invariant that prevents time progression.
- *Committed locations*: these are even more restrictive(time cannot pass, and the only possible transition must immediately exit the committed location). Transitions can only interleave with other committed locations.

Using urgent and committed locations effectively can help reduce unnecessary interleaving and minimize the number of clocks in a model, improving efficiency.

Urgent channels Urgent channels provide a mechanism for immediate synchronization. When two automata reach corresponding locations linked by an urgent channel, synchronization happens instantaneously. To maintain efficiency, transitions involving urgent channels cannot have clock constraints.

5.6.1 Query language

Uppaal allows users to verify system properties using a subset of Timed Computation Tree Logic. The following are some key query types:

- `E<> P` (P is reachable): there exists a path from the initial state leading to a state where P holds.
- `A<> P` (P is inevitable): on all paths from the initial state, a state where P holds will eventually be reached.
- `A[] P` (P is an invariant): in all reachable states from the initial state, P always holds.
- `E[] P` (P is potentially always true): there exists a path where P holds in all states along that path.
- `P -> Q` (P leads to Q): if P holds in a state, then eventually Q will hold in a future state.

Expressions within queries can describe specific locations, variable constraints, or clock constraints. Special keywords, such as `deadlock`, allow checking for potential system deadlocks. Uppaal's query language focuses on safety properties, ensuring that the system behaves as expected under all conditions.

CHAPTER 6

Hoare logic

6.1 Introduction

When verifying software, we often compare two approaches: testing and proof. Testing, also known as dynamic analysis, involves executing a program with various inputs to observe its behavior. It provides practical insights but is limited to a finite set of execution traces, meaning it cannot guarantee the absence of errors. In contrast, proofs, or static analysis, aim to establish correctness mathematically, ensuring that certain errors cannot occur. While proofs offer exhaustive guarantees, they are often impractical due to their complexity and limited automation.

Despite their impracticality in many real-world scenarios, learning proofs is valuable. They help us develop a structured way of thinking about program correctness, assist in software development and code inspection, and provide foundational insights into automated theorem provers.

6.1.1 Correctness

Program correctness defines a contract between the client and the implementation. Given an initial condition, a function f must ensure a specific outcome after execution.

This relationship is formally described using preconditions (assumptions about inputs) and postconditions (expected outcomes), which are expressed as predicates (boolean functions over the program state).

Definition (*Function correctness*)**.** A function is considered correct with respect to its specification if, given a valid precondition, it produces the expected postcondition upon execution.

Definition (*Program partial correctness*)**.** A program is partially correct if, whenever the precondition holds and the program terminates, the postcondition also holds.

Definition (*Program total correctness*)**.** A program is totally correct if, given a valid precondition, it both terminates and ensures the postcondition holds.

6.2 Syntax and semantic

A Hoare triple $\{\phi_1\}P\{\phi_2\}$ is a formal notation used to express program correctness, where ϕ_1 and ϕ_2 are predicates, and P is a program. This notation captures the relationship between the program's initial and final states.

Definition (*Hoare partial correctness*)**.** Partial correctness means that if execution starts in a state s satisfying ϕ_1, and if P terminates, then the resulting state s' must satisfy ϕ_2.

Definition (*Hoare total correctness*)**.** Total correctness strengthens this by requiring that, in addition to satisfying ϕ_2 upon termination, P must always terminate when started in a state satisfying ϕ_1.

For programs that do not contain loops, partial and total correctness are equivalent since termination is already guaranteed.

Strongest postconditions The strongest postcondition of a program P with respect to a precondition ϕ_1 is the most precise description of the states that P can reach when starting from ϕ_1. Formally, if $\{\phi_1\}P\{\phi_2\}$ holds and for any ϕ'_2 where $\{\phi_1\}P\{\phi'_2\}$ also holds, it follows that $\phi_2 \implies \phi'_2$, then ϕ_2 is the strongest postcondition.

Weakest preconditions The weakest precondition of a program P with respect to a postcondition ϕ_2 is the least restrictive condition on the initial state that guarantees P will achieve ϕ_2. Formally, if $\{\phi_1\}P\{\phi_2\}$ holds and for any ϕ'_1 where $\{\phi'_1\}P\{\phi_2\}$ also holds, it follows that $\phi'_1 \implies \phi_1$, then ϕ_1 is the weakest precondition.

6.3 Axioms

Hoare logic operates in a backward reasoning manner, working from the desired postconditions toward the preconditions. The fundamental axioms of Hoare logic include:

1. *Assignment rule*: assignments update the program state by substituting values into variables. If $y := t$ assigns the value of t to y, then the precondition must hold for the state after substitution:
$$\{\phi[t/y]\}y := t\{\phi\}$$
2. *Composition rule*: if executing P_1 transforms a state satisfying ϕ into a state satisfying ϕ', and P_2 transforms ϕ' into ϕ'', then executing P_1 followed by P_2 ensures ϕ'' holds:
$$\frac{\{\phi\}P_1\{\phi'\}, \{\phi'\}P_2\{\phi''\}}{\{\phi\}P_1; P_2\{\phi''\}}$$
3. *Conditional rule*: the correctness of an if-else statement depends on verifying both branches separately:
$$\frac{\{\phi \wedge c\}P_1\{\phi'\}, \{\phi \wedge \neg c\}P_2\{\phi'\}}{\{\phi\} \text{ if } c \text{ then } P_1 \text{ else } \{\phi'\} \text{ if } c \text{ then } P_2}$$
4. *Consequence rule*: if a weaker precondition ϕ implies a stronger one σ, and σ guarantees σ', which in turn implies the desired postcondition ϕ', then the Hoare triple remains valid:
$$\frac{\phi \to \sigma, \{\sigma\}P\{\sigma'\}, \sigma' \to \phi'}{\{\phi\}P\{\phi'\}}$$

5. *While rule*: for a loop while c do P, if executing P preserves ϕ as long as c holds, then when the loop terminates, ϕ must still be true while c is false:
$$\frac{\{\phi \wedge c\}P\{\phi\}}{\{\phi\} \text{ while } c \text{ do } P \text{ then } \{\phi \wedge c\}}$$

6.3.1 Loop correctness

To prove the correctness of a loop, we introduce an invariant (a property that remains true throughout execution):

1. *Initialization*: the invariant must hold before the loop starts:
$$\phi \implies \text{Inv}$$
2. *Partial correctness*: each iteration preserves the invariant:
$$\{\text{Inv} \wedge c\}P\{\text{Inv}\}$$
3. *Total correctness*: when the loop terminates, the invariant and the negation of the loop condition together must imply the desired postcondition:
$$\{\text{Inv} \wedge \neg c\} \implies \{\phi \wedge \neg c\}$$

By proving these properties, we ensure that the loop behaves as expected, either partially (if termination is not guaranteed) or totally (if termination is ensured).

6.4 Structural rules

The structural rules of Hoare logic define how logical connectives interact with Hoare triples, allowing us to reason about complex program properties systematically:

- *Conjunction rule*: if executing P ensures ψ_1 when starting from ϕ_1, and ensures ψ_2 when starting from ϕ_2, then P also ensures both conditions simultaneously when started from the conjunction of the two preconditions:
$$\frac{\{\phi_1\}P\{\psi_1\}\{\phi_2\}P\{\psi_2\}}{\{\phi_1 \wedge \phi_2\}P\{\psi_1 \wedge \psi_2\}}$$
- *Disjunction rule*: if P guarantees ψ_1 when starting from ϕ_1, and guarantees ψ_2 when starting from ϕ_2, then if either ϕ_1 or ϕ_2 holds initially, executing P will ensure either ψ_1 or ψ_2:
$$\frac{\{\phi_1\}P\{\psi_1\}\{\phi_2\}P\{\psi_2\}}{\{\phi_1 \vee \phi_2\}P\{\psi_1 \vee \psi_2\}}$$
- *Existential quantification rule*: if a Hoare triple holds for all values of v, then it also holds for any specific instance of v, meaning we can introduce a universal quantifier over v in both preconditions and postconditions:
$$\frac{\{\phi\}P\{\psi\}}{\{\forall v, \phi\}P\{\forall v, \psi\}}$$
- *Universal quantification*: if a Hoare triple holds for some value of v, then it must hold for at least one such value, allowing us to introduce existential quantification:
$$\frac{\{\phi\}P\{\psi\}}{\{\exists v, \phi\}P\{\exists v, \psi\}}$$