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 Probabilistic model checking . . . 21
3.5.1 Markov chains . . . 21
3.5.2 Probabilistic Computation Tree Logic . . . 22
3.5.3 Markov decision processes . . . 23

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 $\langle S, \mathrm{Act}, \to, I, \mathrm{AP}, L\rangle$, where:

- S is a set of states.
- Act is a set of input symbols (also called actions).
- $\to\subseteq \times \mathrm{Act} \times S$ 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.
- $L : S \to 2^{\mathrm{AP}}$ 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 $\langle s, i, s'\rangle \in\to$.

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 $\langle S, \mathrm{Act}, \rightarrow, I, \mathrm{AP}, L\rangle$ 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 AP set must also include all locations from the program graph.
- While this transformation results in a very large AP set, in practice, only a small portion is usually relevant for analyzing system properties.

2.3 Concurrency

Given two transition systems:

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

Their interliving is defined as:

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

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

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

In practice, the two TS 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:

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

This straightforward composition allows both transition systems to operate independently.

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

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

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:

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

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 $\mathrm{Act}_1 \cap \mathrm{Act}_2$, handshaking reduces to standard interleaving:

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

If H includes all common actions, we simply write:

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

Given two transition systems:

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

Their handshaking synchronization is defined as:

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

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:

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

Then all processes can synchronize on these actions. In this case, the handshaking operator $\|_H$ is associative, meaning we can compose multiple transition systems as:

$$\mathrm{TS} = \mathrm{TS}_1 \|_H \mathrm{TS}_2 \|_H \cdots \|_H \mathrm{TS}_n$$

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 FIFO channels, which explicitly define the direction of communication. Now, transitions in a system include:

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

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 FIFO 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 FIFO 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 $\text{TS} = \langle S, \text{Act}, \rightarrow, I, \text{AP}, L\rangle$. The enabled actions at a state s are given by $\text{Act}(s) = \left\{\alpha \in \text{Act} \mid \exists s' \in s, s \overset{\alpha}{\rightarrow} s'\right\}$ For an infinite execution fragment $\rho = s_0 \overset{\alpha_0}{\rightarrow} s_1 \overset{\alpha_1}{\rightarrow} \ldots$ 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:

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

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

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

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

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 $\text{TS} \models_{\mathcal{F}} P$ if and only if:

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

This follows the hierarchy:

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

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:

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

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 ($\text{TS} \models P_{\text{safe}}$) or false ($\text{TS} \not\models P_{\text{safe}}$), 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 $P_{\text{safe}}$
2: Construct the product transition system $\text{TS} \otimes \mathcal{A}$
3: Check the invariant $P_{\text{inv}(\mathcal{A})}$ with proposition $\neg F = \wedge_{q \in F} \neg q$ on $\text{TS} \otimes \mathcal{A}$
4: if $\text{TS} \otimes \mathcal{A}$ 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 $\text{TS} \otimes \mathcal{A}$ 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:

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

Here, $|\text{TS}|$ 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 TS $\otimes \mathcal{A}$, which combines paths in TS with runs in $\mathcal{A}$.

2. *Graph analysis*: check if there exists a path in 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 TS are non-accepting, meaning 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:

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

Here, a is an atomic proposition, $\bigcirc\phi$ (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 ($\Diamond\phi \overset{\text{def}}{=} \text{true} \cup \phi$) and always ($\Box\phi \overset{\text{def}}{=} \neg\Diamond\neg\phi$) 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:

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

The timed interpretation of the operators is as follows:

- $\bigcirc^k\phi$ 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 Time 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 Time Logic are expressed in terms of states, rather than actions.

Action-based fairness is more intuitive and straightforward but Linear Time Logic fairness is equally expressive. Action-based fairness assumptions can be translated into Linear Time 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 Time Logic, fairness can be expressed using different types of fairness constraints:

1. *Unconditional fairness*: $\Box\blacklozenge\psi$.
2. *Strong fairness*: $\Box\blacklozenge\phi \to \Box\blacklozenge\psi$.
3. *Weak fairness*: $\blacklozenge\Box\phi \to \blacklozenge\Box\psi$.

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 Time Logic formula ϕ, and fairness assumption fair:*

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

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 Time Logic formula can be transformed into PNF, but this transformation requires new dual operators.

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

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

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

Theorem 3.4.2. *For any Linear Time Logic formula ϕ, there exists an equivalent Linear Time 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:

$$\phi R \psi \overset{\text{def}}{=} \neg(\neg\phi \cup \neg\psi)$$

The semantycs is $\sigma \models \phi R \psi$ if $\forall j \geq 0, \sigma[j \ldots] \models \psi$ or $\exists i \geq 0 \quad (\sigma[i \ldots] \models \phi) \wedge \forall k \leq i \quad \sigma[k \ldots] \models \psi$. 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 Time Logic with the release operator is:

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

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

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

For any Linear Time 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 Time Logic formula ϕ can be represented by a nondeterministic Büchi automaton Let words(ϕ) be the set of ω-words satisfying an Linear Time 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 Time Logic formula ϕ, there exists a corresponding generalized Büchi automaton $\mathcal{G}_\phi$. For any Linear Time 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 Time 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 Time 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 Time Logic formulas ϕ_n with $|\phi_n|\, \mathcal{O}(poly(n))$ such that every nondeterministic Büchi automaton has at least 2^n states.*

While the NBA construction can have an exponential number of states, these automata are more expressive than Linear Time Logic formulas themselves.

Complexity The time and space complexity of the Linear Time 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 NBA 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 Time Logic formulas can be determined by checking the emptiness of the corresponding NBA. 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 Time 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 $\mathcal{A} = \left\langle Q, 2^{\text{AP}}, \delta, Q_0, F \right\rangle$ with $\mathcal{L}_\omega(\mathcal{A}) = \text{words}(\phi)$
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 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.

3.5.1 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 $\mathcal{M} = \langle S, \text{Pr}, \ell_{\text{init}}, \text{AP}, L \rangle$ where:

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

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

3.5.1.1 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.

3.5.2 Probabilistic Computation Tree Logic

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

3.5.2.1 Syntax

The syntax of Probabilistic Computation Tree Logic consists of state and path formulae. State formulae 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, $a \in AP$ is an atomic proposition, ϕ is a path formula, and $J \subseteq [0,1]$ is a probability interval.

Path formulae define properties over execution paths:

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

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

3.5.2.2 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:

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

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

3.5.2.3 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 3.5.1. *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:*

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

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

Property 3.5.2. There is no Computation Tree Logic formula that is equivalent to $\mathbb{P}_{>0}(\Box a)$.

Property 3.5.3. There is no qualitative Probabilistic Computation Tree Logic formula that is equivalent to $\forall\Diamond a$.

Property 3.5.4. There is no qualitative Probabilistic Computation Tree Logic formula that is equivalent to $\exists\Box a$.

3.5.3 Markov decision processes

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

Theorem 3.5.2. *For a finite A 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:*

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

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.