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

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 $\langle S, \text{Act}, \rightarrow, I, \text{AP}, L \rangle$, where:

- S is a set of states.
- Act is a set of input symbols (also called actions).
- $\rightarrow \subseteq \times \text{Act} \times S$ 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.
- $L : S \rightarrow 2^{\text{AP}}$ 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 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 Concurrecy

Given two transition systems:

$$\mathrm{TS}_1 = \langle S_1, \mathrm{Act}_1, \to_1, I_1, \mathrm{AP}_1, L_1 \rangle \qquad \mathrm{TS}_2 = \langle S_2, \mathrm{Act}_2, \to_2, I_2, \mathrm{AP}_2, L_2 \rangle$$

Their interliving is defined as:

$$\mathrm{TS}_1 \mid\mid\mid \mathrm{TS}_2 = \langle S_1 \times S_2, \mathrm{Act}_1 \cup \mathrm{Act}_2, \to_1, I_1 \times I_2, \mathrm{AP}_1 \cup \mathrm{AP}_2, L \rangle$$

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

$$\mathrm{TS}_1 = \langle S_1, \mathrm{Act}_1, \to_1, I_1, \mathrm{AP}_1, L_1 \rangle \qquad \mathrm{TS}_2 = \langle S_2, \mathrm{Act}_2, \to_2, I_2, \mathrm{AP}_2, L_2 \rangle$$

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:

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

This allows for synchronized execution across multiple processes.

2.4 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:

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

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 $\mathrm{capacity}(c) = 0$, the sender and receiver must synchronize instantly (just like standard handshaking), but with a different syntax.
- If $\mathrm{capacity}(c) > 0$, 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.

2.5 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.

2.5.1 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.

2.5.2 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.