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 $\sigma = i_1 i_2 i_3, \ldots$ 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:

$$s_0 i_1 s_1 i_2 s_2 \ldots$$

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 $\sigma = i_1 i_2 \ldots i_k$ and a finite run $r_\sigma = s_0 i_1 s_1 i_2 s_2 \ldots i_k s'$.

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:

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

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.