形式验证与模型检查
Formal Verification & Model Checking

By, Fahad Sabah

TFSE_Fall2025_北京工业大学，中国

TFSE_Fall2025_Beijing University of Technology, China

软件架构的定义
Formal Verification

In simple terms, Formal Verification is the process of using mathematical reasoning to prove that a system (like a computer chip or software) behaves correctly, according to its specification.

Think of it like this:

Traditional Testing: *You try a bunch of examples and see if it works. ("I tested the elevator for floors 1, 3, and 10, and it worked.")*

Formal Verification: *You mathematically prove that it will work for every possible scenario, no matter how rare. ("I have proven that this elevator will never get stuck between floors, will always answer calls, and will never open its doors while moving, under any condition.")*

简单来说，形式验证是利用数学推理证明系统（如计算机芯片或软件）按照其规范正确行为的过程。

可以这样想：
传统测试：你尝试很多例子，看看是否有效。（“我测试了1、3、10层的电梯，成功了。”）

形式验证：你用数学证明它适用于所有可能的情景，无论多么罕见。（“我已经证明这部电梯永远不会在楼层间卡住，永远会接听呼叫，且无论在什么情况下移动时都不会打开门。”）

The Core Idea: The "What" vs. The "How"

核心理念：“做什么”与“怎么做”

Formal Verification separates two things:

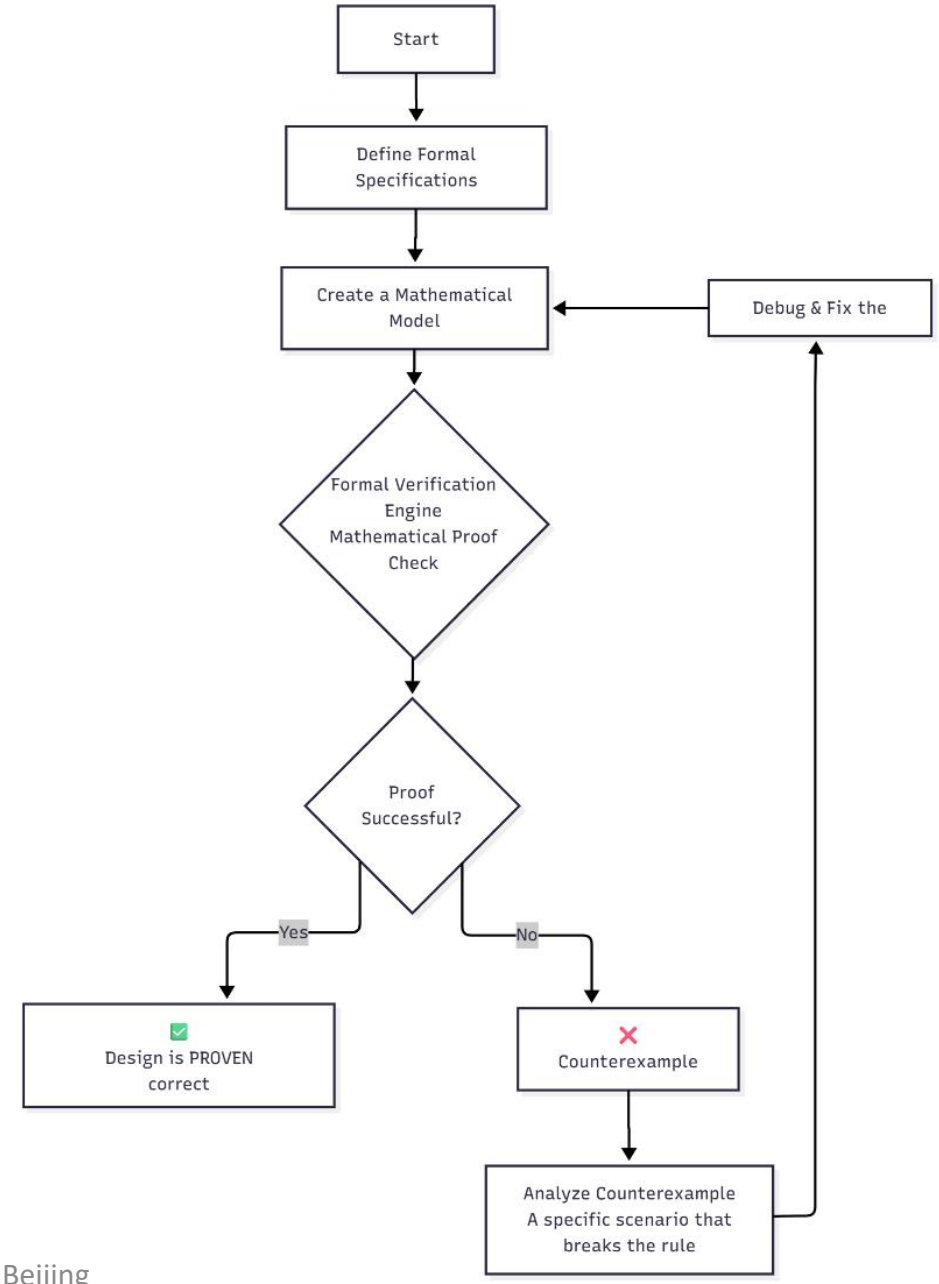
- The Specification (The "What"): A set of strict, mathematical rules that define what the system is supposed to do. (e.g., "The 'unlock' door shall only be active when the correct 4-digit code is entered.")
- The Implementation (The "How"): The actual code or hardware design that describes how the system is built.
- The goal of Formal Verification is to prove, beyond any doubt, that the Implementation always satisfies the Specification.

Simple Explanation [Formal Verification]

Here is a flowchart that illustrates the process:

- Model the System: The chip or software design is translated into a formal model—a set of logical equations and states that a computer can understand mathematically.

- Define the Specifications: The desired behavior is written as a set of formal properties. These are often "assertions" (things that must always be true) or "cover" statements (interesting scenarios that should be possible).
  - Example Assertion: assert (door_open -> speed == 0); // "The door must only be open when the speed is zero."



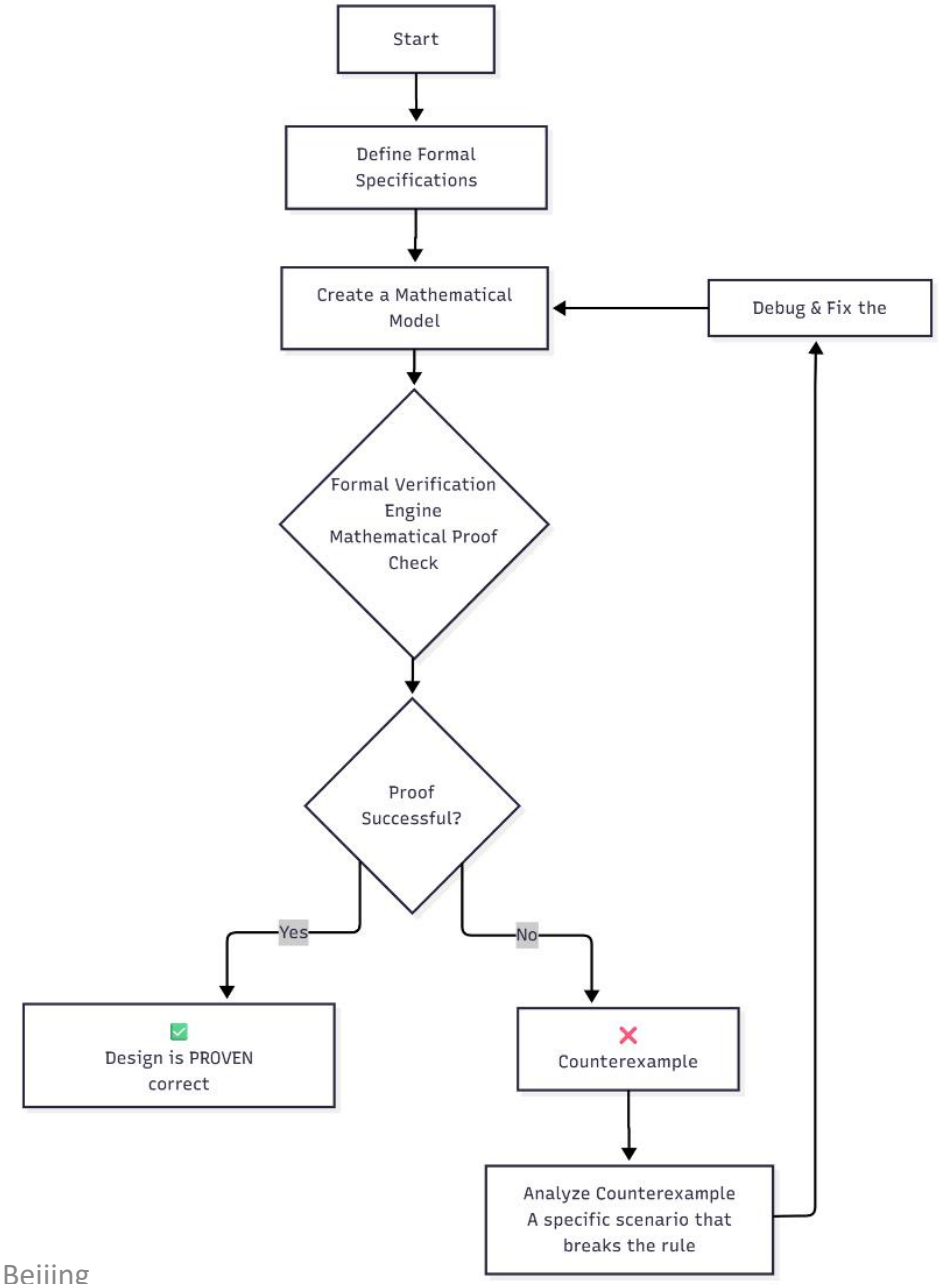
Simple Explanation [Formal Verification]

Run the Formal Tool (The "Engine"): A specialized tool, called a formal verification engine, takes the model and the specifications. It doesn't test with numbers; instead, it explores the entire state space using techniques like Theorem Proving or Model Checking.

The Two Possible Outcomes:

- ☑ Proof: The tool concludes that the specification holds for all possible inputs and states. This is the ultimate goal—a guarantee of correctness.

- ❌ Counterexample: The tool finds a specific sequence of inputs (a scenario) that causes the design to violate the specification. This is incredibly valuable, as it often uncovers hidden, complex bugs that traditional testing would likely miss for years.



Outcomes:

- If the property is violated, the tool returns a "counterexample" (a trace of execution) that demonstrates the violation.
- If the property holds for all possible states, the verification is complete.
- If the model is too large for the tool to explore completely, it may run out of memory.

A Concrete Example: A Traffic Light Controller

- Specification (The Rule): "The lights for intersecting roads must never be green at the same time."
- Implementation (The Design): The code that controls the lights.
- Traditional Testing might simulate a day's worth of traffic and never see a failure.
- Formal Verification would prove that no combination of sensor inputs, timers, or states could ever lead to both lights being green simultaneously.

Where is it Used?

It's critical in industries where failure is not an option:

- ❑Microprocessor Design (Intel, AMD)
- ❑Aerospace & Avionics (flight control systems)
- ❑Medical Devices (pacemakers)
- ❑Cryptocurrency & Blockchain (smart contracts)
- ❑Automotive (autonomous driving systems)

In short, Formal Verification is the highest standard of quality assurance, providing a mathematical guarantee that a system will behave as intended.

Example 1: The Smart Door Lock

Scenario

- Imagine a smart door lock that opens only when you enter the correct 4-digit code.

Specification (The Rules)

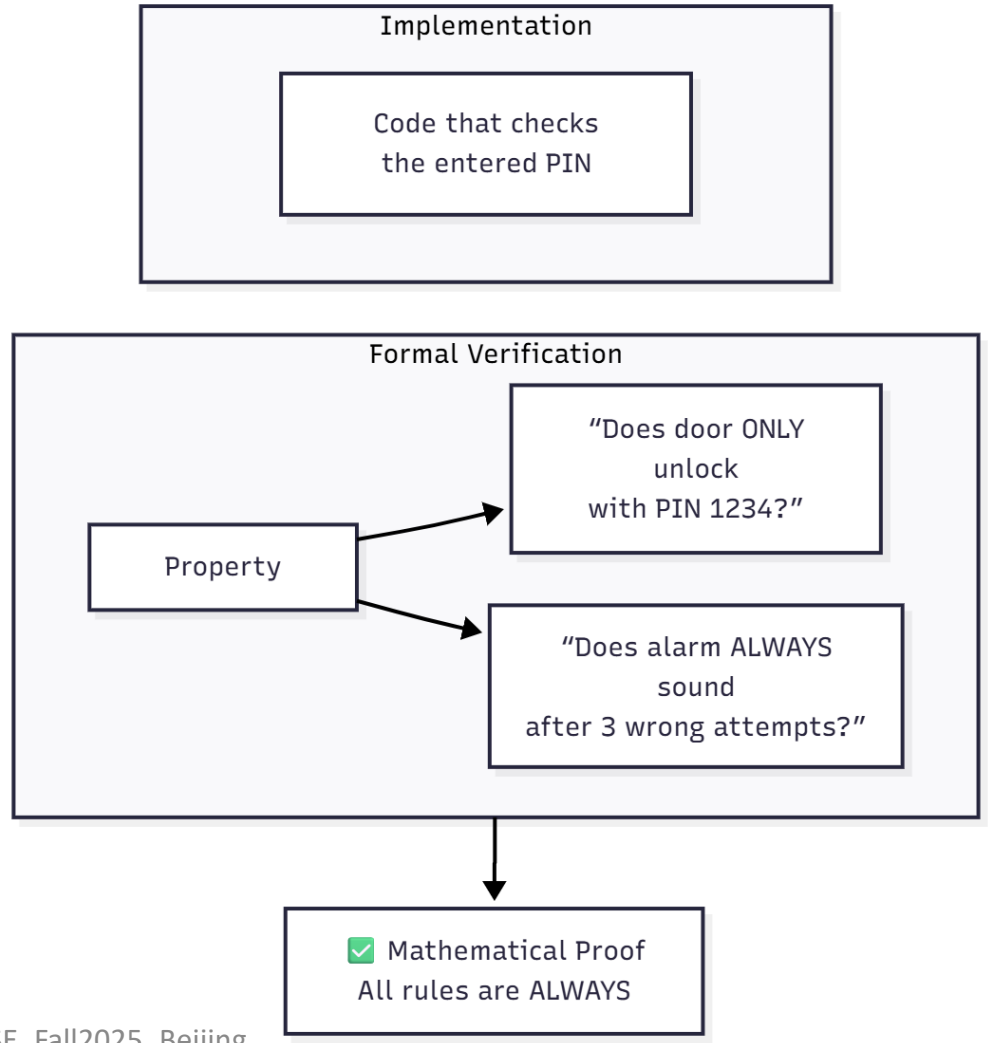
- Rule 1: The door can only unlock when the correct code "1234" is entered.
- Rule 2: The door must remain locked for all other codes.
- Rule 3: After 3 wrong attempts, the alarm must sound.

Implementation vs. Verification

What Formal Verification Does:

- Instead of just testing codes like "1111", "9999", or "1234", the formal tool considers every possible 4-digit combination (0000-9999) and every possible sequence of attempts to mathematically prove:

- No single wrong code can ever open the door
- The alarm will always trigger exactly on the 3rd wrong attempt
- There are no hidden backdoors or special codes that could open the door



Example 2: The Elevator Controller

Scenario

- An elevator that moves between 3 floors.

Specification (The Rules)

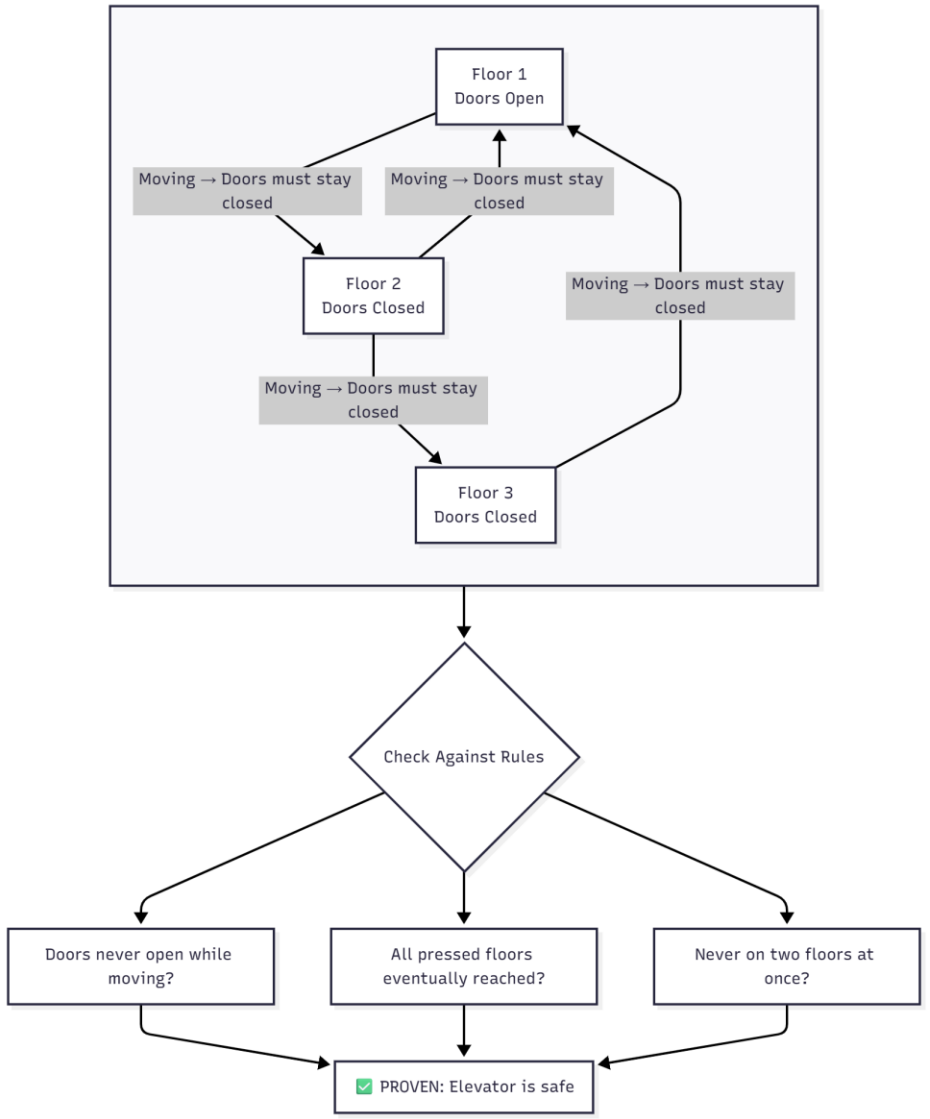
- Safety Rule: Doors must NEVER open while the elevator is moving
- Liveness Rule: If you press a floor button, the elevator must eventually reach that floor
- Logic Rule: The elevator can't be on two floors at once

What Formal Verification Does:

The tool explores every possible sequence of events:

- ❑What if someone presses all buttons at once?
- ❑What if the power fails between floors?
- ❑What if the door sensor breaks while moving?
- 3 Floors, 2 Door States, 2 Moving States, 8 Request Combinations???

It proves that even in these edge cases, the doors will never open while moving, and the elevator will always eventually serve all requests.



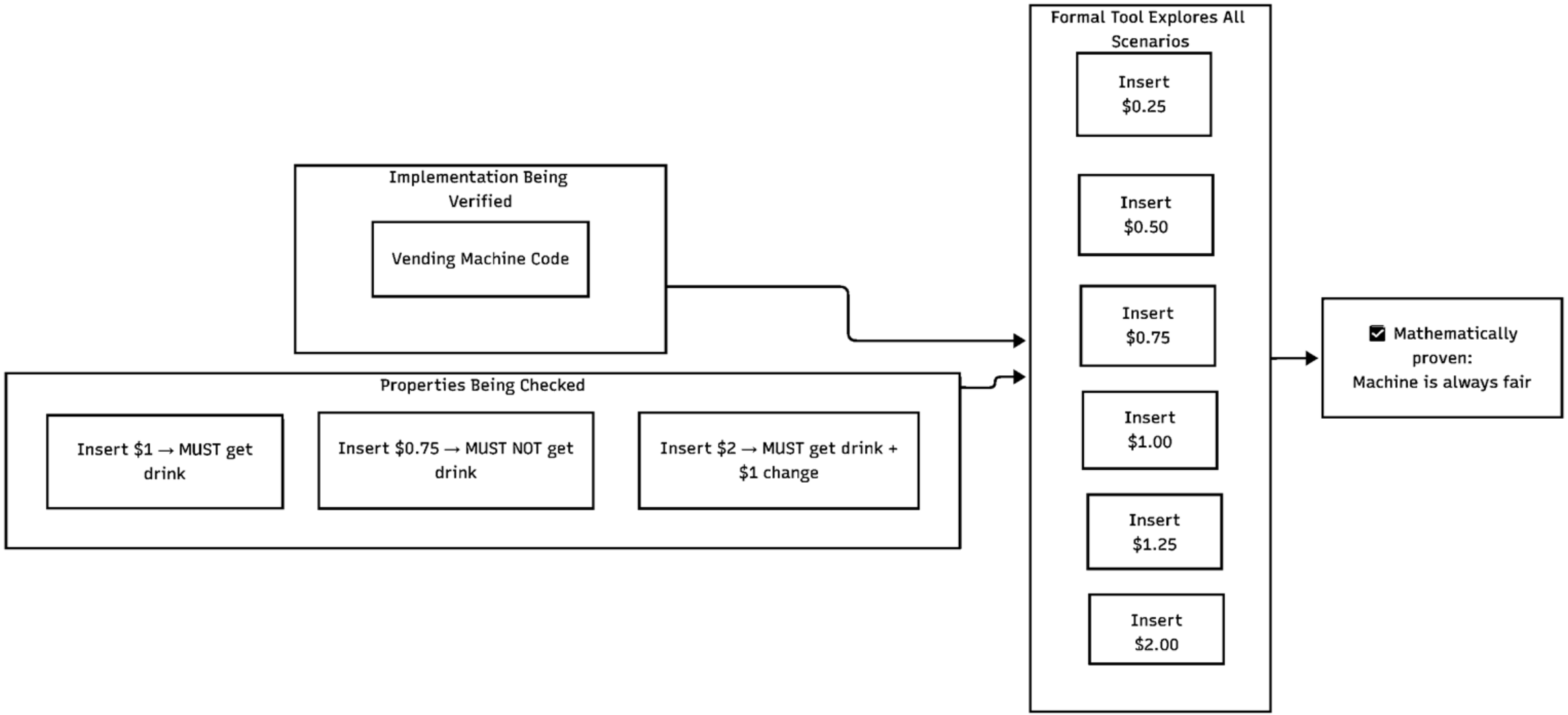
Example 3: The Vending Machine

Scenario

A simple vending machine that sells drinks for $1.

Specification (The Rules)

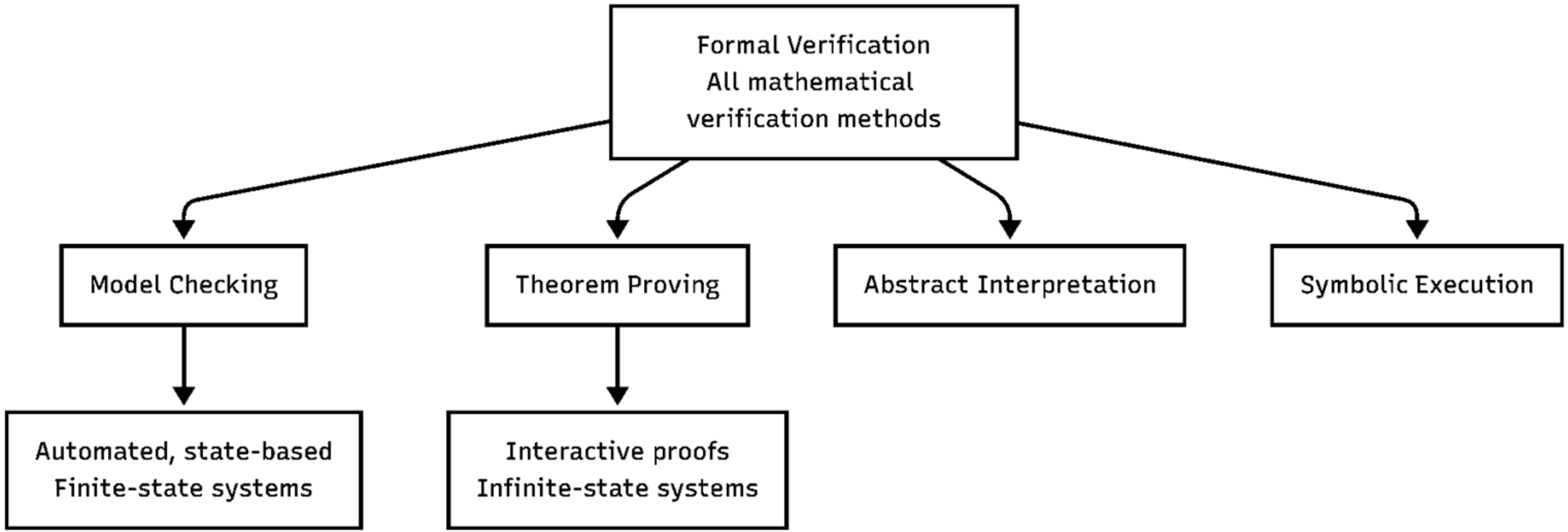
- ❑Fairness Rule: You must get a drink if you pay $1
- ❑No Free Lunch: You must NOT get a drink if you pay less than $1
- ❑Correct Change: If you pay $2, you must get a drink + $1 change


Implementation Being Verified
Vending Machine Code
Properties Being Checked
Insert $1 → MUST get drink
Insert $0.75 → MUST NOT get drink
Insert $2 → MUST get drink + $1 change
Formal Tool Explores All Scenarios
Insert $0.25
Insert $0.50
Insert $0.75
Insert $1.00
Insert $1.25
Insert $2.00
Mathematically proven:
Machine is always fair

What Formal Verification Does:

- Instead of testing with a few coin combinations, the formal tool considers every possible sequence of coins:
- 25¢ → 25¢ → 25¢ → 25¢ (should get drink)
- 25¢ → 25¢ → 25¢ (should NOT get drink)
- $1 bill (should get drink)
- $2 bill (should get drink + $1 change)

It mathematically proves the machine will NEVER give a free drink or keep your money without giving a drink.


Formal Verification
All mathematical
verification methods
Model Checking
Theorem Proving
Abstract Interpretation
Symbolic Execution
Automated, state-based
Finite-state systems
Interactive proofs
Infinite-state systems

Model checking

- An automated, state-based technique for formal verification.
- It systematically explores all possible states and transitions of a system's model to verify if a given property (like a safety property) holds true.

Process:

- Construct a formal model of the system.
- Formalize the desired properties or specifications.
- Use a model checker to automatically verify the model against the properties.

Key Characteristics:

☑ Fully automated - you push a button, it gives you yes/no + counterexamples
☑ Exhaustive - checks all possible states and all possible execution paths
☑ Provides counterexamples - when it fails, it shows you exactly how to reproduce the bug
❌ Limited to finite-state systems - can't handle infinite states directly
❌ State explosion problem - complex systems have too many states to check practically

elevator example was perfect for model checking because:

❑Finite floors (3)
❑Finite state variables (doors open/closed, moving/stopped)
❑Finite request combinations

Finite Automata

- Finite automata are abstract machines used to recognize patterns in input sequences, forming the basis for understanding regular languages in computer science.

- Consist of states, transitions, and input symbols, processing each symbol step-by-step.
- If ends in an accepting state after processing the input, then the input is accepted; otherwise, rejected.
- Finite automata come in deterministic (DFA) and non-deterministic (NFA), both of which can recognize the same set of regular languages.

Formal Definition of Finite Automata

A finite automaton can be defined as a tuple:

{ Q, Σ, q, F, δ }, where:

- Q: Finite set of states
- Σ: Set of input symbols
- q: Initial state
- F: Set of final states
- δ: Transition function

Types of Finite Automata

There are two types of finite automata:

- ❑Deterministic Finite Automata (DFA)
- ❑Non-Deterministic Finite Automata (NFA)

1. Deterministic Finite Automata (DFA)

A DFA is represented as {Q, Σ, q, F, δ}. In DFA, for each input symbol, the machine transitions to one and only one state. DFA does not allow any null transitions, meaning every state must have a transition defined for every input symbol.

DFA consists of 5 tuples {Q, Σ, q, F, δ}.

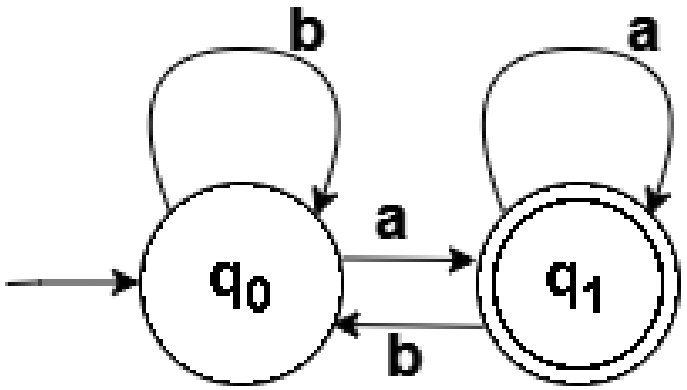
- Q : set of all states.
- Σ : set of input symbols. (Symbols which machine takes as input)
- q : Initial state. (Starting state of a machine)
- F : set of final state.
- δ : Transition Function, defined as δ : Q X Σ --> Q.

Example:

DFA that accepts all strings ending with 'a'.

Given:

- Σ = {a, b},
- Q = {q0, q1},
- F = {q1}



State \ Symbol	a	b
q0	q1	q0
q1	q1	q0

In this example, if the string ends in 'a', the machine reaches state q1, which is an accepting state.

2) Non-Deterministic Finite Automata (NFA)

NFA is similar to DFA but includes the following features:

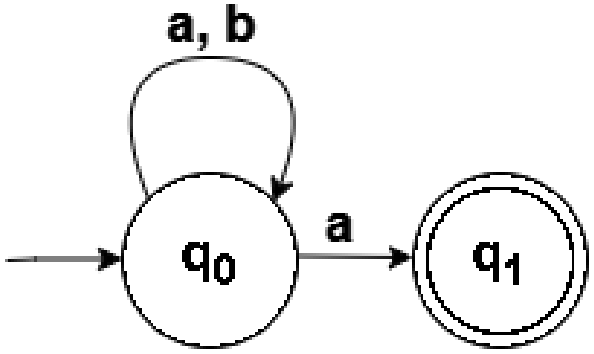
- It can transition to multiple states for the same input.
- It allows null (ϵ) moves, where the machine can change states without consuming any input.

Example:

NFA that accepts strings ending in 'a'.

Given:

- Σ = {a, b},
- Q = {q0, q1},
- F = {q1}



State \ Symbol	a	b
q0	{q0,q1}	q0
q1	φ	φ

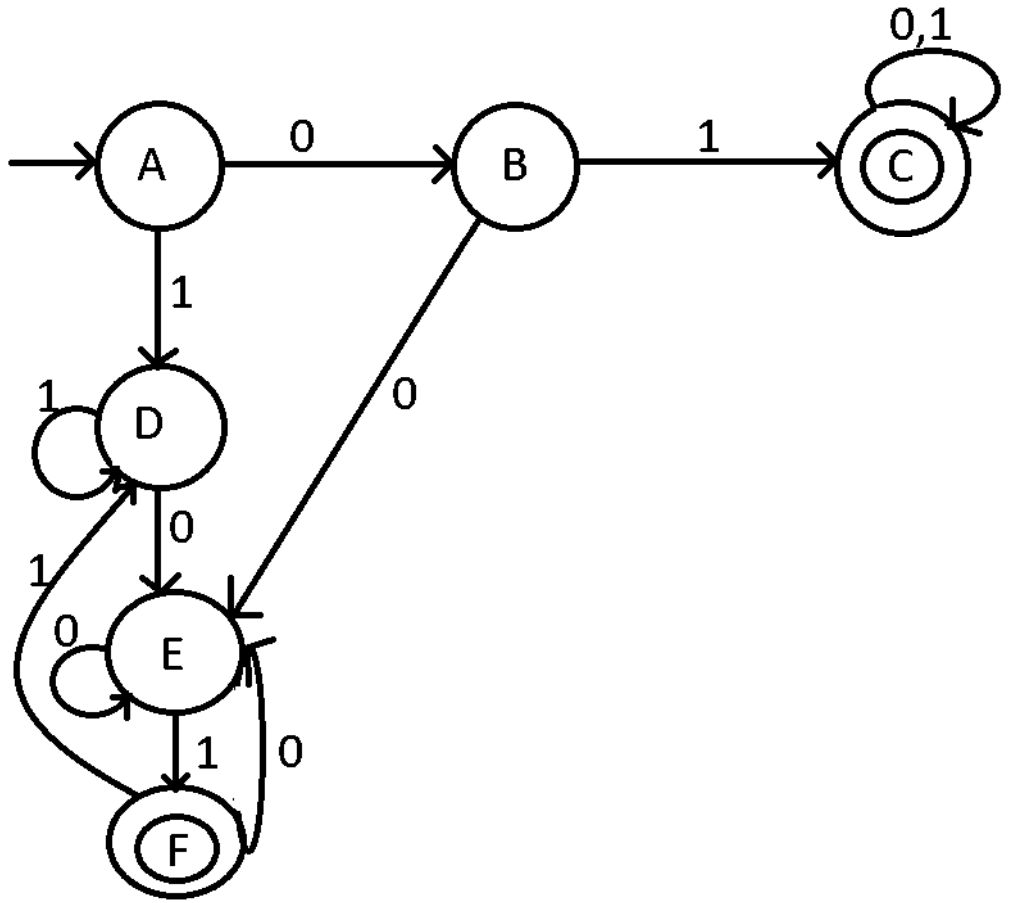
Transition table and Transition rules of the DFA

State	Input (0)	Input (1)
--->A	B	D
B	E	C
C*	C	C
D	E	D
E	E	F
F*	E	D

Transition Rules

1. δ :(A,0) = B
2. δ :(A,1) = D
3. δ :(B,0) = E
4. δ :(B,1) = C
5. δ :(C,0) = C
6. δ :(C,1) = C
7. δ :(D,0) = E
8. δ :(D,1) = D
9. δ :(E,0) = E
10. δ :(E,1) = F
11. δ :(F,0) = E
12. δ :(F,1) = D

DFA [accepts the string if the string either starts with "01" or ends with "01"]

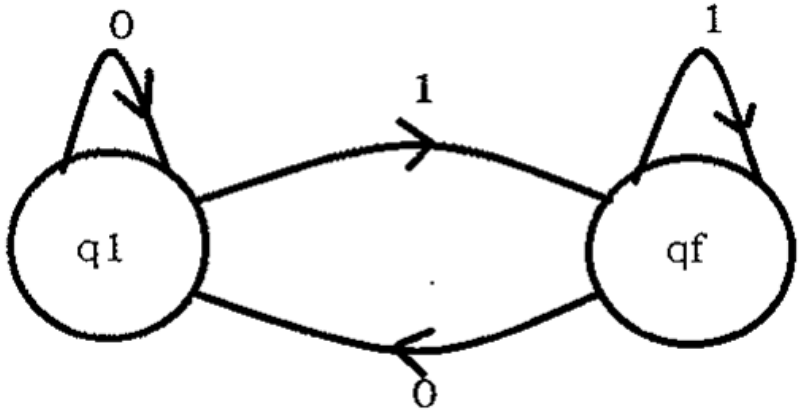


DFA to identify strings that contain the last character ‘1’.

```
q1()
{
    Read the next character
    if (no input character)
        printf("string not accepted")
    else if(input=='1')
        return qf() # we are calling the function qf
    else if(input=='0')
        return q1()
}
```

```
qf()
{
    Read the next character
    if (no input character)
        printf("string accepted")
    else if(input=='1')
        return qf() # we are calling the function qf
    else if(input=='0')
        return q1()
}
```

DFA to identify strings that contain the last character ‘1’.



Transition Table:

	Symbol	
State	0	1
q1	q1	qf
qf	q1	qf

Transition Rules???

Buchi Automata

A run is accepting if it visits at least one accepting state infinitely often.

$B = (Q, \Sigma, \delta, q_o, F)$

$Q = \{S0, S1\}$

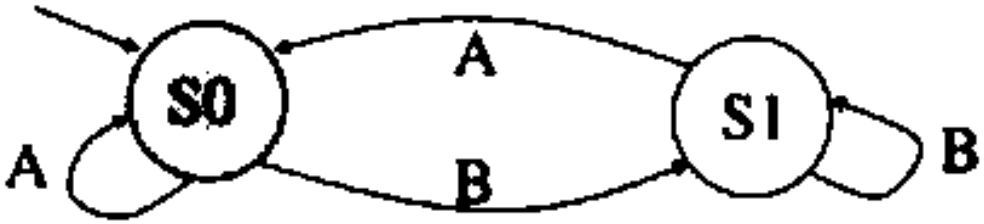
$\Sigma = \{A, B\}$

$q_o = S0$

$F = \{S0\}$

δ:

- S0 --A--> S0
- S0 --B--> S1
- S1 --B--> S1
- S1 --A--> S0



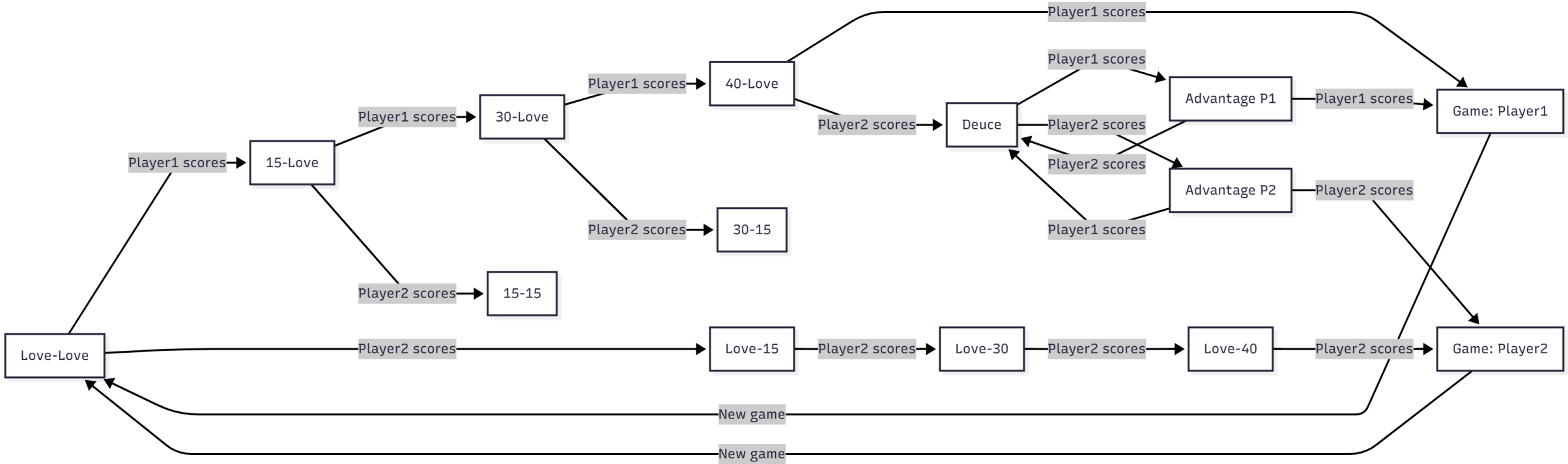
Accepting Words:

B, B, A, A, A, A, ... ☑

A, A, A, A, ... ☑

B, A, B, A, B, A, ... ✗ (p not always A eventually)

Tennis Game Scoring Automaton



Automata Mapping:

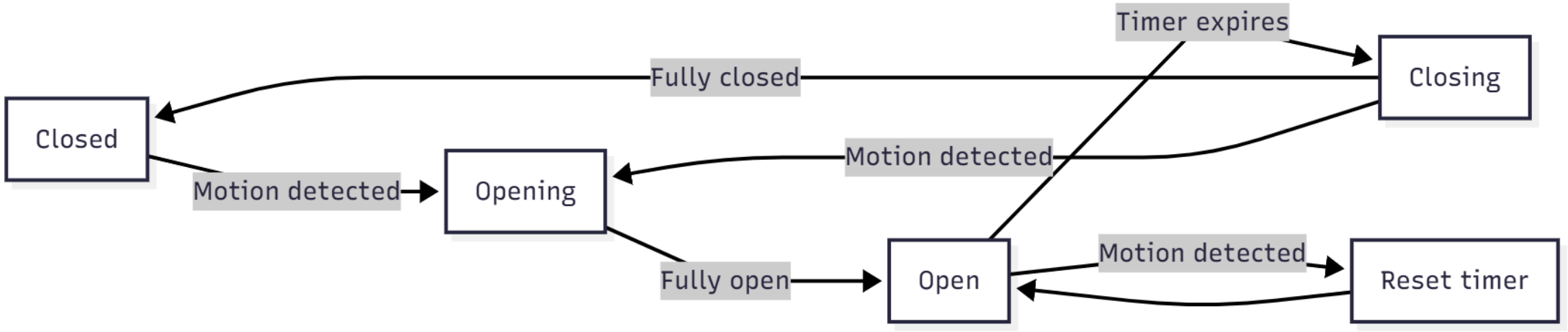
- Start State: Love-Love (0-0)
- Input Alphabet: {P1_scores, P2_scores, new_game}
- Accept States: {Game: Player1, Game: Player2}
- Transition Function: Rules of tennis scoring

Formal Definition:

- TennisAutomaton = (Q, Σ, δ, q_o, F)
- Q = {Love-Love, 15-Love, 30-Love, 40-Love, Love-15, Love-30, Love-40, 15-15, 30-15, 15-30, 40-15, 15-40, 30-30, 40-30, 30-40, Deuce, Advantage_P1, Advantage_P2, Game_P1, Game_P2}
- Σ = {P1_score, P2_score, new_game}
- q_o = Love-Love
- F = {Game_P1, Game_P2}

Automatic Door Automaton

- ❑Real-World Considerations:
- ❑Safety property: Never close on someone
- ❑Liveness property: Eventually open when motion detected
- ❑Timing constraints: How long to stay open



Linear Temporal Logic (LTL)

- Linear Temporal Logic (LTL), which is the language we use to specify properties for model checking
- It's called "linear" because it views time as a sequence of states stretching into the future.

Basic Temporal Operators:

Operator	Symbol	Meaning	Example
Always	□	From now on forever	□safe
Eventually	◇	At some future time	◇success
Next	○	In the next state	○start
Until	U	Until something happens	busy U done

Example 1: Elevator Safety Properties

-- Safety: Doors never open while moving
□ !(doors_open ∧ moving)

-- Liveness: If button pressed, elevator eventually comes
□ (button_pressed → ◇ current_floor = requested_floor)

-- Response: After arriving, doors eventually open
□ (arrived_at_floor → ◇ doors_open)

-- Non-blocking: Elevator doesn't stay forever on one floor
□ (◇ moving ∨ ◇ doors_open)

Example 2: Traffic Light Controller

-- Safety: Never green in both directions
□ !(north_green ∧ south_green)

-- Liveness: Eventually becomes green in each direction
□ (◇ north_green) ∧ □ (◇ south_green)

-- Fairness: If waiting, eventually get green
□ (car_waiting_north → ◇ north_green)

-- Sequence: Yellow always comes between green and red
□ (green → ○ yellow) ∧ □ (yellow → ○ red)

Key Takeaways: Formal Verification & Model Checking

Formal Verification

- ❑Mathematical proof of system correctness
- ❑Exhaustive verification - considers all possible scenarios
- ❑Beyond testing - proves absence of bugs rather than finding some
- ❑Used in safety-critical systems (aerospace, medical, automotive)

Model Checking

- ❑Automated technique for formal verification
- ❑Systematically explores all possible system states
- ❑Finite-state systems only (due to state explosion problem)
- ❑Provides counterexamples when properties fail