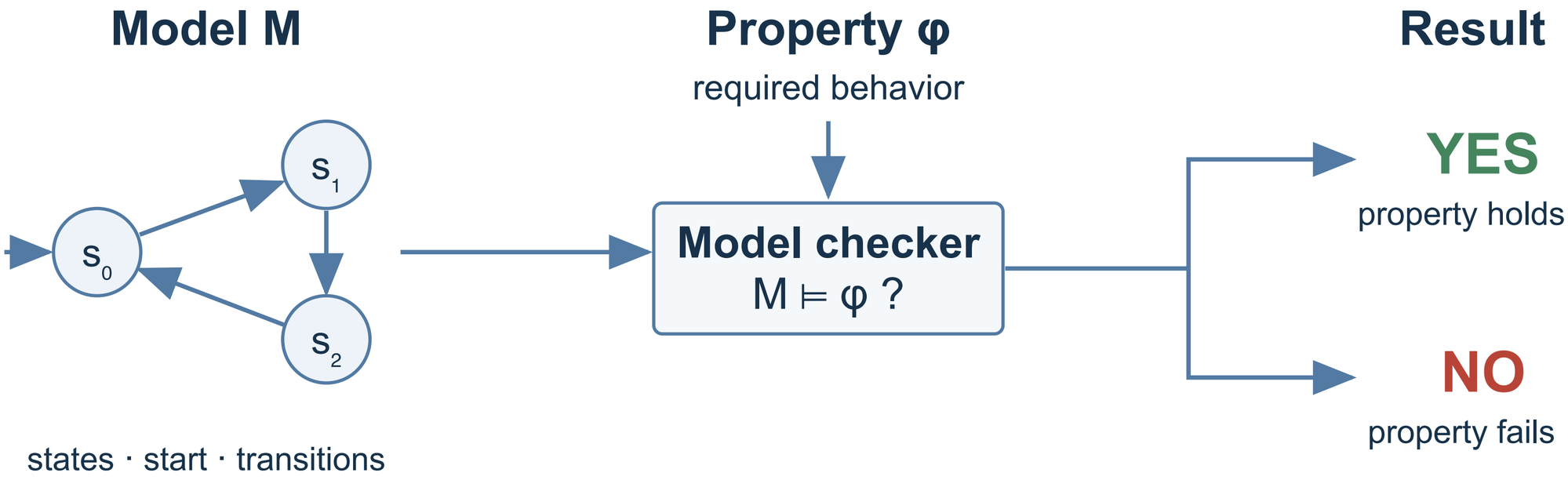
Model checking and symbolic model checking

Checking properties of executions

CMPUT 664 · Formal Verification · November 2, 2026

What is model checking?

Model checking algorithmically determines whether a system model satisfies a specified property.

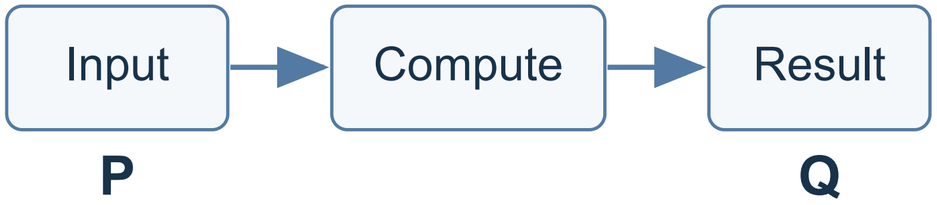


Why ongoing behavior became central

Controllers and protocols must behave correctly **while they keep interacting**.

A classic correctness question

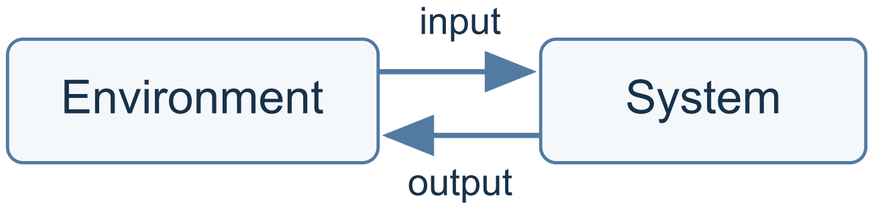
Floyd (1967) · Hoare (1969)



If it returns,
is the result correct?

Ongoing systems

Controllers · communication protocols



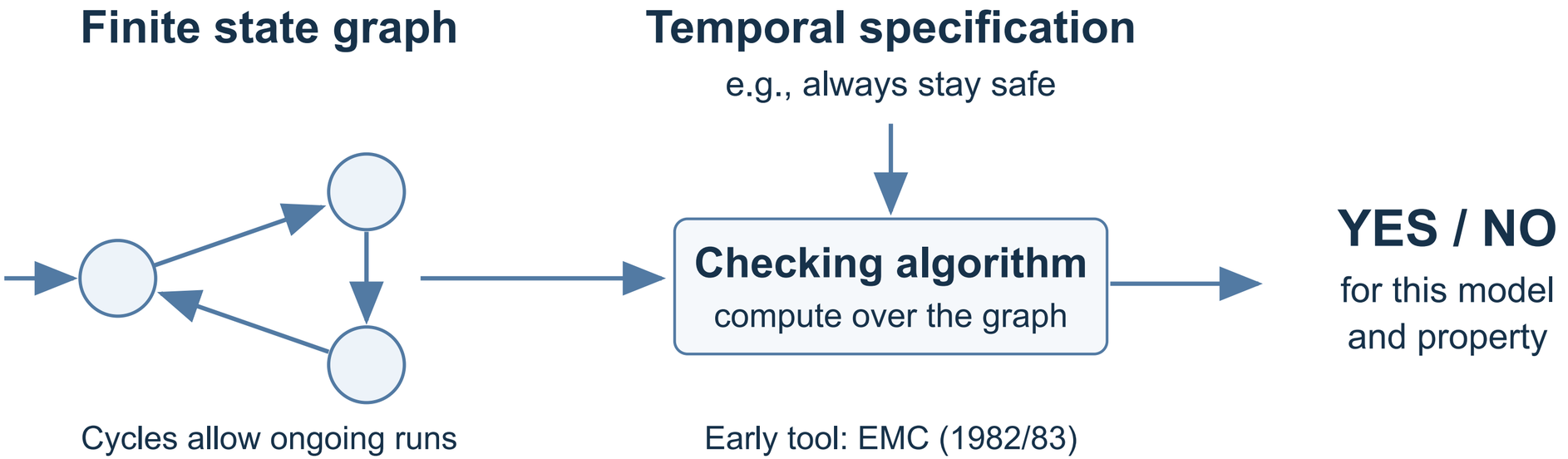
Interaction continues

Always stay safe?
Eventually respond?

Temporal logic expresses requirements over time: Pnueli applied it to concurrent programs in 1977.

1981–82: make temporal verification algorithmic

Model checking combined **temporal specifications** with algorithms on **finite state graphs**.



Finite states can describe infinitely long executions. The challenge becomes the **size of the state space**.

Where model checking is used

Each application asks whether an **allowed event sequence** can violate a requirement.

Application	Behavior model	Example question
Hardware protocols Futurebus+ / SMV	 cache states + messages	**Can two valid copies of one block disagree?**
Device drivers Windows / SLAM	 control flow + lock state	**Can a path release a lock it does not hold?**
Distributed services AWS designs / TLA+ · TLC	 messages + failures + recovery	**Can failures and recovery lose committed data?**

Two approaches to the same verification goal

Both establish that behavior satisfies a specification; they **organize the argument differently**.

Deductive verification

Program logic: e.g., Hoare logic
Tools: Dafny (verifier), Lean (proof assistant)

Program + specification

↓

Derive and prove obligations
using contracts and invariants

Proofs can cover unbounded data

Challenge
Finding useful proofs and summaries

Finite-state model checking

The classical setting studied here
Tools: e.g., SMV and TLC

Behavior model + specification

↓

Compute satisfaction
using states and transitions

Complete answer on the finite model

Challenge
Too many state combinations

Both can use SAT/SMT solvers and invariants, and can be combined.

A roadmap for the lecture

1. **Model checking:** determine whether the property holds by analyzing the state graph.
2. **Symbolic model checking:** perform the same analysis using compact descriptions of sets of states.
3. **Counterexample search:** encode a bounded execution as a SAT/SMT query and determine what its answer proves.

A program that keeps running

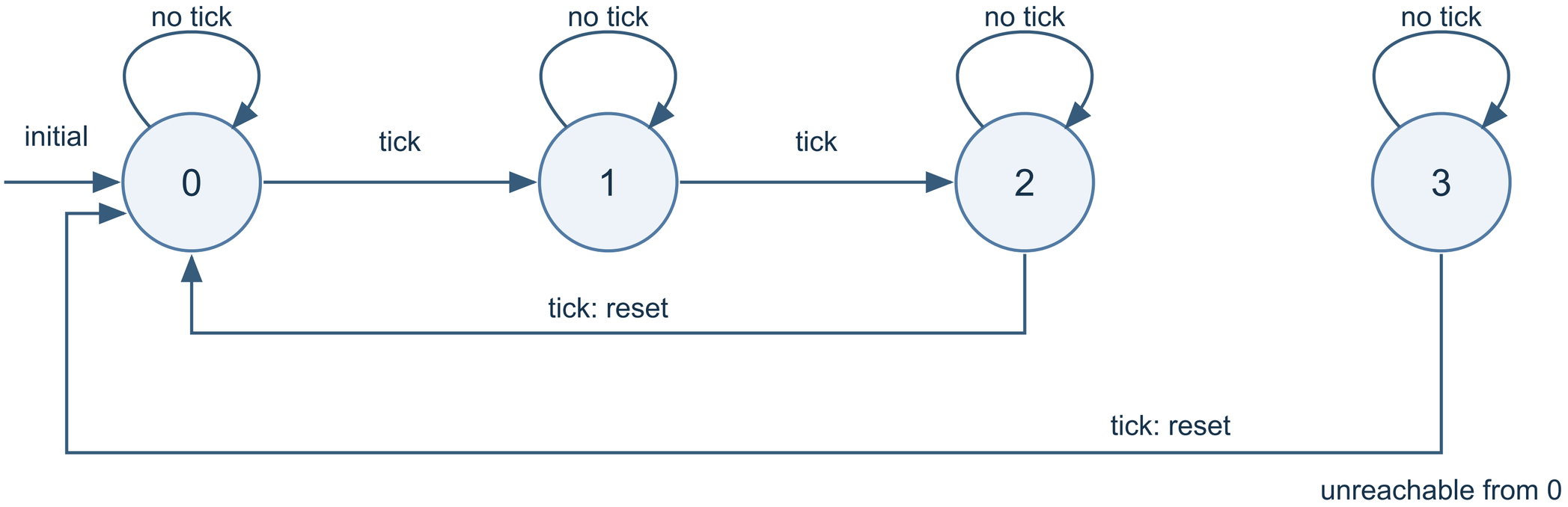
A two-bit counter has four possible values, but the loop can run indefinitely.

```
1 x : unsigned 2-bit := 0;
2 while true do
3     tick := inputBoolean();
4     if tick then
5         if x < 2 then x := x + 1
6         else          x := 0
```

Safety: always within $0, 1, 2$? **Progress:** eventually reach 2?

The transition system comes from the code

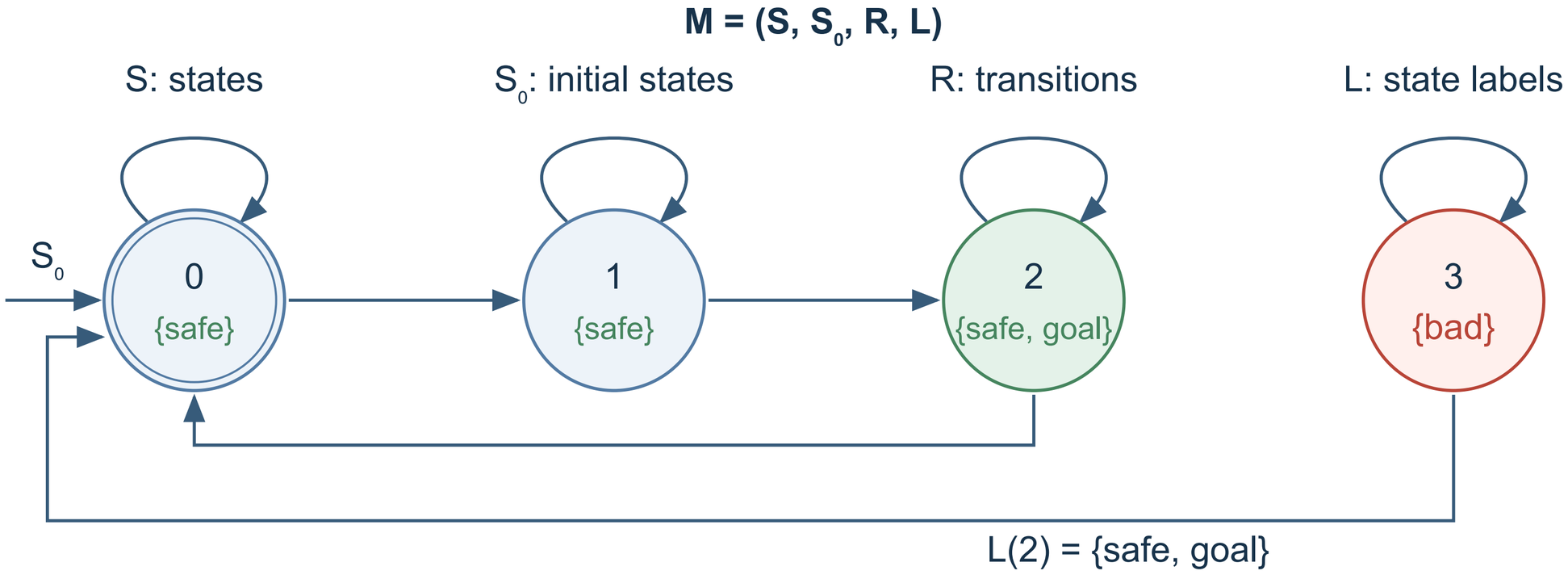
Each node records x at the start of an iteration. Each arrow represents one iteration.



Both choices are possible at every step. The checker must account for every tick/no-tick sequence.

The graph is a Kripke structure

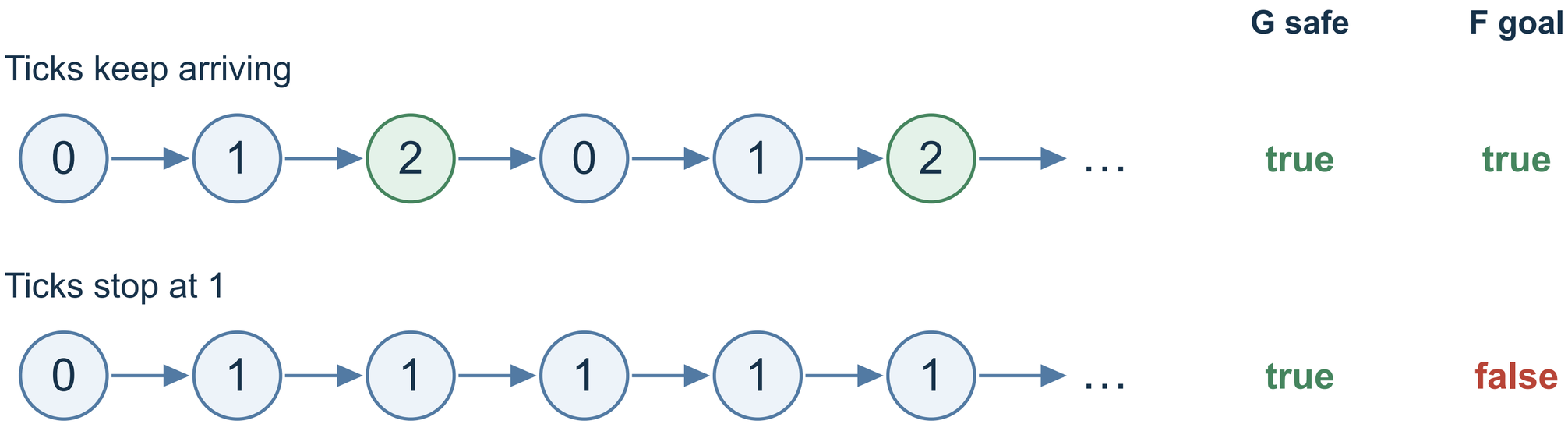
A Kripke structure is a transition graph with facts attached to each state.



$$\text{safe} \equiv x \leq 2, \qquad \text{goal} \equiv x = 2, \qquad \text{bad} \equiv x = 3.$$

Temporal logic specifies behavior over time

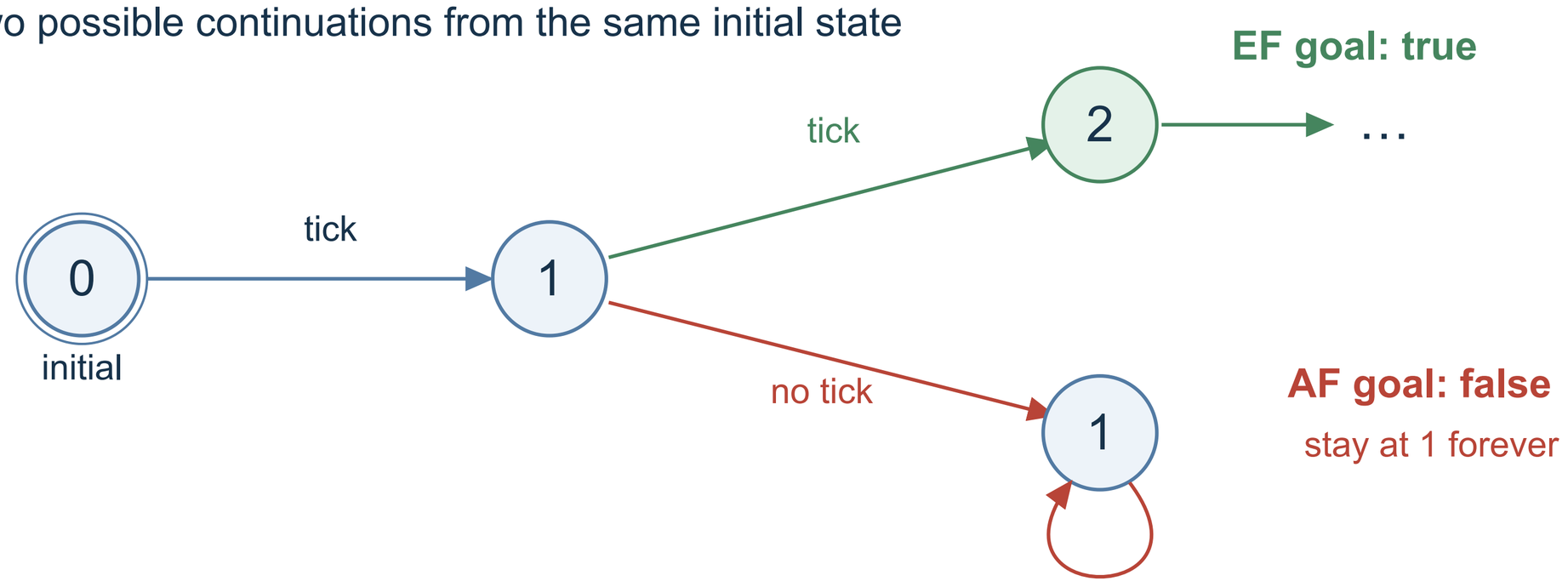
In **linear temporal logic (LTL)**, G means always and F means now or later.



Both executions stay safe, but only the first reaches the goal.

Possible progress and guaranteed progress differ

Computation Tree Logic (CTL): E = **some path**; A = **every path**.



At state 0: AG safe holds, EF goal holds, and AF goal fails.

Counterexamples depend on the property

Buggy variant: change the guard from $x < 2$ to $x \leq 2$.

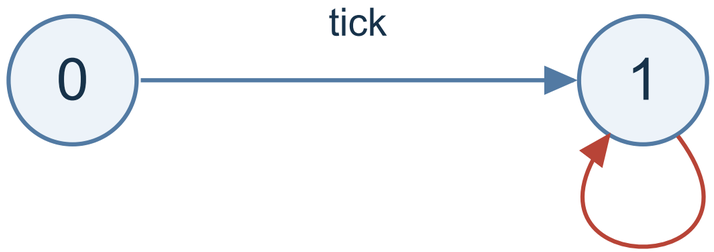
Safety failure: buggy guard



Finite prefix

refutes AG safe

Progress failure: correct code



no tick forever

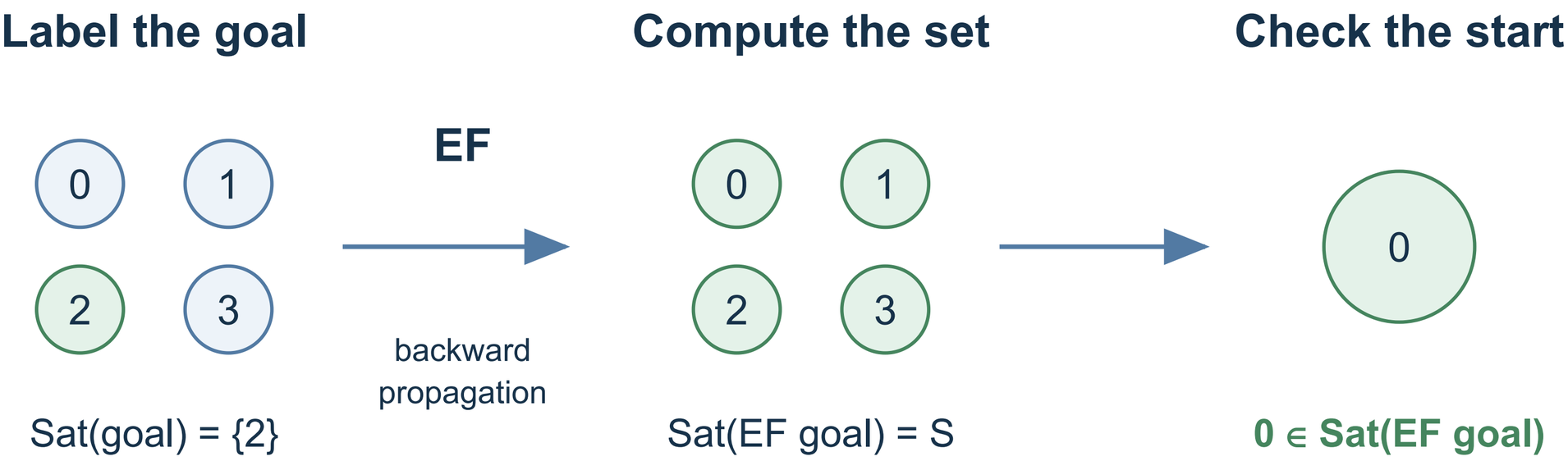
Prefix + loop

refutes AF goal

A finite wait may still be followed by progress. The displayed prefix and loop describe an execution that never reaches 2.

CTL checking computes where a property holds

Compute a state set for each part of the formula, then check the initial state.



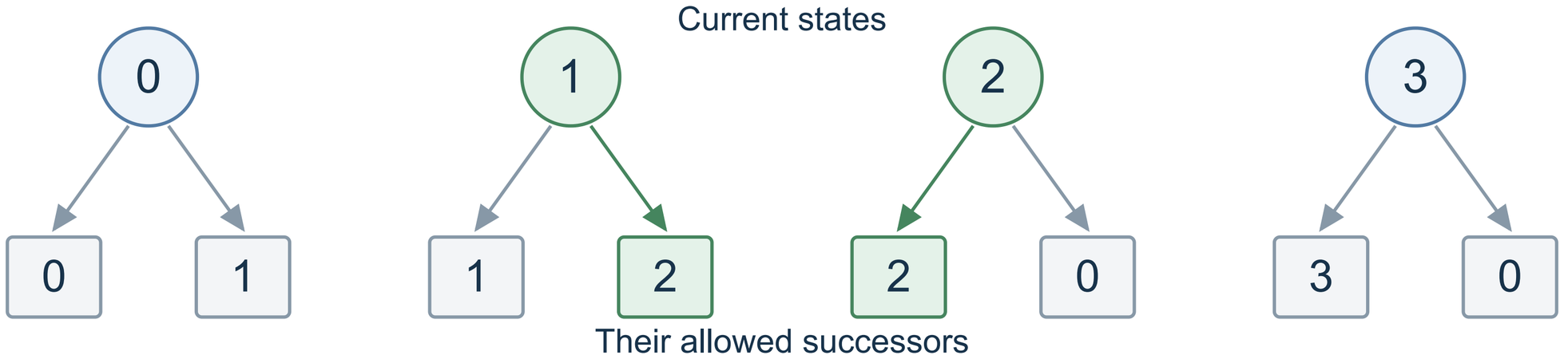
$S = \{0, 1, 2, 3\}$ is the state universe; $\mathrm{Sat}(\varphi)$ is the subset satisfying φ.

One step backward: some successor or every successor?

$\mathrm{Pre}_E(Z)$: **some successor in** Z. $\mathrm{Pre}_A(Z)$: **every successor in** Z.

Target Z = {2}

Every source has a successor outside Z



Some successor in Z: {1, 2}

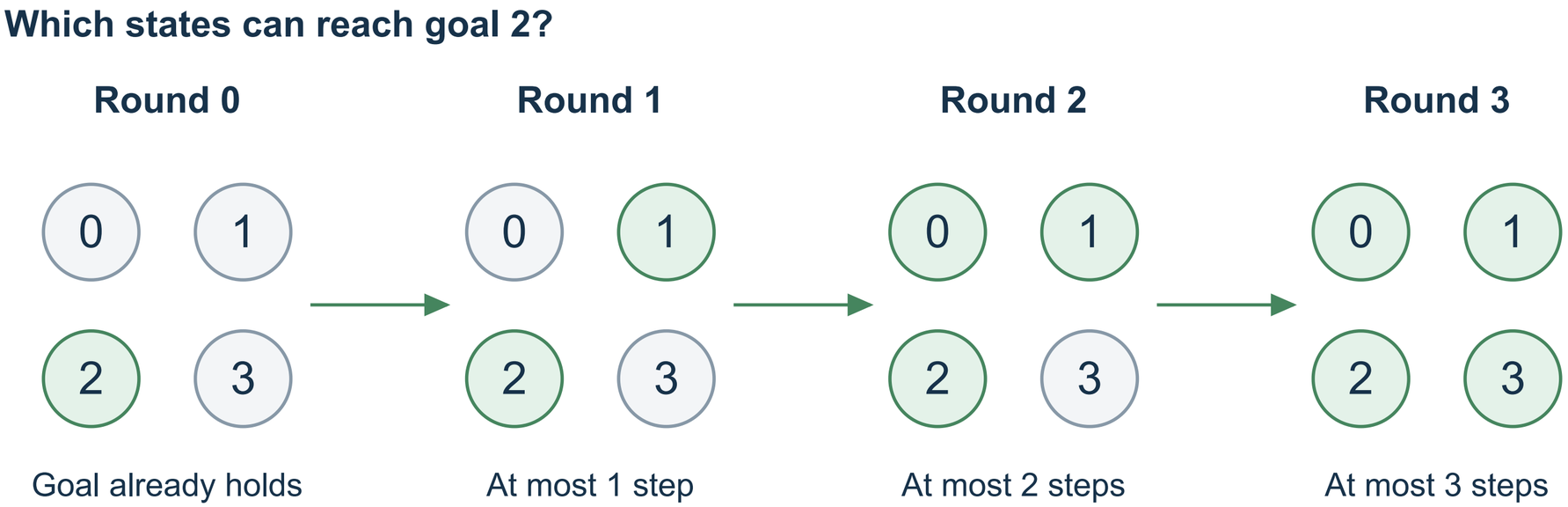
Every successor in Z: none

EF uses the some-successor step; AF uses the every-successor step. Repeat these steps to reason about eventuality.

Eventually on some path: grow backward

Start at the goal states. Add any state with a successor already included.

$$Z_0 = \{2\}, \qquad Z_{i+1} = Z_0 \cup \mathrm{Pre}_E(Z_i).$$

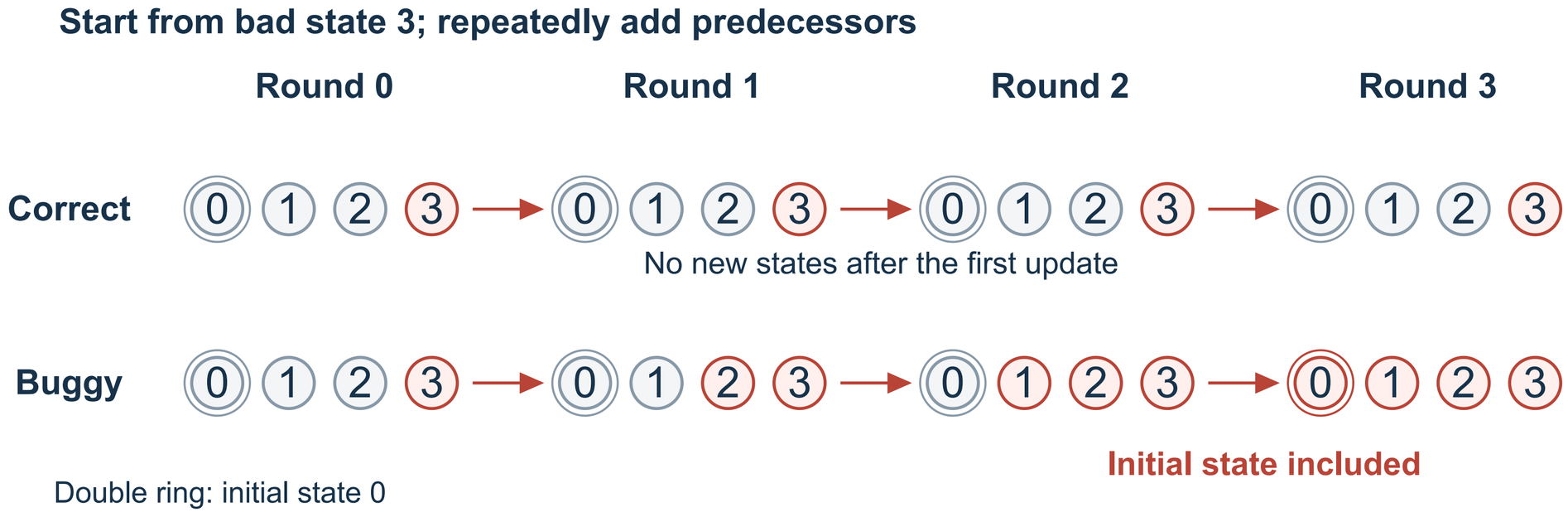


The set only grows within a finite universe, so it must reach a **fixed point**. Here 0 is included, so EF goal holds initially.

Safety reduces to the same reachability computation

A safety violation exists exactly when the initial state can reach a bad state.

$$AG\,\text{safe} \equiv \neg EF\,\text{bad}, \qquad \text{bad} \equiv x = 3.$$



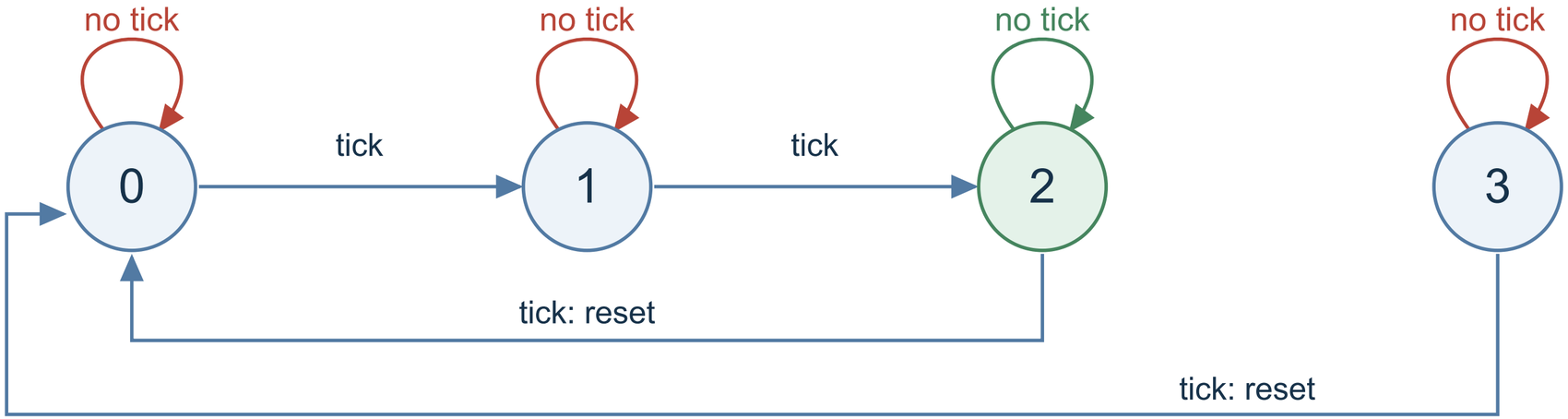
Only the buggy model's bad-reachability set contains initial state 0.

Eventually on every path: require all successors

Keep the goal states. Add another state only when **every successor** is already included.

$$Y_0 = \{2\}, \qquad Y_{i+1} = Y_0 \cup \mathrm{Pre}_A(Y_i).$$

Goal 2 stays marked; no other state can be added



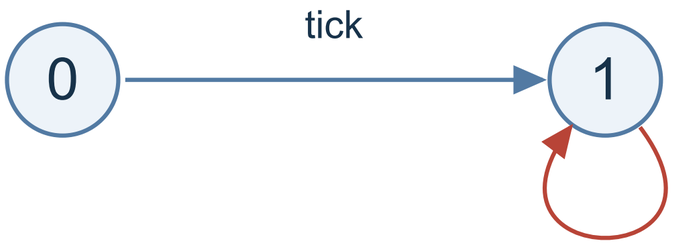
Y0 = {2} → Y1 = {2} : fixed

The set stays $\{2\}$. From initial state 0, idling forever prevents guaranteed arrival at the goal.

The environment assumption changes progress

If ticks occur **infinitely often**, the counter must eventually reach 2.

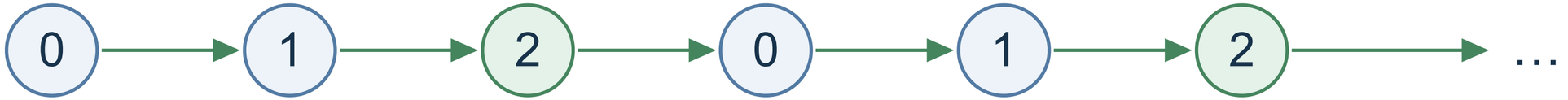
Original model: permanent idling is allowed



no tick forever

Excluded by assumption

Infinitely many ticks: the counter keeps traversing the cycle

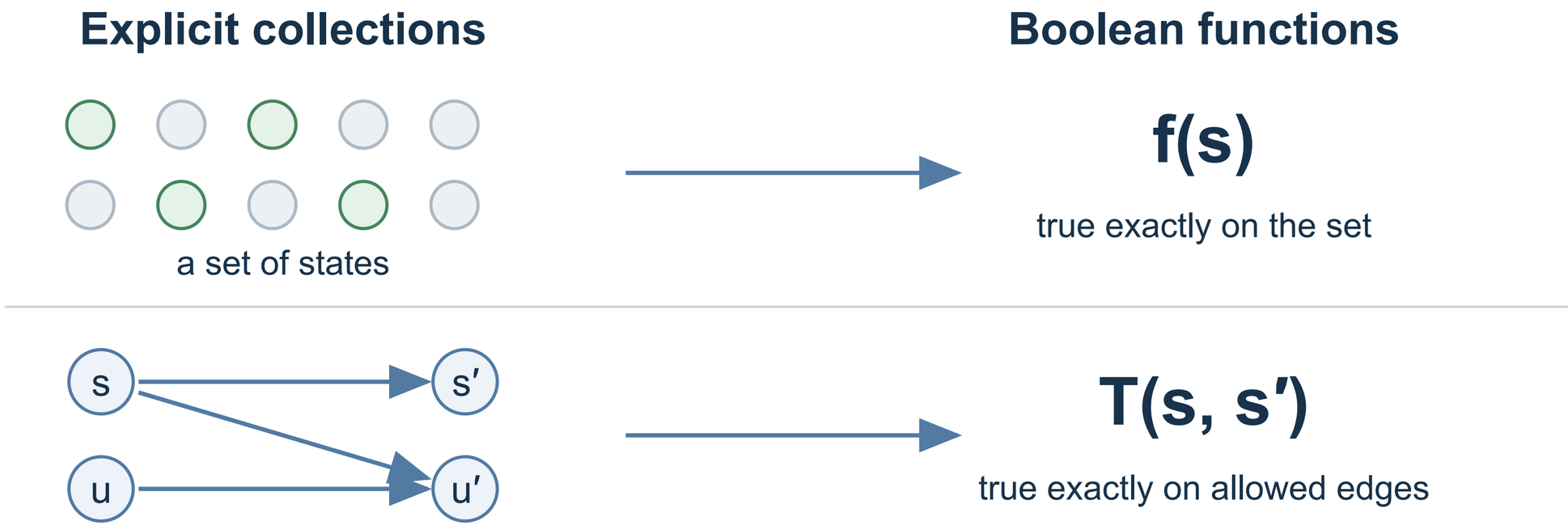


Finite stretches of idle steps are still allowed.

Requiring infinitely many ticks is a **fairness assumption**. The earlier AF check included executions that idle forever.

Symbolic model checking changes the representation

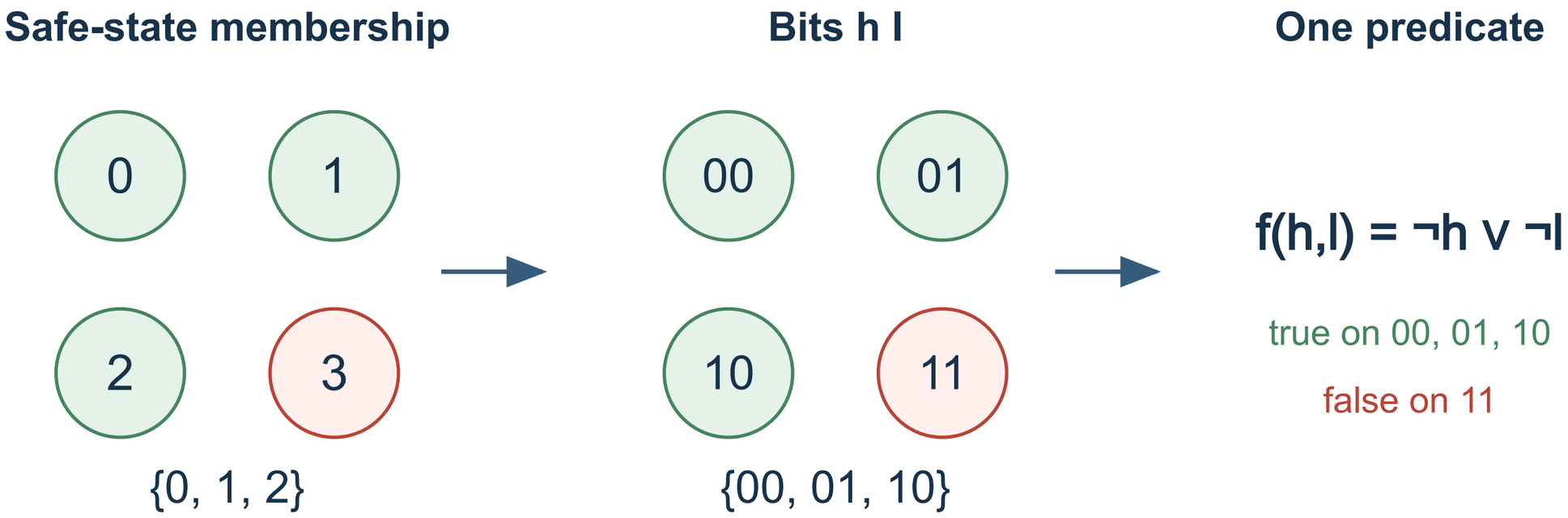
n state bits allow 2^n states. Explicit state sets can become too large to store.



The symbolic approach (Burch et al., 1990) runs these algorithms on **Boolean descriptions** of sets and transitions.

State sets become Boolean functions

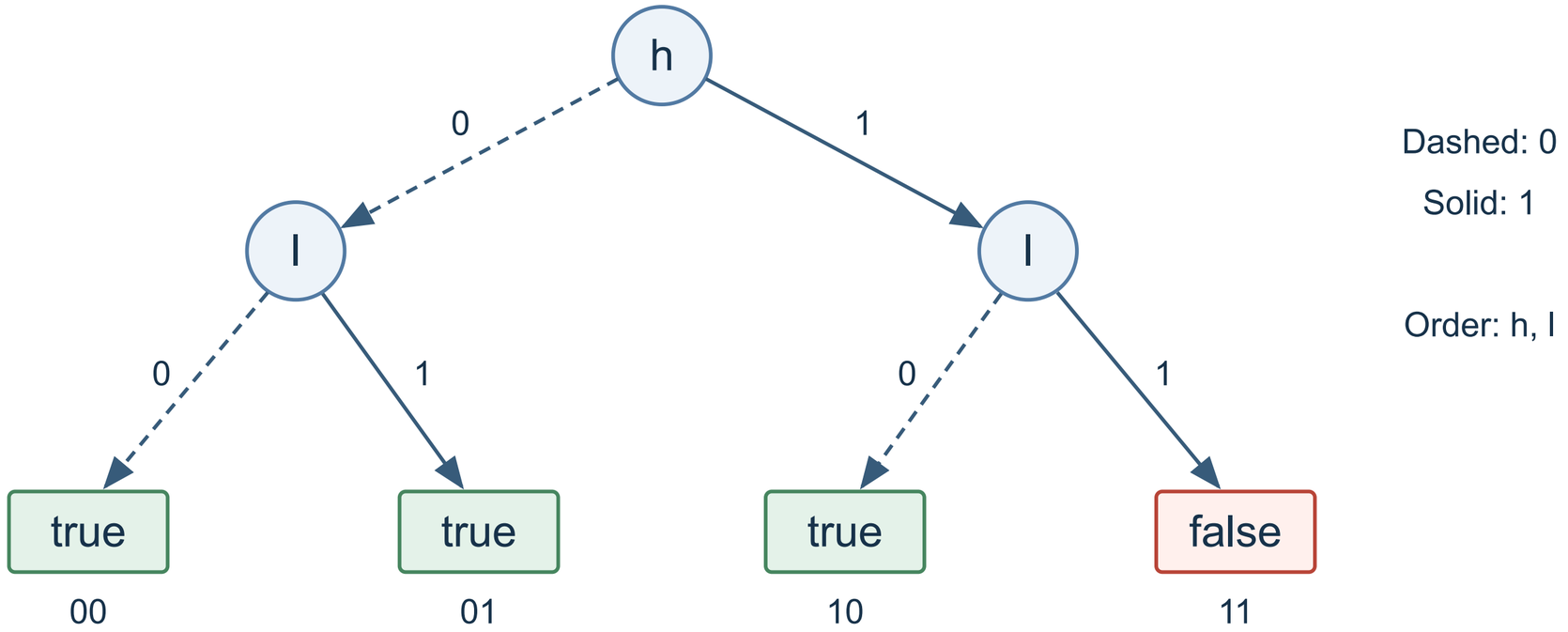
Write $x = 2h + l$. A characteristic function is true exactly on the set's members.



$$\text{Init} = \neg h \wedge \neg l, \quad \text{goal} = h \wedge \neg l, \quad \text{bad} = h \wedge l.$$

Binary decision diagrams

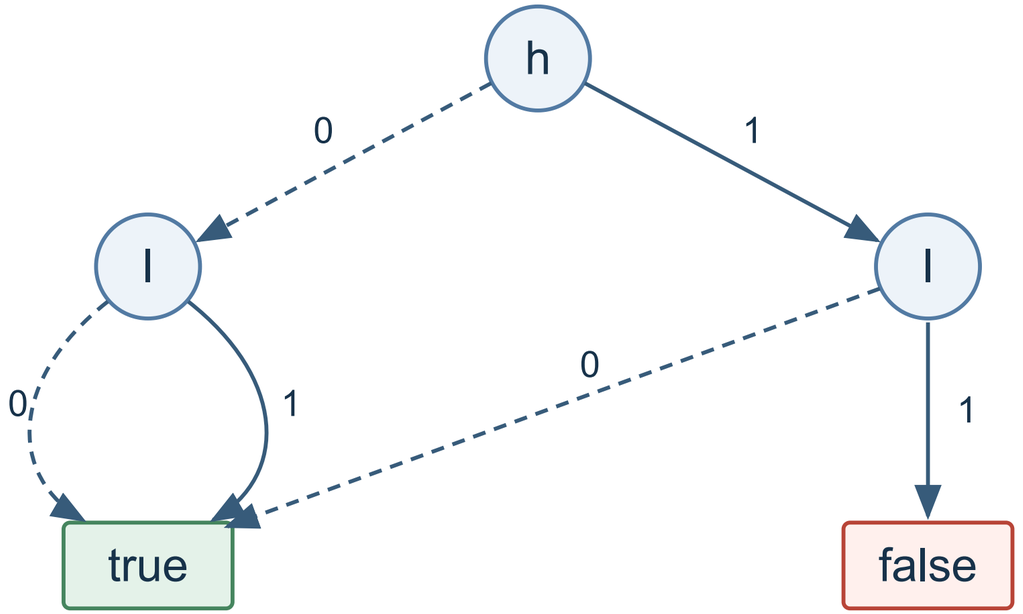
A **binary decision diagram (BDD)** evaluates $f(h, l) = \neg h \vee \neg l$ by testing the bits.



BDD edges choose bit values, so a path tests membership of one state in the safe set.

Reduction 1: share identical subgraphs

The three `true` leaves compute the same function. Merge them into one shared node.



Dashed: 0

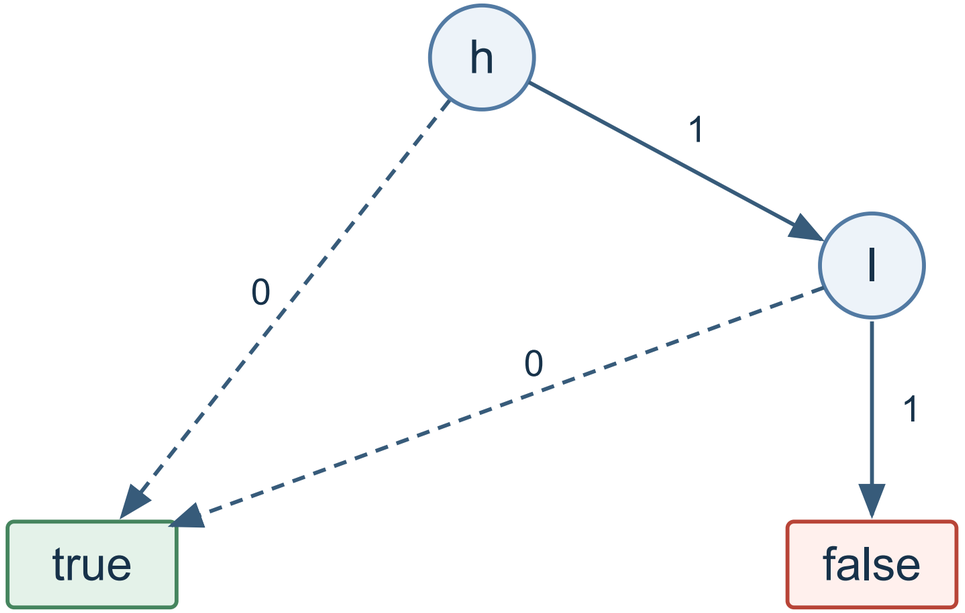
Solid: 1

Order: h, l

Both outcomes of the left l test now lead to that same node.

Reduction 2: remove a redundant test

Below $h = 0$, either value of l leads to `true`. Bypass the test.



Dashed: 0

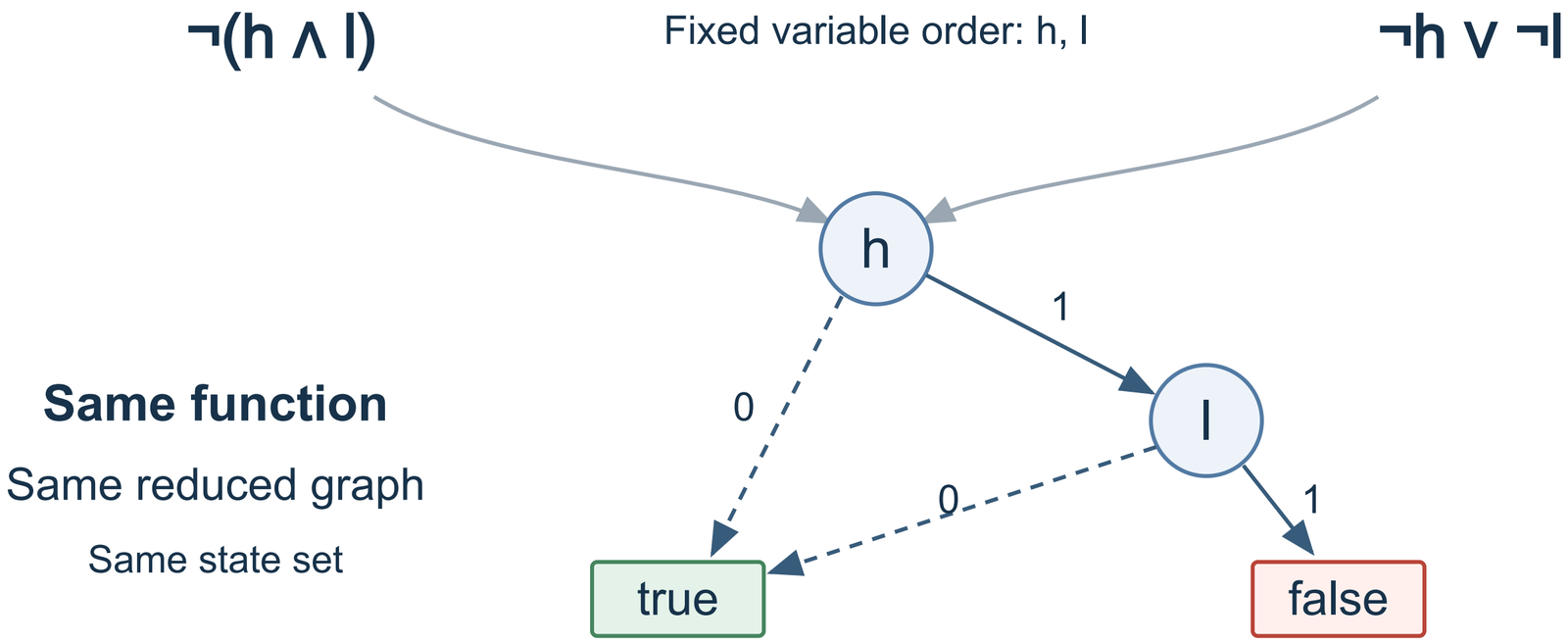
Solid: 1

Order: h, l

Skipping l leaves it free: the $h = 0$ path represents both 00 and 01.

What an ordered, reduced BDD gives us

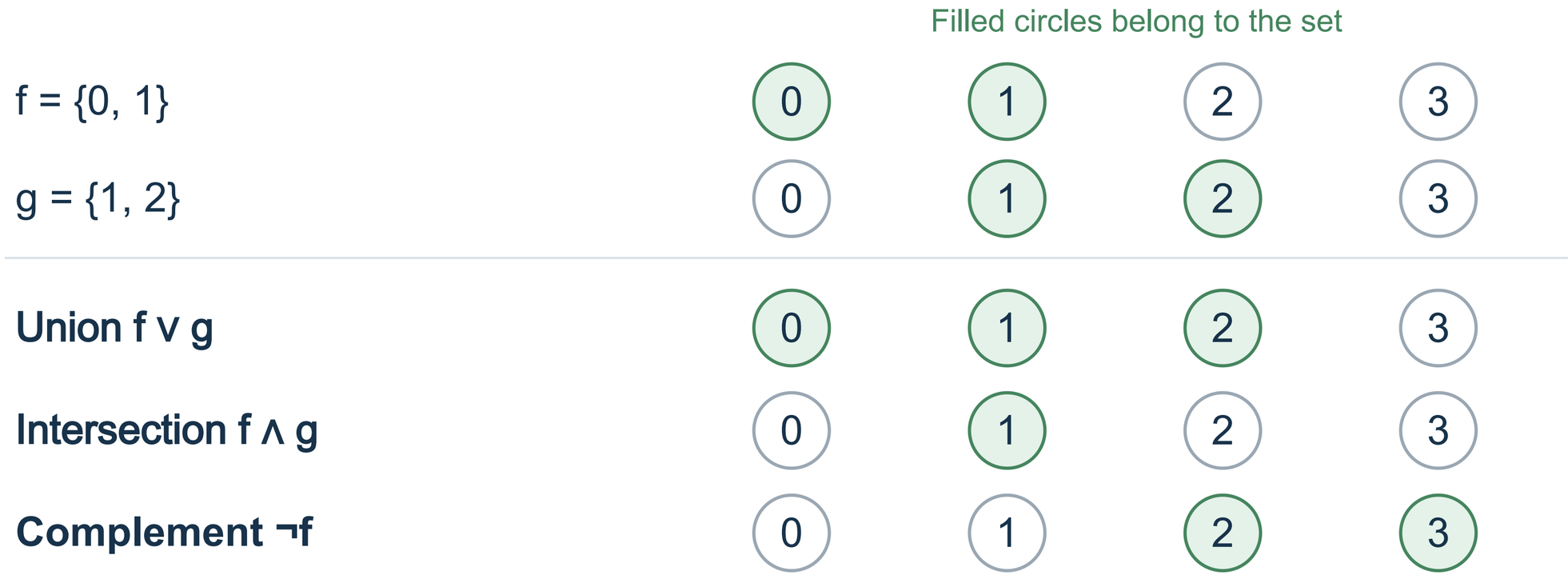
With a **fixed variable order**, equivalent functions have the same reduced BDD.



When an update returns the same reduced BDD, the represented set has stopped changing: a **fixed point**.

Set operations become Boolean operations

Union, intersection, and complement become $\vee$, $\wedge$, and $\neg$.

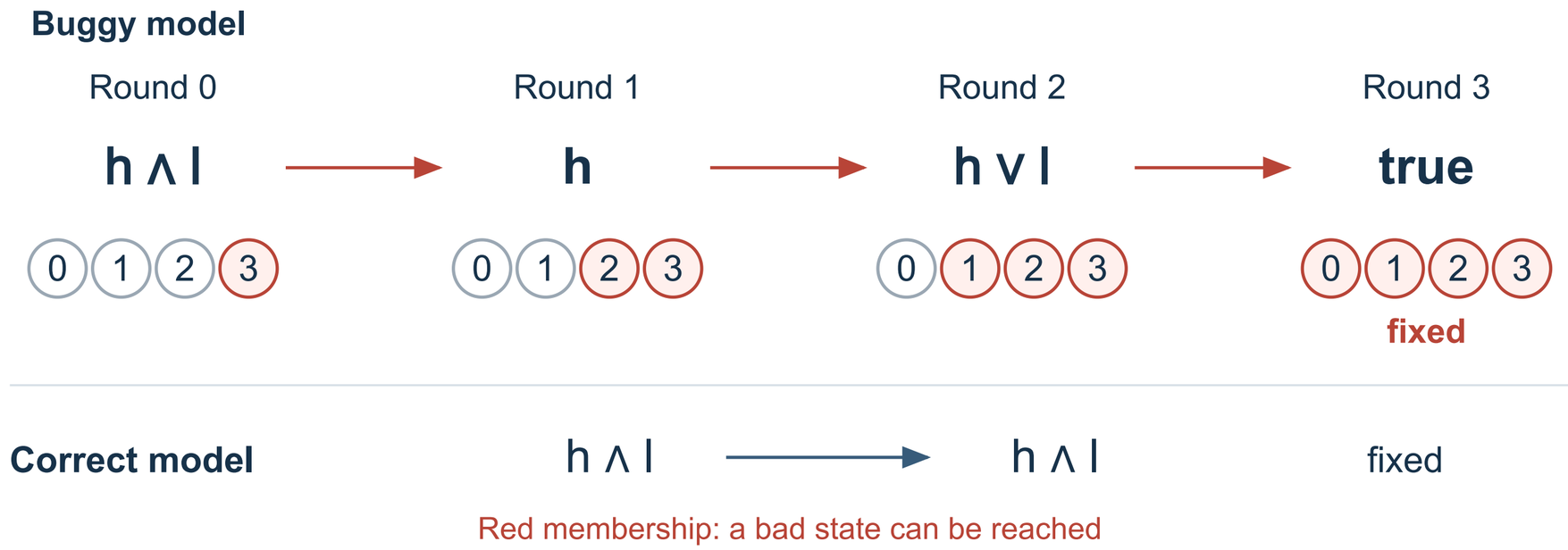


These Boolean operations implement the set updates used by the checker.

The same temporal algorithm runs on BDDs

Repeat backward reachability, now storing each state set as a BDD.

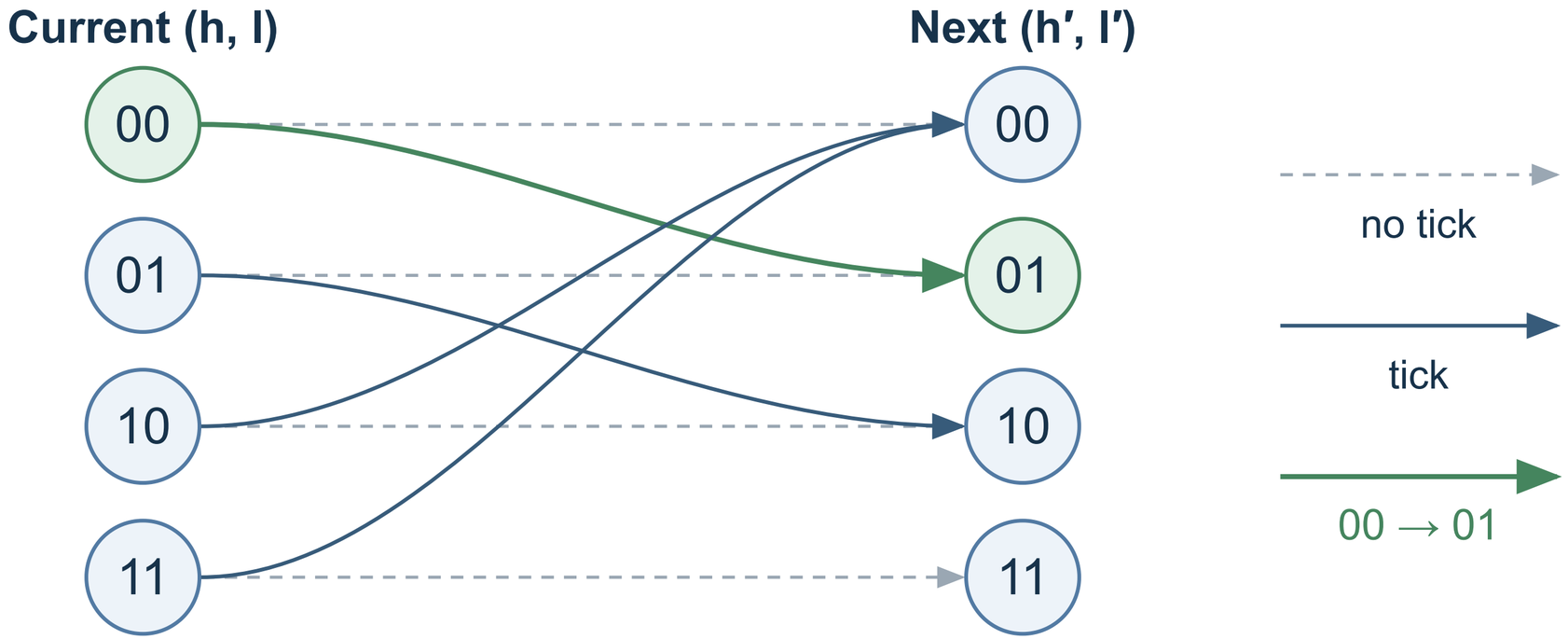
$$Z_0 = \text{bad}, \qquad Z_{i+1} = \text{bad} \vee \text{Pre}_E(Z_i).$$



Intersect with $\text{Init} = \neg h \wedge \neg l$: only the buggy result includes 0.

Symbolic predecessors need the transition relation

$T(s, s')$ represents the Kripke relation R: it is true exactly when $(s, s') \in R$.



$$T(x, x') \equiv (x' = x) \vee (x < 2 \wedge x' = x + 1) \vee (x \geq 2 \wedge x' = 0).$$

Symbolic predecessors perform the same graph operation

Keep the sources of edges that lead into the target set.

$$\mathrm{Pre}_E(Z)(h,l) = \underbrace{\exists h', l'}_{\text{eliminate next bits}} . \underbrace{T(h,l,h',l') \wedge Z(h',l')}_{\text{allowed edge into the target}} .$$

Existential elimination combines the two branches of a Boolean variable:

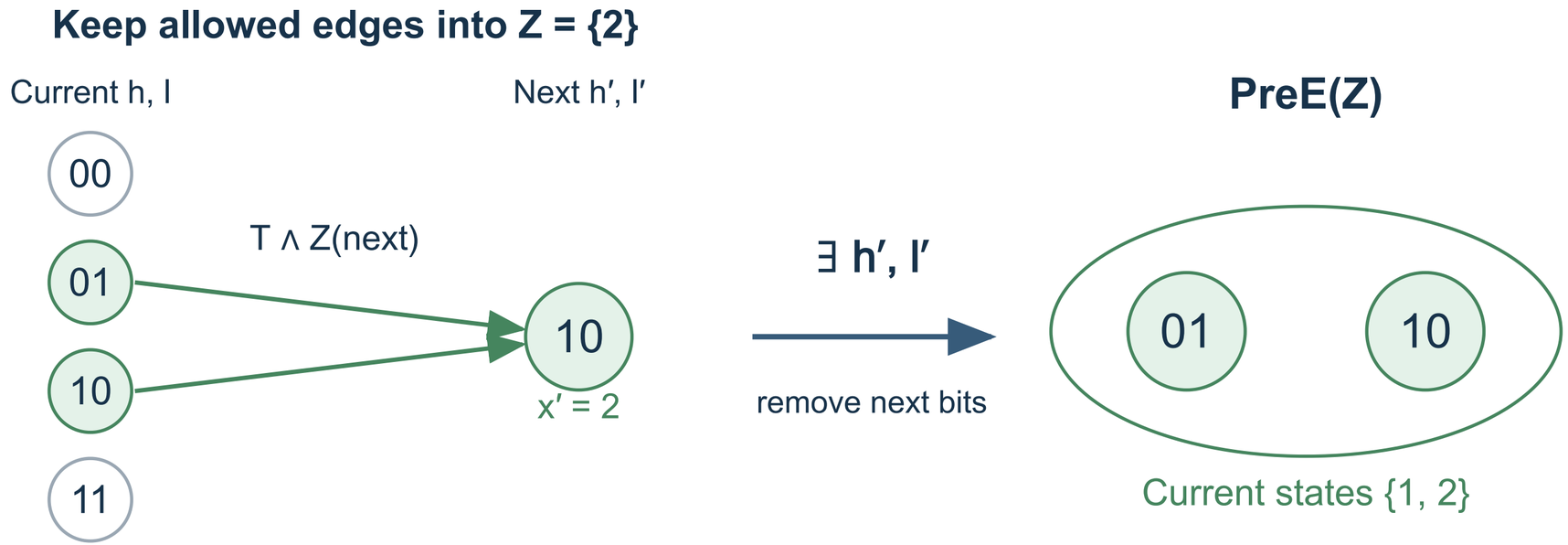
$$\exists v.\ f = f[v := 0] \vee f[v := 1].$$

The result is a function of **current-state bits only**.

Work one symbolic predecessor calculation

Keep the edges into state 2, then retain their starting states.

$$\exists h', l'.\ T(h, l, h', l') \wedge h' \wedge \neg l' = (\neg h \wedge l) \vee (h \wedge \neg l).$$

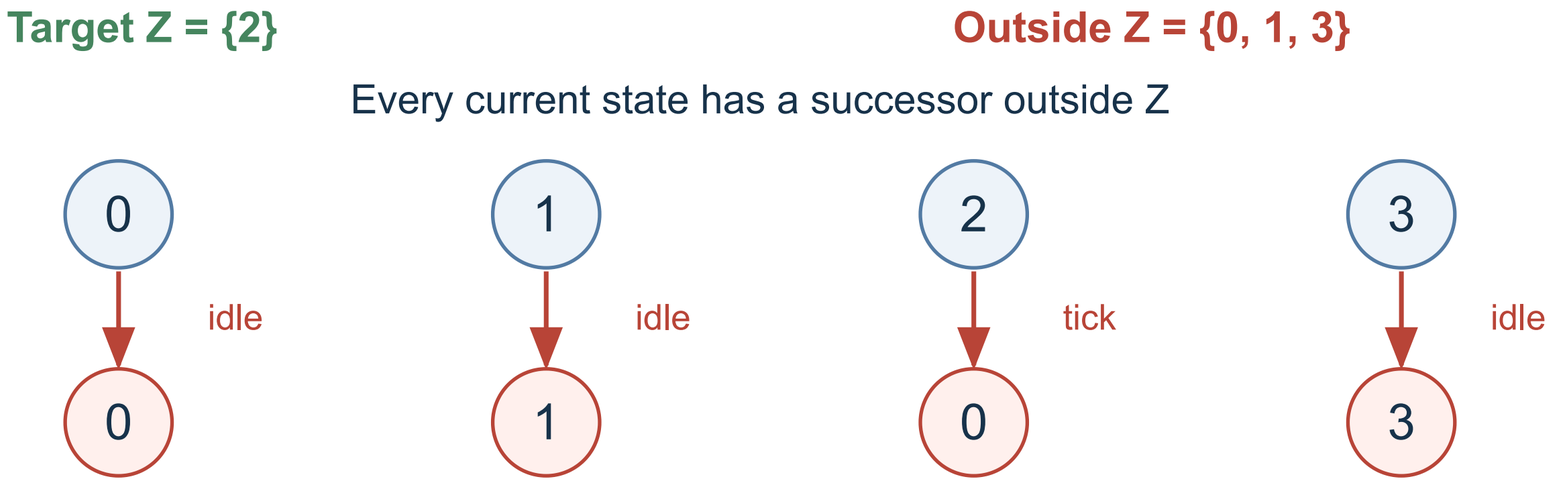


The BDD represents $\{1, 2\}$, exactly the predecessor set obtained from the graph.

Universal properties use the same BDD operations

Every successor is in Z exactly when **no successor is outside** Z.

$$\mathrm{Pre}_A(Z) = \neg\,\mathrm{Pre}_E(\neg Z).$$

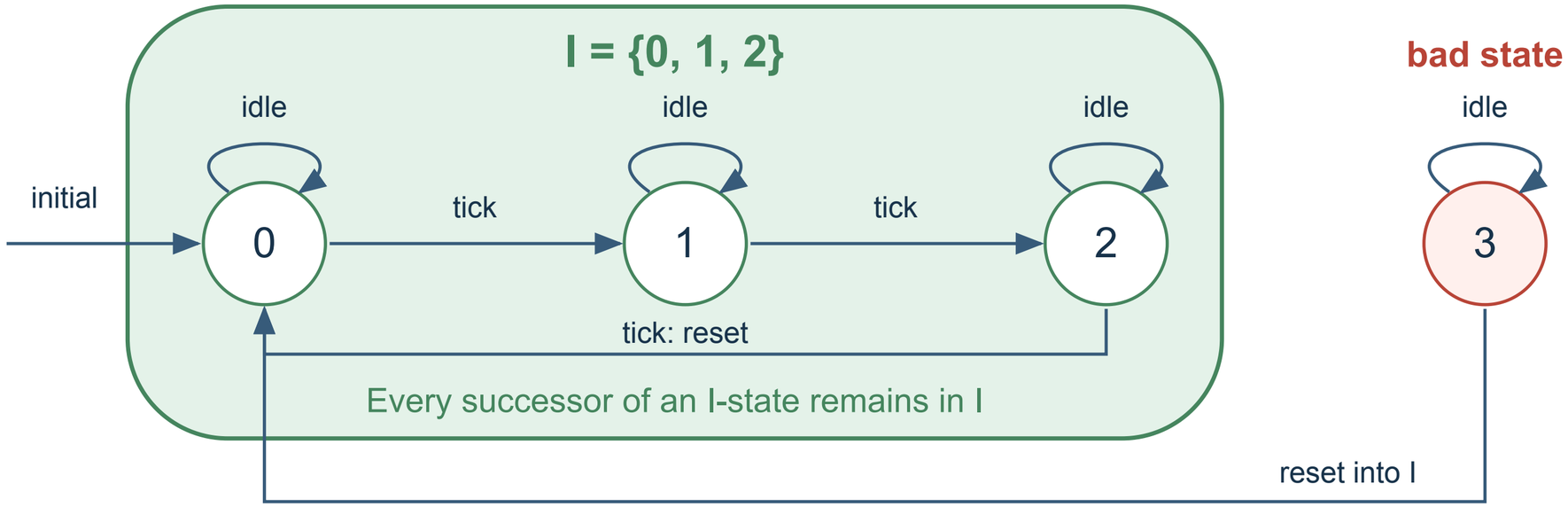


Starting at $Y_0 = \{2\}$, the AF update adds nothing. The symbolic result still excludes initial state 0.

Backward reachability produces a safety invariant

In the **correct model**, take the states from which badness is unreachable.

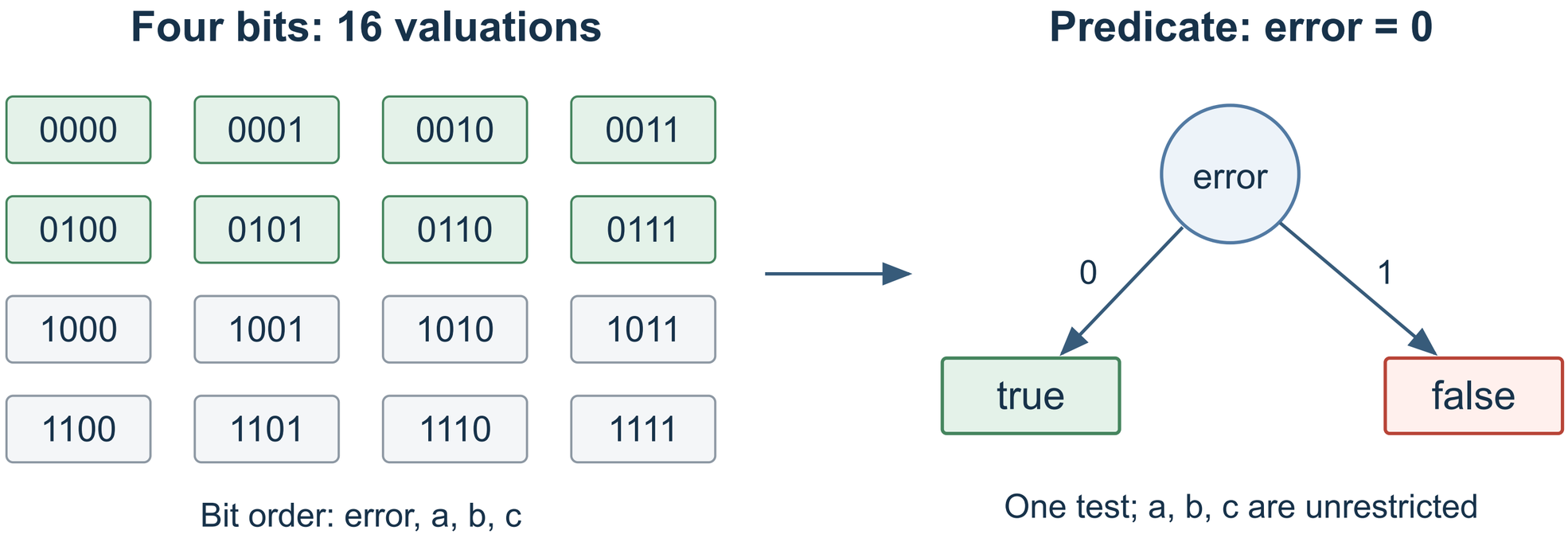
$$I = S \setminus \mathrm{Sat}(EF\ \mathrm{bad}) = \{0, 1, 2\}.$$



Inductive invariant: contains 0, is closed under transitions, and excludes 3. Safety holds after any number of steps.

BDDs reduce representation cost when structure allows it

For a four-bit encoding (error, a, b, c), consider the set $\text{error} = 0$.



Here one test represents eight valuations. **Compression depends on structure**; BDDs can still require exponential space.

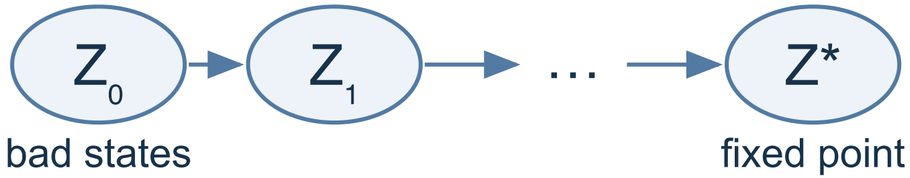
Bounded model checking searches for a bad execution

When state-set operations are costly, a **bounded execution query** offers another way to search for a bug.

Can an execution reach a bad state?

Fixed-point checking

Backward state-set iteration



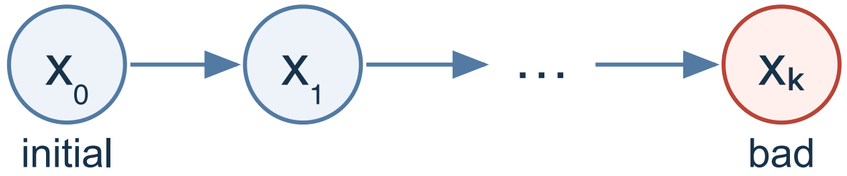
Shown here: BDD operations

Initial state $\in Z_i$: bug found

Initial state $\notin Z^*$: safe

Bounded model checking

Execution of exactly k transitions



Execution formula → SAT/SMT

SAT: a bad execution

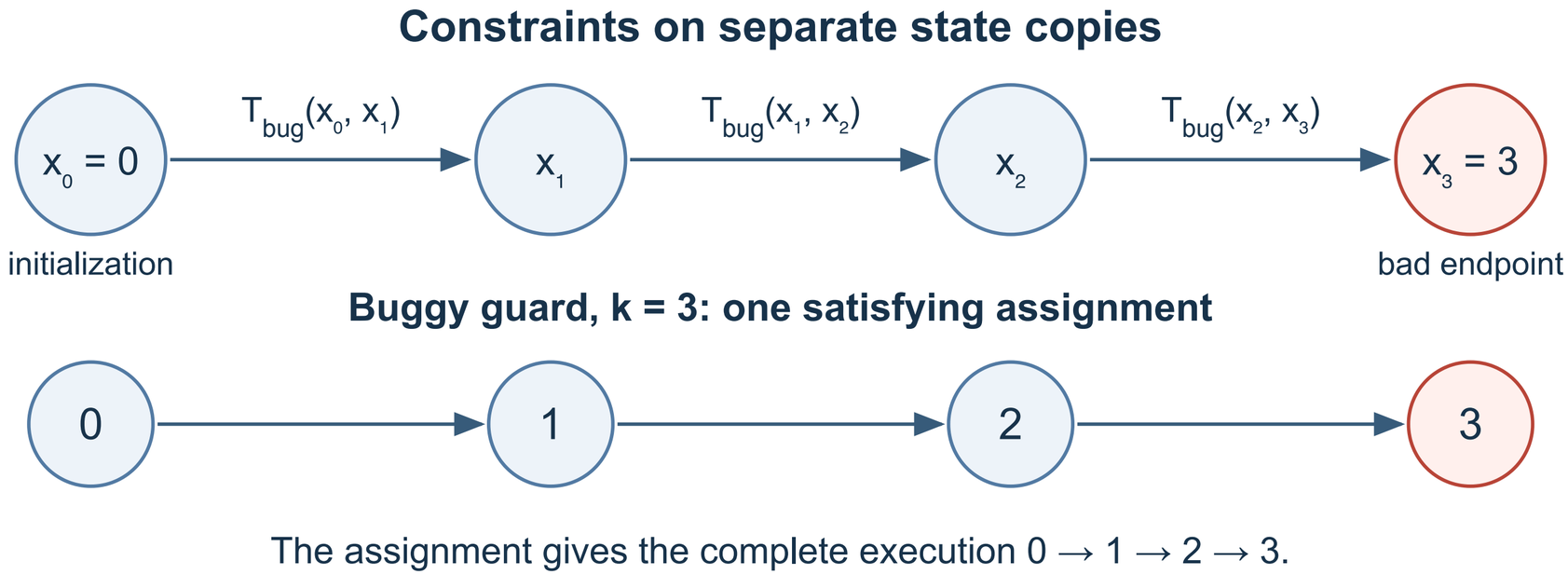
UNSAT: none of exactly k steps

The model checker chooses the bound and constructs the query; **SAT/SMT solves the formula.**

The bounded query encodes the whole execution

With buggy guard $x \leq 2$, **solve for a length-k execution ending at** 3.

$$B_k \equiv (x_0 = 0) \wedge \bigwedge_{i=0}^{k-1} T_{\text{bug}}(x_i, x_{i+1}) \wedge (x_k = 3).$$



The assignment gives the complete execution $0 \to 1 \to 2 \to 3$.

What does bounded checking establish?

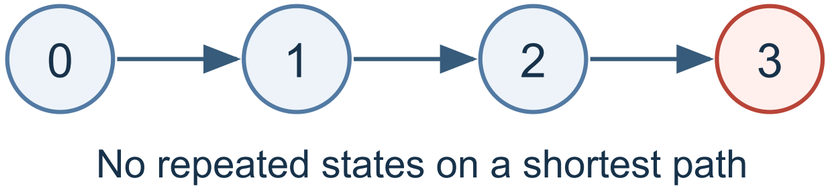
A bounded UNSAT answer covers **only the encoded lengths.**

Buggy counter: a larger bound reveals the violation



A separate completeness argument:

N states ⇒ shortest bad path ≤ N − 1



A shortest bad path in an N-state model uses at most $N-1$ transitions. UNSAT at **every depth** $0, \ldots, N-1$ proves safety.

The model-checking workflow revisited

The checking algorithm and its stopping argument determine coverage.

Checking algorithm	Implementation shown	Coverage
Fixed-point state-set computation	Explicit sets or BDD operations	All execution lengths once the set stabilizes
Bounded execution search	SAT/SMT queries	Encoded lengths; safety requires a completeness argument

SAT/SMT can also support **unbounded proofs** by helping construct an inductive invariant.