



VANDERBILT UNIVERSITY
College of Connected Computing

Day 2 — Model Checking with nuXmv

FMAIV: Formal Methods & AI-Assisted Verification

Taylor Johnson — Associate Professor of Computer Science, Computer Engineering
& Electrical Engineering; Associate Dean for Graduate Education, College of
Connected Computing · taylor.johnson@vanderbilt.edu · taylortjohnson.com

Ben Wooding — Postdoctoral Scholar, Institute for Software Integrated Systems ·
ben.wooding@vanderbilt.edu · woodingben.com

Vanderbilt University

Day 2 of 4

Day 2 — Model Checking

Reactive systems, temporal logic, and nuXmv

Run it live (no install): Codespaces (full toolset) or Colab — today's notebook
02_day2_model_checking . All materials: <https://github.com/ttj/fmaiv>

Where we are

Day 1, in Z3, we asked: *can the counter reach $x = 11$ within N steps?*

- Answer: **unsat** — but only for the N we could afford to enumerate.
- Bounded model checking **refutes** short counterexamples; it does not **prove** safety.

Today: ask the same question for all N at once, via a model checker.

Today's roadmap

Block	Topic
L1	Reactive systems + the SMV input language
L2	Temporal logic: CTL and LTL
L3	Model-checking algorithms + live nuXmv

Running example throughout: the same `counter.smv`.

"nuXmv" vs "NuSMV" — what you'll actually run

The slides say **nuXmv**; the course autograder runs **NuSMV 2.6.0**. They share the SMV language and the `is true / is false` verdict format, so everything here works in both.

Capability	NuSMV 2.6.0 (what you run)	nuXmv (superset)
Symbolic (BDDs, binary decision diagrams) model checking	✓	✓
Bounded model checking (SAT)	✓	✓
k-induction for invariants	✓	✓
IC3 / PDR, infinite-state (SMT)	—	✓

We *mention* the nuXmv-only features but never require them.

Learning objectives for Day 2

By the end of today you will be able to:

- Model a reactive system as a synchronous transition system and write it in SMV.
- Read and write **CTL** (computation tree logic) and **LTL** (linear temporal logic) properties, and say which can express what.
- Explain the trade-offs among explicit-state, symbolic (BDD, binary decision diagram), and bounded model checking.
- Run **nuXmv** on an SMV file and read an **INVARSPEC** / **LTLSPEC** / **CTLSPEC** verdict — including a counterexample trace.

L1 — Reactive systems and SMV

Two kinds of computation

Functional (transformational)	Reactive
input → compute → output → halt	runs forever, step by step
a compiler, <code>sort(list)</code>	a thermostat, a traffic light, a CPU
correctness = right output	correctness = right behavior <i>over time</i>

Verification of reactive systems is about **infinite behaviors**, not a single answer.

Synchronous reactive components

The model behind SMV: a component that, each **round**, reads its inputs and updates its state.

- **Inputs** — values the environment chooses each round (our `press`).
- **State variables** — the component's memory (`mode`, `x`).
- **Initialization** — the allowed initial states.
- **Reaction** — a relation from (current state, inputs) to next state.

Everything happens in lock-step on a global clock: read inputs, compute next state, repeat.

The synchrony hypothesis

We pretend each reaction is **instantaneous**: the next state is computed before the next input arrives.

- Real hardware/software takes time; we abstract that away.
- Justified when the system is fast relative to its environment.
- Lets us model time as a sequence of discrete rounds — exactly what a model checker enumerates.

Where reactive components show up

- **Automobiles** — engine control, anti-lock braking systems (ABS), the throttle logic from the Toyota case (Day 1).
- **Avionics** — flight control loops, the Traffic Collision Avoidance System (TCAS).
- **Hardware** — every synchronous digital circuit is an SRC on the clock edge.
- **Protocols** — TCP, mutual-exclusion, cache coherence.

The counter is a deliberately tiny instance of the same shape.

SMV: a textbook input language

```
MODULE main
  VAR
    mode : {off, on};      -- enumerated state
    press : boolean;       -- nondeterministic INPUT (no init/next)
    x     : 0..25;         -- bounded integer state

  DEFINE count_max := 10;  -- a named constant
```

- **VAR** declares state variables **and** inputs.
- A variable with **no** **init** / **next** clause is a free input — nuXmv lets the environment pick any value each round (this is our **press**).
- **0..25** makes **x** finite, so the state space is finite. The wide range is **headroom**: in the correct counter only **0-10** ever occur (12 states reachable from the initial state), and the spare range lets the off-by-one demo reach **x = 11** without overflowing the declared type.

SMV: the transition relation

```
ASSIGN
  init(mode) := off;
  init(x)     := 0;
  next(mode) := case
    mode = off & !press           : off;
    mode = off & press            : on;
    mode = on  & !press & x < count_max : on;
    mode = on  & (press | x ≥ count_max): off;
    TRUE                          : mode;  -- default
  esac;
  next(x) := case
    mode = on & !press & x < count_max : x + 1;
    mode = on & (press | x ≥ count_max) : 0;
    TRUE                               : x;
  esac;
```

In SMV, `&` = and, `|` = or, `!` = not; a `case` picks the **first** guard that matches (top to bottom), and `TRUE :` is the catch-all default.

This is the same counter, line for line — and the same four guards as the Z3 `step()` from Day 1.

Specifications live in the same file

```
INVARSPEC x ≤ count_max;           -- safety: always true
INVARSPEC (mode = off) → (x = 0);   -- conditional safety
CTLSPEC   AG AF (mode = off & x = 0); -- always eventually "home"
LTLSPEC   F (x = count_max);        -- eventually maximal
```

- $\rightarrow$ reads “implies”: $A \rightarrow B$ says whenever A holds, B must too.
- **INVARSPEC** p — p holds in every reachable state (pure safety).
- **CTLSPEC** / **LTLSPEC** — richer temporal properties (next block).
- nuXmv checks **every** spec in the file when you run it.

Four SMV modeling idioms

The whole language is small. Four idioms cover almost everything you'll write:

- **Free input** — a `VAR` with no `init` / `next`. The environment picks any value each round (`press`). This is how nondeterminism enters with *zero* extra syntax.
- **Explicit nondeterminism** — `next(s) := {idle, waiting};` means the next value is chosen from the set, nondeterministically. (Used in `mutex.smv`: `process1 = idle : {idle, waiting};`.)
- **DEFINE** — `DEFINE d := b | c;` is a *macro*: every use of `d` is textually replaced by `(b | c)`. No new state variable, no extra BDD variable — just a name for a sub-expression.
- **MODULE** — a reusable component, instantiated like an object: `proc1 : user(sem);`. Each instance gets its own copy of the local variables (`proc1.state`).

Synchronous vs. asynchronous composition

When you compose components, *whose clock ticks?*

- **Synchronous** — every component takes a step **together**, each round. One step of the whole = one step of *each* part. (Hardware on a shared clock; our `traffic_light.smv` updates all three lights every tick.)
- **Asynchronous** — one step of the whole = a step by **exactly one** component; the others' variables stay unchanged (an *interleaving*). (Threads, distributed processes.)

SMV models asynchrony *inside* the synchronous framework: add a scheduler that picks who moves (this is `peterson.smv`'s scheduler variable).

counter.smv has true *and* false specs on purpose

`counter.smv` includes specs that **hold** and specs that **fail**:

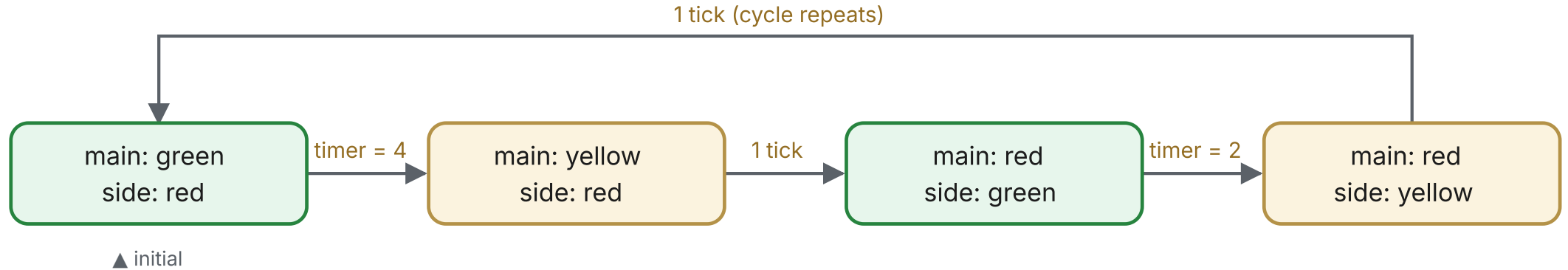
- ✓ `x ≤ count_max` — holds.
- ✓ `(mode = off) → (x = 0)` — holds.
- ✗ `x < count_max` — fails (x reaches exactly 10).
- ✗ `x ≤ count_max / 2` — fails (x reaches 10 > 5).

Don't assume every spec is meant to pass. Read each verdict against what you expect.

Other models you'll meet today

File	System	Idiom it shows
counter.smv	the running counter	nondeterministic input
traffic_light.smv	four-phase intersection	timed phases, mutual exclusion of greens
mutex.smv	two-process mutual exclusion	flags + turn (idle/waiting/critical)
peterson.smv	Peterson's algorithm	interleaving scheduler + fairness
elevator.smv	single-car elevator	request handling, safety
gcd_01.smv	Euclid's GCD	explicit program counter (a <i>program</i> as a TS)

traffic_light.smv as a state machine



The cycle the file walks: **main green** → **main yellow** → **all-red** → **side green** → **side yellow** → **all-red** → main green. The `timer` counts ticks within each phase; a `next_dir` ghost variable carries the handover direction through the all-red phase (the figure simplifies the two 1-tick clearance ticks for clarity). Safety invariants — never two greens, never two yellows, mutually exclusive — hold throughout; we confirm with nuXmv.

Actuated controller: `traffic_light_actuated.smv`

Real intersections aren't pre-timed: the side road only gets green **when a car arrives**. Model the arrival as a non-deterministic free input `side_request : boolean` (`IVAR`); the controller stays in main green until a request appears after a minimum hold.

The branching matters. The path “no request ever” stays in main_green forever; every path with a request reaches side_green. **EF** and **AF** now genuinely differ — the slide-worthy CTL/LTL distinction:

Spec	Verdict	Why
<code>EF (side_light = green)</code>	holds	some path takes a request and reaches side_green
<code>AF (side_light = green)</code>	fails	the no-request path is a counterexample — stays main_green forever
<code>AG AF (side_light = green)</code>	fails	same counterexample, from every reachable main-green state
<code>EG (main_light = green)</code>	holds	exactly the no-request path
<code>AG (side_request → EF (side_light = green))</code>	holds	requests are servable
<code>G F side_request → G F (side_light = green)</code> (LTL)	holds	no starvation when requests keep coming

Same SMV vocabulary — the difference is the `IVAR` and a minimum-hold guard.

A program as a transition system: gcd_01.smv

```
-- while (a ≠ b) { if (a > b) a = a-b; else b = b-a; }
next(pc) := case
    pc = l1 & a ≠ b : l2;    -- loop guard true
    pc = l1 & a = b : l5;    -- done
    pc = l2 & a > b : l3;    -- then-branch
    pc = l2 & a ≤ b : l4;    -- else-branch
    pc = l3 | pc = l4: l1;    -- back to loop head
    pc = l5          : l5;    -- terminal self-loop
esac;
```

Add a **program counter** ranging over line labels; loops become back-edges.

L1 recap

- Reactive systems run forever; correctness is about **behavior over time**.
- A **synchronous reactive component** = inputs + state + init + reaction.
- **SMV** writes that directly: **VAR** (state + free inputs), **init** / **next** (the relation), **INVARSPEC** / **CTLSPEC** / **LTLSPEC** (the properties).
- Any sequential **program** becomes a transition system via a program counter.



Break

We resume after the break with temporal logic — CTL and LTL.

L2 — Temporal logic: CTL and LTL

Why propositional logic isn't enough

Propositional logic talks about **one** state: " $x \leq 10$ here."

Reactive correctness needs claims about **executions**:

- " $x \leq 10$ in *every* reachable state" (always).
- "the system *eventually* returns to **off**" (eventually).
- "no green light *until* the other turns red" (until).

These are **temporal** modalities. Propositional logic can't say "always" or "eventually."

Two property shapes: safety and liveness

Almost every requirement is one of these (or a combination):

- **Safety** — *“something bad never happens.”* A counterexample is a **finite path** to the bad state. (e.g. x never exceeds 10; two greens never lit together.)
- **Liveness** — *“something good eventually happens.”* A counterexample is an **infinite run** (a lasso) where the good thing never occurs. (e.g. every waiting process eventually enters its critical section.)

LTL — linear temporal logic

LTL views the future as a **single path** (implicitly, all paths). Operators on a path:

- $X p$ — **neXt**: p in the next state.
- $F p$ — **Finally** (eventually): p at some future state.
- $G p$ — **Globally** (always): p at every future state.
- $p U q$ — **Until**: p holds until q becomes true (and q does eventually become true).
- $p R q$ — **Release** (the dual of *until*): q must stay true up to and including the moment p first becomes true — and forever if p never does.

Reading LTL on a trace

A trace is a sequence of states $s_0, s_1, s_2, \dots$

$G p$: p true at $s_0, s_1, s_2, \dots$ (every position)

$F p$: p true at some s_i

$X p$: p true at s_1

$p U q$: q true at some s_i , and p true at $s_0 \dots s_{i-1}$

A finite system has infinite traces that end in a **lasso** (a cycle): a stem then a repeating loop.

One trace, marked up

Fix one counter run and mark where `mode = on` holds (●) vs not (○):

```
position:  s0   s1   s2   s3   s4   s5   s6   ...
mode=on:   ○   ●   ●   ●   ●   ○   ●   ...
           (off) (on) (on) (on) (on) (off) (on)
```

We'll read each LTL operator against **this one trace**, from position `s0`. The valuation function gives each proposition a 0/1 at each position; an operator is a claim about the rest of the trace from where you stand.

Notation on the next slides: $(\rho, n) \models \psi$ reads "trace ρ at position n satisfies ψ "; $\exists$ = "there exists", $\forall$ = "for all".



Caption: the same run, drawn as a timeline — every LTL operator below is read off this picture.

$X\ p$ — neXt, on the trace

$X\ p$ ("in the next state, p ") looks **exactly one step** forward.

- At s_0 : $X\ (\text{mode} = \text{on})$ asks "is $\text{mode} = \text{on}$ at s_1 ?" — s_1 is $\bullet \rightarrow \text{true}$.
- At s_4 : $X\ (\text{mode} = \text{on})$ asks about s_5 — s_5 is $\circ \rightarrow \text{false}$.

$$(\rho, n) \models X\ p \quad \text{iff} \quad (\rho, n+1) \models p$$

X is the only operator that pins down a single, exact position. Everything else quantifies over a *range* of future positions.

F p — Finally / eventually, on the trace

F p ("at some position now or later, **p**") is an **existential** over future positions.

- **F (x = count_max)** on the counter: is there *any* position where **x = 10**? On a counting run, yes → **true**.
- **F (mode = on)** at **s0**: **s1** is already ● → **true** (the witness can be the current state or any later one).

$$(\rho, n) \models F p \quad \text{iff} \quad \exists m \geq n. (\rho, m) \models p$$

One witnessing position is enough. **F** is how you say "reachable along this run."

$G\ p$ — Globally / always, on the trace

$G\ p$ ("at every position from here on, p ") is the **universal** dual of F .

- $G\ (x \leq \text{count_max})$ on the counter: every reachable state has $x \leq 10 \rightarrow \text{true}$ (this is the safety invariant).
- $G\ (\text{mode} = \text{on})$ at s_0 : s_0 is $\bigcirc \rightarrow \text{false}$ (one bad position kills it).

$$(\rho, n) \models G\ p \quad \text{iff} \quad \forall m \geq n. (\rho, m) \models p \quad G\ p \equiv \neg F \neg p$$

A single position where p fails refutes $G\ p$. G / F are De Morgan duals: "always" = "never not."

$p \text{ U } q$ — Until, on the trace

$p \text{ U } q$ (" p holds until q does, and q *does* eventually") bundles a deadline with an obligation.

- $(\text{mode} = \text{on}) \text{ U } (\text{mode} = \text{off})$ at $s1$: $\text{mode} = \text{on}$ at $s1..s4$, then $\text{mode} = \text{off}$ at $s5 \rightarrow \text{true}$.
- It would be **false** if mode never returned to off (the obligation " q eventually" is unmet) — even though p held the whole time.

$$(\rho, n) \models p \text{ U } q \text{ iff } \exists m \geq n. (\rho, m) \models q \wedge \forall i, n \leq i < m. (\rho, i) \models p$$

F and G are special cases: $F q \equiv \text{true} \text{ U } q$, and U is the most expressive basic operator.

$p \ R \ q$ — Release (the dual of Until)

$p \ R \ q$ (" q must hold up to **and including** the step where p first becomes true — and forever if p never does"):

$$p \ R \ q \equiv \neg(\neg p \ U \neg q)$$

- q is the **guarantee**; p is the event that **releases** it.
- If p never happens, q must hold forever — so $G \ q \equiv \text{false} \ R \ q$.

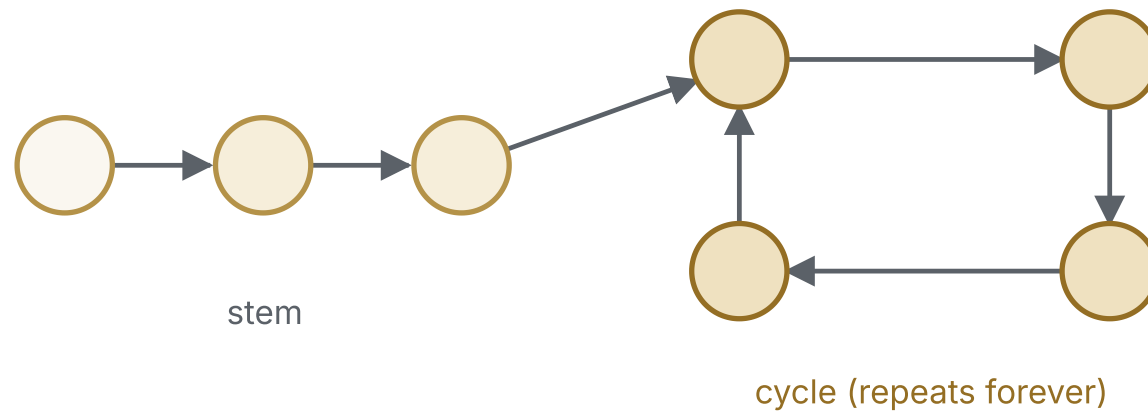
Example: $(\text{mode} = \text{off}) \ R \ (x \leq \text{count_max})$ — " $x \leq 10$ stays true at least until the counter is off." Since $x \leq 10$ always holds, this is **true** whether or not mode ever turns off.

Why finite-state $\Rightarrow$ decidable: the lasso

A trace is infinite, but a finite system has only finitely many states — so any infinite run must **revisit** a state, then it can repeat that loop forever.

- Every infinite behavior = a **stem** (finite prefix) + a **cycle** (repeated forever) = a **lasso**.
- There are only finitely many distinct stems and cycles.
- So "does *some* infinite run violate the property?" becomes a **finite** search over lassos.

That is why model checking decides $G p$, $F G p$, "infinitely often," etc., even though they quantify over infinite time.



Caption: every infinite run through finitely many states is a lasso — a stem into a repeating cycle — which is what makes infinite-time properties decidable.

How the check works: LTL $\rightarrow$ Büchi $\rightarrow$ emptiness

The lasso is the *what*; here is the *how* — the classic algorithm (SPIN's **explicit-state** engine; nuXmv reaches the same verdict *symbolically*, next block):

1. **Negate & translate.** Build a **Büchi automaton** for $\neg\varphi$ — it accepts exactly the *bad* runs (those violating φ).
2. **System as automaton.** View the model K as an automaton whose language is all its executions.
3. **Product.** Form $K \times B(\neg\varphi)$; its language is the runs of K that are *also* bad = $L(K) \cap L(\neg\varphi)$.
4. **Emptiness.** Is that language empty? **Empty** $\Rightarrow$ **no bad run** $\Rightarrow K \models \varphi$. Non-empty $\Rightarrow$ an accepting **lasso** — your counterexample.

It is valid because $L(K) \subseteq L(\varphi)$ **iff** $L(K) \cap L(\neg\varphi) = \emptyset$ — the same “assert the negation, hunt for a witness” move as SAT/SMT. Emptiness = a linear-time search for a reachable accepting cycle (nested depth-first search).

LTL patterns you'll actually write

Formula	English
$G\ p$	invariant: always p
$F\ p$	reachability: eventually p
$G\ F\ p$	infinitely often p
$F\ G\ p$	eventually always p (stabilization)
$G\ (p \rightarrow F\ q)$	response : every p is eventually followed by q

G F p — infinitely often, worked

G F p = "at **every** position, **p** happens **again** later" = **p** recurs forever (the **recurrence** pattern).

- Counter: **G F (mode = off & x = 0)** is **true** — every fair run keeps returning home. (This is a *shipped* **counter.smv** spec that passes.)
- Traffic light: **G F (main_light = green)** is **true** — the cycle relights main green forever.

Violating trace looks like: **p** happens a *last* time and then never again — e.g. a run that gets stuck **mode = on** would falsify **G F (mode = off)**.

F G p — stabilization, worked

$F G p$ = "from **some** position on, p holds **forever** after" = the system settles (the **persistence** pattern).

- Counter: $F G (\text{mode} = \text{on})$ is **false** — reaching $x = 10$ always forces **off**, so **on** can never become permanent. (Shipped failing spec.)
- A thermostat reaching and holding setpoint: $F G (|\text{temp} - \text{target}| < 1)$ — **true** if it eventually stops oscillating.

$F G p$ and $G F p$ are the two most-confused combinations: $G F$ = "comes back forever," $F G$ = "stays put eventually." $F G p$ is **stronger** ($F G p \Rightarrow G F p$).

$G (p \rightarrow F q)$ — response, worked

$G (p \rightarrow F q)$ = “**every** time p happens, q happens **at or after** it” — the single most common real requirement.

- Mutex: $G (\text{process1} = \text{waiting} \rightarrow F (\text{process1} = \text{critical}))$ — “every request is eventually served” (no starvation). Shipped `mutex.smv` spec.
- $\text{request} \rightarrow F \text{ grant}$, $\text{press} \rightarrow F \text{ door_opens}$, $\text{alarm} \rightarrow F \text{ shutdown}$.

Violating trace: a p with no later q — e.g. `process1` waits, but a cycle keeps letting `process2` in forever while `process1` never enters. That stuck cycle is the counterexample.

Quick warning: $F p \ \& \ F q \neq F (p \ \& \ q)$

Splitting a temporal operator across a connective changes the meaning.

- $F p \ \& \ F q$ — p happens *sometime*, q happens *sometime* — possibly at **different** positions.
- $F (p \ \& \ q)$ — p and q true at the **same** position.

On the trace $x = 0, 1, 0, 1, 0, 1, \dots$: $F(x=0) \ \& \ F(x=1)$ holds, but $F(x=0 \ \& \ x=1)$ is impossible. The first is **strictly weaker**.

CTL — computation tree logic

CTL views the future as a **branching tree** of all possible paths. Each temporal operator is paired with a **path quantifier**:

- **A** — for **all** paths from this state.
- **E** — there **exists** a path from this state.

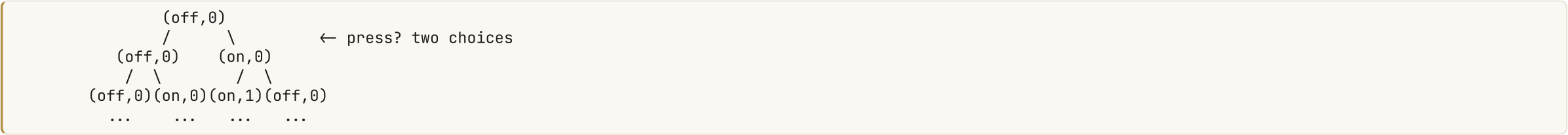
Then **AX, EX, AF, EF, AG, EG, A[p U q], E[p U q]**.

CTL operators that matter

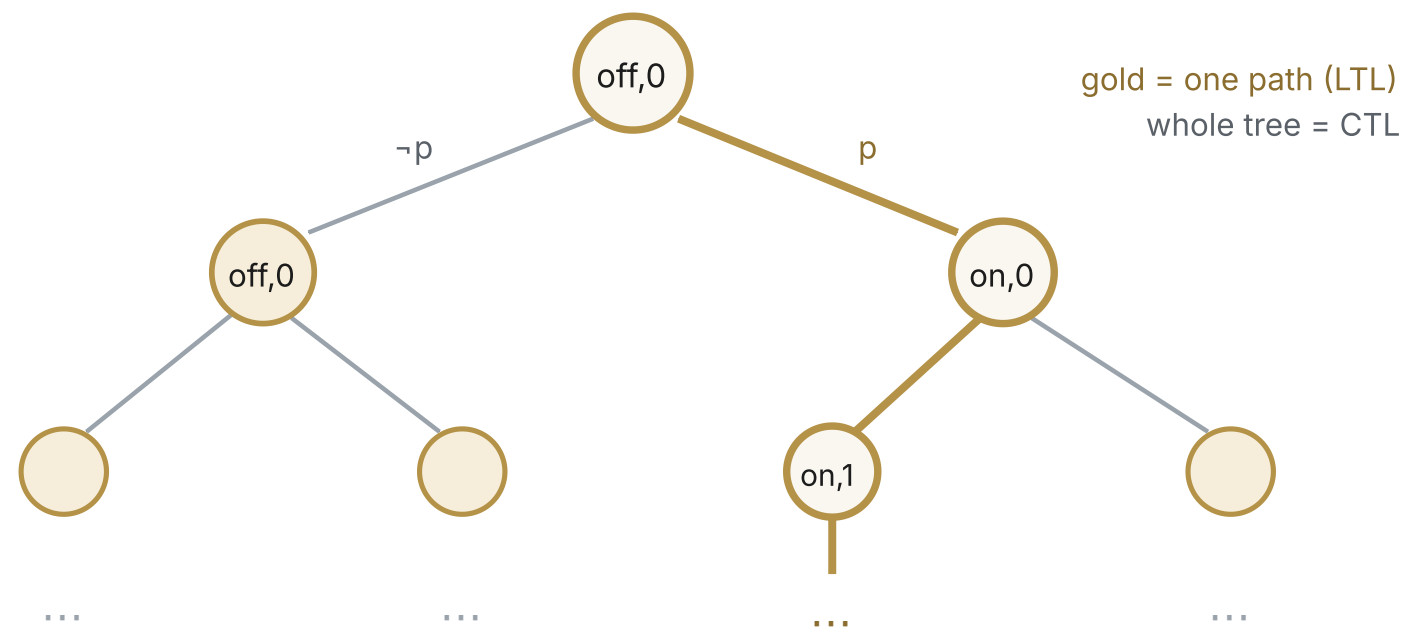
Formula	English
$AG\ p$	invariant: p on all reachable states
$EF\ p$	reachability: some path reaches p
$AF\ p$	inevitability: every path eventually hits p
$EG\ p$	some path keeps p forever
$AG\ EF\ p$	from everywhere, p is still reachable ("recoverable")

The computation tree

Unfold the transition system from the initial state: each branch is a nondeterministic choice. The result is the **computation tree** — *all* possible futures at once.



- An **LTL** formula is read along **one path** (one root-to-leaf branch).
- A **CTL** formula is read at **nodes**, with **A** (all branches from here) / **E** (some branch from here).



CTL operators, one at a time

Each pairs a path quantifier (**A** all / **E** some) with a temporal operator, read at the **current node**:

	X (next)	F (eventually)	G (always)
A (all paths)	AX p : p at <i>every</i> next state	AF p : <i>every</i> path hits p	AG p : p on <i>every</i> reachable state
E (some path)	EX p : p at <i>some</i> next state	EF p : <i>some</i> path reaches p	EG p : <i>some</i> path keeps p forever

Plus **A**[p U q] / **E**[p U q]. Counter readings:

- **EF** ($x = 10$) — **true**: some path counts up to 10.
- **AF** ($x = 10$) — **false**: the path that keeps pressing never lets x climb.
- **EG** ($\text{mode} = \text{on}$) — **false**: every on-run is forced off at $x = 10$.

AG EF p — recoverability

AG EF p nests two quantifiers: “**on every** reachable state (AG), **some** path gets back to p (EF).”

- “No matter where the system wanders, it can **always still return** to p.”
- Counter: AG EF (mode = off) — **true**: from any reachable state, some run returns home. (The **reset** property.)
- Mutex: AG EF (process1 = critical) — “process 1 can always eventually get in (on *some* schedule).” Shipped, passes.

This is the canonical property **CTL can express and LTL cannot** — the next slide says why.

LTL vs CTL — incomparable

Neither subsumes the other:

- **CTL-only:** $AG\ EF\ p$ — “from every reachable state, p is still reachable.” LTL cannot say this (no existential path quantifier).
- **LTL-only:** $F\ G\ p$ — “eventually p holds forever.” CTL’s $AF\ AG\ p$ is *not* equivalent.

Most tools (NuSMV/nuXmv included) support both. **CTL*** is the larger logic that freely mixes path quantifiers (A / E) with temporal operators — it contains both LTL and CTL.

Why the two are incomparable

The argument is about **paths vs. nodes**, not difficulty:

- **AG EF p** is **CTL-only**. **EF** says “some future path reaches **p**.” LTL only ever quantifies over *all* paths of the system — it has **no existential path quantifier**, so it cannot say “from here, a path back to **p** exists.” Branching is essential.
- **F G p** is **LTL-only**. It means “on this path, **p** eventually stays on.” The natural CTL translation **AF AG p** is **strictly stronger**: **AF AG p** demands a *single* moment after which *all* branches keep **p**, but **F G p** only constrains each path on its own. Two trees can agree on every path’s **F G p** yet differ on **AF AG p**.

CTL* drops the “one quantifier per operator” rule (e.g. **AF G p**) and contains both.

Properties of the counter

Property	Formula	Holds?
x never exceeds 10	$AG (x \leq 10)$	✓
can always get home	$AG EF (mode = off)$	✓
eventually stuck on	$F G (mode = on)$	✗
$x = 10$ reachable from anywhere	$AG EF (x = 10)$	✓

Quick check: match the English to the formula

1. Every reachable state has $x \leq 10$.
2. Every reachable state can return to $\text{mode} = \text{off}$.
3. The system eventually settles in $\text{mode} = \text{on}$ forever.
4. From every state, some path reaches $x = 10$.

Choices (not in order): A. $F\ G\ (\text{mode} = \text{on})$ · B. $AG\ EF\ (x = 10)$ · C. $AG\ (x \leq 10)$
· D. $AG\ EF\ (\text{mode} = \text{off})$

Answers: $1 \rightarrow C$, $2 \rightarrow D$, $3 \rightarrow A$, $4 \rightarrow B$.

A note on fairness

`counter.smv` has `FAIRNESS mode  $\neq$  off;` — restrict attention to executions where `mode  $\neq$  off` holds **infinitely often**.

- Fairness rules out “unrealistic” runs (e.g. the environment never pressing).
- It changes which liveness properties hold.
- Use it carefully: it is an assumption about the environment, not the system.

Worked safety counterexample (finite path)

Suppose we *broke* mutex so both processes could enter together and checked `INVARSPEC !(process1 = critical & process2 = critical)`. A safety violation is a **finite path** to the bad state:

```
-- specification is false; as demonstrated by:  
→ State 1.1 ← process1=idle,    process2=idle,    flag1=F, flag2=F  
→ State 1.2 ← process1=waiting, process2=waiting, flag1=T, flag2=T  
→ State 1.3 ← process1=critical, process2=critical    ← BOTH in critical
```

You can **point at** the bad state and replay the exact 3-step sequence that reached it. That is the whole counterexample — finite, concrete, replayable.

Worked liveness counterexample (a lasso)

Real `counter.smv` spec `G F (mode = on & x = count_max)` ("infinitely often we're on with x=10") is **false**. A liveness violation is a **lasso** — a stem into a repeating cycle where the good event never recurs:

```
-- specification G F (mode = on & x = count_max) is false
→ State 1.1 ← mode=off, press=FALSE, x=0      ← stem
-- Loop starts here
→ State 1.2 ← mode=off, press=FALSE, x=0
→ State 1.2 ← mode=off, press=FALSE, x=0      ← cycle: repeat forever
```

The cycle stays `off` forever (the environment simply never presses), so `(mode = on & x = 10)` *never* happens — the promised event is absent on an infinite run.

L2 recap

- **Temporal logic** expresses claims about executions, not single states.
- **LTL** (linear, one path) — X , F , G , U , R ; patterns $G F$ (∞ -often), $F G$ (stabilize), $G(p \rightarrow F q)$ (response).
- **CTL** (branching, A / E on the computation tree) — AG , EF , AF , EG , $AG EF$ (recoverable).
- They are **incomparable** *because* LTL reads paths and CTL reads tree nodes: AG $EF p$ needs E (CTL-only); $F G p \neq AF AG p$ (LTL-only).
- A **safety** violation is a finite path; a **liveness** violation is a **lasso** (stem + cycle).
- **Fairness** restricts to realistic executions for liveness.



Break

We resume after the break with model-checking algorithms and live nuXmv.

L3 — Algorithms and live nuXmv

Three model-checking algorithms

Algorithm	Stores	Strength	Limitation
Explicit-state	visited-state set (BFS/DFS)	simple, rich properties	state explosion
Symbolic (BDD)	boolean encoding of state <i>sets</i>	>100 state bits	variable-order sensitive; weak on arithmetic
Bounded (SAT/SMT)	length- N unrolling	no state space at all	refutes only; needs completeness arg

NuSMV/nuXmv run the **symbolic** and **bounded** engines; explicit-state enumeration is the SPIN family (a different tool).

How hard is model checking? (1) the infinite-state wall

Ask: “is `p` an invariant of transition system `T`?” The answer depends entirely on whether `T` is finite-state.

- **Infinite state (e.g. unbounded `int` variables): UNDECIDABLE.** With mathematical integers, `T` can encode an arbitrary program / Turing machine — so invariant verification is as hard as the halting problem. The standard proof reduces from **Minsky two-counter machines**.
- Intuition: there is **no a-priori bound** on the reachable states, so “examine them all” never terminates.

This is why our SMV models bound every variable (`x : 0..25`) — and why **Day 3 (Lean)** needs *induction*, not enumeration, for unbounded systems.

How hard is model checking? (2) finite-state = PSPACE

Make every variable finite and the problem becomes **decidable** — but not cheap:

- **k** boolean variables $\Rightarrow$ up to 2^k states. A verifier *can* search them all, so it terminates.
- Invariant / reachability checking for finite-state systems is **PSPACE-complete**. Since $\text{PSPACE} \supseteq \text{NP}$, it is at least as hard as NP-complete SAT; whether that containment is *strict* ($\text{PSPACE} \neq \text{NP}$) is a famous open question.
- The practical face of that exponent is the **state-explosion problem**: state count blows up with variables, components, and interleavings.

Property logic	Model-checking complexity
CTL	$O(S \cdot \varphi)$ — linear in states $\times$ formula
LTL	$O(S \cdot 2^{ \varphi })$ — linear in states, exponential in formula size

(As a decision problem, LTL model checking is PSPACE-complete in the formula; the bound shown is for the standard Büchi-automaton construction.)

Symbolic model checking: sets as formulas

Represent a **set of states** by a boolean formula true exactly on that set.

- Reachable set $R_0 = \text{init}$ (the initial states).
- $R_{k+1} = R_k \cup \text{Image}(R_k)$ — add the one-step successors ($\text{Image}(R)$ = all states reachable in one step from R ; $\cup$ = set union).
- Stop when $R_{k+1} = R_k$ (a **fixpoint** — nothing new appears): that's all reachable states.
- Safety check: is $R_\infty \cap \text{Bad} = \emptyset$? ($\cap$ = states in both sets; $\emptyset$ = empty set — so no bad state is reachable.)

The whole computation is boolean-formula manipulation — no state is ever enumerated individually.

The symbolic transition relation

A model checker never lists states — it computes with **formulas over the variables**.

- **Initial states** $\rightarrow$ a formula φ_I . For the counter: $\varphi_I \equiv (\text{mode} = \text{off}) \wedge (x = 0)$.
- **Transitions** $\rightarrow$ a formula φ_T over the variables **and their primed (next-state) copies** (x' = value of x *after* the step).

The counter's increment edge is one disjunct of φ_T :

$$(\text{mode} = \text{on}) \wedge \neg p \wedge (x < 10) \wedge (\text{mode}' = \text{on}) \wedge (x' = x + 1)$$

If T has k variables, a *set of states* is a formula over k variables and the *transition relation* is a formula over $2k$ variables. Variables a step doesn't touch need an explicit $y' = y$ (a **frame condition**).

Computing successors: the image $\text{Post}(A)$

Given a region A (a set of states), its **image** is the set of one-step successors:

$$\text{Post}(A) = \{ t \mid \exists s \in A, (s, t) \in \varphi_T \}$$

Three mechanical steps — the heart of symbolic search:

1. **Conjoin:** $A(s) \wedge \varphi_T(s, s')$ — all transitions starting in A .
2. **Project:** $\exists s. (A \wedge \varphi_T)$ — quantify away the *current*-state variables, keeping only reachable next-states s' .
3. **Rename** $s' \rightarrow s$ — so the result is again a region over the ordinary variables.

$$\text{Post}(A) = \text{Rename}(\exists s. (A \wedge \varphi_T), s' \rightarrow s)$$

Worked image computation (with Z3)

One real variable; transition $\varphi_T : x' = 2x + 1$; region $A : 0 \leq x \leq 10$. Compute $\text{Post}(A)$:

1. **Conjoin:** $(0 \leq x \leq 10) \wedge (x' = 2x + 1)$
2. **Project out x :** $\exists x. (0 \leq x \leq 10) \wedge (x' = 2x + 1)$ — simplifies to $1 \leq x' \leq 21$
3. **Rename $x' \rightarrow x$:** $1 \leq x \leq 21$ — exactly the image of $[0, 10]$ under $x \mapsto 2x + 1$.
✓

```
(declare-fun xP () Real)
(assert (exists ((x Real)) (and (<= 0 x) (<= x 10) (= xP (+ (* 2 x) 1)))))
(apply qe)           ; quantifier elimination → (and (>= xP 1) (<= xP 21))
```

Projection is existential quantifier elimination

"Project out x " = " $\exists x$ " = drop x but keep what it still forces on the rest.

- **Booleans** (BDD-friendly): $\exists x. \varphi = \varphi[x \mapsto 0] \vee \varphi[x \mapsto 1]$ — substitute both values, OR the results.
- **Linear arithmetic**: Fourier–Motzkin elimination, or Z3's **qe** (as above).

A symbolic model checker needs just **six region operations**:

Op	Set meaning	On formulas
Conj (A,B)	$A \cap B$	$A \wedge B$
Disj (A,B)	$A \cup B$	$A \vee B$
Diff (A,B)	$A \setminus B$	$A \wedge \neg B$
IsEmpty (A)	$A = \emptyset?$	a SAT test (unsat = empty)
Exists (A,X)	project out X	quantifier elimination
Rename (A,X,Y)	rename vars	substitution

The fixpoint, on the counter

Run the reachable-set iteration by hand — it lands on the **same 12 states** as Day 1's BFS:

$$R_0 = \{(\text{off}, 0)\}$$

$$R_1 = R_0 \cup \{(\text{on}, 0)\}, \quad R_2 = R_1 \cup \{(\text{on}, 1)\}, \quad \dots$$

$$R_{11} = R_{10} \cup \{(\text{on}, 10)\}, \quad R_{12} = R_{11} \text{ (fixpoint)}$$

Now $R_{12} \cap \{x > 10\} = \emptyset$ — the safety invariant holds, and **no individual state was ever enumerated**: each R_k is one boolean formula (one BDD).

The frontier: symbolic BFS

Don't re-expand the whole set each step — push only the **frontier** (the newly-found states):

```
Reach := Init;   New := Init           # New = the frontier
while New ≠ ∅:
  if New ∩ Bad ≠ ∅: return REACHABLE    # property violated → counterexample
  New := Post(New) \ Reach              # successors not seen before
  Reach := Reach ∪ New
return UNREACHABLE                     # fixpoint: nothing new ⇒ invariant holds
```



Each ring is one BFS layer `reachi`; the dashed gold band is the frontier `New` that `Post` pushes outward. It stops after the reachable graph's **diameter** (for the counter, 11 steps — then `New = ∅`).

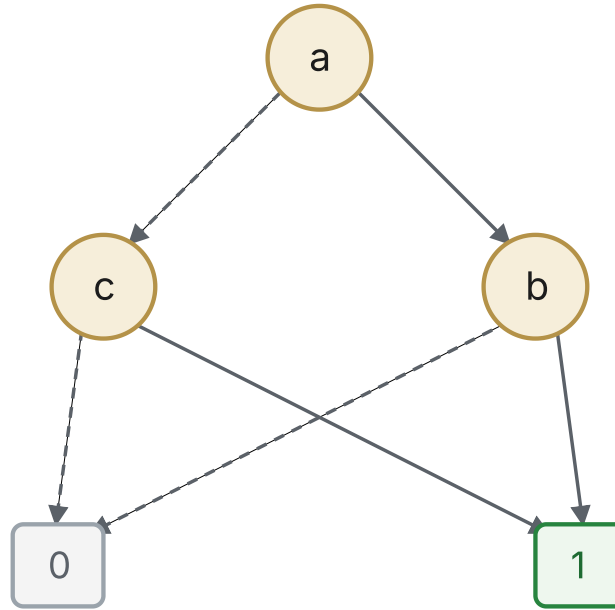
BDDs — binary decision diagrams

A **BDD** is a canonical, compressed decision tree for a boolean function.

- **Shannon expansion:** $f = (x \wedge f|_{x=1}) \vee (\neg x \wedge f|_{x=0})$ — split on one variable ($\wedge$ and, $\vee$ or, $\neg$ not; $f|_{x=1}$ is f with x fixed true).
- Apply recursively over a **fixed variable order** → a **DAG** (directed acyclic graph — a decision tree with identical subgraphs shared).
- Reduce: merge identical subgraphs, drop redundant nodes.
- Result is **canonical**: two functions are equal iff their BDDs are identical.

A BDD, drawn

The function $f = \text{ite}(a, b, c) = (a \wedge b) \vee (\neg a \wedge c)$ — “if **a** then **b** else **c**” — as a BDD:

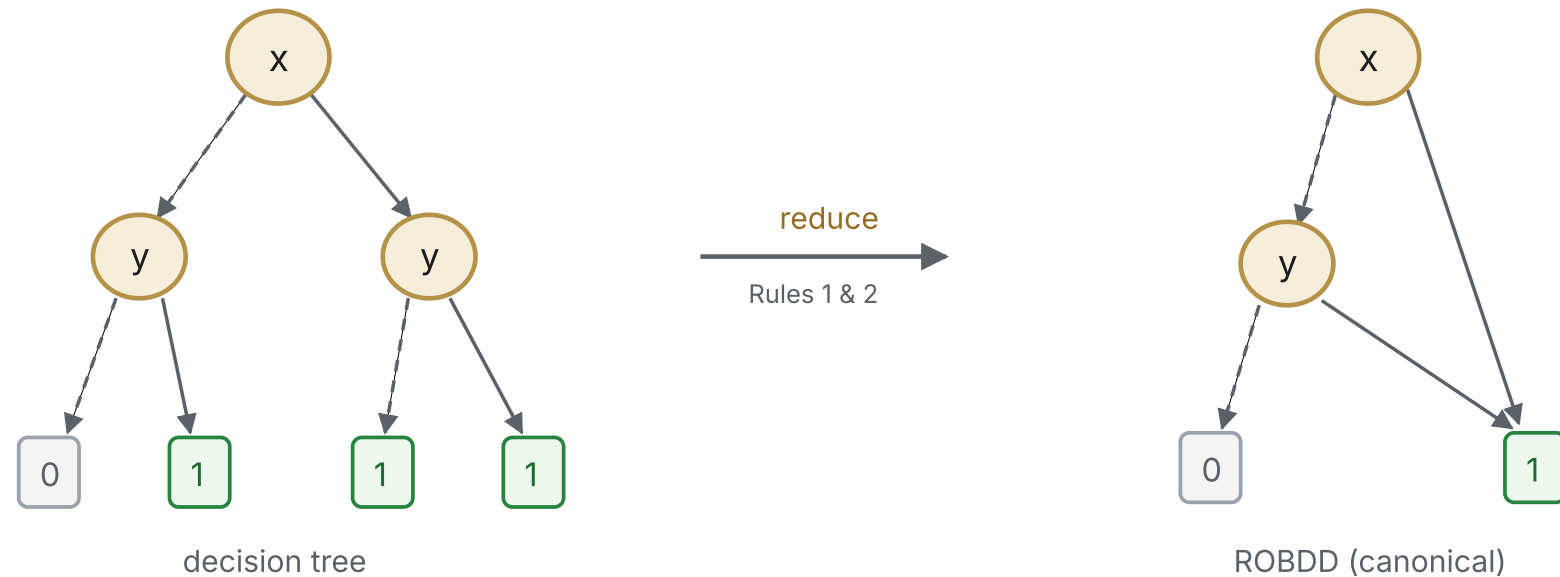


Solid edge = that variable is **1**; dashed = **0**. Follow your path of choices down to a terminal box (the output). The single shared **0** and **1** are what make it a **DAG**, not a tree — and why function equality becomes pointer-equality of BDDs.

Build and reduce BDDs yourself in the browser: bit.ly/fmaiv_smvis.

Building and reducing a BDD

Start from the Shannon-expansion **decision tree** for $f = x \vee y$, then apply two rules until none apply:



- **Rule 1 — merge** isomorphic subgraphs (identical nodes/leaves become one): the three **1**-leaves collapse to one.
- **Rule 2 — eliminate** any node whose two children are identical (a redundant test): the **y** on the **x = 1** branch disappears — **x = 1** already forces **1**.

The result is **canonical** for a fixed order, so equality is pointer-equality and **IsEmpty** / validity are just “is it the **0** / **1** terminal?” Build them yourself: bit.ly/fmaiv_smvis.

BDDs: variable ordering is everything

The same function can be **tiny or exponential** depending on variable order.

- Good order → linear-size BDD.
- Bad order → exponential blow-up.
- Finding the optimal order is itself **NP-complete** (Bollig & Wegener, 1996); tools use heuristics + dynamic reordering.

Arithmetic (multipliers) has **no** good order — BDDs scale poorly on it. That's where SAT/SMT (Day 1, Day 4) wins.

Bounded model checking, revisited

Day 1's idea, now in context:

- Unroll the transition relation **k** times into one big SAT/SMT formula.
- Ask: is there a length-**k** path to a bad state?
- **SAT** → bug found (a real counterexample). **UNSAT** → no bug *of length* $\leq k$.

To make BMC **complete**, add *k-induction* or compute a *completeness threshold* (the state graph's diameter) — both unpacked in the next two slides.

Inductive invariants: a safety proof in two checks

To prove a safety property P ("nothing bad ever happens") you rarely compute the exact reachable set. Instead find an **inductive invariant** J :

- **Initiation:** every initial state satisfies it — $\text{Init} \Rightarrow J$.
- **Consecution:** every step from a J -state stays in J — $J \wedge R \Rightarrow J'$.
- **Strength:** it implies what you wanted — $J \Rightarrow P$.

All three $\Rightarrow P$ holds for **every** reachable state — no unrolling, no bound.

- **Checking** a candidate is two SMT queries: consecution fails *iff* $J \wedge R \wedge \neg J'$ is **SAT** — the solver hands back the offending step.
- **The catch:** P itself is usually *not* inductive (e.g. $y \geq 1$ under $y' = y + x$ needs the helper $x \geq 1$). Model checking's job is to **find** an inductive J ; on **Day 3** you'll **supply** one by hand and prove it in Lean.

From bounded to complete: k-induction and IC3

Two ways to turn BMC's *bug-finding* into an unbounded *proof*:

- **k-induction = BMC made complete.** *Base*: no counterexample in the first **k** steps (a BMC query). *Step*: whenever **P** holds along **k** consecutive states, it holds at the next — with a “no repeated state” (simple-path) constraint so it terminates. Pass both $\Rightarrow$ **P** holds forever. Looking back **k** steps succeeds where 1-step induction fails.
- **IC3 / PDR** (Property-Directed Reachability). Builds an inductive invariant *incrementally* as a chain of **frames** (over-approximations of “reachable in $\leq i$ steps”), learning a small clause from each *counterexample-to-induction* — **without ever unrolling** the transition relation. Often the fastest engine.

nuXmv runs both (its **check_invar_ic3** uses IC3) — so when nuXmv proves your **G**-property, this is what is happening internally.

Running nuXmv

```
nuXmv counter.smv      # batch: check every spec in the file
```

Interactive session:

```
nuXmv -int counter.smv
> go                # build the model
> check_invar       # all INVARSPECs
> check_ctlspec     # all CTLSPECs
> check_ltlspec     # all LTLSPeCs
> quit
```

No install? Run it in your browser: bit.ly/fmaiv_smvis — the **smvis** tool runs NuSMV and visualizes state graphs and BDDs interactively.

What a passing run looks like

```
-- specification  $x \leq \text{count\_max}$  is true  
-- specification  $(\text{mode} = \text{off}) \rightarrow (x = 0)$  is true  
-- specification  $\text{AG} (\text{AF} (\text{mode} = \text{off} \ \& \ x = 0))$  is true
```

Each **INVARSPEC** / **CTLSPEC** / **LTLSPEC** prints **is true** or **is false**.

Reading a counterexample

Weaken `next(x)` (off-by-one: allow `x < count_max + 1`) and re-run:

```
-- specification x ≤ count_max is false
-- as demonstrated by the following execution sequence
→ State: 1.1  ← mode = off, x = 0,  press = TRUE    (off → on)
→ State: 1.2  ← mode = on,  x = 0,  press = FALSE
→ State: 1.3  ← mode = on,  x = 1,  press = FALSE
...
→ State: 1.12 ← mode = on,  x = 10, press = FALSE
→ State: 1.13 ← mode = on,  x = 11, press = FALSE    ← x > count_max
```

The trace is a **witness** — the exact input sequence that breaks the property. (13 states: one press to flip on, then 11 increments; the violation shows at `1.13`.)

Counterexamples for different property types

- **Invariant / AG false** → a finite path to a bad state.
- **EF p true** → a finite witness path that reaches p .
- **F p false / liveness** → a **lasso**: a stem plus a cycle where p never happens.

Same algorithms, industrial scale

The exact methods on these slides run on the largest chips and systems:

- **Cadence JasperGold, Synopsys VC Formal** — symbolic + BMC on every modern SoC.
- **SymbiYosys** (open-source) — SAT-based on Verilog.
- **Intel** post-FDIV — formal datapath verification (via Symbolic Trajectory Evaluation, a cousin of these methods) as a release gate.

The counter is a toy; the engine is production-grade.

L3 recap

- **Complexity**: invariant checking is **undecidable** for infinite-state (int vars $\Rightarrow$ Turing-complete), **PSPACE** for finite-state — i.e. state explosion. CTL MC is linear; LTL is exponential in formula size.
- **Symbolic / bounded** — the two engines NuSMV/nuXmv run (explicit-state is SPIN's domain).
- **Symbolic MC** computes the reachable set as a fixpoint over **BDDs**; ordering is everything.
- **BMC** refutes cheaply; k-induction makes it complete.
- A **counterexample** is a concrete, replayable witness — finite path or lasso.

Wrap + work

Day 2 in one slide

- A reactive system is a transition system; **SMV** writes it directly.
- **Temporal logic** (CTL/LTL) expresses correctness over executions.
- A **model checker** decides those properties exhaustively for finite state.
- Output is a **true/false verdict** plus, on failure, a **counterexample**.
- Same counter as Day 1 — but now verified for **all** behaviors, not just length $\leq N$.

Day 3 preview

- Same counter. New tool: **Lean 4**.
- When finite state isn't enough — parametric **n**, unbounded integers, real mathematics.
- Prove the invariant **by induction on the transition relation**, not by enumeration.
- AI (Claude Code) as a proof-drafting partner.

In-session exercise

With `nuXmv` (or **smvis** in your browser — bit.ly/fmaiv_smvis) on `counter.smv`:

1. Confirm the passing `INVARSPEC` s are `true` and the deliberately-false ones are `false`.
2. Introduce the off-by-one bug in `next(x)`; read the counterexample; identify the exact press sequence.
3. Revert. Write **one new** CTL or LTL property of your own and check it.

Homework (ungraded, for depth)

Pick **one** of `traffic_light.smv`, `mutex.smv`, `gcd_01.smv`:

- Verify the two given properties; explain each verdict.
- Add and verify **one safety** and **one liveness** property of your own.
- For `mutex`: state mutual exclusion (`AG !(process1 = critical & process2 = critical)`) and no-starvation (`AG(process1 = waiting → AF process1 = critical)`).

See `assignments/day02.md`.

References for Day 2

- **Clarke, Grumberg, Kroening, Peled, Veith.** *Model Checking* (2nd ed.), MIT Press 2018 — the canonical reference.
- **Baier, Katoen.** *Principles of Model Checking*, MIT Press 2008 — CTL/LTL chapters.
- **Cavada et al.** *The nuXmv Symbolic Model Checker*, CAV 2014.
- **Bryant.** *Graph-Based Algorithms for Boolean Function Manipulation*, IEEE TC 1986 — BDDs.
- **nuXmv user manual** — <https://nuxmv.fbk.eu/>
- **smvis** (browser NuSMV + BDD/state-graph visualizer) — **run it:** https://bit.ly/fmaiv_smvis · source: <https://github.com/verivital/smvis>

Full list: repo [README.md](#).

Next session: Day 3

Repo: github.com/ttj/fmaiv — slides, the four `.smv` examples, and the assignment.