



VANDERBILT UNIVERSITY
College of Connected Computing

Day 1 — Foundations: Logic, Transition Systems, SAT, SMT

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 1 of 4

Day 1 — Foundations

Logic, transition systems, SAT, SMT

Run it live (no install): Codespaces (full toolset) or Colab — today's notebook
`01_day1_logic_sat_smt`. All materials: <https://github.com/ttj/fmaiv>

Your day at a glance

Block	Length	Topic
Opening	~10 min	Recap, roadmap, learning objectives
L1	~50 min	Why now: the AI × FM asymmetry; bug stories; industrial deployment
Break	~10 min	
L2	~50 min	Propositional + first-order logic; transition systems
Break	~5 min	
L3	~45 min	SAT, SMT, Z3 — live coding
Wrap + exercise	~10 min	One-line recaps; homework pointer

The four-day arc

Day	Theme	Tools	Question answered
1	Logic + SAT + SMT	Z3	"Is there a counterexample of length $\leq N$?"
2	Model checking	nuXmv	"Is there a counterexample, ever?"
3	Theorem proving	Lean 4, Claude Code	"Why is there no counterexample?"
4	Program + frontier	CBMC, Cryptol, SAW	"Does the source match the spec?"

A counterexample is one concrete behavior that violates the property; N is a step bound.

A single running example — a small "counter to 10" reactive system — appears in every day, in five different encodings (Day 4 has two).

Learning objectives for Day 1

By the end of today, you will be able to:

- Distinguish satisfiability, validity, and entailment in propositional and first-order logic.
- Define a transition system $T = (S, S_0, \rightarrow, AP, L)$ and write one for a small reactive system.
- Express a bounded-reachability question as an SMT formula over an appropriate theory.
- Drive Z3 from both SMT-LIB and Python; read the solver's `sat` / `unsat` / model output.
- Explain what bounded model checking does and does not prove.

L1 — Why now: AI × FM asymmetry

The bottleneck has flipped

- **Generative AI produces code, proofs, and engineering artifacts faster than any human can read.**
- The bottleneck is no longer *production*. It is *validation*.
- Formal methods is the branch of computer science designed to validate these artifacts mechanically.

Terence Tao has described large Lean formalization projects as a new mode of collaboration: contributors he has never met submit proof steps in a language he is still learning, and the proof assistant's kernel — not a human referee — arbitrates whether each step is correct (paraphrased; Scientific American, 2024).

Why now? The economics of cheap generation

- **Compute investment is rising sharply.** Global AI investment now runs, as an order-of-magnitude framing, at something like NASA's *entire annual budget every one-to-three weeks*; Sutton's "bitter lesson" says the methods that win are the ones that ride more compute — so generation keeps getting cheaper and better.
- **The flip side the hype skips:** for many concrete tasks a *small* or *neuro-symbolic* solution beats a frontier LLM — cheaper, faster, often more accurate (NVIDIA, *Small Language Models are the Future of Agentic AI*, arXiv 2506.02153). On **image-based string/arithmetic acceptance**, **our** neuro-symbolic finite/pushdown automata reached **70–100%** accuracy where GPT/Claude/Gemini-class VLMs scored **$\leq 50\%$** (and $\sim 0\%$ on multi-operator arithmetic) — at roughly **1000× lower latency** and a few dollars of compute (Sasaki, Lopez & Johnson, *Neurosymbolic Finite and Pushdown Automata*, NeuS 2025).
- **Both arrows point at verification.** Cheap generation buries us in artifacts to check — and the cost-effective, *small* artifacts are exactly the ones formal methods can still scale to.

Three signals: AI is shipping verifiable math

- **Polynomial Freiman–Ruzsa conjecture** — the informal proof (Gowers, Green, Manners, Tao) was posted November 2023; a public collaboration of 20+ people, led by Tao with Yael Dillies and Bhavik Mehta, formalized it in Lean in about three weeks (mid-November to early December 2023).
- **Gemini Deep Think — IMO 2025 gold medal** — 35/42 points, solved 5 of 6 problems within the 4.5-hour window, all in natural language.
- **DeepSeek-Prover-V2 (671B) — 88.9% on miniF2F** (April 2025) at a large pass@8192 sample budget (~82% at pass@32), the standard benchmark of competition-level theorems formalized in Lean 4.

Three counter-signals: AI is shipping broken code

- **A large fraction of LLM-generated code carries known security vulnerabilities** — studies report anywhere from ~30% to over 60% depending on language and prompt; one 2023 audit (FormAI, Tihanyi et al.) found ~51% of 112,000 C programs generated by GPT-3.5 contained at least one CWE-classified vulnerability (detected via formal verification with ESBMC).
- **~20% of LLM-recommended packages don't exist** — “slopsquatting” is now a named attack vector: an attacker registers the hallucinated package name.
- **AWS, by contrast, applies formal methods to both its crypto and its AI** — s2n-TLS’s crypto kernels are checked with SAW proofs on every commit (March 2026 added a HOL Light-verified AES-XTS to s2n-bignum), and AWS Bedrock’s **Automated Reasoning checks** now validate LLM outputs against formal policies with up to **99% verification accuracy** (2025).

This season alone: AI crosses two thresholds

- **Finding flaws at scale.** Anthropic's **Project Glasswing** (2026) turned a frontier model loose on critical software and surfaced **thousands of zero-day vulnerabilities** across every major OS and browser — including a **27-year-old** bug in OpenBSD and a **16-year-old** bug in FFmpeg that fuzzers had executed **five million times** and still missed. (anthropic.com/glasswing)
- **Proving new theorems.** In **May 2026** an **OpenAI** reasoning model autonomously produced a proof that **rules out the long-conjectured near-linear ($n^{1+o(1)}$) form of the bound** on Erdős's 1946 unit-distance problem — a fixed polynomial improvement (exponent δ later pinned to 0.014 by **Will Sawin**) — one of the first times AI has independently advanced a central open problem in a subfield. (openai.com)
- **The lesson.** Testing ran that FFmpeg line five million times and learned nothing; *reasoning about the code* found the bug. When AI both writes and breaks software, **proof — not more testing — tells you which side you are on.**

Coding has changed: generative → agentic

- **Generative AI** = *generating* stuff (text, code). **Agentic AI** = *generating + doing* — it plans, edits files, runs tools and tests, reads the errors, and iterates.
- Tools: **Claude Code**, Cursor, GitHub Copilot, OpenAI Codex — inside your editor (VS Code), wired to your shell and, via **MCP** (Model Context Protocol), to external tools and data.
- What that buys, from real recent use (our own, late 2025):
 - ~10,000 lines of **C#** → **Python** in ~3 hours; ~10k lines **MATLAB** → **Python**; new NNV benchmarks in ~1 hour.
 - Live lecture demos, Docker environments built during a Zoom interview, APIs learned on the fly.
- **The pattern that makes it work — *oracles***: agentic AI iterates wherever it can *check* itself — a parser, a simulation, a differential test against a known-good artifact, or a formal proof. Verification is the highest-confidence oracle.

The *generation* half of engineering just got cheap. That is exactly why the *validation* half — this course — becomes the bottleneck.

Vibe coding → vibe engineering

Karpathy named “**vibe coding**” (Feb 2025) — “give in to the vibes, forget the code exists.” A year on: “**agentic engineering**” — orchestrate agents while *acting as oversight*.

	Vibe coding	Agentic engineering
Code review	skipped	rigorous (like PR review)
Testing	minimal / absent	comprehensive suites
Planning	none	design docs + specs
Ownership	casual acceptance	full human responsibility

But AI-generated code is *often wrong*: Veracode’s 2025 GenAI report found security flaws in **~45%** of AI-written samples; Apiiro reported AI assistants ship code **~4× faster but with ~10× the vulnerabilities** (2025); MIT Sloan documents the *hidden costs* — technical debt that destabilizes systems (2025) (industry/vendor reports). (Plus the ~51% vulnerable C and ~20% hallucinated packages from before.) **You own what the AI writes.**

Dijkstra warned of exactly this in *On the Foolishness of “Natural Language Programming”* (EWD667, 1978): natural language is too ambiguous to be a safe programming medium. Vibe coding is that idea *automated*.

Generation is cheap; **correctness is the bottleneck**. That makes **verification the essential activity** — the subject of the next four days.

How this course was built — and why that's the point

This very deck is an agentic-engineering artifact:

- **Pulled** the source CS 6315 material from the LMS — 14 weeks of slides **and** lecture-video transcripts.
- **Analyzed and repurposed** it into this 4-day arc with an AI agent; **added** new material (the AI × FM framing, the figures you're seeing, the neural-network frontier).
- **Verified everything**: every Z3 query runs, every Lean *solution* module builds with no **sorry** (the starters carry intended ones), every SMV / CBMC / Cryptol example checks — the agent *proposed*, the tools *disposed*.

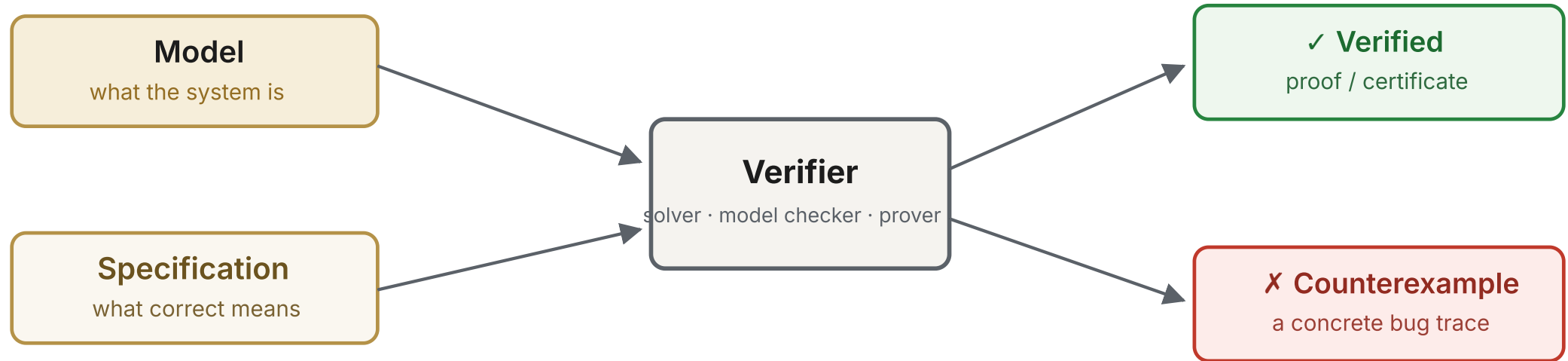
That last step is the whole difference between vibe coding and engineering — and it's the muscle the next four days build.

What this workshop teaches you to do

You should leave the course able to:

1. **Encode** a system you care about as a transition system.
2. **Specify** what “correct” means in propositional, temporal, or higher-order logic.
3. **Discharge** the specification via SAT/SMT (Day 1), model checking (Day 2), or theorem proving (Day 3).
4. **Connect** the spec back to actual code (Day 4) via CBMC and Cryptol/SAW.
5. **Use AI assistants** in this loop while keeping the kernel as the source of truth.

Verification in one picture



A **verifier** takes a **model** (what the system *is*) and a **specification** (what *correct* means) and returns one of exactly two things: a **proof** that *every* behavior meets the spec, or a **counterexample** — one concrete behavior that breaks it. Every tool this week is an instance of this picture.

The verification triple

Every formal-verification effort, regardless of tool, instantiates the same triple:

$$\left(\underbrace{\text{model}}_{\text{what the system is}}, \underbrace{\text{specification}}_{\text{what correct means}}, \underbrace{\text{proof}}_{\text{evidence}} \right)$$

Pillar	Model	Specification	Proof
SAT / SMT (Day 1)	propositional or QF-theory formula	(<code>check-sat</code>) query	unsat core, model, certificate
Model checking (Day 2)	finite transition system	CTL / LTL formula	invariant, BDD, or counterexample
Theorem proving (Day 3)	dependent type theory	a <code>theorem</code> statement	a term that inhabits the type
Program verification (Day 4)	imperative C program	Hoare triple / assertion	inductive invariant or refuted trace

Testing vs. proof, in two quotes

"Program testing can be used to show the presence of bugs, but never to show their absence!" — **Edsger W. Dijkstra**, *Notes on Structured Programming*, 1970

"Beware of bugs in the above code; I have only proved it correct, not tried it." — **Donald E. Knuth**, 1977

Testing **samples** behaviors; verification reasons about **all** of them. That gap is the whole reason this field exists — and exactly what the FFmpeg bug (run five million times by fuzzers, still missed) showed.

Famous bugs that motivated FM (1)

Therac-25 (1985–87) — radiation therapy machine, six known accidents, at least three deaths.

A race condition in operator input timing let the machine fire its high-current electron beam — the beam normally aimed at a target to generate X-rays — directly at the patient with the beam-spreading target retracted. Patients received on the order of 100× the intended dose.

- Root cause: software interlocks replaced hardware interlocks (Therac-20 had hardware safety; -25 did not).
- The same software bug existed in Therac-20 and never killed anyone because hardware caught it.
- Paper of record: Leveson & Turner, *Computer*, 1993.
sunnyday.mit.edu/papers/therac.pdf

Famous bugs (2)

Ariane 5, Flight 501 (1996) — roughly \$370M in lost hardware (rocket and four scientific satellites); exploded ~37 seconds after launch.

A 64-bit floating-point horizontal velocity overflowed when converted to a 16-bit signed integer in the inertial reference system. The conversion was unprotected, so the overflow raised an exception that shut the system down — and the backup, running identical code, had already failed the same way. The rocket veered off course and self-destructed.

- The bug was in code reused from Ariane 4, where the upper bound on horizontal velocity was lower.
- The Ariane 4 envelope made the conversion safe; nobody re-checked it for Ariane 5.
- Catching it is an overflow/range question — *does the 64-bit horizontal velocity always fit in a signed 16-bit integer across Ariane 5's actual flight envelope?* Range analysis and abstract interpretation (later, Astrée at Airbus) are built to decide exactly this; the reused Ariane 4 argument was simply never re-validated for the faster vehicle.

Famous bugs (3)

Intel Pentium FDIV (1994) — division returned wrong answer for ~1 in 9 billion inputs. Five entries in the floating-point divider's SRT lookup table — 5 of the ~1066 cells that should have held a value (the table has 2048 slots) — returned 0 instead of +2. The bug cost Intel **\$475M** in replacement chips and reshaped industry practice forever.

- After FDIV, Intel built an in-house formal-verification capability based on Symbolic Trajectory Evaluation and progressively made formal proof of arithmetic datapaths part of release sign-off.
- Today, Intel, AMD, Apple, ARM, and every major semiconductor company runs formal hardware verification at scale. Cadence JasperGold and Synopsys VC Formal are the products.
- Without FDIV, the modern formal-hardware industry might not exist.

Famous bugs (4)

Toyota unintended acceleration (2007–10) — multiple fatal accidents; \$1.2B U.S. Department of Justice criminal penalty (2014).

A NASA/NHTSA study (2010–11) examined the electronic throttle but could not confirm a software cause of large unintended acceleration. A separate code review by Michael Barr — expert testimony in *Bookout v. Toyota* (Oklahoma 2013) — found:

- Recursion and unbounded stack growth; stack overflow possible.
- Thousands of global variables — no encapsulation.
- A watchdog timer that could be silenced by the very task it was meant to monitor.
- The throttle could be driven open by a single bit-flip and stay there.

The jury awarded \$3M and found “reckless disregard.” Toyota recalled roughly 8–9 million vehicles.

Famous bugs (5)

Boeing 737 MAX MCAS (2018–19) — 346 deaths across two crashes.

The Maneuvering Characteristics Augmentation System was a software workaround for an aerodynamic change. It:

- Could push the nose down based on input from a *single* angle-of-attack sensor.
- Re-activated about 5 seconds after each pilot trim input, repeating in a cycle that fought the pilots.
- Was not documented in pilot training materials.
- Ran only with the autopilot already off, so “autopilot off” did not stop it — halting it required the stabilizer-trim cutout switches.

The fix compares *both* AoA sensors, fires MCAS only once per event, and caps its authority — plus revised training. The certification process itself failed: the FAA delegated assessment to Boeing.

Formal methods catch these too — sometimes first

The flip side of the disaster stories: formal methods finding the deep bug that testing misses.

- **Deep Space 1 Remote Agent** (NASA, 1999) — SPIN model checking of the spacecraft's autonomy software found **five** concurrency bugs the team agreed testing would never have caught. Months later the *same kind* of bug — a missing critical section racing two threads into a **deadlock** — actually froze the craft 96 million km from Earth. Model checking had flagged exactly that failure class beforehand.
- **Astrée + Airbus** — abstract interpretation **proved the absence of run-time errors** (overflow, divide-by-zero, out-of-bounds) across the A340/A380 fly-by-wire primary control software: hundreds of thousands of lines of floating-point C, tuned down to **zero false alarms**.

Deep interleaving-dependent races and arithmetic overflow are precisely the bugs testing almost never reaches — and exactly what this week's tools target.

These systems are complex — getting it all right is hard

Modern safety-critical systems are too large and intricate for anyone to get *everything* right by inspection. But **certain classes of question** — overflow, race, deadlock, off-by-one, missing case in a switch — are exactly the ones a machine can answer mechanically.

- Therac-25 → race conditions → CTL liveness properties checked by a model checker.
- Ariane 5 → integer overflow → CBMC's default overflow check.
- Pentium FDIV → arithmetic correctness → SAW + Cryptol equivalence proof.
- Toyota → stack/recursion → CBMC + abstract interpretation.
- MCAS → input redundancy → TLA+ specification of sensor voting.

This week's tools can answer all five.

Industrial deployment today (overview)

Domain	Tools in production	Example
Cloud crypto	SAW, Cryptol, Dafny, Lean	AWS s2n-TLS, s2n-bignum, Cedar policy language
Semiconductors	JasperGold, VC Formal, SymbiYosys	Apple Silicon, Intel, AMD, ARM
Avionics	SCADE, Astrée, SPARK/Ada	Airbus A380, Rafale, NASA
Automotive	CBMC, Polyspace, AbsInt	ISO 26262 ASIL-D components
Space	SPARK/Ada, Frama-C	SPARK powers Ariane 6 IRS
Verified OS	Coq, Isabelle	seL4 microkernel
Verified compilers	Coq	CompCert

AWS Provable Security in detail

AWS publishes their verification stack:

- **s2n-TLS** — crypto kernels (HMAC, DRBG) verified with **SAW** against Cryptol specs, re-run on every commit; deployed across AWS services.
- **s2n-bignum** — verified integer/crypto **assembly** (P-256/384/521, X25519, Ed25519, RSA, AES-XTS), each carrying a **HOL Light** machine-checked proof, run per commit.
- **AWS Cedar** — authorization policy language, **formally specified in Lean 4** (public `cedar-spec` repo); originally verified in **Dafny**, then migrated to Lean.
- **AES-XTS storage encryption** — used by EBS/Nitro, proved in HOL Light as part of s2n-bignum (March 2026 announcement).
- **Dafny for AWS authorization** — the core authorization engine has a Dafny spec + proof, validated against quadrillions of real requests (the separate Zelkova / IAM Access Analyzer tool uses SMT directly).

AWS's Provable Security group frames verification as a continuous-integration signal: like a test suite, a proof either passes or fails on every commit, and a failing proof blocks the change.

What formal methods does *not* give you

Honesty about the limits:

- **"Verified" means "matches the spec"** — a wrong spec, verified, is still wrong.
- **The model is not the system** — overflow, timing, hardware faults live outside most models.
- **Cost is real** — proofs and harnesses take effort; reserve them for what matters.
- **Undecidability/scale** bite — not everything is provable, and not everything that is, is fast.

When to reach for which tool

Situation	Reach for
"Is there a bug in the next N steps?"	SAT/SMT, bounded MC (Day 1, CBMC)
"Is this finite system ever wrong?"	model checking (Day 2)
"Unbounded / parametric / mathematical"	theorem proving (Day 3)
"Does this <i>code</i> match a spec?"	CBMC, SAW (Day 4)

No single tool dominates — match the question to the method.

The wider landscape: what this week samples

This week goes deep on four engines — SAT/SMT, model checking, theorem proving, source-level proof. They sit inside a much larger toolbox you should know exists:

Approach	Question it answers	Representative tools	Here?
Static analysis	"Any bug-shaped patterns, cheaply, at scale?"	Coverity, Clang SA, Infer, CodeQL	mentioned
Abstract interpretation	"A <i>sound</i> over-approximation of all runs?"	Astrée, IKOS, MOPSA	Day 4 kin
Testing & fuzzing	"Concrete inputs that break it?"	OSS-Fuzz, AFL++, KLEE, QuickCheck	contrast
Runtime verification	"Is <i>this</i> execution violating the spec, live?"	RV monitors, temporal-logic checkers	mentioned
Model-based design	"Generate code from a model that was checked"	Simulink/Stateflow, SCADE/Lustre	mentioned
Deductive verification	"Full functional correctness of real code"	Frama-C, Dafny, Why3, Viper	Day 3–4 kin
Type & refinement systems	"Correctness the compiler enforces for free"	Rust, LiquidHaskell, F*	Day 3 kin

The axis underneath them all: **automation** $\updownarrow$ **expressiveness**. Testing and static analysis are push-button but shallow; theorem proving is arbitrarily expressive but laborious. Choosing the lightest tool that still answers your question *is* the engineering judgment this week trains.

Live demo: Claude Code → Z3 → graph 3-coloring

We will:

1. Open Claude Code in VS Code.
2. Ask: "draft an SMT-LIB encoding for 3-coloring the complete graph K_4 ."
3. Run the resulting `.smt2` file with `z3`.
4. Observe `unsat` (4 vertices, 3 colors, every pair adjacent — impossible).
5. Change K_4 to the cycle C_5 and observe `sat` with a model.

New notation? `sat` / `unsat`, SMT-LIB, and Z3 all get a proper treatment in the SAT/SMT block later today — here, just watch the workflow.

The kernel arbitrates. The LLM proposes.

Quick check: SAT or UNSAT?

Symbol key (we define these properly next block): $\neg$ not, $\wedge$ and, $\vee$ or, $\rightarrow$ implies, $\forall$ for all, $\exists$ there exists, $\mathbb{Z}$ the integers.

For each formula, predict **sat** (some assignment makes it true) or **unsat** (none can):

1. $p \wedge \neg p$
2. $p \vee \neg p$
3. $(p \rightarrow q) \wedge (q \rightarrow r) \wedge p \wedge \neg r$
4. $\forall x. x + 0 = x$, in linear integer arithmetic
5. $\exists x, y \in \mathbb{Z}. x^2 + y^2 = 25 \wedge x > 0 \wedge y > 0$

Call out your prediction for each — then we'll run each one through Z3.

L1 recap

- Formal methods exists because the AI $\times$ human asymmetry has tipped: AI generates faster than humans can read.
- Every FM tool fits the **(model, specification, proof)** triple.
- Real bugs (Therac, Ariane, Pentium, Toyota, MAX) anchor each tool we'll see this week.
- Industry runs FM at scale: AWS, Intel, Apple, Airbus, NASA, the verified OS / compiler community.
- Pattern: **AI proposes, kernel disposes.**



Break — 10 minutes

Coffee, restroom, hallway questions.

We resume after the break with propositional and first-order logic.

L2 — Propositional + first-order logic; transition systems

What logic actually is (for our purposes)

A **logic** is a triple:

- **Syntax** — a grammar that decides which strings are formulas.
- **Semantics** — a function from formulas to truth-values, parametric in an *interpretation*.
- **Proof system** — rules for deriving formulas from formulas, sound and (ideally) complete w.r.t. the semantics.

For today: syntax + semantics. Proof systems return on Day 3.

Propositional logic — syntax

Atomic propositions: $p, q, r, \dots$ — variables each ranging over $\{\perp, \top\}$ (the two truth values: $\perp$ = false, $\top$ = true).

Formulas are built inductively:

$$\varphi ::= p \mid \neg \varphi \mid \varphi \wedge \varphi \mid \varphi \vee \varphi \mid \varphi \rightarrow \varphi \mid \varphi \leftrightarrow \varphi$$

Read this grammar as “a formula is an atom, or a negation, or two formulas joined by a connective.” ($::=$ means “is one of the following forms”; $|$ separates the alternatives.)

Connectives: $\neg$ not, $\wedge$ and, $\vee$ or (inclusive), $\rightarrow$ implies (if...then), $\leftrightarrow$ if and only if.

Examples:

- $p \wedge q$ — both
- $\neg p \vee q$ — equivalent to $p \rightarrow q$
- $(p \rightarrow q) \wedge (q \rightarrow p)$ — equivalent to $p \leftrightarrow q$

Propositional logic — semantics

A **valuation** v is a function $\text{Var} \rightarrow \{\perp, \top\}$.

Write $\llbracket \varphi \rrbracket_v$ for the **truth value (meaning) of φ under v** — either $\top$ (true) or $\perp$ (false). It is defined by recursion on the structure of φ :

φ	$\llbracket \varphi \rrbracket_v$ (its truth value under v)
p	$v(p)$
$\neg \varphi$	$\top$ iff $\llbracket \varphi \rrbracket_v = \perp$
$\varphi \wedge \psi$	$\top$ iff $\llbracket \varphi \rrbracket_v = \llbracket \psi \rrbracket_v = \top$
$\varphi \vee \psi$	$\top$ iff at least one is $\top$
$\varphi \rightarrow \psi$	$\top$ iff $\llbracket \varphi \rrbracket_v = \perp$ or $\llbracket \psi \rrbracket_v = \top$

We write $v \models \varphi$ (read “ v **satisfies** φ ”) if $\llbracket \varphi \rrbracket_v = \top$ — i.e. the assignment v makes φ come out true. The symbol $\models$ recurs all week: an assignment or model on the left, a formula on the right.

The three questions

For a propositional formula φ :

Question	Definition	Decision problem
Satisfiability	exists v with $v \models \varphi$	NP-complete (Cook 1971)
Validity	every v satisfies φ	co-NP-complete
Entailment	$\Gamma \models \varphi$ (Γ a set of premises) — every v that satisfies all of Γ satisfies φ	co-NP-complete (finite Γ ; check $\Gamma \cup \{\neg\varphi\}$ unsat)

Everything reduces to satisfiability:

- φ valid $\iff \neg\varphi$ unsatisfiable.
- $\Gamma \models \varphi \iff \Gamma \cup \{\neg\varphi\}$ unsatisfiable.

($\iff$ means “exactly when / if and only if”. *NP-complete* = verifiable in polynomial time given a witness; no known polynomial-time algorithm (a fast one would prove $P = NP$). *co-NP-complete* is the mirror image, for “always true” questions.)

Worked examples

Formula	Verdict	Witness
$p \wedge \neg p$	unsatisfiable	none — contradiction
$p \vee \neg p$	valid	every v ; classical tautology
$p \rightarrow (q \rightarrow p)$	valid	every v ; "weakening"
$(p \rightarrow q) \wedge (q \rightarrow r) \wedge (p \rightarrow r)$	satisfiable	$v(p) = v(q) = v(r) = \top$
$(p \rightarrow q) \wedge p \wedge \neg q$	unsatisfiable	modus ponens contradiction

The last row is an **entailment** check in disguise: $\{p \rightarrow q, p\} \models q$ holds *because* adding $\neg q$ makes the set unsatisfiable — exactly the $\Gamma \cup \{\neg\varphi\}$ trick from the previous slide.

Two arrows: $\vdash$ (provable) vs $\models$ (true)

There are two completely different ways to say “ φ follows from premises Γ ”:

- $\Gamma \models \varphi$ — **semantic** (“entails”): *every* interpretation that makes all of Γ true also makes φ true. About **meaning** — defined by the truth tables.
- $\Gamma \vdash \varphi$ — **syntactic** (“derives”): there is a *finite proof* of φ from Γ using a fixed set of **inference rules** (e.g. *modus ponens*: from a and $a \rightarrow b$, conclude b). About **symbol-pushing** — no truth values mentioned.

A proof system is judged by how these two line up:

- **Sound:** $\Gamma \vdash \varphi \Rightarrow \Gamma \models \varphi$ — “you can’t derive anything false.” Anything you *prove* really is *true*.
- **Complete:** $\Gamma \models \varphi \Rightarrow \Gamma \vdash \varphi$ — “everything true is derivable.” Nothing true escapes the proof system.

First-order logic — what's new

Adds:

- **Variables and quantifiers:** $\forall x. \varphi(x)$ ("for all x , φ holds") and $\exists y. \varphi(y)$ ("there exists a y for which φ holds").
- **Function symbols and predicate symbols** of fixed arity. Constants are 0-ary functions.
- **Terms** built from variables and function applications.

A first-order **signature** Σ lists the function and predicate symbols and their arities. A **structure** $\mathcal{M}$ for Σ gives:

- A nonempty domain $|\mathcal{M}|$ — the set of objects the quantifiers range over (e.g. all the integers).
- An interpretation of each function symbol as an actual function on $|\mathcal{M}|$.
- An interpretation of each predicate symbol as an actual relation on $|\mathcal{M}|$.

First-order satisfiability

$$\mathcal{M} \models \varphi$$

means "formula φ is true in structure $\mathcal{M}$ ".

- **Satisfiable:** exists $\mathcal{M}$ with $\mathcal{M} \models \varphi$.
- **Valid:** every $\mathcal{M}$ models φ .

First-order satisfiability is undecidable in general (Church, Turing, 1936). No algorithm decides it for every formula.

What saves us: **decidable fragments**.

Quantifier order changes the meaning

A **structure** (the integers $\mathbb{Z}$ with $+$, $<$, etc.) is what makes a quantified sentence true or false. The *order* of mixed quantifiers is not cosmetic — swap $\forall$ and $\exists$ and you usually change the claim:

Sentence	Reading	Over $\mathbb{Z}$
$\forall x. \exists y. y > x$	every x has <i>some</i> bigger y (depends on x)	true
$\exists y. \forall x. y > x$	<i>one</i> y beats <i>every</i> x at once	false

Same atoms, opposite verdicts. The first lets y depend on x ; the second demands a single y that works for all x .

- **Same-type swaps are safe:** $\forall x. \forall y. \varphi \equiv \forall y. \forall x. \varphi$, and $\exists x. \exists y. \varphi \equiv \exists y. \exists x. \varphi$.
- **De Morgan for quantifiers** (negation toggles the quantifier): $\neg \forall x. \varphi \equiv \exists x. \neg \varphi$ and $\neg \exists x. \varphi \equiv \forall x. \neg \varphi$.

Decidable theories that matter

Theory	Domain	Examples	Decidable?
EUF (equality + uninterpreted functions)	black-box functions	$f(a) = b \wedge f(b) \neq c$	yes
LIA (linear integer arithmetic)	$\mathbb{Z}$, linear	$3x + 2y = 7 \wedge x > 0$	yes
LRA (linear real arithmetic)	$\mathbb{R}$, linear	$x + y \leq 1 \wedge x \geq 0$	yes
BV (bit-vectors)	fixed-width words	$x \& (x - 1) = 0$	yes
Arrays	$A : I \rightarrow V$	$\text{store}(a, i, v)[i] = v$	yes
NIA (nonlinear integer arithmetic)	$\mathbb{Z}$, nonlinear	$x \cdot y = z \wedge \dots$	undecidable
NRA (nonlinear real arithmetic)	$\mathbb{R}$, nonlinear	$x^2 + y^2 = 25$	decidable (Tarski 1951); CAD (Collins 1975) is doubly-exponential

“*Linear*” = never multiply two unknowns; “*nonlinear*” allows $x \cdot y$ or x^2 . A *bit-vector* is an integer in a fixed number of bits (like a 32-bit machine word); $\&$ is bitwise-AND. For arrays, $\text{store}(a, i, v)$ is array a with index i set to v , and $a[i]$ reads index i . EUF treats each function as a black box — equal inputs give equal outputs, nothing more. Z3 supports all of these. The `(set-logic ...)` directive tells it which fragment to assume.

Sets, relations, functions (one-slide refresher)

A **set** is a collection. We write $a \in A$.

A **relation** R from A to B is a subset of $A \times B$.

A **function** $f : A \rightarrow B$ is a special relation: every $a \in A$ relates to exactly one $b \in B$.

For Day 1 specifically:

- A transition system's state space S is a set.
- The transition relation $\rightarrow$ is a binary relation on S .
- A labeling function $L : S \rightarrow 2^{AP}$ takes states to sets of true atomic propositions.

Transition systems

A **transition system** is a tuple

$$T = (S, S_0, \rightarrow, AP, L)$$

where:

- S — the set of states (possibly infinite)
- $S_0 \subseteq S$ — the initial states ($\subseteq$ = "is a subset of": every initial state is a state)
- $\rightarrow \subseteq S \times S$ — the transition relation (write $s \rightarrow s'$ for " s can step to s' ")
- AP — a set of atomic propositions
- $L : S \rightarrow 2^{AP}$ — a labeling (2^{AP} is the *powerset* of AP — all its subsets — so $L(s)$ is the set of propositions true in state s)

Heads-up: the arrow $\rightarrow$ here is the *transition* relation between states — a different use from logical "implies" $\rightarrow$ in the logic section.

A **trace** is a (finite or infinite) sequence $s_0, s_1, s_2, \dots$ with $s_0 \in S_0$ and $s_i \rightarrow s_{i+1}$ for all i .

A state s is **reachable** if some trace contains it.

The running example: the counter

State and input:

- $\text{mode} \in \{\text{off}, \text{on}\}$ ($\in$ = “is an element of”; mode is off or on)
- $x \in \mathbb{N}$ ($\mathbb{N}$ = the natural numbers 0, 1, 2, ...; the dynamics keep it in 0–10 — exactly what we’ll *prove*)
- $\text{press} : \text{Bool}$ (external input — true if the button is pressed this step)

Behavior:

- Initially $(\text{mode}, x) = (\text{off}, 0)$.
- From off, a press flips to on.
- In on, each step with !press increments x while $x < 10$.
- In on with press or $x \geq 10$, return to $(\text{off}, 0)$.

What we will verify, all week:

- $x \leq 10$ (safety)
- $\text{mode} = \text{off} \rightarrow x = 0$ (consistency)
- $x > 0 \rightarrow \text{mode} = \text{on}$ (consistency)

The counter, formally

$$S = \{\text{off}, \text{on}\} \times \mathbb{N}$$

$$S_0 = \{(\text{off}, 0)\}$$

Transition relation $\rightarrow$ (parameterized by a non-deterministic **press** at each step):

$$(\text{off}, x) \xrightarrow{\neg p} (\text{off}, x)$$

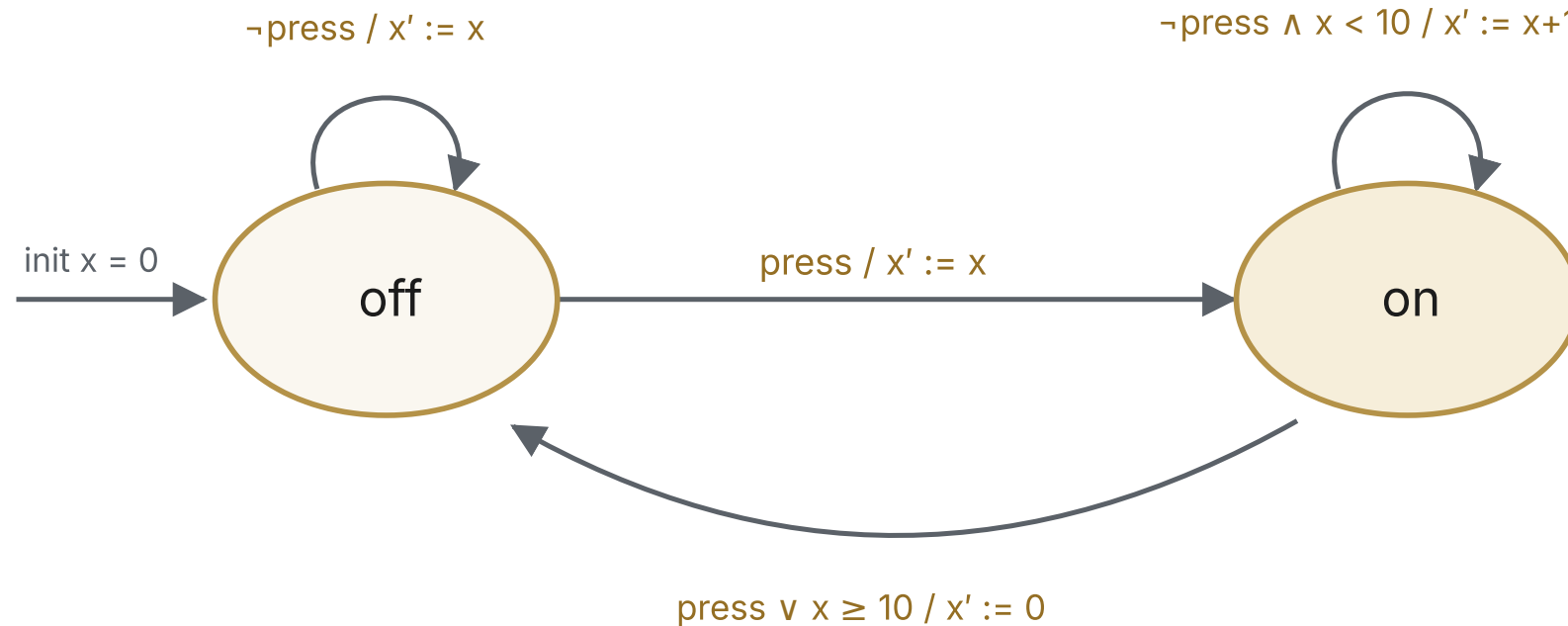
$$(\text{off}, x) \xrightarrow{p} (\text{on}, x)$$

$$(\text{on}, x) \xrightarrow{\neg p \wedge x < 10} (\text{on}, x + 1)$$

$$(\text{on}, x) \xrightarrow{p \vee x \geq 10} (\text{off}, 0)$$

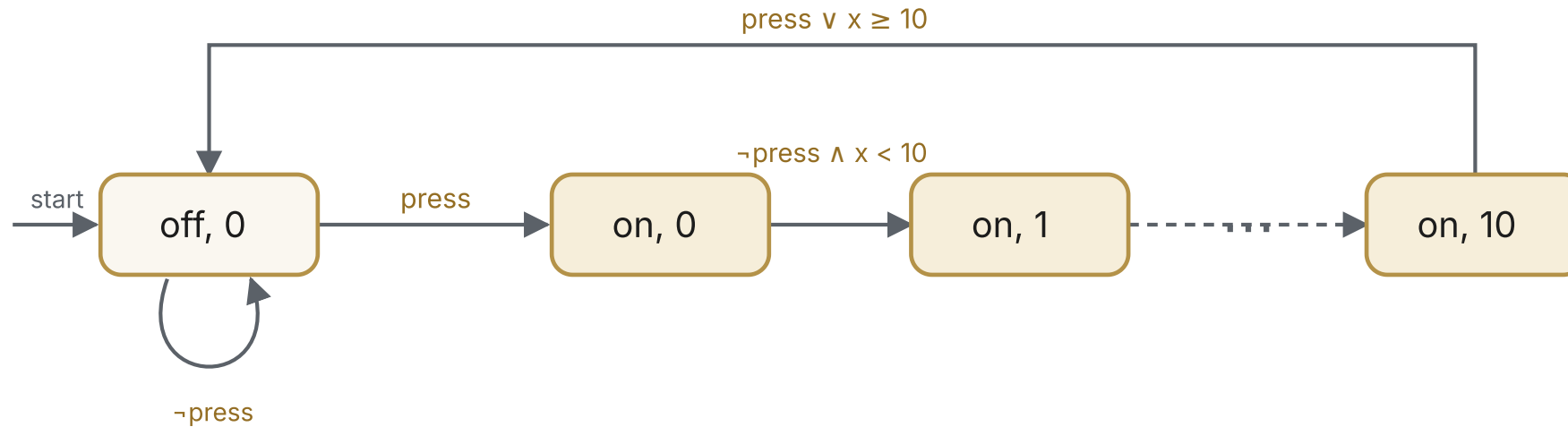
Atomic propositions: `mode_off`, `mode_on`, `x=0`, `x=10`, ... as needed.

The counter as a symbolic state machine



Two control modes; the count `x` stays a **variable**. Each edge is **guard / update** — exactly the symbolic transition relation φ_T over (s, s') . The *next* slide “unrolls” this into explicit states by enumerating `x = 0, 1, ..., 10`.

The counter as an explicit state machine



This **unrolls** the symbolic machine above: x becomes part of the state. The initial state `off, 0` (gold); `on, 0 ... on, 10` count up under `-press`; `press` or `x = 10` returns to `off`. Four drawn states stand in for the twelve reachable ones.

Reachable states: BFS by hand

Starting from (off, 0):

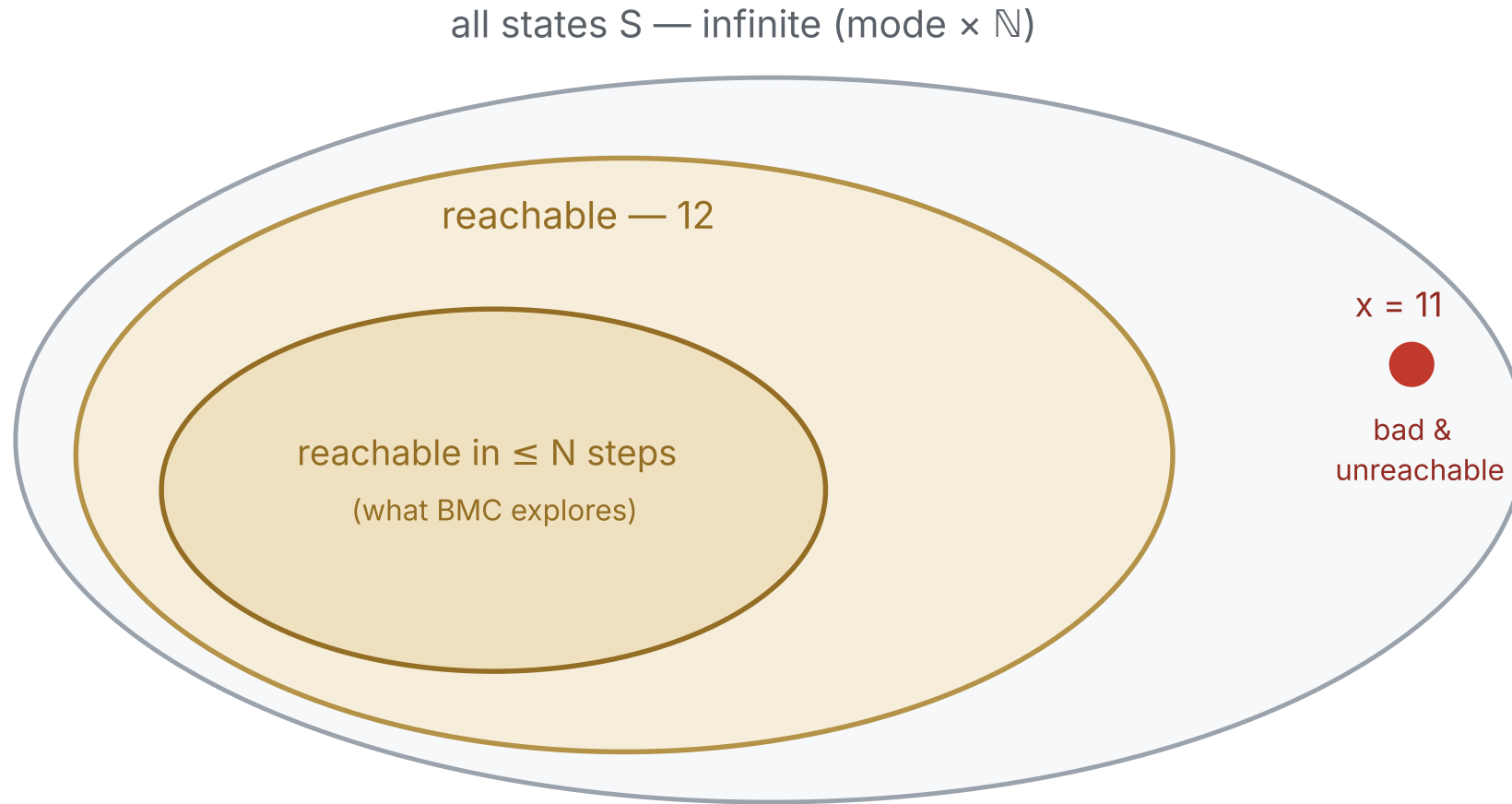
```
(off, 0)                ← initial
  ↓ press
(on, 0)
  ↓ !press
(on, 1) (on, 2) (on, 3) ... (on, 10)
  ↓ press (or x=10)
(off, 0)                ← back to initial
```

The state space is infinite ($x \in \mathbb{N}$), but the reachable set is just **12 states**:

$\{(\text{off}, 0)\} \cup \{(\text{on}, k) : 0 \leq k \leq 10\}$ — which is exactly why “is $x = 11$ ever reachable?” is a sharp safety question.

The safety property $x \leq 10$ holds on all twelve. ✓

Reachable, bounded, bad — the picture



- **All states** (infinite): every (mode, x) pair you could write down ($x \in \mathbb{N}$).
- **Reachable** (12): what the system can actually get to from $(\text{off}, 0)$.
- **Reachable in $\leq N$ steps**: what bounded model checking (BMC) explores — it grows toward the reachable boundary as N rises.

Quick check: which is which?

Match each English sentence to one of (a), (b), or (c).

1. "On every trajectory from any initial state, x stays in $[0, 10]$."
2. "There exists some trajectory along which $x = 10$ eventually."
3. "Whenever mode is off, x must be 0."

Choices (formulas in informal English):

- 1. Safety invariant
- 2. Reachability witness
- 3. Conditional invariant

L2 recap

- Logic = syntax + semantics. We live mostly in the SAT/SMT slice today.
- The three questions — satisfiability, validity, entailment — all reduce to satisfiability.
- First-order logic is undecidable in general; **decidable theories** (LIA, BV, EUF, Arrays, LRA) are what SMT solvers actually handle.
- A **transition system** $T = (S, S_0, \rightarrow, AP, L)$ is the substrate every tool this week operates on.
- The running example — the counter to 10 — has 12 reachable states; we'll verify " $x \leq 10$ " in five different ways.



Break — 5 minutes

We resume after the break with SAT, SMT, and live Z3.

L3 — SAT, SMT, and Z3

SAT — the canonical NP-complete problem

SAT: given a propositional formula φ , is there a valuation v with $v \models \varphi$?

Cook–Levin theorem (Cook 1971, Levin 1973, independently): SAT is NP-complete.

Two consequences:

1. Every problem in NP reduces (in polynomial time) to a SAT instance.
2. If SAT had a polynomial-time algorithm, $P = NP$.

In practice: SAT solvers handle millions of variables and clauses on industrial instances. The theoretical worst case is unreachable on real workloads.

Proving by refutation: the one move behind every solver

A SAT/SMT solver does essentially one thing: **find a model** (a satisfying assignment) — or report there is none (**UNSAT**). So how can it *prove* a property, which is a claim about *all* inputs?

Assert the negation and check satisfiability.

- ψ is **valid** (always true) iff $\neg\psi$ is **UNSAT**.
- P **entails** Q iff $P \wedge \neg Q$ is **UNSAT** — nothing makes the premises true and the conclusion false.

Socrates: assert $\forall x. \text{Man}(x) \rightarrow \text{Mortal}(x)$, $\text{Man}(\text{socrates})$, and $\neg\text{Mortal}(\text{socrates})$. The solver returns **unsat** $\Rightarrow$ the syllogism is valid.

This is the spine of the whole day: the live demos and bounded model checking all *assert that something bad is possible and let the solver fail to find it*.

CNF: the canonical input form

A **literal** is an atom or its negation: p or $\neg p$.

A **clause** is a disjunction of literals: $(p \vee \neg q \vee r)$.

A formula in **conjunctive normal form (CNF)** is a conjunction of clauses:

$$(p \vee \neg q) \wedge (\neg p \vee r) \wedge (q \vee \neg r)$$

Every propositional formula can be converted to CNF (Tseytin transformation, linear blow-up).

SAT solvers consume CNF.

DPLL — the first complete SAT algorithm

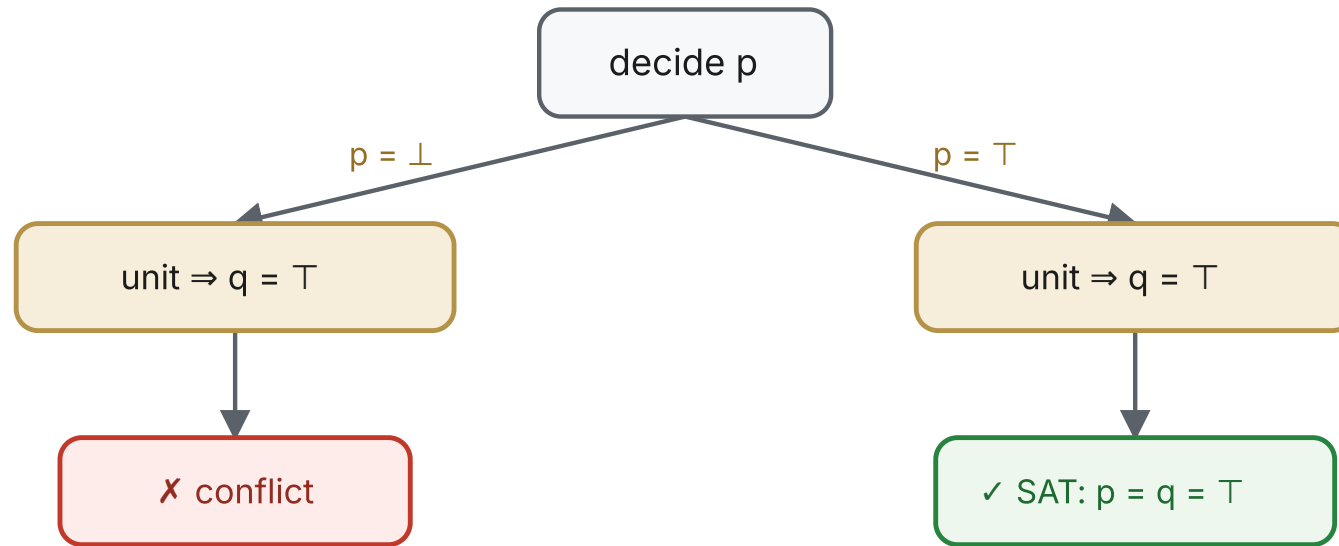
Davis–Putnam–Logemann–Loveland (1962):

```
DPLL( $\varphi$ ):  
  if no clauses left:      return SAT  
  if  $\varphi$  has empty clause: return UNSAT  
  apply unit propagation  
  apply pure literal elimination  
  choose unassigned literal  $\ell$   
  if DPLL( $\varphi \wedge \ell$ ): return SAT  
  if DPLL( $\varphi \wedge \neg\ell$ ): return SAT  
  return UNSAT
```

The kernel of every modern SAT solver. **CDCL** (Conflict-Driven Clause Learning, late 1990s) keeps the DPLL skeleton but adds clause learning from each backtrack — turning each conflict into a permanent constraint.

DPLL in action: a tiny trace

Solve $\varphi = (p \vee q) \wedge (\neg p \vee q) \wedge (p \vee \neg q)$. Decide a variable, propagate the forced literals, backtrack on conflict.



- $p = \perp$: clause $(p \vee q)$ forces $q = \top \rightarrow$ now $(p \vee \neg q)$ has all literals false: **conflict**, backtrack.
- $p = \top$: clause $(\neg p \vee q)$ forces $q = \top \rightarrow$ every clause satisfied: **SAT**, model $p = q = \top$.

DPLL's two automatic rules: unit propagation + pure literals

Before DPLL ever *guesses*, it applies two rules that force assignments for free:

- **Unit propagation** — a clause with one unassigned literal left *forces* that literal true (it's the only way to satisfy the clause). One forced literal often shrinks other clauses to units → a **cascade**.
- **Pure literal** — a variable that appears with only *one* polarity (always x , or always $\neg x$) in the remaining clauses can be set to satisfy all of them. It can never cause a conflict, so set it and drop those clauses.

Worked run on

$$\varphi = (\neg a \vee b) (\neg b \vee c) (a \vee c) (d \vee \neg c) (e).$$

Step	Rule	Action	Result
1	unit	clause (e) is a unit	$e = \top$
2	pure	d appears only positively	$d = \top$, drop $(d \vee \neg c)$
3	decide	guess on a	$a = \top$
4	unit	$(\neg a \vee b)$ now forces b	$b = \top$
5	unit	$(\neg b \vee c)$ now forces c	$c = \top$
6	done	$(a \vee c)$ already true	SAT : $a=b=c=d=e=\top$

Boolean resolution: a proof of UNSAT

When a formula is unsatisfiable, DPLL's failure is itself a **proof** — and the proof rule is **resolution**:

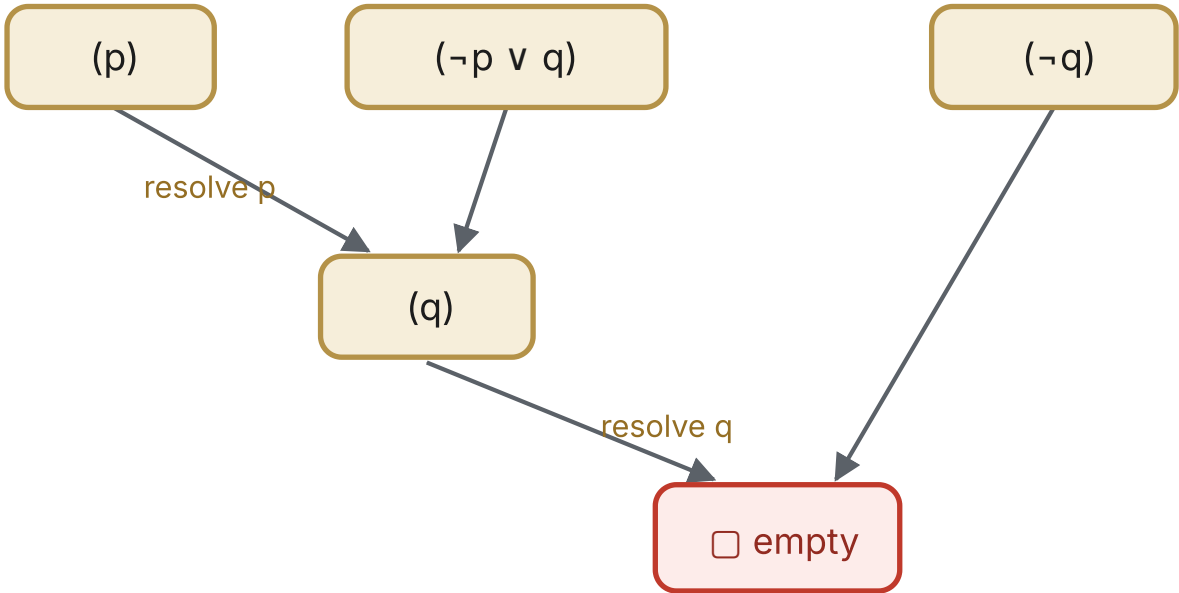
$$\frac{(A \vee \ell) \quad (B \vee \neg \ell)}{(A \vee B)}$$

Two clauses sharing a variable ℓ with *opposite* signs resolve into a new clause that drops ℓ (the **resolvent**). Keep resolving; if you ever derive the **empty clause** $\square$ (a clause with no literals — unsatisfiable by definition), the original formula is UNSAT.

Refute $\varphi = (p) (\neg p \vee q) (\neg q)$:

#	Clauses resolved	On	Resolvent
1	$(p), (\neg p \vee q)$	p	(q)
2	$(q), (\neg q)$	q	$\square$

Empty clause derived $\Rightarrow$ **UNSAT**. This is a checkable certificate: a referee re-runs the two steps without trusting the solver.



Cores and certificates: trust a tiny checker

The resolution proof above is a **certificate** — a small, independent checker re-validates “UNSAT” without trusting the solver. Solvers expose two practical handles (SMT-LIB commands):

- **(get-proof)** — the full derivation of $\perp$. A ~hundred-line checker can validate a solver that is hundreds of thousands of lines. *This is “AI proposes, the kernel disposes” at the solver level.*
- **(get-unsat-core)** — a *minimal* subset of your assertions that is already contradictory. It pinpoints **which** assumptions clashed — invaluable when a spec comes back **unsat** and you don’t know why.

A **sat** answer ships a **model** you can plug in and check. An **unsat** answer ships a **proof** you can re-check. Either way, you never have to trust the solver itself.

What makes modern SAT solvers actually fast

Four key engineering choices, each circa 1996–2003:

1. **CDCL** — learn from every conflict; the search shrinks as you go.
2. **Watched literals** — propagate without scanning every clause.
3. **VSIDS branching** (Variable State Independent Decaying Sum) — prefer literals that recently appeared in conflicts.
4. **Restart policies** — abandon the search tree periodically, keep learned clauses.

Result: SAT competition benchmarks went from “100 variables, sometimes” in 1990 to “millions of variables, routinely” in 2025.

SMT — SAT + theories

SMT = Satisfiability Modulo Theories.

Take a SAT solver. Add a **theory solver** that can decide formulas over a specific structure (integers, reals, bit-vectors). When the SAT layer guesses a propositional assignment, the theory solver checks whether that assignment is consistent over the theory.

DPLL(T) framework (Ganzinger, Hagen, Nieuwenhuis, Oliveras, Tinelli, 2004):

```
SAT solver guesses literal assignments → theory solver checks consistency
                                         → if inconsistent, return T-conflict
                                         → SAT learns the negation of the
                                           inconsistent partial assignment
```

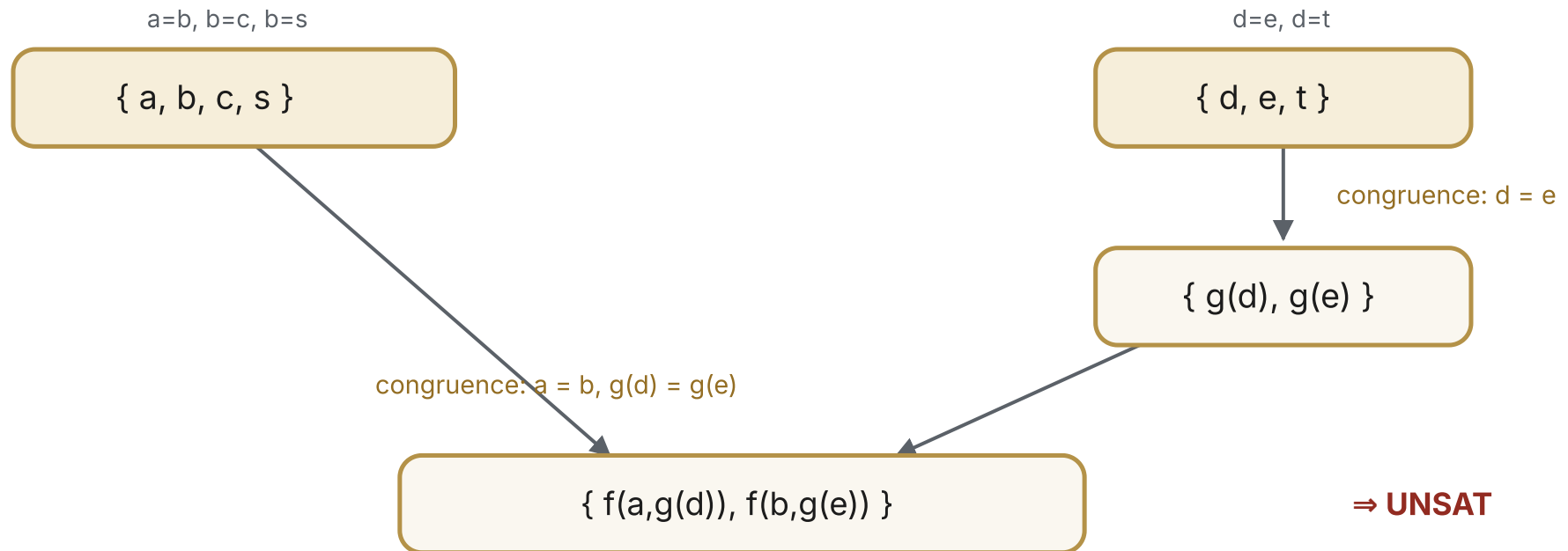
The SAT engine drives; the theories arbitrate.

Inside a theory solver: equality + uninterpreted functions

How does the **EUF** theory solver decide a conjunction of equalities? **Congruence closure**:

1. Put each term in its own class; **merge** classes joined by an **=**.
2. **Congruence rule**: if $x_1 = y_1, \dots, x_n = y_n$ then $f(x_1, \dots) = f(y_1, \dots)$ — merge those too.
3. A disequality $u \neq v$ with u, v in the **same** class $\Rightarrow$ **UNSAT**.

Decide $a = b, b = c, d = e, b = s, d = t, f(a, g(d)) \neq f(b, g(e))$:



The asserted $f(a, g(d)) \neq f(b, g(e))$ lands *inside* the gold class — contradiction.

The DPLL(T) loop: SAT engine $\rightleftarrows$ theory solver

The congruence-closure procedure (previous slide) is one **theory solver**. DPLL(T) wires it to the SAT engine in a loop. First, **abstract** each theory atom to a fresh Boolean:

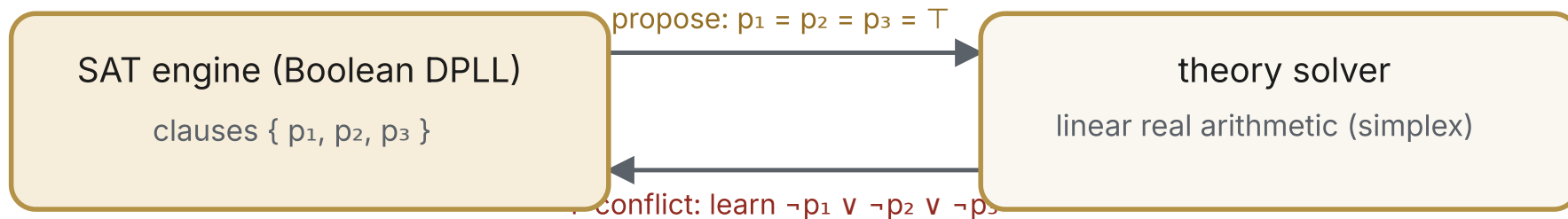
$$\underbrace{x \geq 0}_{p_1} \quad \underbrace{y \geq 0}_{p_2} \quad \underbrace{x + y < 0}_{p_3}$$

Now the SAT engine sees only $p_1 \wedge p_2 \wedge p_3$ and reasons Boolean-only. The cycle:

1. **SAT proposes** a Boolean assignment: $p_1 = p_2 = p_3 = \top$.
2. **Theory solver checks** the *concrete* meaning for T -consistency: is $x \geq 0 \wedge y \geq 0 \wedge x + y < 0$ satisfiable over the reals? **No**.
3. Theory solver returns one of:
 - **T -conflict** — the assignment is theory-inconsistent. Hand back a **theory lemma** $\neg(p_1 \wedge p_2 \wedge p_3)$, i.e. $\neg p_1 \vee \neg p_2 \vee \neg p_3$.
4. **SAT learns** that lemma as a new clause and **backjumps** — it will never propose all three together again.

Loop until the SAT engine finds a theory-consistent model (**sat**) or runs out of assignments (**unsat**).

DPLL(T) round, drawn



$x \geq 0 \wedge y \geq 0 \wedge x+y < 0 : \text{UNSAT over } \mathbb{R}$

The SAT engine and the theory solver pass messages until they agree:

- **propose** → SAT sends a candidate Boolean assignment.
- **check** → theory solver tests it over the real structure ($\mathbb{Z}$, $\mathbb{R}$, bit-vectors, ...).
- **respond** → **T**-conflict (learn a clause), **T**-propagate (force a literal), or **T**-consistent (accept).

When the theory solver finally says *consistent*, that Boolean model **plus** the theory witness is the SMT model.

SMT-LIB: the standard input language

```
(set-logic QF_LIA)           ; quantifier-free LIA
(declare-const x Int)        ; declare integer x
(declare-const y Int)
(assert (= (+ x y) 7))        ; x + y = 7
(assert (> x 0))              ; x > 0
(assert (> y 0))              ; y > 0
(check-sat)                  ; → sat
(get-model)                  ; → ((x 1) (y 6)) or similar
```

- **set-logic** declares which theories you'll use.
- **declare-const** introduces a free constant.
- **assert** adds a constraint.
- **check-sat** returns **sat**, **unsat**, or **unknown**.

Run it: **z3_smtlib_demo.smt2**.

Common Z3 logics worth knowing

Logic	Theory	Common use
QF_LIA	Linear integer arithmetic	counter steps, loop bounds, simple invariants
QF_LRA	Linear real arithmetic	hybrid systems, neural network bounds
QF_BV	Bit vectors of fixed width	crypto, hardware, popcount
QF_UF	Uninterpreted functions	abstract data types, reasoning about opaque APIs
QF_AUFLIA	Arrays + uninterpreted functions + LIA	memory safety, heap reasoning
LIA	LIA + quantifiers	induction-style reasoning
ALL	"Z3, pick"	exploration; never for production

SMT in the wild: generating a test case

We don't only ask "is this safe?" — we can ask the solver to **build an input** that drives code down a chosen path. Take Euclid's GCD:

```
unsigned GCD(unsigned x, unsigned y) {    // requires y > 0
    while (true) {
        unsigned m = x % y;
        if (m == 0) return y;
        x = y;
        y = m;
    }
}
```

Goal: find inputs that make the loop run *exactly twice*. We can't know the trip count by hand — so we ask Z3.

The key step: single static assignment (SSA)

A variable is reassigned each iteration, but a formula can't reassign anything. **SSA** fixes this: give each write a **fresh subscripted name** ($x_0, x_1, \dots$), then conjoin one equation per statement. Unroll two iterations:

$$\begin{array}{ll} (y_0 > 0) & \text{precondition} \\ \wedge (m_0 = x_0 \bmod y_0) \wedge \neg(m_0 = 0) & \text{iter 1: didn't return} \\ \wedge (x_1 = y_0) \wedge (y_1 = m_0) & \text{iter 1: updates} \\ \wedge (m_1 = x_1 \bmod y_1) \wedge (m_1 = 0) & \text{iter 2: returned} \end{array}$$

`check-sat` → **sat**, with model $x_0 = 2$, $y_0 = 4$ (then $m_0 = 2$, $x_1 = 4$, $y_1 = 2$, $m_1 = 0$). So `GCD(2, 4)` runs the loop exactly twice. ($a \bmod b$ = remainder; the subscripts are *versions*, not array indices.)

Code: `day01/examples/z3_test_gen.py` — runs this encoding (parameterized by `length` and `bits`), then asks Z3 for a *new* input each iteration via a blocking clause; toggle `opt_integer` to compare Int vs BitVec performance.

Z3 from Python (live demo)

```
import z3

x = z3.Int("x")
y = z3.Int("y")
s = z3.Solver()
s.add(x + y == 7, x > 0, y > 0)
print(s.check())    # sat
print(s.model())    # [y = 1, x = 6] (or similar)
```

The Python binding gives you:

- Same expressive power as SMT-LIB.
- Easy quantifier and array support.
- Iterative use (incremental solving with `push` / `pop`).
- Programmatic model extraction.

We'll run `day01/examples/z3_smoke.py` live.

Live: pigeonhole in SAT

`day01/examples/z3_pigeonhole.py` — encode " $n + 1$ pigeons into n holes" as SAT.

```
def pigeonhole(num_pigeons: int, num_holes: int):
    s = z3.Solver()
    p = [[z3.Bool(f"p_{i}_{j}") for j in range(num_holes)]
          for i in range(num_pigeons)]
    for i in range(num_pigeons):
        s.add(z3.Or(p[i]))                # each pigeon in  $\geq 1$  hole
    for j in range(num_holes):
        for i1 in range(num_pigeons):
            for i2 in range(i1 + 1, num_pigeons):
                s.add(z3.Not(z3.And(p[i1][j], p[i2][j]))) # no sharing
    return s.check()                          #  $\rightarrow$  unsat for all  $n \geq 1$ 
```

For $n = 1, 2, 3, 4, 5$: every answer is **unsat**. The pigeonhole principle, verified by machine.

Live: Sudoku as constraints, not search

`day01/examples/puzzles/sudoku.py` — one integer per cell, three “all-different” rules, plus the clues.

```
cells = [[z3.Int(f"c_{r}_{c}") for c in range(9)] for r in range(9)]
for r in range(9):
    for c in range(9):
        s.add(cells[r][c] ≥ 1, cells[r][c] ≤ 9)           # each cell holds a digit 1-9
for r in range(9):
    s.add(z3.Distinct(cells[r]))                          # each row: all 9 different
for c in range(9):
    s.add(z3.Distinct([cells[r][c] for r in range(9)]))   # each column: all different
for br in range(3):
    for bc in range(3):
        s.add(z3.Distinct([cells[3*br+dr][3*bc+dc]
                           for dr in range(3) for dc in range(3)]))
# pin the given clues, then s.check() / s.model()
```

We never write a backtracking search — we *state what a solution is* and let Z3 find one. (`z3.Distinct(xs)` = “these values are pairwise unequal”.)

Same shape, more puzzles in `examples/puzzles/`: **N-Queens**, **KenKen**, **magic squares**.

Sudoku at the Boolean level: the pure-SAT encoding

5	3			7				
6			1	9	5			
	9	8					6	
8				6				3
4			8		3			1
7				2				6
	6					2	8	
			4	1	9			5
				8			7	9

The **Distinct** version is SMT *syntactic sugar*. In **pure SAT** there are no integers — only Booleans (this is the classic Rosen encoding, and what bit-blasting produces internally):

- **One Boolean per (cell, value):** $p(i, j, n)$ = "row i , col j holds n " — $9 \cdot 9 \cdot 9 = 729$ variables.
- **Givens:** a unit clause $p(i, j, n)$ for each shaded clue.
- **Every row / column / block contains every n :** one disjunction $\bigvee$ over the nine positions, per value n (the block clauses are the fiddly ones).
- **No cell holds two numbers:** $\neg p(i, j, n) \vee \neg p(i, j, n')$ for every $n \neq n'$.

Same puzzle, two levels: **81 integers + Distinct** (SMT, previous slide) vs **729 Booleans + clauses** (SAT). SMT just hides the bit-

Bounded model checking: from SAT to system verification

The pattern for asking “can the counter reach $x = 11$ in $\leq N$ steps?”:

1. Introduce state variables for each step $k = 0, \dots, N$:

$$\text{mode}_0, x_0, \text{mode}_1, x_1, \dots, \text{mode}_N, x_N$$

2. Assert the initial constraint: $\text{mode}_0 = \text{off} \wedge x_0 = 0$.
3. For each k , assert the transition relation: $T(\text{mode}_k, x_k, \text{press}_k, \text{mode}_{k+1}, x_{k+1})$.
4. Assert the property's negation at step N : $x_N = 11$.
5. **check-sat**. SAT $\Rightarrow$ counterexample of length $\leq N$. UNSAT $\Rightarrow$ no counterexample of length $\leq N$ (says nothing about longer paths).

BMC, written as one formula

The five steps collapse into a single formula. Let $I(s)$ mean " s is initial", $T(s, s')$ mean " s can step to s' ", and $p(s)$ be the safety property. The **bounded unrolling** to depth k :

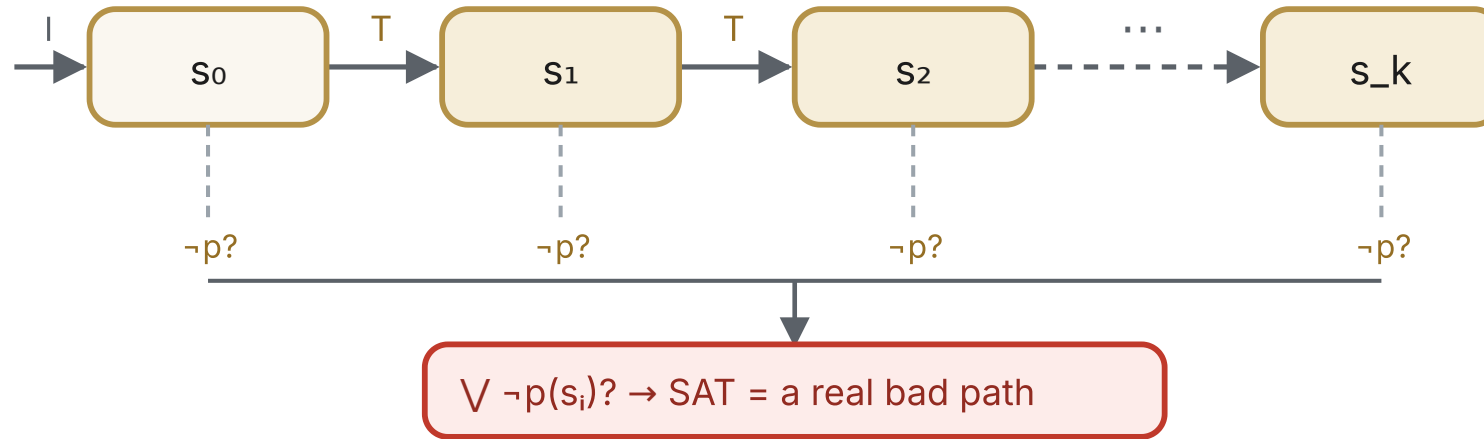
$$W(k) = \underbrace{I(s_0)}_{\text{start legal}} \wedge \underbrace{\bigwedge_{i=0}^{k-1} T(s_i, s_{i+1})}_{\text{a real } k\text{-step path}} \wedge \underbrace{\bigvee_{i=0}^k \neg p(s_i)}_{p \text{ fails somewhere}}$$

Read it as three demands at once: *begin in an initial state, follow the transition relation for k steps, and break p at some step.*

$p \text{ holds on all paths of length } \leq k \iff W(k) \text{ is UNSAT}$

- **SAT** $\rightarrow$ the model *is* a concrete counterexample trace $s_0 \rightarrow s_1 \rightarrow \dots$ that reaches a bad state.
- **UNSAT** $\rightarrow$ no violation within k steps (says **nothing** about step $k+1$).

The unrolling, drawn



Every step is a fresh copy of the state variables; T chains them; the property p is tested at each copy, and the $\bigvee$ asks “did it fail anywhere?”

When does bounded become complete?

BMC is a bug-finder: UNSAT at depth k only certifies “safe for $\leq k$ steps.” When can we stop and claim “safe, **ever**”?

- **Completeness threshold (CT)** — a depth such that UNSAT up to CT implies the property holds at *every* depth. If you check that far and still get UNSAT, you’ve actually proved it.
- One sound (if loose) value of CT is the **diameter**: the longest shortest-path between any two reachable states — once you’ve unrolled past it, every reachable state has already appeared.

The catch (our own caveat): **computing the exact CT is as hard as model checking itself**. In practice we use an over-approximation, and often just run out of resources first.

- Good case: systems *without counters* (e.g. some hardware) have small diameters — BMC closes quickly.
- Bad case: a counter to N has diameter $\sim N$; deep bugs hide past any practical k .

Worked BMC: the two-bit counter

The smallest system where the bound *matters*. Two bits ℓ, r count $00 \rightarrow 01 \rightarrow 10 \rightarrow 11 \rightarrow 00$; property $p = \neg(\ell \wedge r)$ ("never both bits set"):

$$I : \neg\ell \wedge \neg r \qquad T : \ell' = (\ell \oplus r) \wedge r' = \neg r$$

($\oplus$ = exclusive-or; primes = next state.) Unroll and ask $W(k)$:

k	states reached	$\neg p$ hit?	$W(k)$
2	00, 01, 10	no	UNSAT — safe so far
3	00, 01, 10, 11	yes at s_3	SAT — counterexample!

The model at $k = 3$ is the trace $00 \rightarrow 01 \rightarrow 10 \rightarrow 11$ — the solver hands you the exact path to the bug.

Live: bounded counter in Z3

day01/examples/z3_counter_bounded.py

```
def bounded_reach_to(forbidden_x: int, num_steps: int):
    s = z3.Solver()
    mode = [z3.Int(f"mode_{k}") for k in range(num_steps + 1)]
    x      = [z3.Int(f"x_{k}") for k in range(num_steps + 1)]
    press = [z3.Bool(f"press_{k}") for k in range(num_steps)]

    s.add(mode[0] == MODE_OFF, x[0] == 0)
    for k in range(num_steps):
        s.add(step(mode[k], x[k], press[k], mode[k+1], x[k+1]))
    # bad state reachable at ANY step k ≤ N (the "≤ N" of BMC)
    s.add(z3.Or([x[k] == forbidden_x for k in range(num_steps+1)]))
    return s.check()
```

For `forbidden_x = 11`: UNSAT for every bound we try (5, 10, 15, 20, 30).

Bounded vs full verification — the gap

No counterexample



Bounded BMC (Day 1): of length $\leq N$ ← does not generalize to all N

Full model check (Day 2): ever ← Day 2 closes this gap

Inductive proof (Day 3): ever, by induction ← Day 3 also closes it

Day 1 to Day 2: lift " $\leq N$ steps" to "all steps".

Day 1 to Day 3: drop the finite state requirement.

Day 1 to Day 4: apply the same idea to C source code.

Quick check: predict the verdict

For each, predict SAT / UNSAT *before* I run it.

1. `(check-sat)` with $(x + y = 7) \wedge (x > 0) \wedge (y > 0)$ in LIA — single step.
2. The counter, asking “can $x = 11$ in ≤ 5 steps?”
3. The counter, asking “can $x = 10$ in ≤ 11 steps?”
4. The pigeonhole with 100 pigeons into 99 holes.
5. The counter, asking “can `mode = on` $\wedge x = 0$ in ≤ 1 step?”

Take a guess for each — then we run each.

L3 recap

- **SAT** = boolean satisfiability; NP-complete; industrially routine for 10^6+ variables.
- **CDCL + watched literals + restarts** is what makes SAT solvers actually fast.
- **SMT** = SAT + decidable theories, via the DPLL(T) framework.
- **Z3** is the canonical SMT solver; SMT-LIB and Python both work.
- **Bounded model checking** is a one-pattern application: unroll the transition relation, assert bad-state at the end, check sat.
- BMC **refutes** counterexamples of length $\leq N$. It does *not* prove the property.

Wrap + post-session work

Day 1 in one slide

- The bottleneck has flipped from production to validation; FM is the substrate.
- Every FM tool fits **(model, specification, proof)**.
- Logic gives us the language for specs; transition systems give us the model.
- SAT and SMT make logical questions about transition systems mechanically decidable.
- Bounded SAT refutes short counterexamples but does *not* prove safety; Day 2's nuXmv lifts that limit.

Day 2 preview

- Same transition system. New tool: **nuXmv**.
- The bounded SMT formula we wrote today turns into an unrolling inside nuXmv's bounded model checker — but nuXmv *also* does full reachability via BDDs.
- **CTL and LTL** — temporal logic for properties that talk about “always”, “eventually”, “until”.
- Live counterexamples on broken specifications.

In-session exercise (15 min, your laptop)

Adapt `day01/examples/z3_counter_bounded.py` :

1. **Find the smallest N for which $x = 10$ is reachable.** Iterate from $N = 1$.
2. When you hit `sat`, ask Z3 for the model. Print the `press` sequence that gets $x = 10$.
3. Confirm the sequence makes sense by reading it off.

You should land on $N = 11$. The first press flips off $\rightarrow$ on; the next 10 are `press = False`, each incrementing x by 1.

Homework (not graded; for depth)

Pick **one**:

(a) Two-counter system. Two counters x and y , each independently bounded by 10. Each step, the system chooses which to increment. Verify $(x + y) \leq 20$ is reachable but not exceeded.

(b) Traffic light. Three states red/yellow/green. Bounded check that “green directly to red, skipping yellow” is *not* reachable in ≤ 20 steps.

(c) Your own. Any small reactive system: vending machine, turnstile, coffee maker. One safety property, encoded in Z3.

Submit (or just bring along to Day 2): the `.py` file, the verdicts at $N = 5, 10, 20$, and a one-paragraph note on what surprised you.

References for Day 1

- **Cook, S.** *The complexity of theorem-proving procedures*. STOC 1971. The original SAT-is-NP-complete paper.
- **Davis, Logemann, Loveland.** *A machine program for theorem-proving*. CACM 1962. The DPLL paper.
- **de Moura, Bjørner.** *Z3: An efficient SMT solver*. TACAS 2008. The Z3 tool paper.
- **Plotkin.** *A structural approach to operational semantics*. JLAP 60–61 (2004). Structural operational semantics — defining program behavior via labelled transition systems.
- **Clarke, Henzinger, Veith, Bloem (eds.).** *Handbook of Model Checking*. Springer 2018. Chapters 1–3 cover today's material in depth.
- **Z3 Guide** (Microsoft): microsoft.github.io/z3guide — interactive Z3 in the browser.

Full reference list at the end of the repo [README.md](#).

Next session: Day 2

GitHub repo: github.com/ttj/fmaiv (public). Slides also published at ttj.github.io/fmaiv.
Questions between sessions are welcome — reach out anytime.