Model Checking

An Overview

Goals

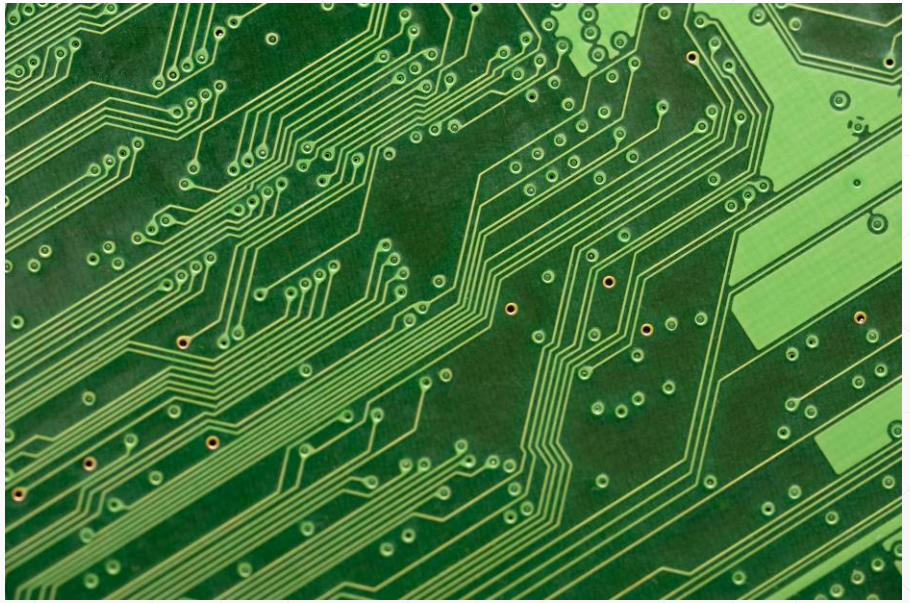
- Vocabulary
- Modeling
- Specification
- High-level understanding of algorithms

Outline

- What is Model Checking?
  - Modeling: Transition Systems
  - Specification: Linear Temporal Logic
- Historical Verification Approaches
  - Explicit-state
  - BDDs
- SAT/SMT-based Verification Approaches
  - Bounded Model Checking
  - K-Induction
- Inductive Invariants

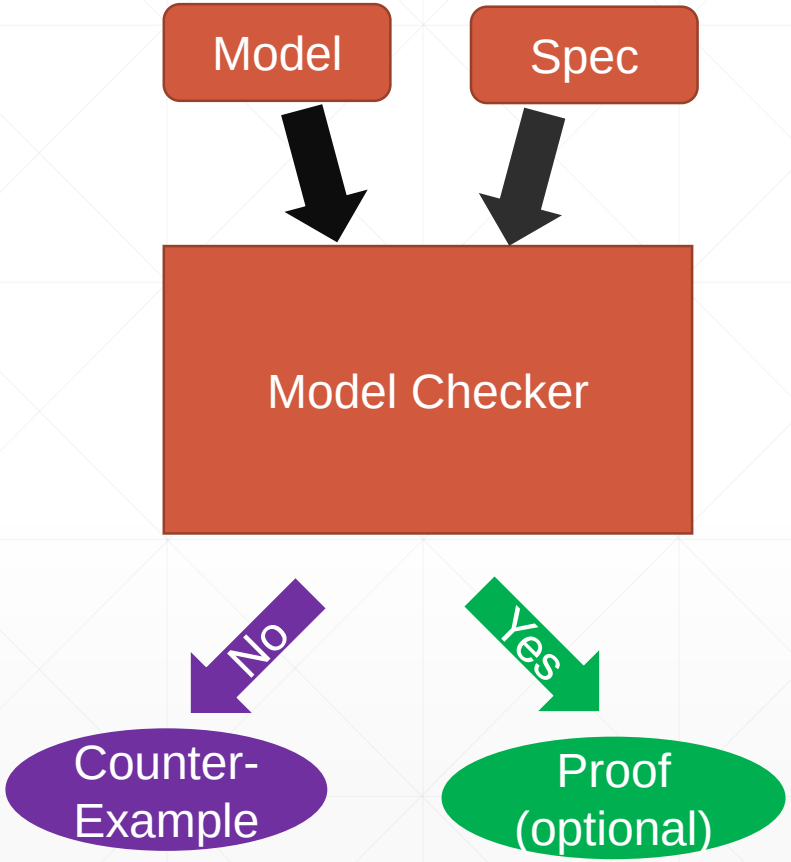
Motivation

- Safety-critical systems
  - Airplanes
  - Space shuttles
  - Railways
- Expensive mistakes
  - Chip design
  - Critical software
- Want to guarantee safe behavior over unbounded time



What is Model Checking?

- An approach for verifying the temporal behavior of a system
- Primarily fully-automated (“push-button”) techniques
- Model
  - Representation of the system
  - Need to decide the right level of granularity
- Specification
  - High-level desired property of system
- Considers infinite sequences
- PSPACE-complete for FSMs



Modeling: Transition Systems

- Model checking typically operates over *Transition Systems*
  - A (symbolic) state machine

- A Transition System is $\langle S, I, T \rangle$
  - S: a set of states
  - I: a set of initial states (sometimes use $Init$ instead of I for clarity)
  - T: a transition relation: $T \subseteq S \times S$
    - $T(s_0, s_1)$ holds when there is a transition from s_0 to s_1

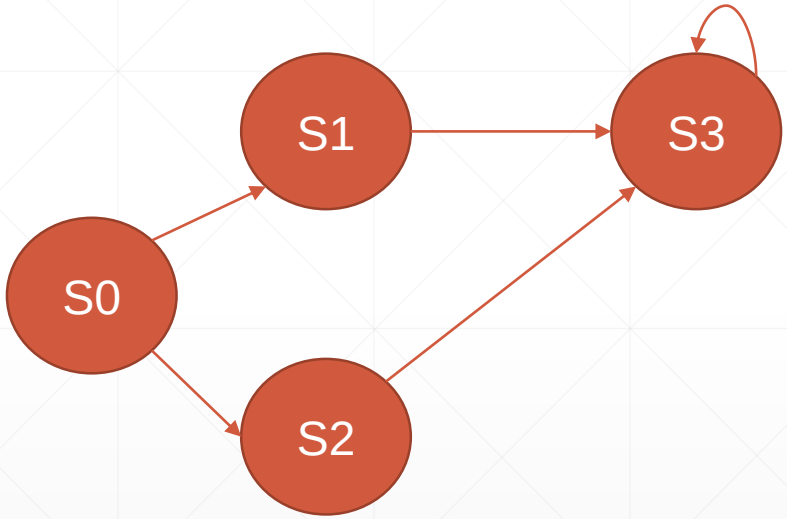
Symbolic Transition Systems in Practice

- States are made up of state variables $v \in V$
  - A state is an assignment to all variables
- A Transition System is $\langle V, I, T \rangle$
  - V: a set of state variables, V' denotes next state variables
  - I: a set of initial states
  - T: a transition relation
    - $T(v_0, \ldots, v_n, v_0', \ldots, v_n')$ holds when there is a transition
    - Note: will often still use s to denote symbolic states (just know they're made up of variables)
- Symbolic state machine is built by translating another representation
  - E.g. a program, a mathematical model, a hardware description, etc…

Symbolic Transition System Example

- 2 variables: $V = \{v_0, v_1\}$
  - $S_0 \coloneqq \neg v_0 \wedge \neg v_1, \quad S_1 \coloneqq \neg v_0 \wedge v_1$
  - $S_2 \coloneqq v_0 \wedge \neg v_1, \quad S_3 \coloneqq v_0 \wedge v_1$
- Transition relation

$$(\neg v_0 \wedge \neg v_1) \Rightarrow \left((\neg v_0' \wedge v_1') \vee (v_0' \wedge \neg v_1')\right) \wedge$$
$$(\neg v_0 \wedge v_1) \Rightarrow (v_0' \wedge v_1') \wedge$$
$$(v_0 \wedge \neg v_1) \Rightarrow (v_0' \wedge v_1') \wedge$$
$$(v_0 \wedge v_1) \Rightarrow (v_0' \wedge v_1')$$



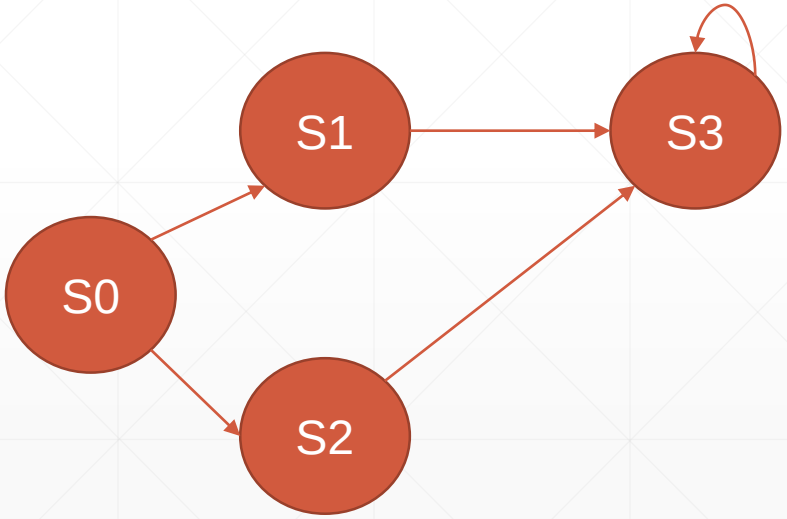
Modeling: Transition System Executions

- An *execution* is a sequence of states that respects I in the first state and T between every adjacent pair

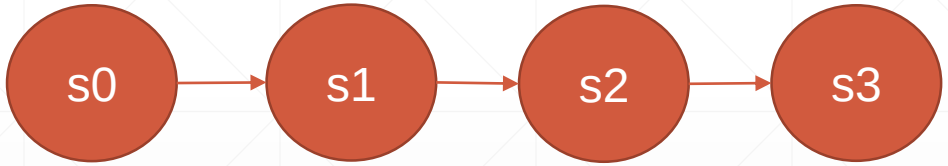
- $\pi := s_0\ s_1\ \ldots s_n$ is a finite sequence if $I(s_0) \wedge \bigwedge_{i=1}^{n} T(s_{i-1}, s_i)$

Meta Note: State Machine vs Execution Diagrams

State Machine uses capitals



Symbolic execution uses lowercase



Concrete Execution:

s0=S0, s1=S2, s2=S3, s3=S3

Specification: Historical

- Original approaches considered equivalence only
  - Model M_1 implements model M_2 **exactly**

- Duality between model and specification
  - The specification is itself a model
  - But the big innovation is that it can be a partially specified model
    - And can have loose definitions of timing, e.g. something **eventually** happens
  - Specification is typically higher-level, abstract behavior
  - Language considerations
    - Specification language should be *sufficiently different* from the implementation language
    - i.e. can always prove that $M_1 \equiv M_1$, but that's useless

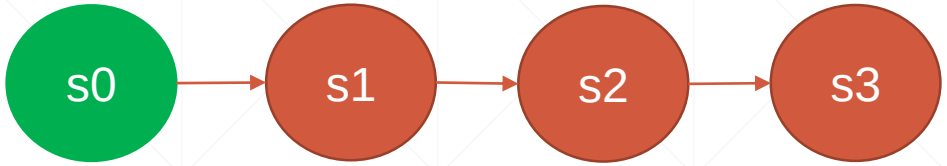
Specification: Linear Temporal Logic

- Notation: $M \models f$
  - Transition System model, M, entails LTL property, f, for ALL possible paths
  - i.e. LTL is implicitly universally quantified

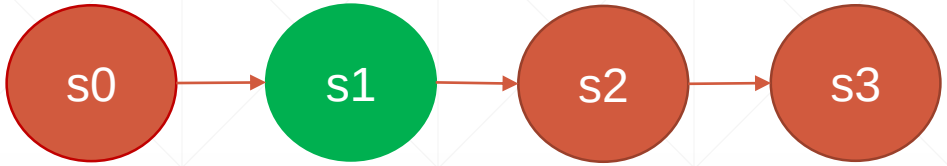
- Other logics include
  - CTL: computational tree logic (branching time)
  - CTL*: combination of LTL and CTL
  - MTL: metric temporal logic (for regions of time)

Specification: Linear Temporal Logic

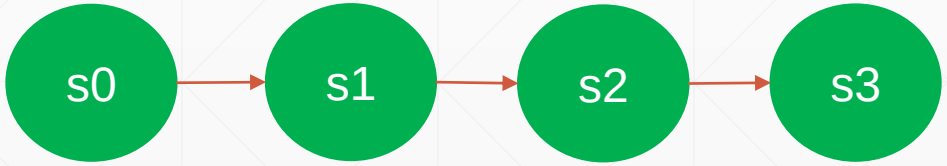
- State formula $P \subseteq S$:
  - Holds iff $s_0 \in P$



- X operator: $X(P)$
  - Next time
  - Holds iff the next state meets property P



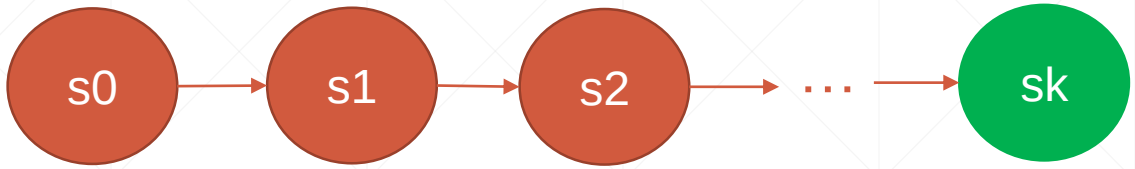
- G operator: G(P)
  - Globally holds
  - True iff every reachable state meets property P



…

Specification: Linear Temporal Logic

- F operator: F(P)
  - Finally
  - True iff P eventually holds



- U operator: P1 U P2
  - Until
  - True iff P1 holds up until (but not necessarily including) a state where P2 holds
  - P2 must hold at some point

Specification: Linear Temporal Logic

- LTL operators can be composed
  - $G\big(Req \Rightarrow F(Ack)\big)$
    - Every request eventually acknowledged
  - $G(F(DeviceEnabled))$
    - The device is enabled infinitely often (from every state, it's eventually enabled again)
  - $F(G(\neg Initializing))$
    - Eventually it's not initializing
    - E.g. there is some initialization procedure that eventually ends and never restarts

Specification Safety vs. Liveness

- Safety: “something bad does not happen”
  - State invariant, e.g. $G(\neg bad)$
- Liveness: “something good eventually happens”
  - Eventuality, e.g. $\mathrm{G}F(good)$
- Fairness conditions
  - Fair traces satisfy each of the fairness conditions infinitely often
  - E.g. only fair if it doesn’t delay acknowledging a request forever
- Every property can be written as a conjunction of a safety and liveness property

Bowen Alpern and Fred B. Schneider. Defining liveness. Information Processing Letters, 21(4):181–185, October 1985.

Specification: Liveness to Safety

- Can reduce liveness to safety checking by modifying the system
- For SAT-based:

  Armin Biere, Cyrille Artho, Viktor Schuppan. Liveness Checking as Safety Checking, Electronic Notes in Theoretical Computer Science. 2002
- Several approaches for first-order logic

- From now on, we consider only safety properties

Historical Verification Approaches: Explicit State

- Tableaux-style state exploration
- Form of depth-first search
- Many clever tricks for reducing search space
- Big contribution is handling temporal logics (including branching time)

Historical Verification Approaches: BDDs

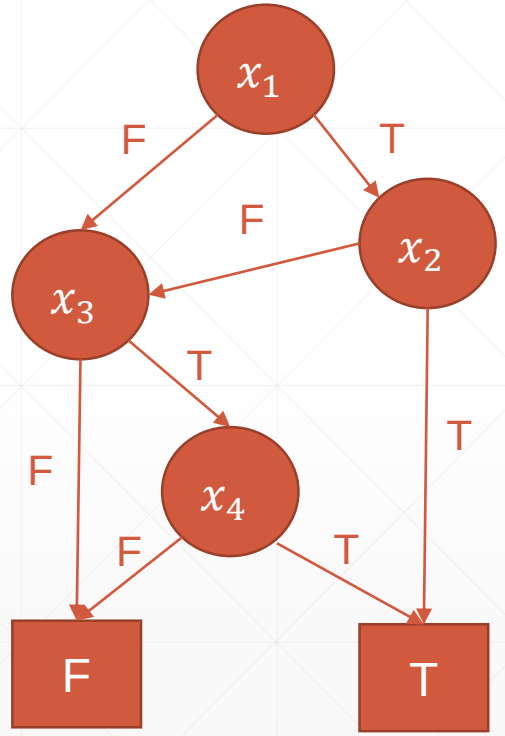
- Binary Decision Diagrams (BDDs)
  - Manipulate sets of states symbolically

J.R. Burch, E.M. Clarke, K.L. McMillan, D.L. Dill, L.J. Hwang. Symbolic Model Checking: 10^{20} States and beyond

- Great BDD resource
  - http://www.ecs.umass.edu/ece/labs/vlsicad/ece667/reading/somenzi99bdd.pdf

Historical Verification Approaches: BDDs

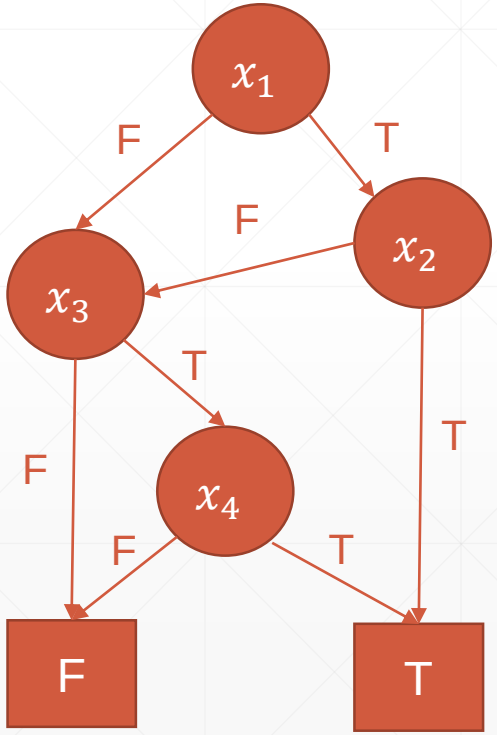
- Represent Boolean formula as a decision diagram
- Example: $(x_1 \wedge x_2) \vee (x_3 \wedge x_4)$
- Can be much more succinct than other representations



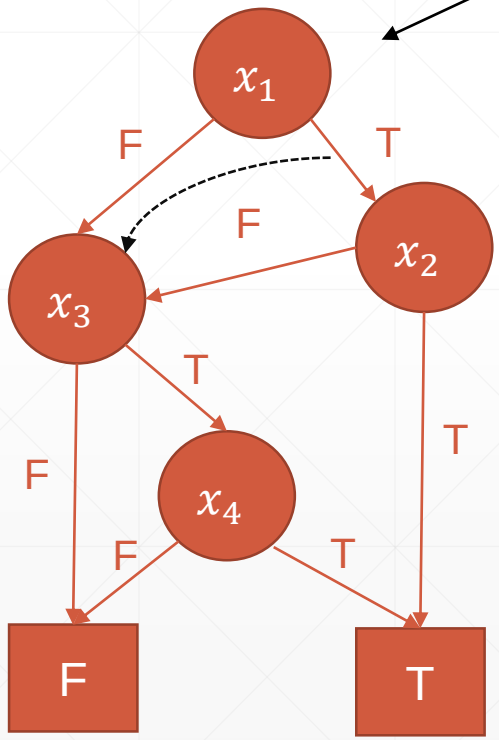
Credit for Example: Introduction to Formal Hardware Verification – Thomas Kropf

BDDs: Cofactoring

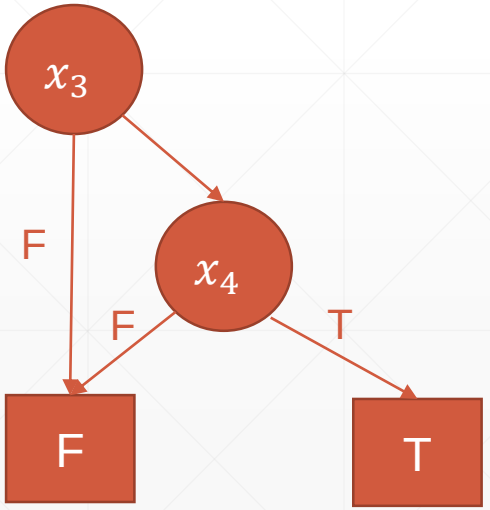
- $f|_{\neg x_2}$ for BDD f is fixing x_2 to be negative



Redirect incoming edges to assignment (F)



After reduction



Credit for Example: Introduction to Formal Hardware Verification – Thomas Kropf

BDD Operators

- Negation
  - Swap leaves (F → T)
- AND
  - All Boolean operators implemented recursively
- These two operators are sufficient

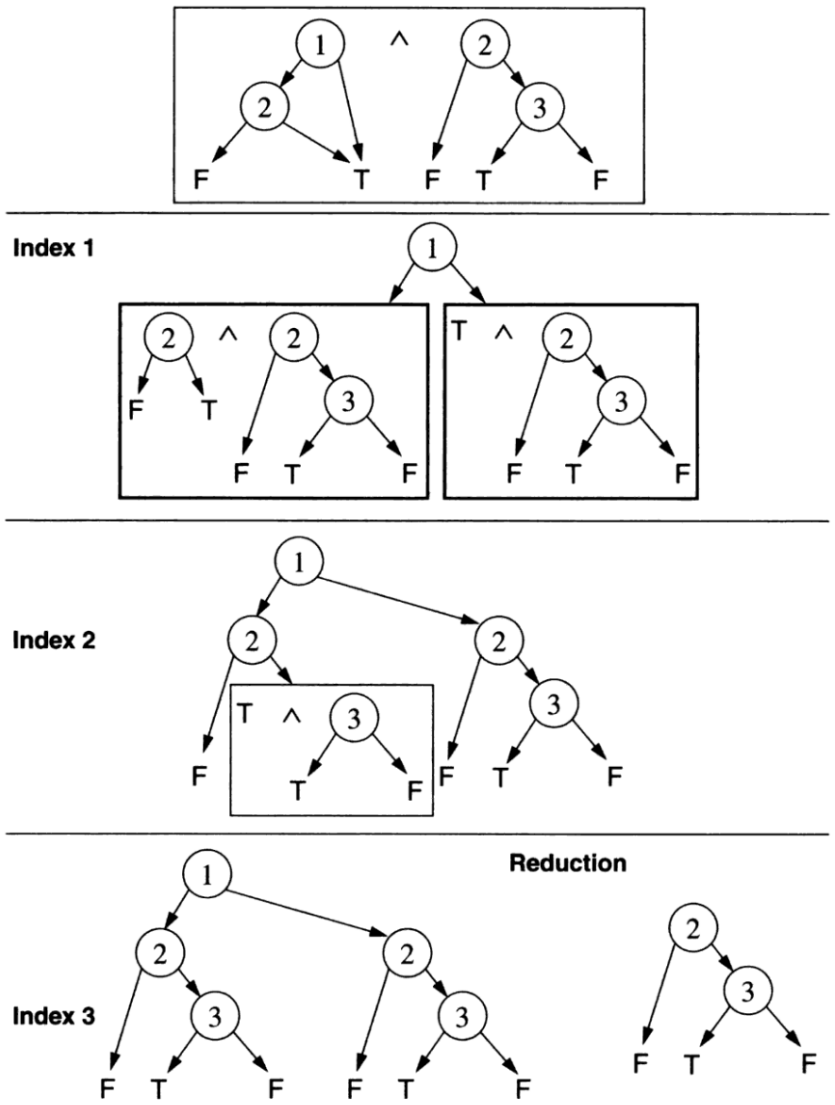


Fig. 2-7. AND-Operation between $x_1 \vee x_2$ and $x_2 \neg x_3$

Image Credit: Introduction to Formal Hardware Verification – Thomas Kropf

BDD Image Computation

- Current reachable states are BDD R
  - Over variable set V
- Compute next states with:
  - $N \coloneqq \exists V\, T(V, V') \wedge R(V)$
  - Existential is cofactoring: $\exists x\, f(x) \coloneqq (x \wedge f|_x) \vee (\neg x \wedge f|_{\neg x})$
    - aka Shannon Expansion
- Grow reachable states
  - $R = R \vee N[V'/V]$
  - Map next-state variables to current state, then add to reachable states

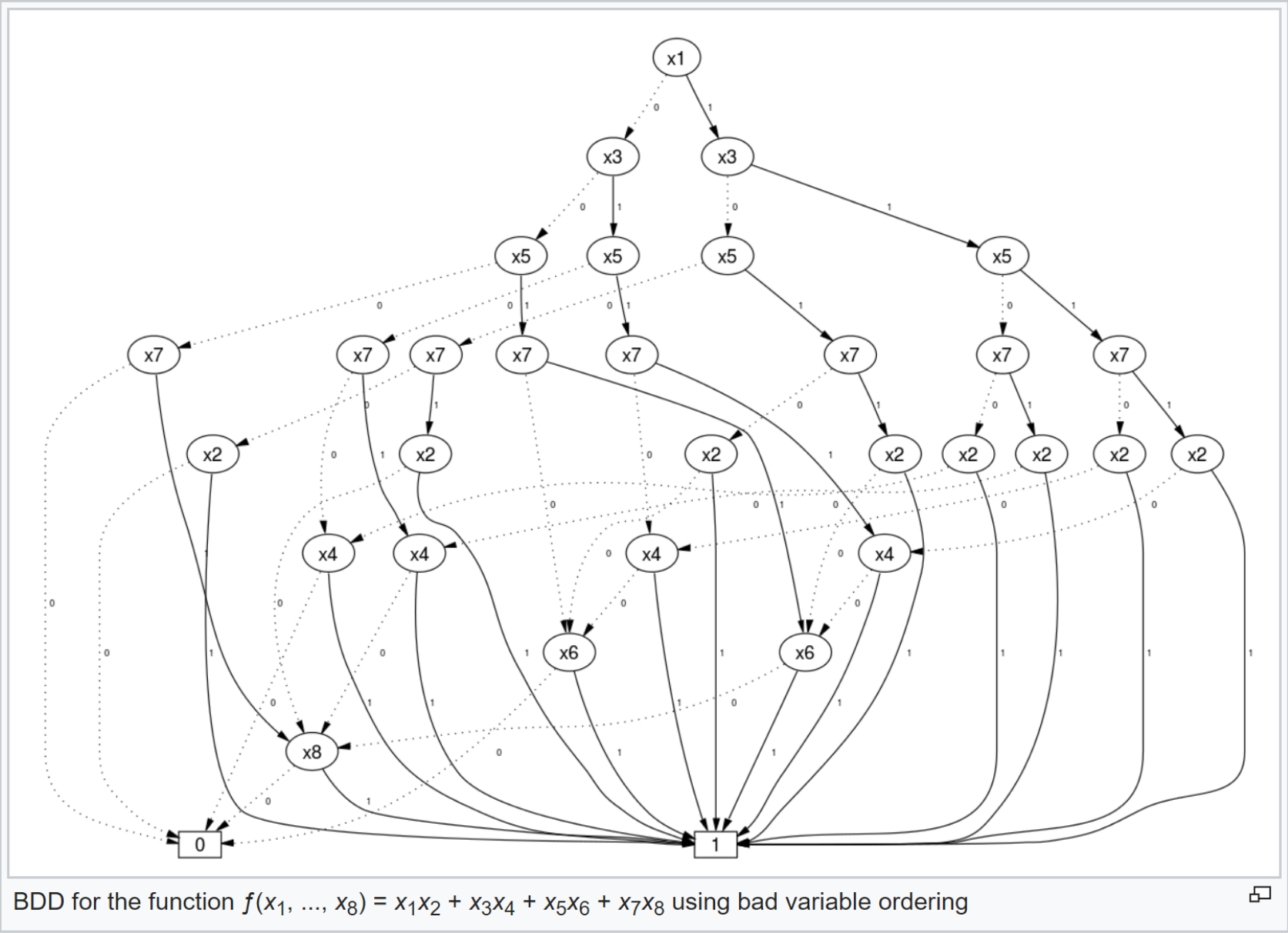
BDD-based model checking

- Start with $R = Init$
- Keep computing image and growing reachable states
- Stop when there's a fixpoint (reachable states not growing)
- Can handle ~10^{20} states
  - More with abstraction techniques and compositional model checking

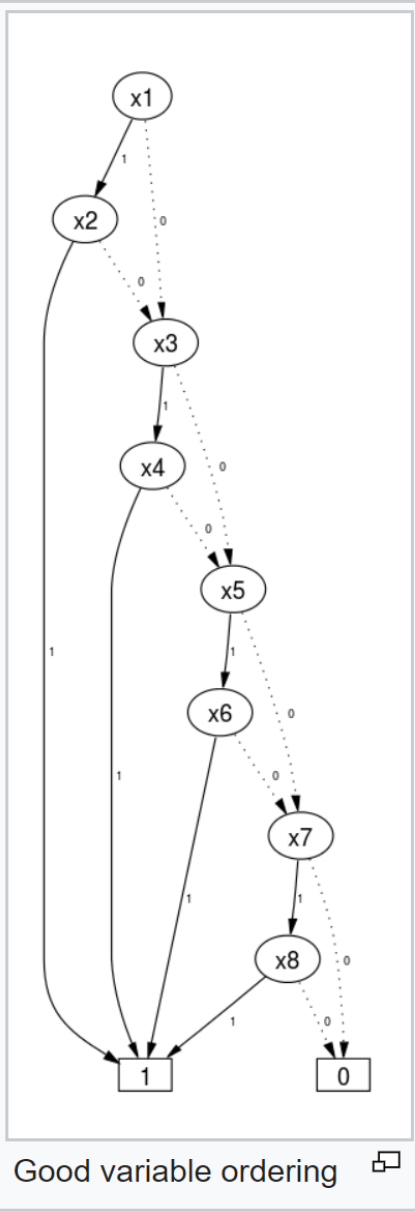
BDD: Variable Ordering

- Good variable orderings can be exponentially more compact
  - Finding a good ordering is NP-complete

- There are formulas that have no non-exponential ordering: multipliers



BDD for the function $f(x_1, ..., x_8) = x_1x_2 + x_3x_4 + x_5x_6 + x_7x_8$ using bad variable ordering



Good variable ordering

Image Credit: https://en.wikipedia.org/wiki/Binary_decision_diagram

SAT-based model checking

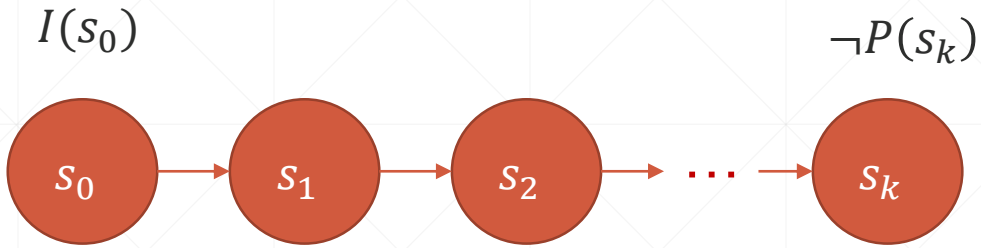
- Edmund Clarke
  - One of the founders of model checking
- SAT solving taking off
- Clarke hired several post-doctoral students to try to use SAT as an oracle to solve model checking problems
- Struggled for a while to find a general technique
  - What if you give up completeness? → Bounded Model Checking

Armin Biere, Alessandro Cimatti, Edmund Clarke, Yunshan Zhu. Symbolic Model Checking without BDDs. TACAS 1999

Bounded Model Checking (BMC)

- Sacrifice completeness for quick bug-finding
- Unroll the transition system
  - Each variable $v \in V$ gets a new symbol for each time-step, e.g. v_k is v at time k
  - Space-Time duality: unrolls temporal behavior into space
- For increasing values of k, check:
  - $I(s_0) \wedge \bigwedge_{i=1}^{k} T(s_{i-1}, s_i) \wedge \neg P(s_k)$
- If it is ever SAT, return FALSE
  - Can construct a counter-example trace from the solver model

BMC Graphically



s_0 must be an initial state

Check if it can violate the property at time k

Bounded Model Checking: Completeness

- Completeness condition: reaching the diameter
  - Diameter: d
    - The depth needed to unroll to such that every possible state is reachable in d steps or less
  - Recurrence diameter: d_r
    - The depth such that *every* execution of the system of length $\geq d_r$ *must* revisit states
    - Can be exponentially larger than the diameter
  - $d_r \geq d$

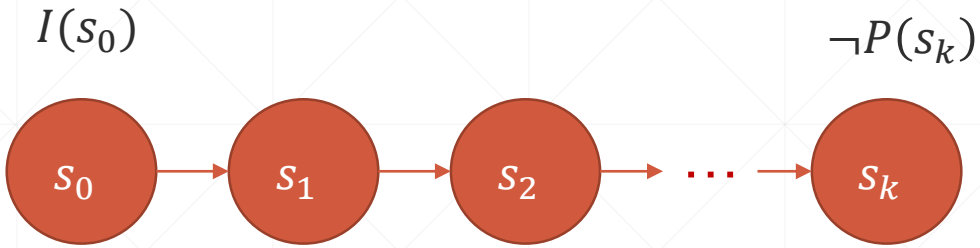
- Very difficult to compute the diameter
  - Requires a quantifier: find d such that any state reachable at $d + 1$ is also reachable in $\leq d$ steps

K-Induction

- Extends bounded model checking to be able to prove properties
- Based on the concept of (strong) mathematical induction
- For increasing values of k, check:
  - Base Case: $I(s_0) \wedge \bigwedge_{i=1}^{k} T(s_{i-1}, s_i) \wedge \neg P(s_k)$
  - Inductive Case: $\left(\bigwedge_{i=1}^{k} T(s_{i-1}, s_i) \wedge P(s_{i-1})\right) \wedge \neg P(s_k)$
  - If base case is SAT, return a counter-example
  - If inductive case is UNSAT, return TRUE
  - Otherwise, continue

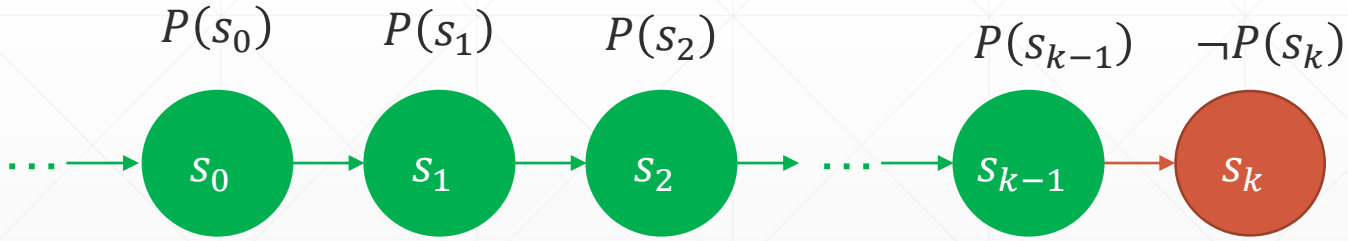
Mary Sheeran, Satnam Singh, and Gunnar Stälmarck. Checking safety properties using induction and a SAT-solver. FMCAD 2000

K-Induction Graphically



Base Case

s_0 must be an initial state

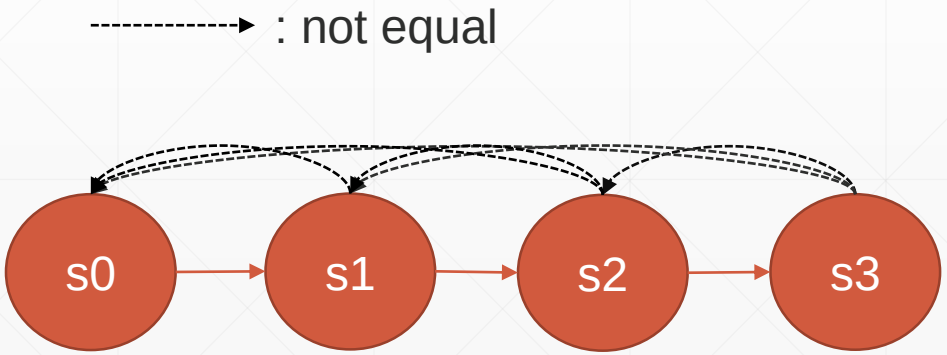


Inductive Case

Arbitrary starting state s_0
such that $P(s_0)$ holds

K-Induction: Simple Path

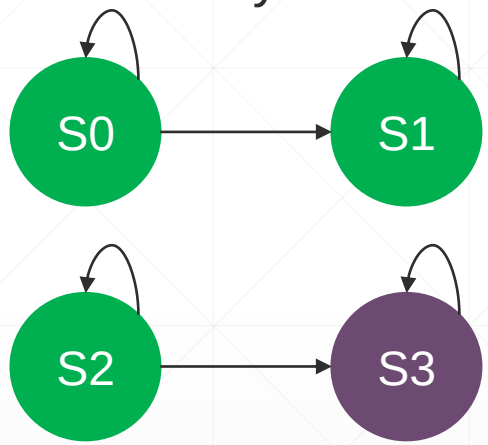
- This approach can be complete over a finite domain
  - requires the simple path constraint
  - each state is distinct from other states in trace
- If simple path is UNSAT, then we can return true

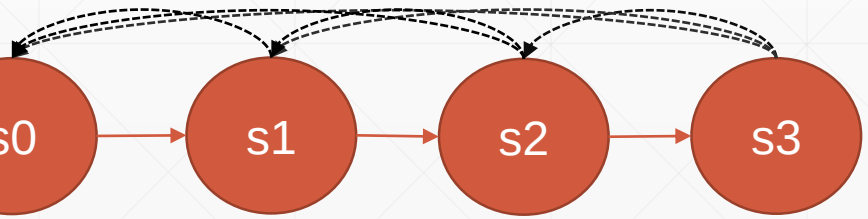


K-Induction: Simple Path

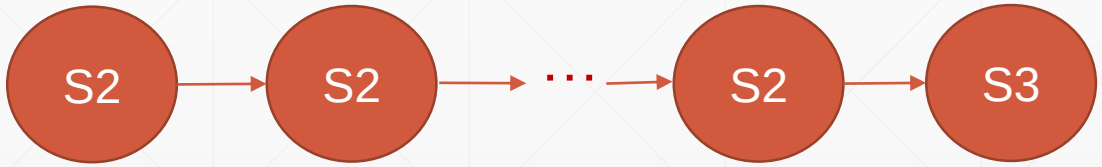
- This approach can be complete over a finite domain
  - requires the simple path constraint
  - each state is distinct from other states in trace
- If simple path is UNSAT, then we can return true

Why?





Without simple path, inductive step could get:

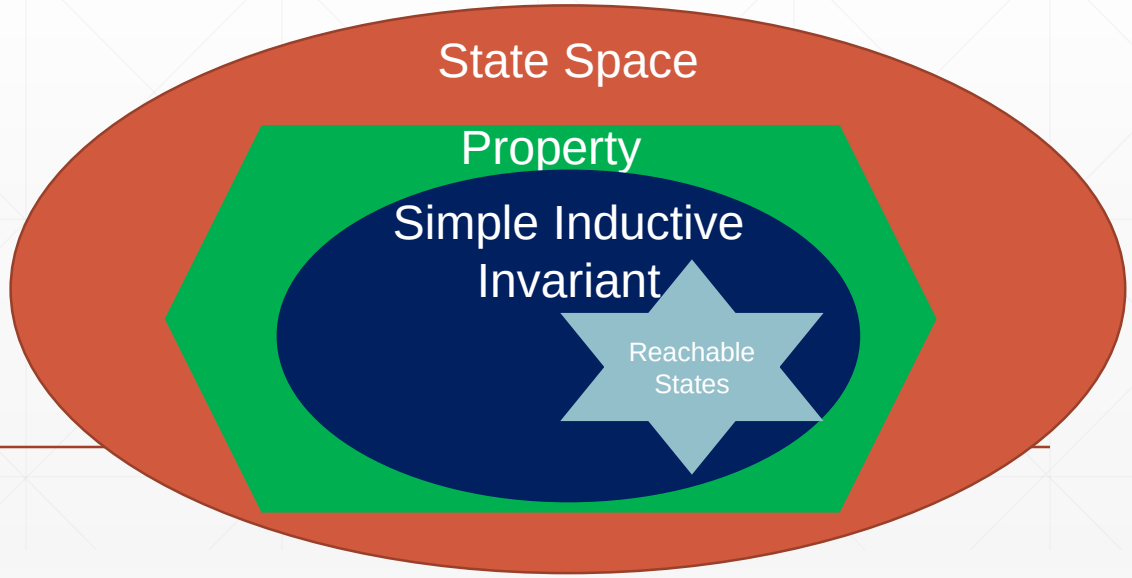


K-Induction Observation

- Crucial observation
  - Does not depend on direct computation of reachable state space
- Beginning of “property directed” techniques
  - We do not need to know the exact reachable states, as long as we can guarantee they meet the property
  - “Property directed” is associated with a family of techniques that build inductive invariants automatically

Inductive Invariants

- The goal of most modern model checking algorithms
- Over finite-domain, just need to show that algorithm makes progress, and it will eventually find an inductive invariant
  - E.g. in the worst case, the reachable states are themselves an inductive invariant
  - Hopefully there's an easier to find inductive invariant that is sufficient
- Inductive Invariant: II
  - $Init(s) \Rightarrow II(s)$
  - $\mathrm{T}(s, s') \wedge II(s) \Rightarrow II(s')$
  - $II(s) \Rightarrow P(s)$



Advanced: Relative Induction

- Inductive Invariant:
  - $a \geq 0 \wedge b \geq 0 \wedge c \geq 0$
- Incremental induction
  - Guess: $a \geq 0$
  - Induction: $c \geq 0$, *relative* to $a \geq 0$
  - Induction: $b \geq 0$, *relative* to $a \geq 0 \wedge c \geq 0$
  - Prove: $a \geq 0$
- Break circularity with induction

```
a = 0 ; b = 0 ; c = 0
while * do:
    assert a ≥ 0
    a' = a + b
    b' = b + c
    c' = c + 1 + a
```

Credit for Example: Oded Padon

Advanced: Relative Induction

```
a = 0 ; b = 0 ; c = 0
while * do:
    assert a ≥ 0
    a' = a + b
    b' = b + c
    c' = c + 1 + a
```

- Break circularity with induction
  - Guess $a \geq 0$
  - $Init \vDash a \geq 0 \wedge c \geq 0$,
  - Relative Induction: $a \geq 0 \wedge c \geq 0 \vDash c' \geq 0$
  - $Init \vDash a \geq 0 \wedge c \geq 0 \wedge b \geq 0$
  - Induction: $a \geq 0 \wedge c \geq 0 \wedge b \geq 0 \vDash a' \geq 0 \wedge c' \geq 0 \wedge b' \geq 0$
- The last inductive proof is a complete proof
  - But obtaining the inductive invariant by first guessing $a \geq 0$, then finding $c \geq 0$ could be easier

Credit for Example: Oded Padon

Advanced Algorithms

- Interpolant-based model checking
  - Constructs an overapproximation of the reachable states
  - Terminates when it finds an inductive invariant or a counterexample
- IC3 / PDR
  - Computes over (under) approximations of forward (backward) reachable states
  - Refines approximations by guessing relative inductive invariants
  - Terminates when it finds an inductive invariant or a counterexample

Thank you!