Bounded Model Checking

COMP6210 - Automated Software Verification

ECS, University of Southampton

ROBDDs and Symbolic Model Checking (Review)

ROBDD is a *canonical form* to represent Boolean functions:

- often more compact than traditional normal forms,
- can be manipulated efficiently.

Reachable state space is represented as an **ROBDD**.

Property verification uses iterative computations on the reachable state space.

Some Limitations in Working with ROBDDs

ROBDDs are fantastic, but they come with a few problems:

- they can often become large,
- variable ordering must be uniform along paths,
- selecting the "right" variable ordering is crucial,
- in some cases *no space-efficient variable ordering exists*.

Are there any alternatives in model checking?

Some Limitations in Working with ROBDDs

ROBDDs are fantastic, but they come with a few problems:

- they can often become large,
- variable ordering must be uniform along paths,
- selecting the "right" variable ordering is crucial,
- in some cases *no space-efficient variable ordering exists*.

Are there any alternatives in model checking?

Yes! SAT-based methods.

Boolean Satisfiability (SAT) Problem

The Boolean satisfiability problem (also **SAT problem**) asks whether a given Boolean formula is satisfiable.

Boolean Satisfiability (SAT) Problem

The Boolean satisfiability problem (also **SAT problem**) asks whether a given Boolean formula is satisfiable.

E.g. is the following Boolean (propositional) formula satisfiable?

$$(x_1 \wedge x_2 \vee x_3) \wedge (x_4 \wedge x_1 \vee x_3) \wedge \neg(x_4 \vee x_1)$$

That is, does there exist an assignment of truth values to the Boolean variables x_1, x_2, x_3, x_4 that makes the formula True?

Boolean Satisfiability (SAT) Problem

The Boolean satisfiability problem (also **SAT problem**) asks whether a given Boolean formula is satisfiable.

E.g. is the following Boolean (propositional) formula satisfiable?

$$(x_1 \wedge x_2 \vee x_3) \wedge (x_4 \wedge x_1 \vee x_3) \wedge \neg(x_4 \vee x_1)$$

That is, does there exist an assignment of truth values to the Boolean variables x_1, x_2, x_3, x_4 that makes the formula True?

Yes! $x_1 =$ False, $x_2 =$ True, $x_3 =$ True, $x_4 =$ False.

Boolean Satisfiability (SAT) Problem

The Boolean satisfiability problem (also **SAT problem**) asks whether a given Boolean formula is satisfiable.

E.g. is the following Boolean (propositional) formula satisfiable?

$$(x_1 \wedge x_2 \vee x_3) \wedge (x_4 \wedge x_1 \vee x_3) \wedge \neg(x_4 \vee x_1)$$

That is, does there exist an assignment of truth values to the Boolean variables x_1, x_2, x_3, x_4 that makes the formula True?

Yes! $x_1 =$ False, $x_2 =$ True, $x_3 =$ True, $x_4 =$ False.

The SAT problem was the first example of an **NP-Complete** problem (*Cook-Levin theorem*, 1971).

Boolean Satisfiability (SAT) Problem

We can encode **many** interesting problems in terms of Boolean satisfiability (SAT).

In particular, many problems in circuit design can often be cast as SAT problems.

Boolean Satisfiability (SAT) Problem

We can encode **many** interesting problems in terms of Boolean satisfiability (SAT).

In particular, many problems in circuit design can often be cast as SAT problems.

Problem: *can we factorise integers using SAT?*

Boolean Satisfiability (SAT) Problem

We can encode **many** interesting problems in terms of Boolean satisfiability (SAT).

In particular, many problems in circuit design can often be cast as SAT problems.

Problem: *can we factorise integers using SAT?*

Yes! Model a *multiplier circuit* as a Boolean formula, and ask if there are inputs to the circuit that result in a given number (e.g. 21, or 10101 in binary) on the output wires.

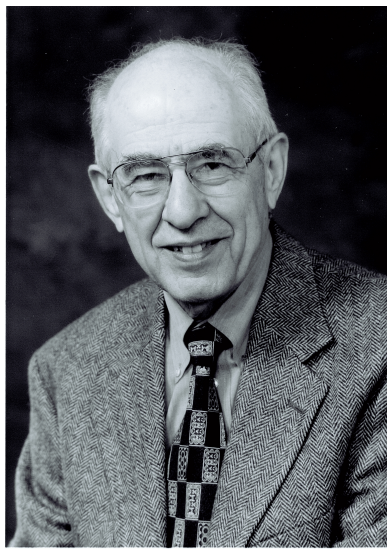
This is equivalent to asking "is 21 a prime number?".

DPLL Algorithm

Davis–Putnam–Logemann–Loveland (DPLL) Algorithm

- Introduced in 1962 as a refinement of an earlier 1960 algorithm by Davis and Putnam.
- A backtracking-based algorithm for solving the **SAT problem**.
- Worst case running time is exponential.
- Serves as the backbone of modern **SAT solvers** (e.g. MiniSAT, Z3).
- Major practical applications in *hardware verification*.

Martin Davis



Hilary Putnam

DPLL and SAT Solvers

DPLL can take an **exponential** amount of time, however it can often find solutions to SAT relatively quickly.

The **DPLL** algorithm works by

1. Converting the Boolean formula into CNF (*Conjunctive Normal Form*)
2. Applying simplification rules
3. Splitting

Conjunctive Normal Form

A *literal* is a atomic formula or its negation (e.g. a propositional variable x_1, or $\neg x_1$).

A *clause* is a disjunction of literals (e.g. $x_1 \vee \neg x_3 \vee x_2$)

A propositional formula is in Conjunctive Normal Form (CNF) if it is a conjunction of one or more clauses, e.g.

$$(x_1 \vee \neg x_3 \vee x_2) \wedge (\neg x_4 \vee \neg x_1 \vee x_3) \wedge (\neg x_5)$$

Any given propositional formula can be converted into CNF by applying transformations

- eliminating double negations (i.e. $\neg\neg\phi$ becomes ϕ)
- applying De Morgan's laws,
- applying the distributive law.

Conjunctive Normal Form

Caveat: Converting a formula into CNF can result in an exponentially large formula.

In **DPLL**, each clause is treated as a *set of literals*, e.g. $(x_1 \vee \neg x_3 \vee x_2)$ is treated as $\{x_1, \neg x_3, x_2\}$.

The overall CNF formula $C_1 \wedge C_2 \wedge \cdots \wedge C_l$, where C_i are clauses, is also thought of as a *set*

$$\{C_1, C_2, \ldots, C_l\}$$

Caveats: an empty clause $\{\}$ is unsatisfiable, whereas an empty *set of clauses* (which is equivalent to True) is considered satisfiable.

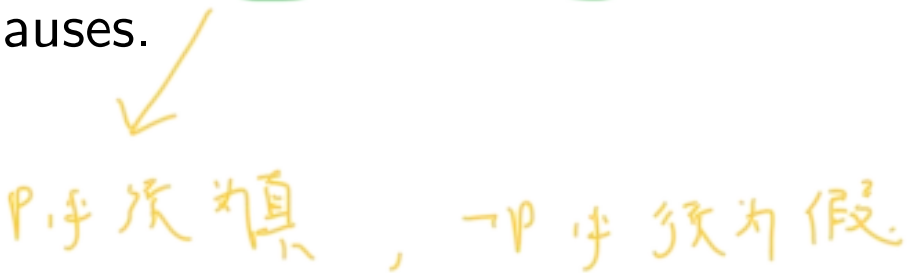
空子句是不可接受的(没有文字的子句.) → 所有对值都为假

空集合是可接受的(没有子句) → 任何对值都为真

DPLL Simplification Rules

There are two main rules

1. *Unit propagation (one literal rule)*: when there is a unit clause (a clause with just one element p), remove $\neg p$ from all the other clauses.



DPLL Simplification Rules

There are two main rules

1. *Unit propagation (one literal rule)*: when there is a unit clause (a clause with just one element p), remove $\neg p$ from all the other clauses.

   [*Intuitively: if p has to be true, then $\neg p$ does not add anything to the other disjunctive clauses.*]

DPLL Simplification Rules

There are two main rules

1. *Unit propagation (one literal rule)*: when there is a unit clause (a clause with just one element p), remove $\neg p$ from all the other clauses.

   [*Intuitively: if p has to be true, then $\neg p$ does not add anything to the other disjunctive clauses.*]

2. *Affirmative-negative rule*: if p occurs either only negated, or only unnegated, then remove all clauses containing p.

肯否原则

p只以一种形式出现，
则移除所有含p的项

DPLL Simplification Rules

There are two main rules

1. *Unit propagation (one literal rule)*: when there is a unit clause (a clause with just one element p), remove $\neg p$ from all the other clauses.

   [*Intuitively: if p has to be true, then $\neg p$ does not add anything to the other disjunctive clauses.*]

2. *Affirmative-negative rule*: if p occurs either only negated, or only unnegated, then remove all clauses containing p.

   [*Intuitively: one can make all clauses containing p True by setting p to True if it is always unnegated, or by setting p to False if it only appears negated.*]

DPLL Simplification Rules

There are two main rules

1. *Unit propagation (one literal rule)*: when there is a unit clause (a clause with just one element p), remove $\neg p$ from all the other clauses.

   [*Intuitively: if p has to be true, then $\neg p$ does not add anything to the other disjunctive clauses.*]

2. *Affirmative-negative rule*: if p occurs either only negated, or only unnegated, then remove all clauses containing p.

   [*Intuitively: one can make all clauses containing p True by setting p to True if it is always unnegated, or by setting p to False if it only appears negated.*]

Both simplification rules *preserve satisfiability*.

DPLL Splitting

The simplification rules by themselves do not generally allow us to check satisfiability.

To make further progress, we must consider *case splits*.

For a set of clauses Γ, choose a propositional variable x and consider two cases:

1. $\Gamma \cup \{x\}$ ("set x to True"),
2. $\Gamma \cup \{\neg x\}$ ("set x to False").

Then apply the simplification rules (unit propagation and affirmative-negative rule) as before.

Applying these steps recursively one obtains a backtracking strategy for solving the SAT problem.
[*If an empty set of clauses is found, the formula is satisfiable.*]

Current State of SAT Solving

Modern implementations of SAT solvers improve on the basic ideas in DPLL.

SAT solvers are capable of solving practical SAT problems with *millions* of propositional variables.

Enormous progress has been made in SAT solving technology over the past 20 years, spurred on by the SAT competition.

http://www.satcompetition.org/

Bounded Model Checking

Safety verification: *can we find a bad state in k steps?*

Verification of temporal properties (LTL, CTL):
can we find a counterexample to the property in k steps?

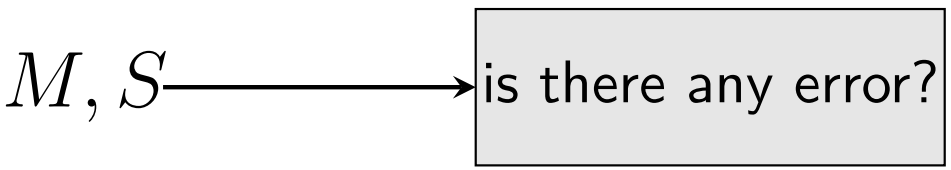
反例

Why bounded?

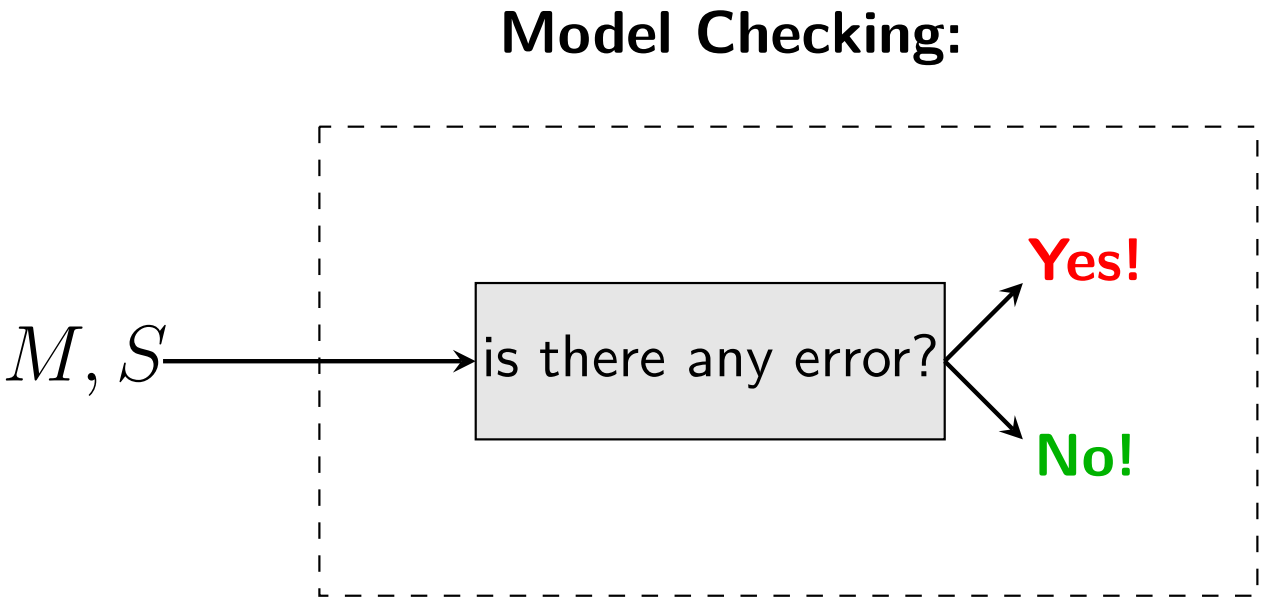
- Model Checking is supposed to be exhaustive, i.e. the *whole state space* of the system is explored. 全面的
- In real world systems, the number of states is usually **too big** (*state space explosion problem*).
- We can explore only a subset of the state space
  ... → Bounded Model Checking (BMC).

Model Checking versus Bounded Model Checking

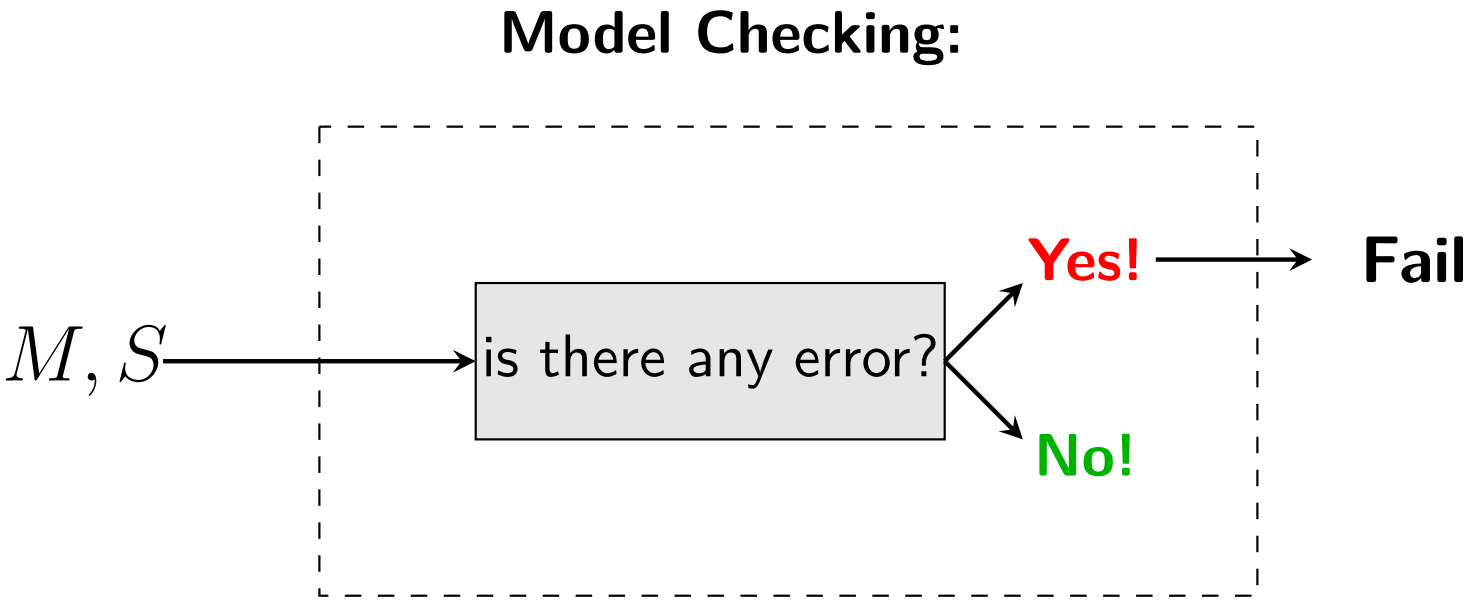
Model Checking:



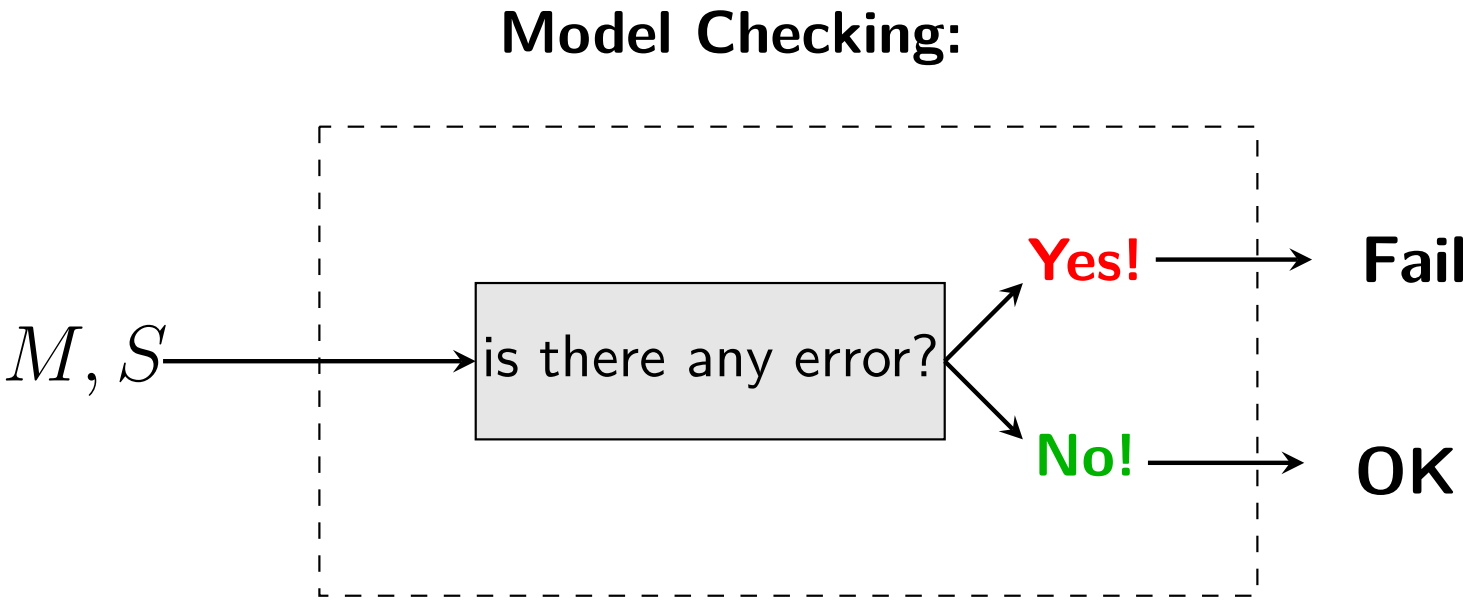
Model Checking versus Bounded Model Checking



Model Checking versus Bounded Model Checking

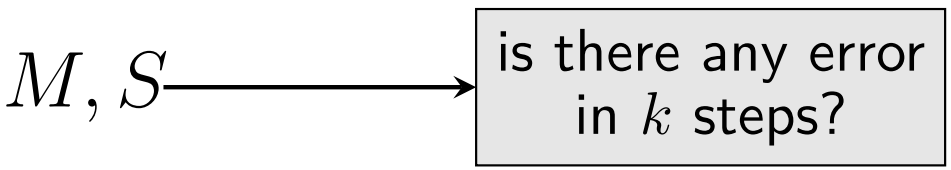


Model Checking versus Bounded Model Checking

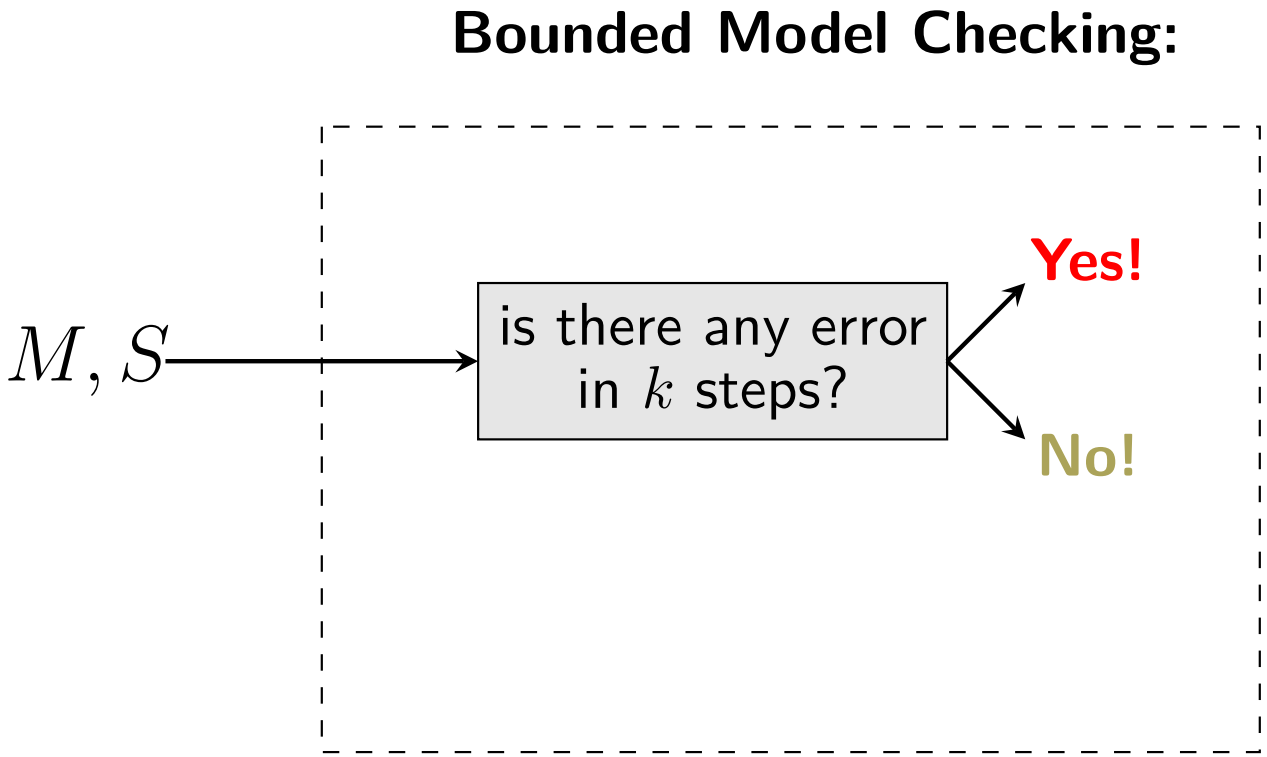


Model Checking versus Bounded Model Checking

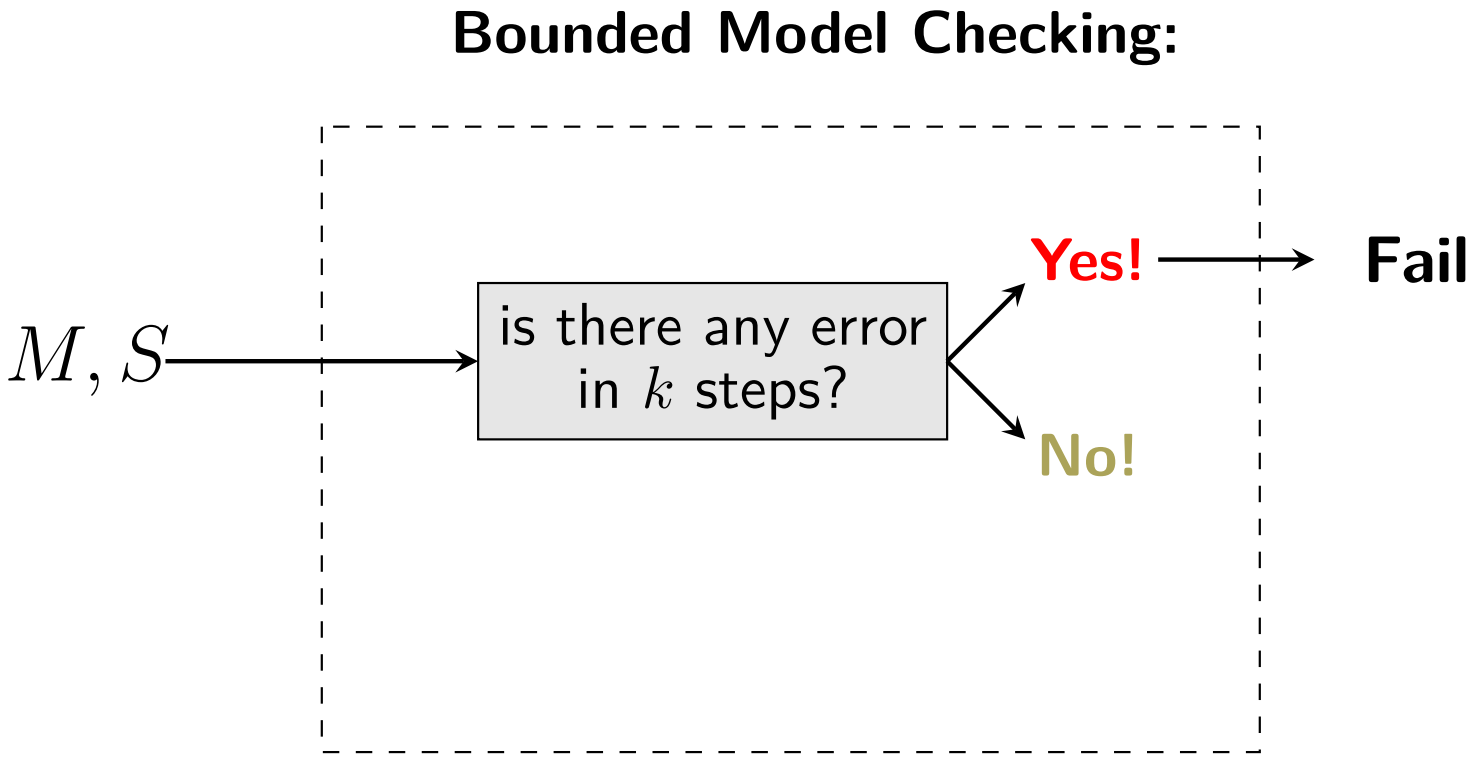
Bounded Model Checking:



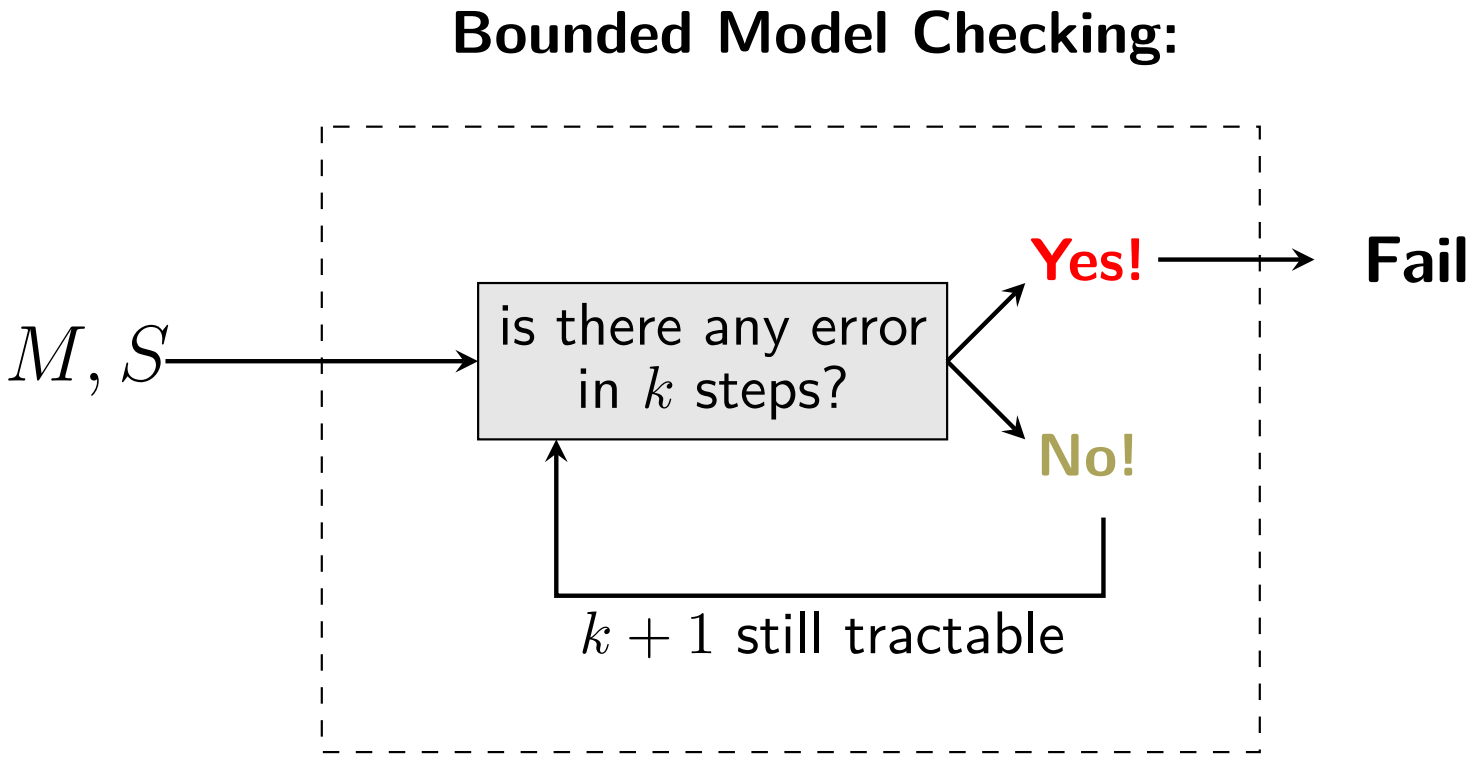
Model Checking versus Bounded Model Checking



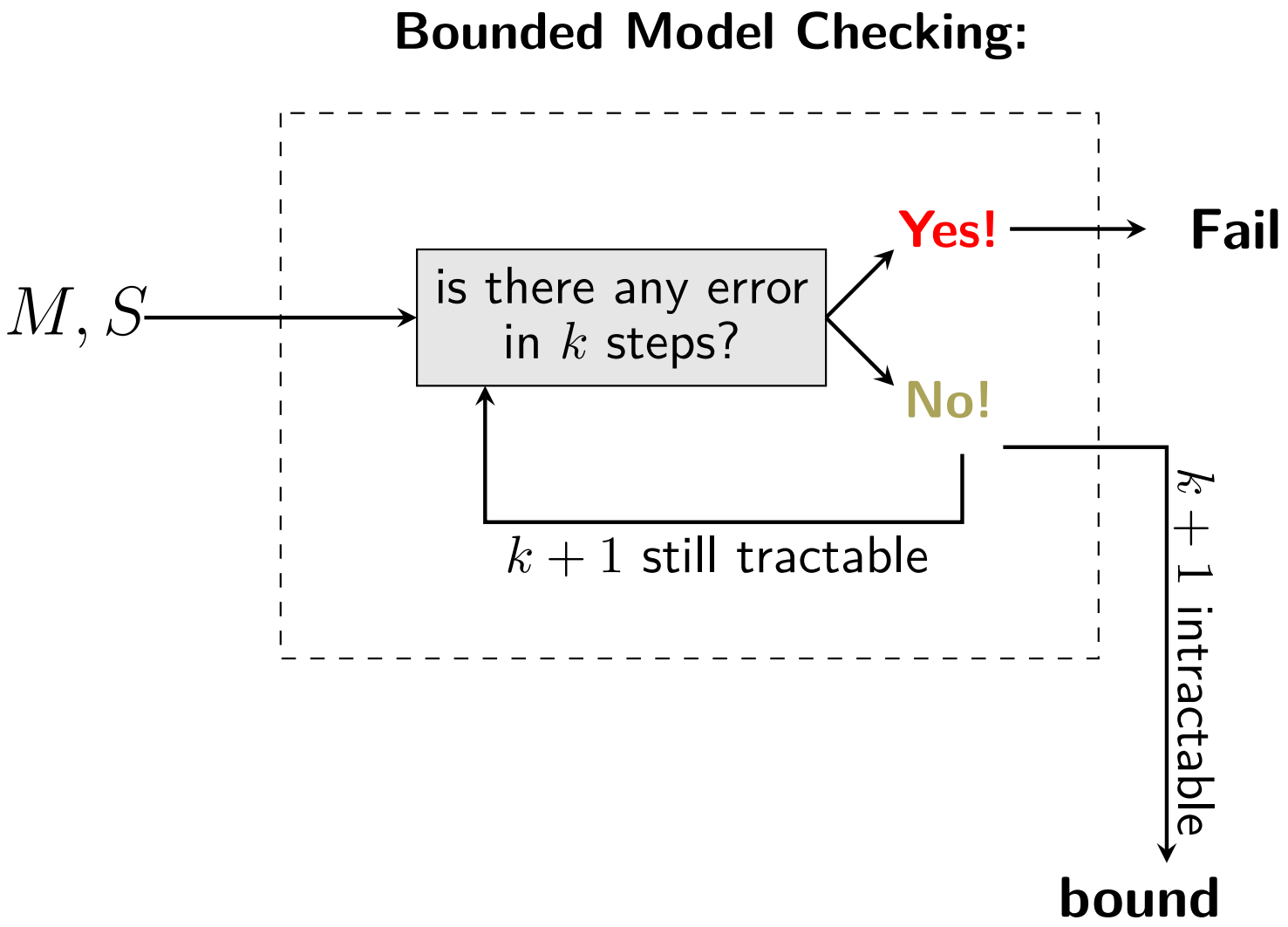
Model Checking versus Bounded Model Checking



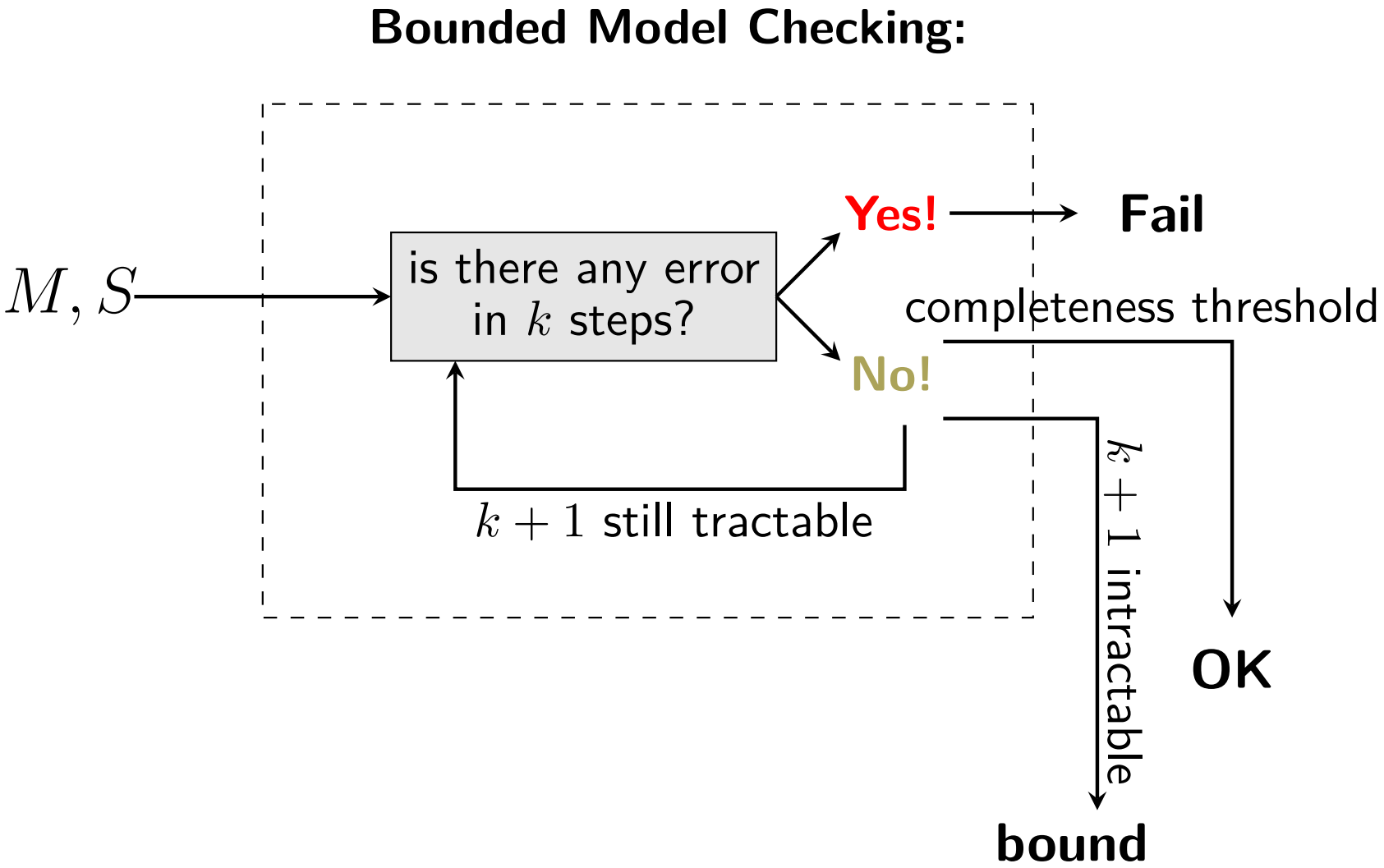
Model Checking versus Bounded Model Checking



Model Checking versus Bounded Model Checking

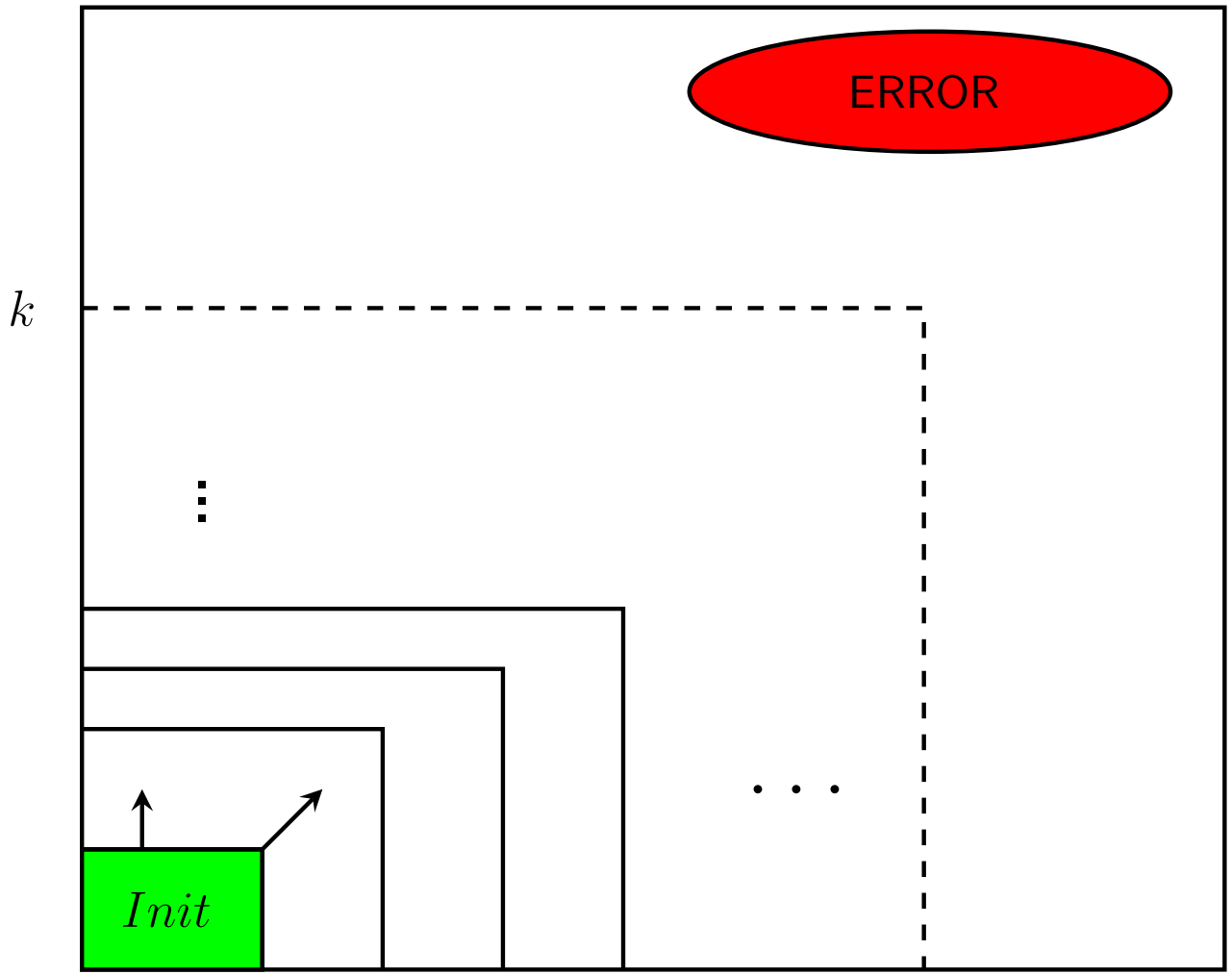


Model Checking versus Bounded Model Checking



Bounded Model Checking (Main Idea)

BMC: check if a property holds for a subset of states.



Bounded Model Checking for Reachability

是否能在k步内到达目标状态

Given a *positive integer* k, is the **Target** reachable in k steps?
(*Think of* **Target** *as defining bad states*.)

We encode a **BMC** problem as a **SAT** problem.

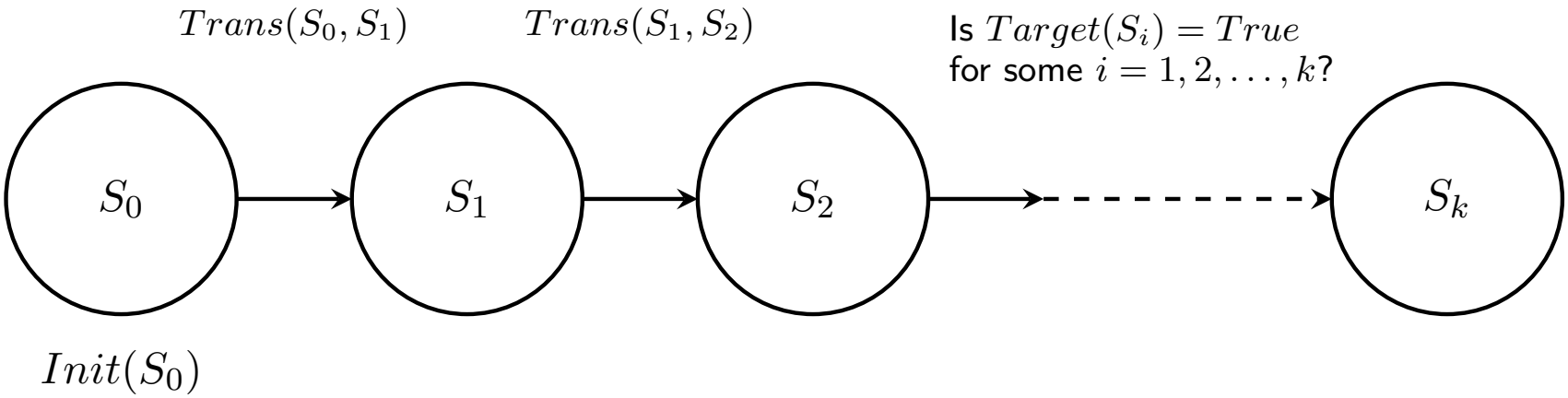
The Bounded Model Checking problem

$$X,\ Init(X),\ Target(X),\ Trans(X, X'),\ k$$

is transformed into a Boolean formula Φ_k

Φ_k is satisfiable iff **Target** is reachable from $Init$ within k steps following $Trans$.

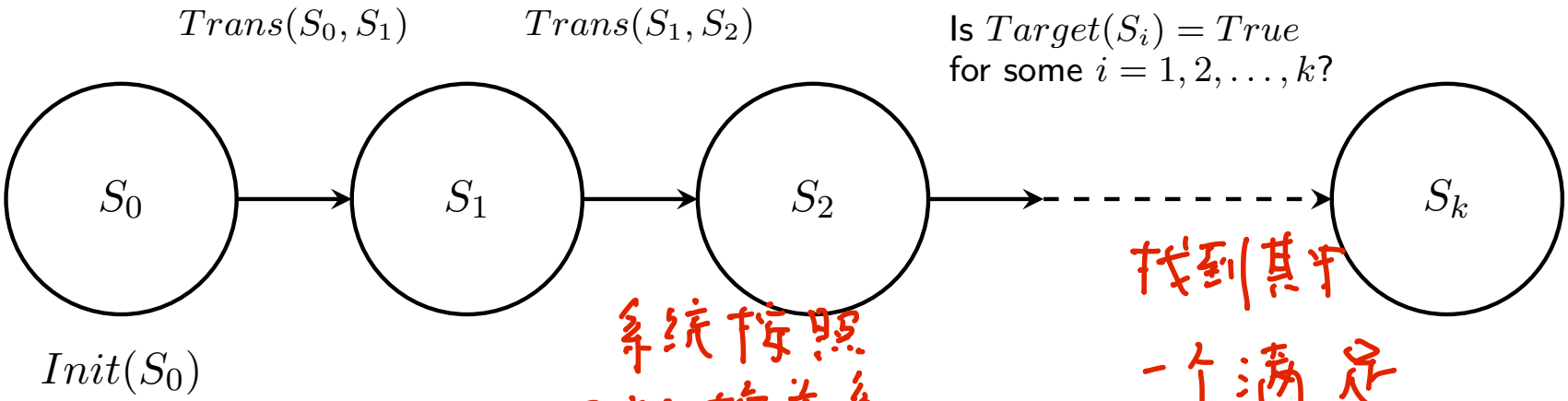
Bounded Model Checking for Reachability



状态变量的集合

- $S_0, S_1, S_2, \ldots, S_k$ are copies of $X = (x_1, x_2, \ldots, x_n)$:
  - $S_0 = ({x_1}^0, {x_2}^0, \ldots, {x_n}^0)$,
  - $S_1 = ({x_1}^1, {x_2}^1, \ldots, {x_n}^1)$,
  - $S_2 = ({x_1}^2, {x_2}^2, \ldots, {x_n}^2)$,
  - $\vdots$
  - $S_k = ({x_1}^k, {x_2}^k, \ldots, {x_n}^k)$.

Bounded Model Checking for Reachability

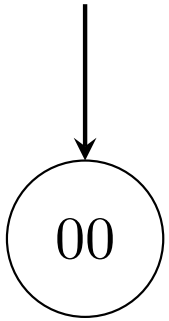


$$\Phi_k = Init(S_0) \wedge \left(\bigwedge_{i=0}^{k-1} Trans(S_i, S_{i+1}) \right) \wedge \left(\bigvee_{i=0}^{k} Target(S_i) \right)$$

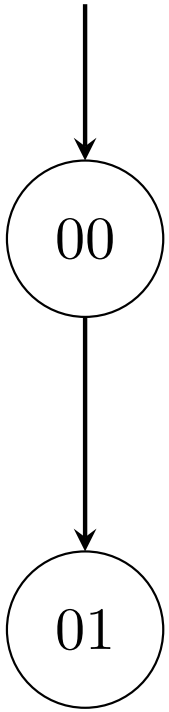
Φ_k is satisfiable if and only if Target is reachable within k steps.

(The size of Φ_k is $|Init| + k|Trans| + k|Target|$.)

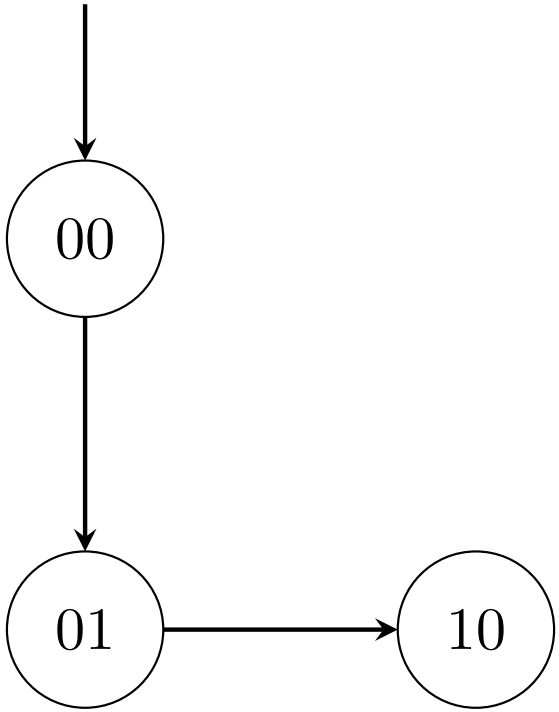
Example: Two Bit Counter



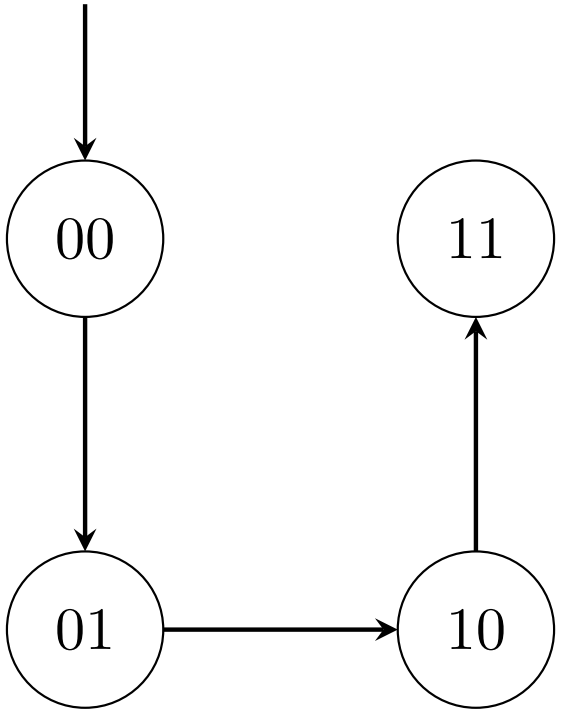
Example: Two Bit Counter



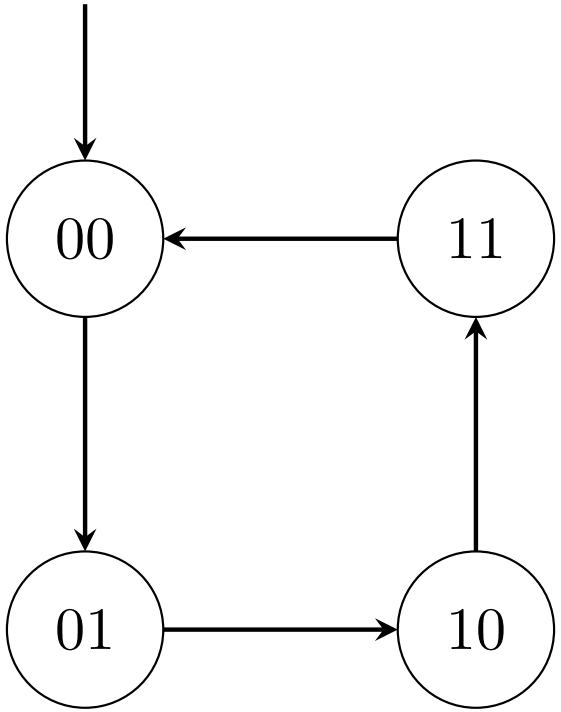
Example: Two Bit Counter



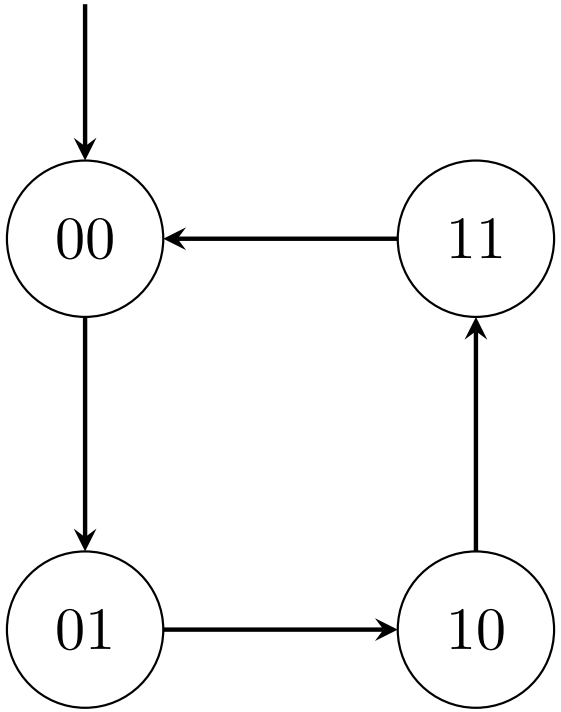
Example: Two Bit Counter



Example: Two Bit Counter

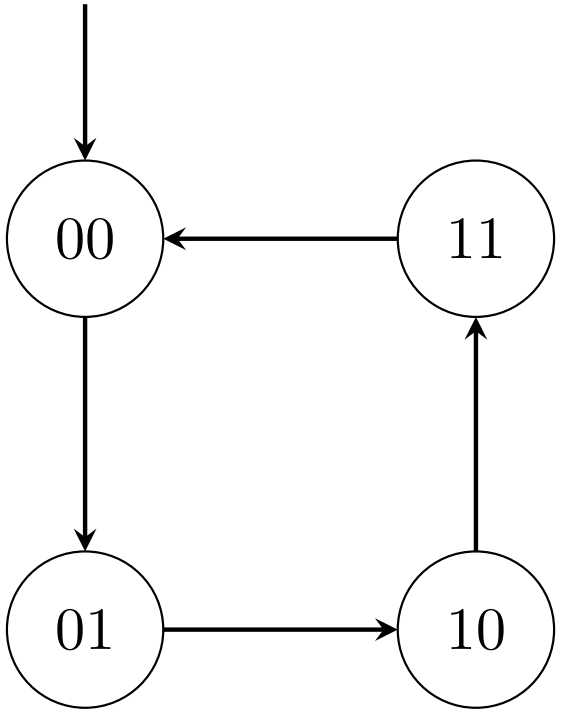


Example: Two Bit Counter



Two Boolean variables l and r (for the left and right bit).

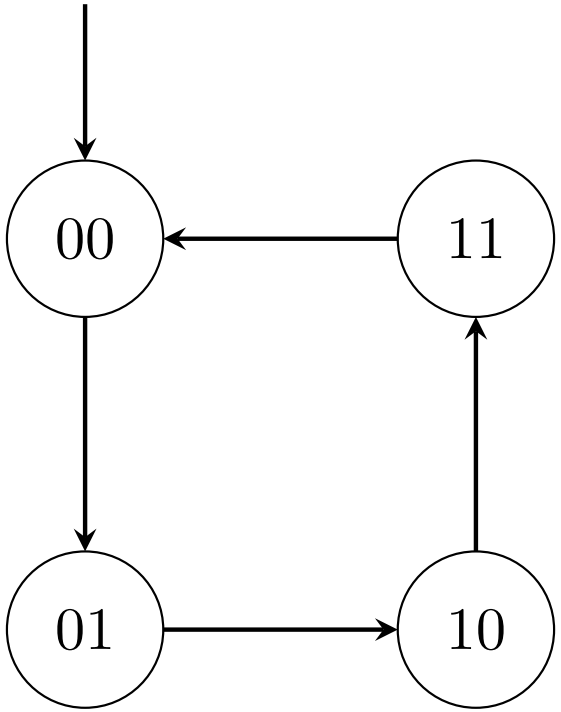
Example: Two Bit Counter



Two Boolean variables l and r (for the left and right bit).

Safety property: $G(\neg l \vee \neg r)$.

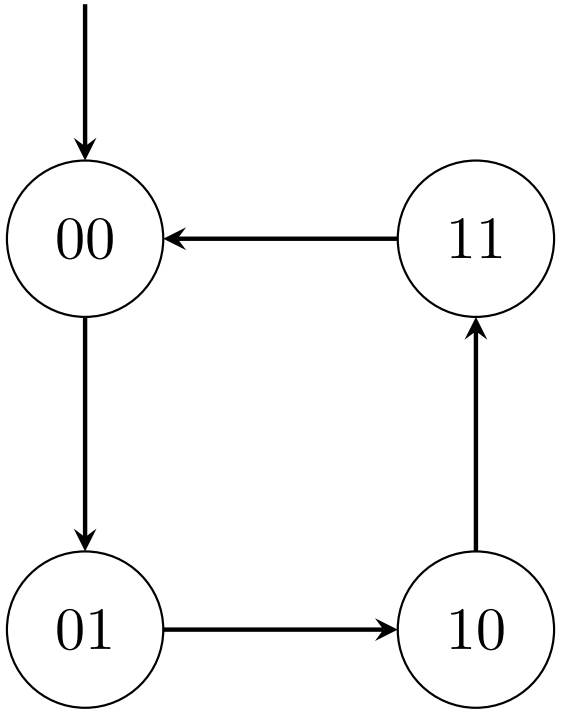
Example: Two Bit Counter



Two Boolean variables l and r (for the left and right bit).

Safety property: $G(\neg l \vee \neg r)$. Therefore $Target \equiv l \wedge r$.

Example: Two Bit Counter

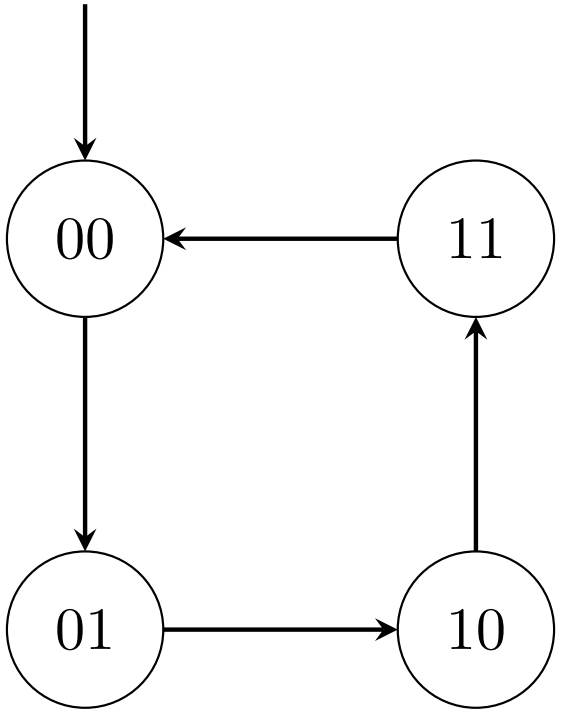


Two Boolean variables l and r (for the left and right bit).

Safety property: $G(\neg l \vee \neg r)$. Therefore $Target \equiv l \wedge r$.

$Init \equiv$

Example: Two Bit Counter

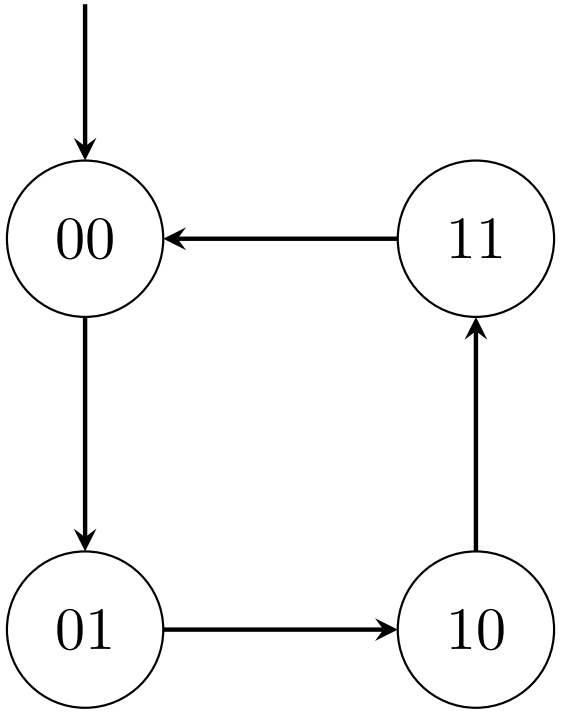


Two Boolean variables l and r (for the left and right bit).

Safety property: $G(\neg l \vee \neg r)$. Therefore $Target \equiv l \wedge r$.

$Init \equiv \neg l \wedge \neg r$.

Example: Two Bit Counter

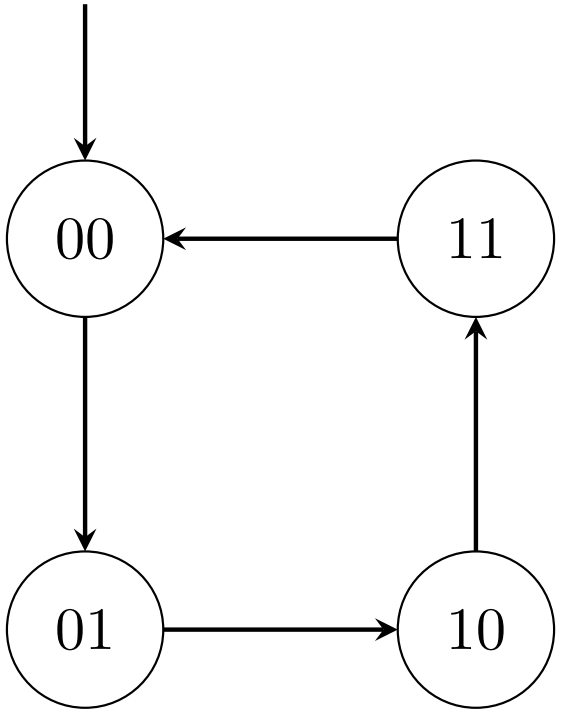


Two Boolean variables l and r (for the left and right bit).

Safety property: $G(\neg l \vee \neg r)$. Therefore $Target \equiv l \wedge r$.

$Init \equiv \neg l \wedge \neg r$. $Trans \equiv$

Example: Two Bit Counter

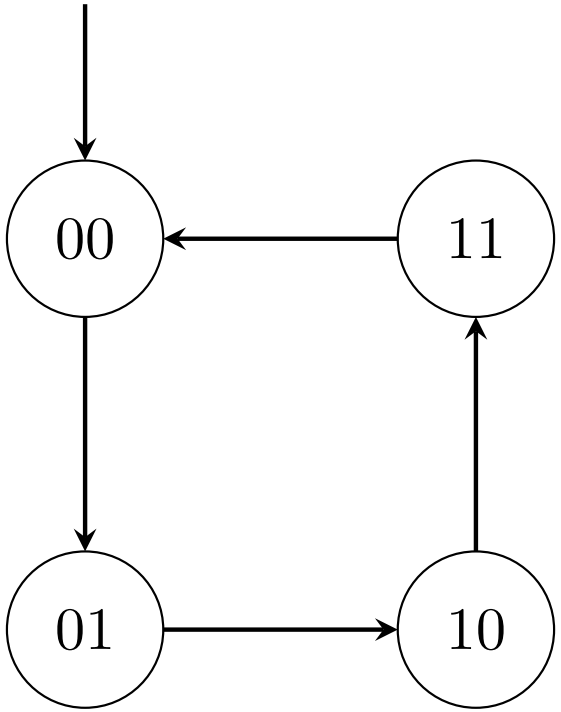


Two Boolean variables l and r (for the left and right bit).

Safety property: $G(\neg l \vee \neg r)$. Therefore $Target \equiv l \wedge r$.

$Init \equiv \neg l \wedge \neg r$. $Trans \equiv (r' =$

Example: Two Bit Counter

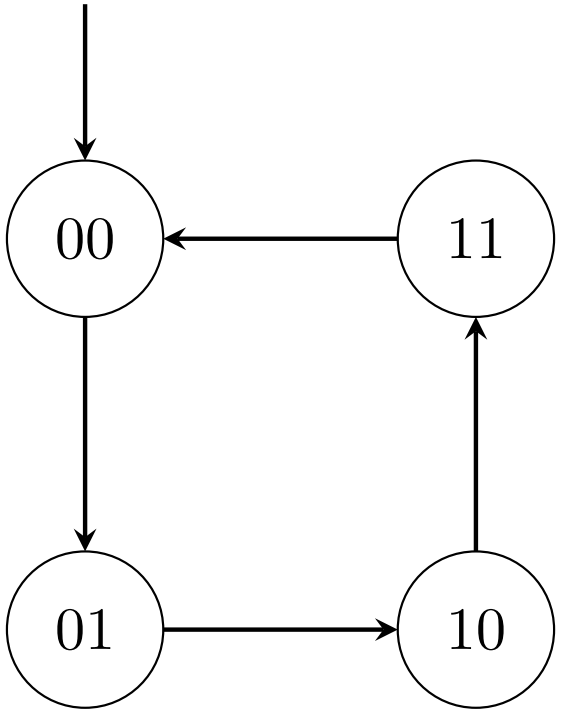


Two Boolean variables l and r (for the left and right bit).

Safety property: $G(\neg l \vee \neg r)$. Therefore $Target \equiv l \wedge r$.

$Init \equiv \neg l \wedge \neg r$. $Trans \equiv (r' = \neg r$

Example: Two Bit Counter

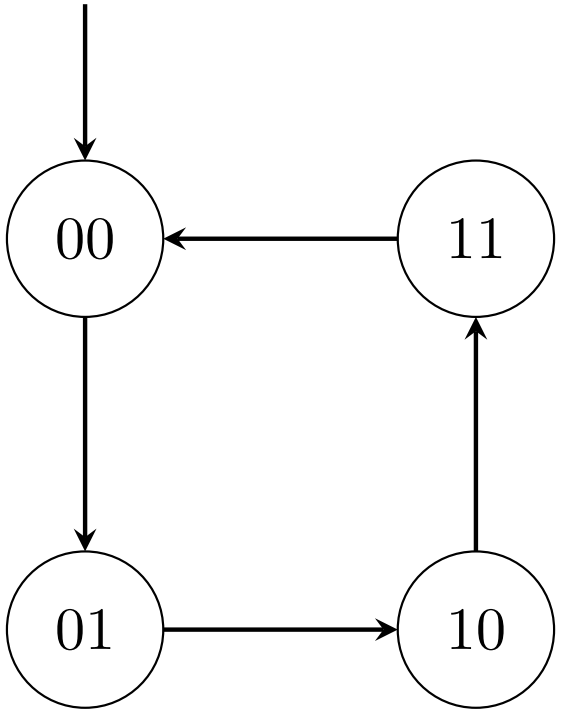


Two Boolean variables l and r (for the left and right bit).

Safety property: $G(\neg l \vee \neg r)$. Therefore $Target \equiv l \wedge r$.

$Init \equiv \neg l \wedge \neg r$. $Trans \equiv (r' = \neg r \wedge l' =$

Example: Two Bit Counter

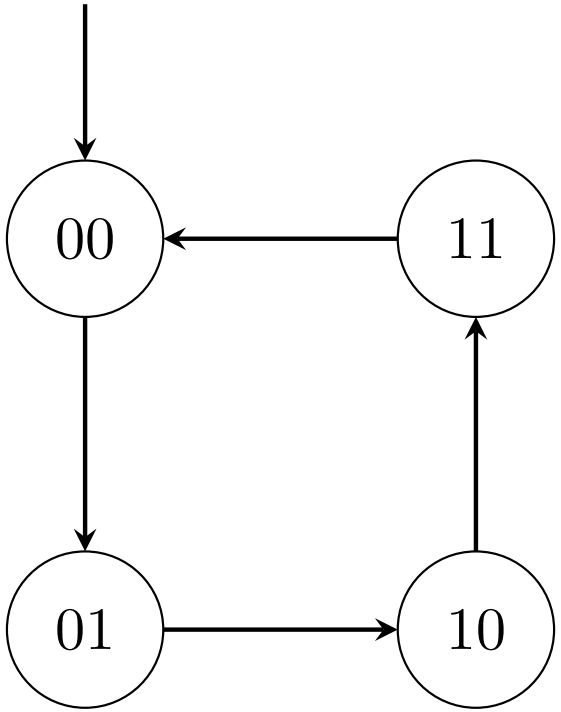


Two Boolean variables l and r (for the left and right bit).

Safety property: $G(\neg l \vee \neg r)$. Therefore $Target \equiv l \wedge r$.

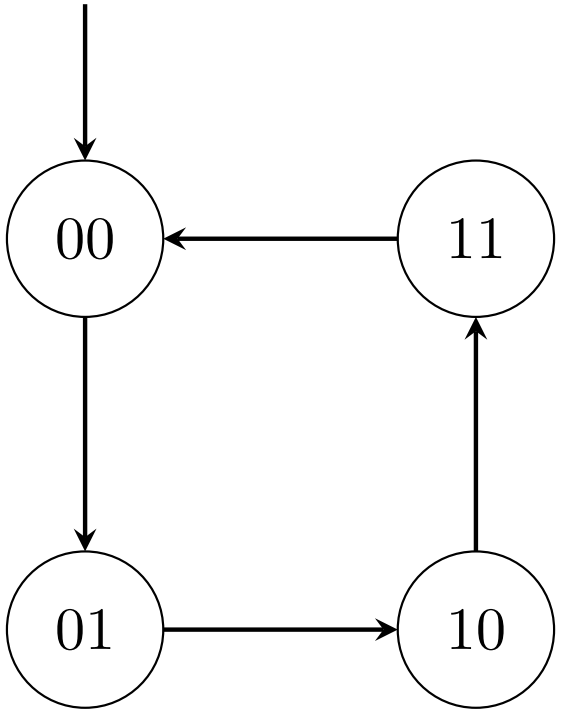
$Init \equiv \neg l \wedge \neg r$. $Trans \equiv (r' = \neg r \wedge l' = (l \neq r))$.

Example: Two Bit Counter



$$\Phi_2 = (\neg l_0 \wedge \neg r_0) \wedge \begin{pmatrix} l_1 = (l_0 \neq r_0) \wedge r_1 = \neg r_0 \wedge \\ l_2 = (l_1 \neq r_1) \wedge r_2 = \neg r_1 \end{pmatrix} \wedge \begin{pmatrix} (l_0 \wedge r_0) \vee \\ (l_1 \wedge r_1) \vee \\ (l_2 \wedge r_2) \end{pmatrix}$$

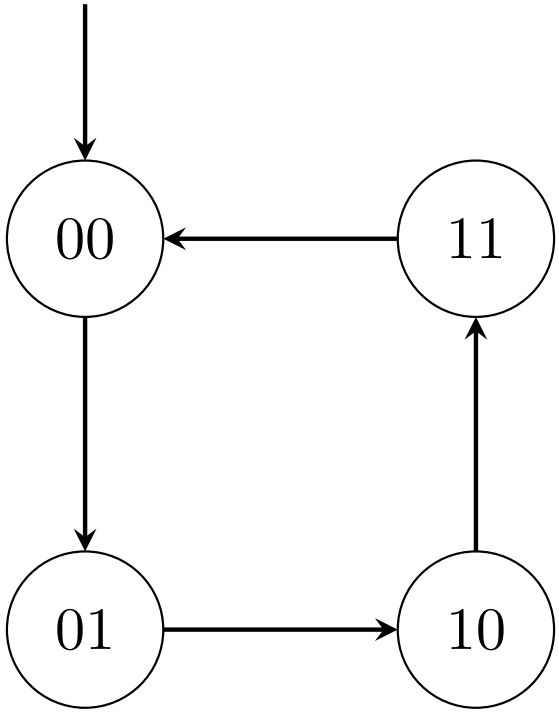
Example: Two Bit Counter



$$\Phi_2 = (\neg l_0 \wedge \neg r_0) \wedge \begin{pmatrix} l_1 = (l_0 \neq r_0) \wedge r_1 = \neg r_0 \wedge \\ l_2 = (l_1 \neq r_1) \wedge r_2 = \neg r_1 \end{pmatrix} \wedge \begin{pmatrix} (l_0 \wedge r_0) \vee \\ (l_1 \wedge r_1) \vee \\ (l_2 \wedge r_2) \end{pmatrix}$$

Φ_2 is **unsatisfiable** (hence, no proof of violation of the safety property).

Example: Two Bit Counter



$$\Phi_2 = (\neg l_0 \wedge \neg r_0) \wedge \begin{pmatrix} l_1 = (l_0 \neq r_0) \wedge r_1 = \neg r_0 \wedge \\ l_2 = (l_1 \neq r_1) \wedge r_2 = \neg r_1 \end{pmatrix} \wedge \begin{pmatrix} (l_0 \wedge r_0) \vee \\ (l_1 \wedge r_1) \vee \\ (l_2 \wedge r_2) \end{pmatrix}$$

Φ_2 is **unsatisfiable** (hence, no proof of violation of the safety property). Φ_3 **is satisfiable** (so the safety property is violated).

Further Reading:

- **Biere, Armin, et al.** "*Symbolic model checking without BDDs.*" International conference on tools and algorithms for the construction and analysis of systems. Springer 1999. https://link.springer.com/content/pdf/10.1007/3-540-49059-0_14.pdf

Optional :

- **Järvisalo, Matti, et al.** "*The international SAT solver competitions*." AI Magazine 33.1 (2012): 89-92. https://www.aaai.org/ojs/index.php/aimagazine

Acknowledgements: Partly based on earlier slides by Dr Corina Cîrstea and Dr Gennaro Parlato, University of Southampton, and John Harrison, Intel Corp.