lecture 22:
background on model checking
5590: software defined networking

anduo wang, Temple University

TTLMAN 401B, R 17:30-20:00

Grand Challenge Problem:
Model Check Concurrent Software

- Edmund M. Clarke
- Department of Computer Science
- Carnegie Mellon University

Outline of Talk

1. Explain what model checking is
2. Some successes of model checking
3. What makes software different
4. Approaches to software model checking
5. Some of my projects

Temporal Logic Model Checking

Temporal Logic Model Checking

- Model checking is an **automatic verification technique** for finite state concurrent systems.

Temporal Logic Model Checking

- Model checking is an **automatic verification technique** for finite state concurrent systems.

- Developed independently by **Clarke, Emerson, and Sistla** and by **Queille and Sifakis** in early 1980's.

Temporal Logic Model Checking

- Model checking is an **automatic verification technique** for finite state concurrent systems.
- Developed independently by **Clarke, Emerson, and Sistla** and by **Queille and Sifakis** in early 1980's.
- **Specifications** are written in **propositional temporal logic.**

Temporal Logic Model Checking

- Model checking is an **automatic verification technique** for finite state concurrent systems.
- Developed independently by **Clarke, Emerson, and Sistla** and by **Queille and Sifakis** in early 1980's.
- **Specifications** are written in **propositional temporal logic.**
- Verification procedure is an **exhaustive search of the state space** of the design.

Advantages of Model Checking

Advantages of Model Checking

- **No proofs!!!**

Advantages of Model Checking

- **No proofs!!!**
- **Fast (compared to other rigorous methods such as theorem proving)**

Advantages of Model Checking

- **No proofs!!!**
- **Fast (compared to other rigorous methods such as theorem proving)**
- **Diagnostic counterexamples**

Advantages of Model Checking

- **No proofs!!!**
- **Fast (compared to other rigorous methods such as theorem proving)**
- **Diagnostic counterexamples**
- **No problem with partial specifications**

Advantages of Model Checking

- **No proofs!!!**
- **Fast (compared to other rigorous methods such as theorem proving)**
- **Diagnostic counterexamples**
- **No problem with partial specifications**
- **Logics can easily express many concurrency properties**

Main Disadvantage

State Explosion Problem:

- Too many processes
- Data Paths

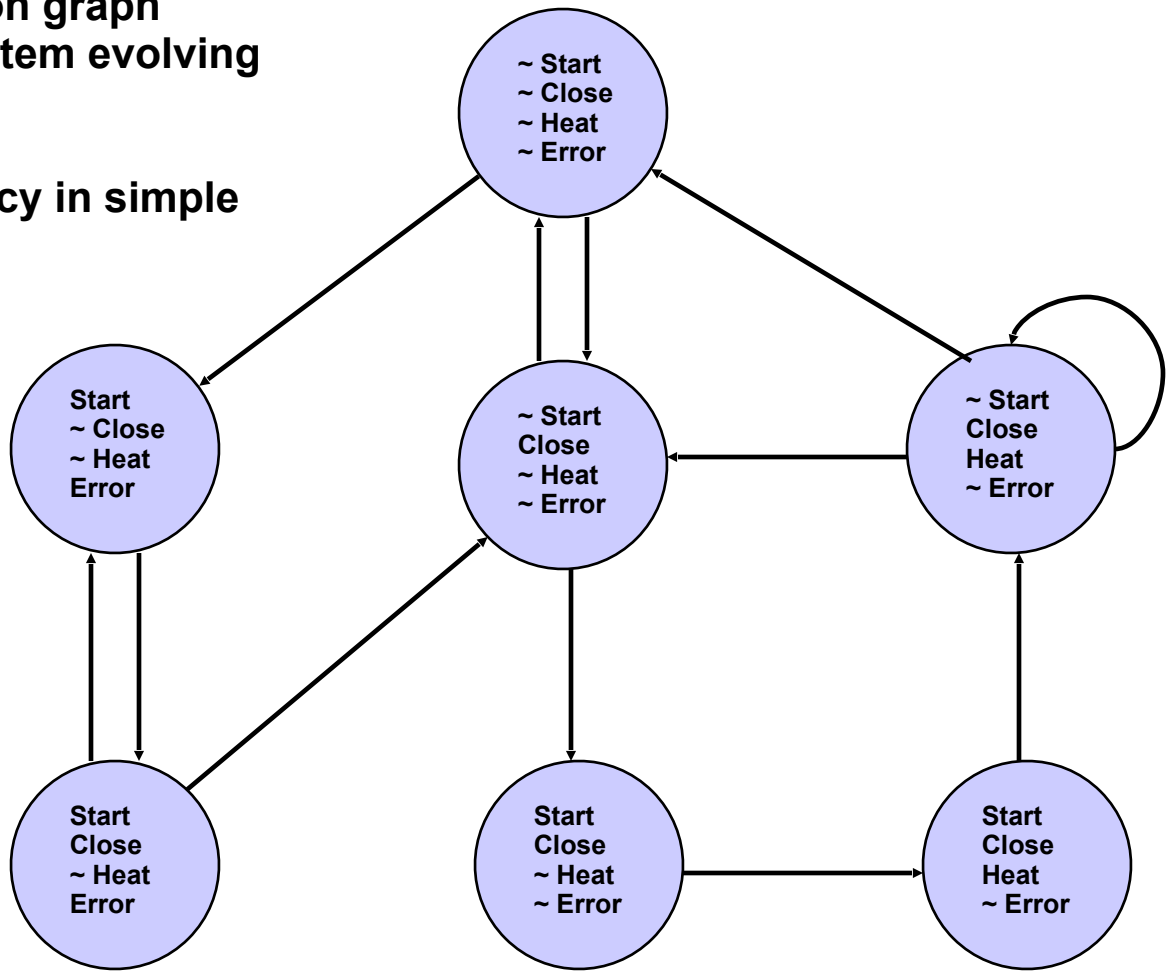
Much progress has been made on this problem recently!

Model of computation

Microwave Oven Example

State-transition graph describes system evolving over time.

No concurrency in simple examples.



Temporal Logic

Temporal Logic

- The oven doesn’t **heat up** until the **door is closed**.

Temporal Logic

- The oven doesn’t **heat up** until the **door is closed**.

- **Not heat_up** holds **until door_closed**

Temporal Logic

- The oven doesn't **heat up** until the **door is closed**.
- **Not heat_up** holds **until door_closed**
- **(~ heat_up) U door_closed**

Basic Temporal Operators

The symbol "**p**" is an atomic proposition, e.g. "**Device Enabled**".

- **F**p - p holds sometime in the ***future.***
- **G**p - p holds ***globally*** in the future.
- **X**p - p holds ***next*** time.
- p**U**q - p holds ***until*** q holds.

Model Checking Problem

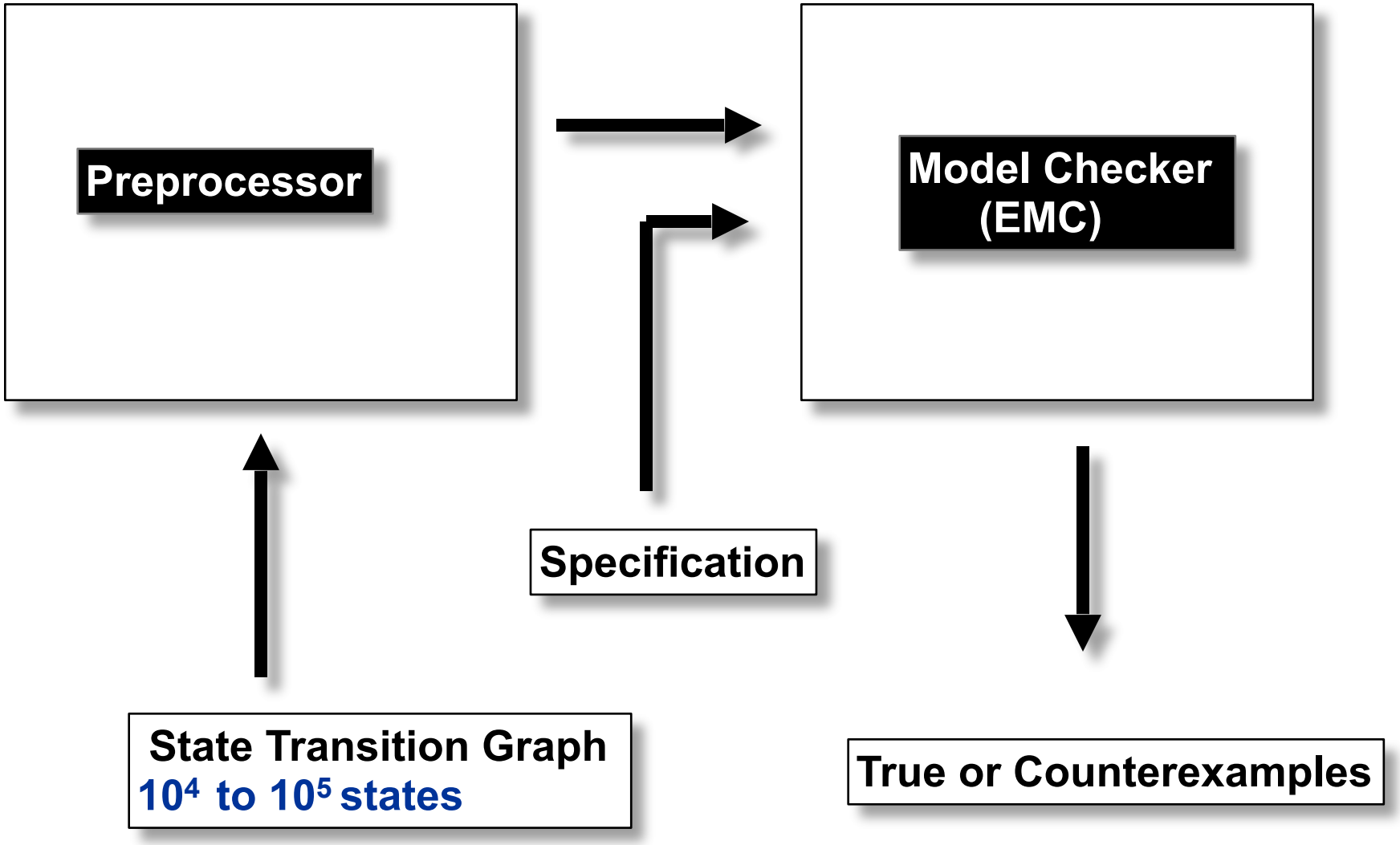
Let ***M*** be a **state-transition graph**.

Let ***f*** be the **specification** in temporal logic.

Find all states ***s*** of ***M*** such that $M, s \models f$.

Efficient Algorithms: CE81, CES83

The EMC System



Breakthrough!

Ken McMillan implemented our model checking algorithm using **Binary Decision Diagrams** in 1987.

Model Checker SMV

Now able to handle much larger examples!!

Combating the State Explosion Problem

- **Binary Decision Diagrams** can be used to represent state transition systems more efficiently.
- The **partial order reduction** can be used to reduce the number of states that must be enumerated.
- Many techniques for alleviating state explosion:
  - **Abstraction.**
  - **Compositional reasoning.**
  - **Symmetry.**
  - **Cone of influence reduction.**
  - **Semantic minimization.**

Model Checker Performance

Model Checker Performance

- Model checkers today can routinely handle systems with between **100** and **1000 state variables**.

Model Checker Performance

- Model checkers today can routinely handle systems with between **100** and **1000 state variables**.
- Systems with **10^{120} reachable states** have been checked. (Compare approx. **10^{78}** atoms in universe.)

Model Checker Performance

- Model checkers today can routinely handle systems with between **100** and **1000 state variables**.
- Systems with **10^{120} reachable states** have been checked. (Compare approx. **10^{78}** atoms in universe.)
- By using appropriate abstraction techniques, systems with an essentially **unlimited number of states** can be checked.

Temporal Logic Model Checkers

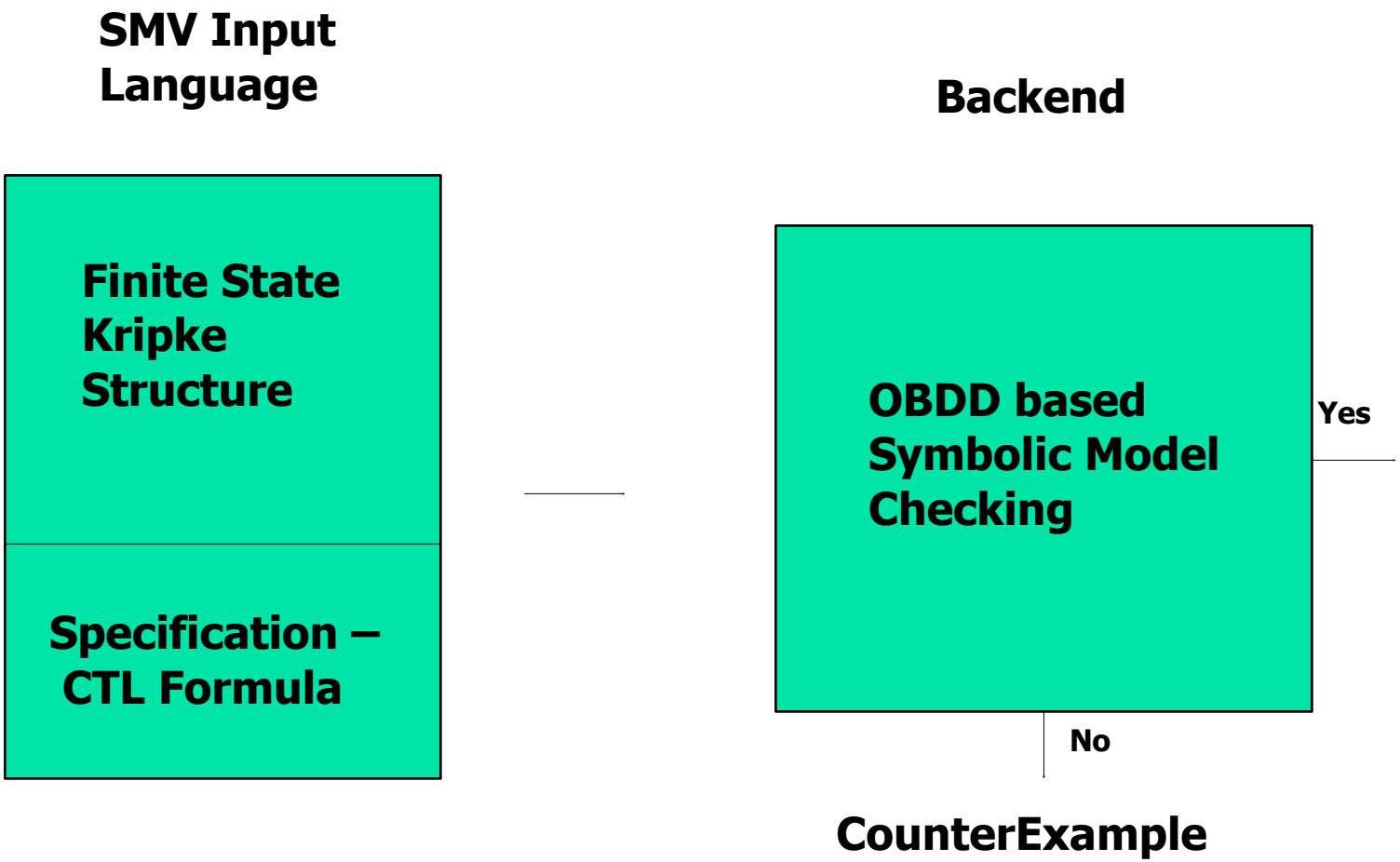
- The first two model checkers were **EMC** (Clarke, Emerson, Sistla) and **Caesar** (Queille, Sifakis).
- **SMV** is the first model checker to use **BDDs**.
- **Spin** uses the **partial order reduction** to reduce the state explosion problem.
- **Verus** and **Kronos** check properties of **real-time systems**.
- **HyTech** is designed for reasoning about **hybrid systems**.

Introduction to SMV

Symbolic Model Verifier

- Ken McMillan, Symbolic Model Checking: An Approach to the State Explosion Problem, 1993.
- Finite-state Systems described in a specialized language
- Specifications given as CTL formulas
- Internal representation using OBDDs
- Automatically verifies specification or produces a counterexample

Overview of SMV



Kripke model

Kripke model A model of basic modal logic is specified by (W, R, L), where:

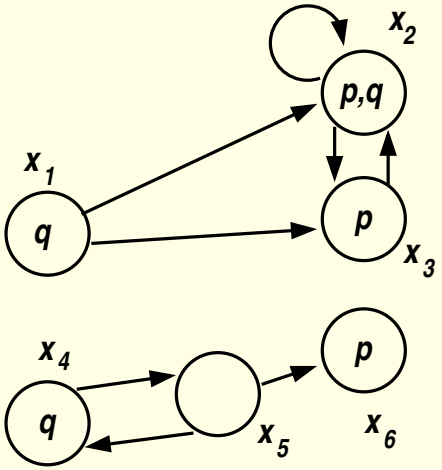
- W is a set, whose elements are called worlds
- $R \subseteq W \times W$: accessibility relation
- $L : W \to \mathcal{P}(\text{PropVar})$: labeling function

Example:

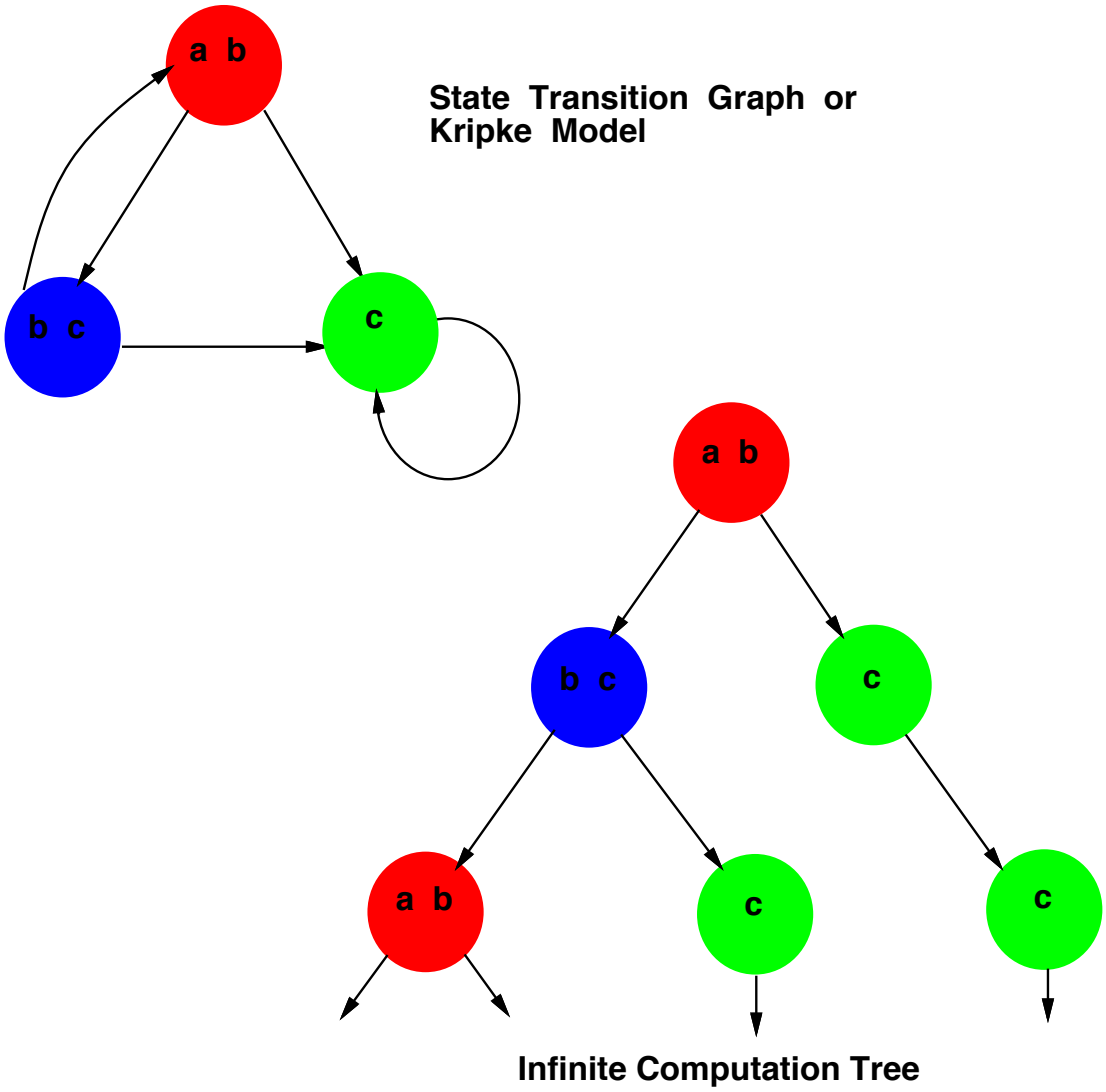
$$W = \{x_1, \ldots, x_6\}$$

$$R = \{(x_1, x_2), (x_1, x_3), (x_2, x_2), (x_3, x_2), (x_4, x_5), (x_5, x_4), (x_5, x_6)\}$$

x	x_1	x_2	x_3	x_4	x_5	x_6
$L(x)$	$\{q\}$	$\{p, q\}$	$\{p\}$	$\{q\}$	$\emptyset$	$\{p\}$



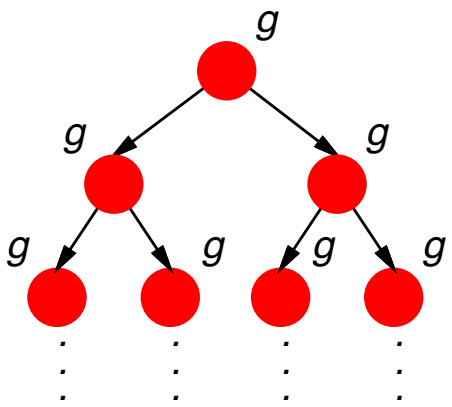
Kripke model and computation tree



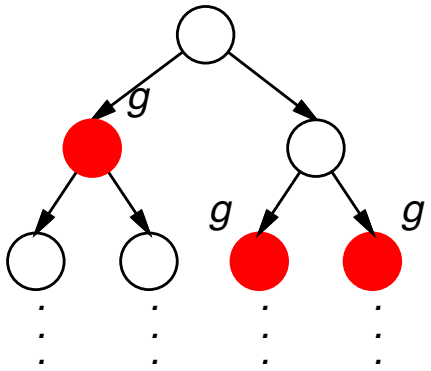
computation tree logic (CTL)

1. Path quantifier:
   - **A**—"for every path"
   - **E**—"there exists a path"
2. Linear-time operators:
   - **X**p—p holds next time.
   - **F**p—p holds sometime in the future
   - **G**p—p holds globally in the future
   - p**U**q—p holds until q holds

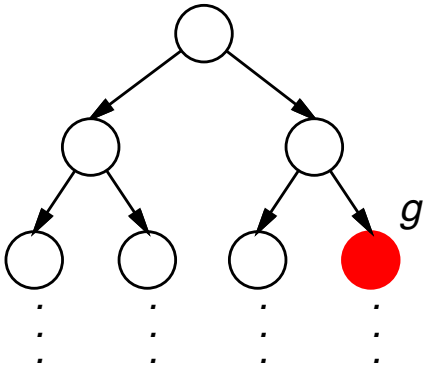
CTL formula



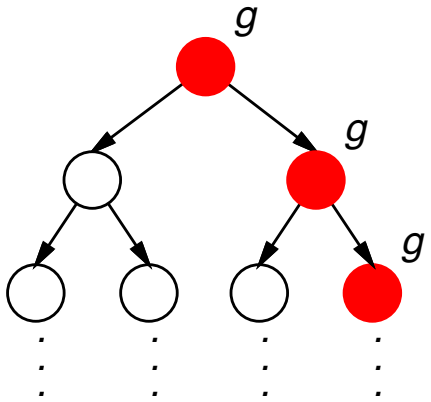
$M, s_0 \models \mathbf{AG}\, g$



$M, s_0 \models \mathbf{AF}\, g$



$M, s_0 \models \mathbf{EF}\, g$



$M, s_0 \models \mathbf{EG}\, g$

typical CTL formula

- **EF**$(Started \wedge \neg Ready)$: it is possible to get to a state where *Started* holds but *Ready* does not hold.
- **AG**$(Req \Rightarrow$ **AF**$Ack)$: if a *Request* occurs, then it will be eventually *Acknowledged*.
- **AG**(**AF** $DeviceEnabled$): *DeviceEnabled* holds infinitely often on every computation path.
- **AG**(**EF** $Restart$): from any state it is possible to get to the *Restart* state.

SMV Language Characteristics

- Allows description of completely synchronous to asynchronous systems, detailed to abstract systems
- Modularized and hierarchical descriptions
- Finite data types: Boolean and enumerated

The first SMV Program

```
MODULE main
VAR
    request: boolean;
    state: {ready, busy};
ASSIGN
    init(state) := ready;
    next(state) :=
        case
                state=ready & request: busy;
                1: {ready, busy};
        esac;
SPEC AG(request -> AF (state = busy))
```

Modularization

```
DEFINE
    a := 0;
VAR
    b : bar(a);
...
MODULE bar(x)
DEFINE
    a := 1;
    y := x;
```

Modularization

```
…
VAR
   a : boolean;
   b : foo(a);
…
MODULE foo(x)
ASSIGN
   x:=1;
```

Asynchronous Composition

```
MODULE main
VAR
    gate1: process inverter(gate3.output);
    gate2: process inverter(gate1.output);
    gate3: process inverter(gate2.output);
SPEC
(AG AF gate1.output) & (AG AF !gate1.output)

MODULE inverter(input)
VAR     output: boolean;
ASSIGN
    init(output) := 0;
    next(output) := !input;
```

Fairness

- FAIRNESS ctl_formulae
  - Assumed to be true infinitely often
  - Model checker only explores paths satisfying fairness constraint
  - Each fairness constraint must be true infinitely often

- If there are no fair paths
  - All existential formulas are false
  - All universal formulas are true

- FAIRNESS running

With Fairness..

```
MODULE main
VAR
    gate1: process inverter(gate3.output);
    gate2: process inverter(gate1.output);
    gate3: process inverter(gate2.output);
SPEC
(AG AF gate1.output) & (AG AF !gate1.output)

MODULE inverter(input)
VAR      output: boolean;
ASSIGN
    init(output) := 0;
    next(output) := !input;

FAIRNESS
  running
```

Shared Data Example

Two Users assign pid to shared data in turn

```
MODULE main
VAR
   data : boolean;
   turn : boolean;
   user0 : user(0, data, turn);
   user1 : user(1, data, turn);
ASSIGN
   next(turn) := !turn;
SPEC
   AG (AF data & AF (!data))
```

Shared data example (cont..)

```
MODULE user(pid, data, turn)
ASSIGN
 next(data) := case
   turn: pid;
   1 : data;
 esac;
```

Run SMV

- smv [options] inputfile
  - **-c cache-size for BDD operations**
  - **-k key-table-size for BDD nodes**
  - **-v verbose**
  - **-int interactive mode**
  - **-r**
    - prints out statistics about reachable state space

Example: Client & Server

```
MODULE client (ack)
VAR
  state : {idle, requesting};
  req : boolean;

ASSIGN
  init(state) := idle;
  next(state) :=
   case
    state=idle : {idle, requesting};
    state=requesting & ack : {idle, requesting};
    1 : state;
   esac;

  req := (state=requesting);
```

```
MODULE server (req)

VAR
  state : {idle, pending, acking};
  ack : boolean;

ASSIGN
  next(state) :=
   case
    state=idle & req : pending;
    state=pending : {pending, acking};
    state=acking & req : pending;
    state=acking & !req : idle;
    1 : state;
   esac;

  ack := (state = acking);
```

Is the specification true?

```
MODULE main
VAR
  c : client(s.ack);
  s : server(c.req);

SPEC AG (c.req -> AF s.ack)
```

- Need fairness constraint:
  - Solution:
    FAIRNESS (c.req -> s.ack)

NuSMV

- Specifications expressible in CTL, LTL and Real time CTL logics
- Provides both BDD and SAT based model checking.
- Uses a number of heuristics for achieving efficiency and control state explosion
- Higher number of features in interactive mode