Formal Verification by Model Checking

Jonathan Aldrich
Carnegie Mellon University

Based on slides developed by Natasha Sharygina

15-413: Introduction to Software Engineering
Fall 2005

Formal Verification by Model Checking

Domain: Continuously operating concurrent systems (e.g. operating systems, hardware controllers and network protocols)

- Ongoing, reactive semantics
  - Non-terminating, infinite computations
  - Manifest non-determinism

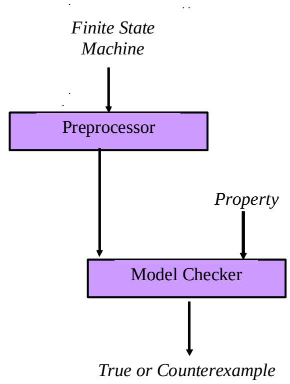
Instrument: Temporal logic [Pnueli 77] is a formalism for reasoning about behavior of reactive systems

Temporal Logic Model Checking

[Clarke,Emerson 81][Queille,Sifakis 82]

- Systems are modeled by **finite state machines**
- **Properties** are written in **propositional temporal logic**
- Verification procedure is an **exhaustive search of the state space** of the design
- **Diagnostic counterexamples**

Temporal Logic Model Checking



What is Model Checking?

Does model M satisfy a property P ?
(written M |= P)

What is "M"?

What is "P"?

What is "satisfy"?

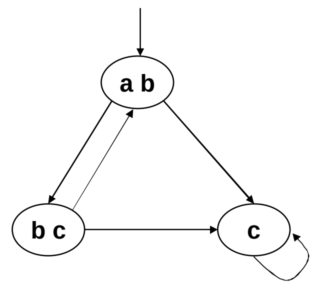
What is "M"?

States: valuations to all variables

Initial states: subset of states

Arcs: transitions between states

Atomic Propositions:
e.g. x = 5, y = true



State Transition Graph or Kripke Model

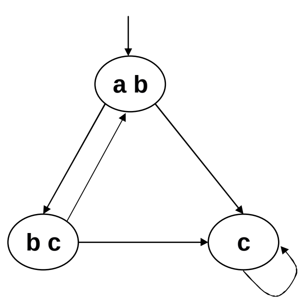
What is "M"?

$$M = \langle S, S_0, R, L \rangle$$

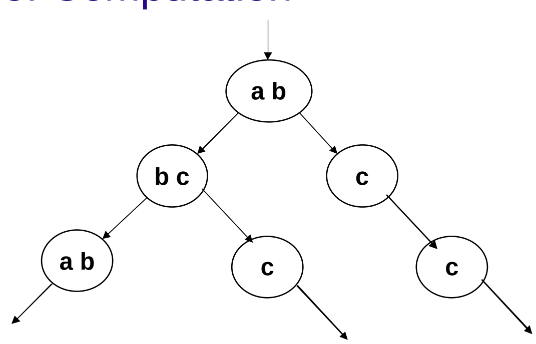
Kripke structure:

- S – finite set of states
- $S_0 \subseteq S$ – set of initial states
- $R \subseteq S \times S$ – set of arcs
- $L : S \rightarrow 2^{AP}$ – mapping from states to a set of atomic propositions

Model of Computation



State Transition Graph

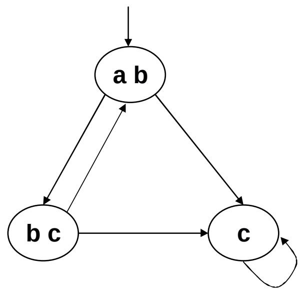


Infinite Computation Tree

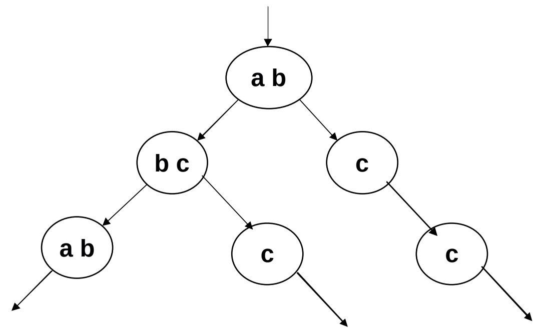
Unwind State Graph to obtain Infinite Tree.

A trace is an infinite sequence of states.

Semantics



State Transition Graph



Infinite Computation Tree

The semantics of a FSM is a set of traces. Semantics of the composition of FSMs is the intersection of traces of individual FSMs.

What is “P”?

Different kinds of temporal logics

Syntax: What are the formulas in the logic?

Semantics: What does it mean for model **M** to satisfy formula **P**?

Formulas:

- Atomic propositions: properties of states
- Temporal Logic Specifications: properties of traces.

Computation Tree Logics

Examples:

Safety (mutual exclusion): no two processes can be at a critical section at the same time

Liveness (absence of starvation): every request will be eventually granted

Temporal logics differ according to how they handle branching in the underlying computation tree.

In a linear temporal logic (LTL), operators are provided for describing system behavior along a single computation path.

In a branching-time logic (CTL), the temporal operators quantify over the paths that are possible from a given state.

Computation Tree Logics

Formulas are constructed from ***path quantifiers*** and ***temporal operators:***

1. ***Path Quantifiers:***
   - **A** – "for every path"
   - **E** – "there exists a path"

2. ***Temporal Operator:***
   - **X**α **-** α holds **next** time
   - **F**α **-** α holds sometime in the **future**
   - **G**α **-** α holds **globally** in the **future**
   - α**U**β **-** α holds **until** β holds

Formulas over States and Paths

- State formulas
  - Describe a property of a state in a model M
  - If $p \in AP$, then p is a state formula
  - If f and g are state formulas, then $\neg f$, $f \wedge g$ and $f \vee g$ are state formulas
  - If f is a path formula, then **E** f and **A** f are state formulas
- Path formulas
  - Describe a property of an infinite path through a model M
  - If f is a state formula, then f is also a path formula
  - If f and g are path formulas, then $\neg f$, $f \wedge g$, $f \vee g$, **X** f, **F** f, **G** f, and f **U** g are path formulas

Notation

- A path π in M is an infinite sequence of states $s_0, s_1, \ldots$ such that for every $i \geq 0$, $(s_i, s_{i+1}) \in R$
- π^i denotes the suffix of π starting at s_i
- If f is a state formula, $M, s \models f$ means that f holds at state s in the Kripke structure M
- If f is a path formula, $M, \pi \models f$ means that f holds along path π in the Kripke structure M

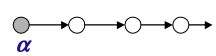
Semantics of Formulas

$M, s \models p$	$\Leftrightarrow p \in L(s)$	$M, \pi \models f$	$\Leftrightarrow \pi = s\ldots \wedge M, s \models f$
$M, s \models \neg f$	$\Leftrightarrow M, s \not\models f$	$M, \pi \models \neg g$	$\Leftrightarrow M, \pi \not\models g$
$M, s \models f_1 \wedge f_2$	$\Leftrightarrow M, s \models f_1 \wedge M, s \models f_2$	$M, \pi \models g_1 \wedge g_2$	$\Leftrightarrow M, \pi \models g_1 \wedge M, \pi \models g_2$
$M, s \models f_1 \vee f_2$	$\Leftrightarrow M, s \models f_1 \vee M, s \models f_2$	$M, \pi \models g_1 \vee g_2$	$\Leftrightarrow M, \pi \models g_1 \vee M, \pi \models g_2$
$M, s \models \mathbf{E}\, g_1$	$\Leftrightarrow \exists \pi = s\ldots \mid M, \pi \models g_1$	$M, \pi \models \mathbf{X}\, g$	$\Leftrightarrow M, \pi^1 \models g$
$M, s \models \mathbf{A}\, g_1$	$\Leftrightarrow \forall \pi = s\ldots\ M, \pi \models g_1$	$M, \pi \models \mathbf{F}\, g$	$\Leftrightarrow \exists k \geq 0 \mid M, \pi^k \models g$
		$M, \pi \models \mathbf{G}\, g$	$\Leftrightarrow \forall k \geq 0 \mid M, \pi^k \models g$
		$M, \pi \models g_1\, \mathbf{U}\, g_2$	$\Leftrightarrow \exists k \geq 0 \mid M, \pi^k \models g_2$
			$\wedge\ \forall 0 \leq j < k\ M, \pi^j \models g_1$

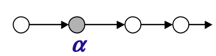
The Logic LTL

Linear Time Logic (LTL) [Pnueli 77]: logic of temporal sequences.
Has form **A** *f* where *f* is a path formula which has no path quantifiers (**A** or **E**)

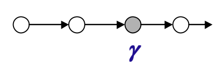
- ***α*: *α* holds in the current state**



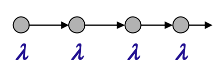
- **AX*α*: *α* holds in the next state**



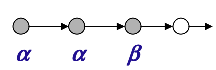
- **AF*γ*: *γ* holds eventually**



- **AG*λ*: *λ* holds from now on**

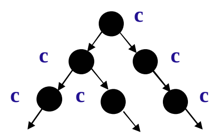


- ***A(α U β): α holds until β holds***

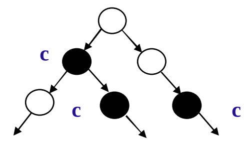


The Logic CTL

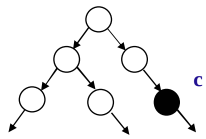
In a branching-time logic (CTL), the temporal operators quantify over the paths that are possible from a given state (s_0). Requires each temporal operator (**X**, **F**, **G**, and **U**) to be preceded by a path quantifier (**A** or **E**).



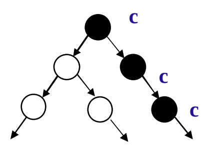
M, s_0 ⊨ AG c



M, s_0 ⊨ AF c



M, s_0 ⊨ EF c



M, s_0 ⊨ EG c

Typical CTL Formulas

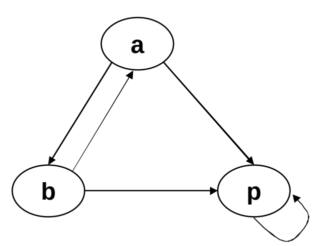
- **EF** (*Started* ∧ ¬ *Ready*): it is possible to get to a state where *Started* holds but *Ready* does not hold.
- **AG** (*Req* ⇒ **AF** *Ack*): whenever *Request* occurs, it will be eventually *Acknowledged.*
- **AG** (*DeviceEnabled*): *DeviceEnabled* always holds on every computation path.
- **AG** (**EF** *Restart*): from any state it is possible to get to the *Restart* state.

Announcements

- Please email your Stack.java file to Marwan for Assignment 8 part 4
  - This will help with the grading

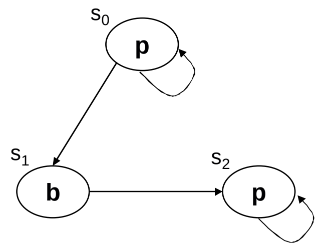
Trivia

- **AG**(**EF** *p*) cannot be expressed in LTL
  - Reset property: from every state it is possible to get to *p*
    - But there might be paths where you never get to *p*
  - Different from **A**(**GF** *p*)
    - Along each possible path, for each state in the path, there is a future state where *p* holds
    - Counterexample: ababab…



Trivia

- **A**(**FG** *p*) cannot be expressed in CTL
  - Along all paths, one eventually reaches a point where *p* always holds from then on
    - But at some points in some paths where *p* always holds, there might be a diverging path where *p* does not hold
  - Different from AF(AG *p*)
    - Along each possible path there exists a state such that *p* always holds from then on
    - Counterexample: the path that stays in s_0



LTL Conventions

- Often leave the initial **A** implicit
- G is sometimes written □
- F is sometimes written ◇

Linear vs. branching-time logics

some advantages of LTL

- LTL properties are preserved under "abstraction": i.e., if $\mathcal{M}$ "approximates" a more complex model $\mathcal{M}'$, by introducing more paths, then
  $\mathcal{M} \models \psi \Rightarrow \mathcal{M}' \models \psi$
- "counterexamples" for LTL are simpler: consisting of single executions (rather than trees).
- The automata-theoretic approach to LTL model checking is simpler (no tree automata involved).
- anecdotally, it seems most properties people are interested in are linear-time properties.

some advantages of BT logics

- BT allows expression of some useful properties like 'reset'.
- CTL, a limited fragment of the more complete BT logic CTL*, can be model checked in time linear in the formula size (as well as in the transition system). But formulas are usually far smaller than system models, so this isn't as important as it may first seem.
- Some BT logics, like μ-calculus and CTL, are well-suited for the kind of fixed-point computation scheme used in symbolic model checking.

CTL Model Checking

- Theorem: Any CTL formula can be expressed in terms of ¬, ∨, **EX**, **EU**, and **EG**.
  - **F** p = true **U** p
  - **A**[x **U** y] = ¬(**EG** ¬y ∨ **E**[¬y **U** ¬(x∨y)])
  - **AX** p = ¬**EX** ¬p
  - **AG** p = ¬**EF** ¬p
- Model checking: determine which states of *M* satisfy *f*
- Algorithm
  - Consider all subformulas of *f*, in order of depth of nesting
  - Initially, label each state with the atomic subformulas that are true in that state
  - For each formula, use information about the states where the immediate subformulas are true to label states with the new formula

Carnegie Mellon

Subformula Labeling

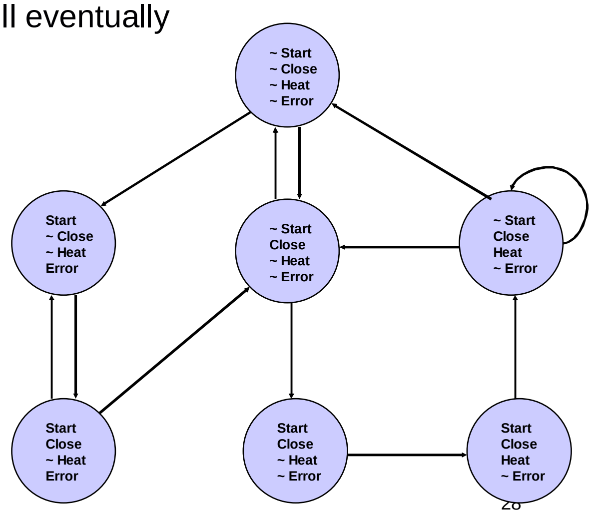
- Case $\neg f$
  – Label each state not labeled with f
- $f_1 \vee f_2$
  – Label each state which is labeled with either f_1 or f_2
- **EX** f
  – Label every state that has some successor labeled with f
- **E**[f_1 **U** f_2]
  – Label every state labeled with f_2
  – Traverse backwards from labeled states; if the previous state is labeled with f_1, label it with **E**[f_1 **U** f_2] as well
- **EG** f_1
  – Find strongly connected components where f_1 holds
  – Traverse backwards from labeled states; if the previous state is labeled with f_1, label it with **EG** f_1 as well

27

Carnegie Mellon

CTL Model Checking Example

- Pressing Start will eventually result in heat



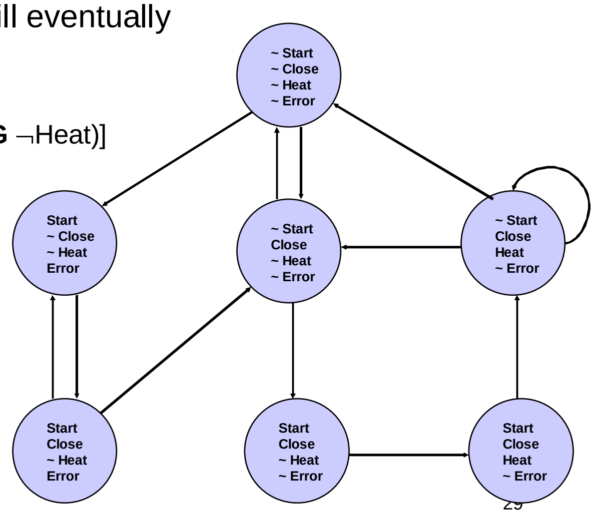
28

CTL Model Checking Example

- Pressing Start will eventually result in heat

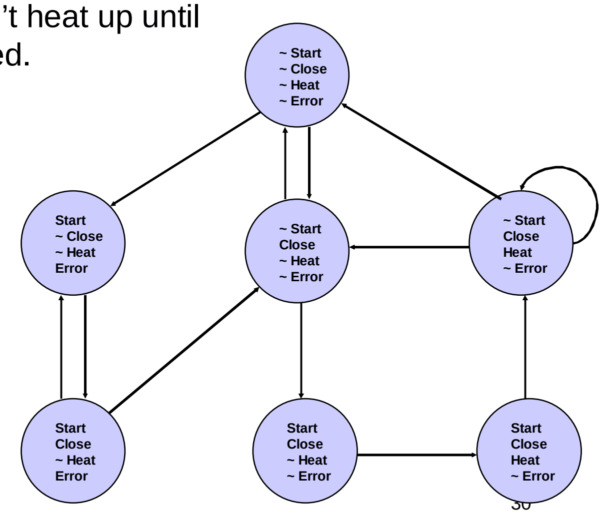
AG(Start ⇒ **AF** Heat)

= ¬**E**[*true* **U** (Start ∧ **EG** ¬Heat)]



CTL Model Checking Example

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



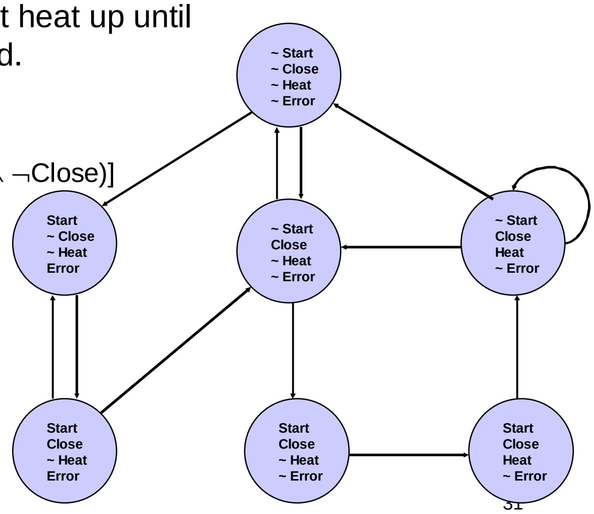
CTL Model Checking Example

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

A[(¬Heat) **U** Close]

= ¬**EG** ¬Close

∧ ¬**E**[¬Close **U** (Heat ∧ ¬Close)]



LTL Model Checking

- Beyond the scope of this course

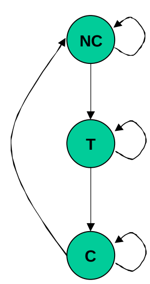
- Canonical reference on Model Checking:
  - Edmund Clarke, Orna Grumberg, and Doron A. Peled. Model Checking. MIT Press, 1999.

SPIN: The Promela Language

- PROcess MEta LAnguage
- Asynchronous composition of independent processes
- Communication using channels and global variables
- Non-deterministic choices and interleavings

An Example

```
mtype = { NONCRITICAL, TRYING, CRITICAL };
show mtype state[2];
proctype process(int id) {
beginning:
noncritical:
     state[id] = NONCRITICAL;
     if
     :: goto noncritical;
     :: true;
     fi;
trying:
     state[id] = TRYING;
     if
     :: goto trying;
     :: true;
     fi;
critical:
     state[id] = CRITICAL;
     if
     :: goto critical;
     :: true;
     fi;
     goto beginning;}
init { run process(0); run process(1); }
```



Enabled Statements

- A statement needs to be enabled for the process to be scheduled.

```
bool a, b;
proctype p1()
{
  a = true;
  a & b;
  a = false;
}
proctype p2()
{
  b = false;
  a & b;
  b = true;
}
init { a = false; b = false; run p1(); run p2(); }
```

These statements are enabled only if both **a** and **b** are true.

In this case **b** is always false and therefore there is a deadlock.

Other constructs

- Do loops

```
do
:: count = count + 1;
:: count = count - 1;
:: (count == 0) -> break
od
```

Other constructs

- Do loops
- Communication over channels

```
proctype sender(chan out)
{
  int x;

  if
  ::x=0;
  ::x=1;
  fi

   out ! x;
}
```


Other constructs

- Do loops
- Communication over channels
- Assertions

```
proctype receiver(chan in)
{
   int value;
   in ? value;
   assert(value == 0 || value == 1)
}
```

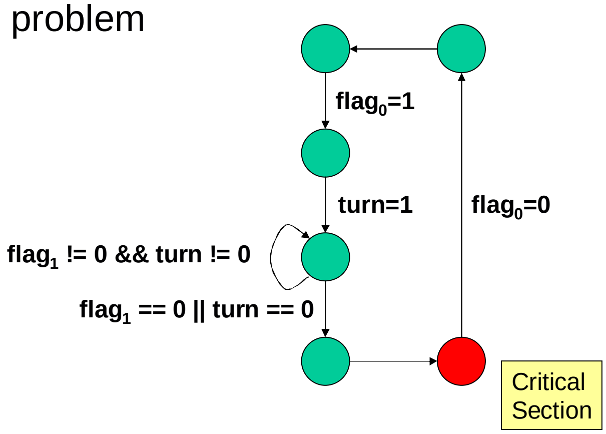
Other constructs

- Do loops
- Communication over channels
- Assertions
- Atomic Steps

```
int value;
proctype increment()
{  atomic {
     x = value;
     x = x + 1;
     value = x;
} }
```


Mutual Exclusion

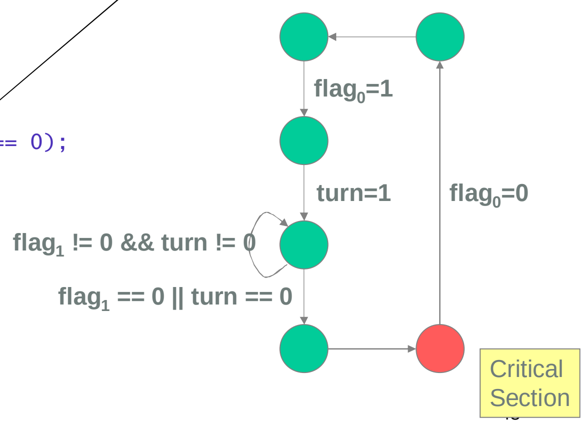
- Peterson's solution to the mutual exclusion problem



Mutual Exclusion in SPIN

```
bool turn;
bool flag[2];
proctype mutex0() {
again:
  flag[0] = 1;
  turn = 1;
  (flag[1] == 0 || turn == 0);
  /* critical section */
  flag[0] = 0;
  goto again;
}
```

guard:
Cannot go past this point until the condition is true



Mutual Exclusion in SPIN

Active process:
automatically creates instances of processes

_pid:
Identifier of the process

assert:
Checks that there are only at most two instances with identifiers 0 and 1

```
bool turn, flag[2];

active [2] proctype user()
{
  assert(_pid == 0 || __pid == 1);
again:
  flag[_pid] = 1;
  turn = 1 - _pid;
  (flag[1 - _pid] == 0 || turn == _pid);

      /* critical section */

  flag[_pid] = 0;
  goto again;
}
```

Mutual Exclusion in SPIN

```
bool turn, flag[2];
byte ncrit;
active [2] proctype user()
{
  assert(_pid == 0 || __pid == 1);
again:
  flag[_pid] = 1;
  turn = 1 - _pid;
  (flag[1 - _pid] == 0 || turn == _pid);

  ncrit++;
  assert(ncrit == 1); /* critical section */
  ncrit--;

  flag[_pid] = 0;
  goto again;
}
```

ncrit:
Counts the number of
Process in the critical section

assert:
Checks that there are always
at most one process in the
critical section

Mutual Exclusion in SPIN

```
bool turn, flag[2];
bool critical[2];

active [2] proctype user()
{
  assert(_pid == 0 || __pid == 1);
again:
  flag[_pid] = 1;
  turn = 1 - _pid;
  (flag[1 - _pid] == 0 || turn == _pid);

  critical[_pid] = 1;
  /* critical section */
  critical[_pid] = 0;

  flag[_pid] = 0;
  goto again;
}
```

LTL Properties:

The processes are never both
in the critical section
AG(!(critical[0] && critical[1]))
[](!(critical[0] && critical[1]))

No matter what happens, a
process will eventually get to
a critical section
[] <> (critical[0] || critical[1])

If process 0 is in the critical
section, process 1 will get to
be there next
[] (critical[0] -> critical[0] U
(!critical[0] U critical[1]))

Carnegie Mellon

Mutual Exclusion in SPIN

```
bool turn, flag[2];
bool critical[2];

active [2] proctype user()
{
  assert(_pid == 0 || __pid == 1);
again:
  flag[_pid] = 1;
  turn = 1 - _pid;
  (flag[1 - _pid] == 0 || turn == _pid);

  critical[_pid] = 1;
  /* critical section */
  critical[_pid] = 0;

  flag[_pid] = 0;
  goto again;
}
```

LTL Properties:

[] !(critical[0] && critical[1])

[] <> (critical[0])
[] <> (critical[1])

[] (critical[0] ->
  (critical[0] U
    (!critical[0] &&
      ((!critical[0] &&
        !critical[1]) U critical[1]))))

* caveat: can't use array indexes in SPIN LTL properties
Have to duplicate code

52

Carnegie Mellon

State Space Explosion

Problem:

Size of the state graph can be exponential in size of the program (both in the number of the program *variables* and the number of program *components*)

$$M = M_1 \parallel \ldots \parallel M_n$$

If each M_i has just 2 local states, potentially 2^n global states

Research Directions: State space reduction

53

Model Checking Performance

•Model Checkers today can routinely handle systems with between 100 and 300 state variables.

•Systems with 10^{120} reachable states have been checked.

•By using appropriate abstraction techniques, systems with an essentially **unlimited number of states** can be checked.

Notable Examples

- **IEEE Scalable Coherent Interface** – In 1992 Dill's group at Stanford used **Murphi** to find several errors, ranging from uninitialized variables to subtle logical errors
- **IEEE Futurebus** – In 1992 Clarke's group at CMU found previously undetected design errors
- **PowerScale multiprocessor** (processor, memory controller, and bus arbiter) was verified by Verimag researchers using CAESAR toolbox
- **Lucent telecom. protocols** were verified by FormalCheck – errors leading to lost transitions were identified
- **PowerPC 620 Microprocessor** was verified by Motorola's Verdict model checker.

The Grand Challenge: Model Check Software

Extract finite state machines from programs written in conventional programming languages

Use a finite state programming language:

- executable design specifications (Statecharts, xUML, etc.).

Unroll the state machine obtained from the executable of the program.

The Grand Challenge: Model Check Software

Use a combination of the state space reduction techniques to avoid generating too many states.

- **Verisoft (Bell Labs)**
- **FormalCheck/xUML (UT Austin, Bell Labs)**
- **ComFoRT (CMU/SEI)**

Use static analysis to extract a finite state skeleton from a program. Model check the result.

- **Bandera** – Kansas State
- **Java PathFinder** – NASA Ames
- **SLAM/Bebop** - Microsoft