Formal Verification by Model Checking

Natasha Sharygina
Carnegie Mellon University

Guest Lectures at the Analysis of Software Artifacts Class, Spring 2005

Outline

Lecture 1: Overview of Model Checking

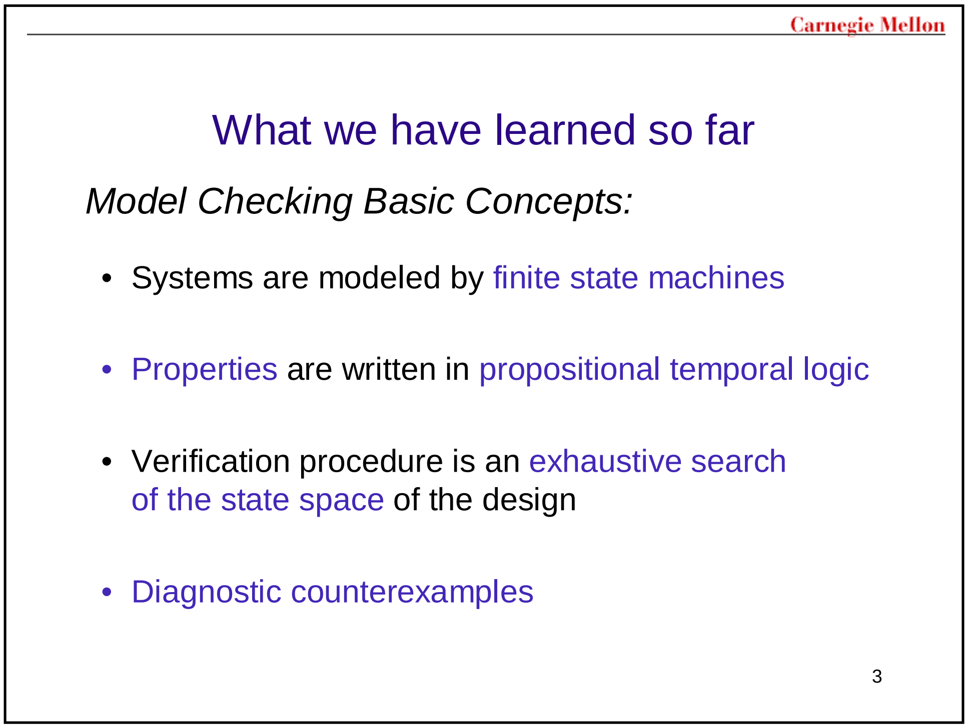
Lecture 2: Complexity Reduction Techniques

Lecture 3: Software Model Checking

Lecture 4: State/Event-based software model checking

Lecture 5: Component Substitutability

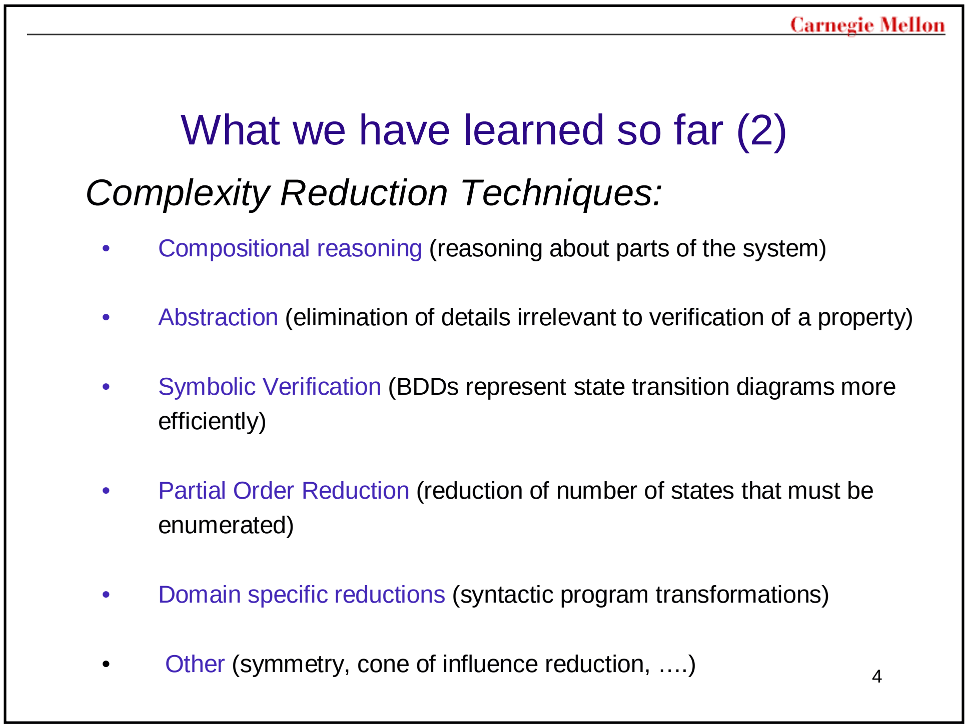
Lecture 6: Model Checking Practicum (Student Reports on the Lab exercises)



What we have learned so far

Model Checking Basic Concepts:

- 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



What we have learned so far (2)

Complexity Reduction Techniques:

- Compositional reasoning (reasoning about parts of the system)
- Abstraction (elimination of details irrelevant to verification of a property)
- Symbolic Verification (BDDs represent state transition diagrams more efficiently)
- Partial Order Reduction (reduction of number of states that must be enumerated)
- Domain specific reductions (syntactic program transformations)
- Other (symmetry, cone of influence reduction, ….)

Today’s Lecture

Various approaches to model checking software

Hypothesis

- Model checking is an algorithmic approach to analysis of finite-state systems
- Model checking has been originally developed for analysis of hardware designs and communication protocols
- Model checking algorithms and tools have to be tuned to be applicable to analysis of software

Application of Model Checking to Hardware Verification

– Simple data structures are used

– Systems are modular

– Mostly finite-state systems

– System components have well defined interfaces

– Mostly synchronous execution

Application of Model Checking to Software Verification

– Complex data structures are used

- Procedural or OO design

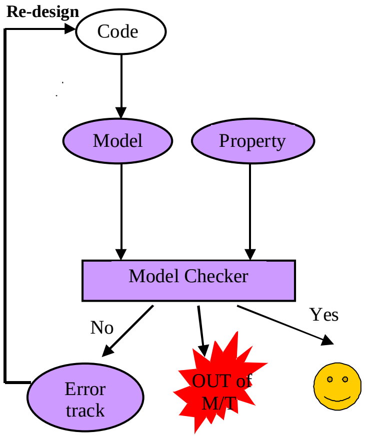
– Non-finite state systems

– System components do not have well defined interfaces

– Complex coordination between SW components

– Synchronous or asynchronous execution

Model Checking Software (code verification)



← *1. Design/Implementation/Testing*

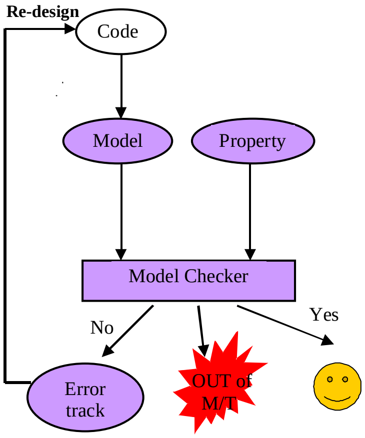
← *2. Modeling/Property Specification*

- Finite-state model extraction
- Simplifications
- Restrictions

← *3. Verification*

- Abstractions
- Divide-and-conquer techniques (when applicable)
- Other complexity reduction techn.

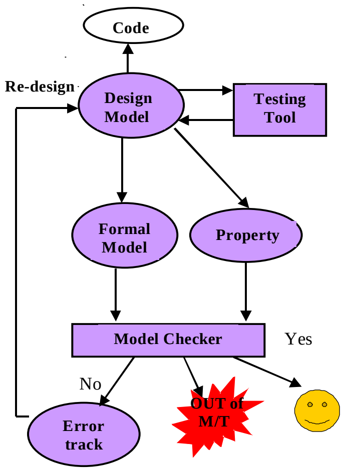
Model Checking Software (code verification)



Limitations:

- *Final* (expensive) *stage* in the program development
- *Consistency* problem between code and model
- Mostly limited to *simplified* systems

Model Checking Software (design verification)



←*4. Code Generation (last stage)*

←*1. Executable Design Specifications*

- Abstraction from low-level to high-level operations

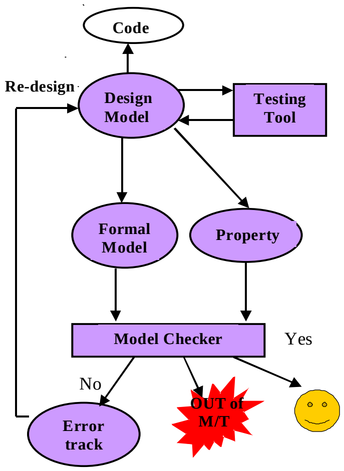
←*2. Modeling/Property Specification*

- Finite-state model extraction

←*3. Verification*

- State space reduction techniques

Model Checking Software (design verification)



Advantages:

- Applied earlier in the design cycle (*Earlier* bug detection)
- *Direct* translation of informal program into formal syntax (no simplifications)
- *Separation* of concerns: abstraction of control from data
- *Domain-specific* property specification

State-of-the-art Software Model Checking

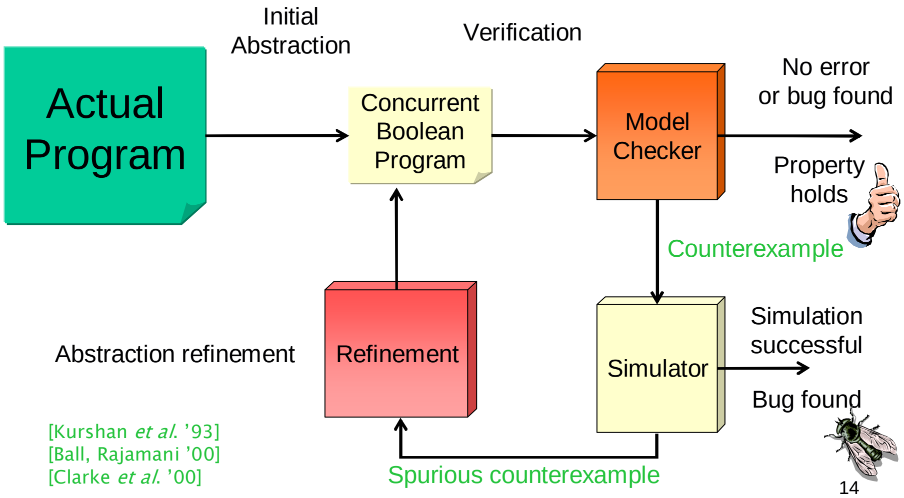
Counterexample-guided abstraction refinement framework (CEGAR)

[Clarke *et al*. ’00] – CMU

[Kurshan *et al*. ’93] – Bell Labs/Cadence

[Ball, Rajamani ’00] – Microsoft Research

CEGAR



[Kurshan *et al*. ’93]
[Ball, Rajamani ’00]
[Clarke *et al*. ’00]

Major Software Model Checkers

- ***FormalCheck/xUML*** **(UT Austin, Bell Labs)**
- ***ComFoRT*** **(CMU/SEI) built on top of *MAGIC* (CMU)**
- ***SPIN*** **(JPL/formely Bell Labs)**
- ***Verisoft*** **(Bell Labs)**
- ***Bandera*** **(Kansas State)**
- ***Java PathFinder*** **(NASA Ames)**
- ***SLAM/Bebop*** **(Microsoft Research)**
- ***BLAST*** **(Berkeley)**
- ***CBMC*** **(CMU)**

Class Presentations

SPIN: explicit state LTL model checker

ComFoRT: explicit state LTL and ACTL* model checker

SPIN: LTL Model Checking

- Properties are expressed in LTL
  - Subset of CTL* of the form:
    - A f

    where f is a path formula which does not contain any quantifiers
- The quantifier A is usually omitted
- G is substituted by □ (always)
- F is substituted by ◊ (eventually)
- X is (sometimes) substituted by ° (next)

LTL Formulae

AGFp in CTL*

AG AF p in CTL

- Always eventually p: $\Box \Diamond p$

- Always after p there is eventually q:
  $\Box (p \rightarrow (\Diamond q))$

AG(p→Fq) in CTL*

AG(p →AFq) in CTL

- Fairness:
  $(\Box \Diamond p) \rightarrow \varphi$

A((GF p) → φ) in CTL*

Can’t express it in CTL

LTL Model Checking

- An LTL formula defines a set of traces
- Check trace containment
  - Traces of the program must be a subset of the traces defined by the LTL formula
  - If a trace of the program is not in such set
    - It violates the property
    - It is a counterexample
  - LTL formulas are universally quantified

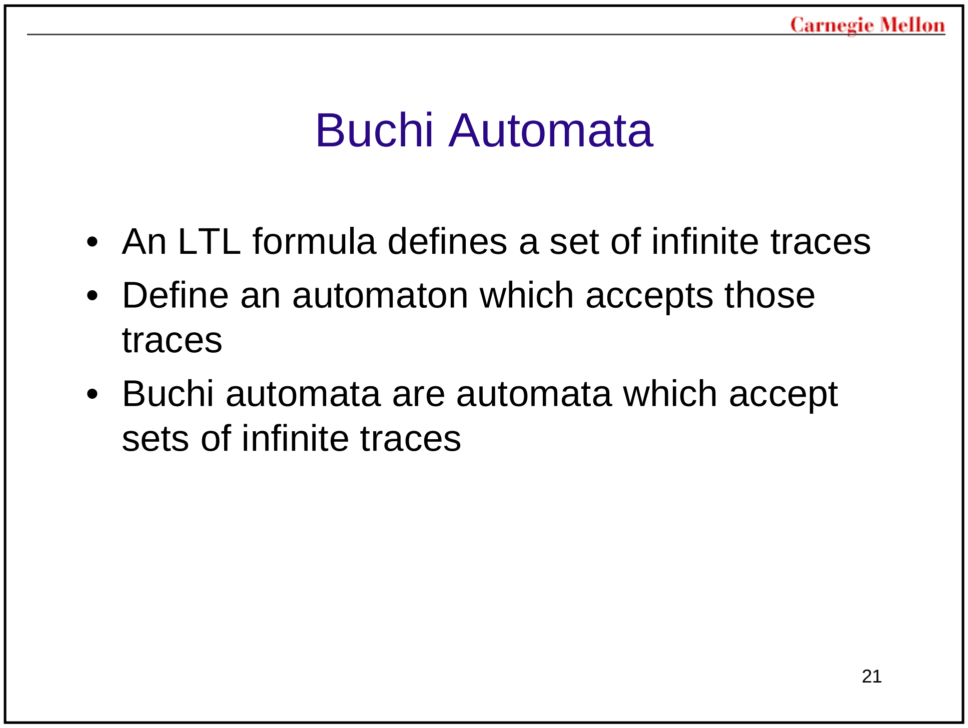
LTL Model Checking

- Trace containment can be turned into emptiness checking
  - Negate the formula corresponds to complement the defined set:

$$set(\phi) = \overline{set(\neg\phi)}$$

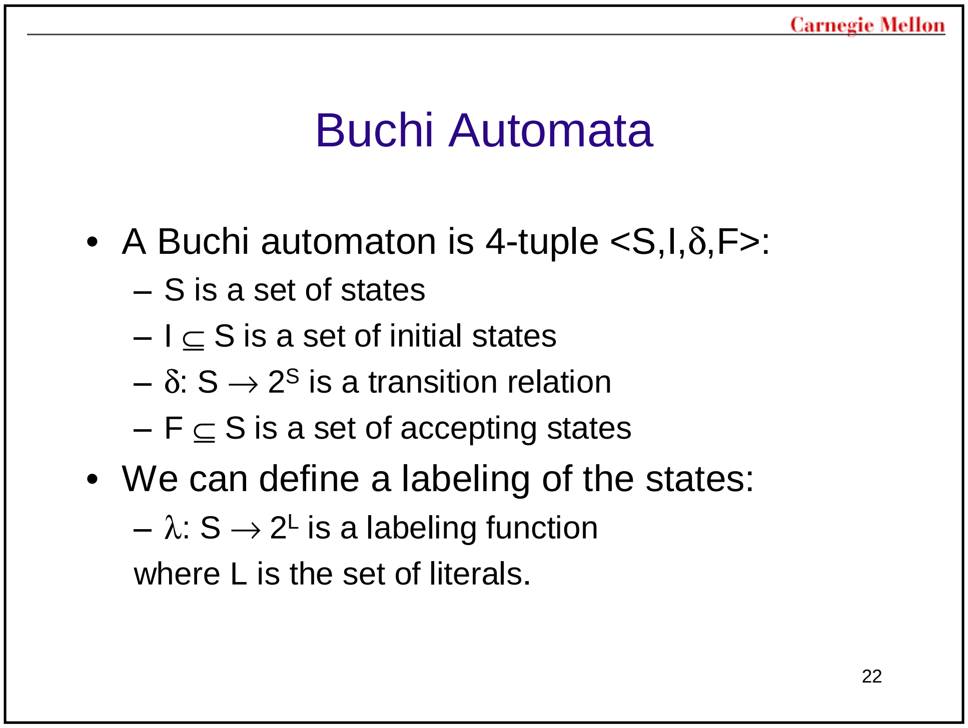
  - Subset corresponds to empty intersection:

$$A \subseteq B \Leftrightarrow A \cap \overline{B} = 0$$



Buchi Automata

- An LTL formula defines a set of infinite traces
- Define an automaton which accepts those traces
- Buchi automata are automata which accept sets of infinite traces



Buchi Automata

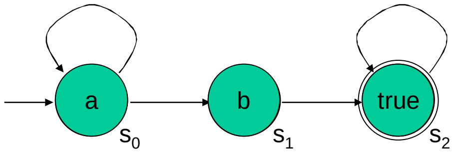
- A Buchi automaton is 4-tuple <S,I,δ,F>:
  - S is a set of states
  - $I \subseteq S$ is a set of initial states
  - δ: $S \rightarrow 2^S$ is a transition relation
  - $F \subseteq S$ is a set of accepting states
- We can define a labeling of the states:
  - λ: $S \rightarrow 2^L$ is a labeling function

  where L is the set of literals.

Buchi Automata

$S = \{ s_0, s_1, s_2 \}$

$I = \{ s_0 \}$



$\delta = \{ (s_0, \{s_0, s_1\}), (s_1, \{s_2\}), (s_2, \{s_2\}) \}$

$F = \{ s_2 \}$

$\lambda = \{ (s_0, \{a\}), (s_1, \{b\}), (s_2, \{\}) \}$

Buchi Automata

- An infinite trace $\sigma = s_0 s_1 \ldots$ is accepted by a Buchi automaton iff:
  - – $s_0 \in I$
  - – $\forall i \geq 0: s_{i+1} \in \delta(s_i)$
  - – $\forall i \geq 0: \exists j > i: s_j \in F$

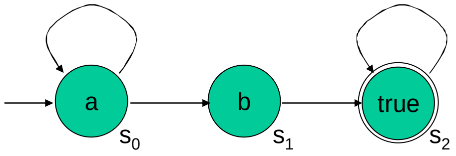
Buchi Automata

- Some properties:
  - Not all non-deterministic Buchi automata have an equivalent deterministic Buchi automata
  - Not all Buchi automata correspond to an LTL formula
  - Every LTL formula corresponds to a Buchi automaton
  - Set of Buchi automata closed under complemention, union, intersection, and composition

---

Buchi Automata

What LTL formula does this Buchi automaton corresponds to (if any)?



a U b

LTL Model Checking

- Generate a Buchi automaton for the negation of the LTL formula to check
- Compose the Buchi automaton with the automaton corresponding to the system
- Check emptiness

LTL Model Checking

- Composition:
  - At each step alternate transitions from the system and the Buchi automaton
- Emptiness:
  - To have an accepted trace:
    - There must be a cycle
    - The cycle must contain an accepting state

LTL Model Checking

- Cycle detection
  - Nested DFS
    - Start a second DFS
    - Match the start state in the second DFS
      - Cycle!
    - Second DFS needs to be started at each state?
      - Accepting states only will suffice
    - Each second DFS is independent
      - If started in post-order states need to be visited at most once in the second DFS searches

---

LTL Model Checking

```
procedure DFS(s)
  visited = visited  ∪ {s}
  for each successor s’ of s
    if s’ ∉ visited then
      DFS(s’)
      if s’ is accepting then
        DFS2(s’, s’)
      end if
    end if
  end for
end procedure
```

LTL Model Checking

```
procedure DFS2(s, seed)
  visited2 = visited2 ∪ {s}
  for each successor s’ of s
    if s’ = seed then
      return “Cycle Detect”;
    end if
    if s’ ∉ visited2 then
      DFS2(s’, seed)
    end if
  end for
end procedure
```

References

- http://spinroot.com/
- ***Design and Validation of Computer Protocols*** by Gerard Holzmann
- ***The Spin Model Checker*** by Gerard Holzmann
- ***An automata-theoretic approach to automatic program verification***, by Moshe Y. Vardi, and Pierre Wolper
- ***An analysis of bitstate hashing***, by G.J. Holzmann
- ***An Improvement in Formal Verification***, by G.J. Holzmann and D. Peled
- ***Simple on-the-fly automatic verification of linear temporal logic***, by Rob Gerth, Doron Peled, Moshe Vardi, and Pierre Wolper
- ***A Minimized automaton representation of reachable states***, by A. Puri and G.J. Holzmann

SPIN: The Promela Language

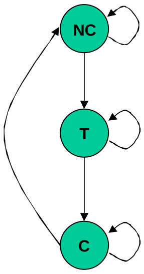
- Process Algebra
  - An algebraic approach to the study of concurrent processes. Its tools are algebraical languages for the specification of processes and the formulation of statements about them, together with calculi for the verification of these statements. [Van Glabbeek, 1987]
- Describes the system in a way similar to a programming language

Promela

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

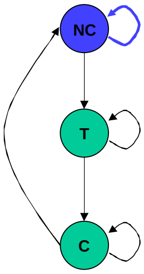
An Example

```
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); }
```



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;
   out ? value;
   assert(value == 0 || value == 1)
}
```

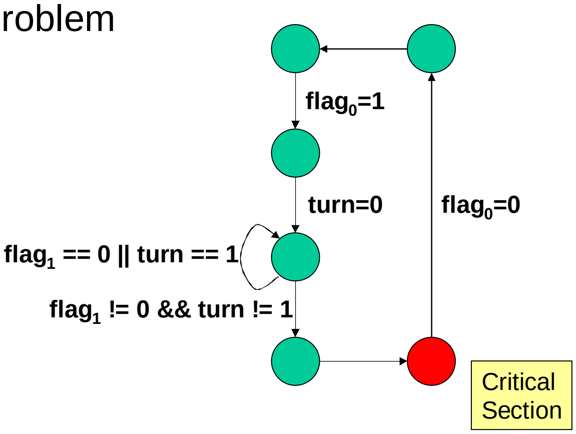
Other constructs

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

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

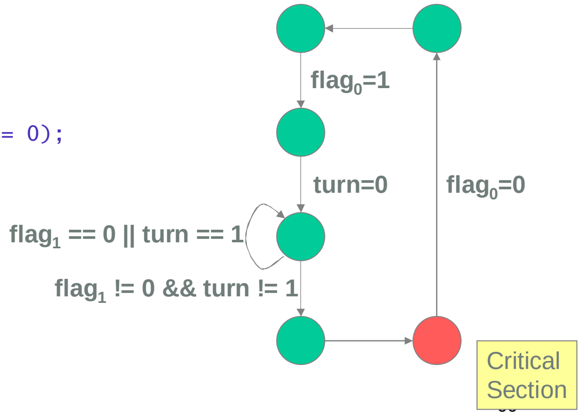
Mutual Exclusion

- Peterson’s solution to the mutual exclusion problem



Mutual Exclusion in SPIN

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



Mutual Exclusion in SPIN

```
bool turn, flag[2];

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

                      /* critical section */

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

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

Mutual Exclusion in SPIN

```
bool turn, flag[2];
byte ncrit;

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

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

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

LTL Properties:

```
[] (critial[0] || critical[1])

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

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