Formal Verification by Model Checking

Jonathan Aldrich
Carnegie Mellon University

Based on slides developed by Natasha Sharygina

17-654/17-754: Analysis of Software Artifacts
Spring 2006

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

- **AG**(Start ⇒ **AF** Heat)

Subformula Labeling

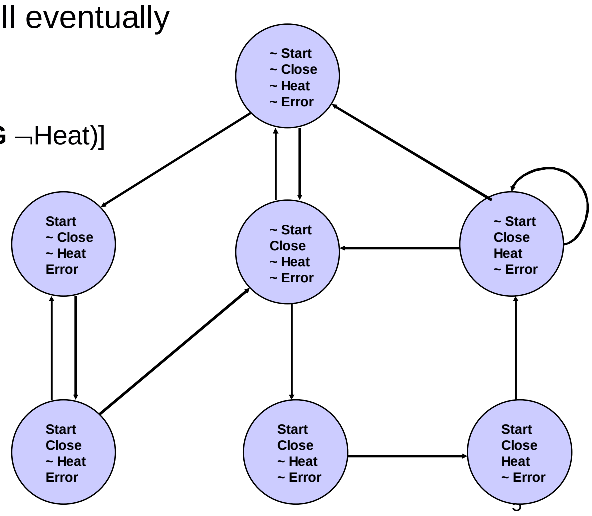
- Case $\neg f$
  - Label each state not labeled with f
- $f_1 \vee f_2$
  - Label each state which is labeled with either f_1 or f_2
- **EX** f
  - Label every state that has some successor labeled with f
- **E**$[f_1$ **U** $f_2]$
  - Label every state labeled with f_2
  - Traverse backwards from labeled 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

CTL Model Checking Example

- Pressing Start will eventually result in heat

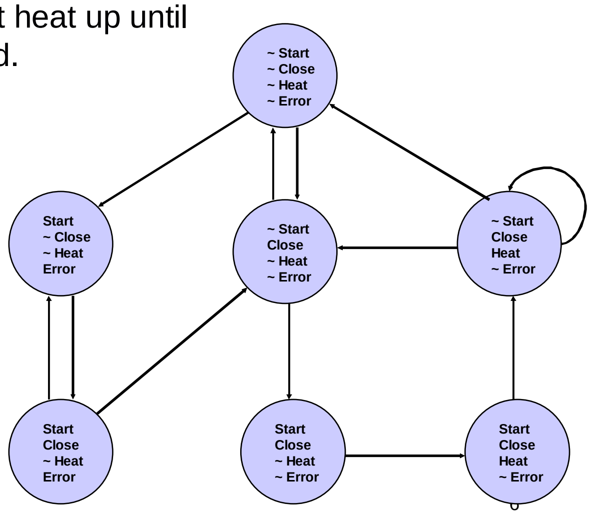
AG(Start $\Rightarrow$ **AF** Heat)

$= \neg$**E**[*true* **U** (Start $\wedge$ **EG** $\neg$Heat)]



CTL Model Checking Example

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



Practice Writing Properties

- If the door is locked, it will not open until someone unlocks it
- If you press ctrl-C, you will get a command line prompt
- The saw will not run unless the safety guard is engaged

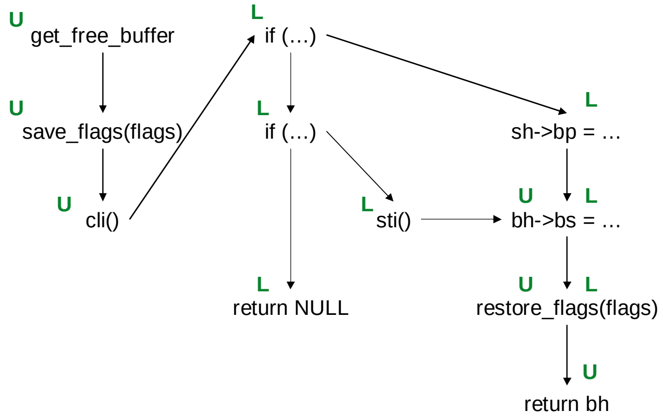
LTL Model Checking

- Beyond the scope of this course
- Canonical reference on Model Checking:
  - Edmund Clarke, Orna Grumberg, and Doron A. Peled. Model Checking. MIT Press, 1999.

Dataflow Analysis as Model Checking

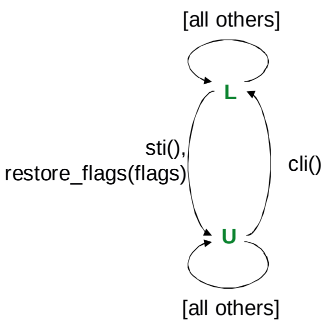
- Consider a lattice that is a tuple of sets:
  - **Var** → $2^{\mathbf{Set}}$
  - e.g. [x → { <, = }, y → { > }] where **Set** = { <, =, > }
- Represent the CFG as a Kripke structure
  - Let N be the nodes in the CFG, with initial node N_0
  - Let E be the edges in the CFG
- Consider the set of *abstract stores* L = $2^{\mathbf{Var \rightarrow Set}}$
  - Choose one element of lattice set for each var
    - e.g. [x → <, y → >]
- Strategy: instead of propagating around sets, see if each individual member of the lattice set can reach each node (may traverse each path multiple times)
  - This is exactly what Metal does!
  - Metal is essentially a model checker

Propagating Elements Instead of Sets



Dataflow Analysis as Model Checking

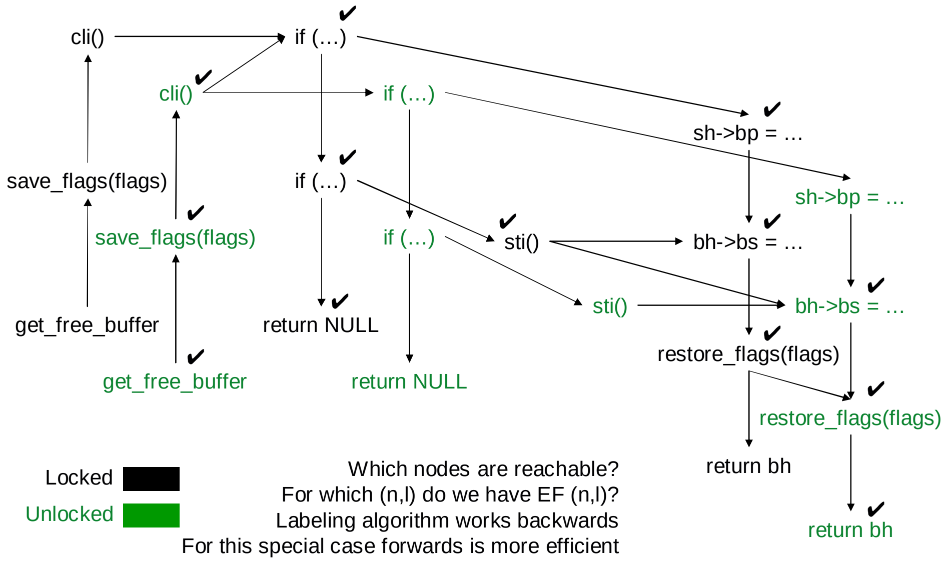
- Let $F = \{ l_1 \rightarrow_e l_2 \mid n_1 \rightarrow_e n_2 \in E \wedge l_2 \in f_{DF}(\{ l_1 \}, n_1) \}$
  - Represents flow functions
  - There's an edge from one lattice value to another, annotated with edge e from the program, iff when we apply the flow function for the source node of the program edge to the singleton set containing the first lattice value, the second lattice value is one of the results
  - We will assume edges are annotated with the source node



Dataflow Analysis as Model Checking

- Consider *synchronous product*
  - Cross product of nodes
    - $N_p = N*L$
  - Edges exist only when there is an edge in both source graphs with the same label
    - $E_p = \{ (n_1,l_1) \rightarrow_e (n_2,l_2) \mid n_1 \rightarrow_e n_2 \in E \wedge l_1 \rightarrow_e l_2 \in F \}$
    - Purpose: matches up edge from n_1 to n_2 (marked n_1) with edge representing flow function for n_1 (also marked n_1)
- Data flow is reachability in product graph
  - Flow(n) = { l | EF (n,l) }

Dataflow Analysis as Model Checking

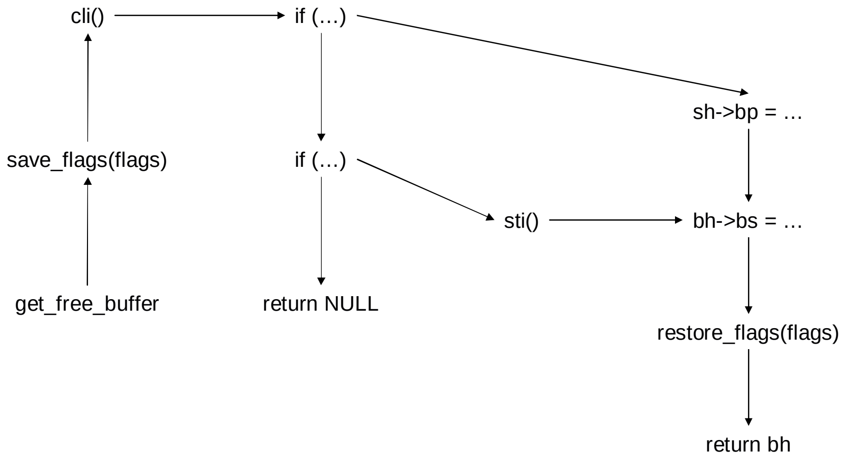


Which nodes are reachable?
For which (n,l) do we have EF (n,l)?
Labeling algorithm works backwards
For this special case forwards is more efficient

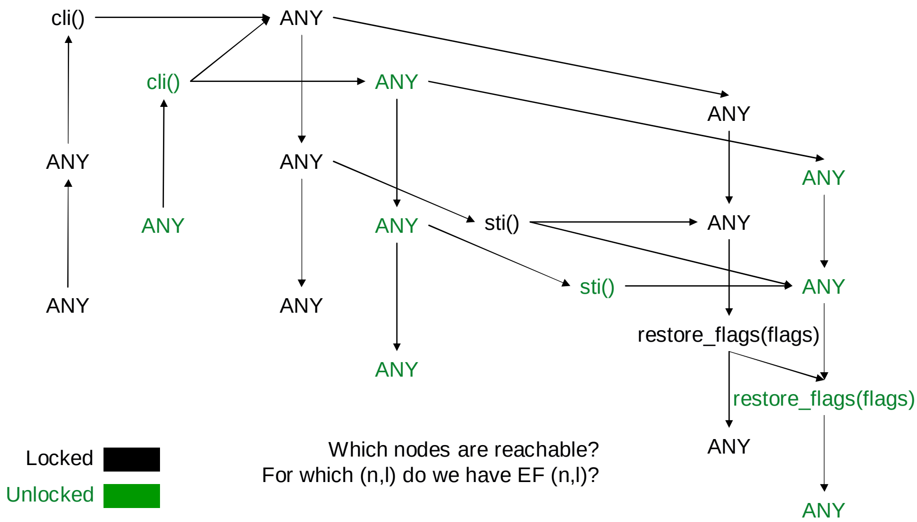
Abstraction

- Data flow values
  - We abstracted program values to two states: locked and unlocked
- Program
  - We represented the program directly as a CFG, without abstracting it
  - But really, we only care about 4 node types:
    - sti()
    - cli()
    - restore_flags(…)
    - anything else
  - Can we abstract the program to just these types of nodes?

Model Checking Abstract Graph



Model Checking Abstract Graph



Which nodes are reachable?
For which (n,l) do we have EF (n,l)?

Duality of Dataflow Analysis and Model Checking

- We've seen how dataflow analysis can be phrased as a model checking problem
  - Applies to all analyses that are tuples of sets
    - A more complex (and inefficient) construction still works if your lattice cannot be phrased as a tuple of sets
  - Benefit: can take advantage of model checking techniques
    - Symbolic representations (beyond scope of course)
    - Counterexample-guided abstraction refinement (CEGAR—next lecture)
  - Cost: explores each path for each set element
    - Unless you're using a symbolic representation or need CEGAR, dataflow analysis will be more efficient
- The converse is possible in some cases
  - Probably impossible for general LTL formulas
    - I have not actually seen impossibility results, but the model checking algorithm involves nested depth-first search which does not match dataflow analysis well

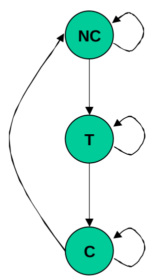
SPIN: The Promela Language

- 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); }
```



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); }
```

An Example

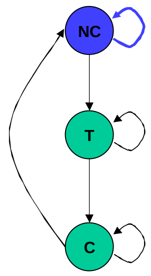
```
mtype = { NONCRITICAL, TRYING, CRITICAL };
show mtype state[2];
proctype process(int id) {
beginning:
noncritical:
    state[id] = NONCRITICAL;
    if
    :: goto noncritical;
    :: true;
    fi;
trying:
    state[id] = TRYING;
    if
    :: goto trying;
    :: true;
    fi;
critical:
    state[id]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

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(); }
```

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.

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)
}
```

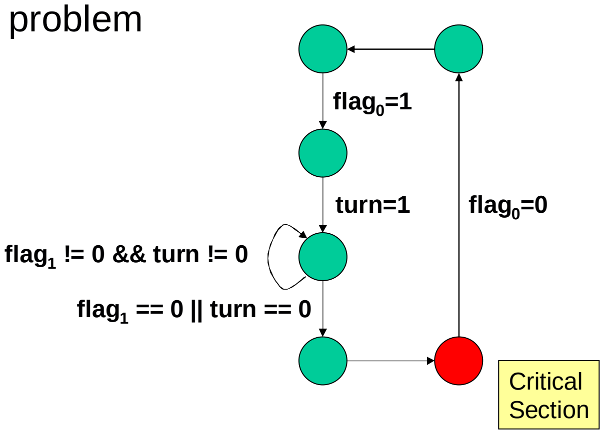

Other constructs

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

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

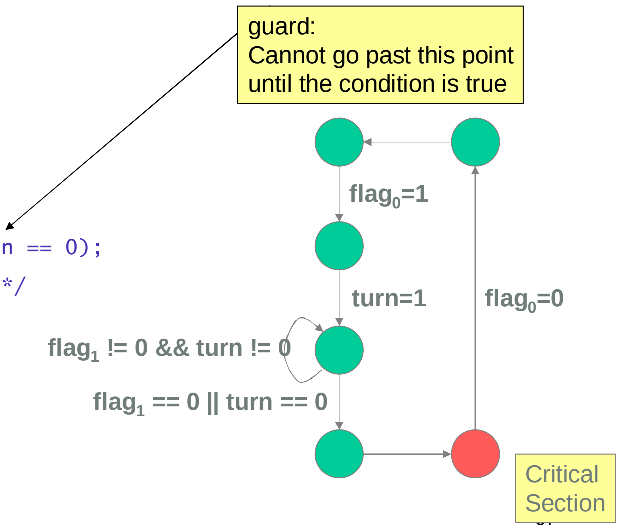
Mutual Exclusion

- 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;
}
```



Mutual Exclusion in SPIN

Active process:
automatically creates instances of processes

```
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;
}
```

_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 = 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

No matter what happens, a process will eventually get to a critical section

If process 0 is in the critical section, process 1 will get to be there next

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*)

$$M = M_1 \mid\mid \ldots \mid\mid M_n$$

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

Research Directions: State space reduction

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