Verification

Lecture 3

Bernd Finkbeiner

UNIVERSITÄT
DES
SAARLANDES

Plan for today

- CTL model checking
  - The basic algorithm
  - Fairness
  - Counterexamples and witnesses

Review: Computation tree logic

modal logic over infinite trees [Clarke & Emerson 1981]

- Statements over states
  - $a \in AP$ — atomic proposition
  - $\neg\,\Phi$ and $\Phi \wedge \Psi$ — negation and conjunction
  - E φ — there exists a path fulfilling φ
  - A φ — all paths fulfill φ
- Statements over paths
  - X Φ — the next state fulfills Φ
  - Φ U Ψ — Φ holds until a Ψ-state is reached

$\Rightarrow$ note that X and U alternate with A and E

- AX X Φ and A EX $\Phi \notin$ CTL, but AX AX Φ and AX EX $\Phi \in$ CTL

Alternative syntax: E $\approx \exists$, A $\approx \forall$, X $\approx \bigcirc$, G $\approx \Box$, F $\approx \Diamond$.

Review: Existential normal form (ENF)

The set of CTL formulas in existential normal form (ENF) is given by:

$$\Phi \;::=\; \text{true} \;\big|\; a \;\big|\; \Phi_1 \wedge \Phi_2 \;\big|\; \neg\Phi \;\big|\; \text{EX}\,\Phi \;\big|\; \text{E}\,(\Phi_1\,\text{U}\,\Phi_2) \;\big|\; \text{EG}\,\Phi$$

For each CTL formula, there exists an equivalent CTL formula in ENF

$$\text{AX}\,\Phi \;\equiv\; \neg\text{EX}\,\neg\Phi$$

$$\text{A}\,(\Phi\,\text{U}\,\Psi) \;\equiv\; \neg\text{E}\,(\neg\Psi\,\text{U}\,(\neg\Phi \wedge \neg\Psi)) \wedge \neg\text{EG}\,\neg\Psi$$

Review: Model checking CTL

- How to check whether state graph *TS* satisfies CTL formula $\widehat{\Phi}$?
  - convert the formula $\widehat{\Phi}$ into the equivalent Φ in ENF
  - compute recursively the set $Sat(\Phi) = \{ q \in S \mid q \models \Phi \}$
  - $TS \models \Phi$ if and only if each initial state of *TS* belongs to $Sat(\Phi)$
- Recursive bottom-up computation of $Sat(\Phi)$:
  - consider the parse-tree of Φ
  - start to compute $Sat(a_i)$, for all leaves in the tree
  - then go one level up in the tree and determine $Sat(\cdot)$ for these nodes

$$\text{e.g.,: } Sat(\underbrace{\Psi_1 \wedge \Psi_2}_{\text{node at level } i}) = Sat(\underbrace{\Psi_1}_{\substack{\text{node at} \\ \text{level } i-1}}) \cap Sat(\underbrace{\Psi_2}_{\substack{\text{node at} \\ \text{level } i-1}})$$

  - then go one level up and determine $Sat(\cdot)$ of these nodes
  - and so on..... until the root is treated, i.e., $Sat(\Phi)$ is computed

Basic algorithm

Require: finite transition system TS with states S and initial states I, and CTL formula Φ (both over AP)

Ensure: $TS \models \Phi$

{compute the sets $Sat(\Phi) = \{ q \in S \mid q \models \Phi \}$}
for all $i \leq |\Phi|$ **do**
  for all $\Psi \in Sub(\Phi)$ with $|\Psi| = i$ **do**
    compute $Sat(\Psi)$ from $Sat(\Psi')$ {for maximal proper $\Psi' \in Sub(\Psi)$}
  end for
end for
return $I \subseteq Sat(\Phi)$

Characterization of *Sat* (1)

For all CTL formulas Φ, Ψ over AP it holds:

$$
\begin{aligned}
Sat(\text{true}) &= S \\
Sat(a) &= \{\, q \in S \mid a \in L(q) \,\}, \text{ for any } a \in AP \\
Sat(\Phi \wedge \Psi) &= Sat(\Phi) \cap Sat(\Psi) \\
Sat(\neg \Phi) &= S \setminus Sat(\Phi) \\
Sat(\text{EX}\, \Phi) &= \{\, q \in S \mid Post(q) \cap Sat(\Phi) \neq \varnothing \,\}
\end{aligned}
$$

for a given finite transition system without terminal states

Characterization of *Sat* (2)

- $Sat(\mathrm{E}\,(\Phi\,\mathrm{U}\,\Psi))$ is the smallest subset T of S, such that:

  (1) $Sat(\Psi) \subseteq T$ and (2) $(q \in Sat(\Phi)$ and $Post(q) \cap T \neq \varnothing) \Rightarrow q \in T$

- $Sat(\mathrm{EG}\,\Phi)$ is the largest subset T of S, such that:

  (3) $T \subseteq Sat(\Phi)$ and (4) $q \in T$ implies $Post(q) \cap T \neq \varnothing$

Computing $Sat(\mathrm{E}(\Phi \,\mathrm{U}\, \Psi))$ (1)

- $Sat(\mathrm{E}(\Phi \,\mathrm{U}\, \Psi))$ is the smallest set $T \subseteq S$ such that:

  $(1)\ Sat(\Psi) \subseteq T \quad$ and $\quad (2)\ (q \in Sat(\Phi)$ and $Post(q) \cap T \neq \varnothing) \ \Rightarrow\ q \in T$

- This suggests to compute $Sat(\mathrm{E}(\Phi \,\mathrm{U}\, \Psi))$ iteratively:

  $$T_0 \ = \ Sat(\Psi) \quad \text{and} \quad T_{i+1} \ = \ T_i \ \cup \ \{\, q \in Sat(\Phi) \mid Post(q) \cap T_i \neq \varnothing \,\}$$

- T_i = states that can reach a Ψ-state in at most i steps via a Φ-path
- By induction on j it follows:

  $$T_0 \subseteq T_1 \subseteq \ldots \subseteq T_j \subseteq T_{j+1} \subseteq \ \ldots \ \subseteq \ Sat(\mathrm{E}(\Phi \,\mathrm{U}\, \Psi))$$

Computing $Sat(\mathrm{E}(\Phi \,\mathrm{U}\, \Psi))$ (2)

- *TS* is finite, so for some $j \geq 0$ we have: $T_j = T_{j+1} = T_{j+2} = \ldots$
- Therefore: $T_j = T_j \cup \{ q \in Sat(\Phi) \mid Post(q) \cap T_j \neq \varnothing \}$
- Hence: $\{ q \in Sat(\Phi) \mid Post(q) \cap T_j \neq \varnothing \} \subseteq T_j$
  - hence, T_j satisfies (2), i.e.,
    $(q \in Sat(\Phi)$ and $Post(q) \cap T_j \neq \varnothing) \Rightarrow q \in T_j$
  - further, $Sat(\Psi) = T_0 \subseteq T_j$ so, T_j satisfies (1), i.e. $Sat(\Psi) \subseteq T_j$
- As $Sat(\mathrm{E}(\Phi \,\mathrm{U}\, \Psi))$ is the <u>smallest</u> set satisfying (1) and (2):
  - $Sat(\mathrm{E}(\Phi \,\mathrm{U}\, \Psi)) \subseteq T_j$ and thus $Sat(\mathrm{E}(\Phi \,\mathrm{U}\, \Psi)) = T_j$
- Hence: $T_0 \subsetneq T_1 \subsetneq T_2 \subsetneq \ldots \subsetneq T_j = T_{j+1} = \ldots = Sat(\mathrm{E}(\Phi \,\mathrm{U}\, \Psi))$

Computing $Sat(\mathrm{E}(\Phi \,\mathrm{U}\, \Psi))$ (3)

Require: finite transition system with states S CTL-formula $\mathrm{E}(\Phi \,\mathrm{U}\, \Psi)$
Ensure: $Sat(\mathrm{E}(\Phi \,\mathrm{U}\, \Psi)) = \{ q \in S \mid q \models \mathrm{E}(\Phi \,\mathrm{U}\, \Psi) \}$

$V := Sat(\Psi)$; {V administers states q with $q \models \mathrm{E}(\Phi \,\mathrm{U}\, \Psi)$}
$T := V$; {T contains the already visited states q with $q \models \mathrm{E}(\Phi \,\mathrm{U}\, \Psi)$}
while $V \neq \varnothing$ **do**
  let $q' \in V$;
  $V := V \setminus \{ q' \}$;
  for all $q \in Pre(q')$ **do**
    if $q \in Sat(\Phi) \setminus T$ **then** $V := V \cup \{ q \}$; $T := T \cup \{ q \}$; **endif**
  end for
end while
return T

Computing $Sat(\mathrm{EG}\,\Phi)$

$V := S \setminus Sat(\Phi)$; {$V$ contains any not visited q' with $q' \not\models \mathrm{E}\,\mathrm{G}\,\Phi$}

$T := Sat(\Phi)$; {T contains any q for which $q \models \mathrm{E}\,\mathrm{G}\,\Phi$ has not yet been disproven}

for all $q \in Sat(\Phi)$ **do** $c[q] := |\,Post(q)\,|$; **od** {initialize array c}

while $V \neq \emptyset$ **do**
    {loop invariant: $c[q] = |\,Post(q) \cap (T \cup V)\,|$}
    let $q' \in V$; {$q' \not\models \Phi$}
    $V := V \setminus \{\, q' \,\}$; {$q'$ has been considered}
    for all $q \in Pre(q')$ **do**
        if $q \in T$ **then**
            $c[q] := c[q] - 1$; {update counter $c[q]$ for predecessor q of q'}
            if $c[q] = 0$ **then**
                $T := T \setminus \{\, q \,\}$; $V := V \cup \{\, q \,\}$; {q does not have any successor in T}
            end if
        end if
    end for
end while
return T

Alternative algorithm for $Sat(\mathrm{EG}\,\Phi)$

1. Consider only state q if $q \models \Phi$, otherwise eliminate q
   - change states to $S' = Sat(\Phi)$,
   - $\Rightarrow$ all removed states will not satisfy EG Φ, and thus can be safely removed
2. Determine all non-trivial strongly connected components in $TS[\Phi]$
   - non-trivial SCC = maximal, connected subgraph with at least one edge
   - $\Rightarrow$ any state in such SCC satisfies EG Φ
3. $q \models \mathrm{EG}\,\Phi$ is equivalent to "some SCC is reachable from q"
   - this search can be done in a backward manner

Complexity

For transition system TS with N states and M edges,
and CTL formula Φ, the CTL model-checking problem $TS \models \Phi$
can be solved in time $\mathcal{O}(|\Phi| \cdot (N + M))$

this applies to both algorithms for EG Φ

Fairness

Arbiter discussed yesterday

```
typedef enum A, B, C, X selection;
typedef enum IDLE, READY, BUSY controller_state;
typedef enum NO_REQ, REQ, HAVE_TOKEN client_state;

module main(clk);
    input clk;
    output ackA, ackB, ackC;
    selection wire sel;
    wire active;

    assign active = pass_tokenA || pass_tokenB || pass_tokenC;

    controller controllerA(clk, reqA, ackA, sel, pass_tokenA, A);
    controller controllerB(clk, reqB, ackB, sel, pass_tokenB, B);
    controller controllerC(clk, reqC, ackC, sel, pass_tokenC, C);
    arbiter arbiter(clk, sel, active);
    client clientA(clk, reqA, ackA);
    client clientB(clk, reqB, ackB);
    client clientC(clk, reqC, ackC);
endmodule
```

Model checking (1)

- Mutual exclusion: No two different acks are given at the same time.

```
AG ( !(ackA=1 * ackB=1 + ackB=1 * ackC=1 + ackC=1
* ackA=1) );
```

```
vis> read_verilog arbiter.v
vis> init_verify
vis> model_check arbiter.ctl

# MC: formula passed -- AG(!((((ackA=1 * ackB=1) +
(ackB=1 * ackC=1)) + (ackC=1 * ackA=1))))
```

Model checking (2)

- Responsiveness: Every request is eventually followed by an ack

```
AG( (reqA = 1) -> AF(ackA = 1) );
AG( (reqB = 1) -> AF(ackB = 1) );
AG( (reqC = 1) -> AF(ackC = 1) );
```

```
vis> read_verilog arbiter.v
vis> init_verify
vis> model_check arbiter.ctl

# MC: formula passed -- AG(!((((ackA=1 * ackB=1) +
(ackB=1 * ackC=1)) + (ackC=1 * ackA=1))))
# MC: formula failed -- AG((reqA=1 -> AF(ackA=1)))
# MC: formula failed -- AG((reqB=1 -> AF(ackB=1)))
# MC: formula failed -- AG((reqC=1 -> AF(ackC=1)))
```

```
module client(clk, req, ack);
    input clk, ack;
    output req;

    reg req;
    client_state reg state;

    wire rand_choice;

    initial req = 0;
    initial state = NO_REQ;

    assign rand_choice = $ND(0,1);

    always @(posedge clk) begin
            case(state)
                NO_REQ:
                    if (rand_choice)
                    begin
                        req = 1;
                        state = REQ;
                    end
```

```
            REQ:
               if (ack) state = HAVE_TOKEN;
            HAVE_TOKEN:
               if (rand_choice)
               begin
                  req = 0;
                  state = NO_REQ;
               end
         endcase
   end
endmodule
```

Fairness constraints

Fairness: We are only interested in paths where the clients release the token infinitely often.

arbiter.fair:

```
!(clientA.state=HAVE_TOKEN);
!(clientB.state=HAVE_TOKEN);
!(clientC.state=HAVE_TOKEN);
```

```
vis> read_fairness arbiter.fair
vis> model_check arbiter.ctl
# MC: formula passed -- AG(!((((ackA=1 * ackB=1) +
(ackB=1 * ackC=1)) + (ackC=1 * ackA=1))))
# MC: formula passed -- AG((reqA=1 -> AF(ackA=1)))
# MC: formula passed -- AG((reqB=1 -> AF(ackB=1)))
# MC: formula passed -- AG((reqC=1 -> AF(ackC=1)))
```