G52CON:
Concepts of Concurrency

Lecture 18 Model Checking II

Brian Logan
School of Computer Science
bsl@cs.nott.ac.uk

Outline of this lecture

- expressing properties in CTL
- example: expressing properties of Peterson's algorithm
- a simple model checking algorithm
- Exercise 6: CTL

Model-based approaches to verification

In a model-based approach

- the system is represented by a finite model M for an appropriate logic;
- the specification is a formula ϕ in the same logic; and
- the verification method consists of computing whether M satisfies ϕ ($M \models \phi$)

This process can be *automated* (model checking).

Model checking and temporal logic

Model checking is based on *temporal logic*

- in classical (propositional) logic, a model is an assignment of truth values to atomic propositions
- the models of temporal logic contain several states and a formula can be true in some states and false in others
- truth is *dynamic* in that formulas can change their truth values as the system evolves from state to state

In model checking, the models are *transition systems* and the properties ϕ are formulas of temporal logic

Syntax of CTL

CTL is a branching-time temporal logic

- a set of atomic propositions p, q, r, …
- standard logical connectives: $\neg$, $\wedge$, $\vee$, $\rightarrow$
- temporal connectives: AX, EX, AF, EF, AG, EG, AU and EU
- formulas: $\phi = p \mid \neg\phi \mid \phi_1 \wedge \phi_2$ ... AX ϕ ... A[ϕ U φ] ...

Temporal connectives

- AX ϕ : on **A**ll paths, ϕ is true in the ne**X**t state
- EX ϕ : on som**E** path, ϕ is true in the ne**X**t state

- AF ϕ : on **A**ll paths, in some **F**uture state ϕ is true
- EF ϕ : on som**E** path, in some **F**uture state ϕ is true

- AG ϕ : on **A**ll paths, in all future states (**G**lobally) ϕ is true
- EG ϕ : on som**E** path, in all future states (**G**lobally) ϕ is true

- A[ϕ U φ] : on **A**ll paths, ϕ is true **U**ntil φ is true
- E[ϕ U φ] : on som**E** path, ϕ is true **U**ntil φ is true

Specifying properties of systems

Given some atomic propositions expressing properties of interest such as *ready*, *started*, *requested*, *acknowledged*, *enabled*, *deadlock* etc., we can express properties such as:

- there exits some state where *started* holds, but *ready* does not:

$$\mathrm{EF}\ (started \wedge \neg ready)$$

- a request for a resource will eventually be acknowledged:

$$\mathrm{AG}(requested \rightarrow \mathrm{AF}\ acknowledged)$$

- a process will eventually be permanently deadlocked:

$$\mathrm{AF}(\mathrm{AG}\ deadlock)$$

- from any state it is possible to get to a restart state:

$$\mathrm{AG}(\mathrm{AF}\ restart)$$

Example: Peterson's algorithm

```
// Process 1
init1;
while(true) {
    // entry protocol
    c1 = true;
    turn = 2;
    while (c2 && turn == 2) {};
    crit1;
    // exit protocol
    c1 = false;
    rem1;
}
```

```
// Process 2
init2;
while(true) {
    // entry protocol
    c2 = true;
    turn = 1;
    while (c1 && turn == 1) {};
    crit2;
    // exit protocol
    c2 = false;
    rem2;
}
```

```
// shared variables
bool c1 = c2 = false;
integer turn == 1;
```

Example: Peterson's algorithm 1

Atomic propositions:

p_1 true when `c1 == true`

q_2 true when `turn == 2`

s_1 true when process 1 is spinning in its entry protocol

c_1 true when process 1 is in its critical section

r_1 true when process 1 is in its remainder

p_2 true when `c2 == true`

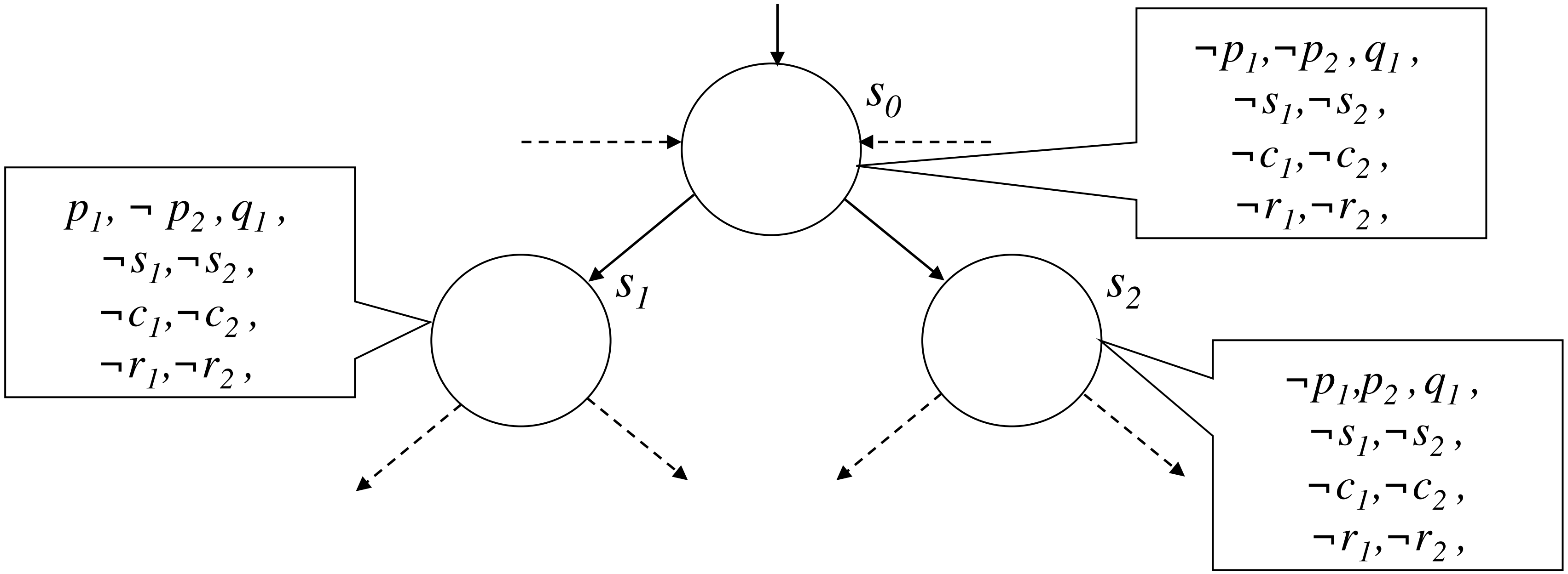
q_1 true when `turn == 1`

s_2 true when process 2 is spinning in its entry protocol

c_2 true when process 2 is in its critical section

r_2 true when process 2 is in its remainder

Example: Peterson's algorithm 2



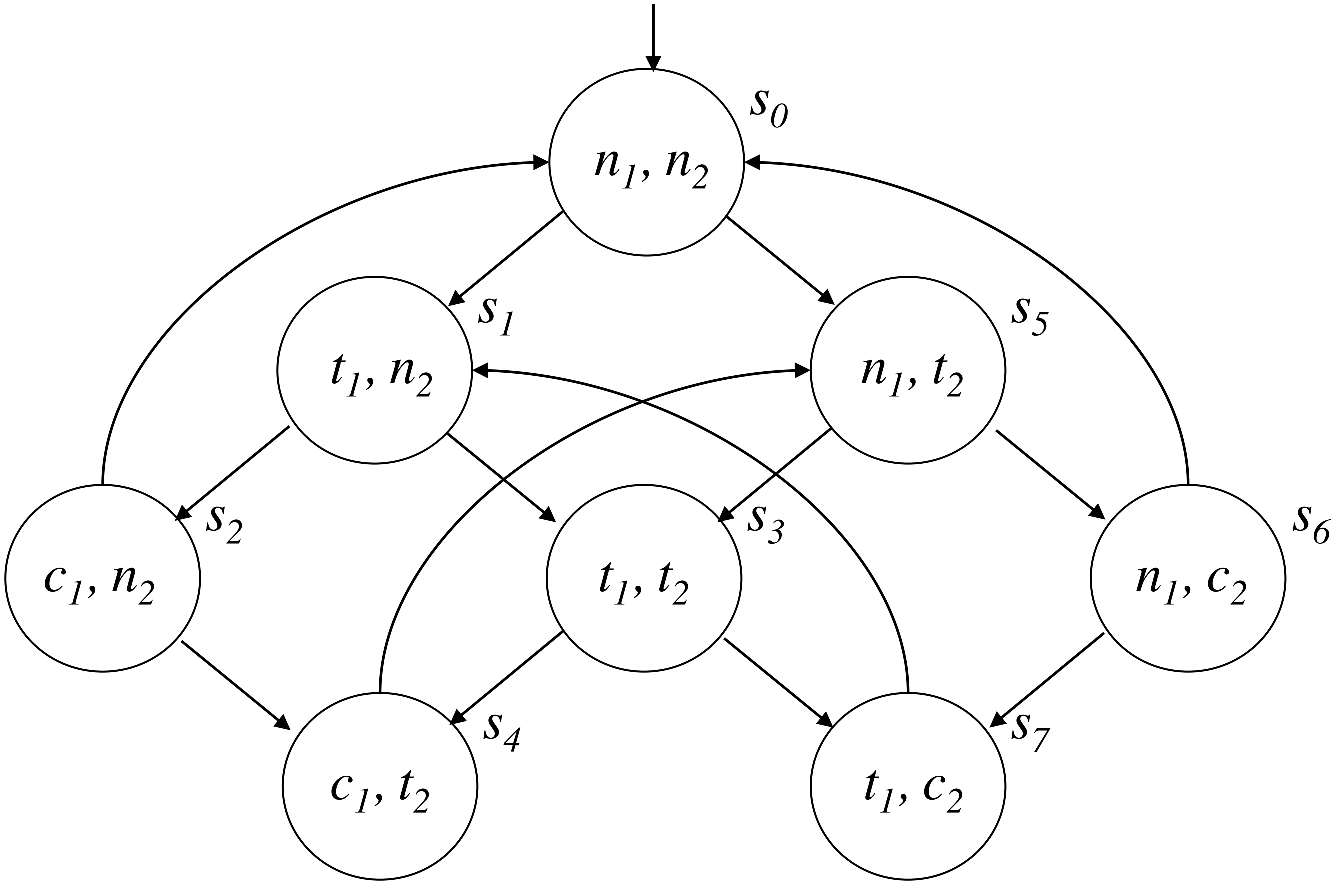
Example: Peterson's algorithm 3

Different abstractions are possible, for example:

- n_i (process i is not in its critical section or trying to enter, i.e., it is initialising or in the remainder)
- t_i (process i is trying to enter its critical section)
- c_i (process i is in its critical section)
- each process undergoes transitions in the cycle $n_i \rightarrow t_i \rightarrow c_i \rightarrow n_i \ldots$
- only one process can make a transition at a time (e.g., a single processor and the transitions are atomic)
- the two processes start off not in their critical sections, in the initial state s_0

Note that this loses information and is *not* a faithful model of Peterson's algorithm

Example: Peterson’s algorithm 4



Note: in each state only those propositions which are true are shown

Question 1: safety & liveness properties

- Express in CTL the following properties for Peterson's algorithm:
  - Mutual Exclusion
  - Absence of Unnecessary Delay
  - Eventual Entry

Question 1: safety & liveness properties

- Mutual Exclusion:

  $AG\ \neg(c_1 \wedge c_2)$

- Absence of Unnecessary Delay:

  $AG\ (t_1 \wedge n_2 \rightarrow AX\ (\neg\ t_2 \rightarrow c_1))$ for process 1

  $AG\ (t_2 \wedge n_1 \rightarrow AX\ (\neg\ t_1 \rightarrow c_2))$ for process 2

- Eventual Entry:

  $AG\ (t_1 \rightarrow AF\ c_1)$ for process 1

  $AG\ (t_2 \rightarrow AF\ c_2)$ for process 2

Question 1: safety & liveness properties

- Mutual Exclusion:

$$AG \neg(c_1 \wedge c_2)$$

in all states on all paths from s_0, $c_1 \wedge c_2$ is false, i.e., process 1 and process 2 are not in their critical sections at the same time

Question 1: safety & liveness properties

- Absence of Unnecessary Delay:

  $AG\ (t_1 \wedge n_2 \rightarrow AX\ (\neg\ t_2 \rightarrow c_1))$ for process 1

  in all states on all paths from s_0, if process 1 is trying to enter its critical section (t_1) and process 2 is not in its critical section or trying to enter (n_2), in that state then …

  in the next state on all paths from that state, if process 2 is not trying to enter its critical section ($\neg\ t_2$), i.e., it hasn't started trying at this transition, then process 1 will enter its critical section (c_1)

Question 1: safety & liveness properties

- Eventual Entry:

  AG ($t_1 \rightarrow$ AF c_1) for process 1

  in all states on all paths from s_0, if process 1 is trying to enter its critical section in that state (t_1), then …

  in some future state on all paths from that state, process 1 will enter its critical section (c_1)

  Note that this formula is ***false*** in our model of Peterson's algorithm, as we have abstracted away the `turn` variable

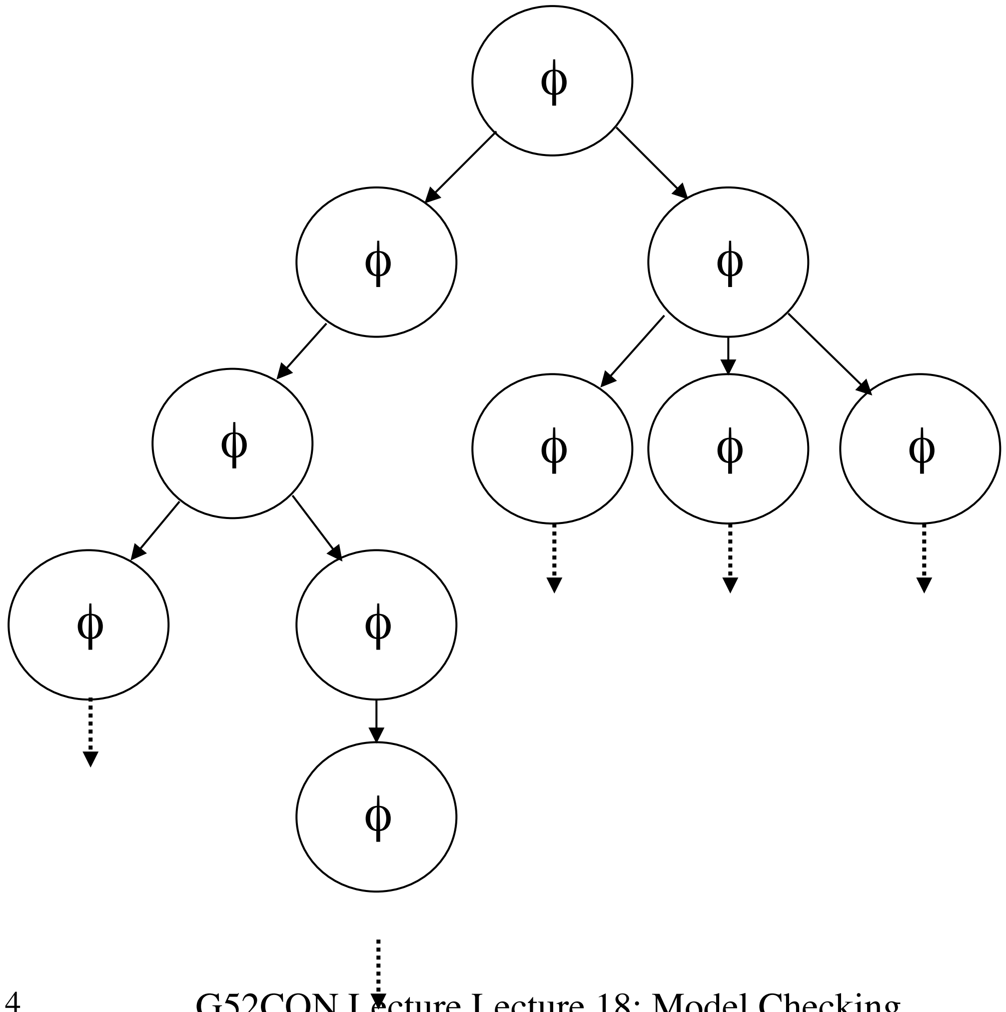
Question 2: CTL truth definitions

- Using CTL truth definitions show that the formula expressing Absence of Unnecessary Delay:

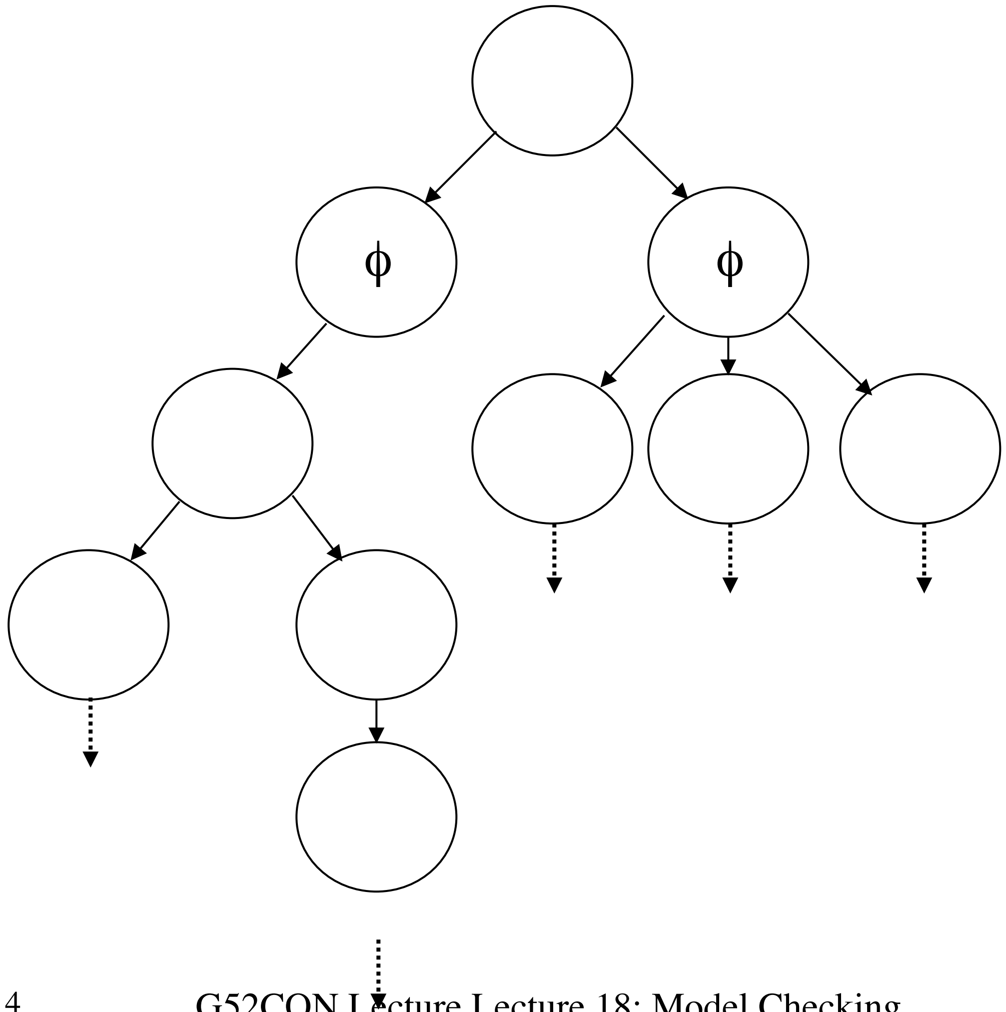
$$AG\ (t_1 \wedge n_2 \rightarrow AX\ (\neg\ t_2 \rightarrow c_1))$$

is true in the state s_0

Example: a system which satisfies AG ϕ



Example: a system where s_0 satisfies AX ϕ



Question 2: CTL truth definitions

- for

$$AG\ (t_1 \wedge n_2 \rightarrow AX\ (\neg\ t_2 \rightarrow c_1))$$

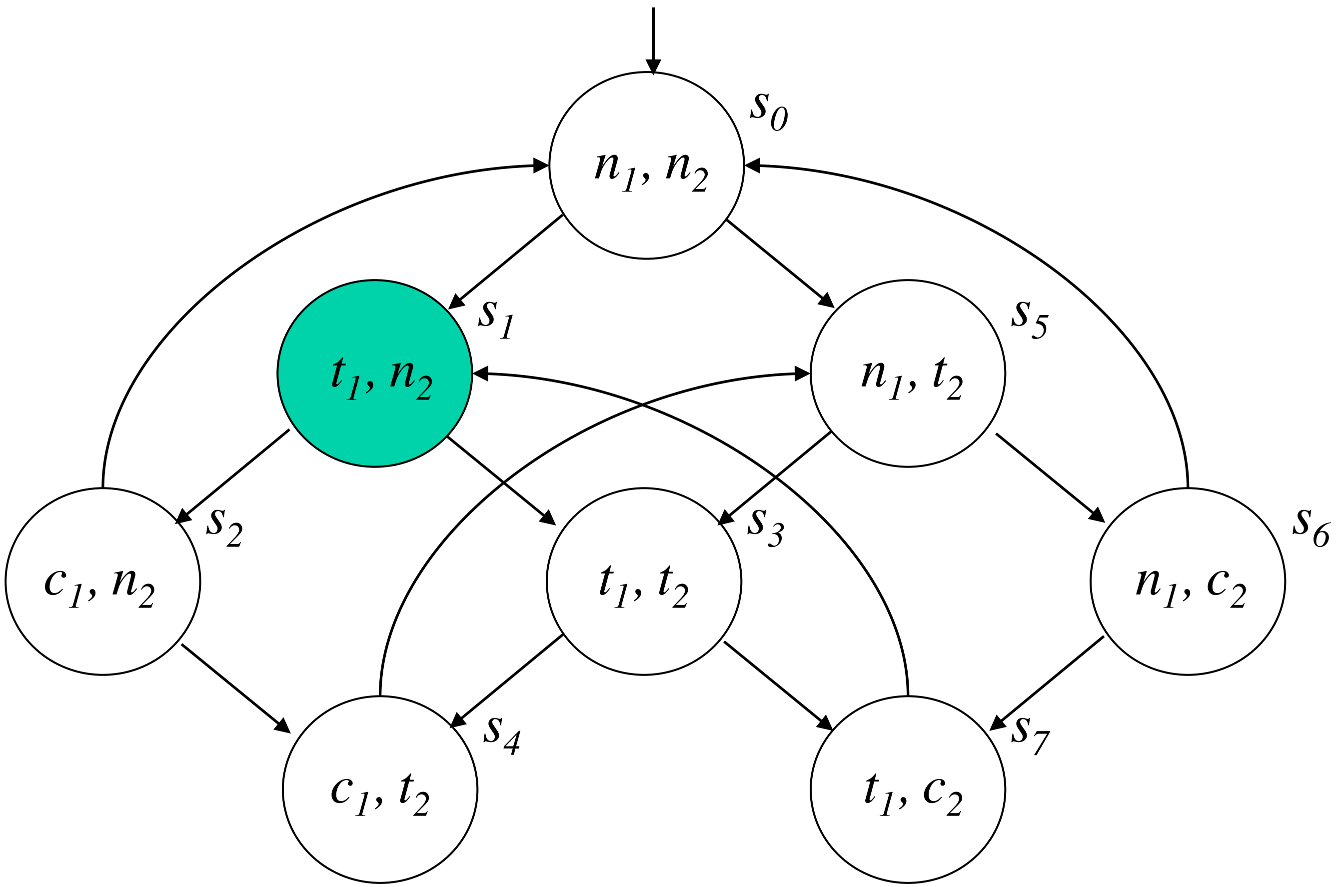
  to be true in the state s_0 then on all paths from s_0 if $t_1 \wedge n_2$ is true in a state, then AX ($\neg\ t_2 \rightarrow c_1$) must also be true in that state

- for

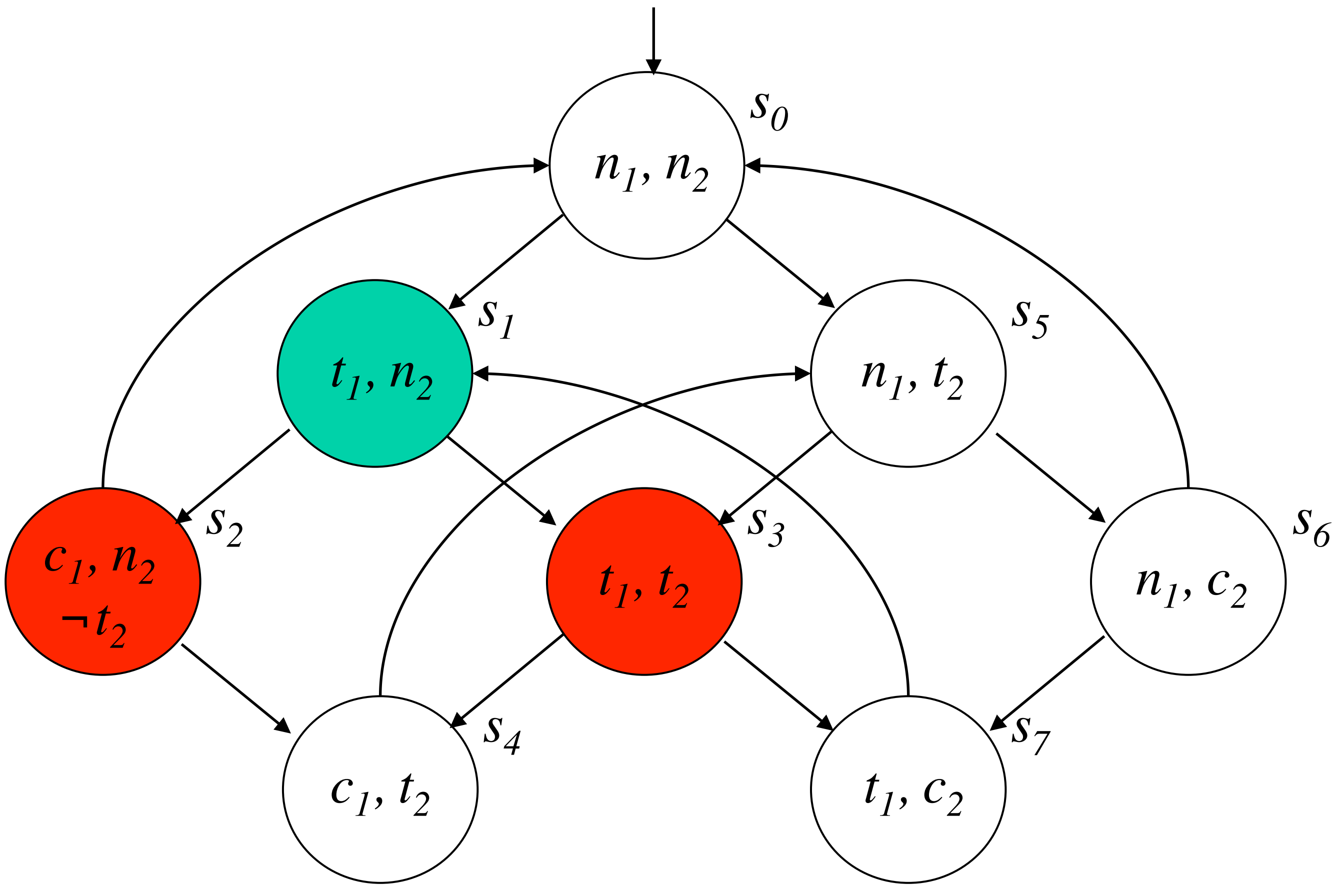
$$AX\ (\neg\ t_2 \rightarrow c_1)$$

  to be true in a state s_i, then must $\neg\ t_2 \rightarrow c_1$ be true in all states s_j reachable from s_i in one step.

Question 2: CTL truth definitions



Question 2: CTL truth definitions



Question 2: CTL truth definitions

- in s_2, t_2 is false and c_1 is true, so $\neg t_2 \rightarrow c_1$ is true
- in s_3, t_2 is true so $\neg t_2 \rightarrow c_1$ is true
- so in s_1

  $$AX\ (\neg t_2 \rightarrow c_1)$$

  is true, and $t_1 \wedge n_2 \rightarrow AX\ (\neg t_2 \rightarrow c_1)$ is also true

Question 2: CTL truth definitions

- in all other states $t_1 \wedge n_2$ is false, so $t_1 \wedge n_2 \rightarrow \text{AX}\,(\neg\, t_2 \rightarrow c_1)$ is true
- as $t_1 \wedge n_2 \rightarrow \text{AX}\,(\neg\, t_2 \rightarrow c_1)$ is true in all states

$$\text{AG}\,(t_1 \wedge n_2 \rightarrow \text{AX}\,(\neg\, t_2 \rightarrow c_1))$$

is true

Verifying properties by model checking

To verify that a program or system satisfies a property, we:

- describe the system using the description language of the model-checker;
- express the property to be verified using the specification language of the model checker; and
- run the model checker with the system description and property to be verified as inputs.

How it works

When the model checker is run

- it generates a model (transition system), M, from the system description;
- converts the property to be verified into a temporal logic formula ϕ and;
- for every state *s* in M, checks whether *s* satisfies ϕ (M, $s \models \phi$)

If the model doesn't satisfy the formula most model checkers also output a trace of the system behaviour that causes the failure.

A model checking algorithm

The simplest algorithm is as follows:

- given a transition system S and a formula ϕ to check

  1. generate the set of subformulas of ϕ; order them by complexity (propositional variables first, then negations of propositional variables, then conjunctions …, ϕ last)

  2. take a subformula ψ from the list and label those states of S which satisfy ψ with ψ

  3. repeat step 2 until all subformulas have been processed

- when we reach the end of the list we see which states satisfy ϕ.

A model checking algorithm 2

To label states of S with subformulas that don't contain CTL connectives:

- since states come with a labelling function, we know how to label states with atomic propositions;

- if current subformula is $\neg\psi$, we label with $\neg\psi$ those states which are not labelled with ψ (note that ψ precedes $\neg\psi$ in the list of subformulas, so we have already labelled the states with ψ);

- if the current subformula is $\psi_1 \wedge \psi_2$, we label those states which are labelled with ψ_1 and ψ_2 with $\psi_1 \wedge \psi_2$.

All other boolean connectives can be expressed in terms of $\neg$ and $\wedge$.

A model checking algorithm 3

To label states with subformulas containing the connectives EX, EU and AF:

- if ϕ is EX ψ, label predecessors of any state labelled ψ by EX ψ;
- if ϕ is E[ψ_1 U ψ_2], first find all states labelled ψ_2 . Then work backwards from those states and so long as we encounter ψ_1 states we label them by E[ψ_1 U ψ_2];
- if ϕ is AFψ, first label all states labelled with ψ with AFψ. Then label a state with AFψ if all its successor states are labelled with AFψ. Repeat until there is no change.

All the other CTL connectives can be expressed in terms of EX, EU and AF.

Overcoming the state explosion problem

- using efficient data structures, called ordered binary decision diagrams, which represent sets of states rather than individual states
- abstracting away variables in the model which are not relevant to the formula being checked
- partial order reduction—for asynchronous systems, several interleavings of component traces may be equivalent as far as satisfaction of the formula to be checked is concerned
- induction—model checking systems with large numbers of identical or similar components can of be implemented by induction on that number
- composition —breaking the verification problem down into several simpler verification problems.

Exercise 6: CTL

```
// Process 1

while(true) {
    r1 = turn;
    if (!r1) {
        <crit1>;
        turn = true;
    }
}
```

```
// Process 2

while(true) {
    r2 = turn;
    if (r2) {
        <crit1>;
        turn = false;
    }
}
```

```
// Shared datastructures
boolean turn = r1 = r2 = false;
```

The next lecture

Revision?