COMP 5404 Computer Aided Verification

- **Instructor:** Doug Howe, HP5360, *howe@scs.carleton.ca*
- **Prerequisite:** general CS background. See instructor if in doubt.
- **Text:** *Logic in Computer Science: Modeling and Reasoning about Systems*, Huth and Ryan.
- **Grading:** 60% assignments (4 or 5), 40% final exam. Final exam will be held during the last lecture.
- **Office Hours:** Mondays 10-12.

Course theme

Software verification using *formal methods*.

Testing

- Edsgar Dijkstra (Turing Award winner):
  "*Testing can never show the absence of bugs, only their presence.*"
- Testing can increase confidence in a program, but not enough for critical applications:
  - Nuclear reactors
  - Avionics
  - Medicine
  - Finance
  - Circuits
  - Network protocol correctness and *security.*
- There have been some extremely costly disasters (Intel, Ariane…)

A "hi-tech" solution

Devise tools that verify, with 100% certainty, that a program satisfies some specification of correctness.

Tool input: program + specification.

Tool output: Yes/no. If "no", may also output error trace.

Form of tool input?

- Tool inputs must be precise.
- Both inputs are expressions in some language:
  1. a programming language
  2. a specification language.
- Programming languages are precise:
  - syntax: what expressions are acceptable
    - specified by grammar or parser.
  - semantics: what do the expressions mean
    - specified by manual or compiler.

Formal languages

- A *formal language* is a language with a mathematically precise syntax and semantics.
- All programming languages are formal languages.
- English (French, Chinese, etc) is not a formal language.
- Mathematics is not a formal language! (It uses English.)
- To make specifications acceptable for tool input, write them in a *formal specification language*.

Formal verification tools

Input: program + formal specification*

Output: yes/no (+ possibly an error trace)

(*) A specification in a formal language.

<u>Unique property of formal verification</u>

Since the inputs are have precise mathematical meaning, the question

Does the program meet the specification?

has a precise answer (as opposed to testing, which answers *maybe).* The tool gives the answer.

Note: there may be bugs in the tool itself, the compiler used to compile the tool, the operating system running the tool, the hardware running the operating system.

<u>Formal specification languages: example</u>

ZFC (Zermelo-Fraenkel set theory with Choice)

- Sufficient for most mathematics.
- Based on predicate calculus.
- Examples:
  - $\forall x\ \exists y\ \ x \in y$
  - $\forall x\ \forall y\ \exists z\ \forall w\ w \in z \Leftrightarrow (w \in x \wedge w \in y)$
  - $\forall n\ \ 3 \leq n\ \Rightarrow\ \neg(\exists x\ \exists y\ \exists z\ \ x \neq 0 \wedge y \neq 0 \wedge z \neq 0 \wedge x^n + y^n = z^n)$
- ZFC syntax: expression built using above symbols (etc).
- ZFC semantics: “first-order structures”.

Formal tool built on ZFC

Input: program + ZFC specification

Output: yes/no

- Big problem: *we can mathematically prove that there is no algorithm for this!*
- **Solution 1:** request user input if needed.
- **Solution 2:** restrict the inputs.

Solution 1: deductive methods

- Require the user to construct a formal *proof* that the program meets the specification.
- Tool provides support:
  - automatic proof of "easy" facts such as
    - basic arithmetic
    - basic theories of data structures (lists, arrays etc)
    - type checking
  - proof by analogy
  - lemma database
- *General, but can require a great deal of direction by user.*

Solution 2: algorithmic

- Find suitable input languages (programming, specification) for which there are efficient algorithms.
- User involvement:
  - produce the inputs
  - set some parameters in the tool for efficiency
- "Push-button" technology.
- Emphasis on finding error traces in case of "no" answer.
  - A simple "no" is not so useful in practice.
- Typical algorithm is a *model checker:* check that spec holds in a model of the program.

Caveat

- Many of the best approaches to Solution 2 are restricted to *finite state systems*.
- Many interesting systems are finite state (e.g. digital circuits).
- Also, many infinite-state systems can be viewed as finite state by *abstracting away* from irrelevant details. E.g.
  - in network protocols, can ignore packet payload
  - in security protocols, can ignore encryption and message details
  - in avionics, can ignore sensor details.

First main topic: CTL model checking

- CTL = “Computation Tree Logic”, a spec language.
- There are many possible programming languages. We’ll use a particular model they can all compile into.

CTL model checker input:

“Kripke structure” model of program

+

CTL specification

Output: yes/no, + if no, then error trace (in many cases)

Preliminaries

Before studying CTL model checking, need to understand:

- State transition systems.
- Kripke structures.
- Computation paths and computation trees.
- The CTL formalism.
- How to translate informal specifications into CTL.
- CTL formula equivalencies (to simplify model checker).

States

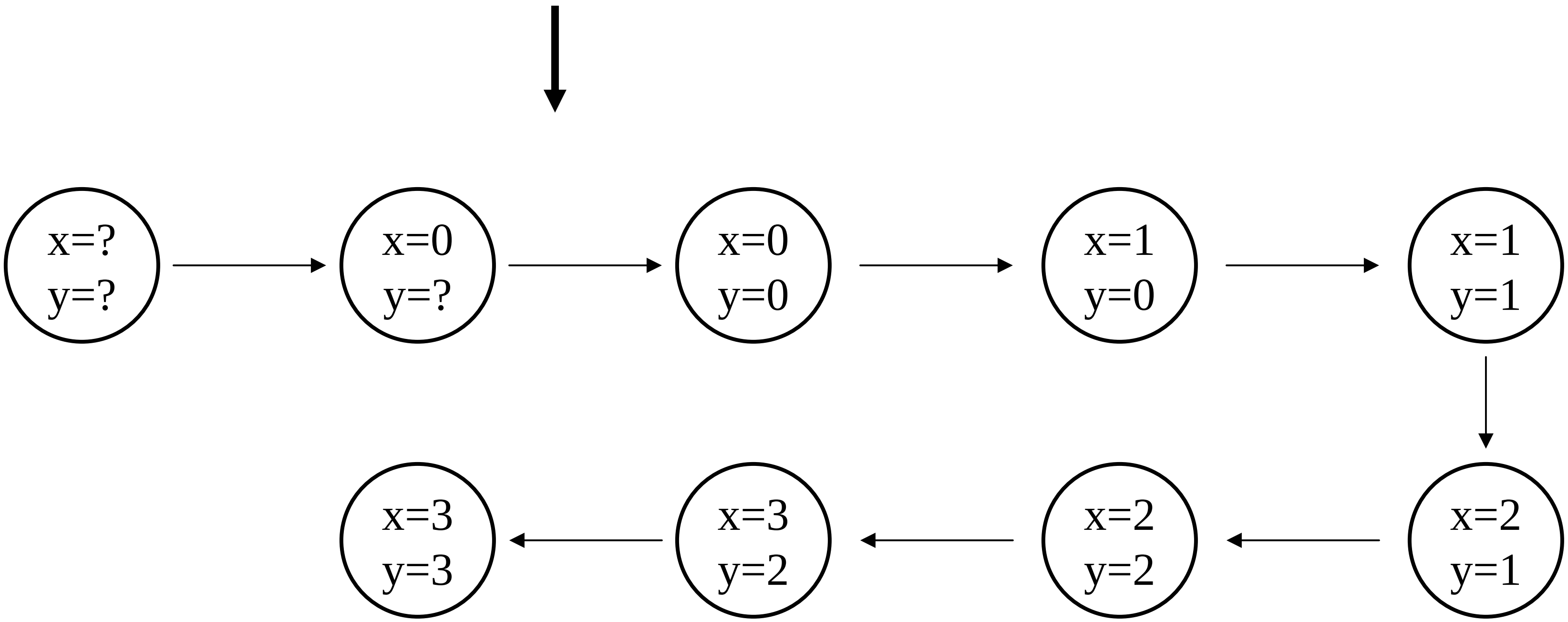
- Program state: program counter + variable values + heap.
- Heap: ignore.
- Program counter: low-level detail, language dependent. Use graphical representation: different nodes are different states.
- Example:

```
x:=0; y:=0;
for i:=1 to 3 do
  (x:=x+1; y:=y+1)
```

If “steps” are assignments, has 8 steps, and 9 states: the initial state and the state after each step.

State transition diagrams

```
x:=0; y:=0;
for i:=1 to 3 do
  (x:=x+1; y:=y+1)
```



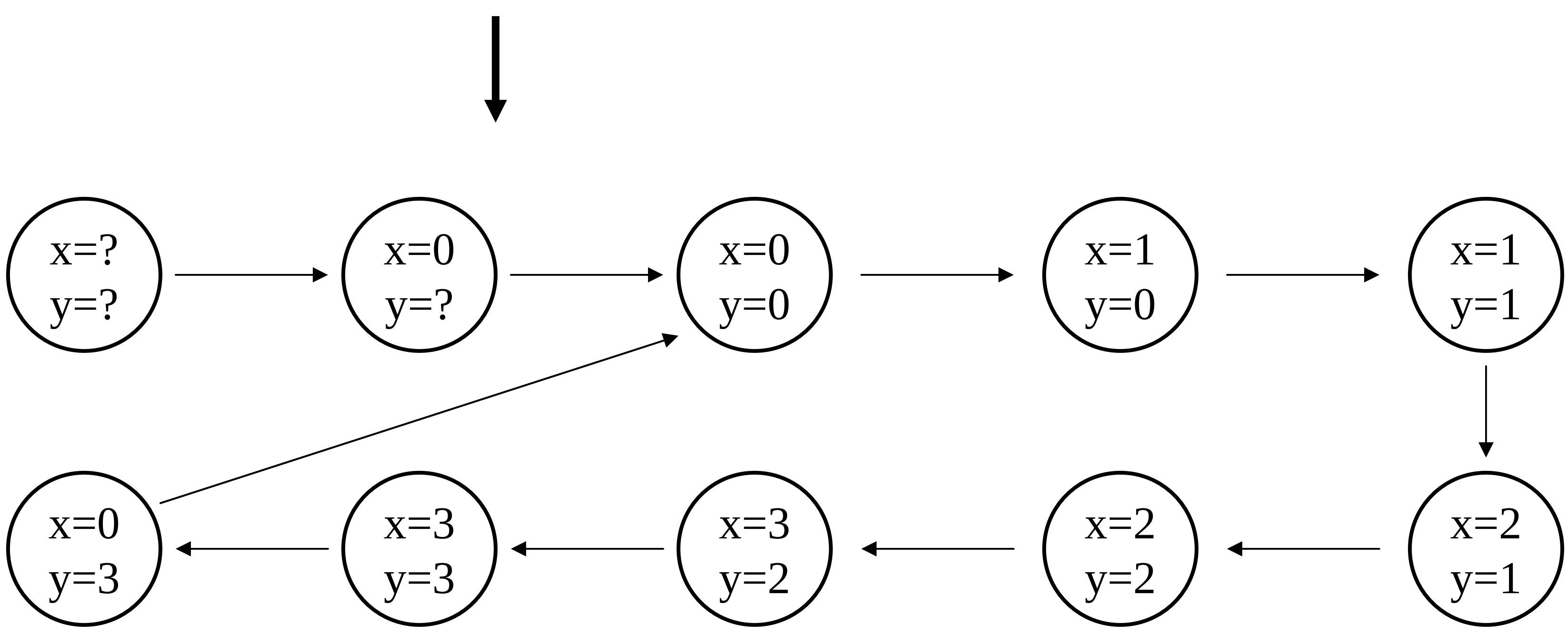
What if we added a new final line (`x:=0; y:=0`)?

Non-terminating programs

- In examples of interest, programs will be effectively *non-terminating*.
- E.g.
  - circuits run indefinitely
  - protocols run repeatedly (and simultaneously)
  - avionics software should keep running until the aircraft is switched off
- Termination is uninteresting: we are interested in what happens while the program is executing.
- Trivial to handle in state transition diagrams.

A non-terminating state-transition system

```
x:=0; y:=0;
L: for i:=1 to 3 do
       (x:=x+1; y:=y+1);
x:=0; y:=0; go to L
```



A trivial kind of model checking

- Suppose the specification is that some boolean expression *e* is true at each state (i.e. *e* is an *invariant*).
- E.g. *x=y* in the previous example is true at every state.
- Can check by running the program and evaluating *e* after each step.
- What about properties that aren't an invariant, e.g.
  - "there is a state where *x=2"?*
  - "infinitely often there are states where *x=2*"
  - "every state where *y=0* is followed by a state where *y=1"*
- What about concurrency and non-determinism?

Finite state systems

- If variables can only take on finitely many values (e.g. *x:[1..10]*, *y:bool)*, then the program is finite state (there are only finitely many program counter values).
- It suffices to restrict variables to boolean values.
- E.g. *x:[0..7]* can be represented by 3 boolean variables using a binary digit representation.
- $x=3$ corresponds to
  x_1=0 and x_2=1 and x_3=1.
- From now on, restrict attention to programs with boolean variables only.

Mutex: an example with concurrency

- Mutual exclusion: each process has a *critical section*: no two processes can be in their critical sections at the same time.
- Don't care what's in each critical section.
- Use "status" variables to track sections. Values *n*, *t* and *c*.
- First process:

```
while 1 do
(<non-cs-1>; st1:=t;
 when st2=n or st2=t do st1:=c;
 <cs-1>;
 st1:=n)
```

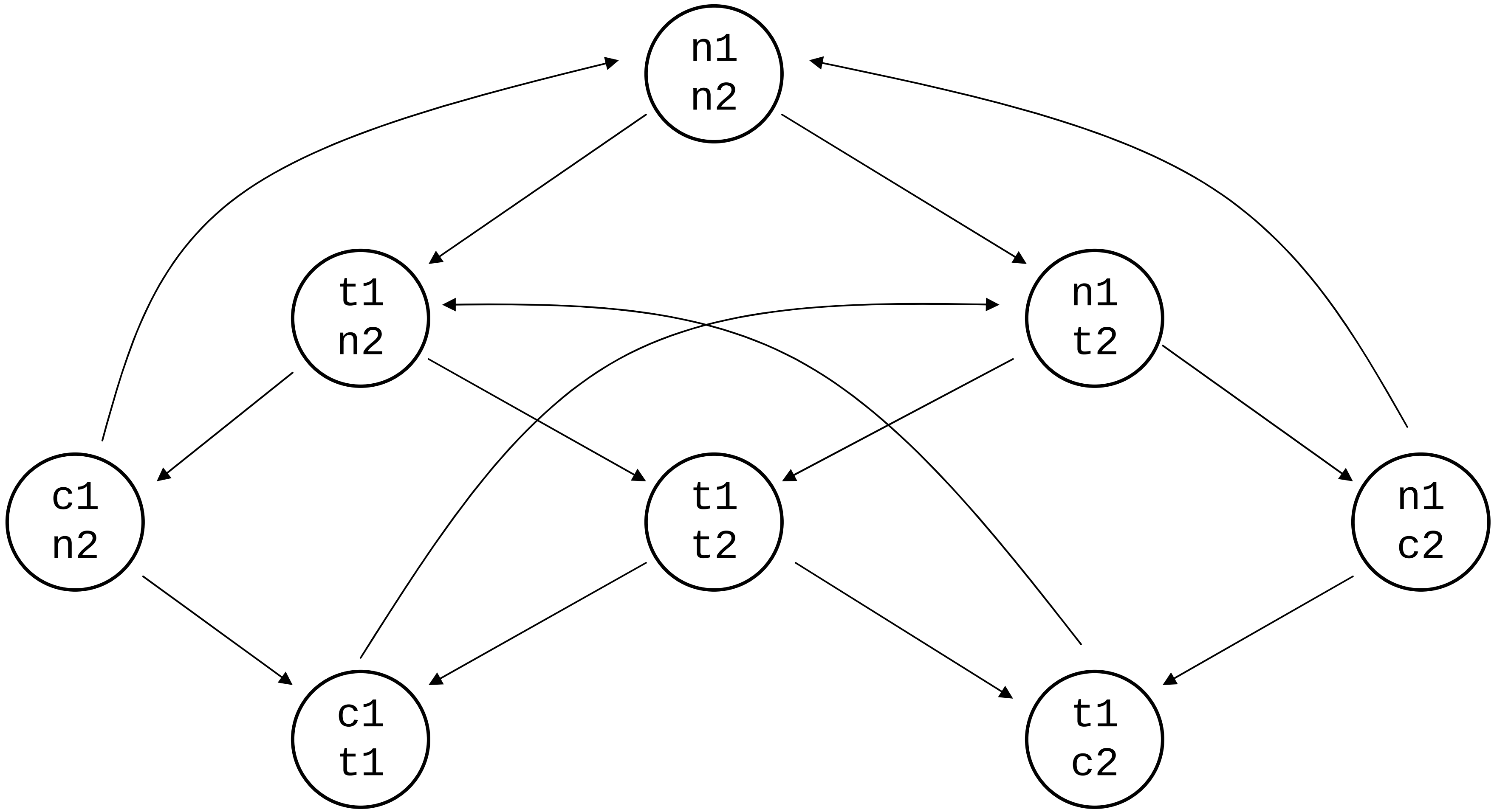
Mutex example, continued

- Second process:

```
while 1 do
(<non-cs-2>; st2:=t;
 when st1=n or st1=t do st2:=c;
 <cs-2>;
 st2:=n)
```

- Run processes concurrently, initially `st1=st2=n.`
- Boolean variables: *n1, n2, c1, c2, t1, t2* represent `st1`, `st2`, e.g. `st1=t` represented by `n1=0, t1=1, c1=0.`
- In diagram, label only with variables with value 1.

State transition diagram for mutex



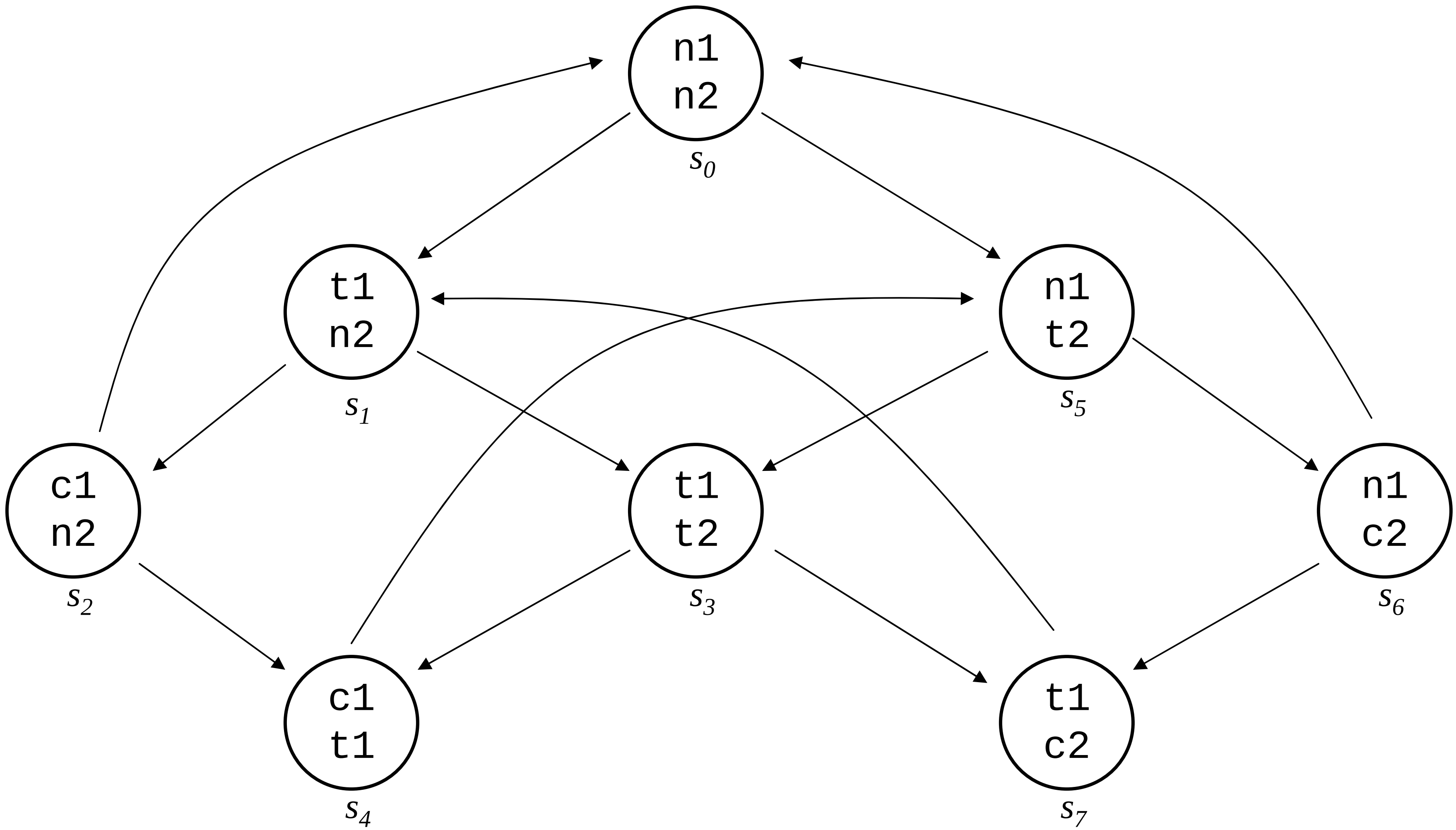
Kripke structures

Definition. Let *AP* be a set of boolean variables. A *Kripke Structure* over *AP* is a triple $M=(S,R,L)$ where

- S is a finite set (of states)
- $R \subseteq S \times S$ such that for all states s, there is a state s' such that $(s,s') \in R$. (R is the *transition relation*.)
- $L \in S \to AP$ (L is the *labelling function*)

Write $s \to s'$ for $(s,s') \in R$.
Sometimes a Kripke structure will have some *initial states*.

Can translate mutex example to a Kripke structure.



$S = \{s_0, s_1, s_2, s_3, s_4, s_5, s_6, s_7\}$ initial states: $S_0 = \{s_0\}$

$R = \{ (s_0, s_1), (s_0, s_5), (s_1, s_2), (s_1, s_3), (s_2, s_0), \ ...\}$

$L(s_0)=\{n1,n2\}$, $L(s_1)=\{t1,n2\}$, $L(s_2)=\{c1,n2\}$, ...

Paths in a Kripke structure

Definition. A *path* in a Kripke structure is an infinite sequence $\pi = \pi_1, \pi_2, \pi_3, \ldots$ where for all $i \geq 1$, $\pi_i \rightarrow \pi_{i+1}$.

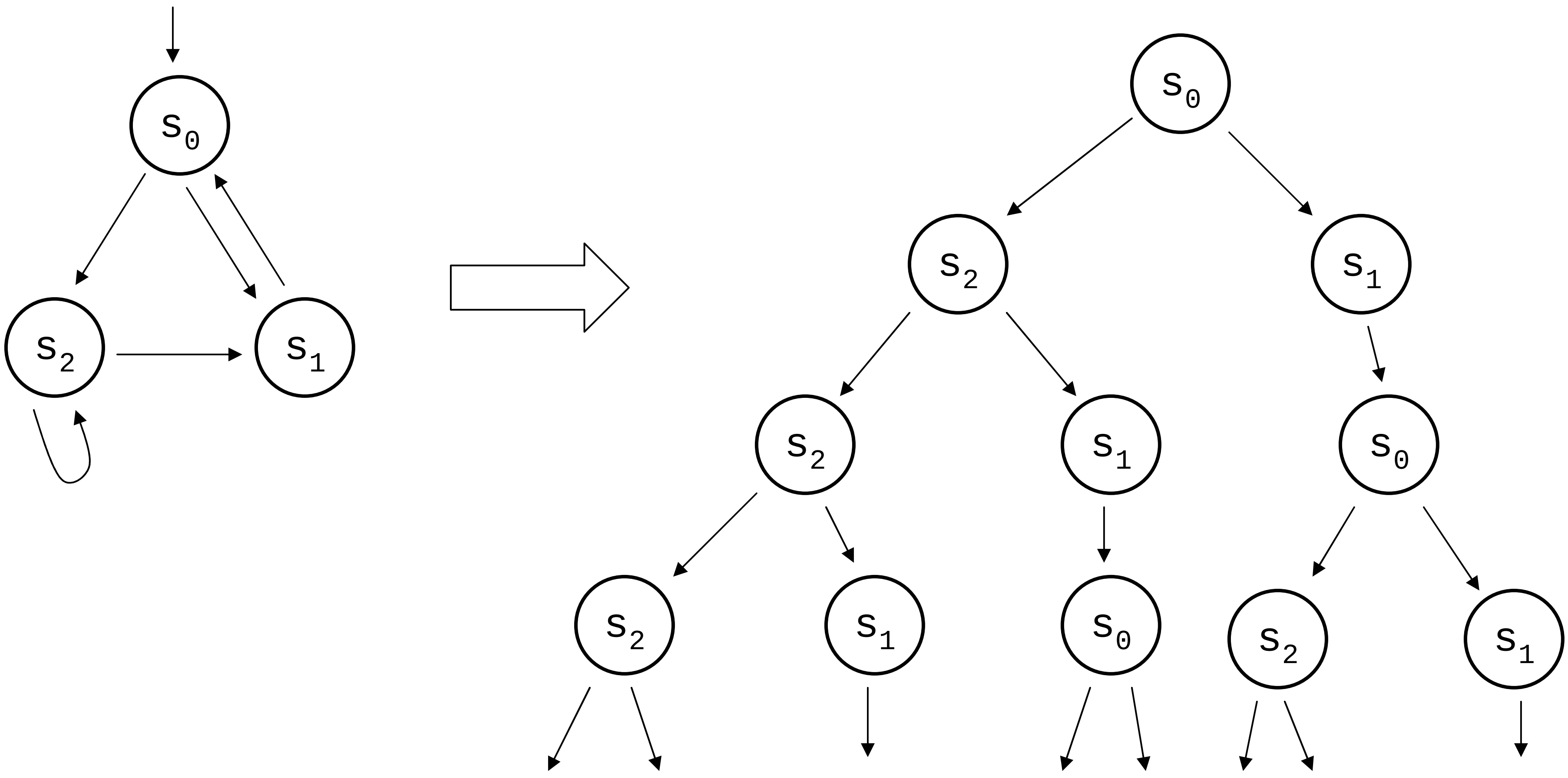
In the example, there is a path

$$s_0 \rightarrow s_5 \rightarrow s_6 \rightarrow s_0 \rightarrow s_5 \rightarrow s_6 \rightarrow s_0 \rightarrow \ldots$$

A path π *starts* at a state s if $\pi_1 = s$.

It's also useful, though not formally necessary, to think about computation *trees*.

Computation trees (via unwinding)



$L(s_0)=\{p\}, \quad L(s_1)=\{r,p\}, \quad L(s_0)=\{q,r\}$

Informal mutex properties

Safety: always have $\neg$(c1$\wedge$c2).

Liveness: whenever t1, eventually c1. (Similarly for process 2)

Non-blocking: It's always true that process 1 can always progress to a state where t1.

No strict sequencing: The protocol is not a trivial one that forces strict alternation (c1 c2 c1 c2 c1 ...).

CTL

- All the mutex properties can be expressed in CTL.
- CTL formulas are properties of *states*.
- Formula builders: *AF*, *EF*, *AG*, *EG*, *AX*, *EX*, *AU*, *EU*, and boolean connectives.
- "A" = on all paths starting at the given state
- "E" = there exists a path starting at the given state
- "X" = in the next state on the path
- "G" = on all states in the path (i.e. <u>G</u>lobally)
- "F" = on some state in the path (i.e. in the <u>F</u>uture)
- "U" = first formula holds until some point where the second formula starts holding.

CTL continued

- CTL is a formal language: precise syntax and semantics.
- There is a linear-time algorithm for model checking CTL:
  - Input: Kripke structure and a CTL formula
  - Output: yes/no, answering the question *does the K.S. satisfy the formula?*
- First algorithm based on explicit analysis of states.
- Later improvement (enormous!) groups states into sets represented by BDD's ("binary decision diagrams").

Syntax of CTL

A CTL formula φ has one of the following forms:

- 0, 1, p, $\neg\ \varphi$, $\varphi \wedge \varphi$, $\varphi \Rightarrow \varphi$, $\varphi \vee \varphi$ (for any variable p in *AP*)
- AX φ , EX φ
- AG φ , EG φ
- AF φ , EF φ
- A[φ U φ], E[φ U φ]

Note AU, EU weird syntax.

CTL semantics

Definition. A Kripke structure *M* satisfies a CTL formula φ if $M,s \models \varphi$, where $M,s \models \varphi$ is defined by induction on the size of φ.

[Details omitted – see text page 157.]

CTL examples: mutex specification

Safety: $AG\ \neg(c1 \wedge c2)$.

Liveness: $AG\ (t1 \Rightarrow AF\ c1)$.

Non-blocking: $AG\ (n1 \Rightarrow EX\ t1)$.

No strict sequencing:

$EF\ (c1 \wedge E[c1\ U\ (\neg c1 \wedge E[\neg c2\ U\ c1])])$.

Some generic CTL examples

1. $\neg$ EF (Start $\wedge \neg$Ready)
2. AG (AF ServiceAvailable)
3. AG (EF Restart)
4. AG (Request $\Rightarrow$ AF Acknowledgment)

Fixing mutex

- Add a new variable `turn` (initially 1).
- First process:

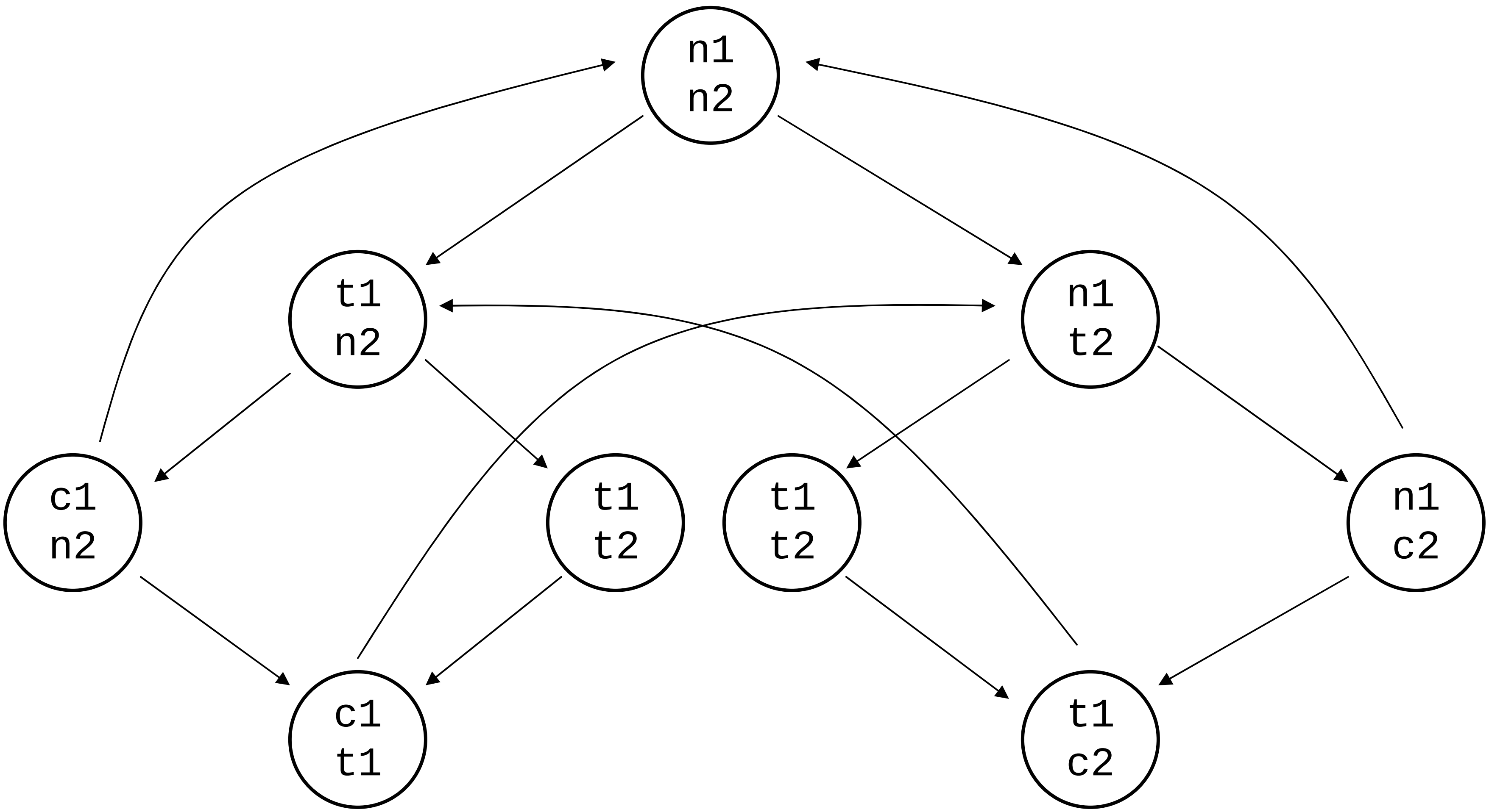
```
while 1 do
(<non-cs-1>; st1:=t;
 when st2=n or (st2=t and turn=1)
   do st1:=c;
 <cs-1>;
 st1:=n)
```

- Second process:

```
while 1 do
(<non-cs-2>; st2:=t;
 when st1=n or (st1=t and turn=2)
   do st2:=c;
 <cs-2>;
 st2:=n)
```

New Kripke structure

- **Note:** we can choose the variable set *AP* it's over. Leave it the same – this means the new variable `turn` is ignored.



Model-checking algorithm simplification

- Only need to consider operators 0, $\neg$, $\wedge$, AF, EU, EX.
- The rest are equivalent via $\equiv$: $\varphi \equiv \psi$ iff the two formulas satisfy the same K.S.'s in the same states.
- $1 \equiv \neg 0$
- $\varphi \vee \psi \equiv \neg(\neg\varphi \wedge \neg\psi)$
- AX $\varphi \equiv \neg$EX$(\neg\varphi)$
- EG $\varphi \equiv \neg$AF$(\neg\varphi)$
- EF $\varphi \equiv$ E[1 U ψ]

Simplification continued

- $AG\ \varphi \equiv \neg EF(\neg\varphi)$
- $A[\varphi\ U\ \psi] \equiv \neg(\ E[\neg\psi\ U\ \neg\varphi\wedge\neg\psi] \vee EG\ \neg\psi\ )$

The CTL model checking algorithm

- Input: Kripke structure *M* and CTL formula φ.
- Output: set of states where φ holds. (Can derive the desired yes/no answer.)
- Idea: label graph with subformulae known to be true, starting with smallest.
- Basically a fairly simple graph algorithm.

Algorithm top-level

`mc(`φ_0`,` M`):`

1. Translate φ_0 so it mentions only 0, $\neg$, $\wedge$, `AF`, `EU`, `EX`, and variables.
2. For each state *s* of *M*, initialize *T(s)* to be the empty set. *T(s)* is the set of subformulae of φ known to be true.
3. Let *l* be a list of all subformulae of φ_0, sorted in nondecreasing order of size.
4. For each φ in *l*, call the procedure *add*(φ).
5. Return the set of all *s* such that $\varphi_0 \in T(s)$.

Definition of add(φ) by pattern matching

- add(p) where p is variable: if $p \in L(s)$ then add p to $T(s)$.

- add($\neg\psi$): for each $s \in S$, if $\psi \notin T(s)$, then add φ to $T(s)$.

- add($\varphi \wedge \psi$): for each $s \in S$, if $\varphi \in T(s)$ and $\psi \in T(s)$ then add φ to $T(s)$.

- add(EXψ): for each $s \in S$ and for each $s' \in S$ such that $s \rightarrow s'$, if $\psi \in T(s)$ then add φ to $T(s)$.

Definition of `add(`φ`)`, continued

- `add(AF`ψ`)`: for each $s \in S$, if $\psi \in T(s)$ then add φ to $T(s)$. Now repeat the following step until no $T(s)$ is changed: if there is a state s such that $\varphi \in T(s')$ whenever $s \rightarrow s'$, then add φ to $T(s)$.

- `add(E[`γ `U` ψ`])`: for each $s \in S$, if $\psi \in T(s)$ then add φ to $T(s)$. Now repeat the following step until no $T(s)$ is changed: if there is a state s such that $\gamma \in T(s)$ and for some s', $s \rightarrow s'$ and $\varphi \in T(s')$, then add φ to $T(s)$.

Complexity

- $|M|$ is the number of states plus the number of transitions.
- $|\varphi_0|$ is the number of subformulae of φ_0.
- Complexity is O($f(|M|, |\varphi_0|)$) – what is *f*?
- `add`, in `AF` and `EU` cases, has triply-nested loops over states.
- So far, looks like $f(x,y)=x^3y$.
- However, other cases are fine, $f(x,y)=x^3y$, and we can optimize the two bad cases.

Optimizing EU

- View as graph problem.
- Looking for all *s* where there is a finite path starting at *s*, with γ “true” along the way, ending at an *s’* where ψ true.
- Consider such paths in reverse:
  - reverse all edges in graph
  - for each node *s’* where $\psi \in T(s')$, run a depth-first search starting with *s’*, only visiting nodes *s* where $\gamma \in T(s)$.
  - for each visited node *s*, add φ to $T(s)$.
- Linear in number of edges of the graph + the number of nodes.

Optimizing AF

- Not so easy.
- Note that $\mathsf{AF}\psi \equiv \neg\mathsf{EG}\neg\psi$, so it suffices to process EG case efficiently.
- Background: a *strongly connected component* in a directed graph is a maximal set of nodes *C* such that any two nodes in *C* are connected by a path using only nodes from *C*, and if *C* has only one node, then there is an edge from the node to itself.
- Linear in size of graph.

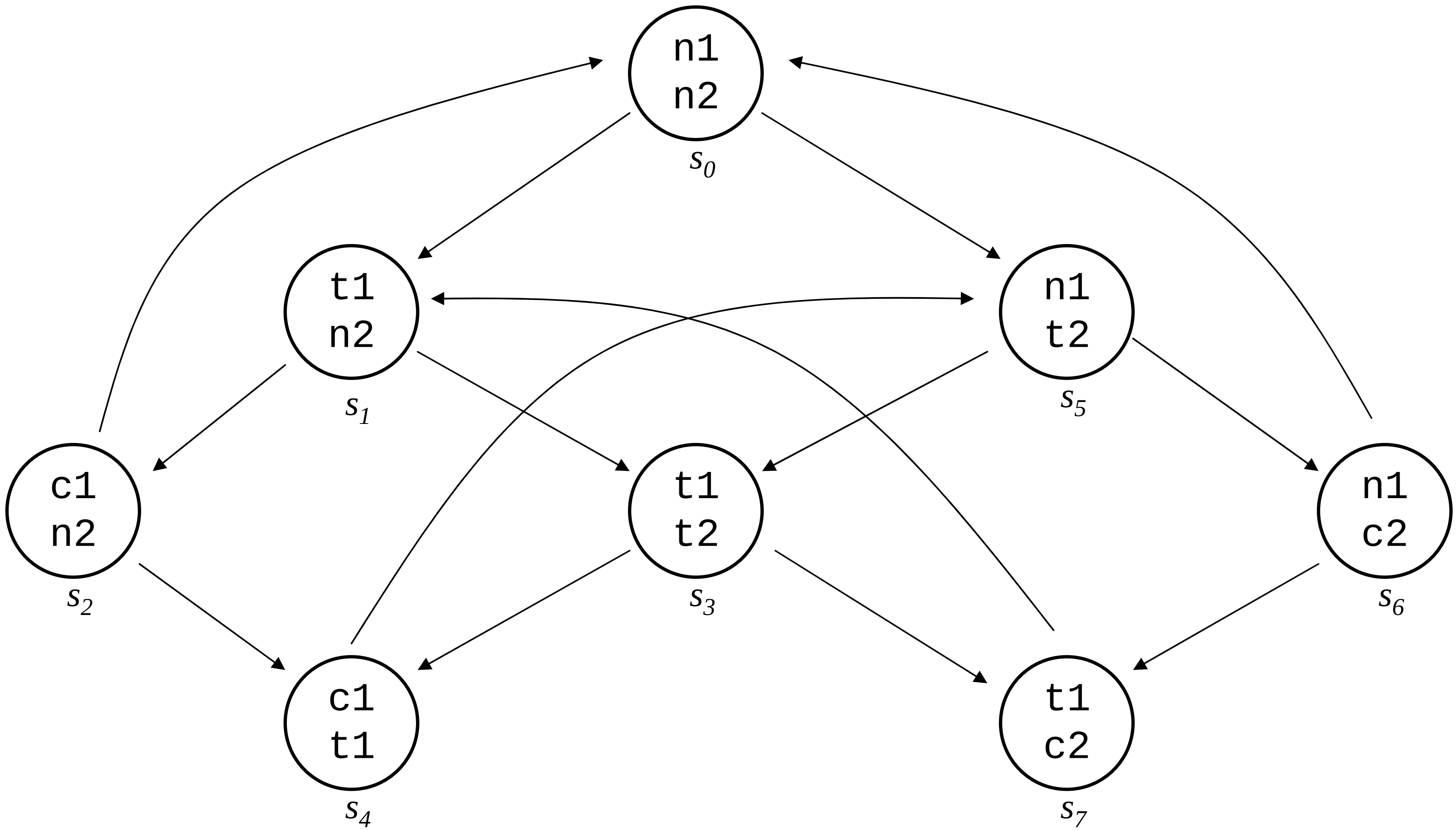
Optimizing `add(AF`ψ`)`: `add(EG`ψ`)`

- View as graph problem.
- Remove all nodes *s* where $\psi \notin T(s)$, call resulting graph *G*.
- Find strongly-connected components $C_1, ..., C_k$ of *G*.
- For each node *s* in one of the C_i, run a depth-first search in the reverse graph of *G*, starting at *s*.
- For each visited node *s*, add φ to $T(s)$.

Example

- Run algorithm on `AG AF(c1 ∨ c2)` and mutex example.
- "Simplifies" to
  `¬E[¬0 U ¬AF¬(¬c1∧¬c2)]`
- Simulate execution by labelling graph with members of *T(s)*.

Example continued



Summary

- Defined Kripke structures, modeling finite-state systems.
- Defined CTL, a formalism for specifying properties of Kripke structures.
- Gave graph-based algorithm for deciding at which states of a K.S. M a given CTL formula φ holds.
- Complexity of algorithm: $O(|\varphi| \times |M|)$.