CS181u Applied Logic & Automated Reasoning

Lecture 6

Transition Systems

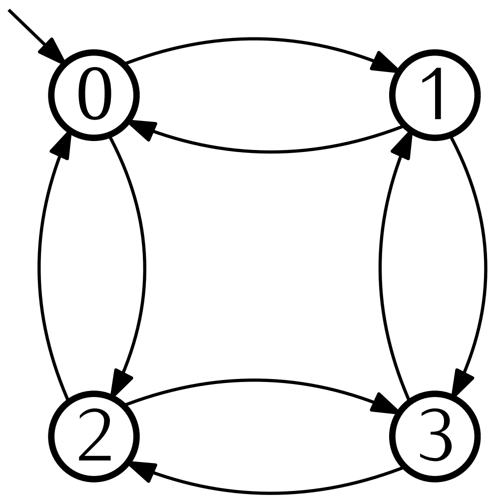
Announcement: office hours

Monday: 4 to 5 in my office Olin 1271, or by appointment

Grutor Hours: fill out the poll on Piazza

Transition System Representations

A transition system $\mathcal{M}$ can be specified by listing out all of the pieces.



States: $S = \{0, 1, 2, 3\}$

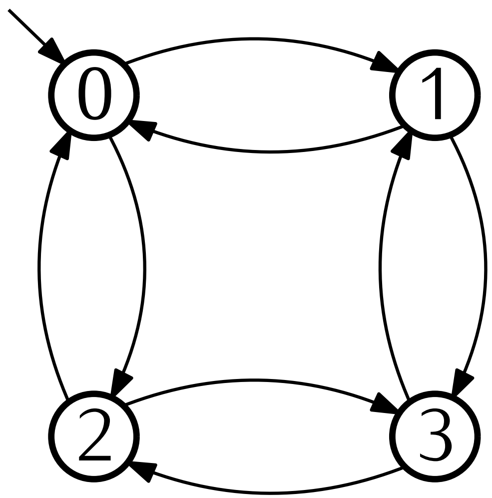
Initial States: $I = \{0\}$

Transitions:

$$R = \left\{ \begin{array}{cccc} (0,1) & (0,2) & (1,3) & (2,3) \\ (1,0) & (2,0) & (3,1) & (3,2) \end{array} \right\}$$

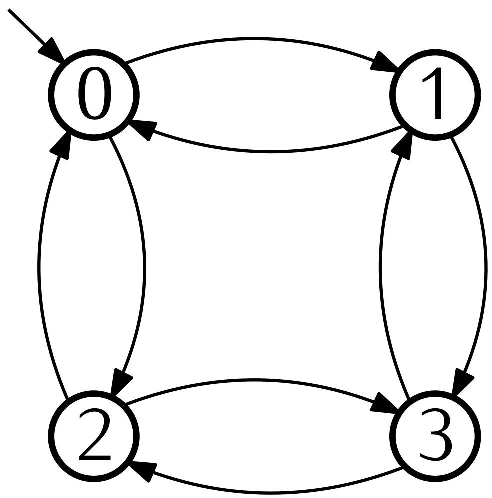
Symbolic Representation

Represent $\mathcal{M}$ using Boolean logic.



Symbolic Representation

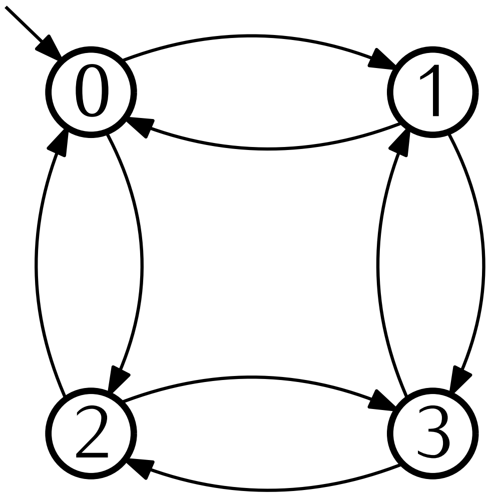
Represent $\mathcal{M}$ using Boolean logic.



States		
0		
1		
2		
3		

Symbolic Representation

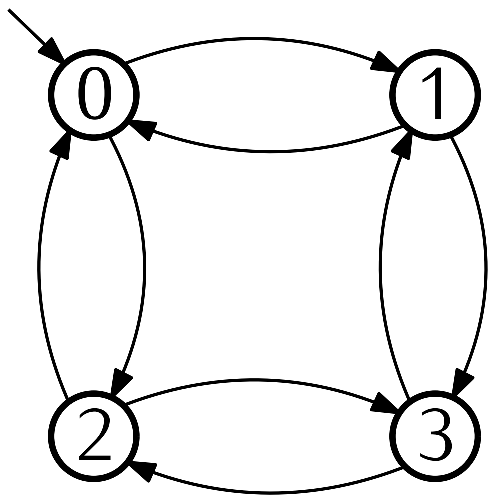
Represent $\mathcal{M}$ using Boolean logic.



States	binary	
	x	y
0	0	0
1	0	1
2	1	0
3	1	1

Symbolic Representation

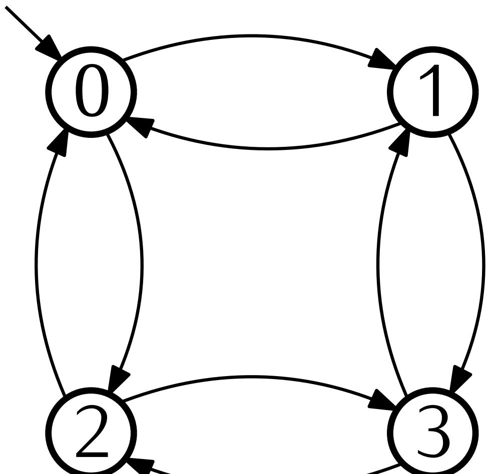
Represent $\mathcal{M}$ using Boolean logic.



States	binary		truth values	
	x	y	x	y
0	0	0	F	F
1	0	1	F	T
2	1	0	T	F
3	1	1	T	T

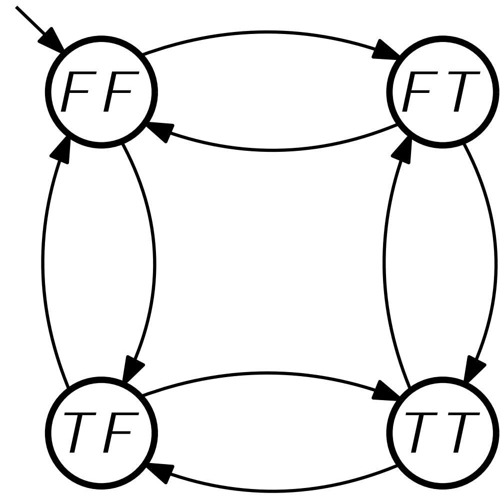
Symbolic Representation

Represent $\mathcal{M}$ using Boolean logic.



Boolean state variables

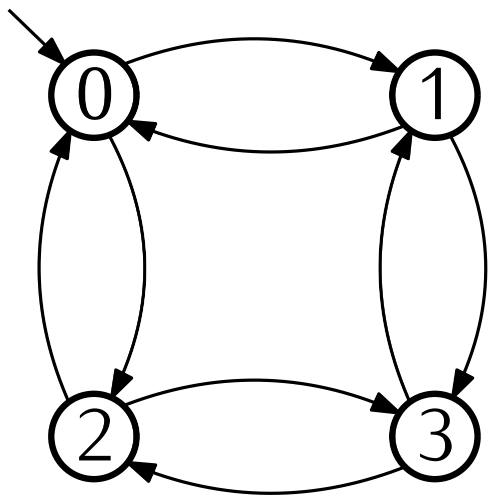
$$V = \{x, y\}$$



States	binary		truth values	
	x	y	x	y
0	0	0	F	F
1	0	1	F	T
2	1	0	T	F
3	1	1	T	T

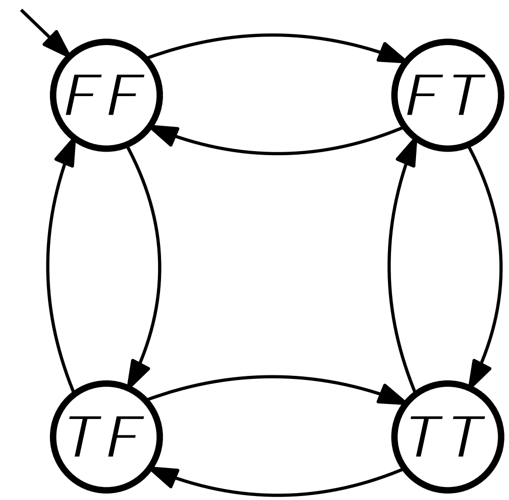
Symbolic Representation

Represent $\mathcal{M}$ using Boolean logic.



Boolean state variables

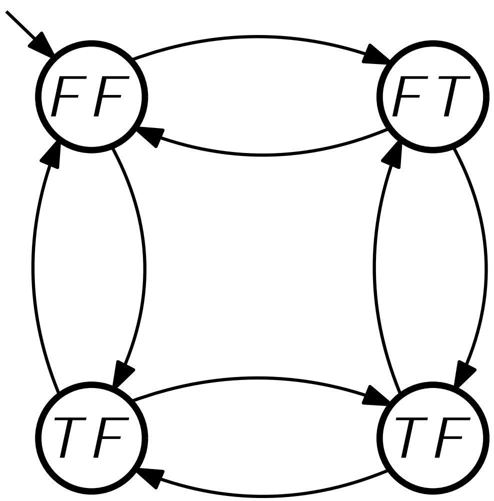
$V = \{x, y\}$



States	binary		truth values		Boolean formula
	x	y	x	y	
0	0	0	F	F	$\neg x \wedge \neg y$
1	0	1	F	T	$\neg x \wedge y$
2	1	0	T	F	$x \wedge \neg y$
3	1	1	T	T	$x \wedge y$

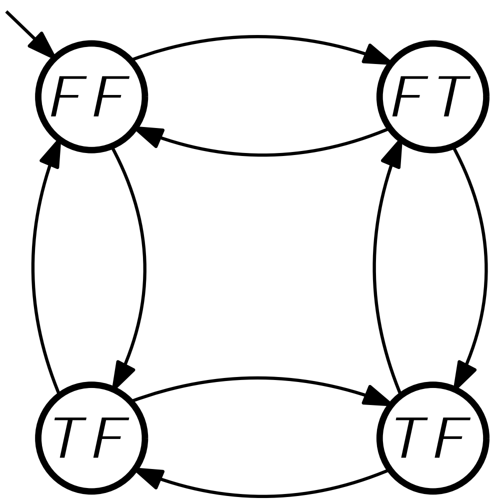
Symbolic Representation

Represent $\mathcal{M}$ using Boolean logic.



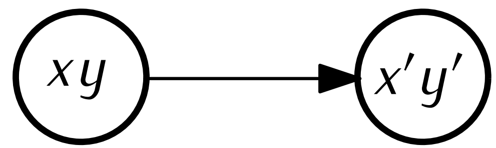
Symbolic Representation

Represent $\mathcal{M}$ using Boolean logic.



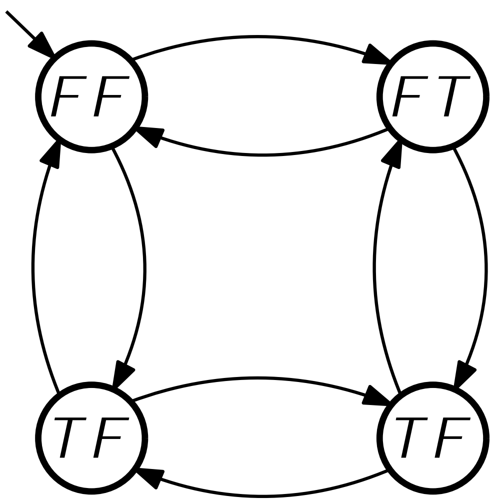
Transitions:

Let the "next" state variables be $V' = \{x', y'\}$



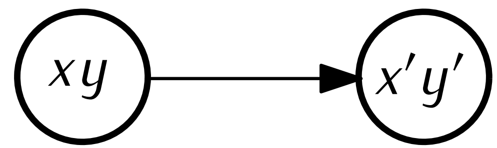
Symbolic Representation

Represent $\mathcal{M}$ using Boolean logic.



Transitions:

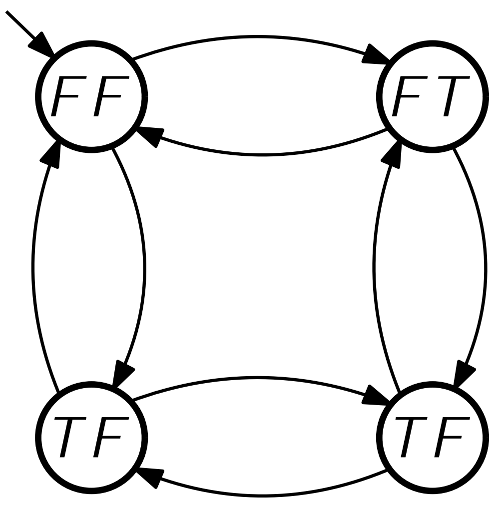
Let the "next" state variables be $V' = \{x', y'\}$



$$R \quad \equiv \quad (x' = x \wedge y' = \neg y) \vee (x' = \neg x \wedge y' = y)$$

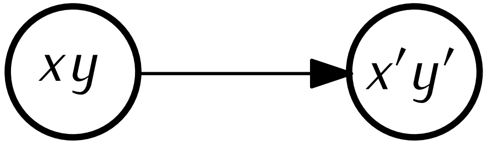
Symbolic Representation

Represent $\mathcal{M}$ using Boolean logic.



Transitions:

Let the "next" state variables be $V' = \{x', y'\}$

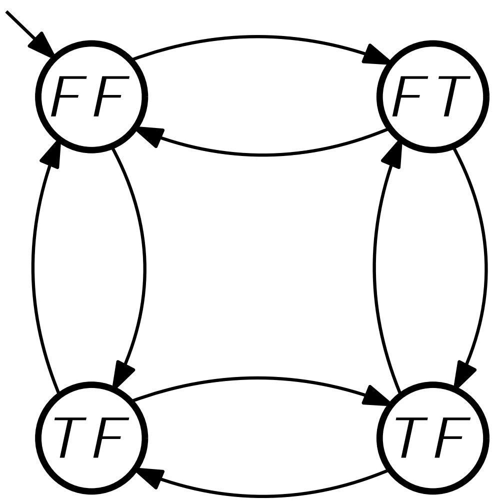


$$R \quad \equiv \quad (x' = x \wedge y' = \neg y) \vee (x' = \neg x \wedge y' = y)$$

"we can get from one state to the next by keeping one variable the same and negating the other"

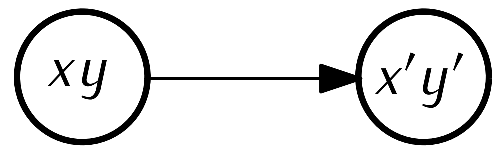
Symbolic Representation

Represent $\mathcal{M}$ using Boolean logic.



Transitions:

Let the "next" state variables be $V' = \{x', y'\}$



$$R \quad \equiv \quad (x' = x \wedge y' = \neg y) \vee (x' = \neg x \wedge y' = y)$$

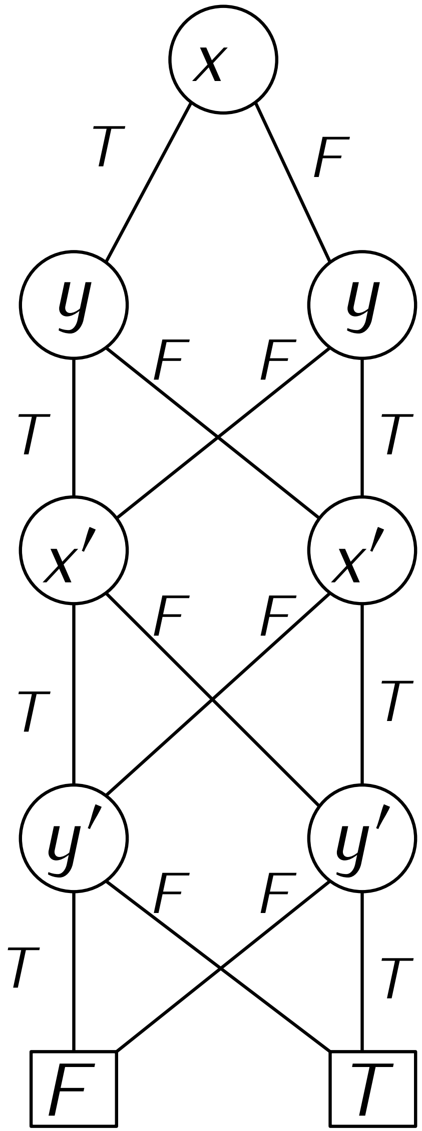
Explicit transitions	$(0,1)$	$(2,3)$	$(1,3)$	$(0,2)$
	$(1,0)$	$(3,2)$	$(3,1)$	$(2,0)$

"we can get from one state to the next by keeping one variable the same and negating the other"

Symbolic Representation

$$R \equiv (x' = x \wedge y' = \neg y) \vee (x' = \neg x \wedge y' = y)$$

BDD for R



Symbolic Model Checking: 10^{20} States and Beyond

J. R. Burch E. M. Clarke K. L. McMillan*
School of Computer Science
Carnegie Mellon University

D. L. Dill L. J. Hwang
Stanford University

Abstract

Many different methods have been devised for automatically verifying finite state systems by examining state-graph models of system behavior. These methods all depend on decision procedures that explicitly represent the state space using a list or a table that grows in proportion to the number of states. We describe a general method that represents the state space *symbolically* instead of explicitly. The generality of our method comes from using a dialect of the Mu-Calculus as the primary specification language. We describe a *model checking* algorithm for Mu-Calculus formulas that uses Bryant's *Binary Decision Diagrams* (1986) to represent relations and formulas. We then show how our new Mu-Calculus model checking algorithm can be used to derive efficient decision procedures for CTL model checking, satisfiability of linear-time temporal logic formulas, strong and weak observational equivalence of finite transition systems, and language containment for finite ω-automata. The fixed point computations for each decision procedure are sometimes complex, but can be concisely expressed in the Mu-Calculus. We illustrate the practicality of our approach to symbolic model checking by discussing how it can be used to verify a simple synchronous pipeline circuit.

This phase: model checking

This phase will focus mostly on model checking.

Proof-based systems are good for programs that take input and then compute a result.

Model checking is good for programs that define reactive systems.

A reactive system consists of multiple components operating concurrently and indefinitely.

Next Few Weeks:

Linear Temporal Logic (LTL)

We will assign symbols for expressing temporal system requirements like *always* (G), *eventually* (F), *next* (X), *until* (U), and a few more. We will give a formal and unambiguous semantics to these symbols.

Transition Systems

We will learn a formal system of specifying transition systems (which we often depict as a transition diagram).

Concurrency Concepts

Safety, liveness, mutual exclusion, ...

Temporal Logic Software

Symbolic Model Verifier (NuSMV)

Next Few Weeks:

Linear Temporal Logic (LTL)

We will assign symbols for expressing temporal system requirements like *always* (G), *eventually* (F), *next* (X), *until* (U), and a few more. We will give a formal and unambiguous semantics to these symbols.

Transition Systems

We will learn a formal system of specifying transition systems (which we often depict as a transition diagram).

Concurrency Concepts

Safety, liveness, mutual exclusion, ...

Today

Temporal Logic Software

Symbolic Model Verifier (NuSMV)

Fun Trivia!

From “Principles of Model Checking”

2.5 Bibliographic Notes

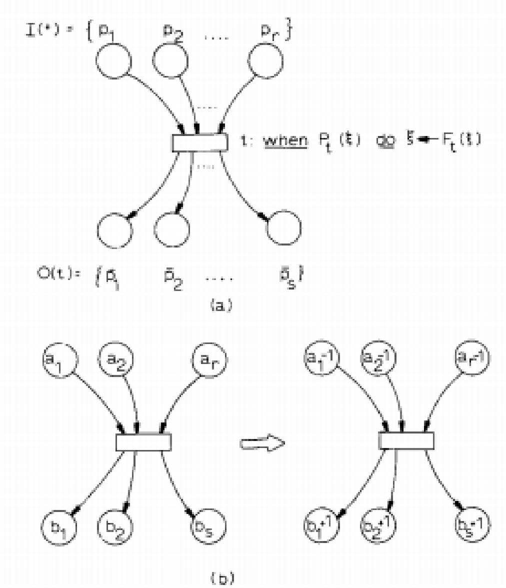
Transition systems. Keller was one of the first researchers that explicitly used transition systems [236] for the verification of concurrent programs. Transition systems are used

CACM 1976

Formal Verification of Parallel Programs

Robert M. Keller
Princeton University

Fig. 1 (a) representation of transition node; (b) action of transition on place variables (assuming $I(t) \cap O(t) = \varnothing$).



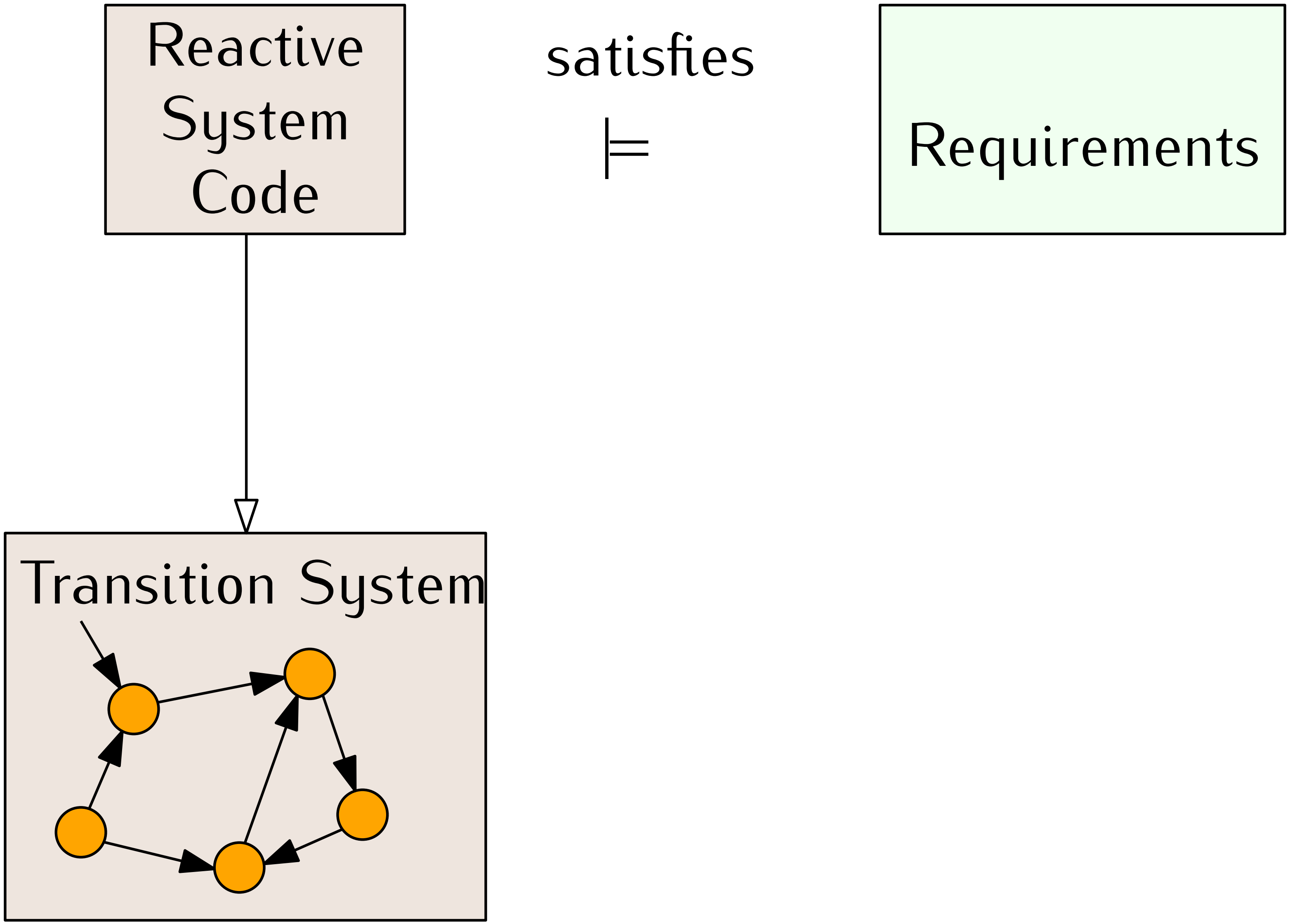
Reactive System Code


Reactive System Code
Requirements

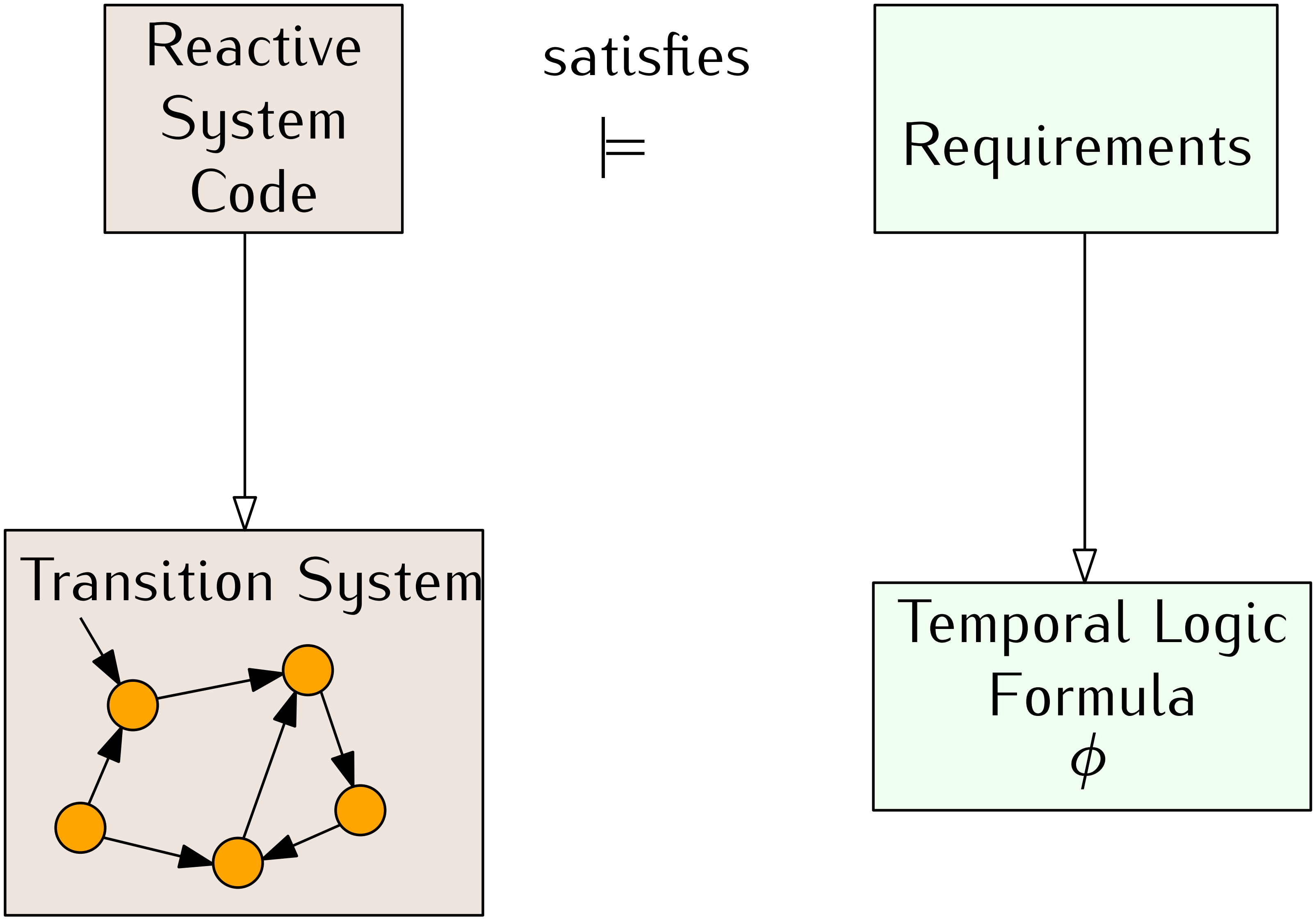
Goal: show that

Reactive System Code	satisfies $\models$	Requirements

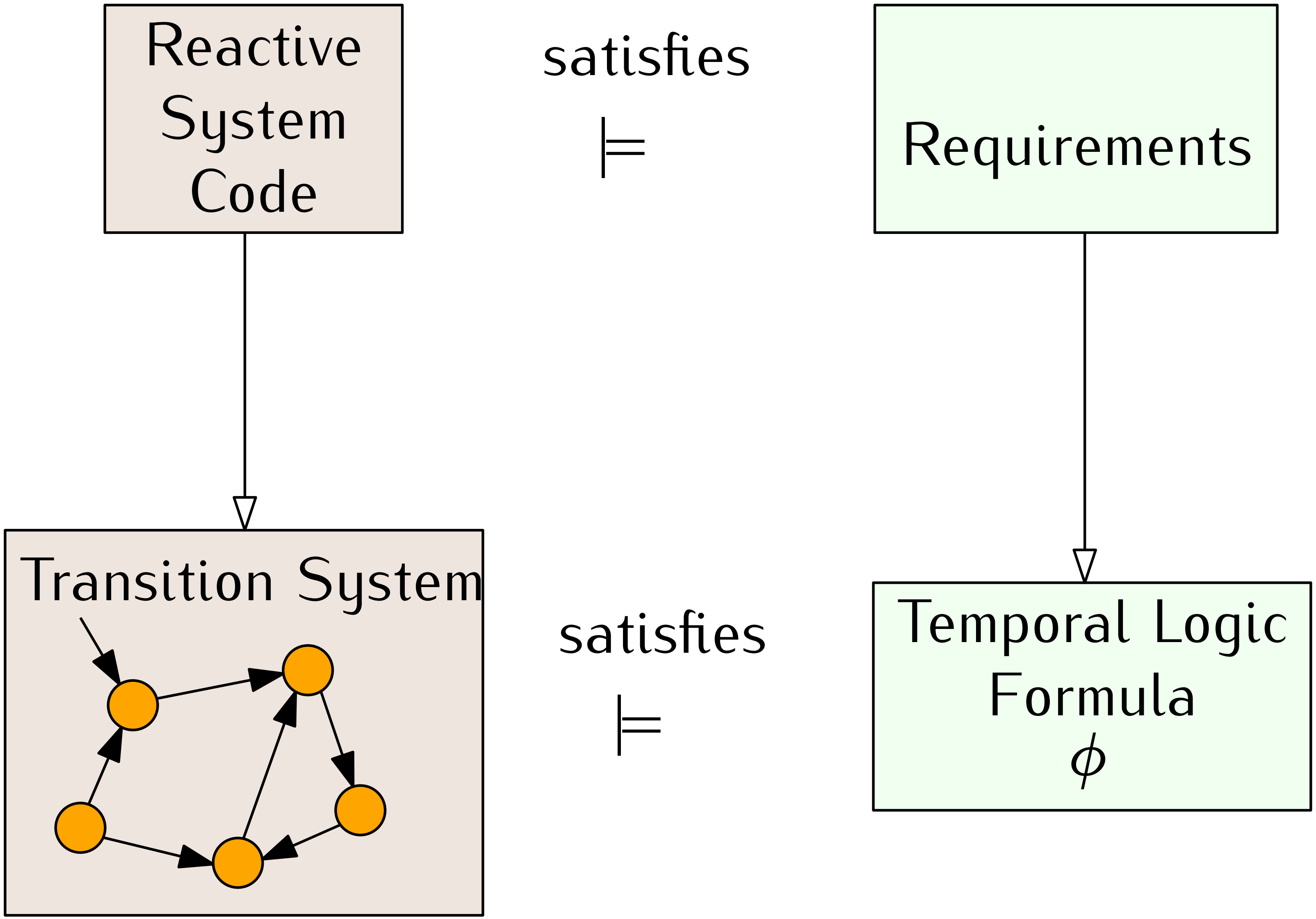
Goal: show that

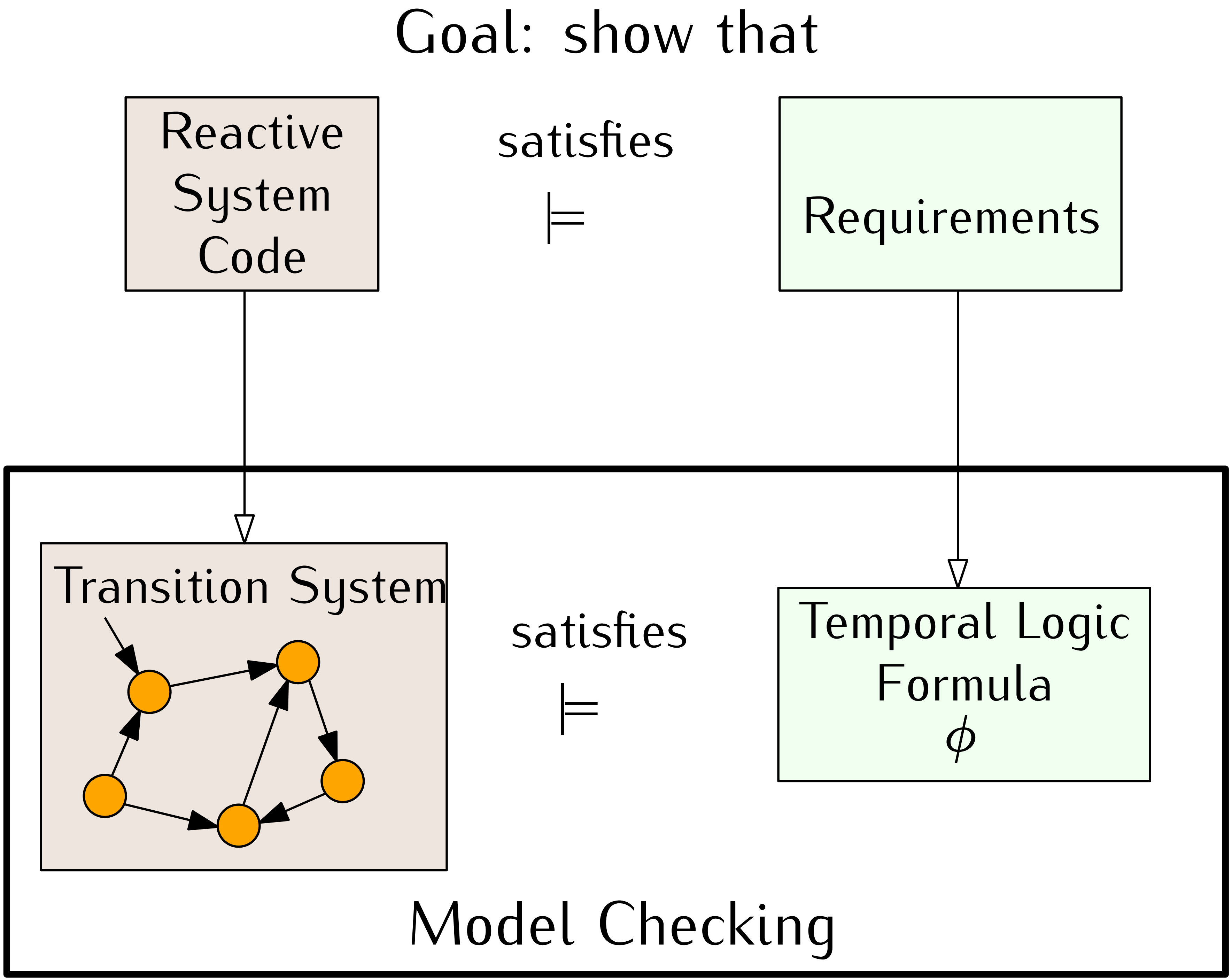


Goal: show that



Goal: show that




Goal: show that
Reactive System Code
satisfies
$\models$
Requirements
Transition System
satisfies
$\models$
Temporal Logic Formula ϕ
Model Checking

Case Study: Mutual Exclusion

Motivation: reactive system is a bank, two ATMs, and two customers that share an account.

Case Study: Mutual Exclusion

Motivation: reactive system is a bank, two ATMs, and two customers that share an account.

Current balance $b = 1000$

C_1 wants to deposit $d_1 = 100$

C_2 wants to deposit $d_2 = 100$

Case Study: Mutual Exclusion

Motivation: reactive system is a bank, two ATMs, and two customers that share an account.

Current balance $b = 1000$

C_1 wants to deposit $d_1 = 100$

C_2 wants to deposit $d_2 = 100$

ATM_1 reads current balance $b_1 = 1000$

ATM_2 reads current balance $b_2 = 1000$

Case Study: Mutual Exclusion

Motivation: reactive system is a bank, two ATMs, and two customers that share an account.

Current balance $b = 1000$

C_1 wants to deposit $d_1 = 100$

C_2 wants to deposit $d_2 = 100$

ATM_1 reads current balance $b_1 = 1000$

ATM_2 reads current balance $b_2 = 1000$

ATM_1 writes $b = b_1 + d_1 = 1100$

ATM_2 writes $b = b_2 + d_2 = 1100$

Case Study: Mutual Exclusion

Motivation: reactive system is a bank, two ATMs, and two customers that share an account.

Current balance $b = 1000$

C_1 wants to deposit $d_1 = 100$

C_2 wants to deposit $d_2 = 100$

ATM_1 reads current balance $b_1 = 1000$

ATM_2 reads current balance $b_2 = 1000$

ATM_1 writes $b = b_1 + d_1 = 1100$

ATM_2 writes $b = b_2 + d_2 = 1100$

Final balance $b = 1100$

Case Study: Mutual Exclusion

Motivation: reactive system is a bank, two ATMs, and two customers that share an account.

Current balance $b = 1000$

C_1 wants to deposit $d_1 = 100$

C_2 wants to deposit $d_2 = 100$

ATM_1 reads current balance $b_1 = 1000$

ATM_2 reads current balance $b_2 = 1000$

ATM_1 writes $b = b_1 + d_1 = 1100$

ATM_2 writes $b = b_2 + d_2 = 1100$

Final balance $b = 1100$

One ATM shouldn't read the balance while another is performing a transaction! Race condition.

```
while(true){

    // non-critical code
    printWelcomeMessage();

    // critical section code
    updateBankBalance();

}
```

```
while(true){

  [[extra code to help with mutual exclusion]]

  // non-critical code
  printWelcomeMessage();

  [[extra code to help with mutual exclusion]]

  // critical section code
  updateBankBalance();

  [[extra code to help with mutual exclusion]]
}
```

```
while(true){

  [[ extra code to help with mutual exclusion ]]

  // non-critical code
  printWelcomeMessage();

  [[ extra code to help with mutual exclusion ]]

  // critical section code
  updateBankBalance();

  [[ extra code to help with mutual exclusion ]]
}
```

We want to focus on the mutual exclusion logic.

```
while(true){

  [[ extra code to help with mutual exclusion ]]

  // non-critical code
  printWelcomeMessage();

  [[ extra code to help with mutual exclusion ]]

  // critical section code
  updateBankBalance();

  [[ extra code to help with mutual exclusion ]]
}
```

We want to focus on the mutual exclusion logic.

Abstraction: forget about details we don't care about at the moment.

```
while(true){

  [[extra code to help with mutual exclusion]]

  // non-critical code
  printWelcomeMessage();

  [[extra code to help with mutual exclusion]]

  // critical section code
  updateBankBalance();

  [[extra code to help with mutual exclusion]]
}
```

We want to focus on the mutual exclusion logic.

Abstraction: forget about details we don't care about at the moment.

```
while(true){
  [[non-critical mutual exclusion code]]
        non-critical operations
  [[waiting to enter critical section code]]
        critical section operations
  [[critical section mutual exclusion code]]
}
```

We want to focus on the mutual exclusion logic.

Abstraction: forget about details we don't care about at the moment.

```
while(true){

  [[non-critical mutual exclusion code]]



  [[waiting to enter critical section]]



  [[critical section mutual exclusion code]]
}
```

```
proc(id, other, turn)
    while(true){
      [[non-critical mutual exclusion code]]



      [[waiting to enter critical section]]



      [[critical section mutual exclusion code]]
    }
```

Define a process, proc, with an id, partial access to the other process, and a shared variable representing whose turn it is.

Labels: n (non-critical), w (waiting), c (critical section).

```
proc(id, other, turn)
    while(true){

      n:     b := TRUE; turn = (id + 1) % 2;



      w:     wait until (!other.b | turn = id)



      c:     b := FALSE;
    }
```

Define a process, proc, with an id, partial access to the other process, and a shared variable representing whose turn it is.

Labels: n (non-critical), w (waiting), c (critical section).

```
proc(id, other, turn)
    while(true){
      n:     b := TRUE; turn = (id + 1) % 2;
      w:     wait until (!other.b | turn = id)
      c:     b := FALSE;
    }
```

Define a process, proc, with an id, partial access to the other process, and a shared variable representing whose turn it is.

Labels: n (non-critical), w (waiting), c (critical section).

```
proc(id, other, turn)
    while(true){
      n:    b := TRUE; turn = (id + 1) % 2;
      w:    wait until (!other.b | turn = id)
      c:    b := FALSE;
    }
```

$$P_0 = \texttt{proc}(0,\ P_1,\ 0)$$
$$P_1 = \texttt{proc}(1,\ P_0,\ 0)$$

Define a process, proc, with an id, partial access to the other process, and a shared variable representing whose turn it is.

Labels: n (non-critical), w (waiting), c (critical section).

```
proc(id, other, turn)
    while(true){
      n:     b := TRUE; turn = (id + 1) % 2;
      w:     wait until (!other.b | turn = id)
      c:     b := FALSE;
    }
```

$$P_0 = \texttt{proc(0, } P_1\texttt{, 0)}$$
$$P_1 = \texttt{proc(1, } P_0\texttt{, 0)}$$

$P_0||P_1$ is a simple reactive system.

Define a process, proc, with an id, partial access to the other process, and a shared variable representing whose turn it is.

Labels: n (non-critical), w (waiting), c (critical section).

```
proc(id, other, turn)
    while(true){
      n:     b := TRUE; turn = (id + 1) % 2;
      w:     wait until (!other.b | turn = id)
      c:     b := FALSE;
    }
```

$$P_0 = \texttt{proc(0, } P_1\texttt{, 0)}$$
$$P_1 = \texttt{proc(1, } P_0\texttt{, 0)}$$

$P_0||P_1$ is a simple reactive system.

Mutual Exclusion Requirement: P_0 and P_1 are never both in the critical section (at line `c`) at the same time.

Idea: Variable *turn* keeps track of whose turn it is. Variable b is only `TRUE` if about to execute line `w` or `c` .

```
proc(id, other, turn)
    while(true){
      n:     b := TRUE; turn = (id + 1) % 2;
      w:     wait until (!other.b | turn = id)
      c:     b := FALSE;
    }
```

$$P_0 = \mathtt{proc(0,}\ P_1\mathtt{,\ 0)}$$
$$P_1 = \mathtt{proc(1,}\ P_0\mathtt{,\ 0)}$$

The state of a single process consists of the values of all variables relevant to that process.

There is an extra "variable", the program counter, typically called pc_i, which keeps track of which line of code is **about** to execute.

```
proc(id, other, turn)
    while(true){
  ➔  n:    b := TRUE; turn = (id + 1) % 2;
     w:    wait until (!other.b | turn = id)
     c:    b := FALSE;
    }
```

$pc_i = n$
line `n` about
to execute.

$$P_0 = \texttt{proc(0, } P_1\texttt{, 0)}$$
$$P_1 = \texttt{proc(1, } P_0\texttt{, 0)}$$

The state of a single process consists of the values of all variables relevant to that process.

There is an extra “variable”, the program counter, typically called pc_i, which keeps track of which line of code is about to execute.

```
proc(id, other, turn)
  while(true){
    n:    b := TRUE; turn = (id + 1) % 2;
    w:    wait until (!other.b | turn = id)
    c:    b := FALSE;
  }
```

$pc_i = w$
line `w` about
to execute.

$$
\begin{aligned}
P_0 &= \texttt{proc(0, } P_1\texttt{, 0)} \\
P_1 &= \texttt{proc(1, } P_0\texttt{, 0)}
\end{aligned}
$$

The state of a single process consists of the values of all variables relevant to that process.

There is an extra "variable", the program counter, typically called pc_i, which keeps track of which line of code is **about** to execute.

```
proc(id, other, turn)
  while(true){
    n:    b := TRUE; turn = (id + 1) % 2;
    w:    wait until (!other.b | turn = id)
--> c:    b := FALSE;
  }
```

$pc_i = c$
line c about
to execute.

$$P_0 = \texttt{proc(0, } P_1\texttt{, 0)}$$
$$P_1 = \texttt{proc(1, } P_0\texttt{, 0)}$$

The state of a single process consists of the values of all variables relevant to that process.

There is an extra "variable", the program counter, typically called pc_i, which keeps track of which line of code is **about** to execute.

```
proc(id, other, turn)
    while(true){
      n:     b := TRUE; turn = (id + 1) % 2;
      w:     wait until (!other.b | turn = id)
      c:     b := FALSE;
    }
```

$$P_0 = \texttt{proc(0, } P_1\texttt{, 0)}$$
$$P_1 = \texttt{proc(1, } P_0\texttt{, 0)}$$

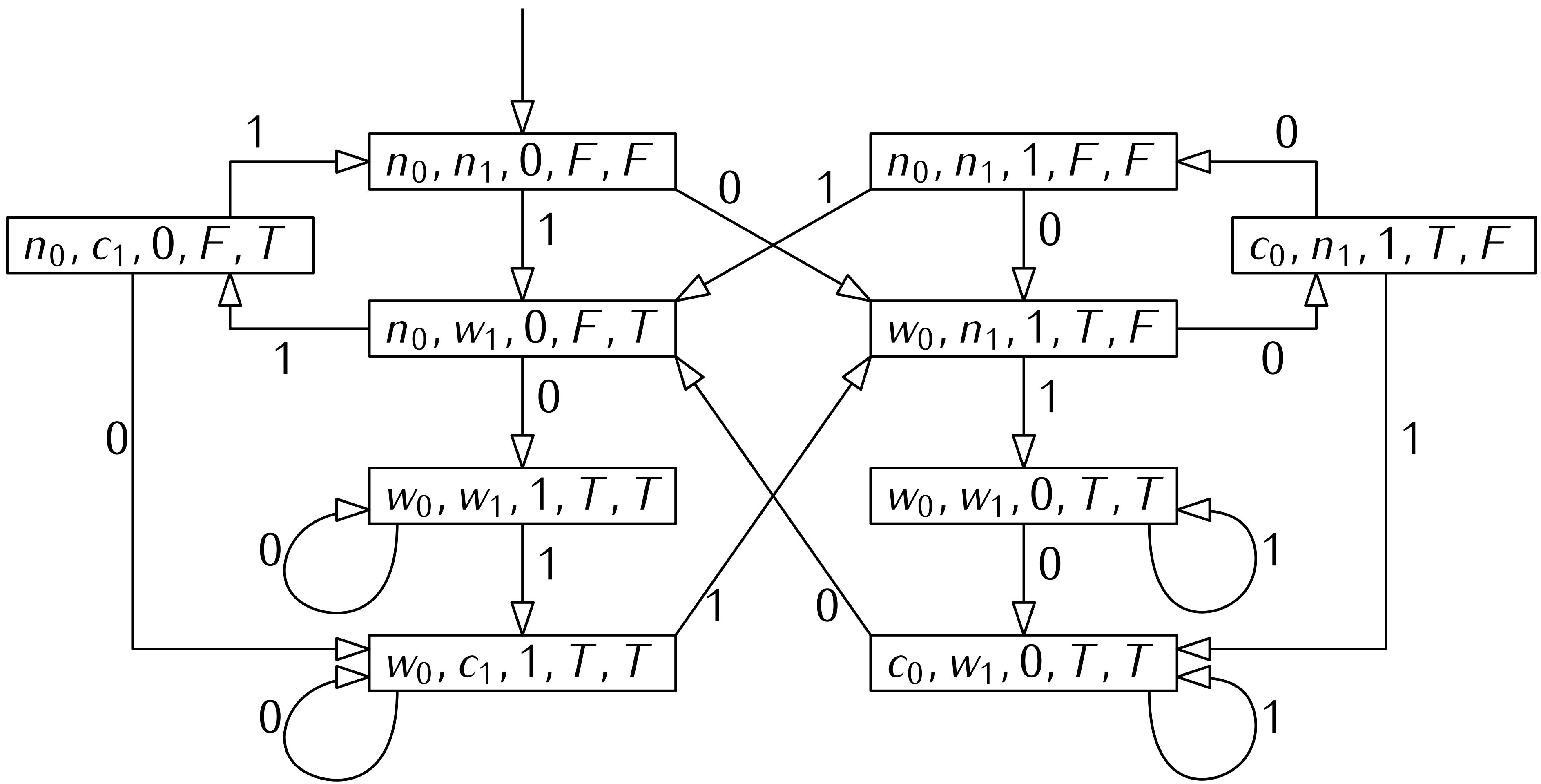
The state of the combined system, $P_0||P_1$ is given by the values of all variables together. The state of this system is completely determined by the tuple

$$(pc_0, pc_1, turn, b_0, b_1)$$

In class activity:

Building intution for transition systems using mutual exclusion as a case study.

Transition system for $P_0||P_1$ from in-class activity.



Transition Systems

A transition system $\mathcal{M} = (S, I, \rightarrow, L)$ is a set of states S and a set of initial states I, along with a transition relation $\rightarrow$ and labelling function L.

The transition relation $\rightarrow$ is equivalent to a set of directed graph edges, with the states as nodes.

For example, $((n_0, n_1, 0, F, F), (n_0, w_1, 0, F, T)) \in\rightarrow$

Alternatively, we can write
$(n_0, n_1, 0, F, F) \rightarrow (n_0, w_1, 0, F, T)$.

Important assumption: no dead states. Every state has an outgoing transition, even if only to itself.

Transition Systems, execution paths

A path in a transition system $\mathcal{M} = (S, I, \rightarrow, L)$ is an infinite sequence of states $s_1, s_2, s_3, \ldots$ such that $s_1 \in I$ and for every $i \geq 1$, $s_i \rightarrow s_{i+1}$

Transition Systems, execution paths

A path in a transition system $\mathcal{M} = (S, I, \rightarrow, L)$ is an infinite sequence of states $s_1, s_2, s_3, \ldots$ such that $s_1 \in I$ and for every $i \geq 1$, $s_i \rightarrow s_{i+1}$

For example, one path from our two-process mutual exclusion transition diagram:

$((n_0, n_1, 0, F, F), (n_0, w_1, 0, F, T), (n_0, c_1, 0, F, T))^\omega$

Transition Systems, execution paths

A path in a transition system $\mathcal{M} = (S, I, \rightarrow, L)$ is an infinite sequence of states $s_1, s_2, s_3, \ldots$ such that $s_1 \in I$ and for every $i \geq 1$, $s_i \rightarrow s_{i+1}$

For example, one path from our two-process mutual exclusion transition diagram:

$((n_0, n_1, 0, F, F), (n_0, w_1, 0, F, T), (n_0, c_1, 0, F, T))^\omega$

We will use the symbol π for paths.
We write $\pi = s_1, s_2, s_3 \ldots$
We write π^i to indicate the ith suffix of π.
e.g. $\pi^3 = s_3, s_4, s_5 \ldots$

Transition System Example

$$S = \{0, 1, 2\} \qquad I = \{0\} \qquad AP = \{p, q, r\}$$

$$\rightarrow = \{(0, 1), (1, 0), (0, 2), (1, 2)\}$$

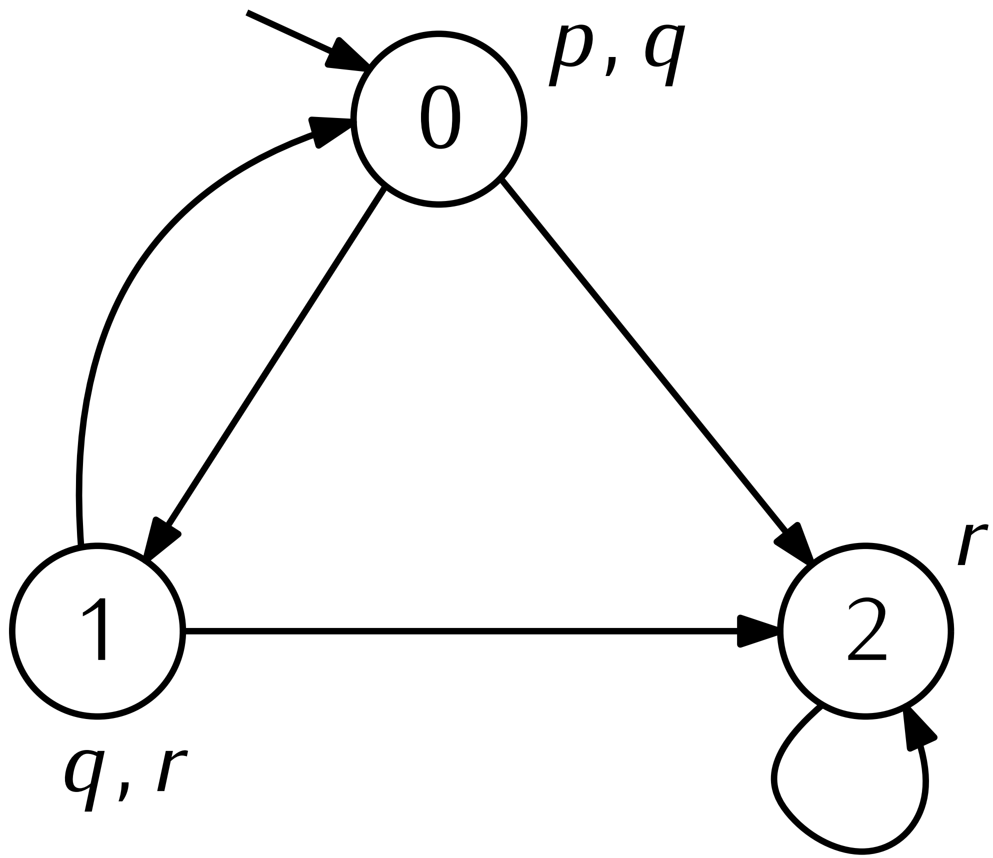
$$L(0) = \{p, q\} \qquad L(1) = \{q, r\} \qquad L(2) = \{r\}$$

Transition System Example

$$S = \{0, 1, 2\} \qquad I = \{0\} \qquad AP = \{p, q, r\}$$

$$\rightarrow = \{(0, 1), (1, 0), (0, 2), (1, 2)\}$$

$$L(0) = \{p, q\} \qquad L(1) = \{q, r\} \qquad L(2) = \{r\}$$



Remember the big picture