Concurrent Systems

3D4 ⟷ CS2016

Operating Systems

Andrew Butterfield
ORI.G39, Andrew.Butterfield@scss.tcd.ie

Trinity College Dublin
Coláiste na Tríonóide, Baile Átha Cliath
The University of Dublin

with thanks to Mike Brady

Abstraction of Concurrent Programming

- A *concurrent program* is a finite set of [sequential] *processes*.
- A process is written using a *finite set* of *atomic statements*.
- Concurrent program execution is modelled as proceeding by executing a sequence of the atomic statements obtained by *arbitrarily interleaving* the atomic statements of the processes.
- A *computation* [a.k.a. a *scenario*] is one particular execution sequence.

Atomicity

- We assume that if two operations s1 and s2 really happen at the same time, it's the same as if the two operations happened in either order.
- E.g. simultaneous writes to the same memory locations:

Sample	
integer g ← 0;	
p	q
p1: g ← 2;	q1: g ← 1

- We assume that the result will be that g will be 2 or 1 after this program, not, for example, 3.

Interleaving

- We model a scenario as an arbitrary interleaving of atomic statements, which is somewhat unrealistic.
- For a *concurrent* system, that's OK, it happens anyway.
- For a *parallel* shared memory system, it's OK so long as the previous idea of atomicity holds at the lowest level.
- For a *distributed* system, it's OK if you look at it from an individual node's POV, because either it is executing one of its own statements, or it is sending or receiving a message.
  - Thus *any* interleaving can be used, so long as a message is sent before it is received.

Level of Atomicity

- The level of atomicity can affect the correctness of a program.

Example: Atomic Assignment Statements	
integer n ← 0;	
p	q
p1: n ← n+1;	q1: n ← n+1;

process p	process q	n
p1: n ← n+1;	q1: n ← n+1;	0
(end)	**q1: n ← n+1;**	1
(end)	(end)	2

process p	process q	n
p1: n ← n+1;	**q1: n ← n+1;**	0
p1: n ← n+1;	(end)	1
(end)	(end)	2

Different Level of Atomicity

Example: Assignment Statements with one Global Reference	
integer n ← 0;	
p	q
integer temp	integer temp
p1: temp ← n	q1: temp ← n
p2: n ← temp + 1	q2: n ← temp + 1

Alternative Scenarios

process p	process q	n	p.temp	q.temp
p1: temp ← n	q1: temp ← n	0	?	?
p2: n ← temp+1	q1: temp ← n	0	0	?
(end)	**q1: temp ← n**	1		?
(end)	**q2: n ← temp+1**	1		1
(end)	(end)	2		

process p	process q	n	p.temp	q.temp
p1: temp ← n	q1: temp ← n	0	?	?
p2: n ← temp+1	**q1: temp ← n**	0	0	?
p2: n ← temp+1	q2: n ← temp+1	0	0	0
(end)	**q2: n ← temp+1**	1		0
(end)	(end)	1		

Concurrent Counting Algorithm

Example: Concurrent Counting Algorithm	
integer n ← 0;	
p	**q**
integer temp	integer temp
p1: do 10 times	q1: do 10 times
p2: temp ← n	q2: temp ← n
p3: n ← temp + 1	q3: n ← temp + 1

- Increments a global variable *n* 20 times, thus *n* should be 20 after execution.
- But, the program is faulty.
  - Proof: construct a scenario where *n* is 2 afterwards.
- Wouldn't it be nice to get a program to do this?

Atomicity & Correctness

- Thus, the level of atomicity specified affects the correctness of a program
  - We will assume that:
    - assignment statements and
    - condition statement evaluations
- are atomic

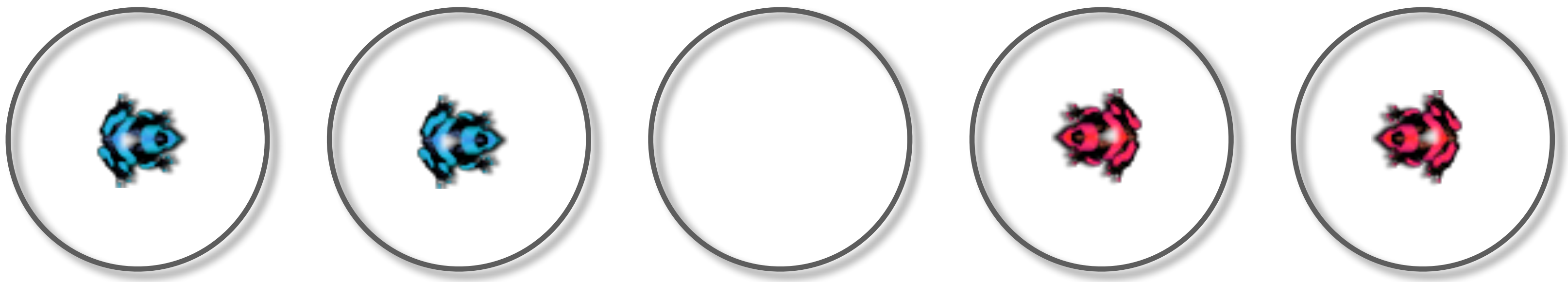
State Diagrams for Processes

- A *state* is defined by a tuple of
  - for each process, the label of the statement available for execution.
  - for each variable, its value.
- Q: What is the maximum number of possible states in such a state diagram?

State Diagrams and Scenarios

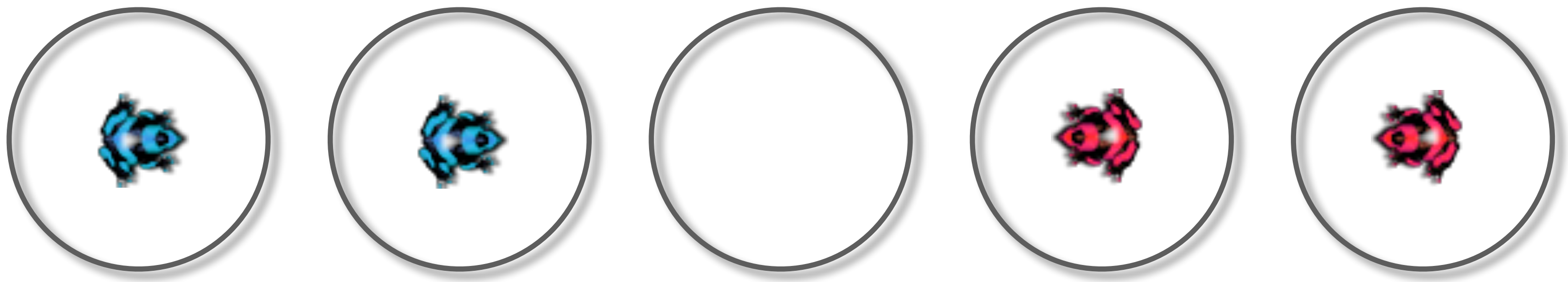
- We could describe all possible ways a program can execute with a state diagram.
  - There is a transition between s_1 and s_2 ("s_1:s_2") if executing a statement in s_1 changes the state to s_2.
  - A state diagram is generated inductively from the starting state.
    - If $\exists$ s_1 and a transition s_1:s_2, then $\exists$ s_2 and a directed arc from s_1:s_2
- Two states are identical if they have the same variable values and the same directed arcs leaving them.
- A *scenario* is one path through the state diagram.

Example — Jumping Frogs

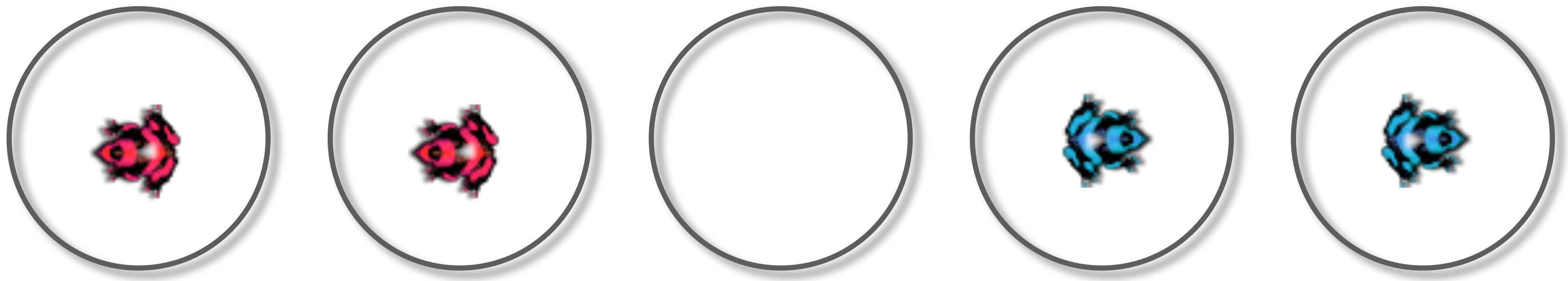


- A frog can move to an adjacent stone if it’s vacant.
- A frog can hop over an adjacent stone to the next one if that one is vacant.
- No other moves are possible.

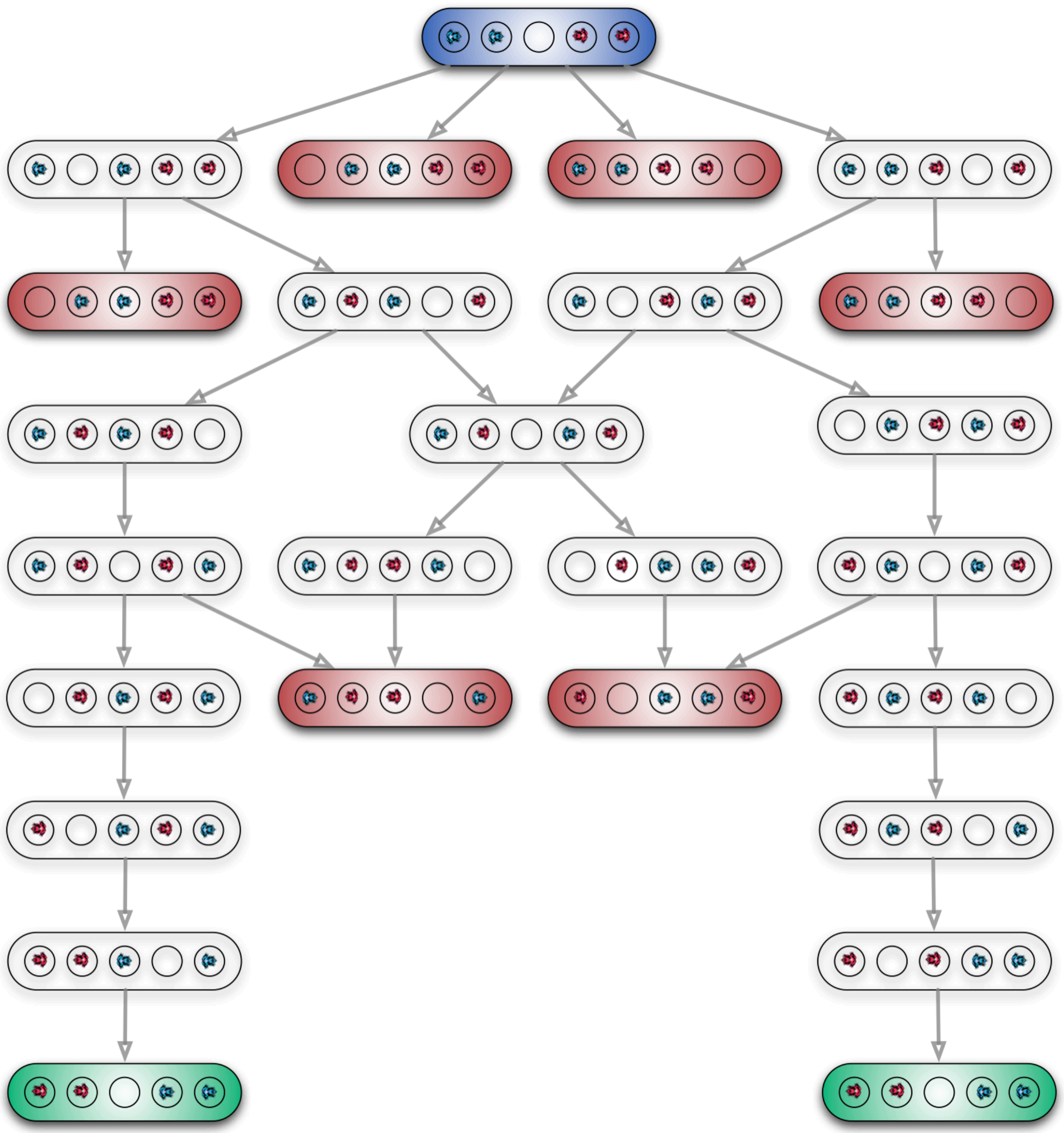
If the frogs can only move "forwards", can we:



move from above to below?

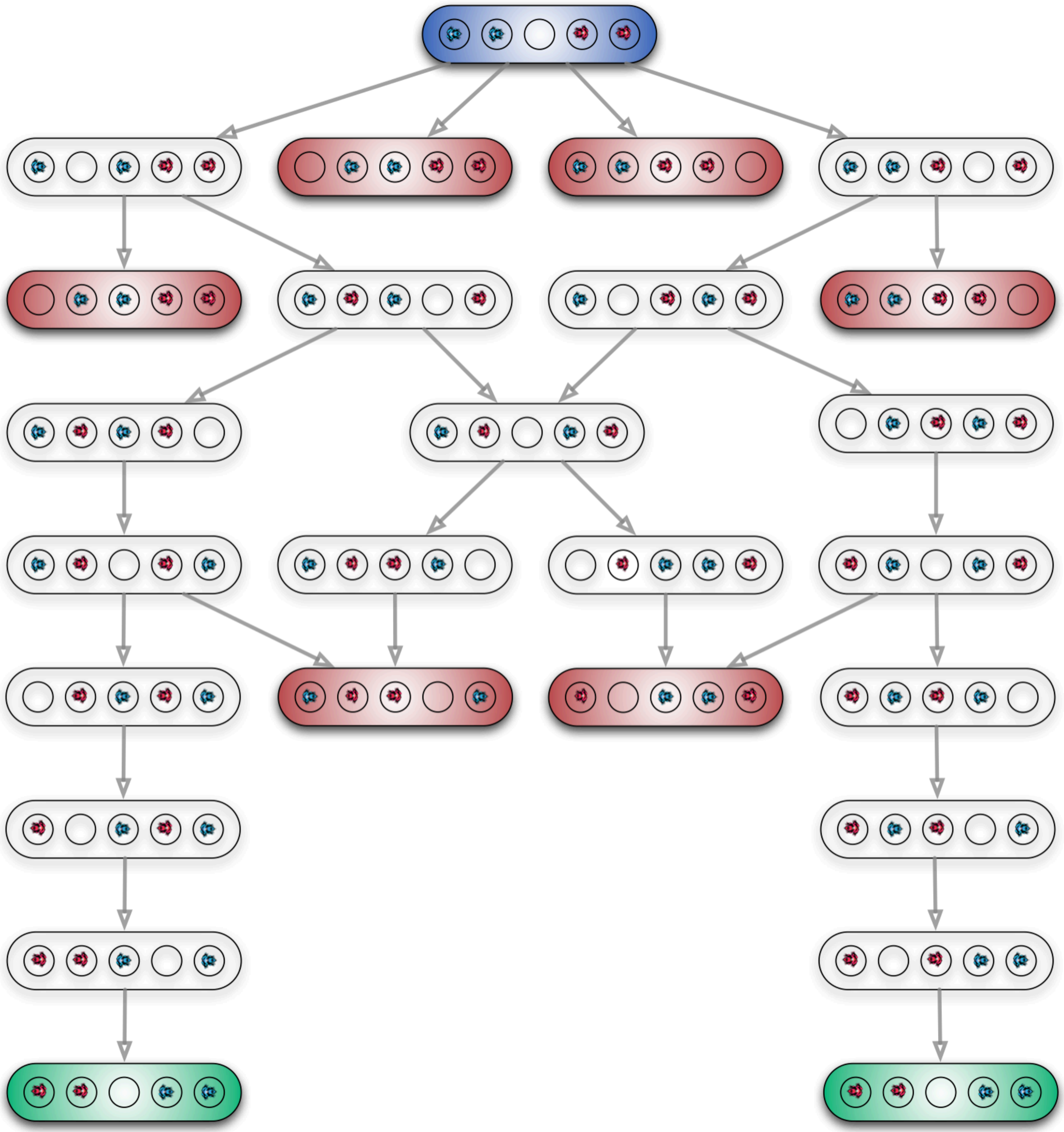


Solution Graph



- So, we have a *finite state-transition diagram* of a *finite state machine (FSM)* as a *complete description* of the behaviour of the four frogs, operating concurrently, no matter what they do according to the rules.

- By examining the FSM, we can state properties as definitely holding, i.e. we can prove properties of the system being modelled.



Solution Graph

- The solution graph makes it clear that this concurrent system—of four frogs that share certain resources—can experience *deadlock*.
- *Deadlock* occurs when the system arrives in a state from which it can not make any transitions (and which is not a desired end-state.)
- *Livelock* (not possible in this system) is when the system can traverse a sequence of states indefinitely without making progress towards a desired end state.
  - If we allow frogs to step back, provided the space immediately behind them is empty, then livelock is possible.