Distributed Systems

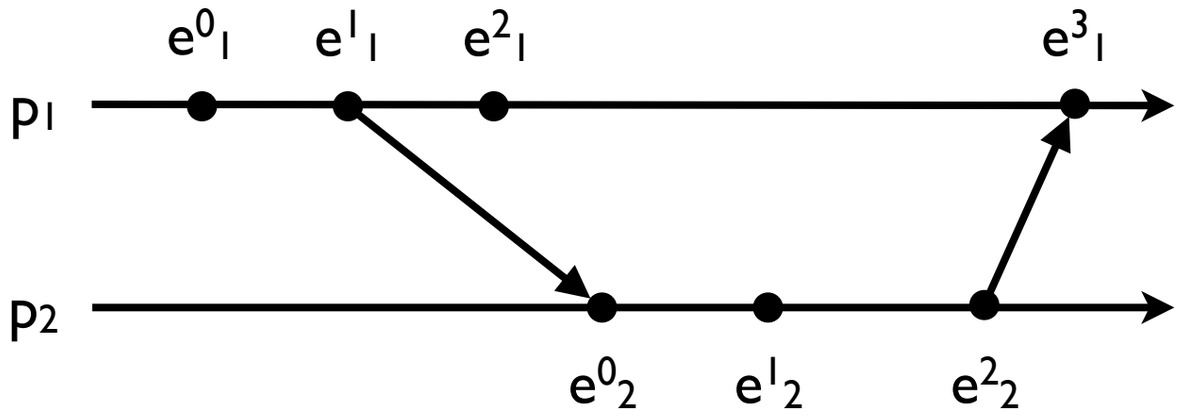
Rik Sarkar
James Cheney

Global State & Distributed Debugging
February 3, 2014

Global State: Consistent Cuts

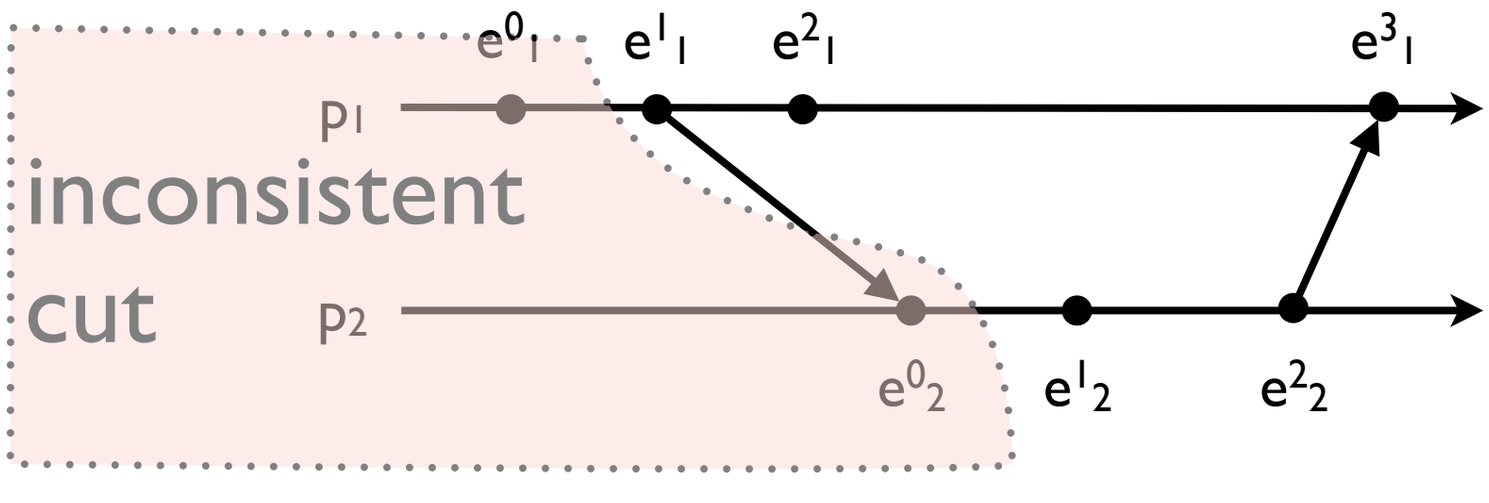
- The global state is the combination of all process states and the states of the communication channels at an instant in time
  - So, if we had synchronized clocks, we could agree on a time for each process to record its state
- Since we cannot "stop time" to observe the *actual* global state, we attempt to find *possible* global state(s)
- A **cut** is a collection of prefix of the (combined) histories of the processes
  - partitioning all events into those occurring "before" and "after" the cut
- The goal is to assemble a meaningful global state from the the local states of processes
  - recorded at (possibly) different but concurrent times

Consistent Cuts



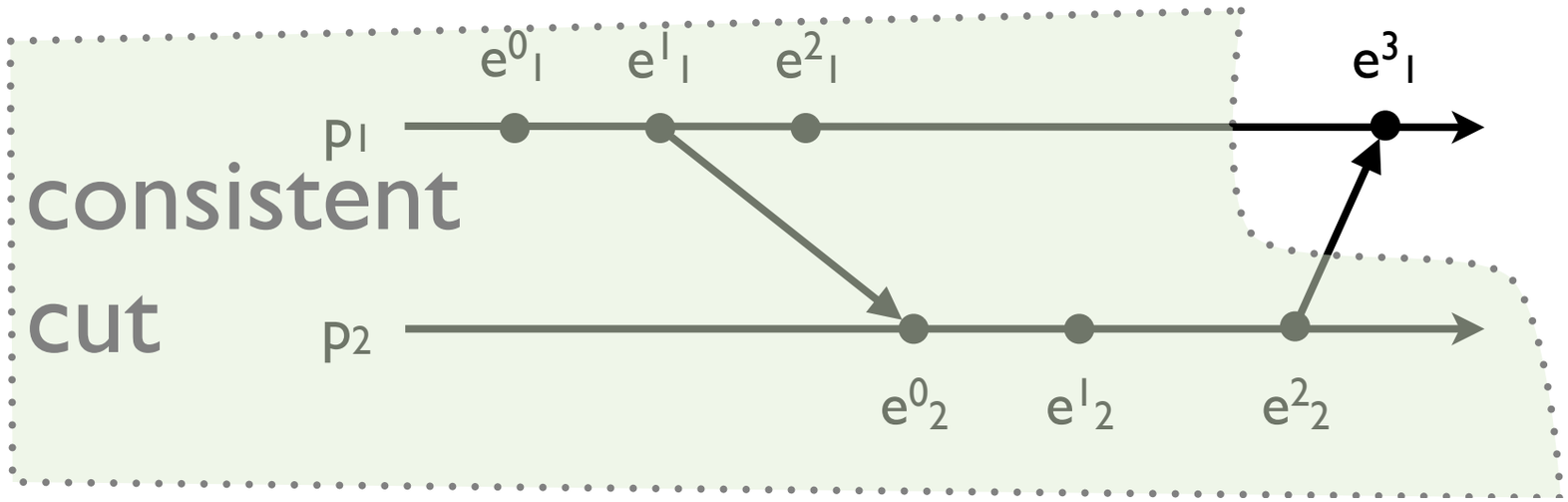
- A **consistent cut** is one which does not violate the happens-before relation →
- If $e_1 \rightarrow e_2$ and e_2 is in cut then e_1 is in cut. That is:
  - both e_1 and e_2 are in the cut or
  - both e_1 and e_2 are after the cut or
  - e_1 is in the cut and e_2 is after the cut
  - but not e_1 is after the cut and e_2 is in the cut

Consistent Cuts



- A **consistent cut** is one which does not violate the happens-before relation →
- If $e_1 \rightarrow e_2$ and e_2 is in cut then e_1 is in cut. That is:
  - both e_1 and e_2 are in the cut or
  - both e_1 and e_2 are after the cut or
  - e_1 is in the cut and e_2 is after the cut
  - but not e_1 is after the cut and e_2 is in the cut

Consistent Cuts



- A **consistent cut** is one which does not violate the happens-before relation →
- If $e_1 \rightarrow e_2$ and e_2 is in cut then e_1 is in cut. That is:
  - both e_1 and e_2 are in the cut or
  - both e_1 and e_2 are after the cut or
  - e_1 is in the cut and e_2 is after the cut
  - but not e_1 is after the cut and e_2 is in the cut

Runs and Linearizations

- A *consistent global state* is one which corresponds to a consistent cut
  - Corresponds to a *frontier*: a set of *n* mutually concurrent events, one from each process $P_1 \ldots P_n$
  - $C = \{e_1,\ldots,e_n\}$ with $e_i \parallel e_j$ whenever $i \neq j$
- A "run" is a total ordering of all events in a global history which is consistent with the local history of each process
- A "linearization" is a total ordering of all events in the global history which is consistent with the happens-before relation →
  - So all linearizations are also runs
- Not all runs pass through consistent global states but all *linearizations* pass only through consistent global states

Global State: Safety and Liveness

- When we attempt to examine the global state, we are often concerned with whether or not a property holds
- Some properties, *B*, are "bad" properties we hope **never** hold
- and some properties, *G*, are "good" properties we hope **always** hold
- **Safety** property: a bad property *B* does not hold for any reachable state
- **Liveness** property: that a good property *G* always (eventually) holds for all reachable states

Global State: Stable and Unstable properties

- Some properties we wish to establish are **stable** properties
  - Such properties may never become true, but once they do they remain true forever
- Consider our four example problems:
  - Garbage is stable: once an object has no valid references (at a process or in transit) will never have any valid references
  - Deadlock is stable: once a set of processes are deadlocked they will always be deadlocked without external intervention
  - Termination is stable: once a set of processes have terminated they will remain terminated without external intervention
  - Debugging is not really a property but the properties we may look for whilst debugging are typically non-stable

Global State: Chandy and Lamport

- Many useful properties are stable
- Such properties may never become true, but once they do they remain true
- Our four example properties:
  - Garbage is *stable*: once an object has no valid references (at a process or in transit) will never have any valid references
  - Deadlock is *stable*: once a set of processes are deadlocked they will always be deadlocked without external intervention
  - Termination is *stable*: once a set of processes have terminated they will remain terminated without external intervention
  - Debugging is not really a property but the properties we may look for whilst debugging are likely *non-stable*

Assumptions

Assumptions

- There is a path between any two pairs of processes, in both directions
- Any process may initiate a global snapshot at any time
- The processes may continue their execution and send/ receive normal messages whilst the snapshot takes place

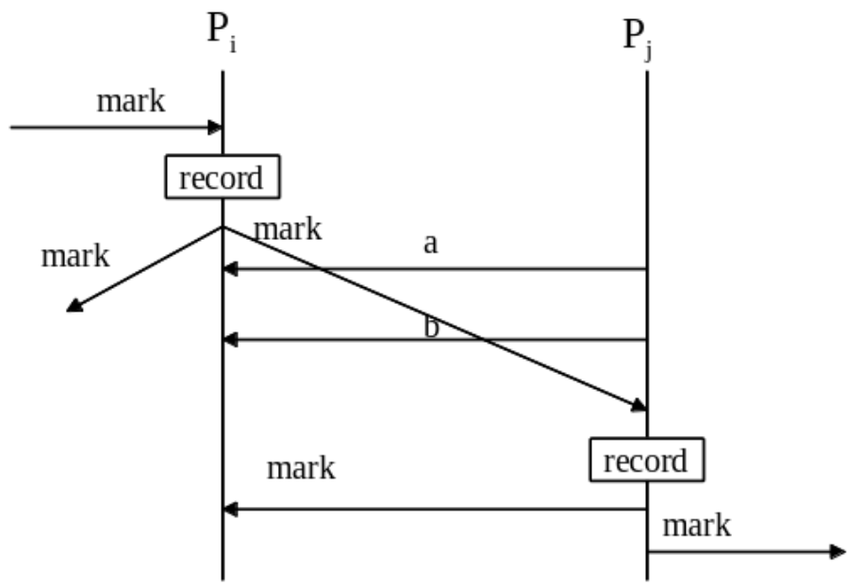
Assumptions

- There is a path between any two pairs of processes, in both directions
- Any process may initiate a global snapshot at any time
- The processes may continue their execution and send/ receive normal messages whilst the snapshot takes place
- Neither channels nor processes fail
- Communication is reliable: every message that is sent arrives at its destination exactly once (no repeats)
- Channels are unidirectional and provide FIFO-ordered message delivery.

Algorithm: Sender

Sending rule for process P_i

1. After P_i receives **marker** and has recorded its state:
2. P_i sends a **marker** message for **each** outgoing channel c
   - **before** it sends any other messages over c
   - **including** the channel that P_i received mark from

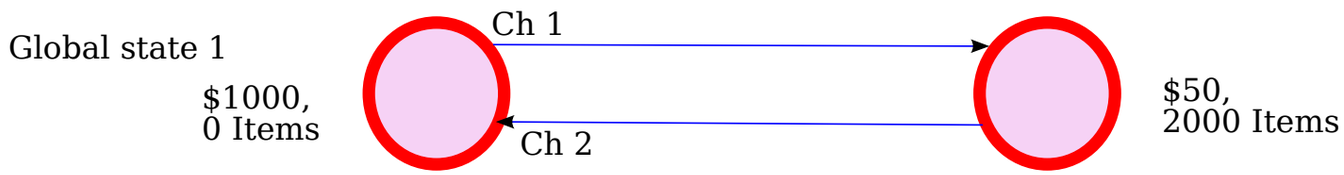


Algorithm: Receiver

Receiving rule for process P_i

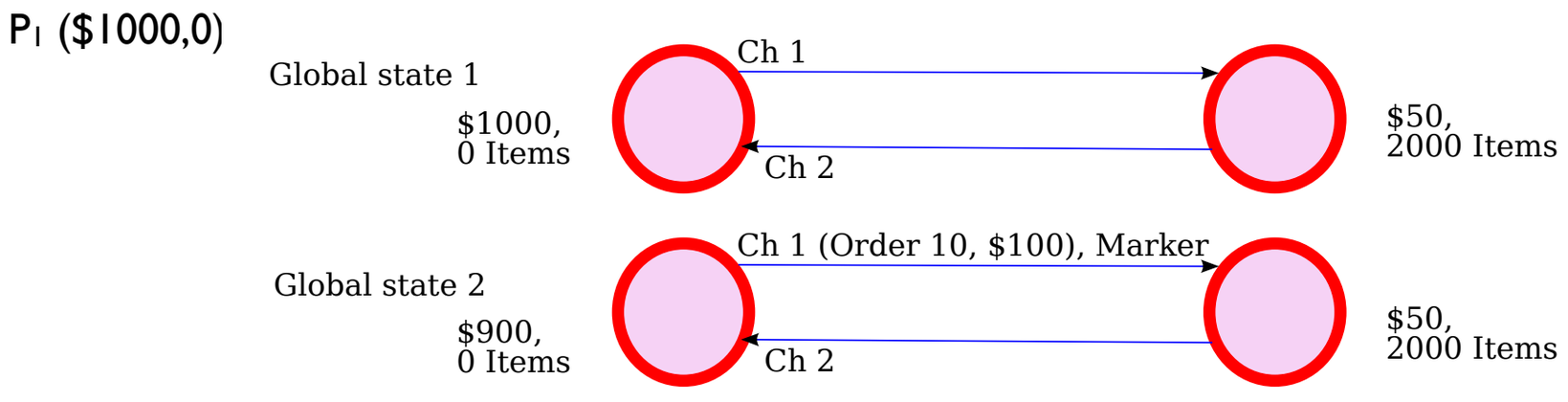
1. On receipt of a **marker** message over channel *c*:
2. **if** P_i has not yet recorded state:
   - record process state now
   - record the state of *c* as the empty set
   - turn on recording of messages arriving on all other channels
3. **else**
   - record the state of *c* as the set of messages it has recorded since P_i first recorded its state

Example



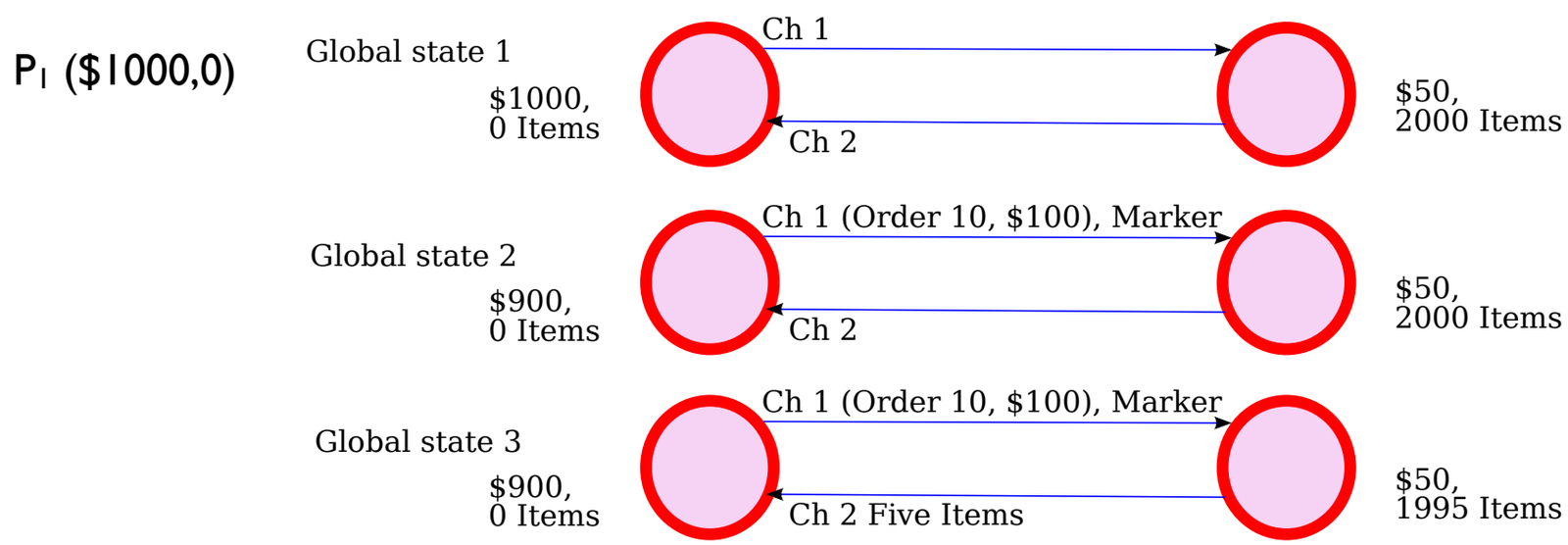
- We begin in this global state, where both channels are empty, the states of the processes are as shown, but we say nothing about what has gone before.

Example



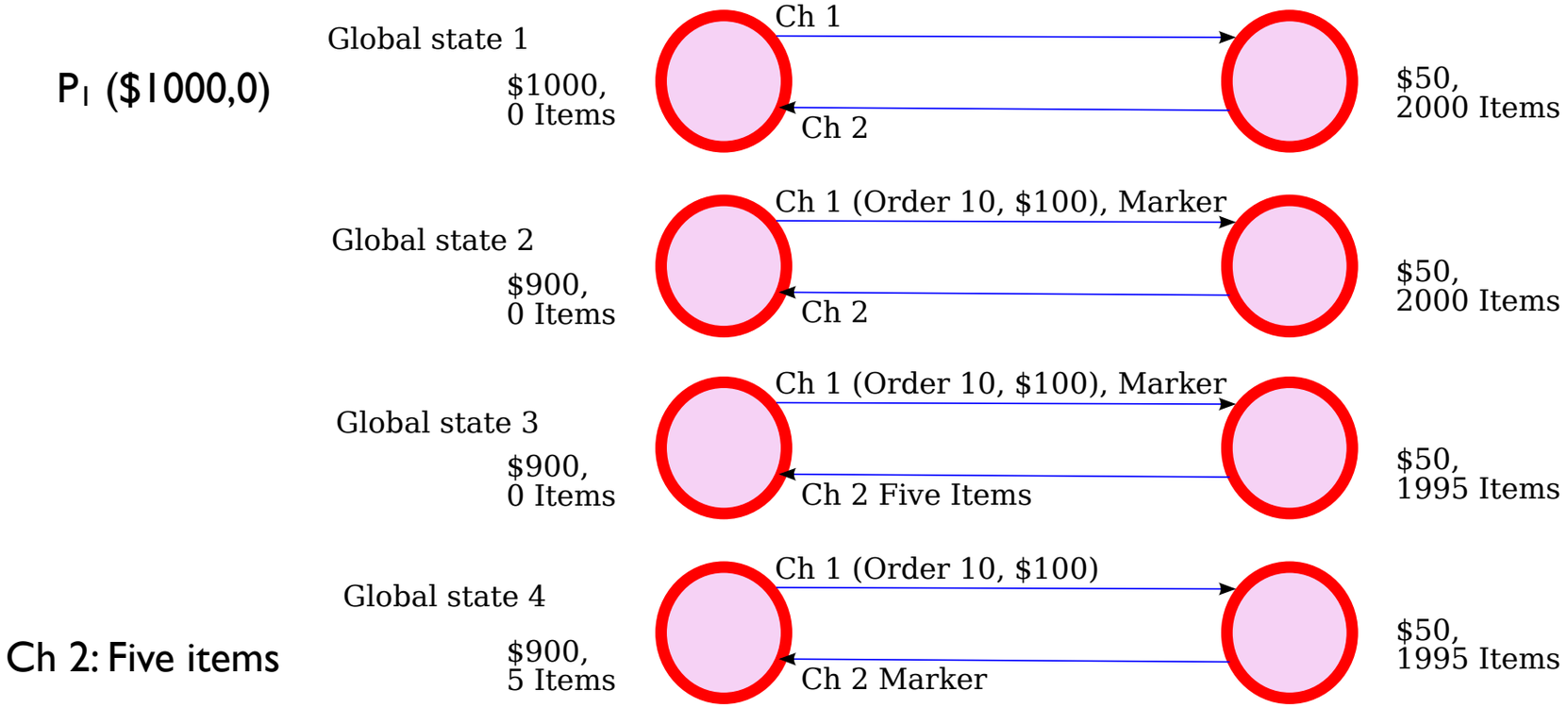
- P_1 decides to begin the snapshot algorithm and sends a Marker message over channel 1 to P_2.
- It then decides to send a request for 10 items at $10 each.

Example



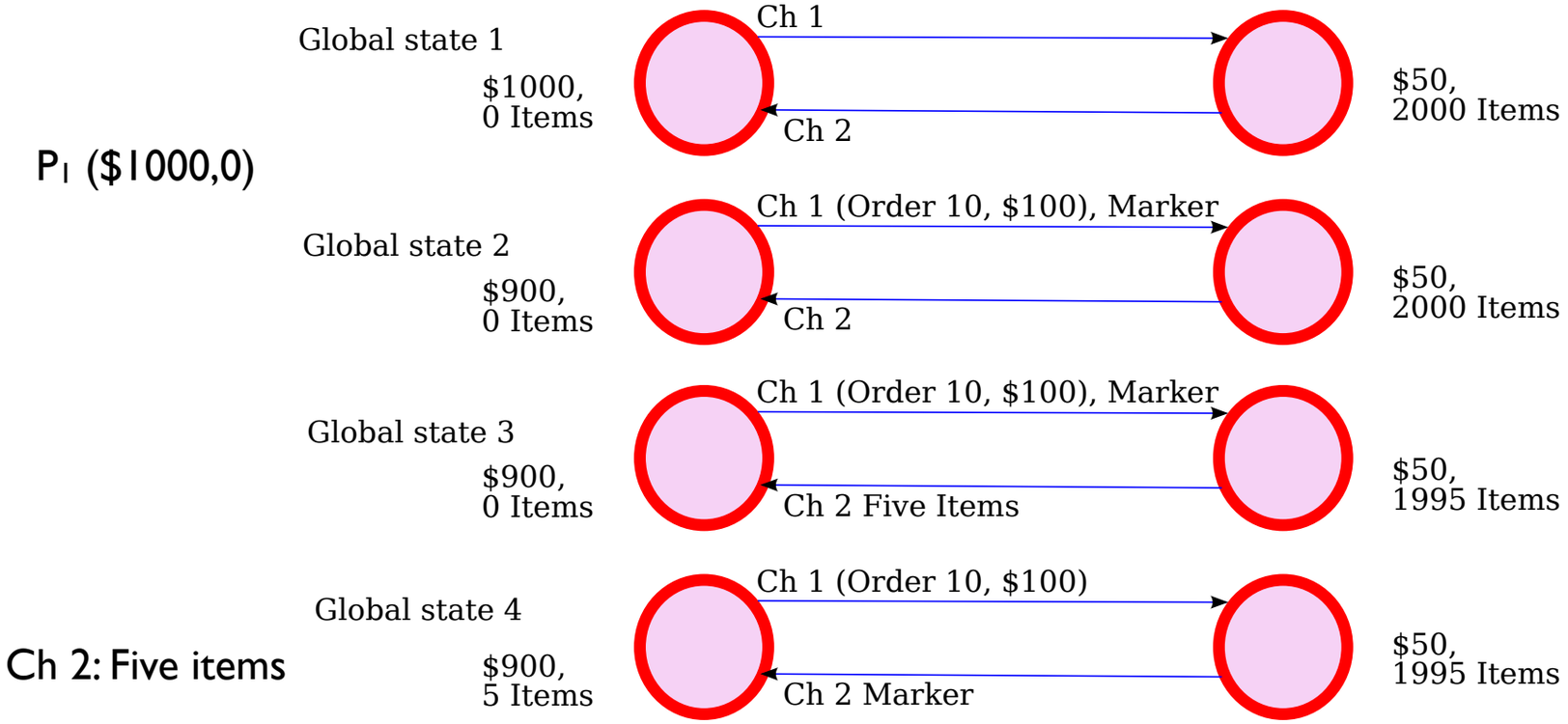
- Meanwhile, P_2 responds to an earlier request and sends 5 items to P_1 over channel 2.

Example



- P_2 receives the Marker message, records its state and sends P_1 a Marker message over channel 2.
- When P_1 receives this Marker message it records the state of channel 2 as containing the 5 items it has received since recording its own state.

Example

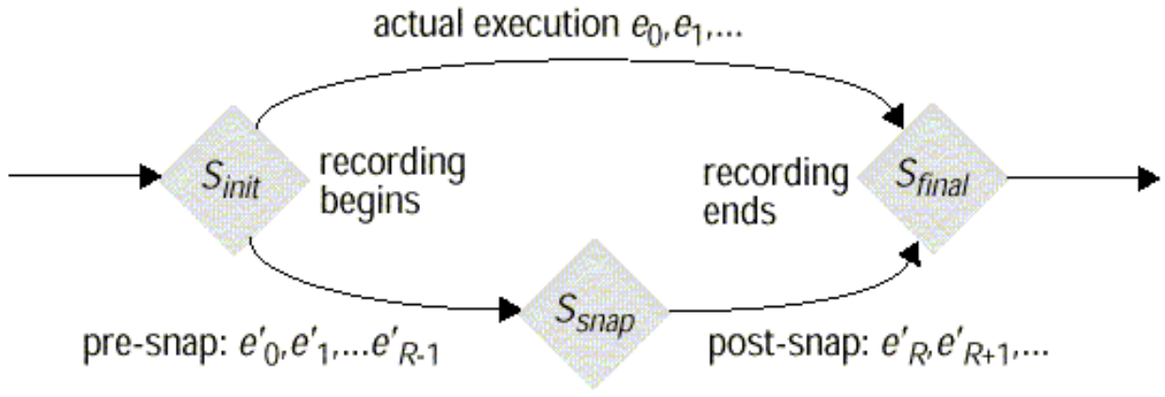


Notice that (in this story) *P2* didn't send "Five items" until **after** *P1* sent marker!
But there is no way to know this - the events are concurrent

Reachability

- The cut found by the Chandy and Lamport algorithm is always a **consistent** cut
- This means that the global state which is characterized by the algorithm is a consistent (possible) global state
  - Though it may not be one that ever "actually" occurred
- However, we can define a **reachability** relationship:
  - between the initial, observed and final global states when the algorithm is run
- Assume that the events actually occurred in a global order $H = e_1, e_2 \ldots$
  - Let S_{init} be the global state immediately before the algorithm commences and S_{final} be the global state immediately after it terminates. Finally S_{snap} is the recorded global state
  - We can find a permutation of H called H' which contains all three states: S_{init}, S_{snap} and S_{final}
- Does not break the happens-before relationship on the events in H

Chandy and Lamport: Reachability



- It may be that there are two events in H, e_n and e_{n+1} such that e_n is a post-snap event and e_{n+1} is a pre-snap event
- However we can swap the order of e_n and e_{n+1} since it cannot be that $e_n \rightarrow e_{n+1}$
- We continue to swap adjacent pairs of events until all pre-snap events are ordered before all post-snap events. This gives us the the linearization H'
- The reachability property of the snapshot algorithm is useful for recording *stable* properties
  - true in $S_{snap} \Rightarrow$ true in S_{init}; false in $S_{snap} \Rightarrow$ false in S_{init}
- However any *non-stable* predicate which is true in the snapshot may or may not be true in any other state

Chandy and Lamport: Use Cases

- No work which depends upon the global state is done until the snapshot has been gathered
- Algorithm is therefore useful for:
  - Evaluating after infrequent changes
  - Stable properties, since the property that you detect to have been true when the snapshot was taken will still be true once the snapshot has been gathered
  - Properties that have a single yes/no answer (Garbage Collection) rather than a range of increasingly appropriate answers (e.g. routing)
  - Properties that need not be acted upon immediately, again for example garbage collection.

Summary

- We looked at Chandy and Lamport's algorithm for recording a global snapshot of the system
- We defined a notion of **reachability** showing that the snapshot can be obtained by permuting concurrent events
- Thus, the snapshot is useful for detecting stable properties

Distributed debugging

- Distributed debugging was the application of our four example applications that stood out for being concerned with **unstable** properties
- This is a problem for our global snap-shot technique since its main usefulness is derived from our reachability relation which in turn means little for a non-stable property
- Distributed debugging is in a sense a combination of logical/vector clocks and global snapshots

Example Non-Stable Condition

- Suppose we are implementing an online poker game
- There is a process representing each player and one representing the pot in the centre of the table
- Players can “send chips” to the pot, and once winners have been decided the pot may send chips back to some of the players.
- We wish to make sure that the total amount of chips in the game never exceeds the initial amount
- It may be less than the initial amount since some chips may be in transit between a player and the centre pot.
- But it cannot be more than the initial amount.

Distributed debugging

- Suppose that we have a history *H* of events $e_1,...,e_n$
- $H = (e_1,...,e_n)$ is the true order of events as they actually occurred in our system
- Recall that a *run* is any ordering of those events in which each event occurs exactly once
- A *linearization* is a *consistent* run
  - A consistent run is one in which the "happens-before" relation is satisfied
  - If $e_i \rightarrow e_j$ then any linearisation (or consistent run) will order e_i before e_j.
  - So all linearisations only pass through consistent states

The possibly relation

- Any linearization *Lin* of our history of events *H* must pass through only consistent states
- A property *P* that is true in any state through which *Lin* passes, was **possibly true** at some global state through which *H* passed
- If this is the case for some property *p* and some linearisation we say *possibly(p)*
- Note: suppose we had taken a global snapshot during the set of events *H* to determine if the property *p* was true and determined that it was: *Snap(p)* evaluates to true.
  - This would imply that *p* was possible: Snap(p) ⟹ possibly(p)
- However the reverse is not true:
  - possibly(p) ⇏ Snap(p)

The definitely relation

- The sister relation to the possibly relation is the definitely relation
- This states that for any linearization Lin of *H*, *Lin* must pass through some consistent global state *S* for which the candidate property is true
- Since *H* is a linearization of itself, then the candidate property was certainly true at some point in the history of events.
  More formally:
- The statement *possibly(p)* means that there is a consistent global state *S* through which at least one linearization of *H* passes such that *S(p)* is true.
- The statement *definitely(p)* means that for all linearisations *L* of *H*, there is a consistent global state *S* through which *L* passes such that *S(p)* is true

Possibly vs Definitely

- If *p* is impossible then ¬*p* definitely holds
  - ¬*(possibly(p))* ⟹ *definitely(¬p)*
- But, from *definitely(¬p)* we cannot conclude ¬*(possibly(p))*.
  - *definitely(¬p)* means that there is at least one state in all linearizations of *H* such that *p* is not true.
  - ¬*(possibly(p))* however would require that ¬*(p)* was true in all states in all linearizations
- Also: *definitely(p)* and *definitely(¬p)* may be true simultaneously but *possibly(p)* and ¬*(possibly(p))* cannot.
  - consider blinking light, *p* = "light is on"

Distributed Debugging: Basic Outline

- The processes must all send messages recording their local state to a **master** process
  - timestamping each state with vector clock value
- The master process **collates** these and extracts the consistent global states
  - using timestamps to reconstruct happens-before order
- From this information the *possibly(p)* and *definitely(p)* relations may be computed.

Collecting the Local States

- Each process sends their initial state to the master process in a state message and thereafter periodically send their local state.
- Preparing and sending these messages may delay the normal operation of the distributed system but does not otherwise affect it
  - so debugging may be turned on and off.
- "Periodically" is better defined in terms of the predicate for which we are debugging.
  - We only need to send state messages to the master process initially and whenever our local state changes
  - We can further restrict to local state changes that affect the predicate in question.
- We can concurrently check for separate predicates as well by marking our state messages appropriately.

State Messages, Timestamps

- The individual processes send states to the master
- Each state message is timestamped with the Vector clock value at the local process sending the state message: $(s_i , V(s_i))$
- If $S = \{(s_1,V(s_1)),...(s_n,V(s_n))\}$ is a timestamped set of state messages received by the master process
- Then S is a consistent global state iff:
  - $V_i[i] \geq V_j[i] \; \forall i, j \in \{1, ..., N\}$

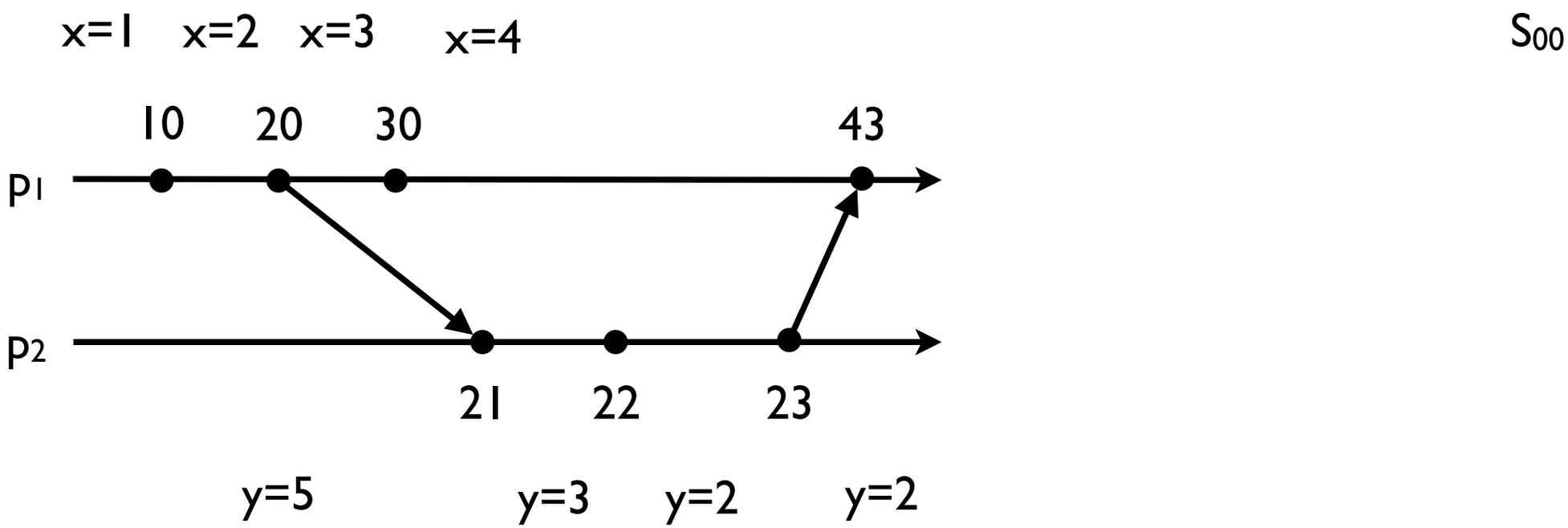
Assembled Consistent Global States

- *S* is a consistent global state iff:
  - $V_i[i] \geq V_j[i]\ \forall i, j \in \{1, \ldots, N\}$
- This says that the number of P_i's events known at P_j when it sent s_j is no more than the number of events that had occurred at P_i when it sent s_i.
- In other words:
  - if the state of one process depends upon another (according to happened-before ordering),
  - then the global state also encompasses the state upon which it depends.

Simple case

- Imagine the simplest case of 2 communicating processes.
- A plausible global state is S_{xy}
- The subscripts x and y refer to the number of events which have occurred at the particular process.
- The "level" of a given state is $x + y$, which is number of events which have occurred globally to give rise to the particular global state S.

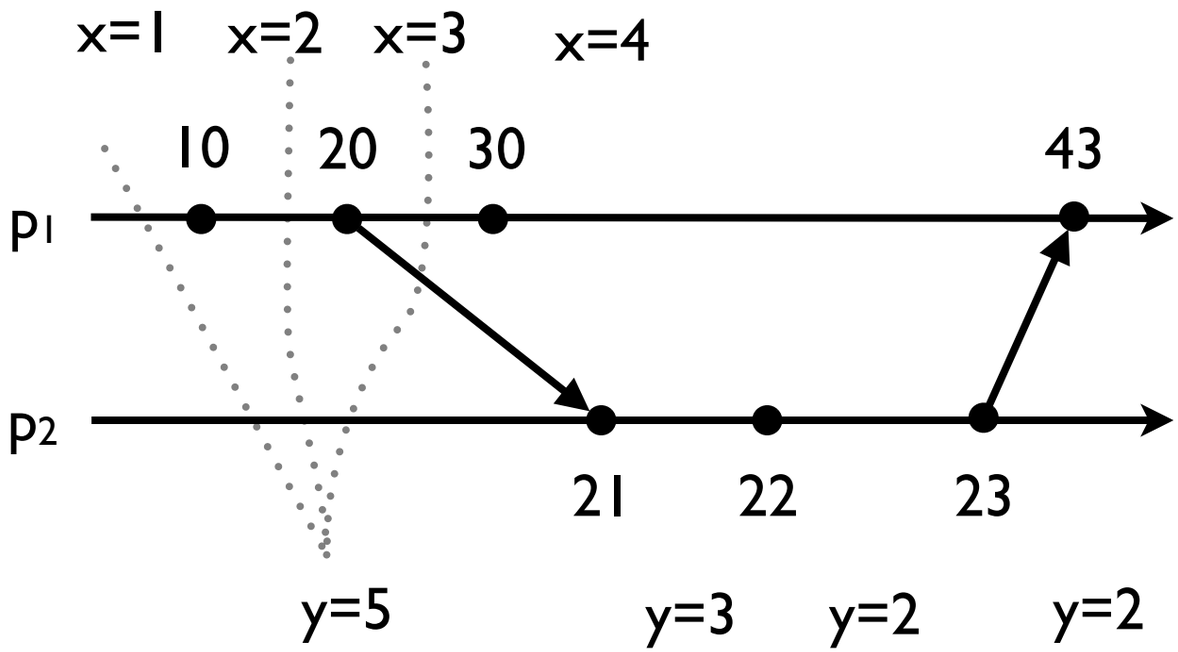
Assembling consistent global states

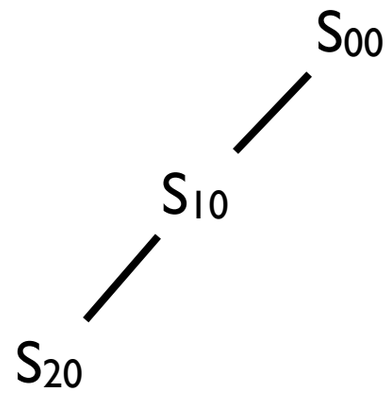


Note: the "global states" include the current process local states in each consistent cut (variables x, y)

Suppose property *p(x,y)* = "*x* = *y*"

Assembling consistent global states

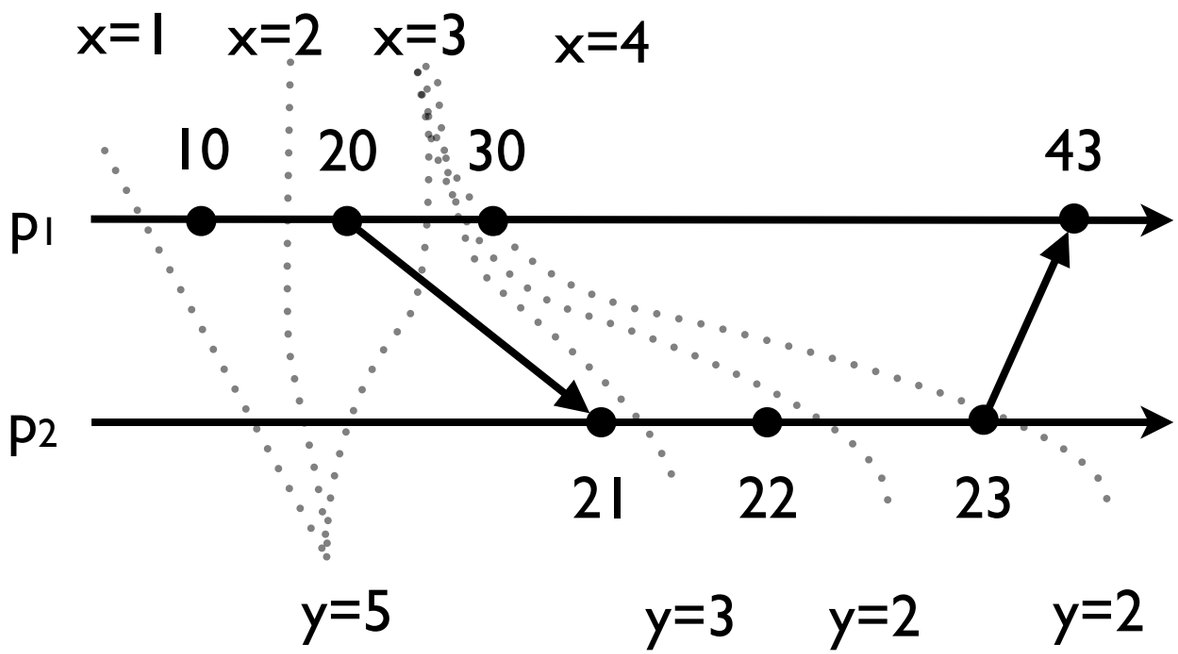


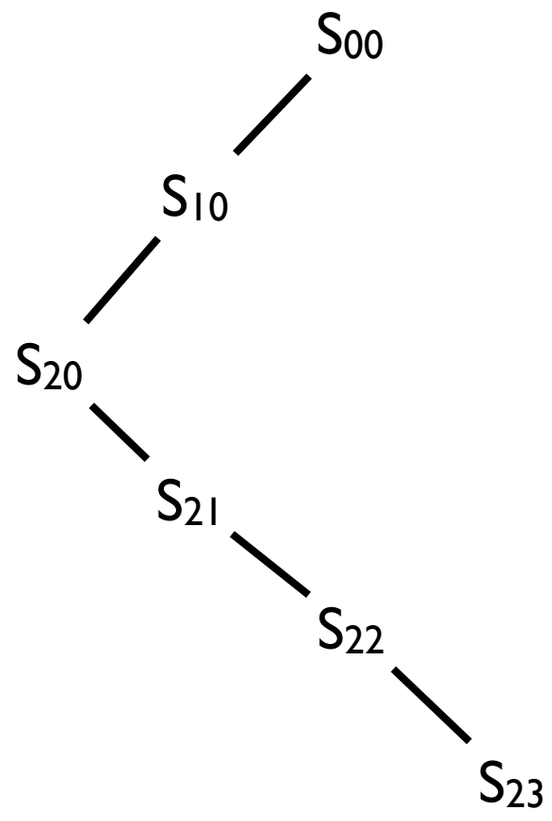


Note: the "global states" include the current process local states in each consistent cut (variables x, y)

Suppose property *p(x,y)* = "*x* = *y*"

Assembling consistent global states

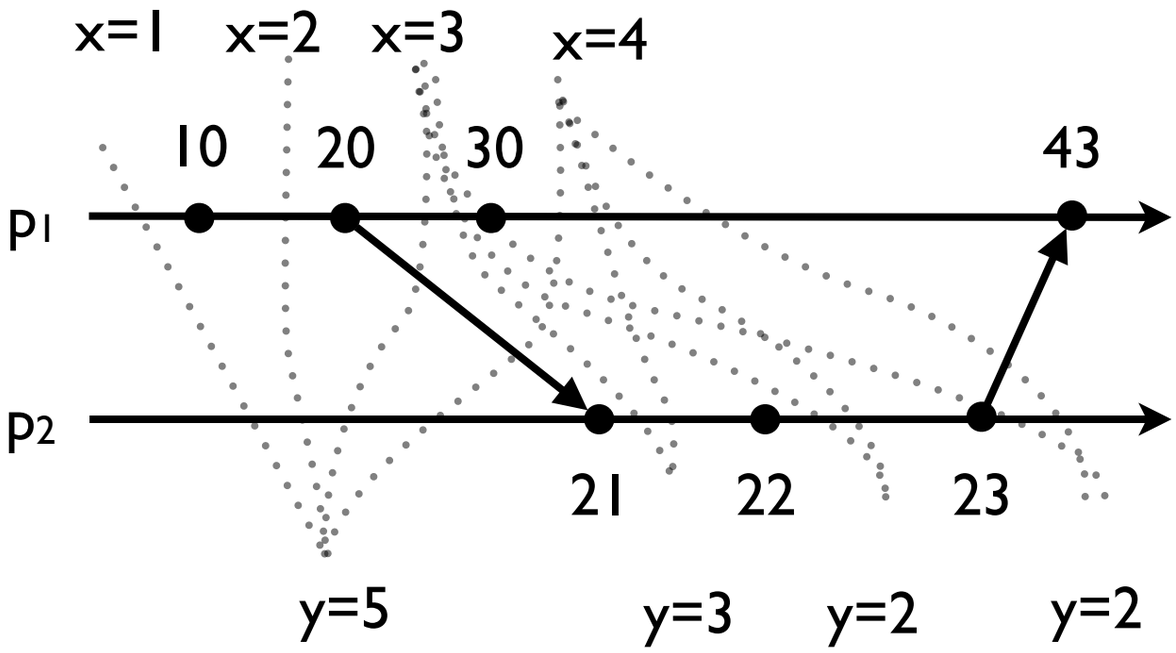


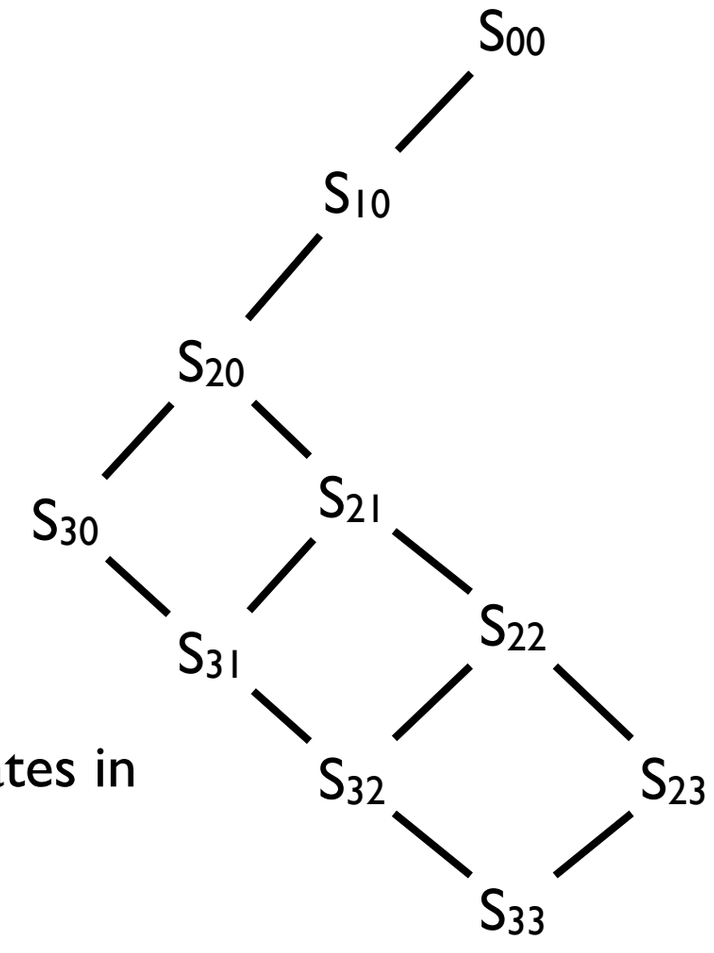


Note: the "global states" include the current process local states in each consistent cut (variables x, y)

Suppose property *p(x,y)* = "*x* = *y*"

Assembling consistent global states

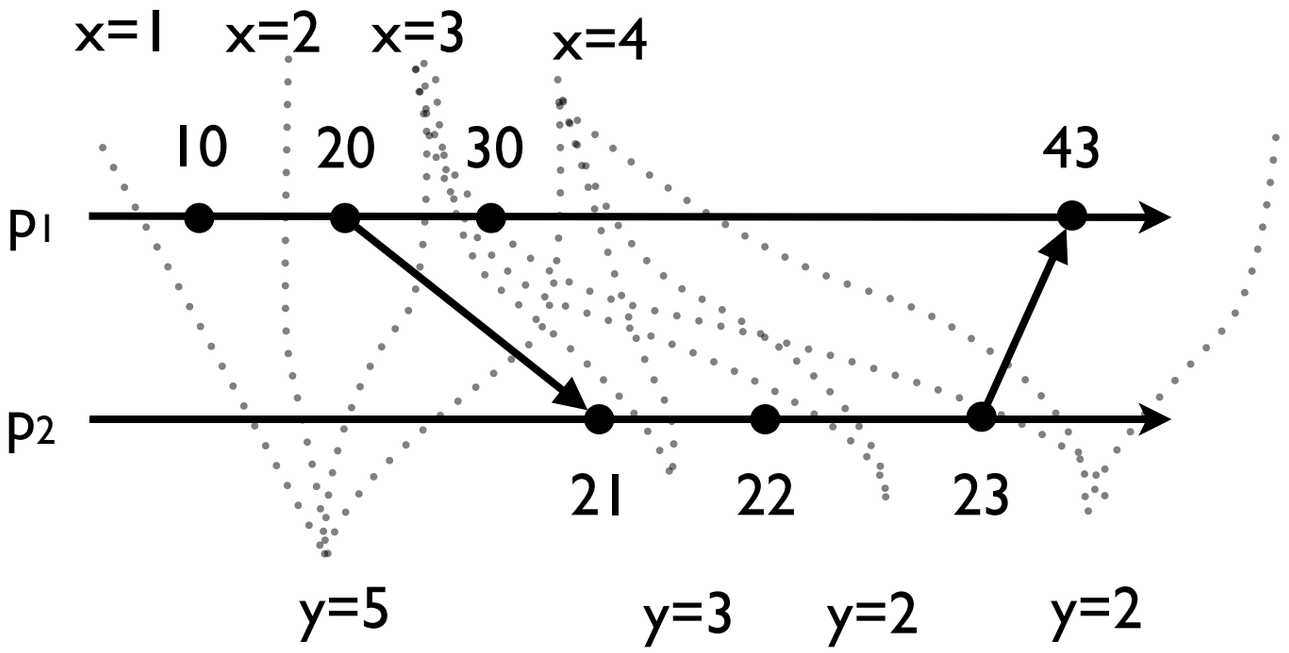


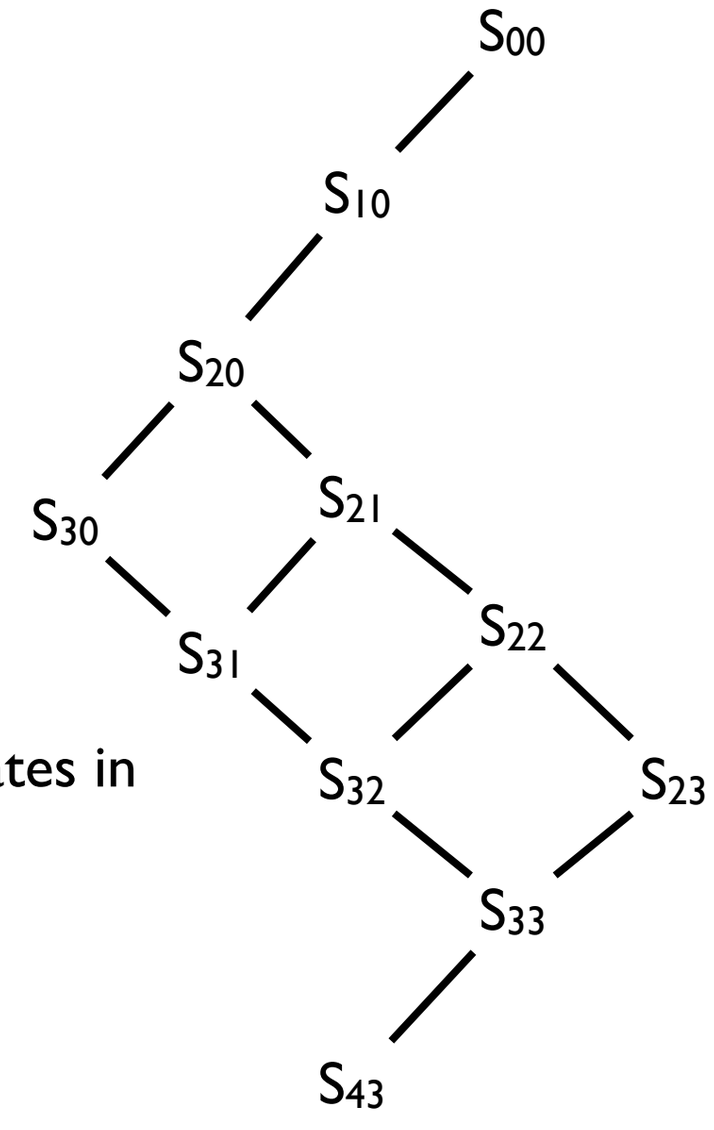


Note: the "global states" include the current process local states in each consistent cut (variables x, y)

Suppose property *p(x,y)* = "*x* = *y*"

Assembling consistent global states





Note: the "global states" include the current process local states in each consistent cut (variables x, y)

Suppose property *p(x,y)* = "*x* = *y*"

Evaluating Possibly

A state $S' = \{s_0^{x0'}, ...s_N^{xN'}\}$ is *reachable* from a state $S = \{s^{x0}_0, ...s^{xN}_N\}$ if:

- S' is a consistent state
  - $level(S') = 1 + level(S)$ and:
  - $x_i' = x_i$ or $x_i' = 1 + x_i \quad \forall\ 0 \leq i \leq N$
- That is, S' is the result of adding exactly one possible "next" event to S
  - leading to a consistent cut/state

Evaluating Possibly

Level = 0

States = {(s^0_0,...s^0_N)}

while (*States* is not empty)

Level = Level + 1

Reachable = {}

for *S'* where *level(S') = Level*

if *S'* is reachable from some state in *States*

then if *p(S')* **then** output "*possibly(p)* is True" and **quit**

else place *S'* in *Reachable*

output "*possibly(p)* is false"

Evaluating Definitely

- Note: First check if p true in the initial state, if so output "definitely(p) is true"

  Level = 0

  States = {(s^0_0,...s^0_N)}

  while (*States* is not empty)

  Level = Level + 1

  ReachableFalse = {}

  for *S'* where *level(S') = Level*

  if *S'* is reachable from some state in *States*

  then if ¬*(p(S'))* **then** place *S'* in *ReachableFalse*

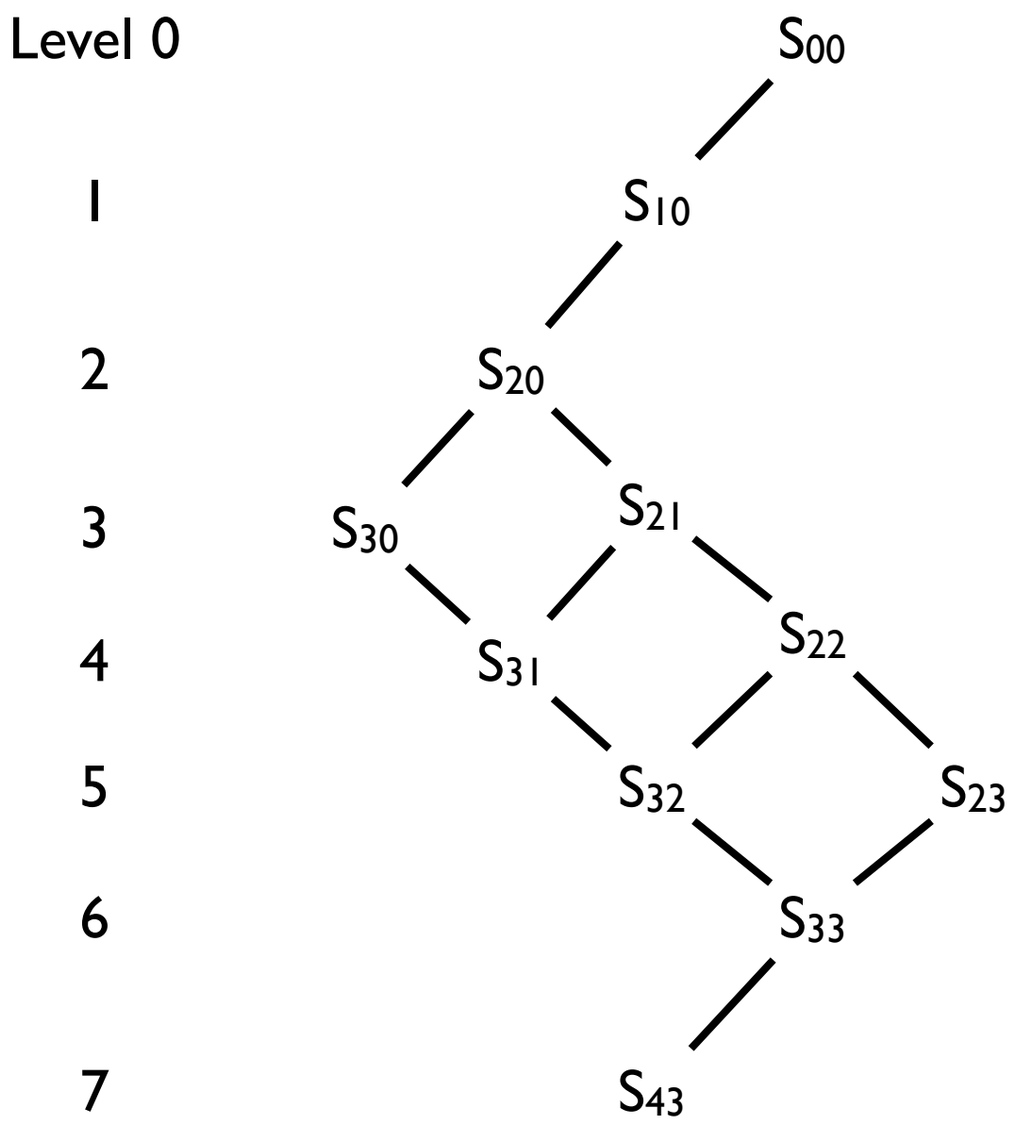
  States = ReachableFalse

  if *Level* is the maximum level recorded

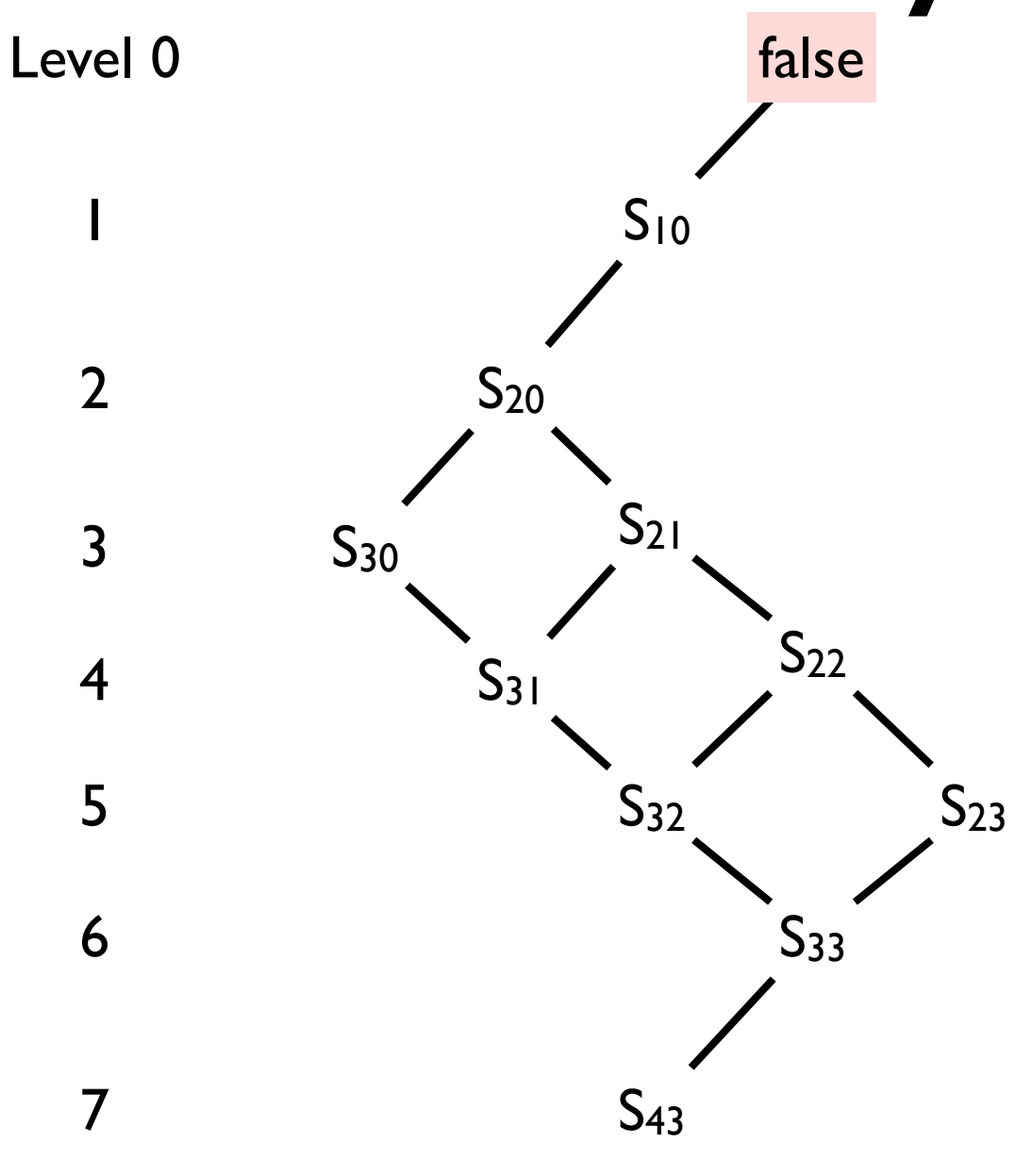
  then output "*definitely(p)* is false"

  else output "*definitely(p)* is true"

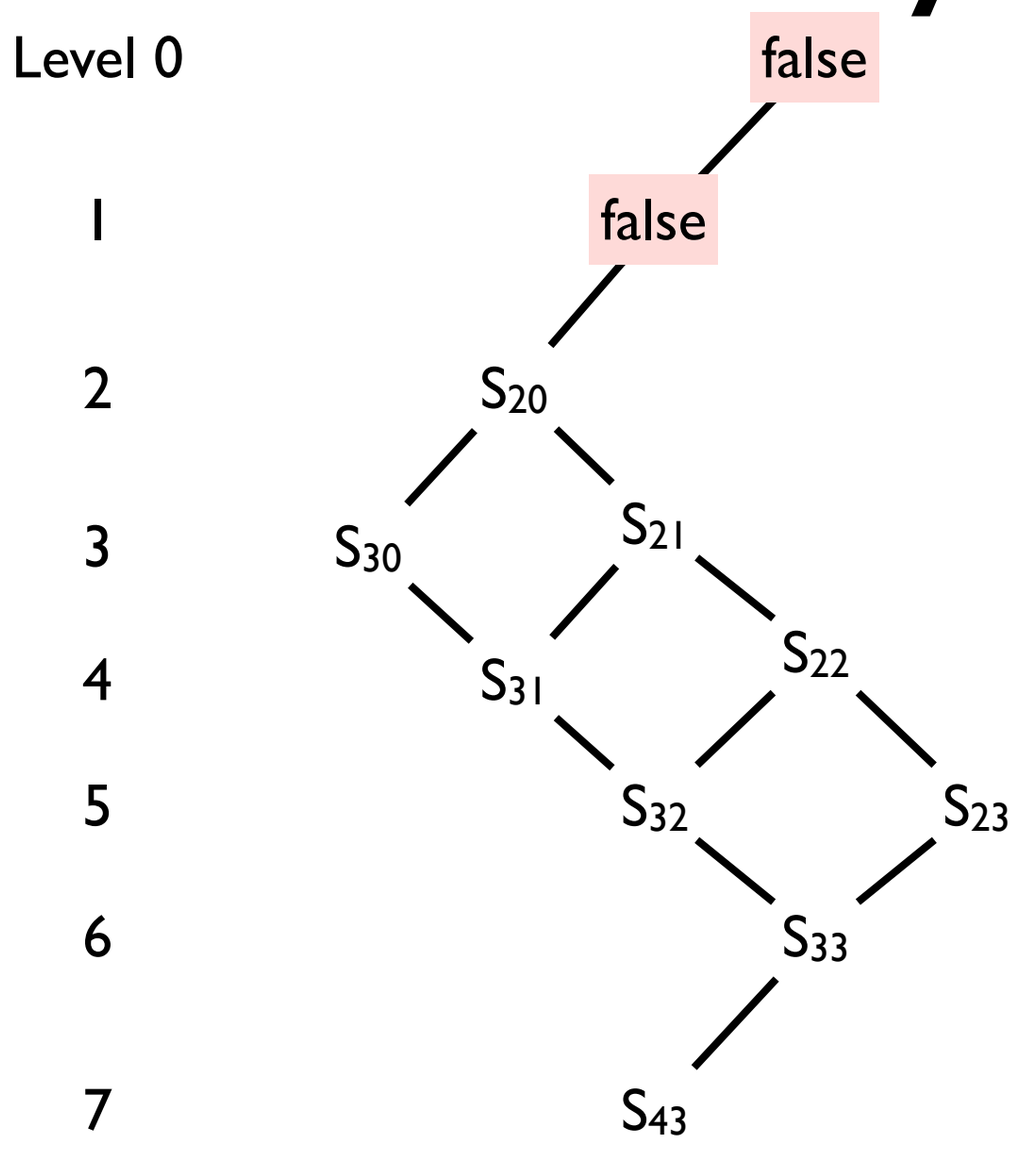
Evaluating Possibly, Definitely



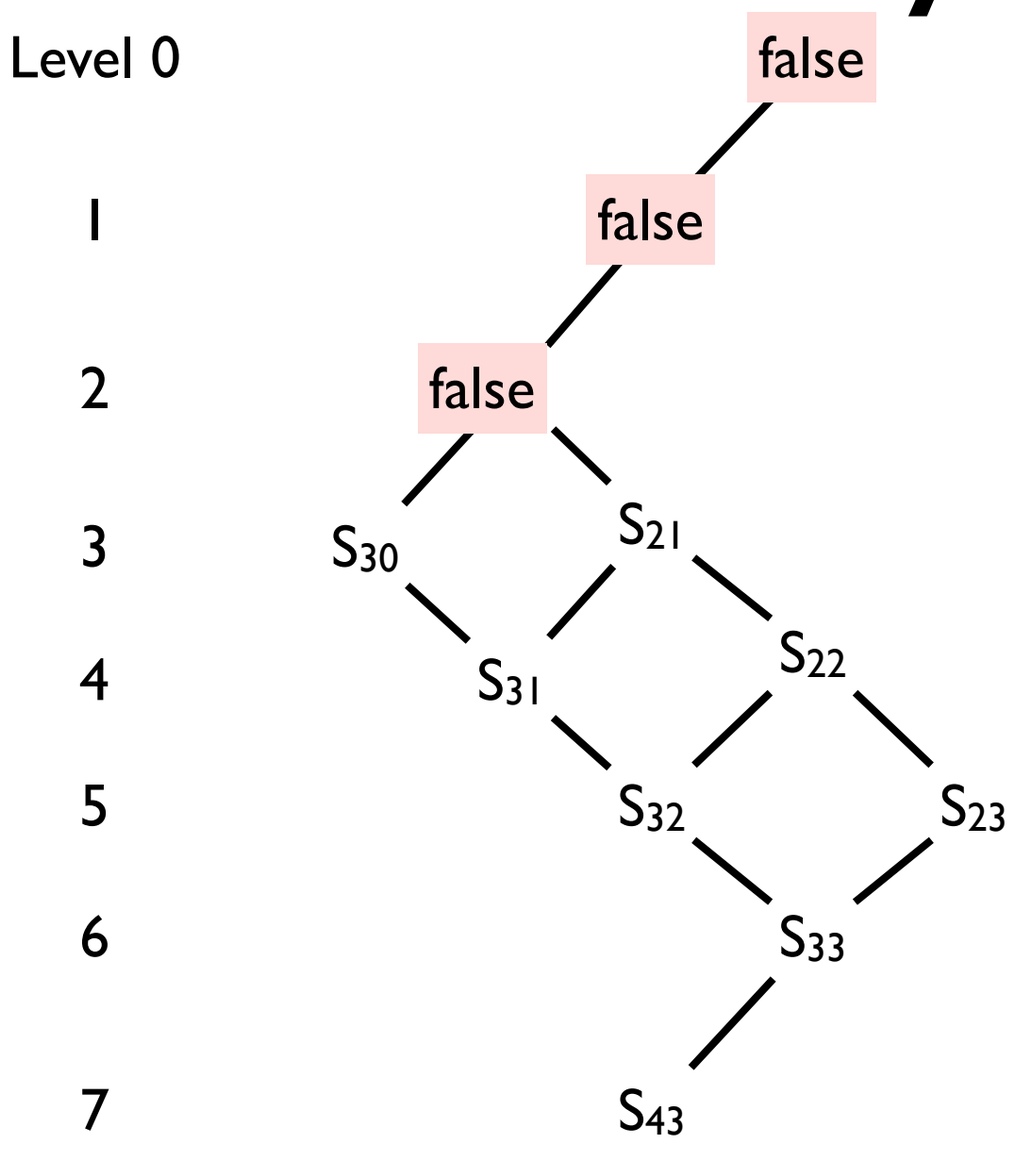
Evaluating Possibly, Definitely



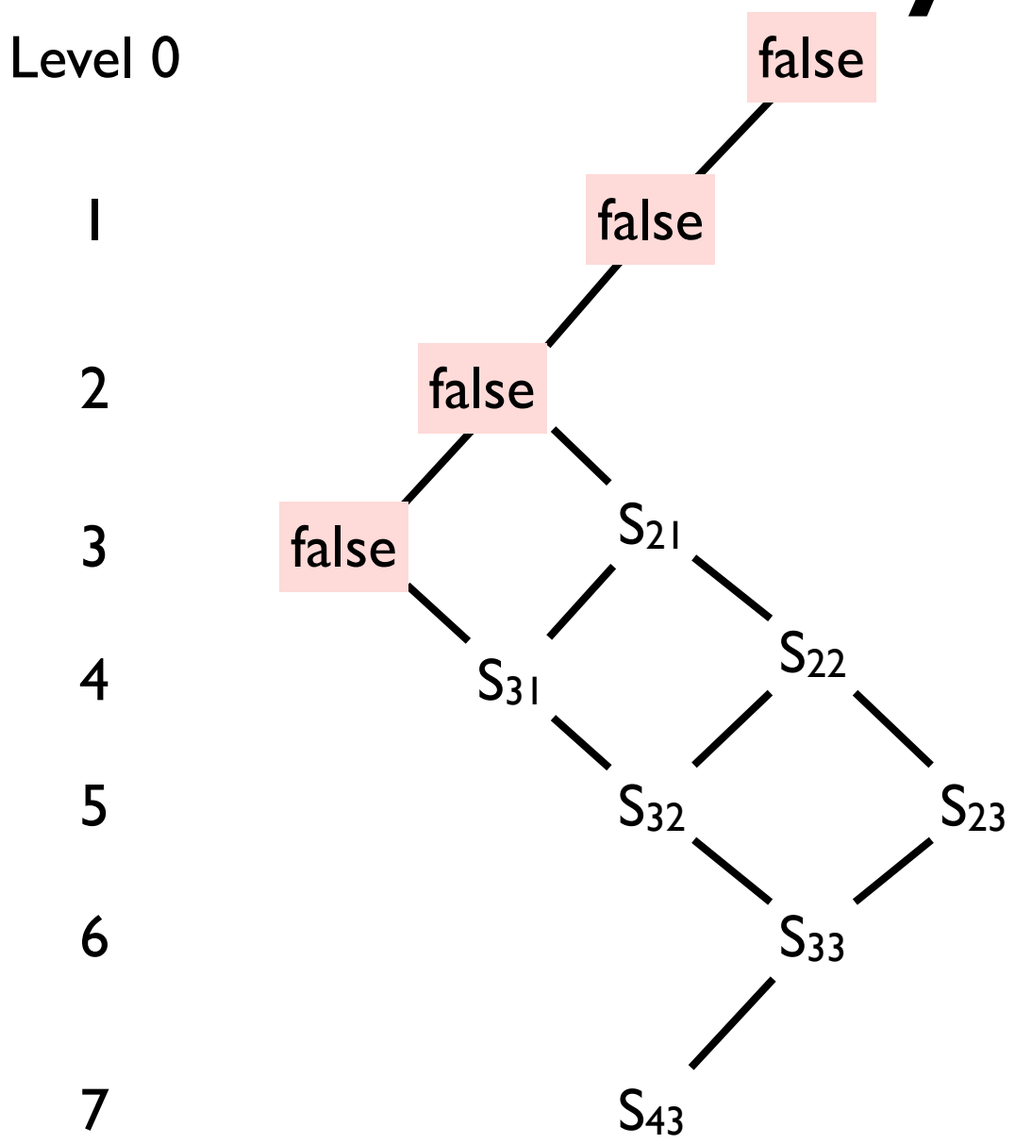
Evaluating Possibly, Definitely



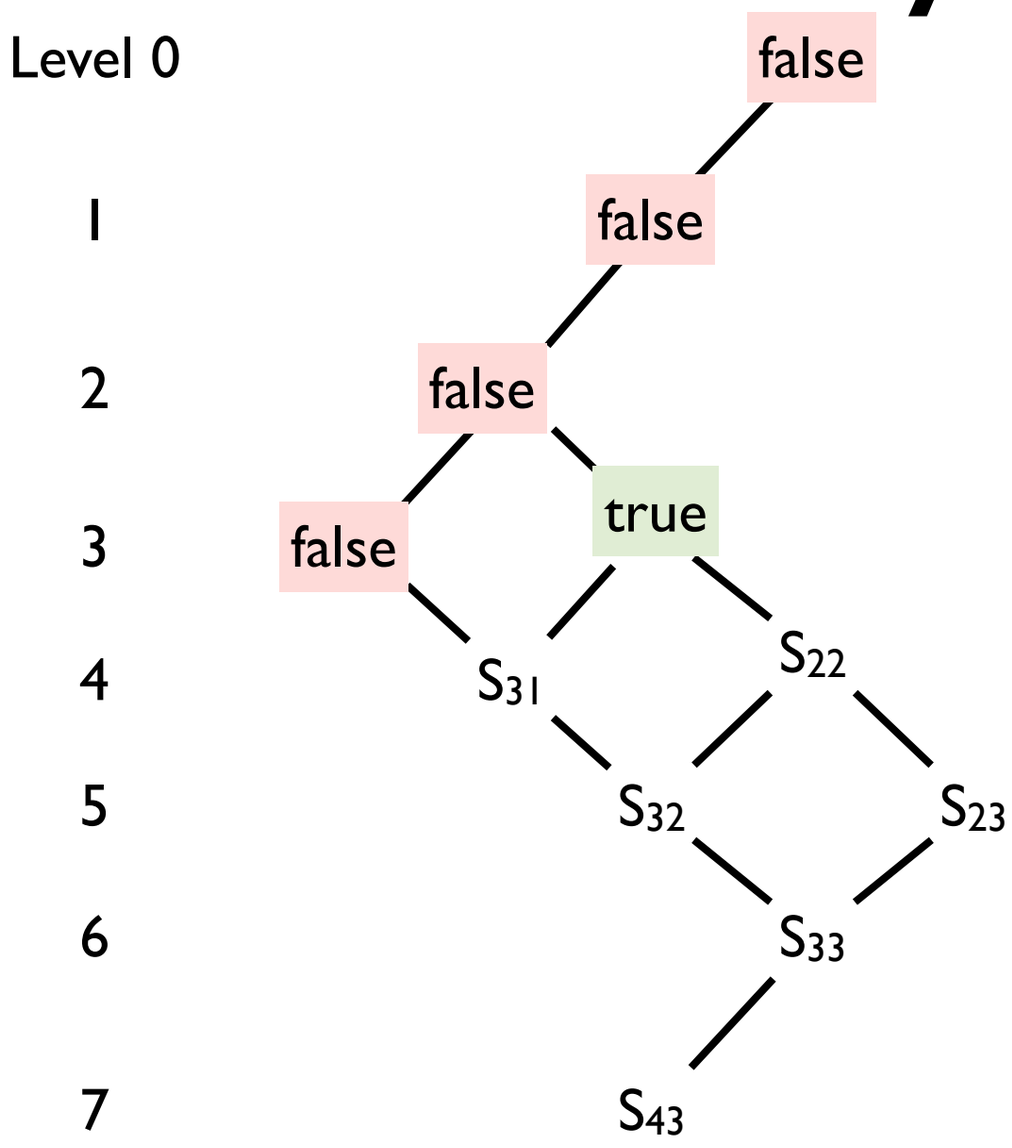
Evaluating Possibly, Definitely



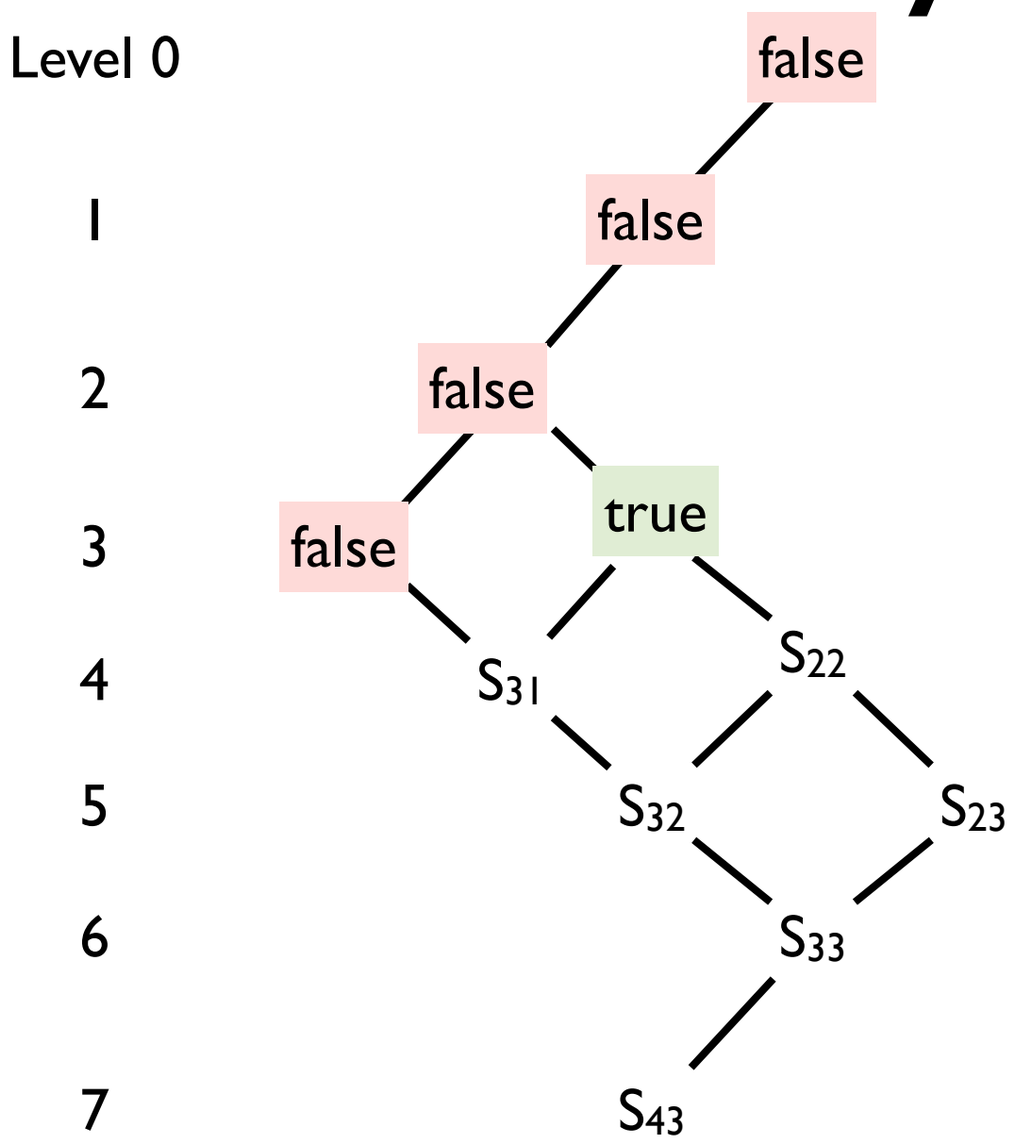
Evaluating Possibly, Definitely



Evaluating Possibly, Definitely

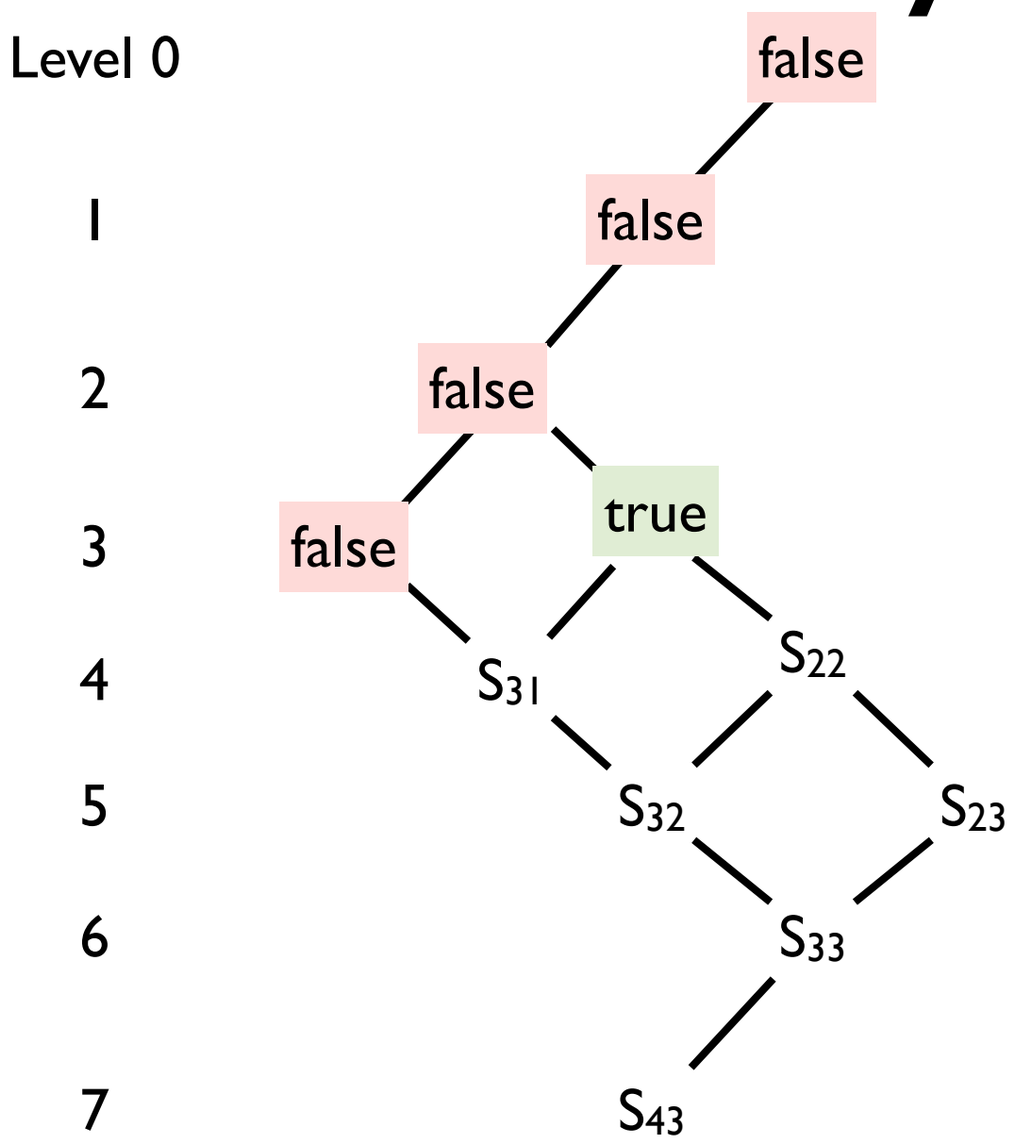


Evaluating Possibly, Definitely

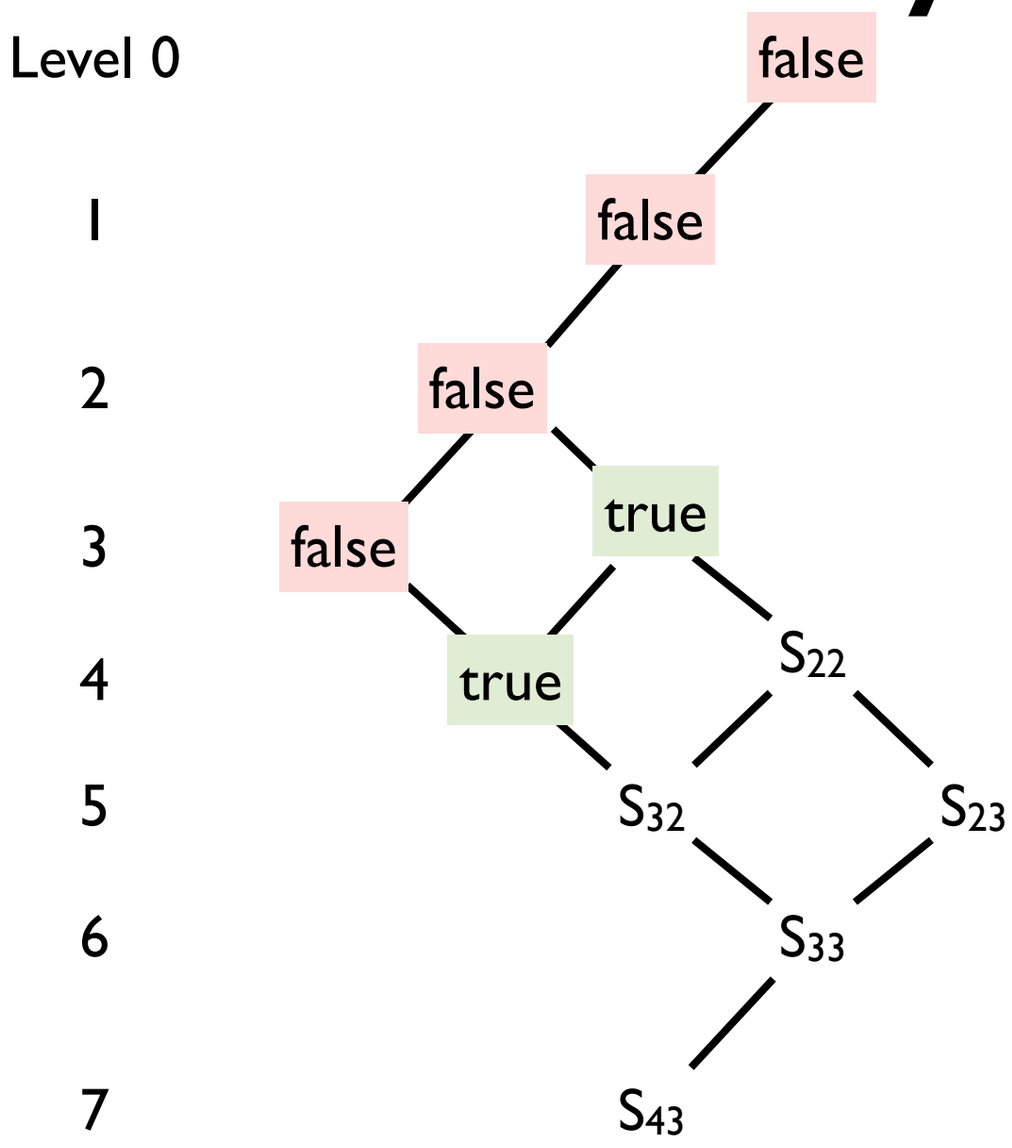


Possibly(p) is True

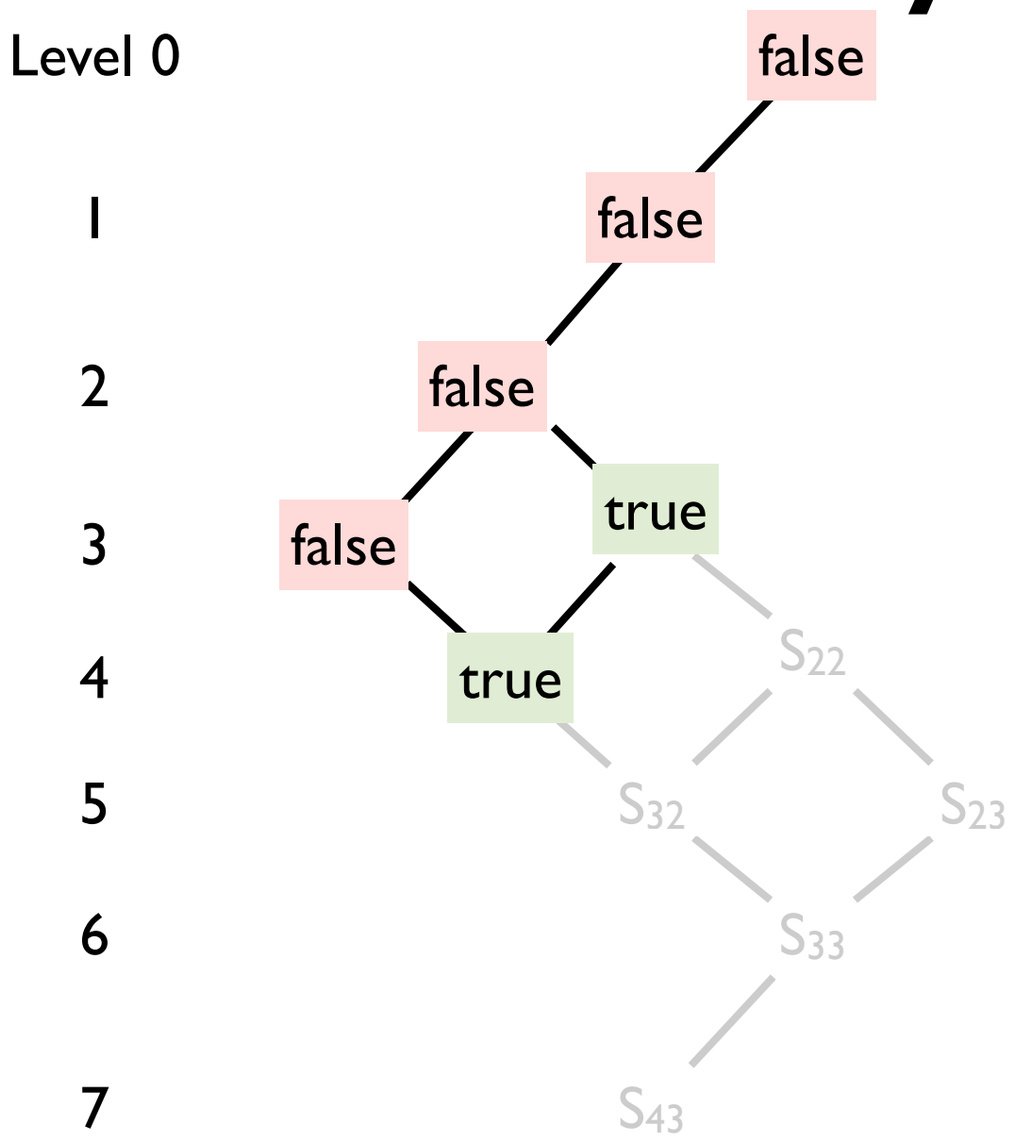
Evaluating Possibly, Definitely



Evaluating Possibly, Definitely



Evaluating Possibly, Definitely



Definitely(p) is True

Evaluating Possibly and Definitely

- Note that the number of states that must be evaluated is potentially huge
- In the worst case, there is no communication between processes, and the property is False for all states
- We must evaluate all **permutations** of states in which each local history is preserved
  - worst-case exponential time (NP-complete in general)
- This system therefore works better if there is a lot of communication and few local updates (which affect the predicate under investigation)

Distributed Debugging (Synchronous case)

- We have so far considered debugging within an **asynchronous** system
- Our notion of a consistent global state is one which could potentially have occurred
- In a synchronous system we have a little more information to make that judgement
- Suppose each process has a clock internally synchronized with the each other to a bound of D.
- With each state message, each process additionally time stamps the message with their local time at which the state was observed.
- For a single process with two state messages (s^x_i, V_i, t_i) and (s^{x+1}_i, V'_i, t'_i) we know that the local state s^x_i was valid between the time interval:
  - $t_i - D$ to $t'_i + D$

Distributed Debugging (Synchronous case)

- Recall our condition for a consistent global state:
  - $V_i[i] \geq V_j[i]\ \forall i, j \in \{1, ..., N\}$
- We can add to that:
  - $t_i - D \leq t_j \leq t'_i + D$ and vice versa forall i,j
- This can help eliminate impossible global states
  - improving performance and making "possibly" more precise
- But if there is a lot of communication (or D is large) then we may not prune the number of states very much

Summary

- Each process sends state update messages to a monitor process whenever a significant event occurs.
- From this the monitor can build up a set of consistent global states which may have occurred in the true history of events
- This can be used to evaluate whether some predicate was possibly true at some point, or definitely true at some point
- This can lead to a combinatorial explosion if there is a lot of concurrency
  - synchronized clocks can be used to decrease the number of concurrent states to consider