VC properties: event ordering

Given two vectors V and V', less than is defined as:

$V < V' \equiv (V \neq V') \wedge (\forall k : 1 \leq k \leq n : V[k] \leq V'[k])$

- Strong Clock Condition: $e \rightarrow e' \equiv VC(e) < VC(e')$
- Simple Strong Clock Condition:
  Given e_i of p_i and e_j of p_j, where $i \neq j$
  $e_i \rightarrow e_j \equiv VC(e_i)[i] \leq VC(e_j)[i]$
- Concurrency
  Given e_i of p_i and e_j of p_j, where $i \neq j$
  $e_i \parallel e_j \equiv (VC(e_i)[i] > VC(e_j)[i]) \wedge (VC(e_j)[j] > VC(e_i)[j])$

VC properties: consistency

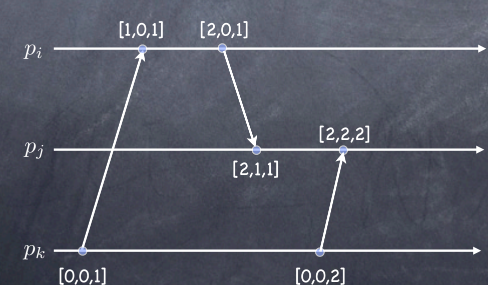
- Pairwise inconsistency
  Events e_i of p_i and e_j of p_j $(i \neq j)$ are pairwise inconsistent (i.e. can't be on the frontier of the same consistent cut) if and only if
  $(VC(e_i)[i] < VC(e_j)[i]) \vee (VC(e_j)[j] < VC(e_i)[j])$
- Consistent Cut
  A cut defined by $(c_1, \ldots, c_n)$ is consistent if and only if
  $\forall i, j : 1 \leq i \leq n, 1 \leq j \leq n : (VC(e_i^{c_i})[i] \geq VC(e_j^{c_j})[i])$

VC properties: weak gap detection

- Weak gap detection
  Given e_i of p_i and e_j of p_j, if $VC(e_i)[k] < VC(e_j)[k]$ for some $k \neq j$, then there exists e_k s.t
  $\neg(e_k \rightarrow e_i) \wedge (e_k \rightarrow e_j)$

VC properties: weak gap detection

- Weak gap detection
  Given e_i of p_i and e_j of p_j, if $VC(e_i)[k] < VC(e_j)[k]$ for some $k \neq j$, then there exists e_k s.t
  $\neg(e_k \rightarrow e_i) \wedge (e_k \rightarrow e_j)$



VC properties: strong gap detection

- Weak gap detection
  Given e_i of p_i and e_j of p_j, if $VC(e_i)[k] < VC(e_j)[k]$ for some $k \neq j$, then there exists e_k s.t
  $$\neg(e_k \rightarrow e_i) \wedge (e_k \rightarrow e_j)$$
- Strong gap detection
  Given e_i of p_i and e_j of p_j, if $VC(e_i)[i] < VC(e_j)[i]$ then there exists e_i' s.t.
  $$(e_i \rightarrow e_i') \wedge (e_i' \rightarrow e_j)$$

VCs for Causal Delivery

- Each process increments the local component of its VC only for events that are notified to the monitor
- Each message notifying event e is timestamped with $VC(e)$
- The monitor keeps all notification messages in a set M

Stability

Suppose p_0 has received m_j from p_j.
When is it safe for p_0 to deliver m_j?

Stability

Suppose p_0 has received m_j from p_j.
When is it safe for p_0 to deliver m_j?

- There is no earlier message in M
  $$\forall m \in M : \neg(m \rightarrow m_j)$$

Stability

Suppose p_0 has received m_j from p_j.
When is it safe for p_0 to deliver m_j?

- There is no earlier message in M
  $\forall m \in M : \neg(m \rightarrow m_j)$
- There is no earlier message from p_j

$TS(m_j)[j] = 1+$ no. of p_j messages delivered by p_0

Stability

Suppose p_0 has received m_j from p_j.
When is it safe for p_0 to deliver m_j?

- There is no earlier message in M
  $\forall m \in M : \neg(m \rightarrow m_j)$
- There is no earlier message from p_j

$TS(m_j)[j] = 1+$ no. of p_j messages delivered by p_0

- There is no earlier message m''_k from p_k, $k \neq j$
  see next slide...

Checking for m''_k

- Let m'_k be the last message p_0 delivered from p_k
- By strong gap detection, m''_k exists only if
  $TS(m'_k)[k] < TS(m_j)[k]$
- Hence, deliver m_j as soon as
  $\forall k : TS(m'_k)[k] \geq TS(m_j)[k]$

The protocol

- p_0 maintains an array $D[1, \ldots, n]$ of counters
- $D[i] = TS(m_i)[i]$ where m_i is the last message delivered from p_i

DR3: Deliver m from p_j as soon as both of the following conditions are satisfied:

$D[j] = TS(m)[j] - 1$

2. $D[k] \geq TS(m)[k], \forall k \neq j$

Properties

Property: a predicate that is evaluated over a run of the program

"every message that is received was previously sent"

Not everything you may want to say about a program is a property:

"the program sends an average of 50 messages in a run"

Safety properties

- "nothing bad happens"
  - no more than k processes are simultaneously in the critical section
  - messages that are delivered are delivered in causal order
  - Windows never crashes
- A safety property is "prefix closed":
  - if it holds in a run, it holds in every prefix

Liveness properties

- "something good eventually happens"
  - a process that wishes to enter the critical section eventually does so
  - some message is eventually delivered
  - Windows eventually boots
- Every run can be extended to satisfy a liveness property
  - if it does not hold in a prefix of a run, it does not mean it may not hold eventually

A really cool theorem

Every property is a combination of a safety property and a liveness property

(Alpern & Schneider)

The challenges of non-stable predicates

- Consider a non-stable predicate Φ encoding, say, a safety property. We want to determine whether Φ holds for our program.

The challenges of non-stable predicates

- Consider a non-stable predicate Φ encoding, say, a safety property. We want to determine whether Φ holds for our program.
- Suppose we apply Φ to Σ^s

The challenges of non-stable predicates

- Consider a non-stable predicate Φ encoding, say, a safety property. We want to determine whether Φ holds for our program.
- Suppose we apply Φ to Σ^s
- Φ holding in Σ^s does not preclude the possibility that our program violates safety!

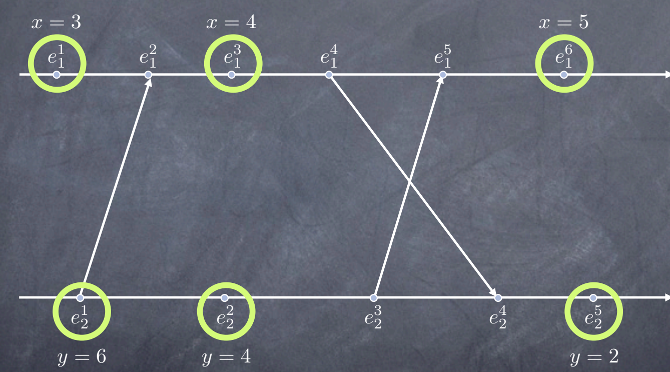
The challenges of non-stable predicates

- Consider now a different non-stable predicate Φ. We want to determine whether Φ ever holds during a particular computation.
- Suppose we apply Φ to Σ^s

The challenges of non-stable predicates

- Consider now a different non-stable predicate Φ. We want to determine whether Φ ever holds during a particular computation.
- Suppose we apply Φ to Σ^s
- Φ holding in Σ^s does not imply that Φ ever held during the actual computation!

Example



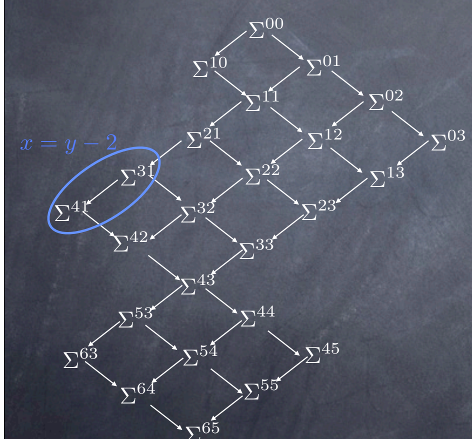
Detect whether the following predicates hold:

$x = y$ $\quad$ $x = y - 2$

Assume that initially:

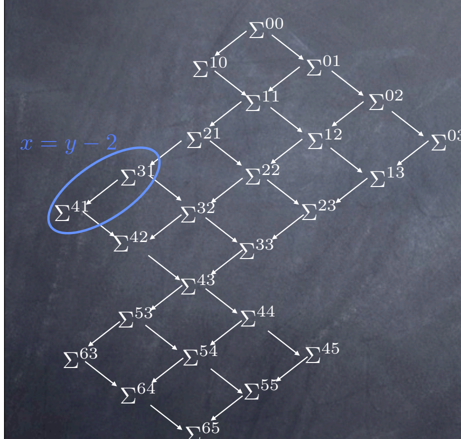
$x = 0; y = 10$

Possibly



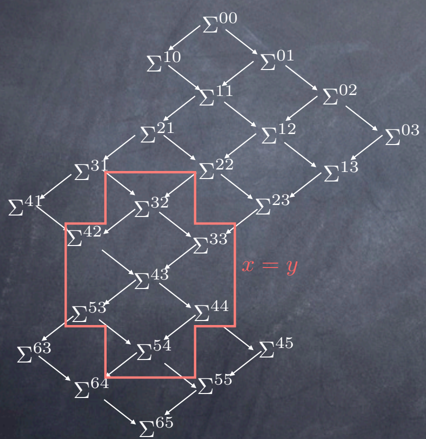
- If Σ^s is Σ^{31} or Σ^{41}, $x = y - 2$ is detected, but it may never have occurred

Possibly



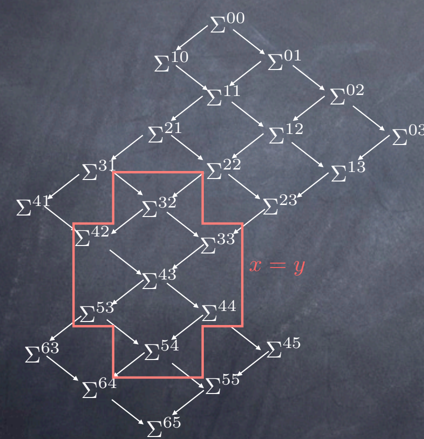
- If Σ^s is Σ^{31} or Σ^{41}, $x = y - 2$ is detected, but it may never have occurred
- Possibly(Φ)
  There exists a consistent observation of the computation O such that Φ holds in a global state of O

Definitely



- We know that $x = y$ has occurred, but it may not be detected if tested before Σ^{32} or after Σ^{54}

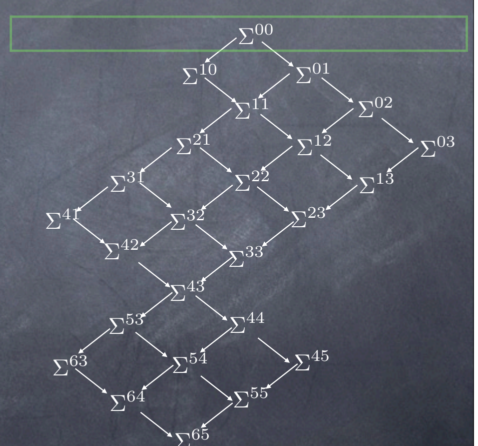
Definitely



- We know that $x = y$ has occurred, but it may not be detected if tested before Σ^{32} or after Σ^{54}
- Definitely(Φ)
  For every consistent observation O of the computation, there exists a global state of O in which Φ holds

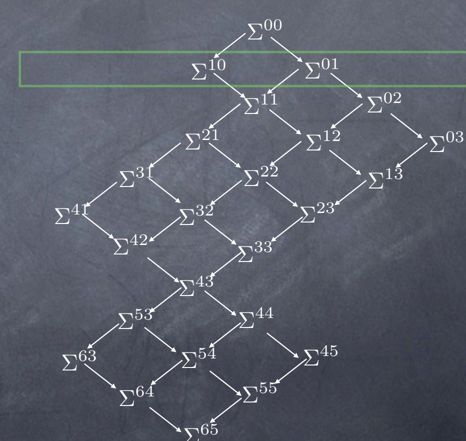
Computing Possibly

- Scan lattice, level after level
- If Φ holds in one global state, then Possibly(Φ)



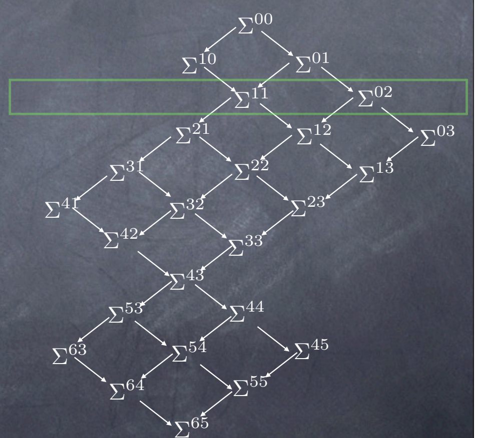
Computing Possibly

- Scan lattice, level after level
- If Φ holds in one global state, then Possibly(Φ)



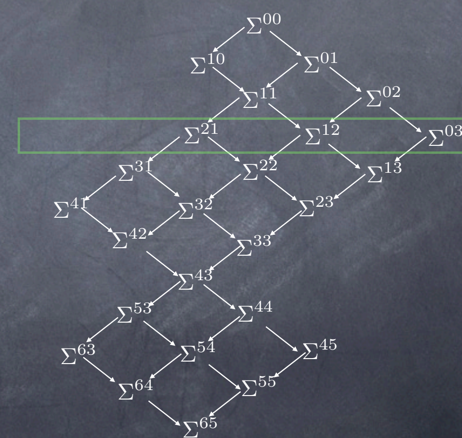
Computing Possibly

- Scan lattice, level after level
- If Φ holds in one global state, then Possibly(Φ)



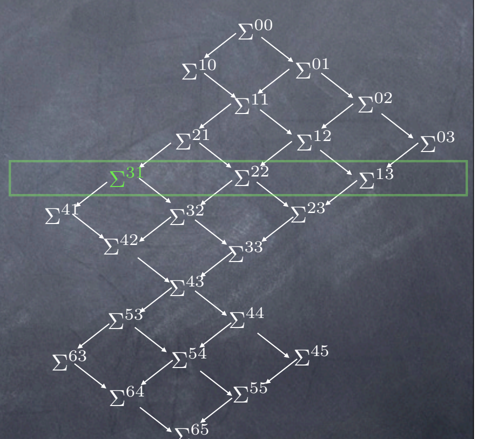
Computing Possibly

- Scan lattice, level after level
- If Φ holds in one global state, then Possibly(Φ)



Computing Possibly

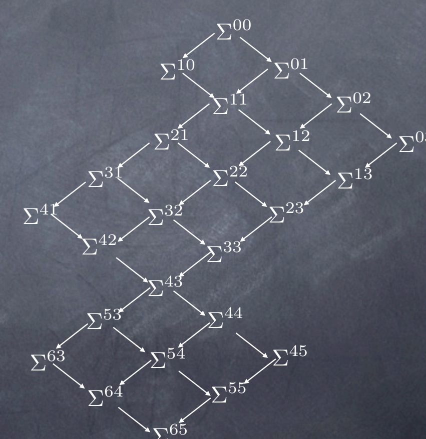
- Scan lattice, level after level
- If Φ holds in one global state, then Possibly(Φ)



Possibly$(x = y - 2)$

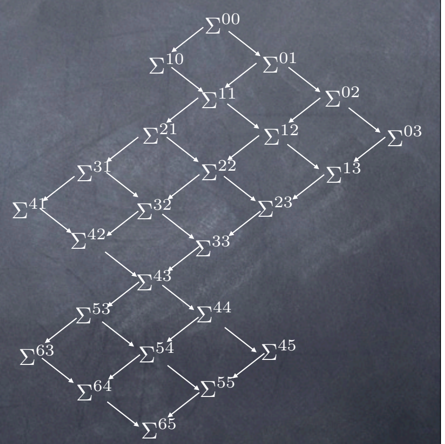
Computing Definitely

- Scan lattice, level after level



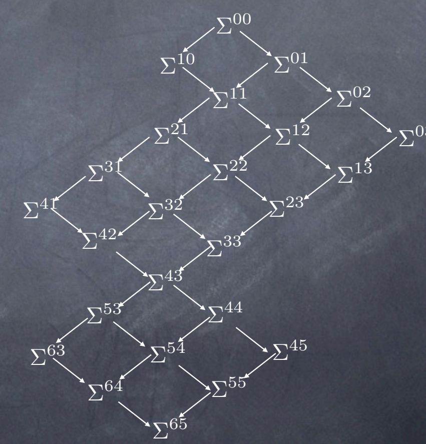
Computing Definitely

- Scan lattice, level after level
- Given a level, only expand nodes that correspond to states for which $\neg\Phi$



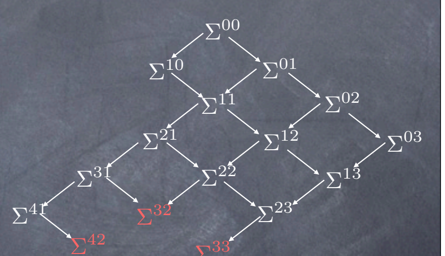
Computing Definitely

- Scan lattice, level after level
- Given a level, only expand nodes that correspond to states for which $\neg\Phi$
- If no such state, then Definitely(Φ)
- If reached last state Σ^l, and $\Phi(\Sigma^l)$, then ¬Definitely(Φ)



Computing Definitely

- Scan lattice, level after level
- Given a level, only expand nodes that correspond to states for which $\neg\Phi$
- If no such state, then Definitely(Φ)
- If reached last state Σ^l, and $\Phi(\Sigma^l)$, then ¬Definitely(Φ)



Definitely $(x = y)$

Building the lattice: collecting local states

- To build the global states in the lattice, p_0 collects local states from each process.
- p_0 keeps the set of local states received from p_i in a FIFO queue Q_i

Key questions:

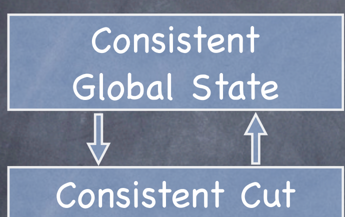
1. when is it safe for p_0 to discard a local state σ_i^k of p_i?
2. Given level i of the lattice, how does one build level $i+1$?

Garbage-collecting local states

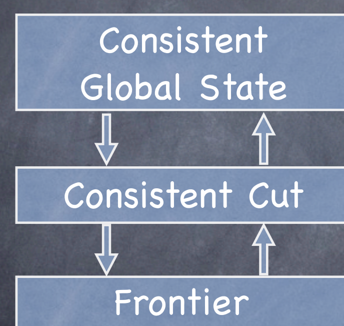
- For each local state σ_i^k, we need to determine:
  - $\Sigma_{min}(\sigma_i^k))$, the earliest consistent state that σ_i^k can belong to
  - $\Sigma_{max}(\sigma_i^k))$, the latest consistent state that σ_i^k can belong to

Defining "earliest" and "latest"

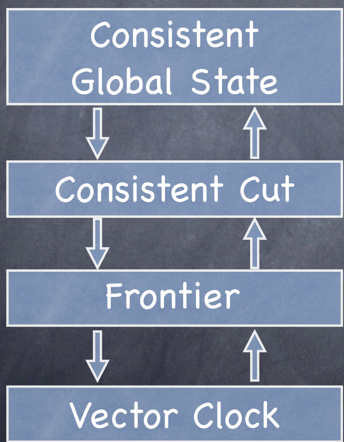
Defining "earliest" and "latest"



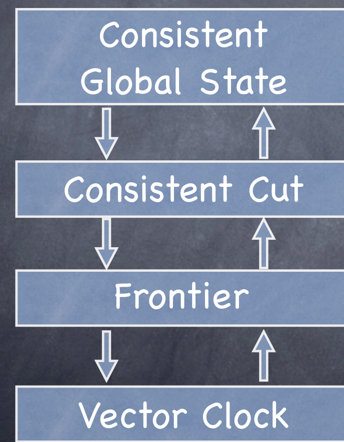
Defining "earliest" and "latest"



Defining "earliest" and "latest"



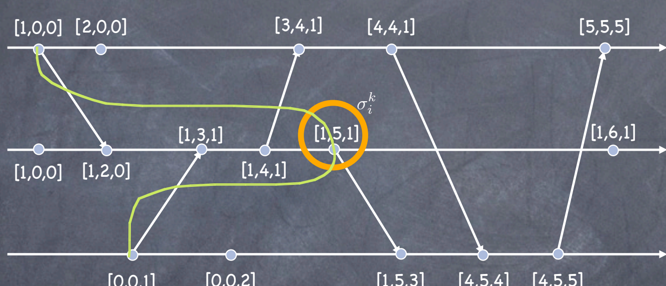
Defining "earliest" and "latest"



Associate a vector clock with each consistent global state

- $\Sigma_{min}(\sigma_i^k))$ is the consistent global state with the lowest vector clock that has σ_i^k on its frontier
- $\Sigma_{max}(\sigma_i^k))$ is the one with the highest

Computing Σ_{min}



- Label σ_i^k with $VC(e_i^k)$

  $\Sigma_{min}(\sigma_i^k) = (\sigma_1^{c_1}, \sigma_2^{c_2}, \ldots, \sigma_n^{c_n}) : \forall j : c_j = VC(\sigma_i^k)[j]$

- $\Sigma_{min}(\sigma_i^k)$ and σ_i^k have the same vector clock!

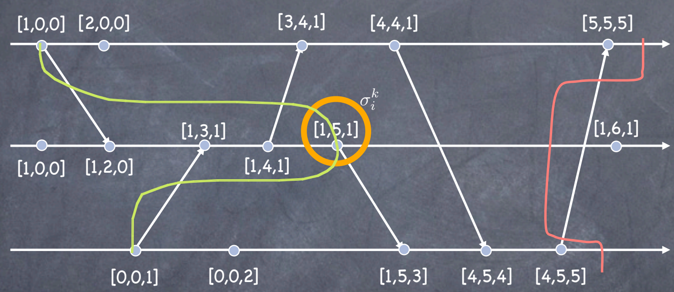
Computing Σ_{max}

$\Sigma_{max}(\sigma_i^k) = (\sigma_1^{c_1}, \sigma_2^{c_2}, \ldots \sigma_n^{c_n}) :$

$\wedge \forall j : VC(\sigma_j^{c_j})[i] \leq VC(\sigma_i^k)[i]$

$\wedge((\sigma_j^{c_j} = \sigma_j^{c_f}) \vee VC(\sigma_j^{c_j+1})[i] > VC(\sigma_i^k)[i]))$

Computing Σ_{max}



$\Sigma_{max}(\sigma_i^k) = (\sigma_1^{c_1}, \sigma_2^{c_2}, \ldots \sigma_n^{c_n})$:

set of local states one for each process, s.t.

$\wedge \forall j : VC(\sigma_j^{c_j})[i] \leq VC(\sigma_i^k)[i]$

$\wedge((\sigma_j^{c_j} = \sigma_j^{c_f}) \vee VC(\sigma_j^{c_j+1})[i] > VC(\sigma_i^k)[i]))$

Computing Σ_{max}

$\Sigma_{max}(\sigma_i^k) = (\sigma_1^{c_1}, \sigma_2^{c_2}, \ldots \sigma_n^{c_n})$:

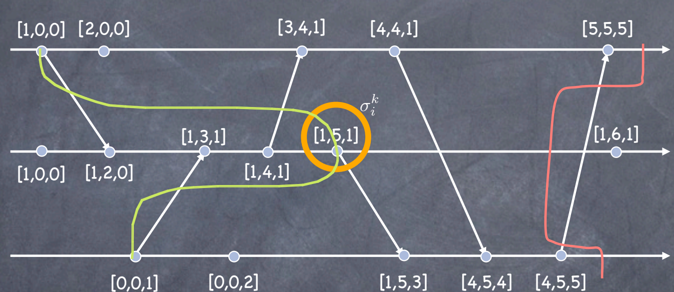
set of local states one for each process, s.t.

$\wedge \forall j : VC(\sigma_j^{c_j})[i] \leq VC(\sigma_i^k)[i]$

all local states are pair-wise consistent with σ_i^k

$\wedge((\sigma_j^{c_j} = \sigma_j^{c_f}) \vee VC(\sigma_j^{c_j+1})[i] > VC(\sigma_i^k)[i]))$

Computing Σ_{max}



$\Sigma_{max}(\sigma_i^k) = (\sigma_1^{c_1}, \sigma_2^{c_2}, \ldots \sigma_n^{c_n})$:

set of local states one for each process, s.t.

$\wedge \forall j : VC(\sigma_j^{c_j})[i] \leq VC(\sigma_i^k)[i]$

all local states are pair-wise consistent with σ_i^k

$\wedge((\sigma_j^{c_j} = \sigma_j^{c_f}) \vee VC(\sigma_j^{c_j+1})[i] > VC(\sigma_i^k)[i]))$

and they are the last such state

Assembling the levels

- To build level l
  - wait until each Q_i contains a local state for whose vector clock:

$$\sum_{i=1}^{n} VC[i] \geq l$$

- To build level $l+1$
  - For each global state $\Sigma^{i_1,i_2,\ldots,i_n}$ on level l, build

$$\Sigma^{i_1+1,i_2,\ldots,i_n}, \Sigma^{i_1,i_2+1,\ldots,i_n}, \ldots, \Sigma^{i_1,i_2,\ldots,i_n+1}$$

  - Using VCs, check whether these global states are consistent