Outlining a Demand Driven Execution Model for CnC

Peter Elmers, Nick Vrvilo

Motivation

- Remove unneeded computation

Outline

- Step inverse
- Demand driven model
- Future work

Preliminary: Step Inverse

- A step definition represents a map from step tags to output collection tags
  - Remark: this map is **one-to-one** if we do not output multiple items into a single collection
- Then we can find the inverse, the map from an item collection tag to the step tag that outputs it.

Abridged example

```
[ int data: i ];

// Init: Set data[1] and prescribe process[1].


( process: x )
    <- [ data : x ]
    -> [ data : x + 1 ],
       ( process: x + 1 ) $when(x + 1 < 5);


// Final: get item data[5]
```

Inverse of ‘process’

```
( process: x )

    <- [ data : x ]

    -> [ data : x + 1 ],

       ( process: x + 1 )
,       $when(x + 1 < 5);
```

```
{
'data': [{x: t1 - 1}],
'process': [{x:
     Piecewise(
       (t1 - 1, t1 < 5),
       (nan, True))
}]
}
```

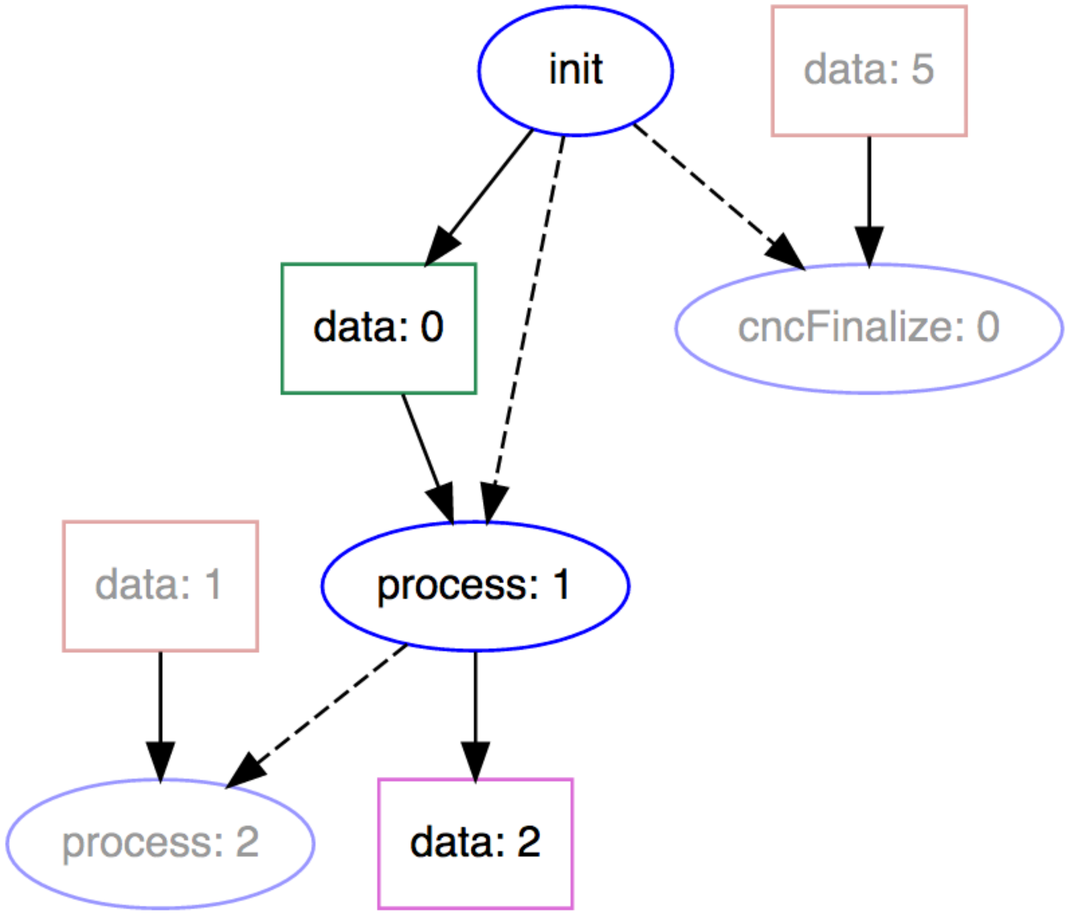
Step inverse uses

- Value of item is wrong?
  - Step inverse to blame the step that put that item
- Deadlock?
  - Use event graph to highlight deadlocked items
  - Run step inverse to find the responsible step

Deadlock blame example

```
Init: Put data[0], prescribe process[1].

( process: x )
    <- [ data : x - 1 ]
    -> [ data : x + 1 ],
       ( process: x + 1 ) $when(x + 1 < 5);
```

Deadlocked graph



Auto-blame output

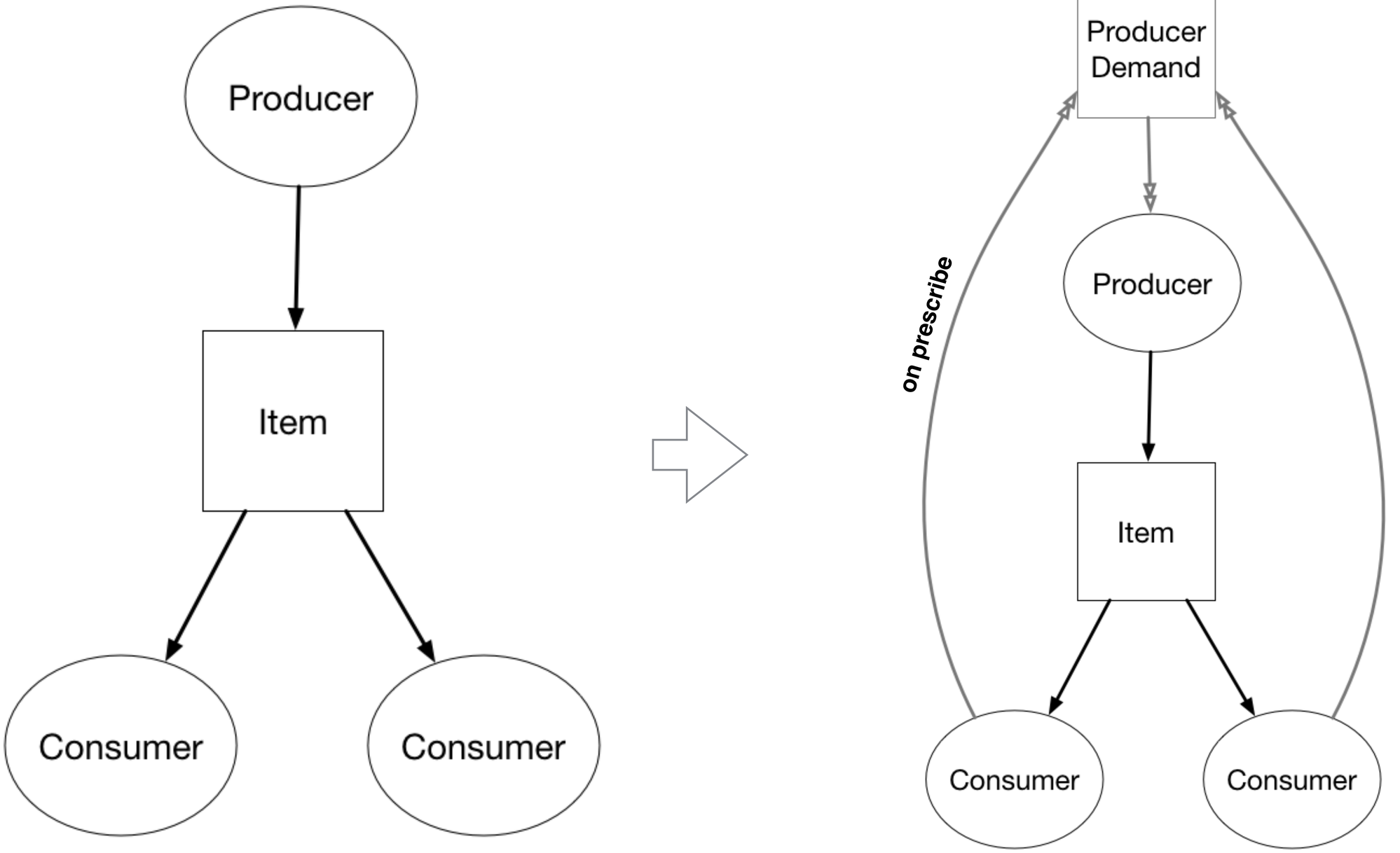
```
Performing automatic blame on potentially deadlocked
items from event log: ['data@1']

{

'data@1': {

  'process': {
    'x': 0

}}}
```

Demand driven rules

- Definition. Only run step when we need its output.
- Condition. Collection producer is unambiguous.

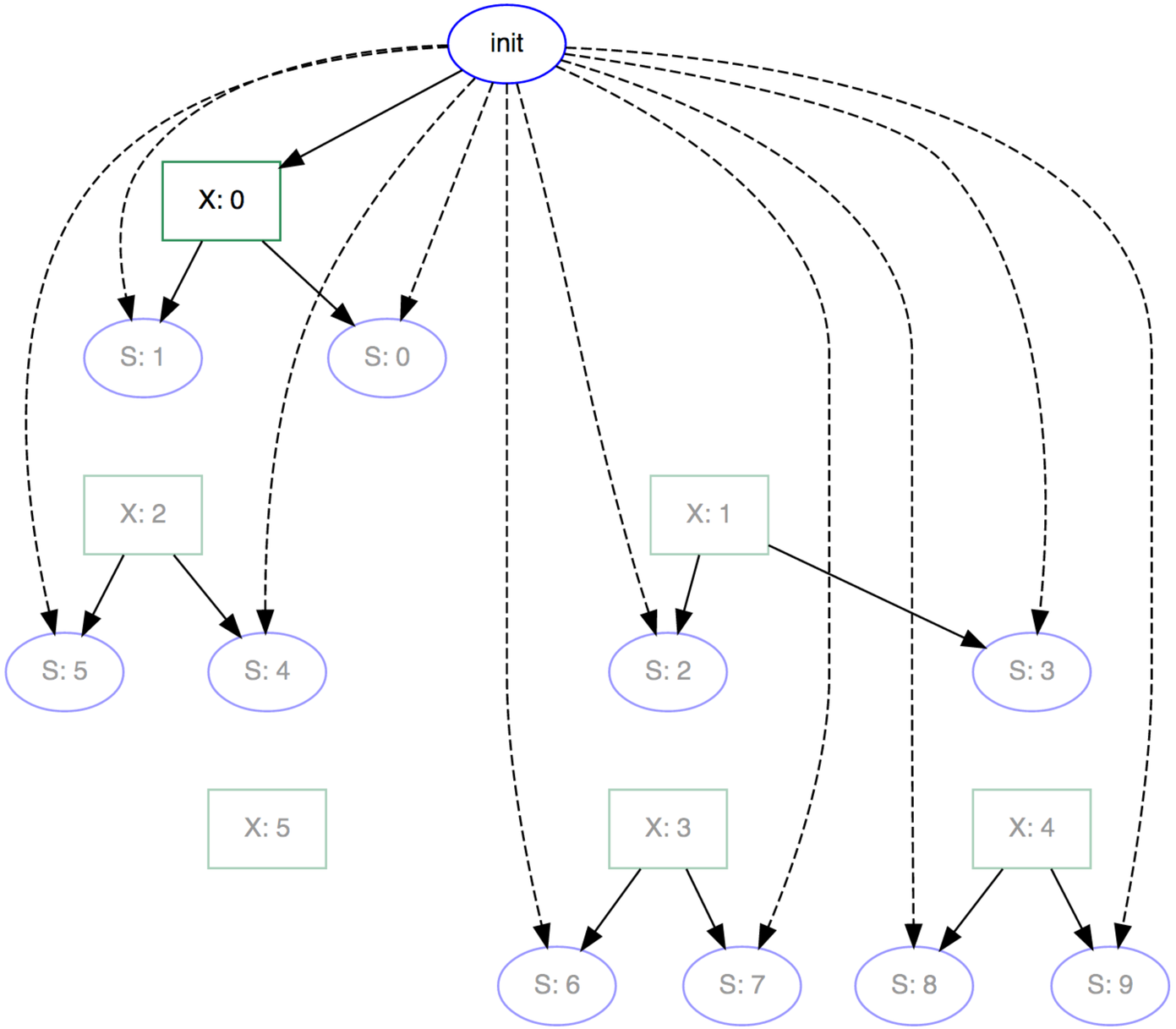
Implementation

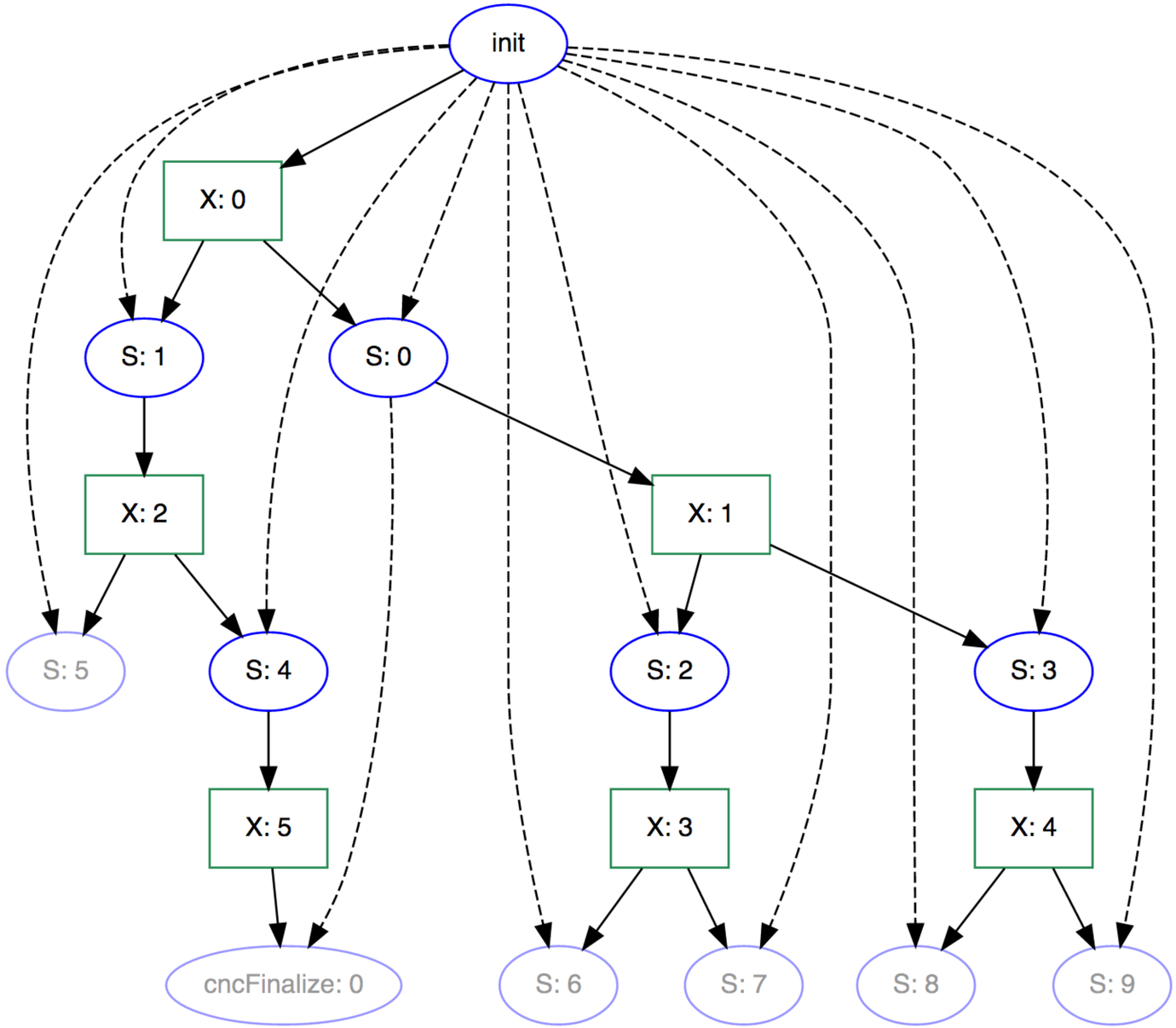
- Shadow collections to track demand

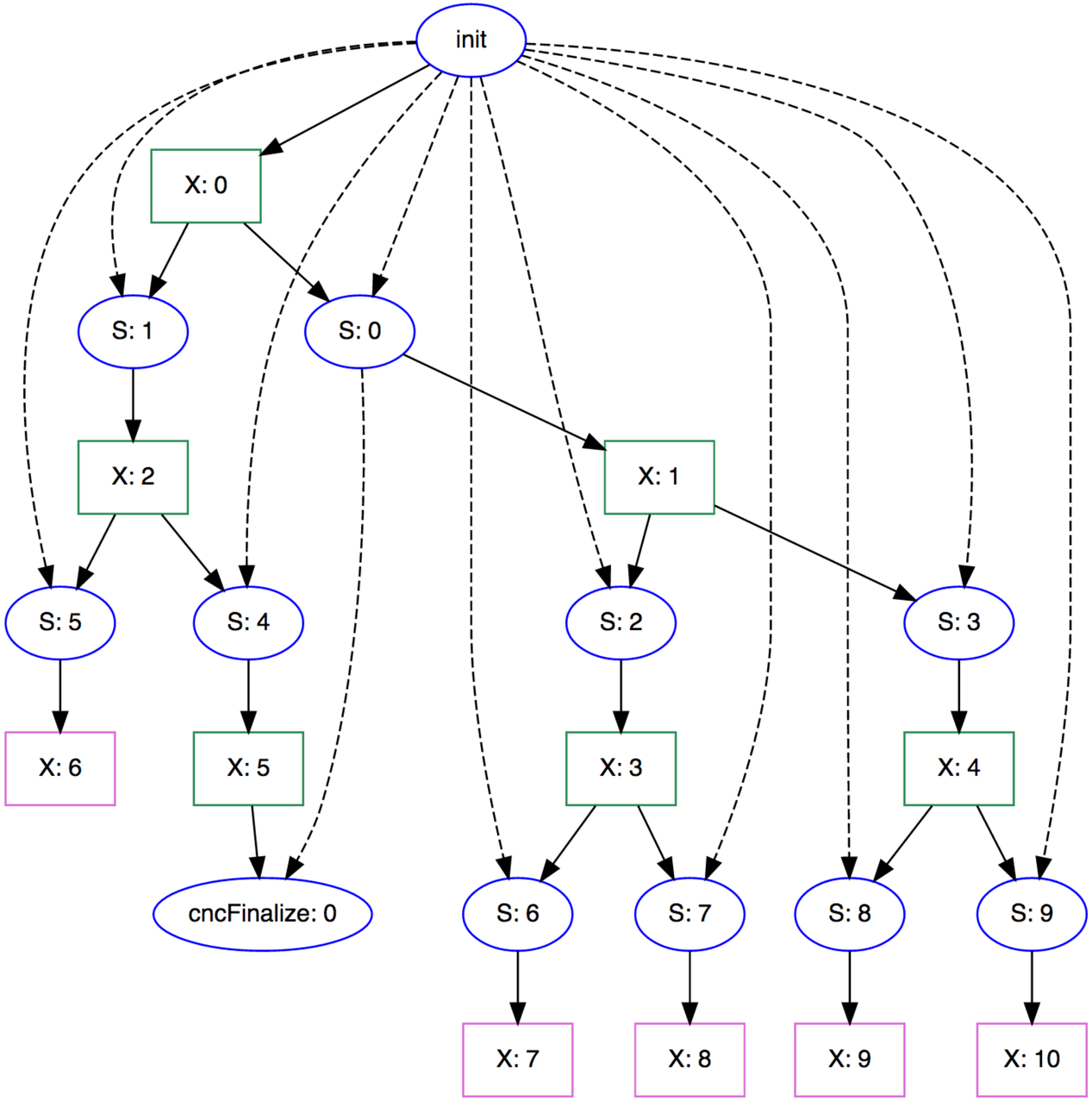

Producer
Item
Consumer
Consumer
Producer Demand
Producer
on prescribe
Item
Consumer
Consumer

Example

```
[ int X: i ];


( $initialize: () ) -> ( S: $range(0, 10) ), [ X: 0 ];


( S: i )
    <- [ X: i / 2 ]
    -> [ X: i + 1 ];


( $finalize: () ) <- [ X: 5 ];
```

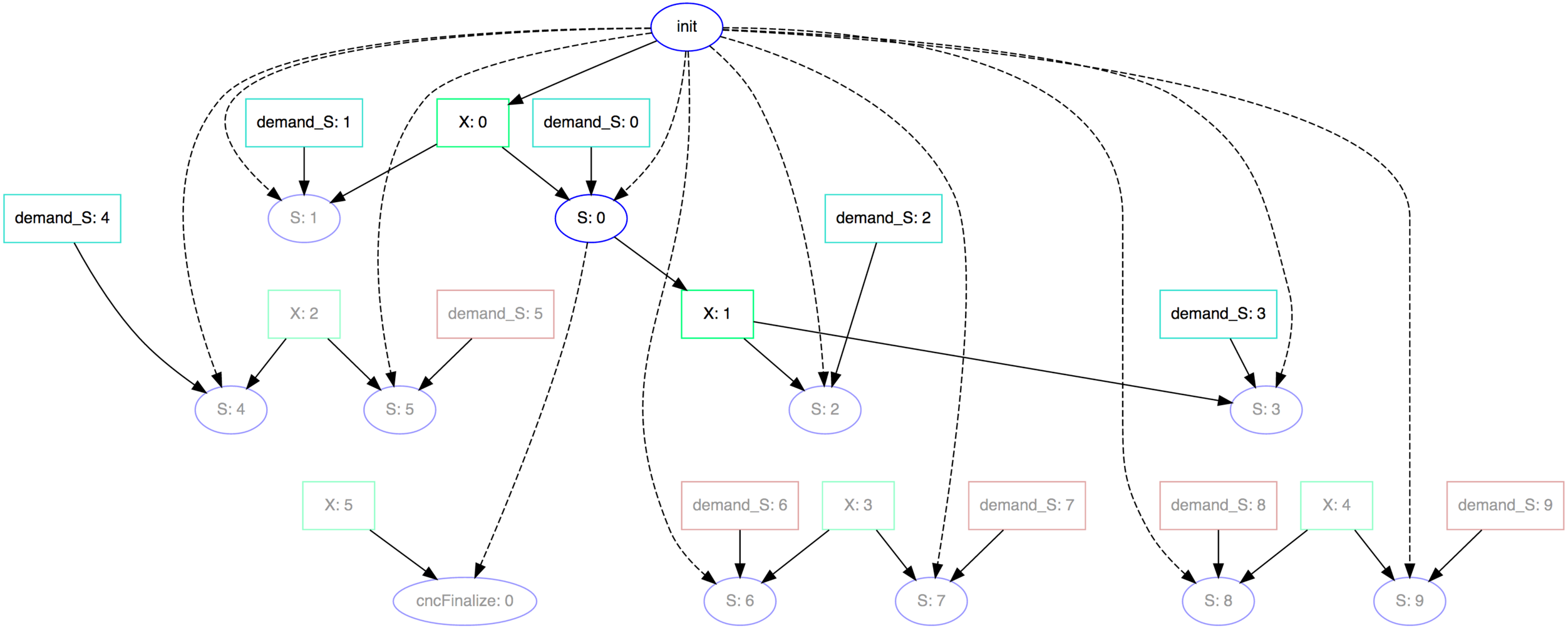

init
X: 0
S: 1
S: 0
X: 2
X: 1
S: 5
S: 4
S: 2
S: 3
X: 5
X: 3
X: 4
S: 6
S: 7
S: 8
S: 9

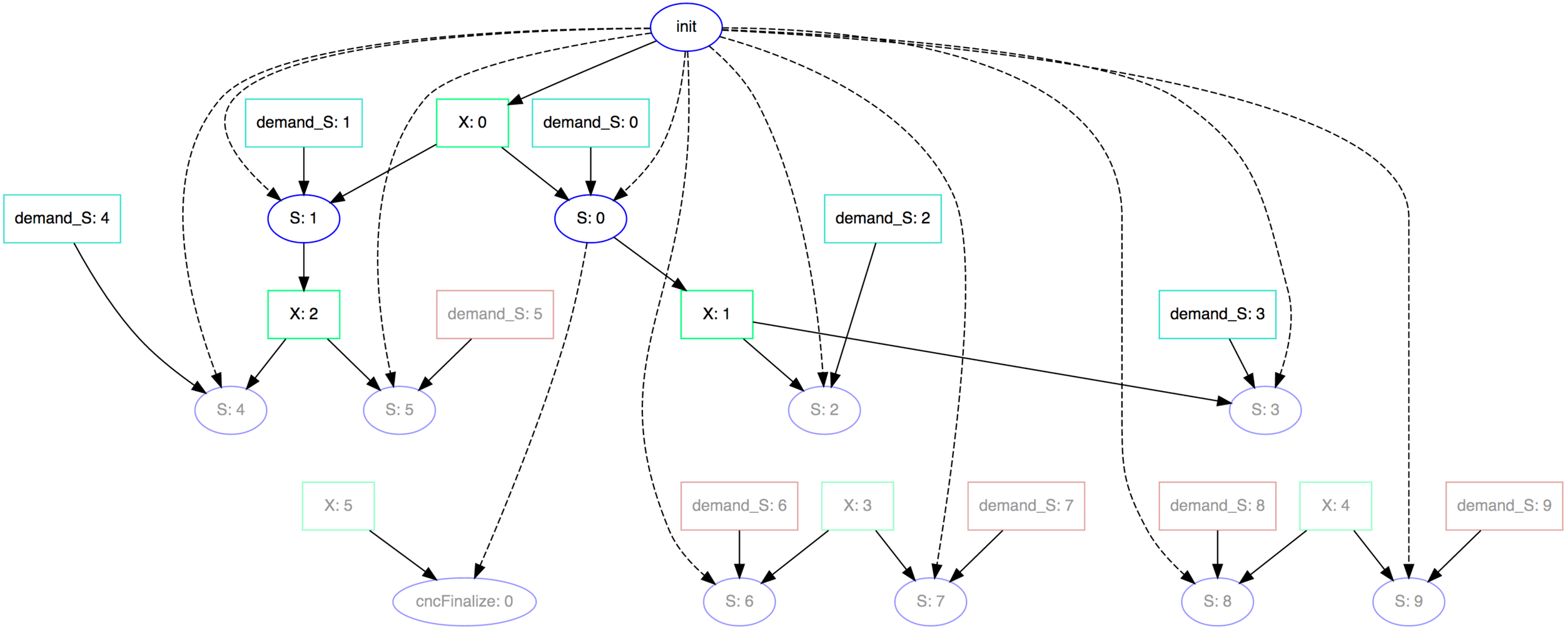

init
X: 0
S: 1
S: 0
X: 2
X: 1
S: 5
S: 4
S: 2
S: 3
X: 5
X: 3
X: 4
cncFinalize: 0
S: 6
S: 7
S: 8
S: 9

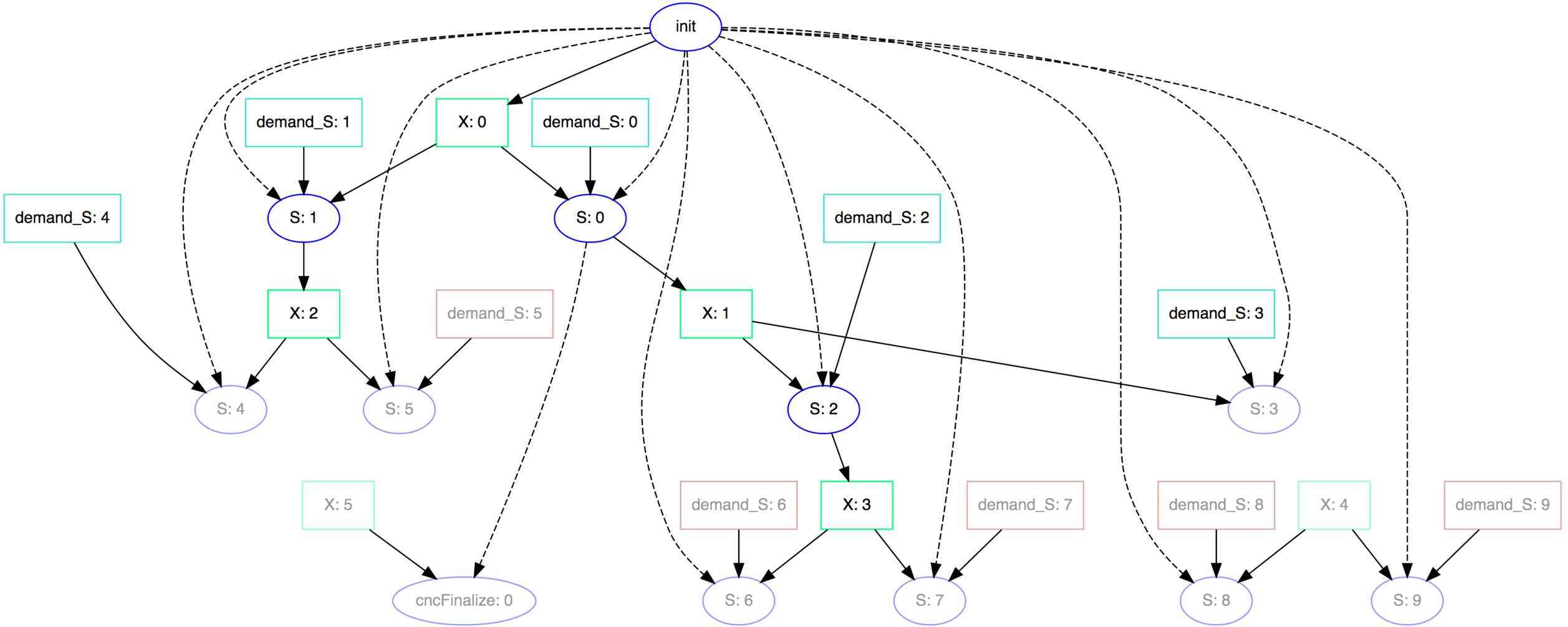

init
X: 0
S: 1
S: 0
X: 2
X: 1
S: 5
S: 4
S: 2
S: 3
X: 6
X: 5
X: 3
X: 4
cncFinalize: 0
S: 6
S: 7
S: 8
S: 9
X: 7
X: 8
X: 9
X: 10

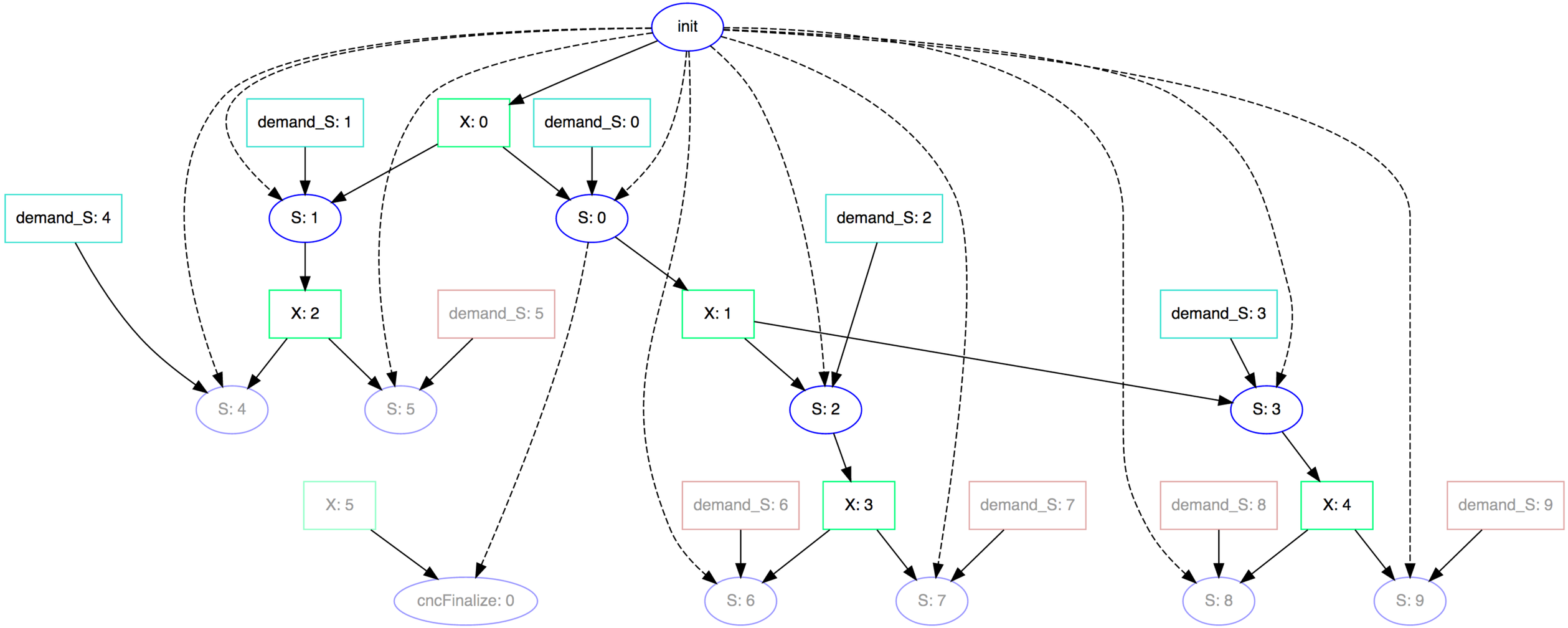
Example

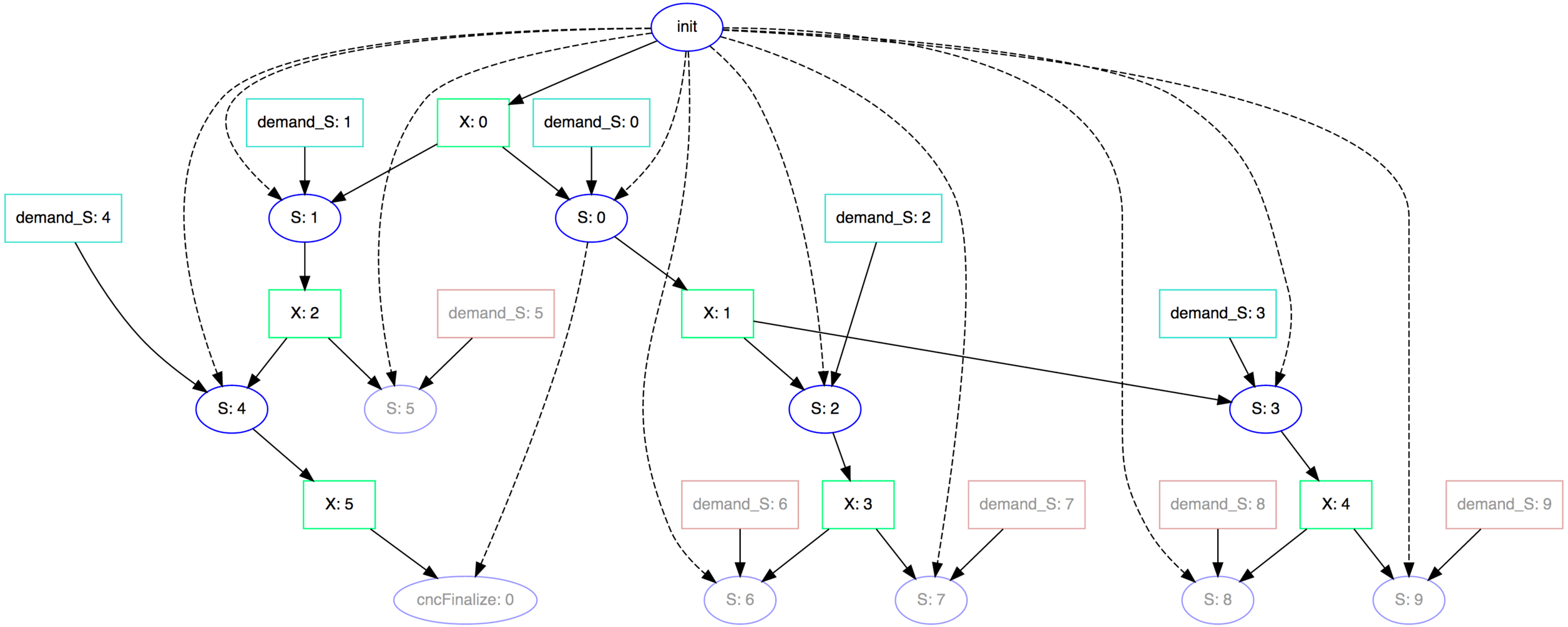
```
[ int X: i ];


( $initialize: () ) -> ( S: $range(0, 10) ), [ X: 0 ];


( S: i )
    <- [ X: i / 2 ]
    -> [ X: i + 1 ];


( $finalize: () ) <- [ X: 5 ];
```

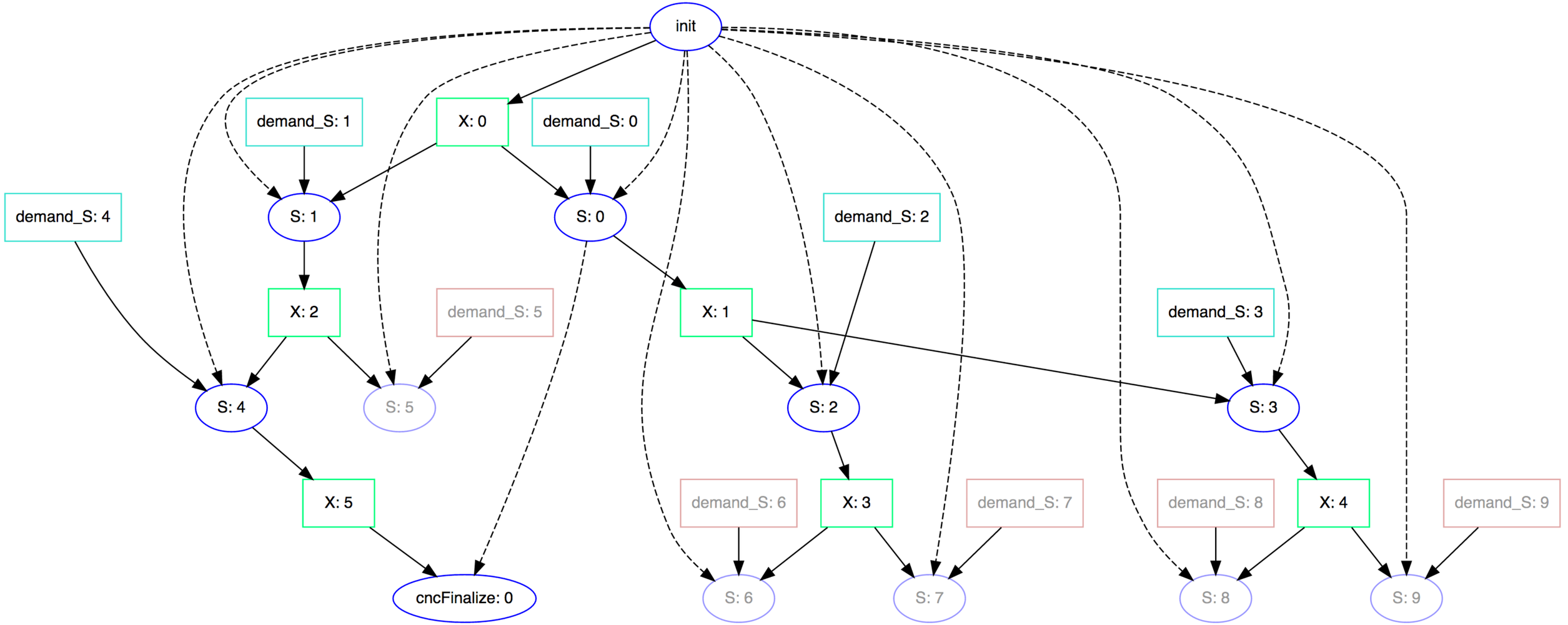

init
demand_S: 1
X: 0
demand_S: 0
demand_S: 4
S: 1
S: 0
demand_S: 2
X: 2
demand_S: 5
X: 1
demand_S: 3
S: 4
S: 5
S: 2
S: 3
X: 5
demand_S: 6
X: 3
demand_S: 7
demand_S: 8
X: 4
demand_S: 9
cncFinalize: 0
S: 6
S: 7
S: 8
S: 9


init
demand_S: 1
X: 0
demand_S: 0
demand_S: 4
S: 1
S: 0
demand_S: 2
X: 2
demand_S: 5
X: 1
demand_S: 3
S: 4
S: 5
S: 2
S: 3
X: 5
demand_S: 6
X: 3
demand_S: 7
demand_S: 8
X: 4
demand_S: 9
cncFinalize: 0
S: 6
S: 7
S: 8
S: 9


init
demand_S: 1
X: 0
demand_S: 0
demand_S: 4
S: 1
S: 0
demand_S: 2
X: 2
demand_S: 5
X: 1
demand_S: 3
S: 4
S: 5
S: 2
S: 3
X: 5
demand_S: 6
X: 3
demand_S: 7
demand_S: 8
X: 4
demand_S: 9
cncFinalize: 0
S: 6
S: 7
S: 8
S: 9


init
demand_S: 1
X: 0
demand_S: 0
demand_S: 4
S: 1
S: 0
demand_S: 2
X: 2
demand_S: 5
X: 1
demand_S: 3
S: 4
S: 5
S: 2
S: 3
X: 5
demand_S: 6
X: 3
demand_S: 7
demand_S: 8
X: 4
demand_S: 9
cncFinalize: 0
S: 6
S: 7
S: 8
S: 9


init
demand_S: 1
X: 0
demand_S: 0
demand_S: 4
S: 1
S: 0
demand_S: 2
X: 2
demand_S: 5
X: 1
demand_S: 3
S: 4
S: 5
S: 2
S: 3
X: 5
demand_S: 6
X: 3
demand_S: 7
demand_S: 8
X: 4
demand_S: 9
cncFinalize: 0
S: 6
S: 7
S: 8
S: 9

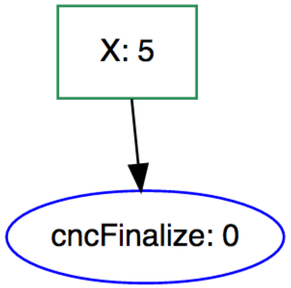

init
demand_S: 1
X: 0
demand_S: 0
demand_S: 4
S: 1
S: 0
demand_S: 2
X: 2
demand_S: 5
X: 1
demand_S: 3
S: 4
S: 5
S: 2
S: 3
X: 5
demand_S: 6
X: 3
demand_S: 7
demand_S: 8
X: 4
demand_S: 9
cncFinalize: 0
S: 6
S: 7
S: 8
S: 9

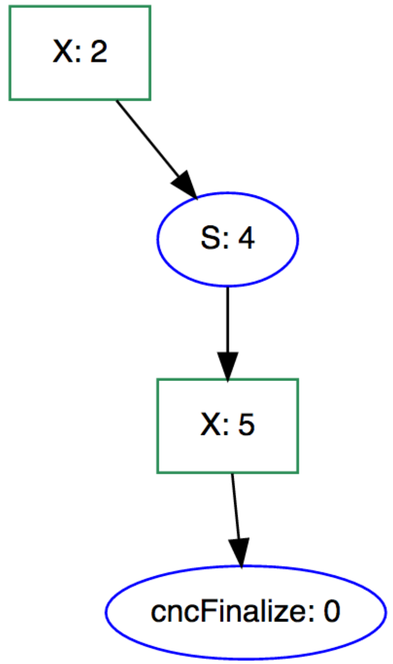
Future directions

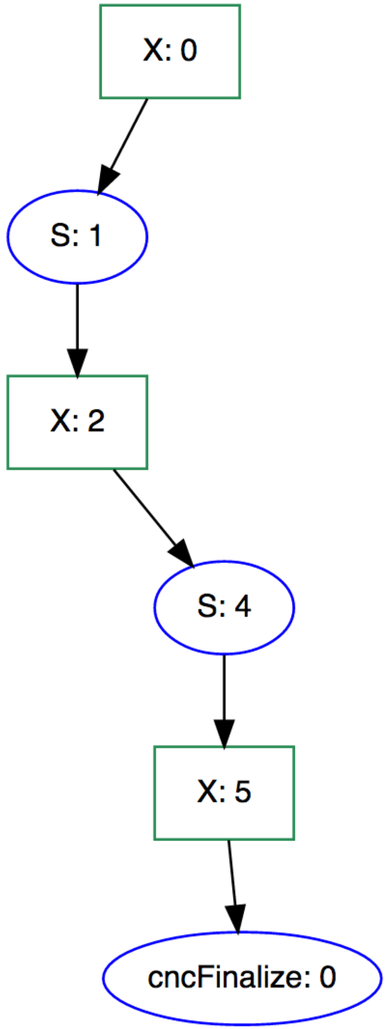
- Statically compute entire execution for some programs
- Combine with speculative execution

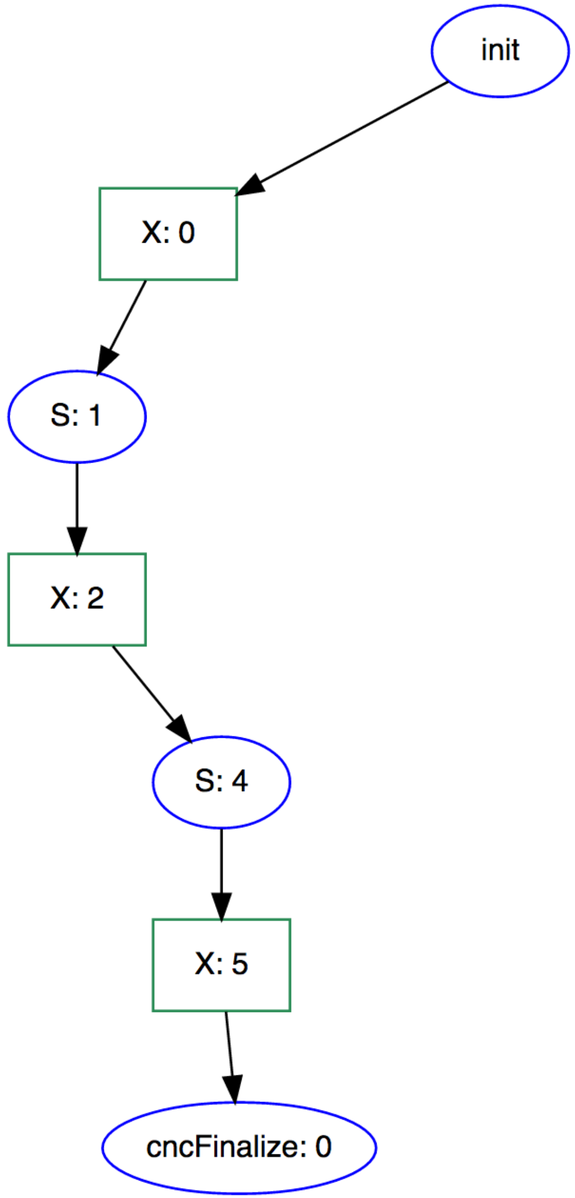
Future directions

- If all collections unambiguous,
  - Derive demand from finalize step back to inputs
  - Simulate entire execution graph before any execution
  - Steps and items visited are necessary to reach finalize
    - Eliminates prescribes, become pure dataflow (I/O) model


X: 5
cncFinalize: 0


X: 2
S: 4
X: 5
cncFinalize: 0


X: 0
S: 1
X: 2
S: 4
X: 5
cncFinalize: 0


init
X: 0
S: 1
X: 2
S: 4
X: 5
cncFinalize: 0

Demand with speculation

- Speculative execution: run steps whose inputs are available, but the program has not requested
- Used if program has both unambiguous and ambiguous collections
- Priority scheme
  - 1. Demanded steps; their output is necessary
  - 2. Regular prescribed steps; user thinks their output is necessary
  - 3. Speculative steps; only if we have extra resources

Wrap up

- Step inverses
- Demand driven execution
- Future possible work