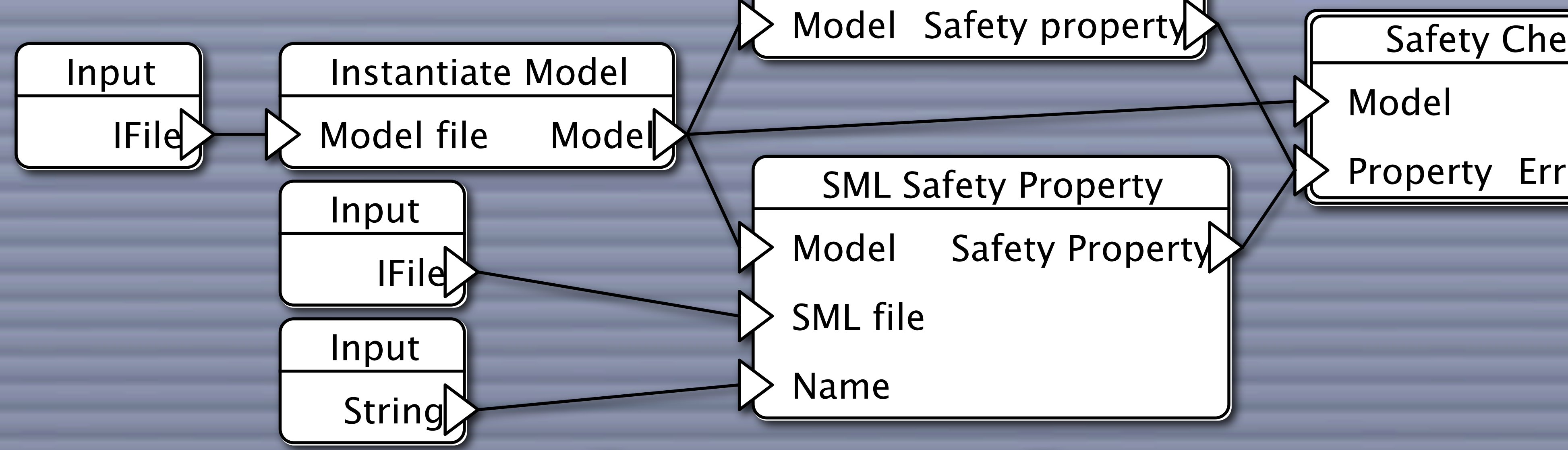
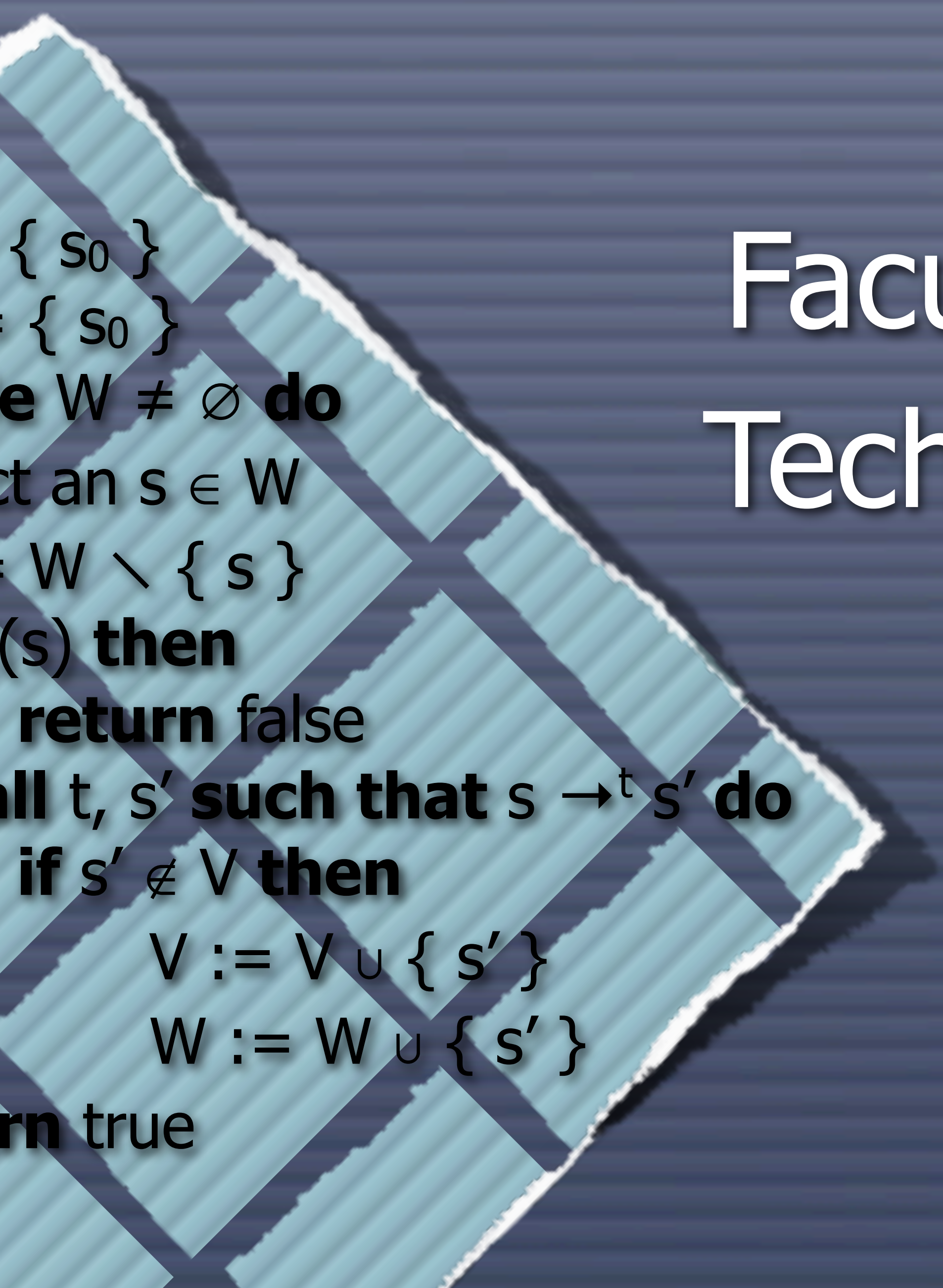
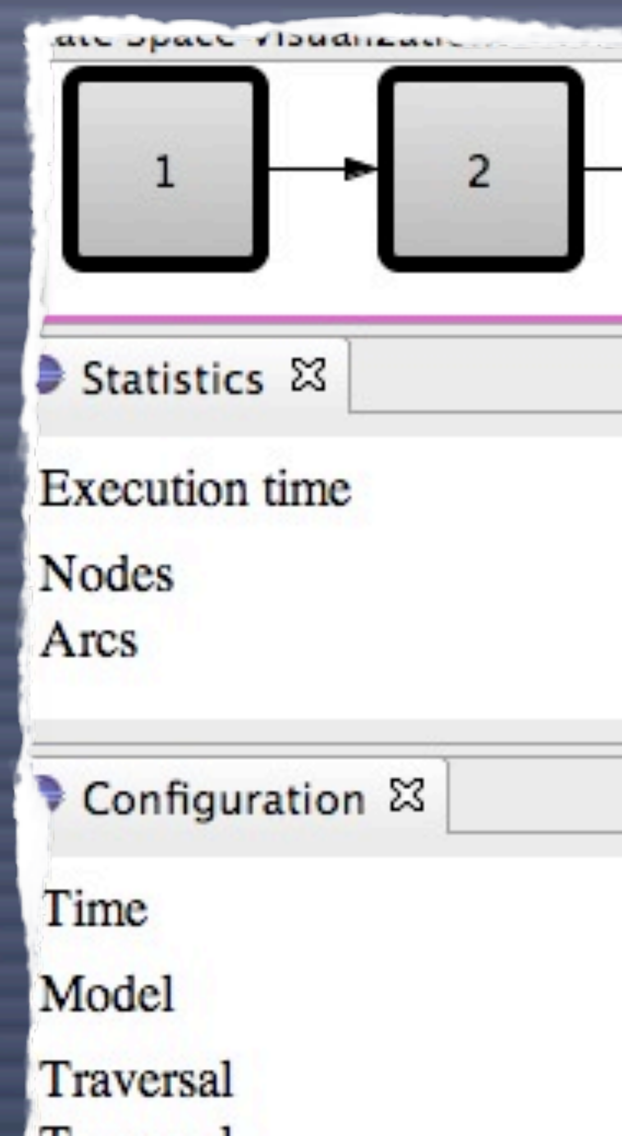


State Space Exploration and ASAP: User Perspective

Michael Westergaard
Faculteit Wiskunde & Informatica
Technische Universiteit Eindhoven
m.westergaard@tue.nl

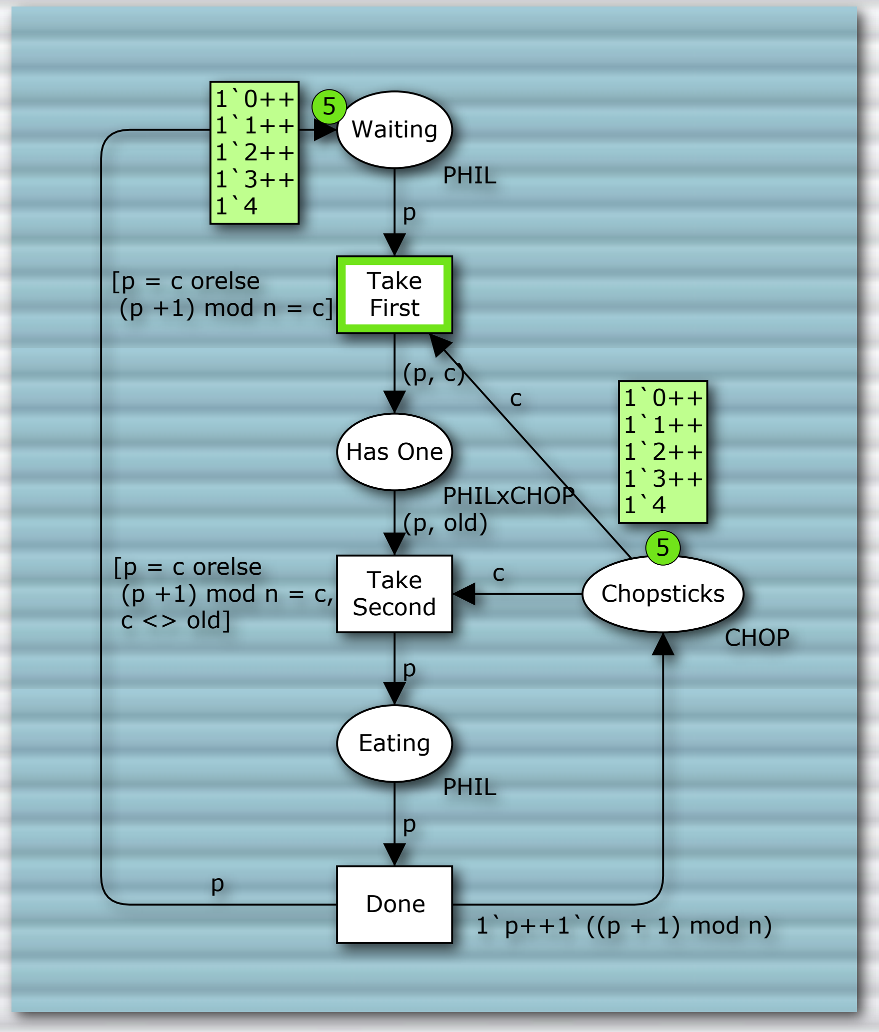




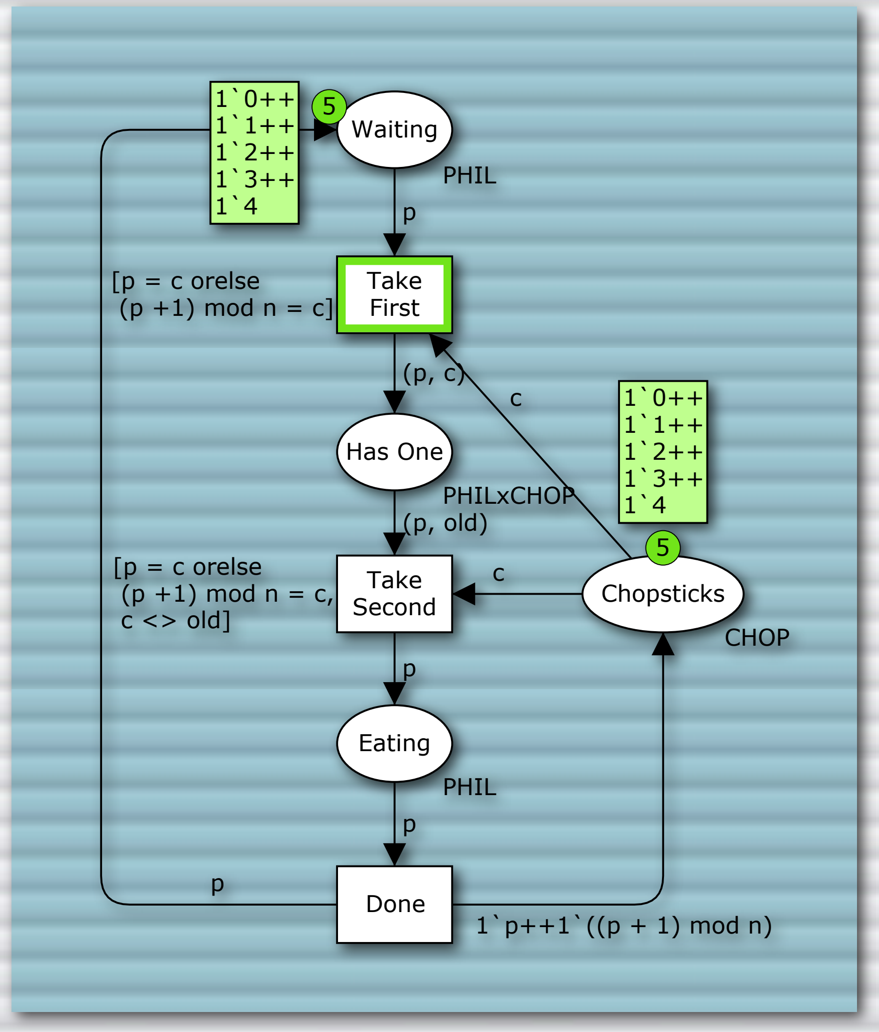
Outline

- User perspective
- JoSEL
- Safety Properties
- Simple reduction techniques
- Linear temporal logic (LTL)

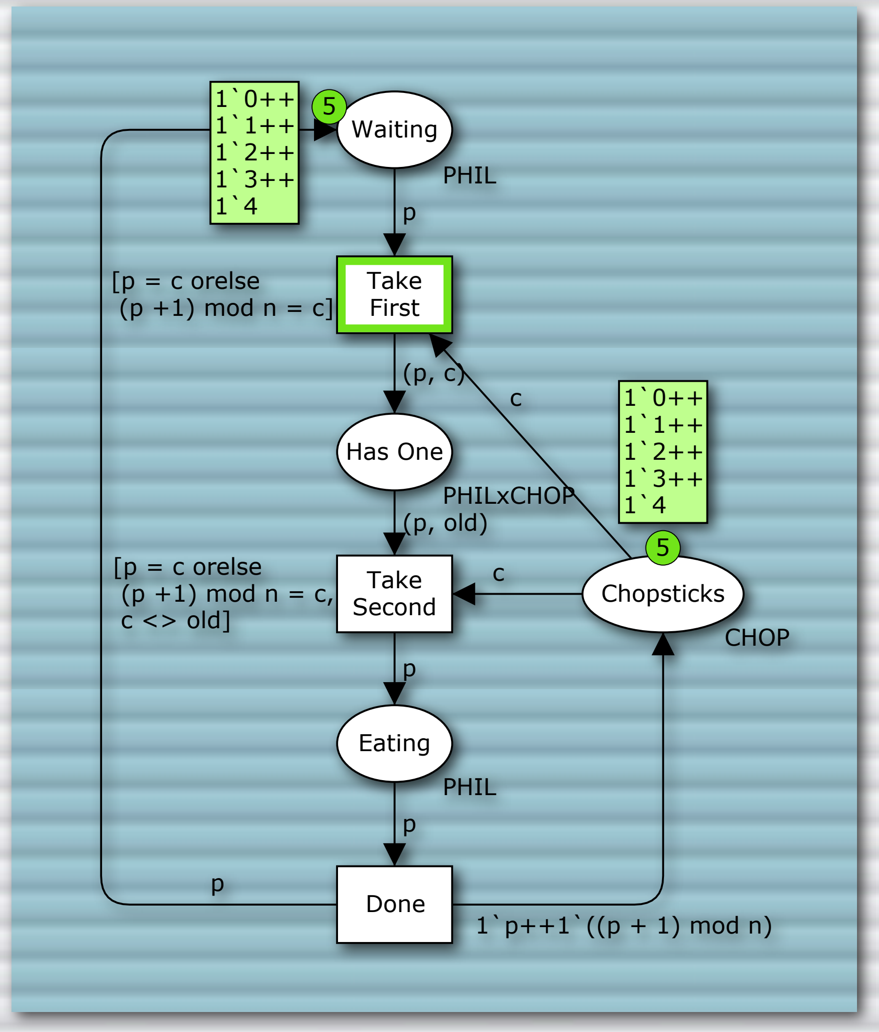
User Perspective

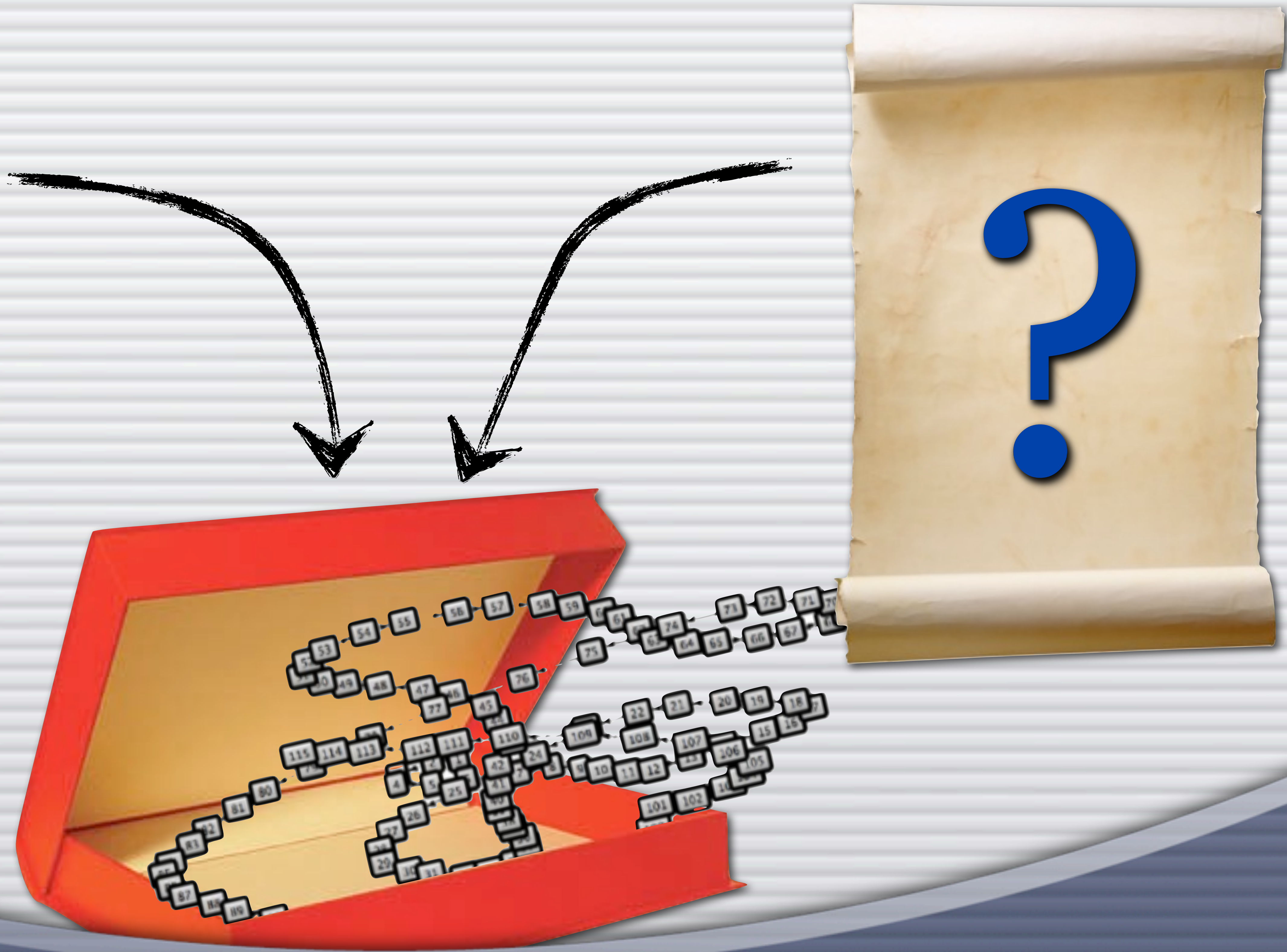


User Perspective



User Perspective

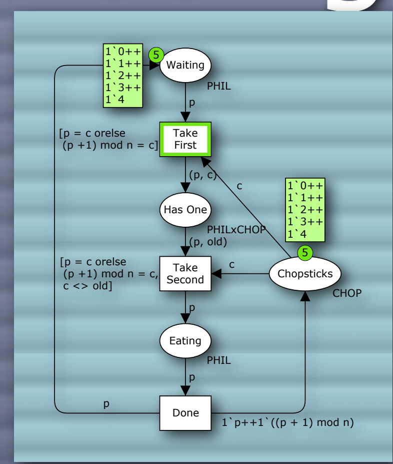




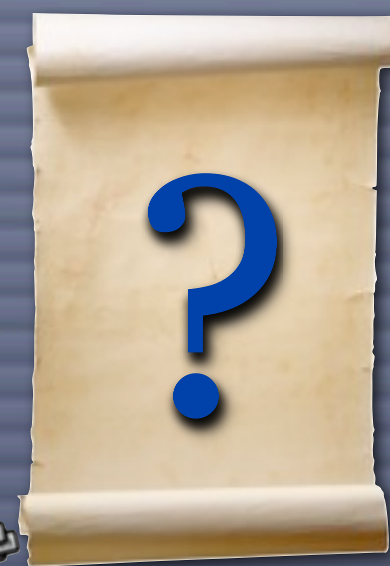
User Perspective

Verification Project

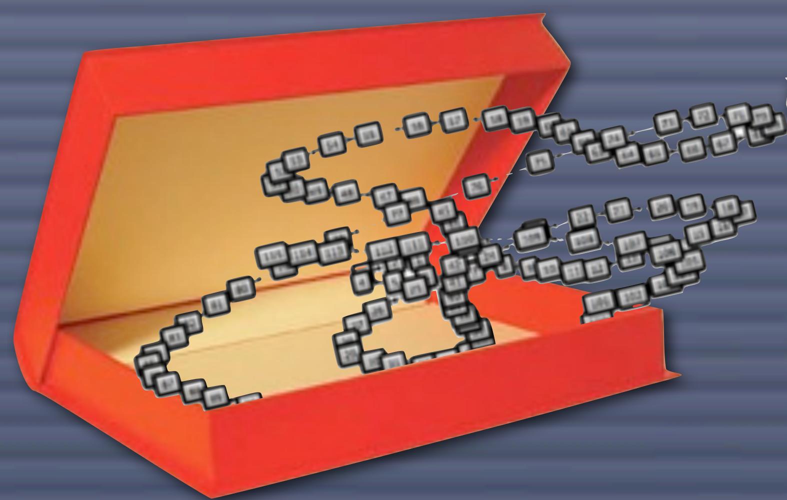
- Verification of a a model is done using verification projects consisting of
  - CPN **Models** to be analyzed



  - **Queries** expressing the properties we are interested in

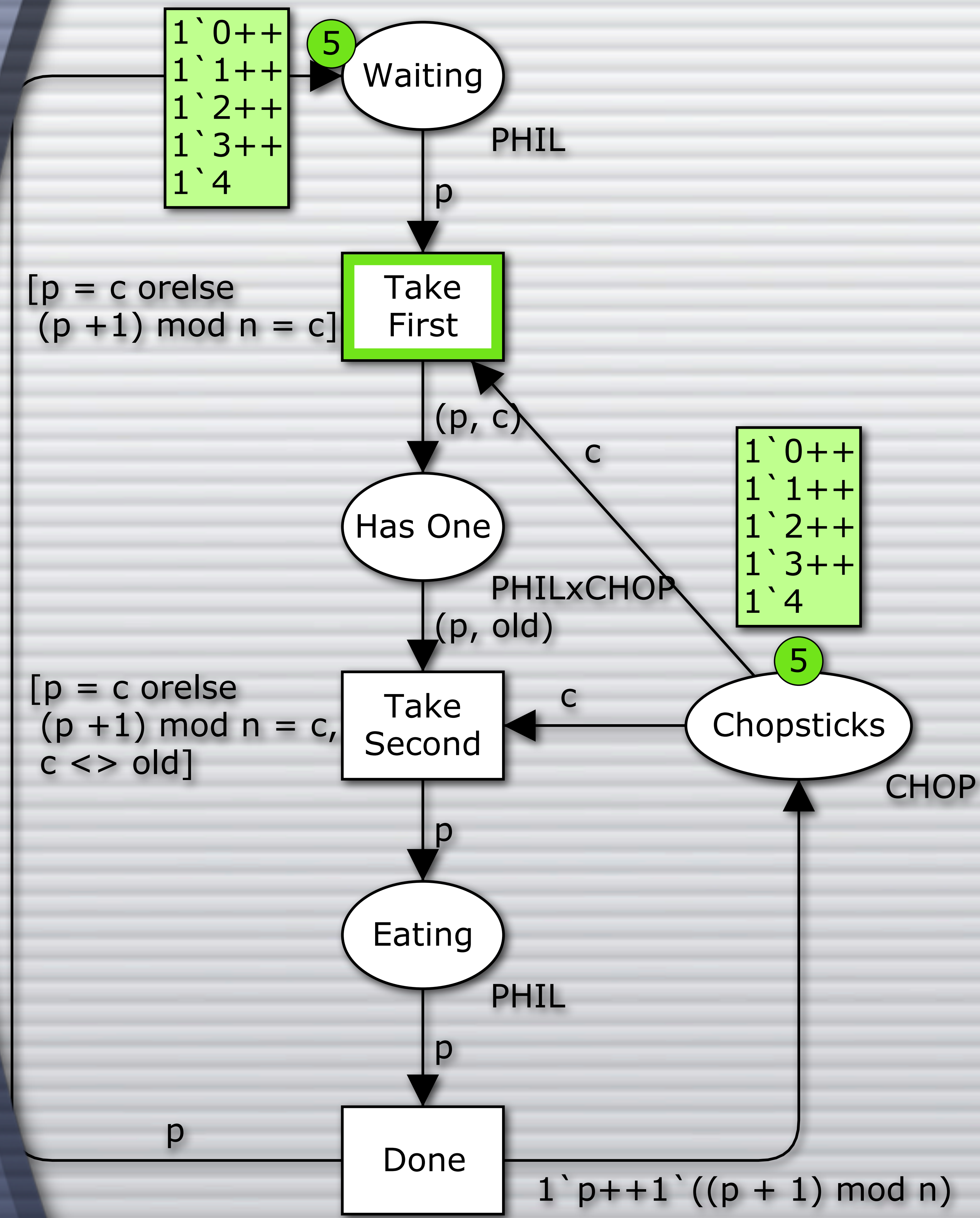


  - **Reports** reflecting results of the queries



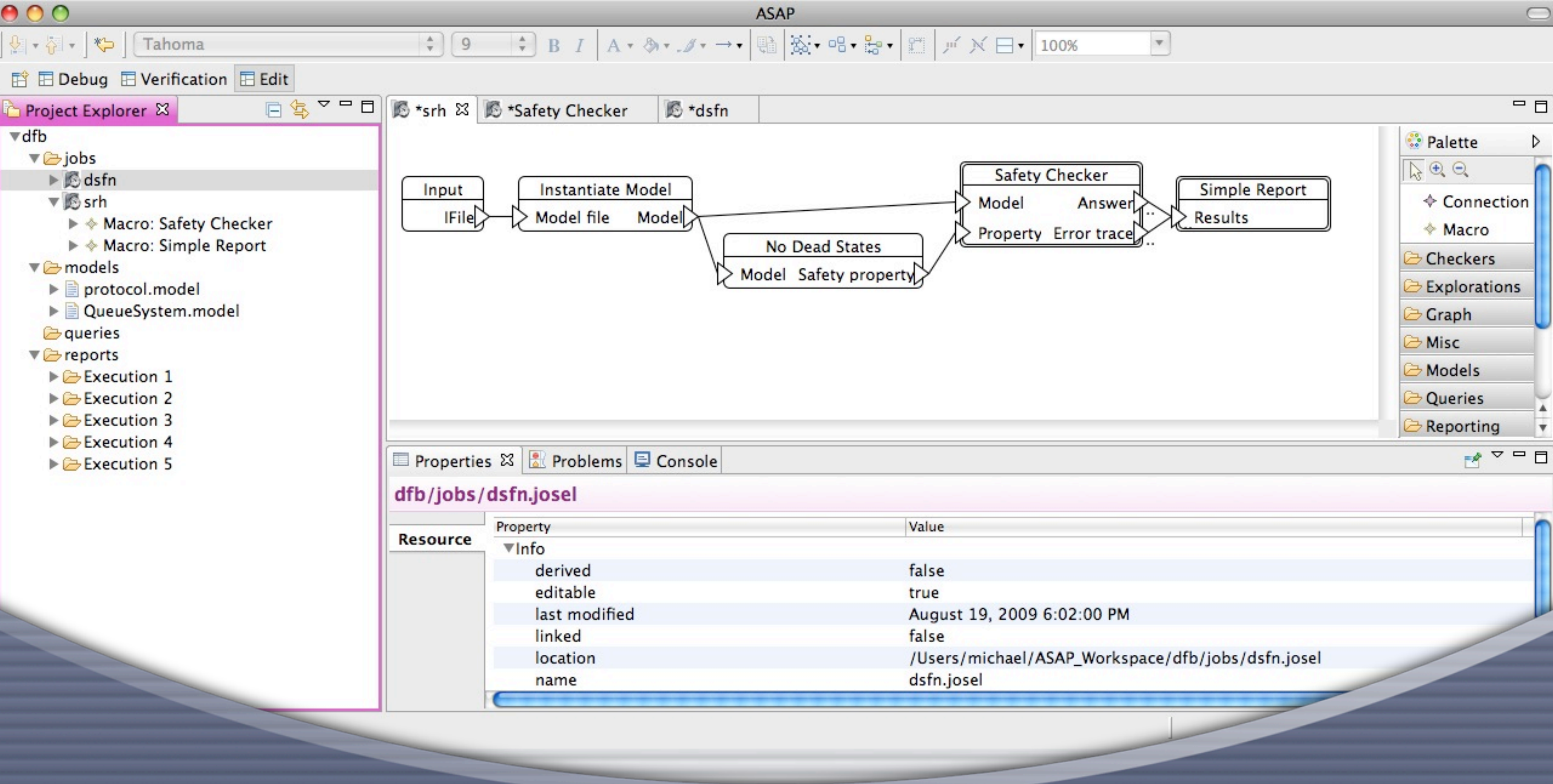
  - How to obtain results from models and queries is described using **verification jobs**

Example: Dining Philosophers



Demo:
Dining Philosophers

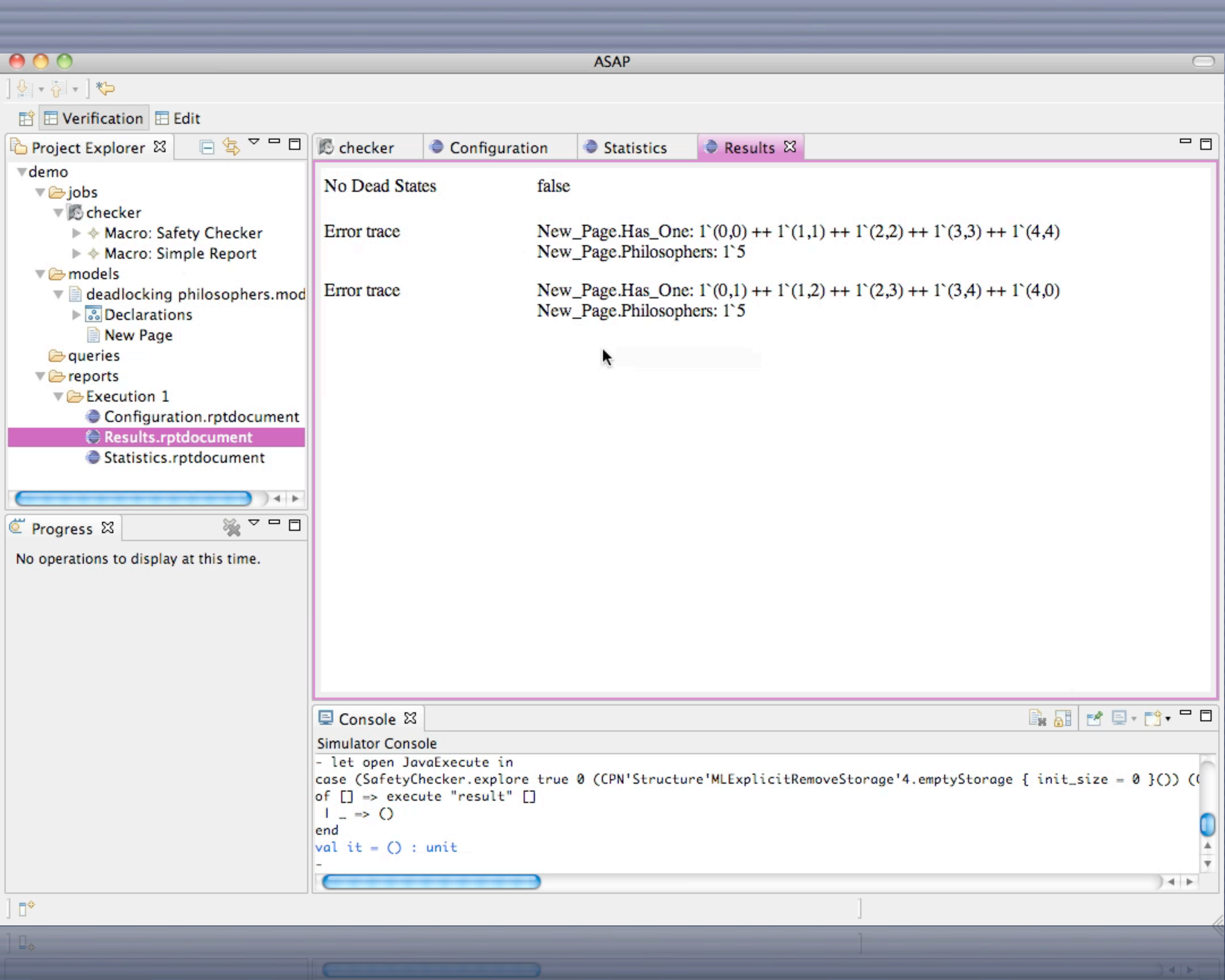
- Do a bit of simple simulation

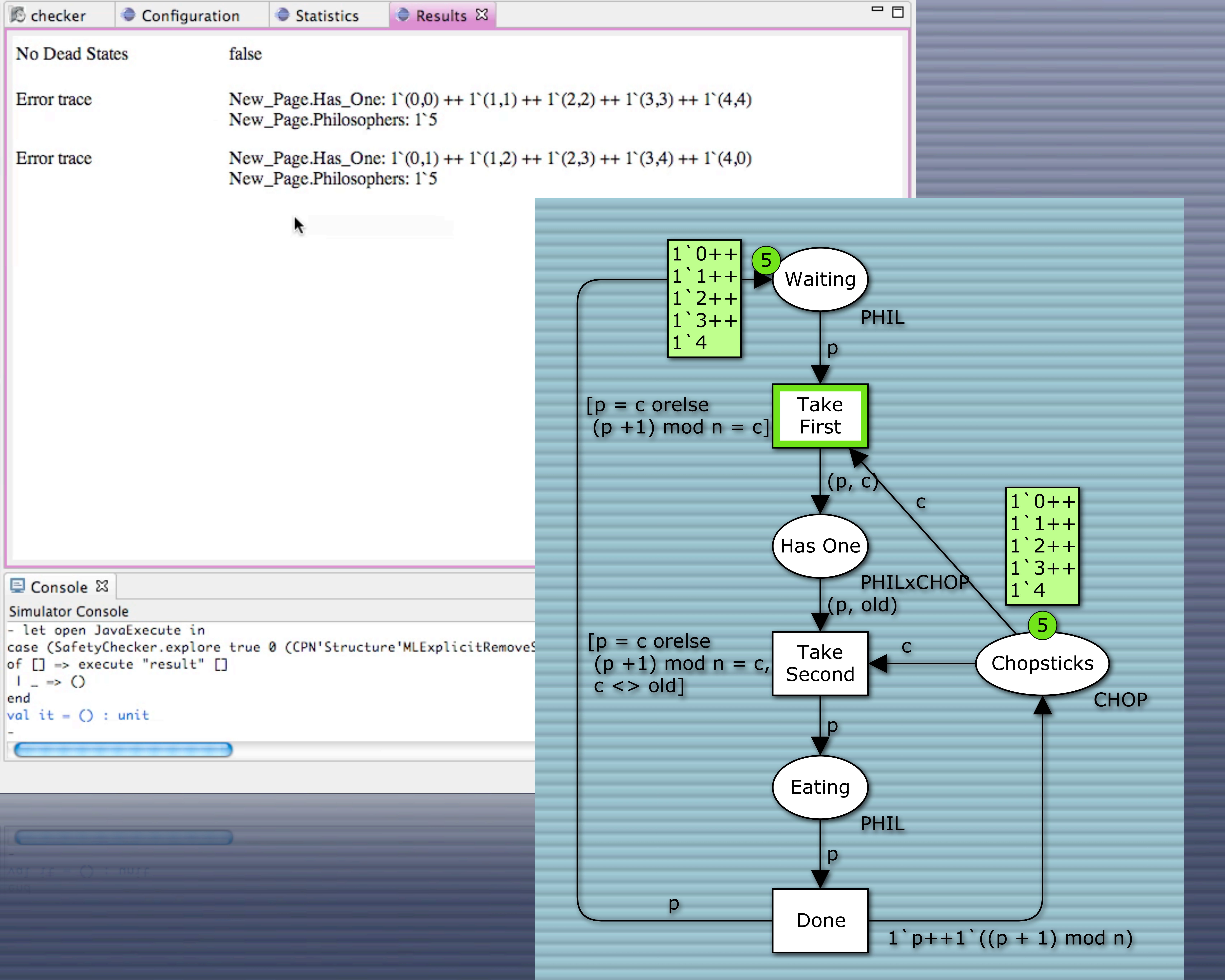


Example:
Check for Deadlocks

Demo:
Check for Deadlocks

- Creation of Verification project
- Loading models
- Creating a Verification job from a template
- Executing a job template
- Reporting


ASAP
Verification
Edit
Project Explorer
demo
jobs
checker
Macro: Safety Checker
Macro: Simple Report
models
deadlocking philosophers.mod
Declarations
New Page
queries
reports
Execution 1
Configuration.rptdocument
Results.rptdocument
Statistics.rptdocument
Progress
No operations to display at this time.
checker
Configuration
Statistics
Results
No Dead States false
Error trace New_Page.Has_One: 1`(0,0) ++ 1`(1,1) ++ 1`(2,2) ++ 1`(3,3) ++ 1`(4,4)
New_Page.Philosophers: 1`5
Error trace New_Page.Has_One: 1`(0,1) ++ 1`(1,2) ++ 1`(2,3) ++ 1`(3,4) ++ 1`(4,0)
New_Page.Philosophers: 1`5
Console
Simulator Console
- let open JavaExecute in
case (SafetyChecker.explore true 0 (CPN'Structure'MLExplicitRemoveStorage'4.emptyStorage { init_size = 0 }()) ((
of [] => execute "result" []
| _ => ()
end
val it = () : unit
-



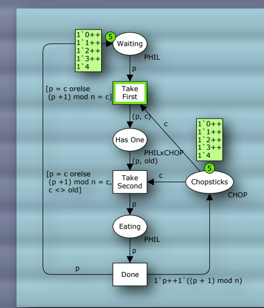
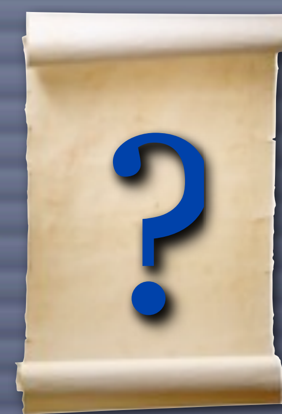
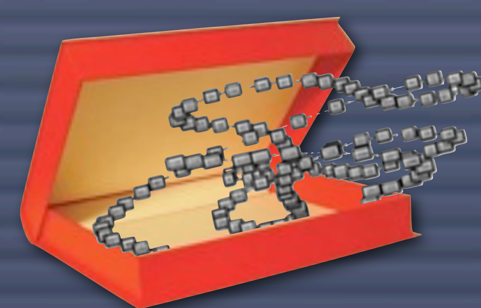
checker
Configuration
Statistics
Results
No Dead States false
Error trace New_Page.Has_One: 1`(0,0) ++ 1`(1,1) ++ 1`(2,2) ++ 1`(3,3) ++ 1`(4,4)
New_Page.Philosophers: 1`5
Error trace New_Page.Has_One: 1`(0,1) ++ 1`(1,2) ++ 1`(2,3) ++ 1`(3,4) ++ 1`(4,0)
New_Page.Philosophers: 1`5
Console
Simulator Console
- let open JavaExecute in
case (SafetyChecker.explore true 0 (CPN'Structure'MLExplicitRemoveS
of [] => execute "result" []
| _ => ()
end
val it = () : unit
1`0++
1`1++
1`2++
1`3++
1`4
5
Waiting
PHIL
p
[p = c orelse
(p +1) mod n = c]
Take
First
(p, c)
c
Has One
PHILxCHOP
(p, old)
[p = c orelse
(p +1) mod n = c,
c <> old]
Take
Second
c
Chopsticks
CHOP
p
Eating
PHIL
p
Done
p
1`p++1`((p + 1) mod n)

JoSEL: Background

- ASAP Supports a wide range of state space methods
  - Depth-first and breadth-first traversal
  - On-line and off-line analysis
  - Bit-state hashing and hash compaction
  - Sweep-line and ComBack methods
  - Safety properties, LTL

JoSEL: Background

- Applying a state space methods consists of

1. Specifying a model to analyze 
2. Making queries expressing desired properties
3. Select method to use for verification 
4. Set parameters of and instantiate the selected method 
5. Execute the traversal
6. Post-process and interpret the results 

JoSEL: Aim

- Develop a high-level language making it possible to tie the model, queries and desired state space method together
- Support research, education and industrial application scenarios

JoSEL: Requirements

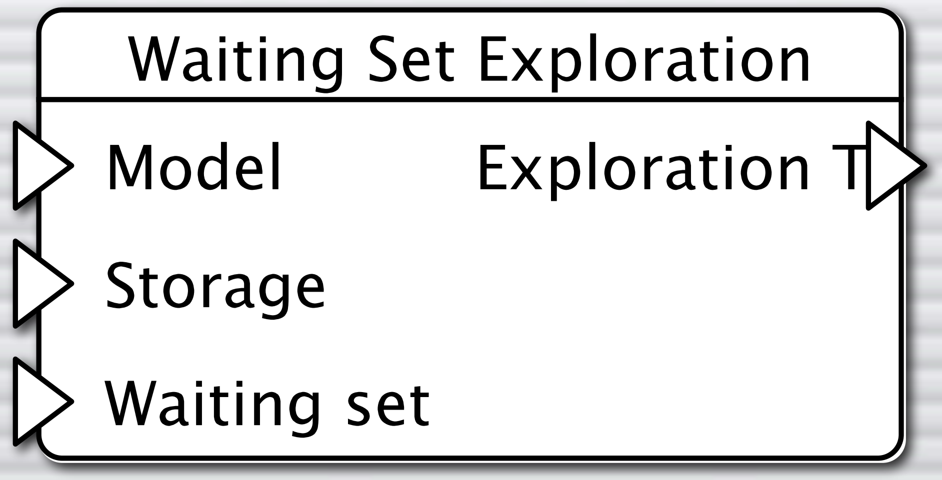
- **Abstraction**: Hide details from users
- **Low-level control**: Make it possible to access details when required for performance

  The hash function used to hash states when storing in a hash table

- **Modularity**: Facilitate construction and use of building blocks (**templates**) in verification jobs
- **Extensibility**: Allow extension for new methods as needed

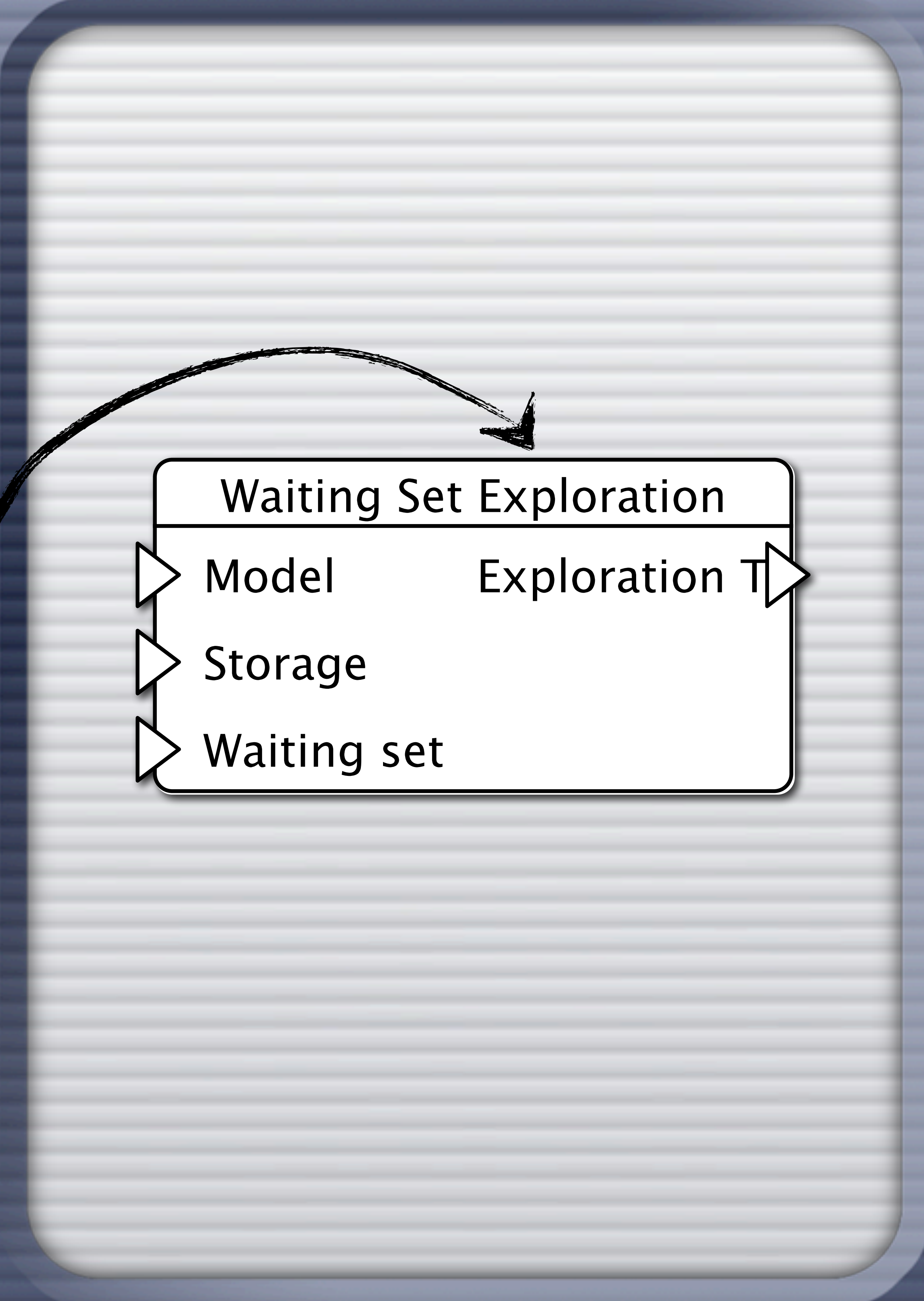
JoSEL Overview

- Graphical language inspired by object flows and hierarchy of CP-nets
- Basic unit is a **task**
- Tasks have typed input and output **ports**



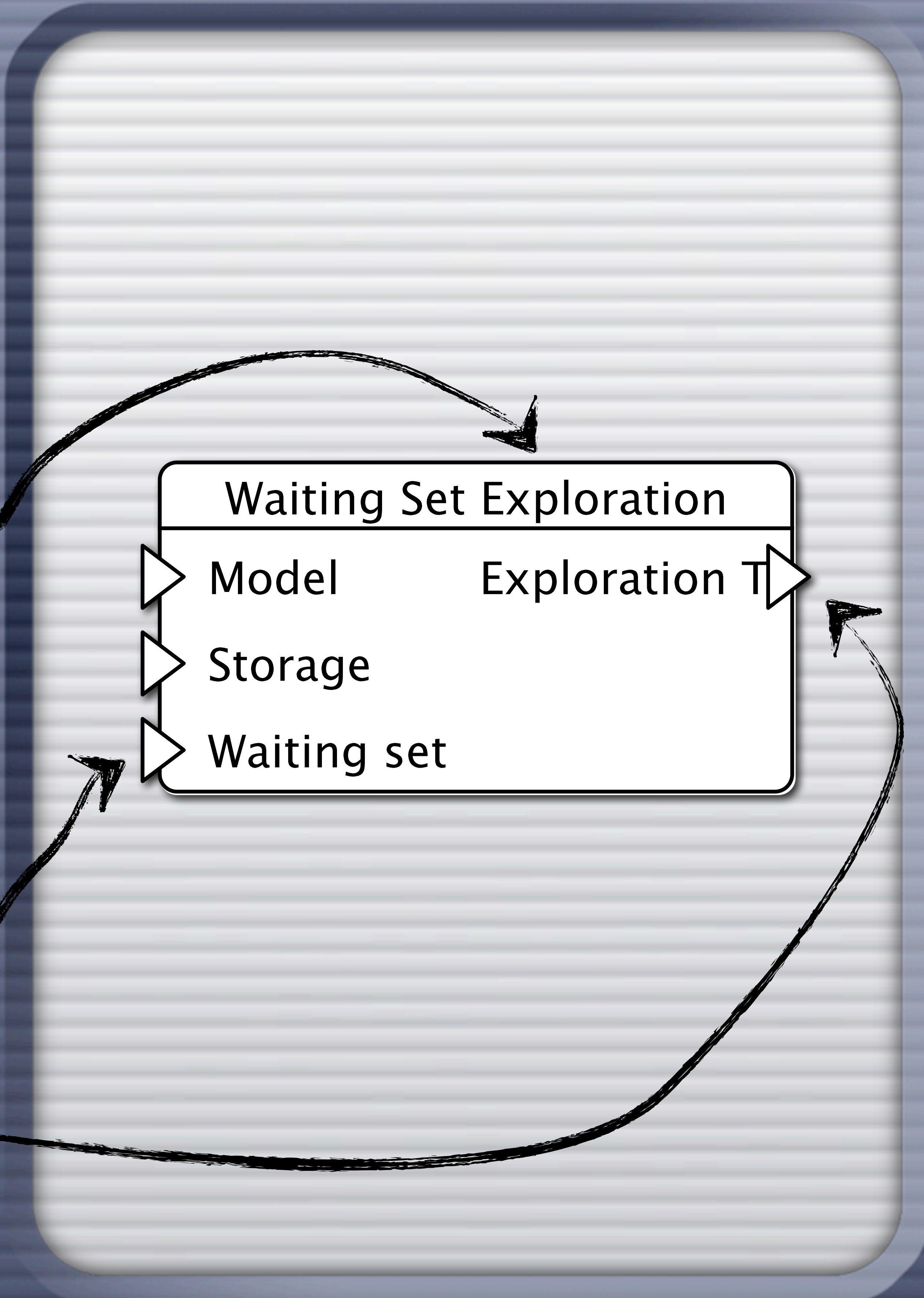
JoSEL Overview

- Graphical language inspired by object flows and hierarchy of CP-nets
- Basic unit is a **task**
- Tasks have typed input and output **ports**

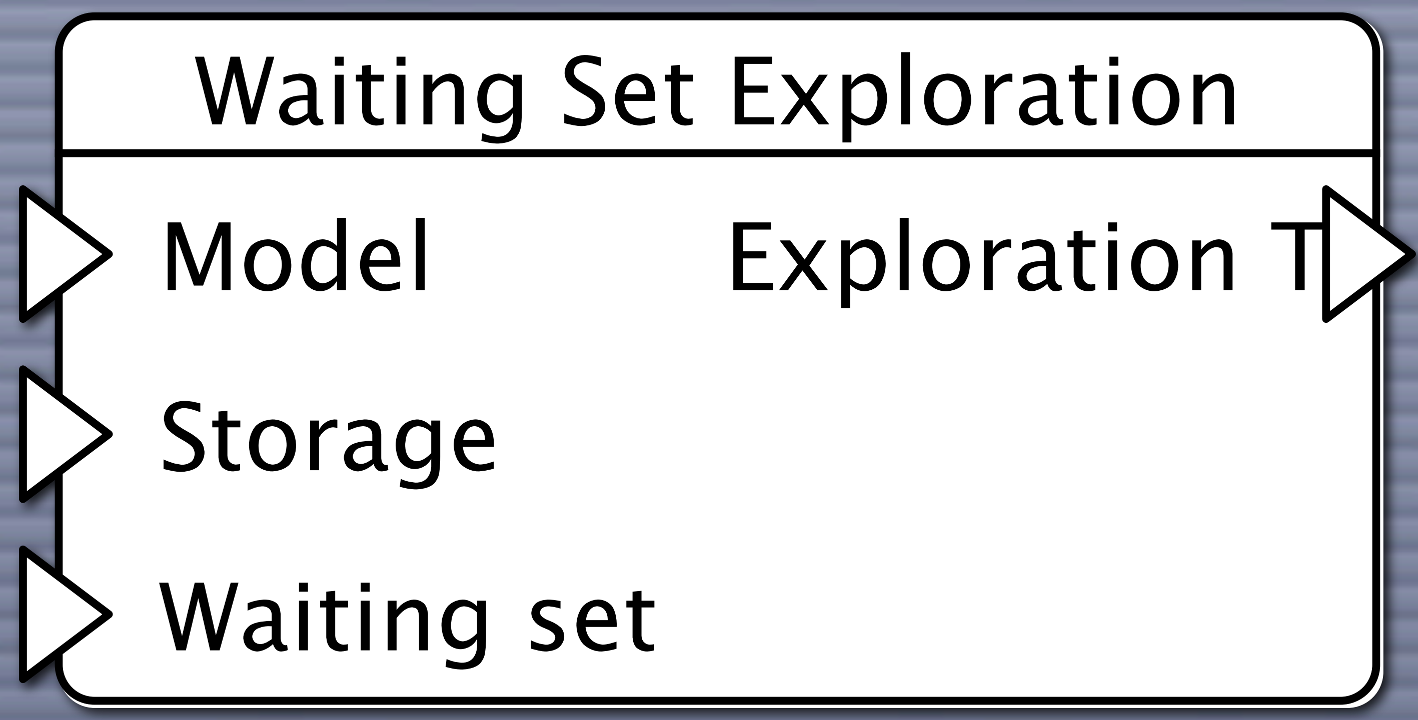


JoSEL Overview

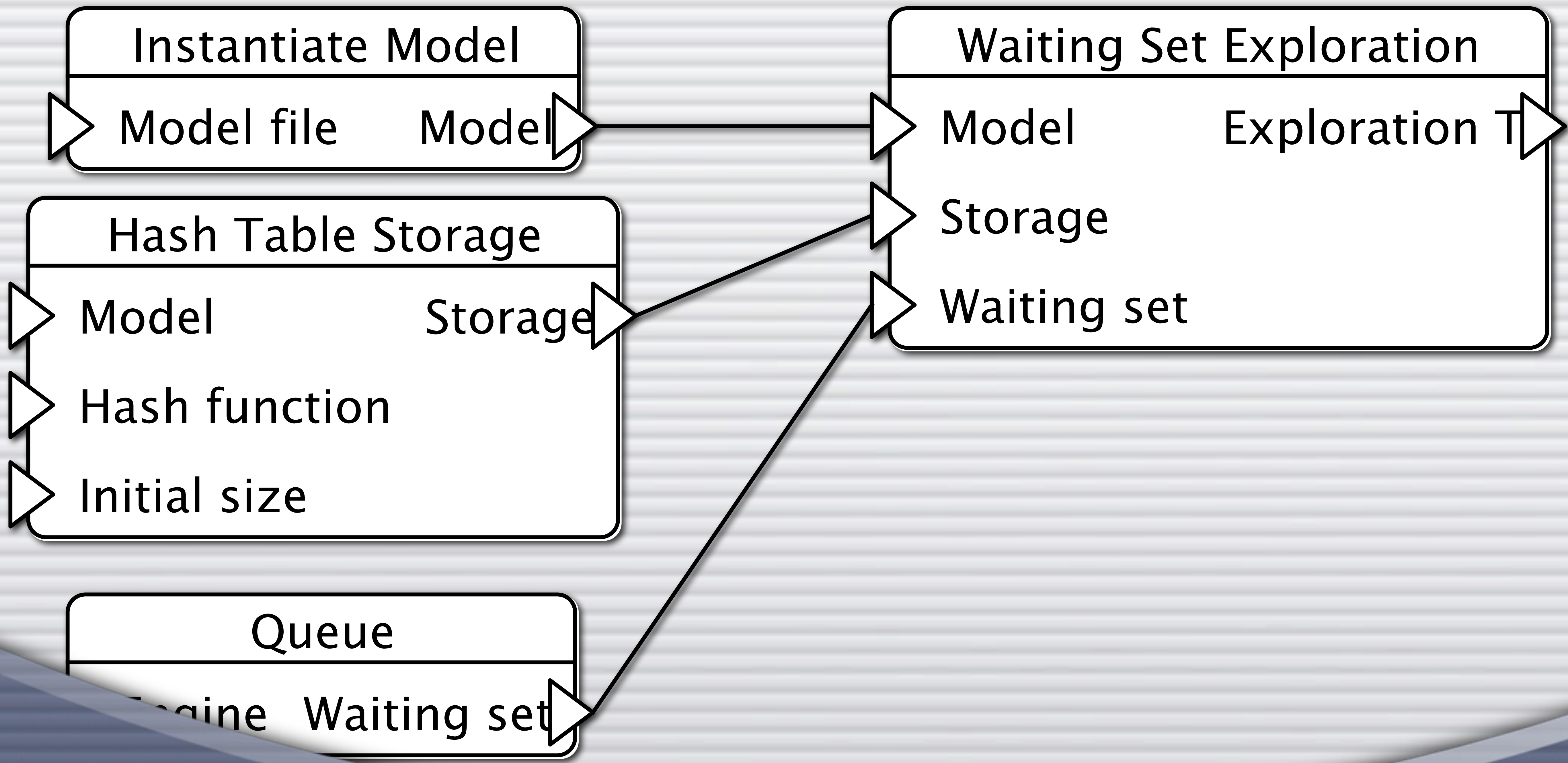
- Graphical language inspired by object flows and hierarchy of CP-nets
- Basic unit is a **task**
- Tasks have typed input and output **ports**



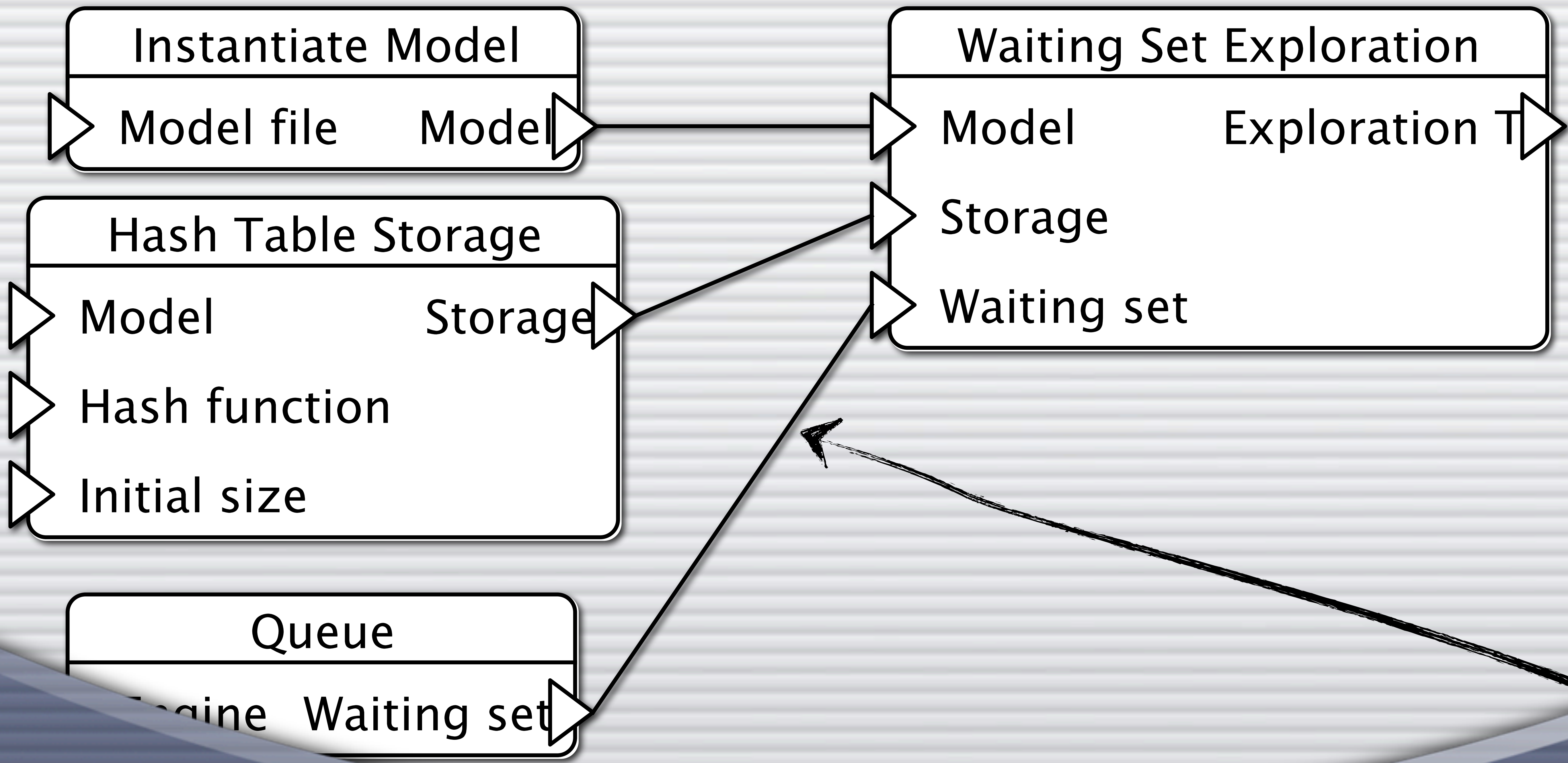
Tasks



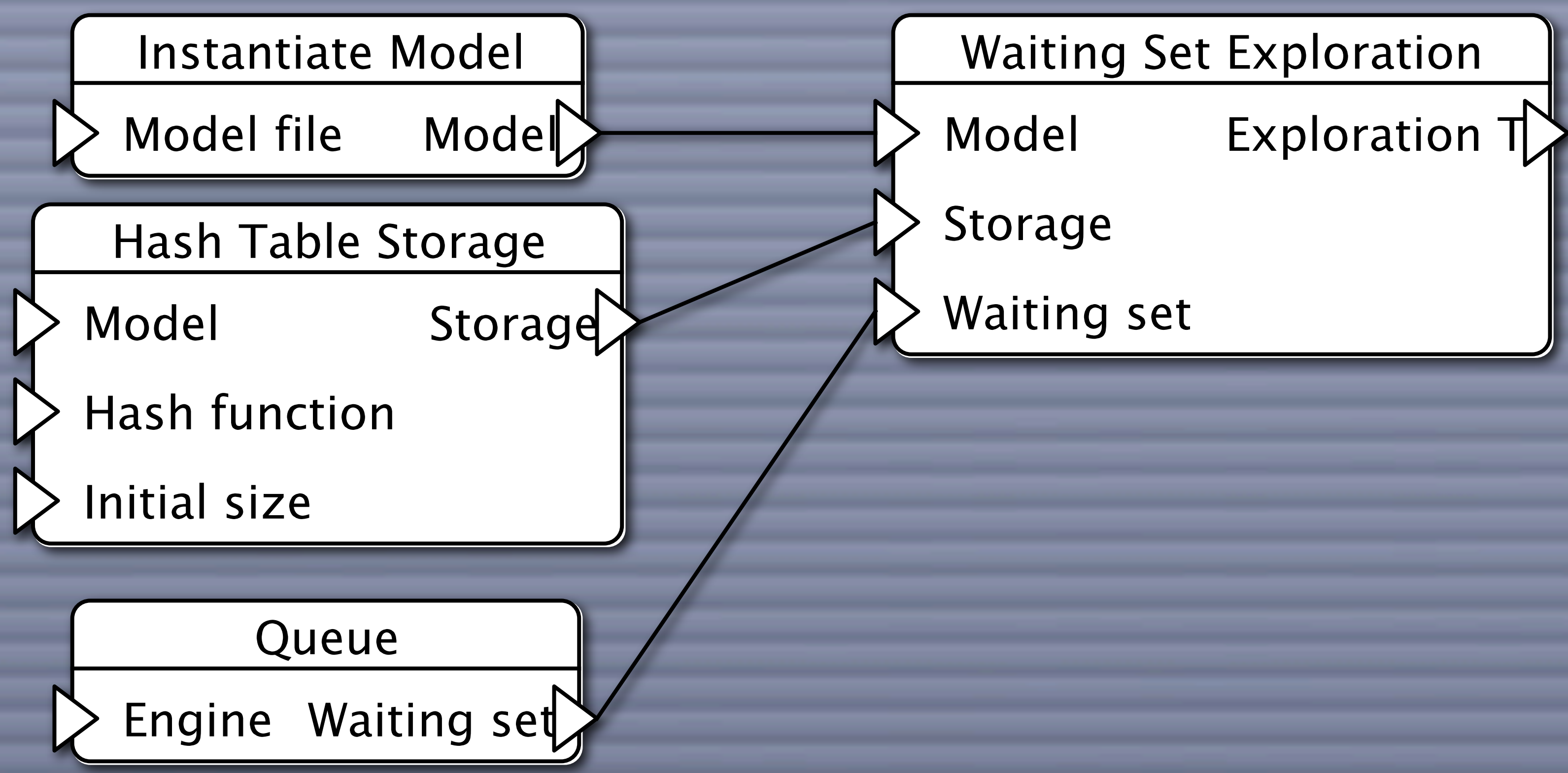
- A task represents a single unit of work/operation: instantiating a data type, loading a file, checking a property, ...
- Input ports represent data required to perform the operation
- Output ports represent data produced by the operation



- Output and input ports can be **connected**
- A **verification job (job)** is a set of tasks and their connections

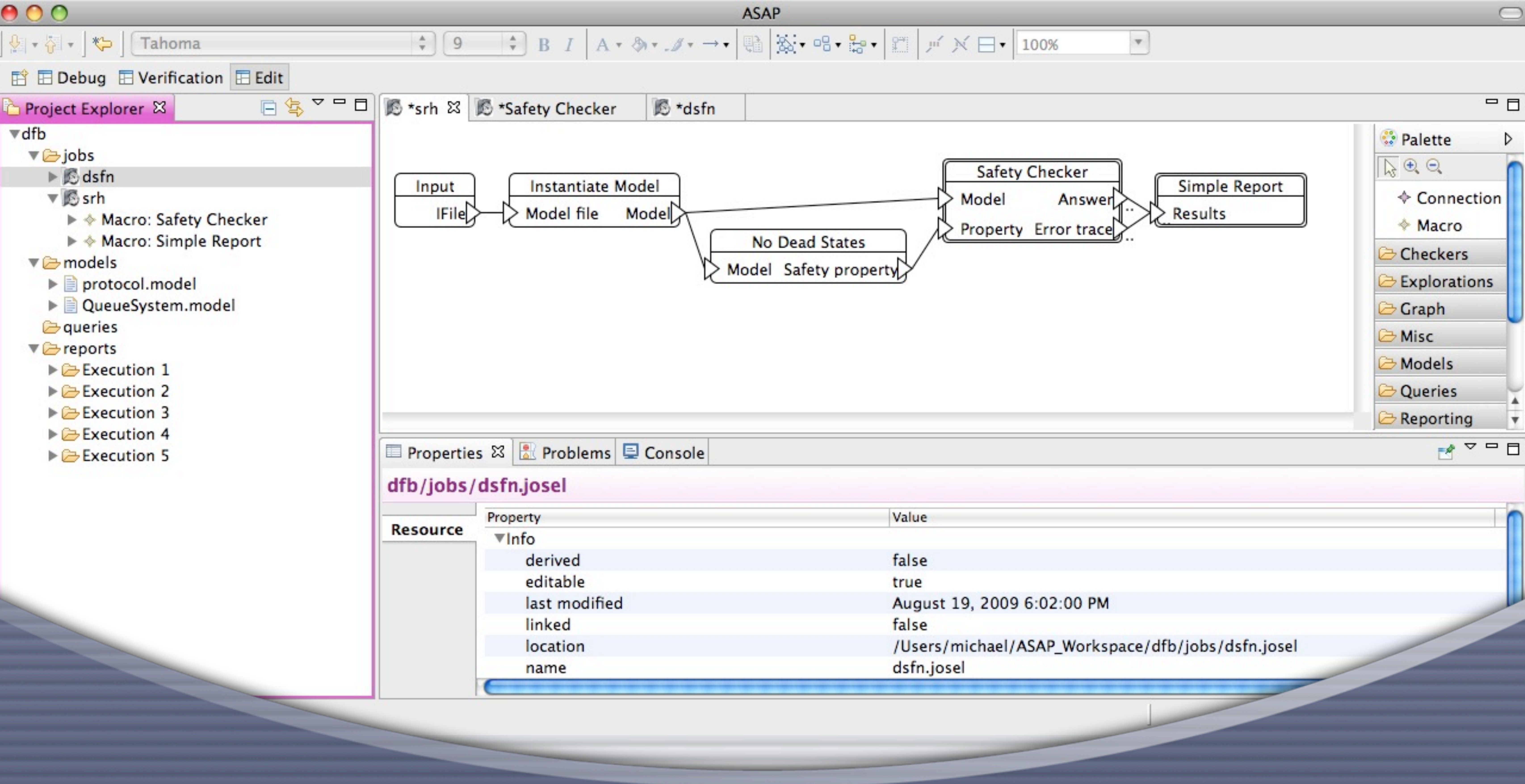


- Output and input ports can be **connected**
- A **verification job (job)** is a set of tasks and their connections

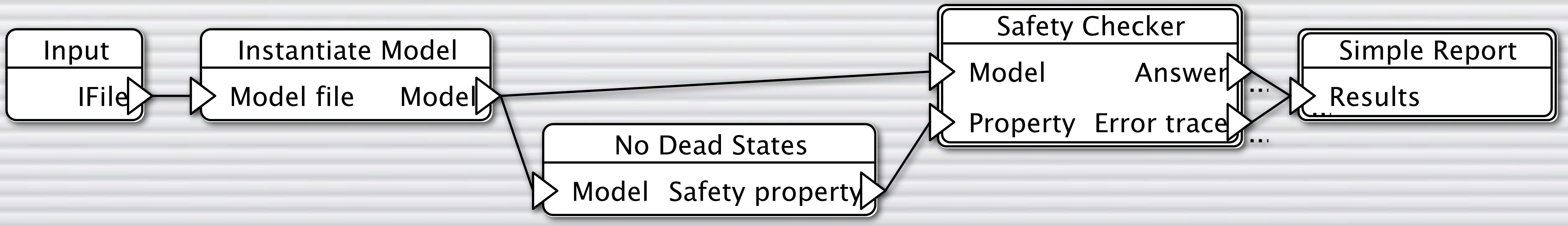


Jobs

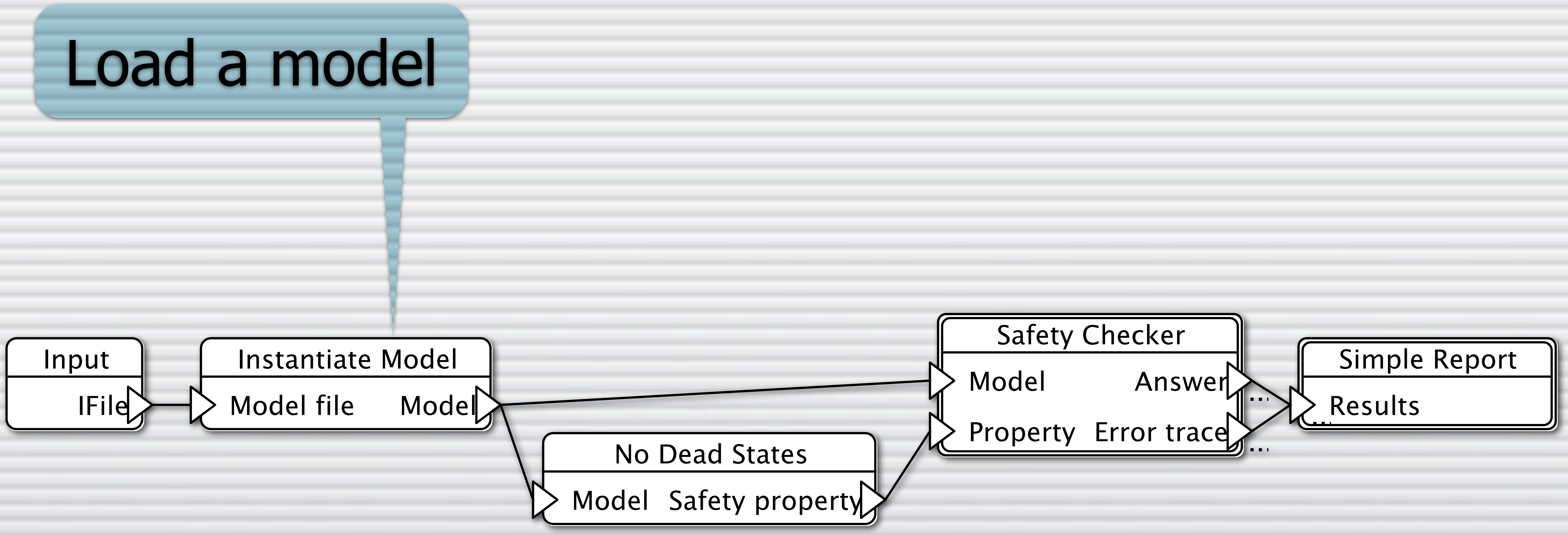
- Connections represent flow of information
- Ports can have multiple connections
  - Can represent split of information
  - Can represent multiple instantiations



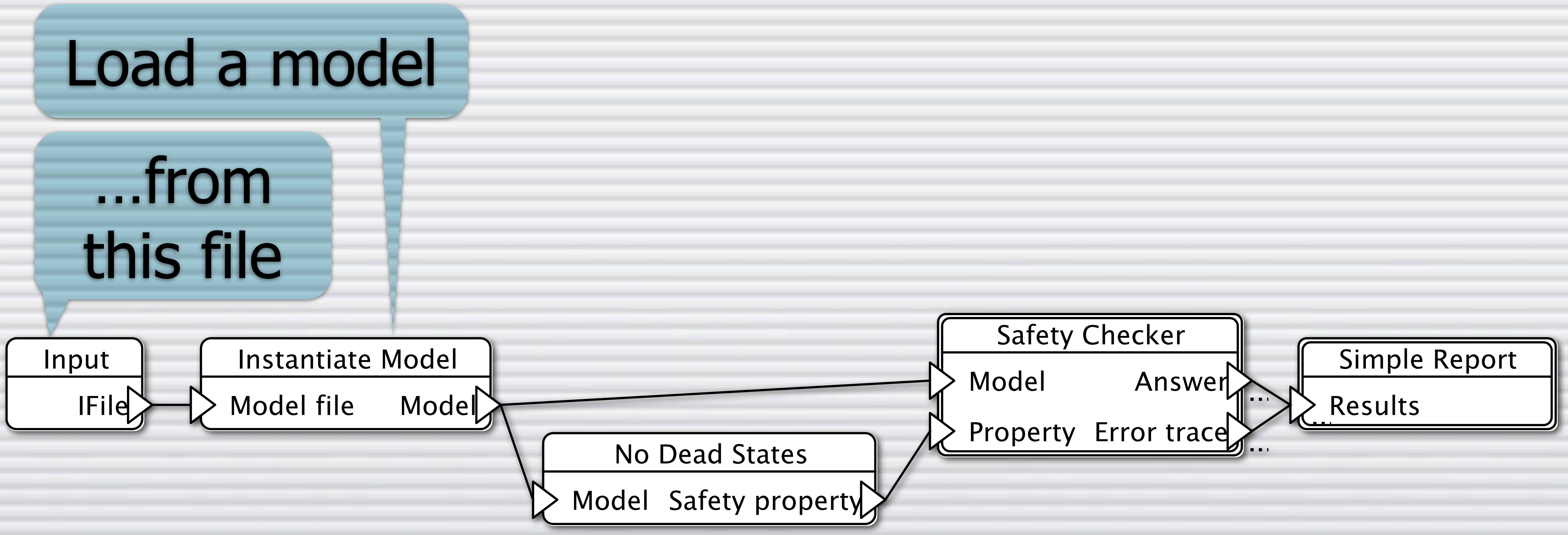
Deadlock Checker



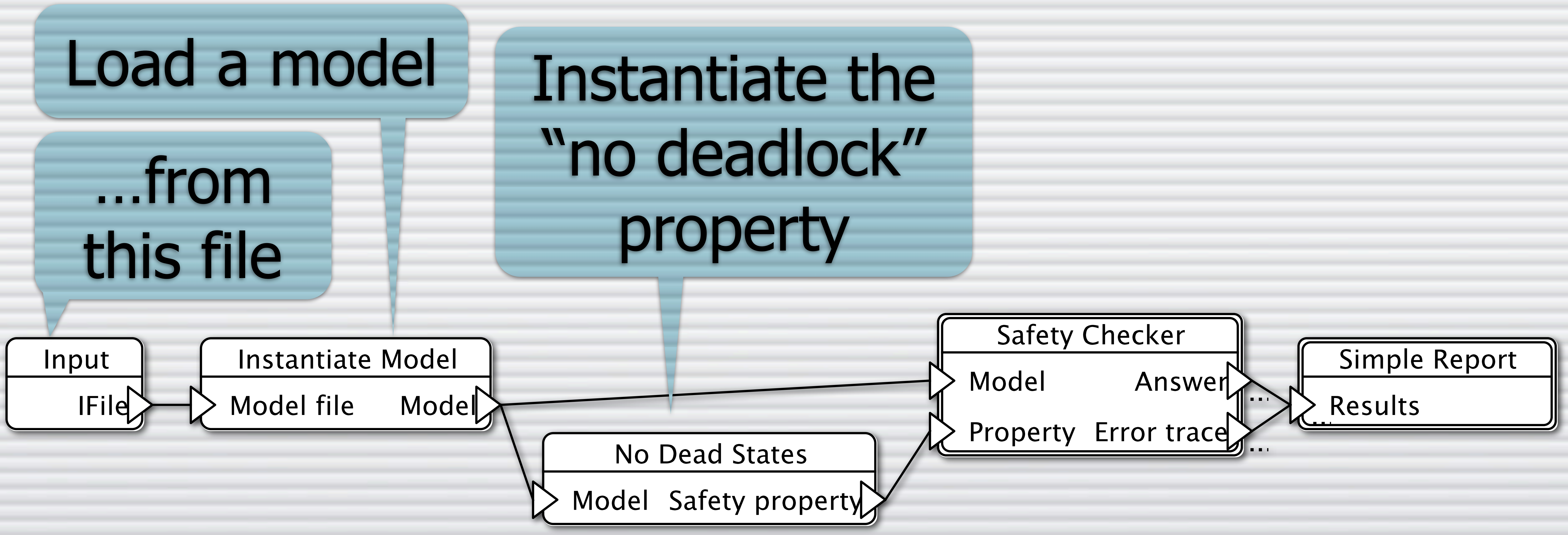
Deadlock Checker



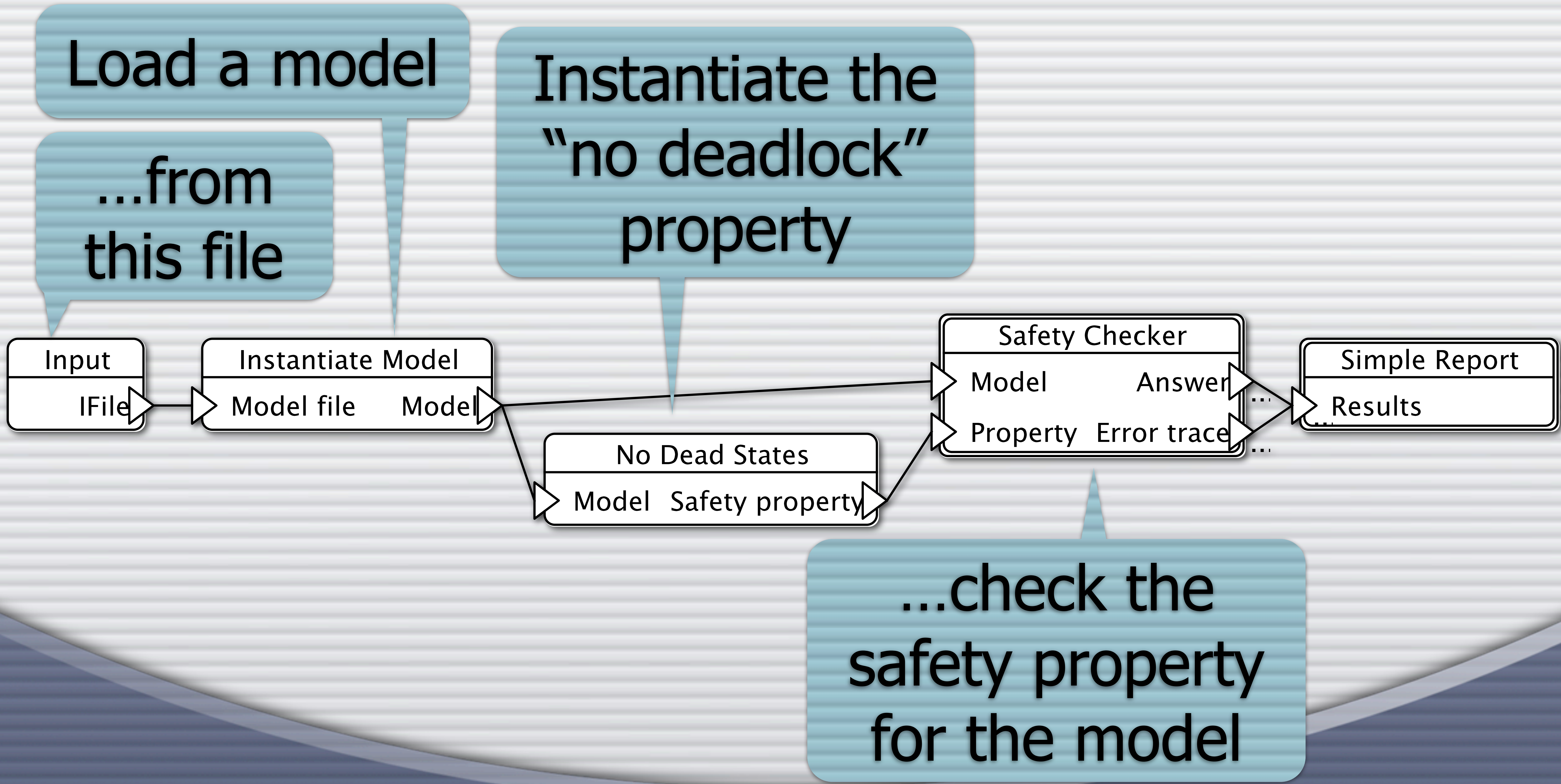
Deadlock Checker



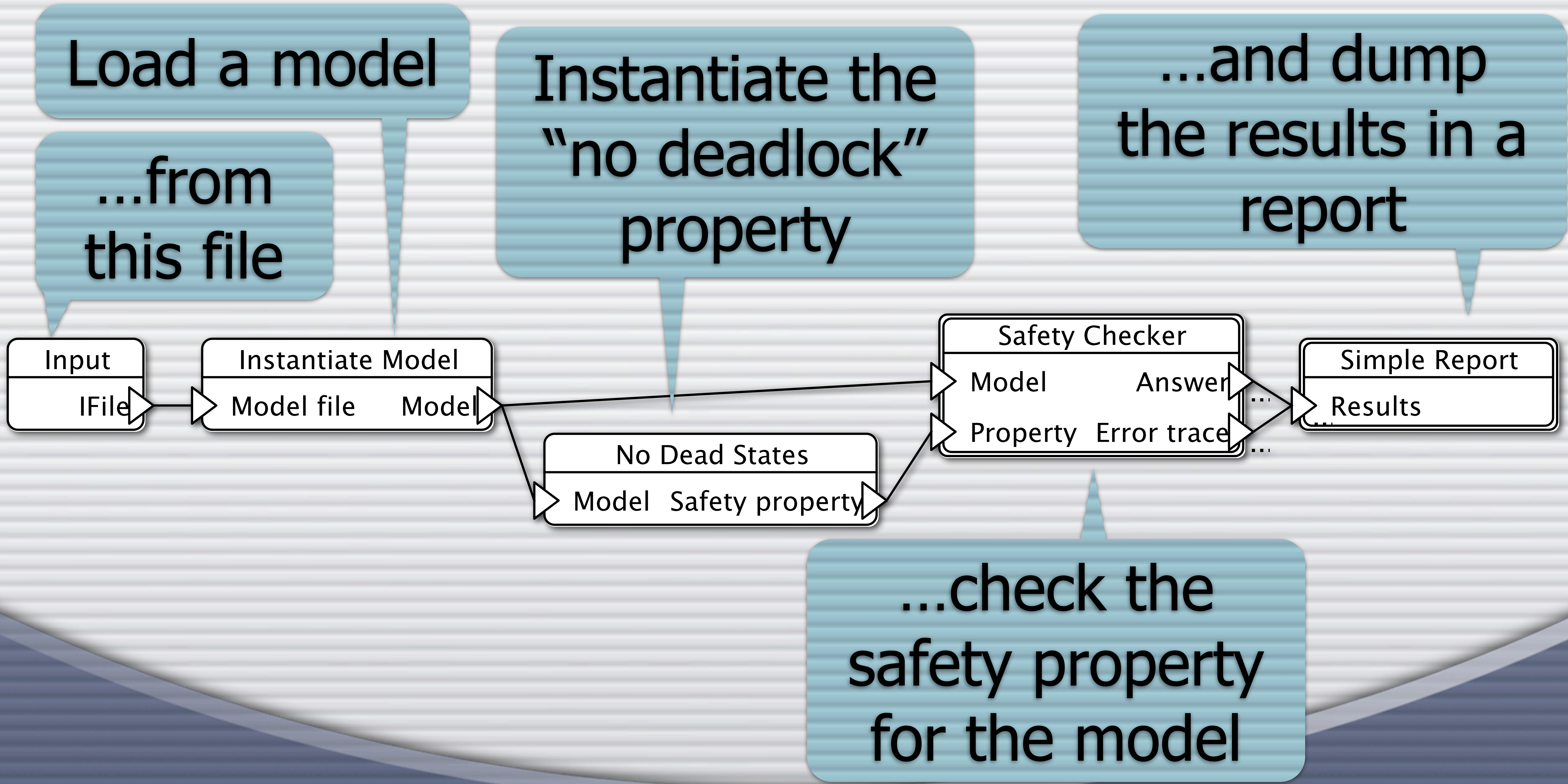
Deadlock Checker



Deadlock Checker



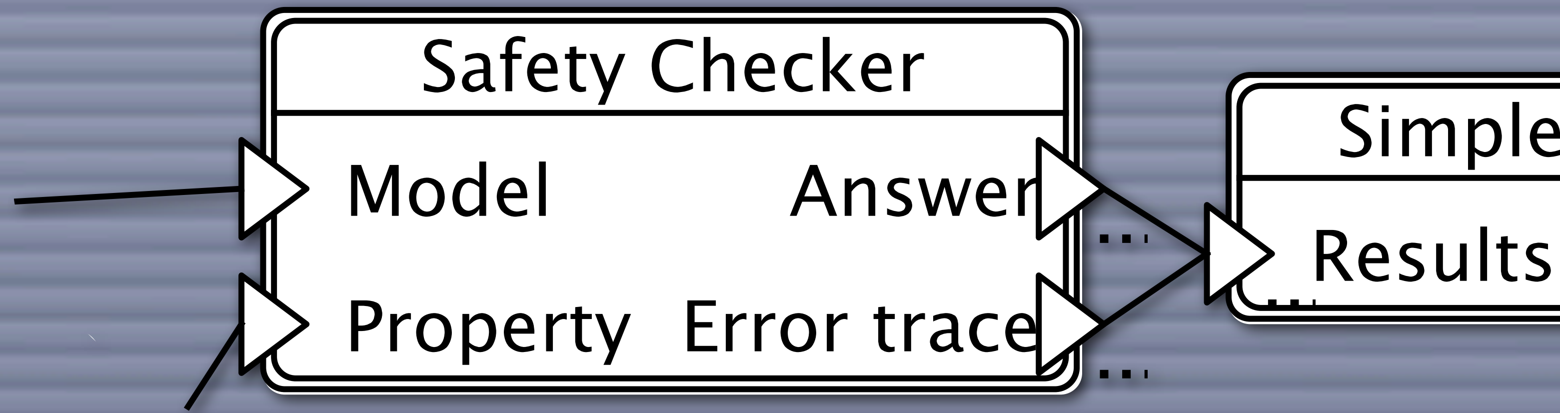
Deadlock Checker



Deadlock Checker

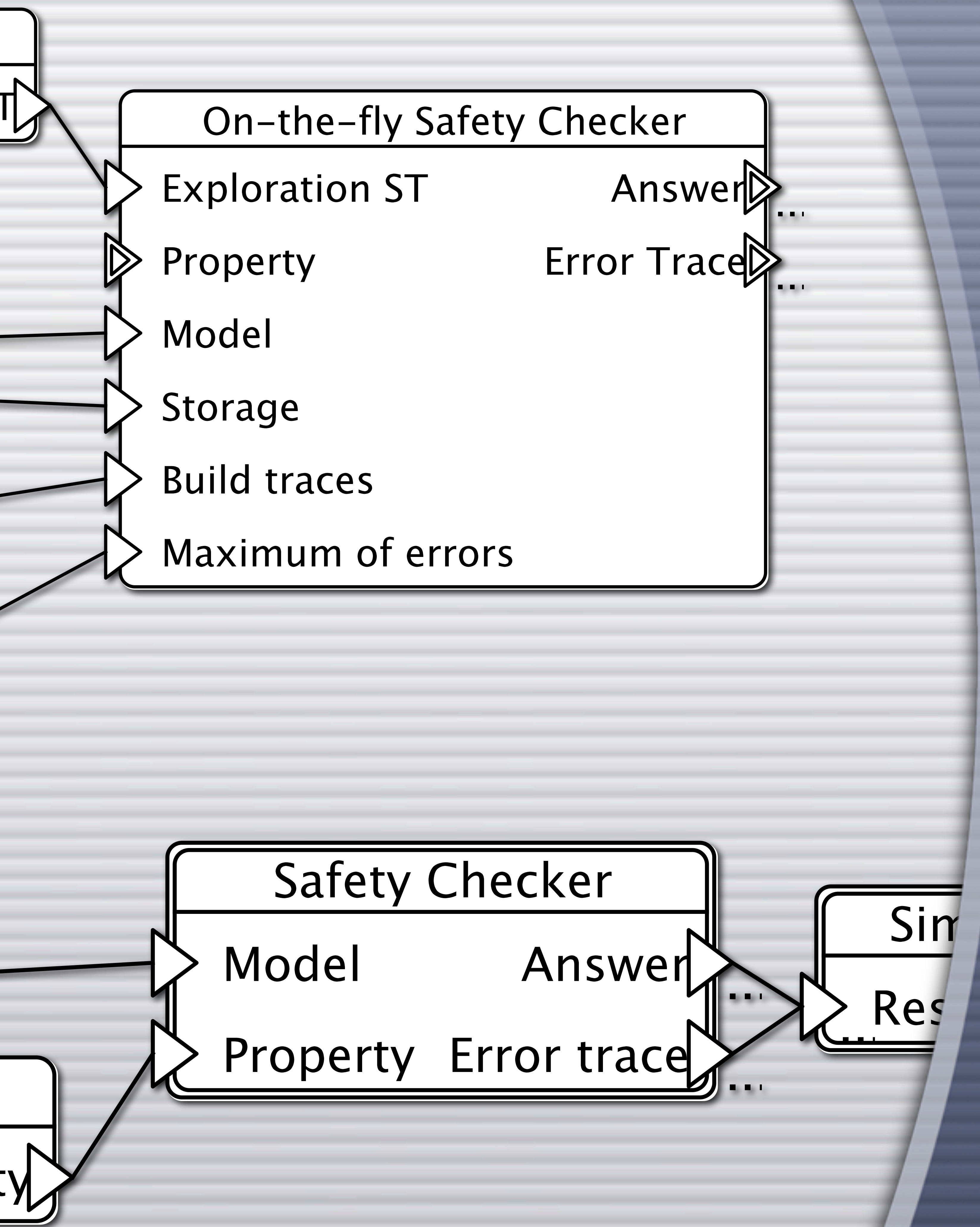
Abstraction

- The "Safety Checker" is not a single unit of work
- It is in fact a **macro** representing multiple tasks, such as instantiating a hash table and performing a BFS

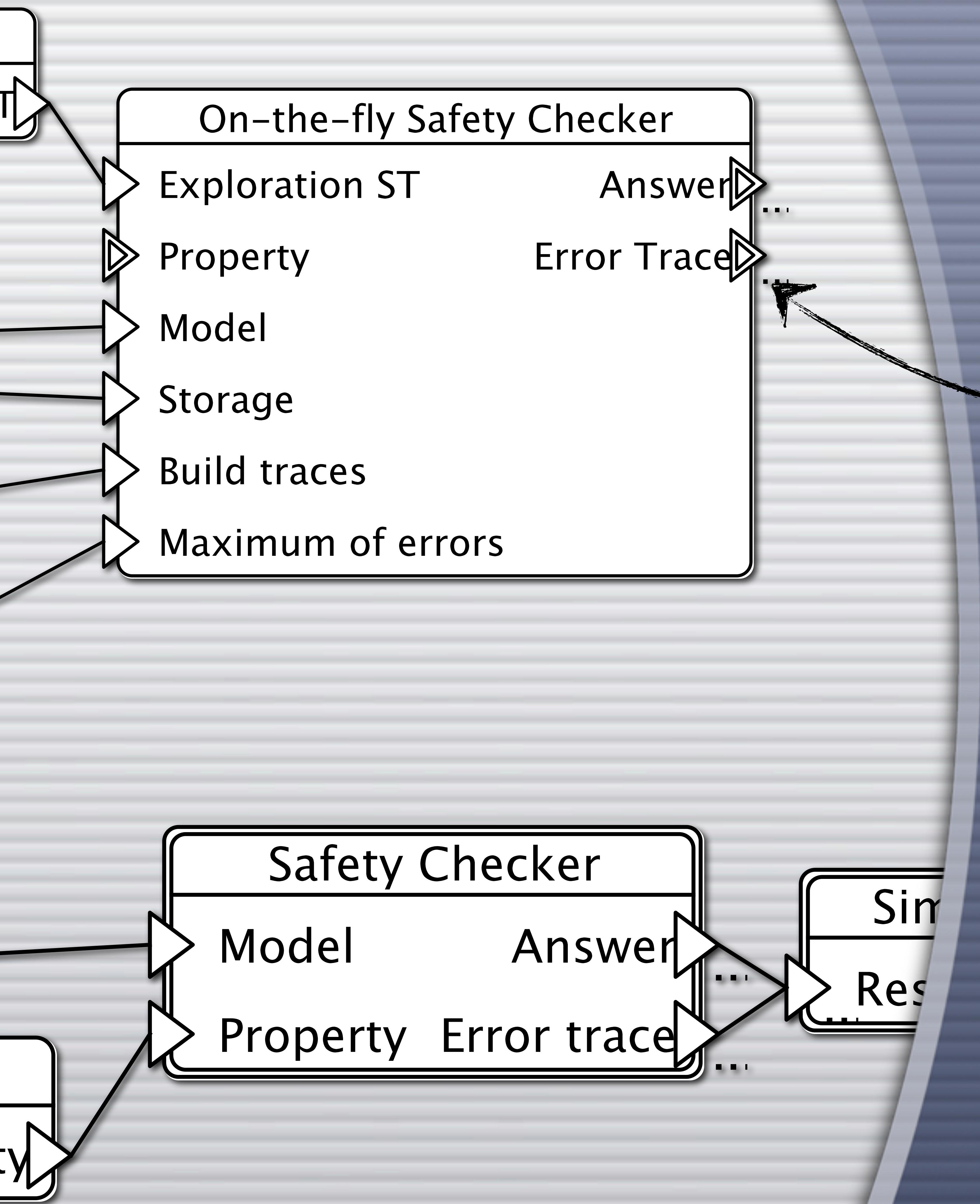


Abstraction

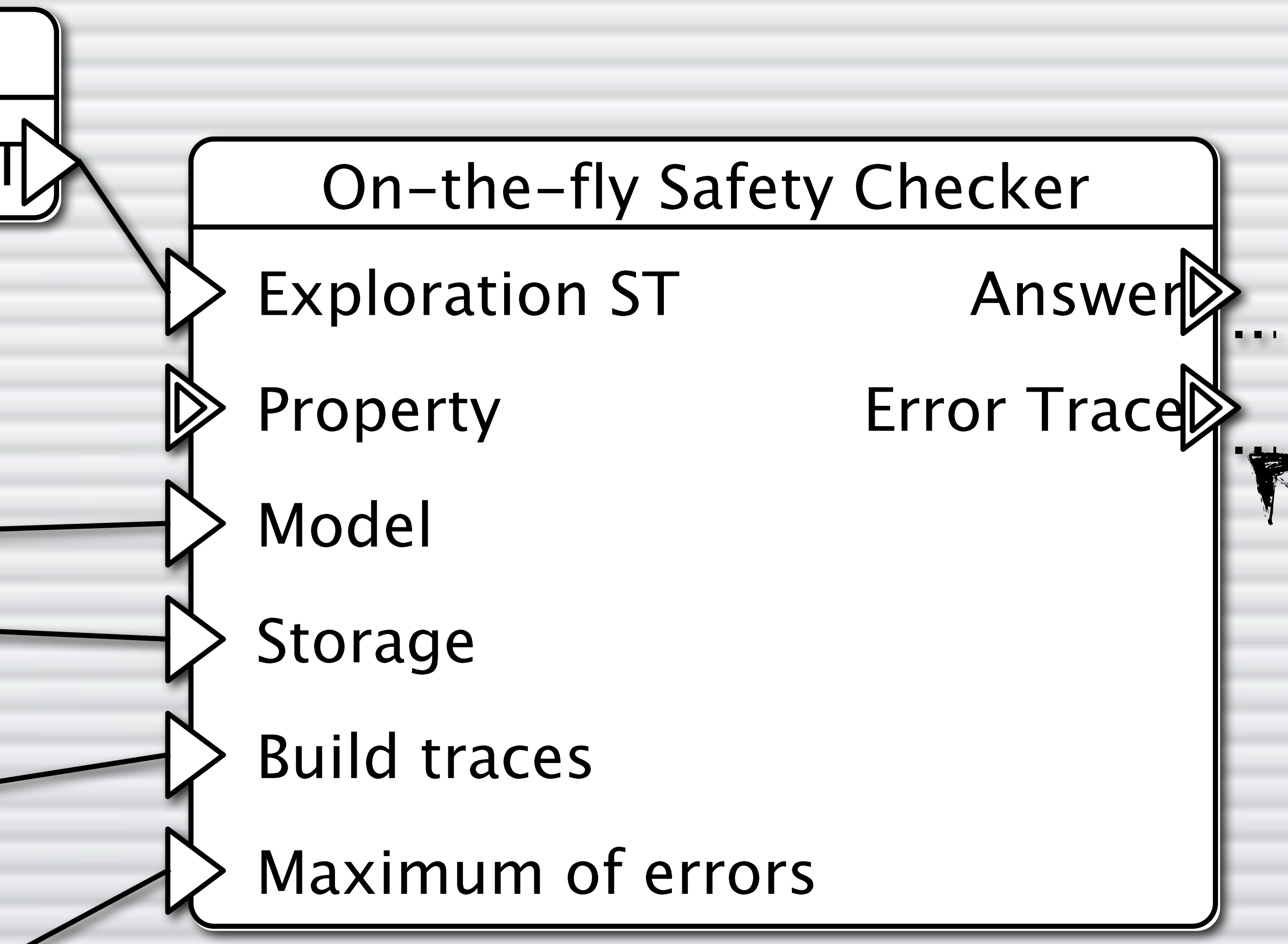
- The “Safety Checker” is not a single unit of work
- It is in fact a **macro** representing multiple tasks, such as instantiating a hash table and performing a BFS



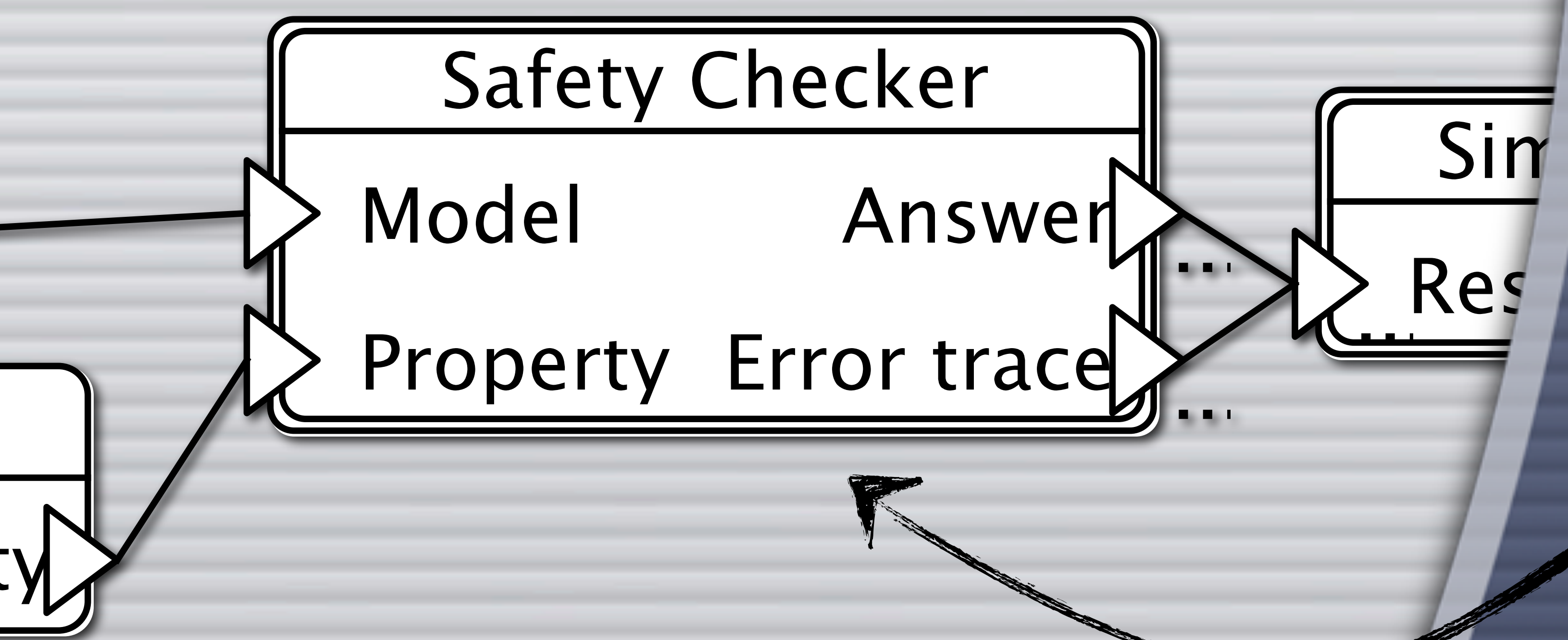
- Jobs can have **exported ports**
- Jobs can be represented by **macro tasks (macros)**



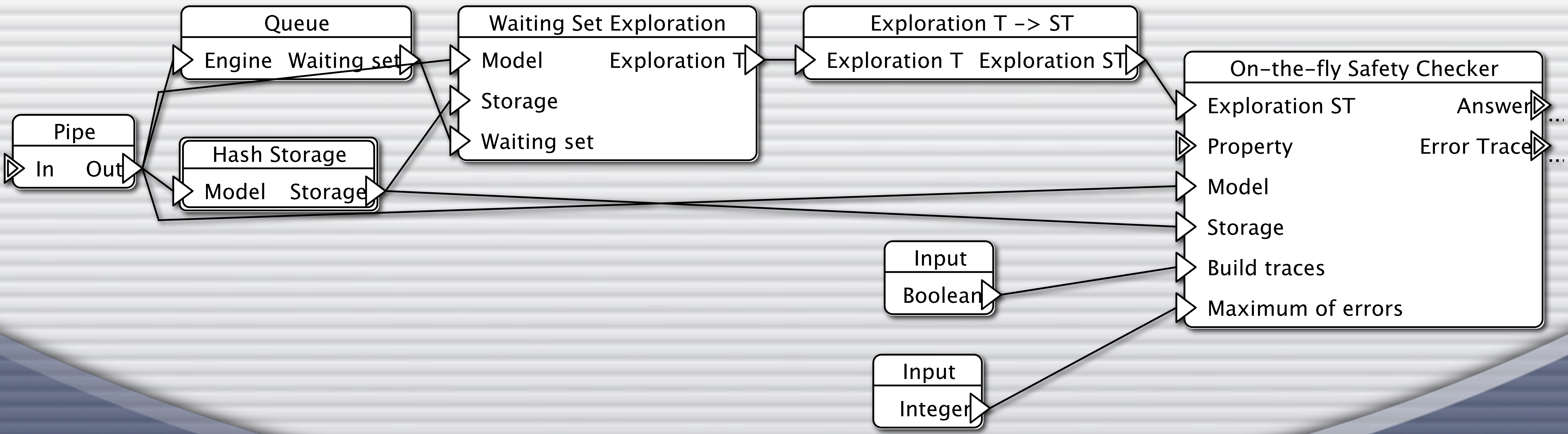
- Jobs can have **exported ports**
- Jobs can be represented by **macro tasks (macros)**



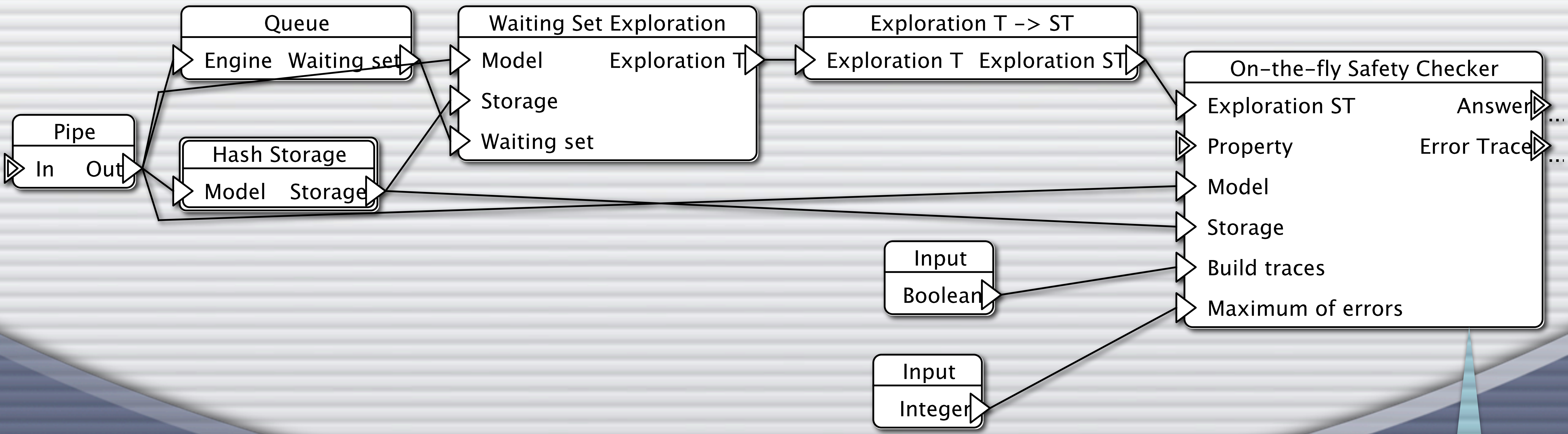
- Jobs can have **exported ports**



- Jobs can be represented by **macro tasks (macros)**

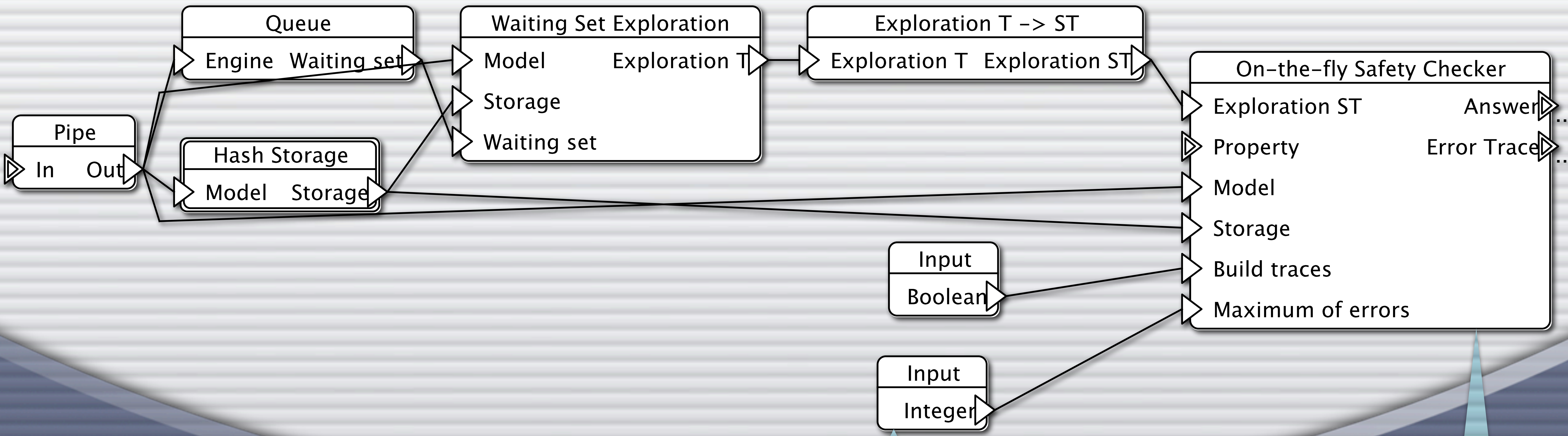


Safety Checker



Safety Check

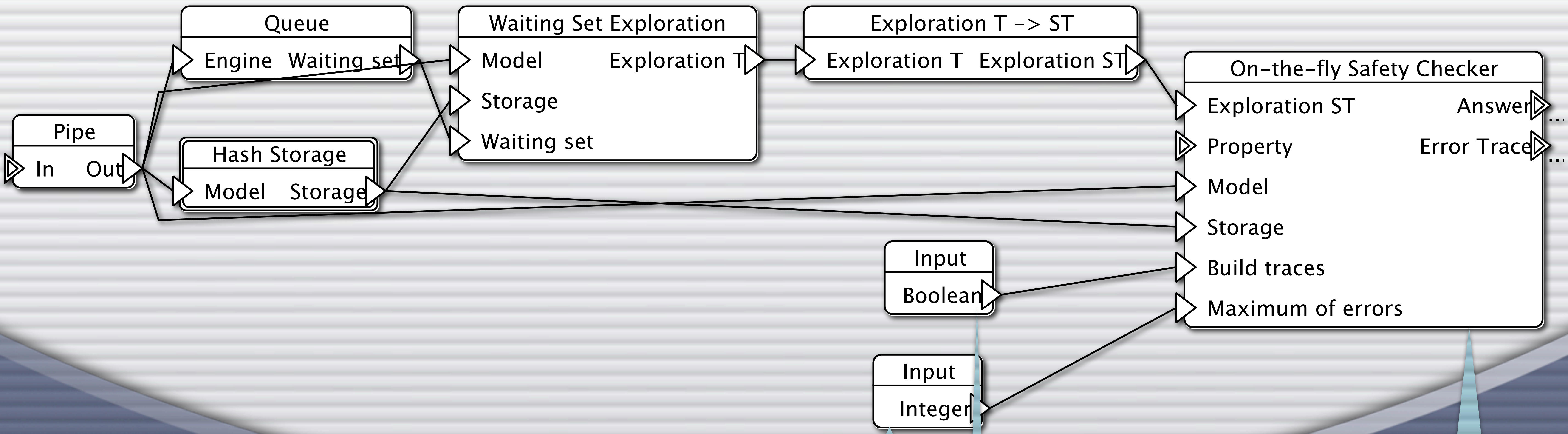
Check the given property using the given exploration



Stop after finding at most 10 errors

Check the given property using the given exploration

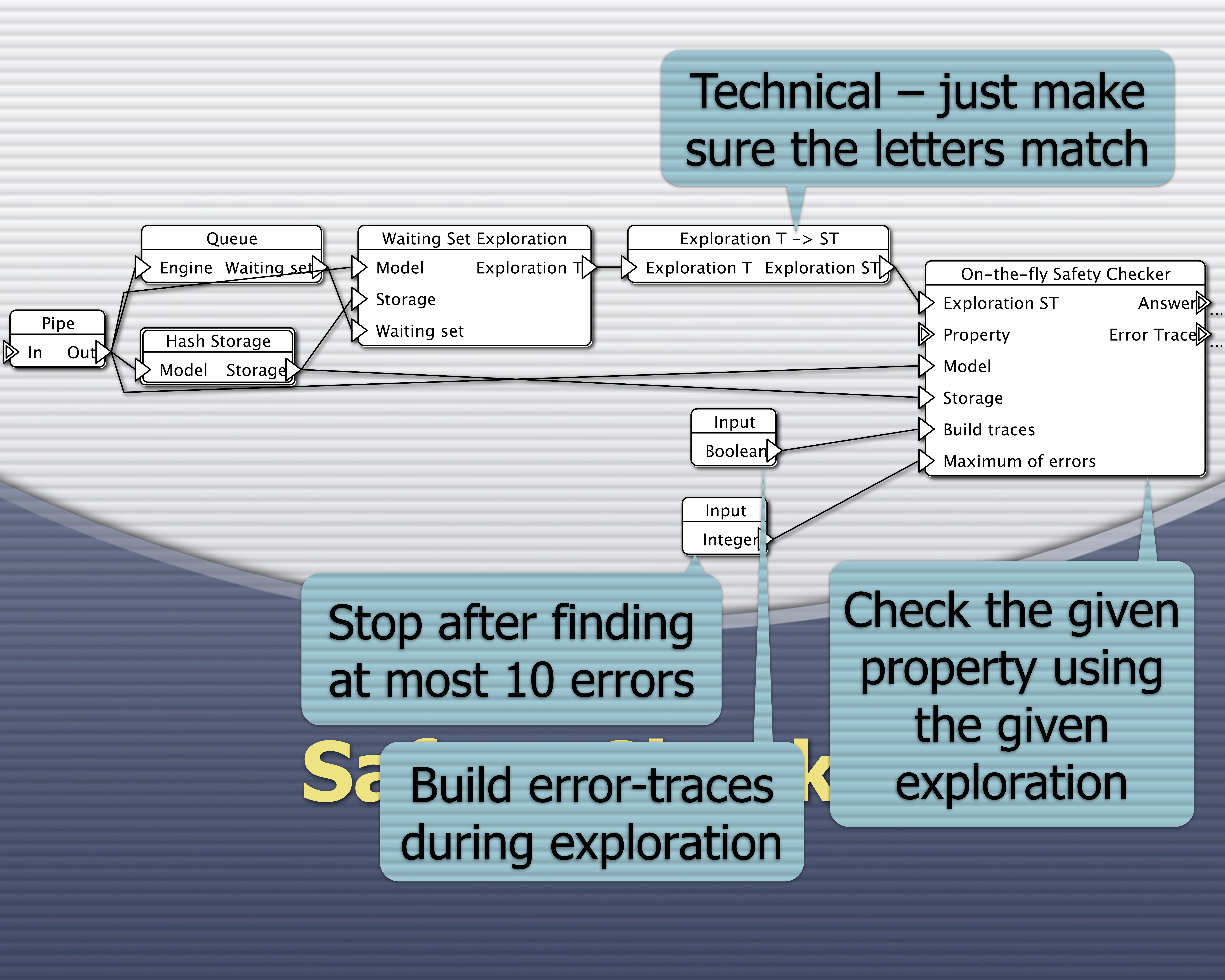
Safety Check



Stop after finding at most 10 errors

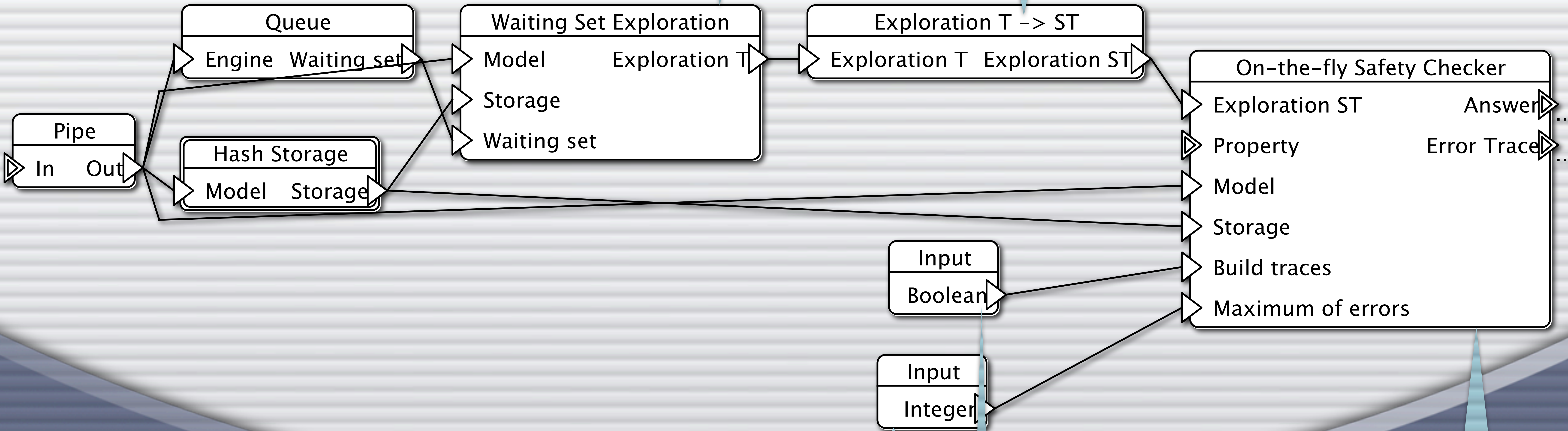
Build error-traces during exploration

Check the given property using the given exploration


Technical – just make sure the letters match
Queue
Engine
Waiting set
Waiting Set Exploration
Model
Exploration T
Storage
Waiting set
Exploration T -> ST
Exploration T
Exploration ST
On-the-fly Safety Checker
Exploration ST
Answer
Property
Error Trace
Model
Storage
Build traces
Maximum of errors
Pipe
In
Out
Hash Storage
Model
Storage
Input
Boolean
Input
Integer
Stop after finding at most 10 errors
Build error-traces during exploration
Check the given property using the given exploration

Exploration algorithm

Technical – just make sure the letters match



Stop after finding at most 10 errors

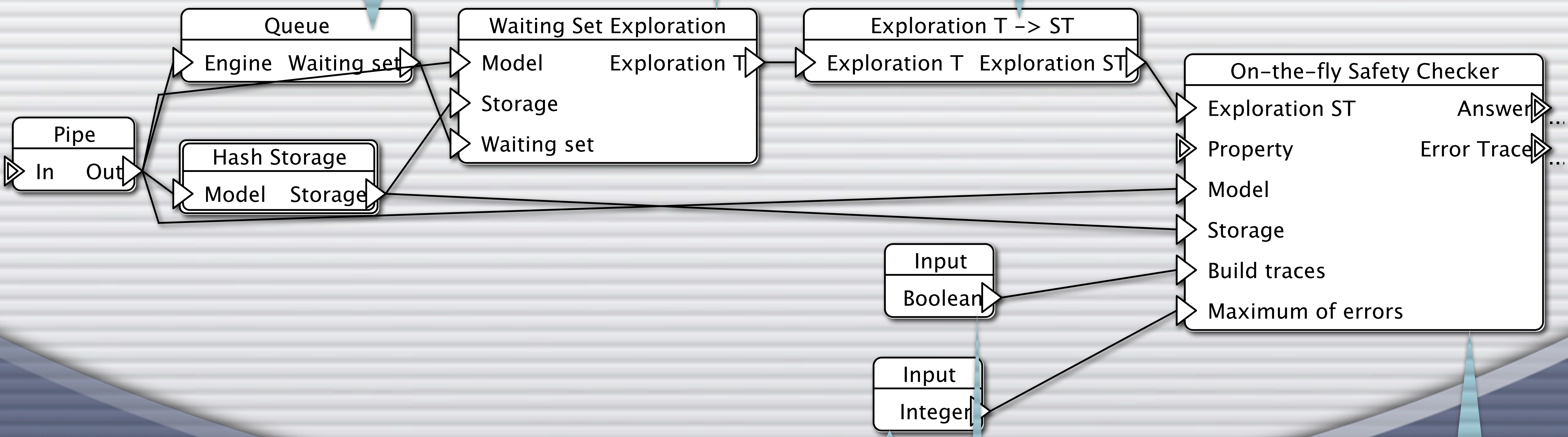
Build error-traces during exploration

Check the given property using the given exploration

Temporary storage is a queue

Exploration algorithm

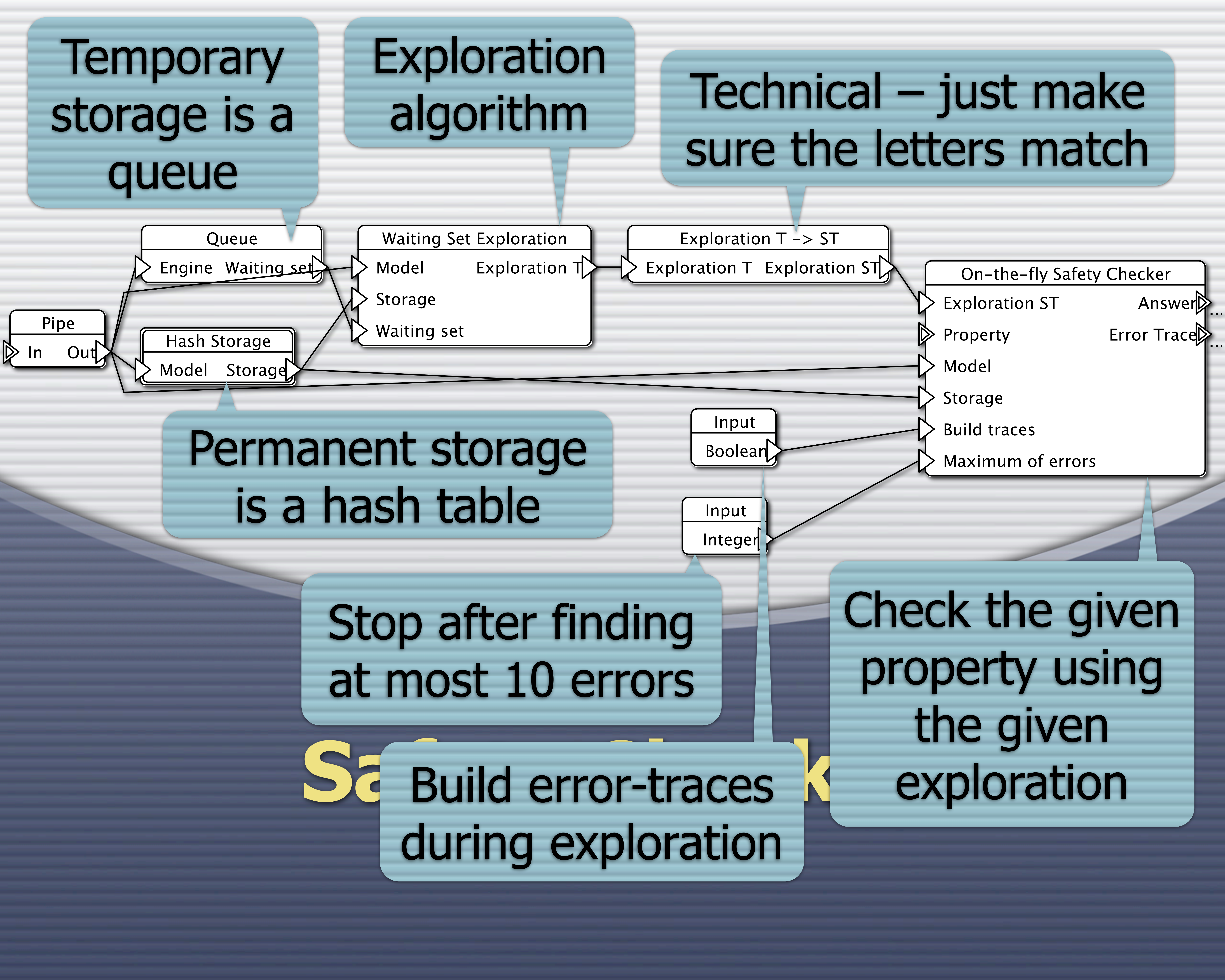
Technical – just make sure the letters match



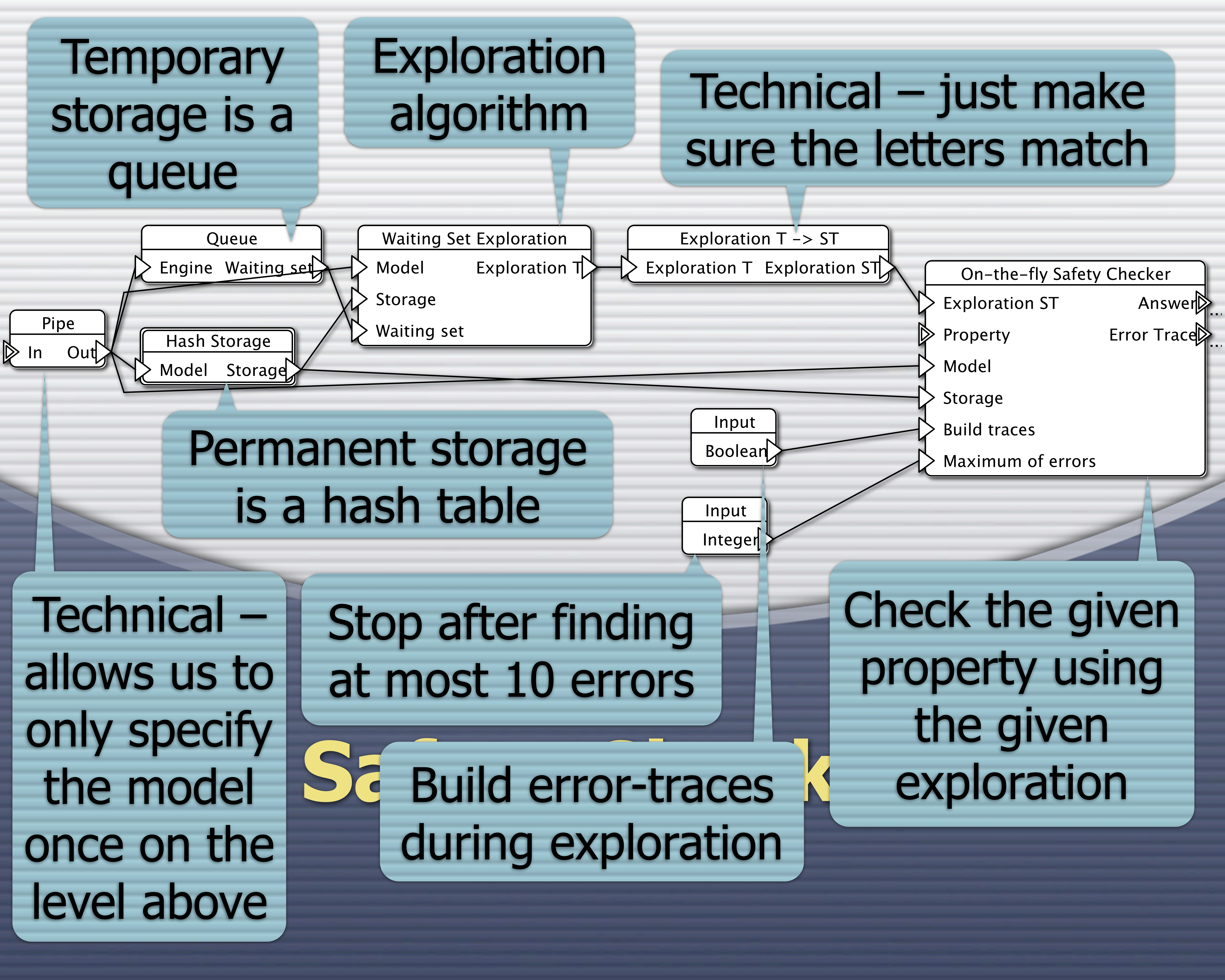
Stop after finding at most 10 errors

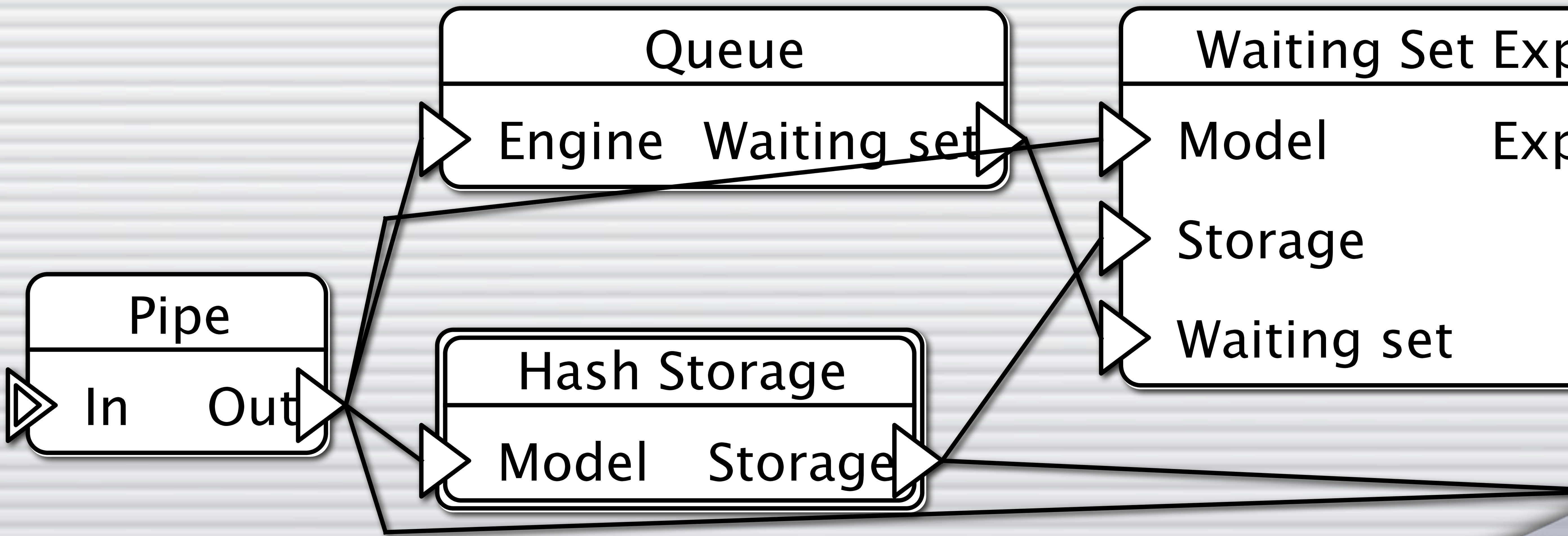
Build error-traces during exploration

Check the given property using the given exploration

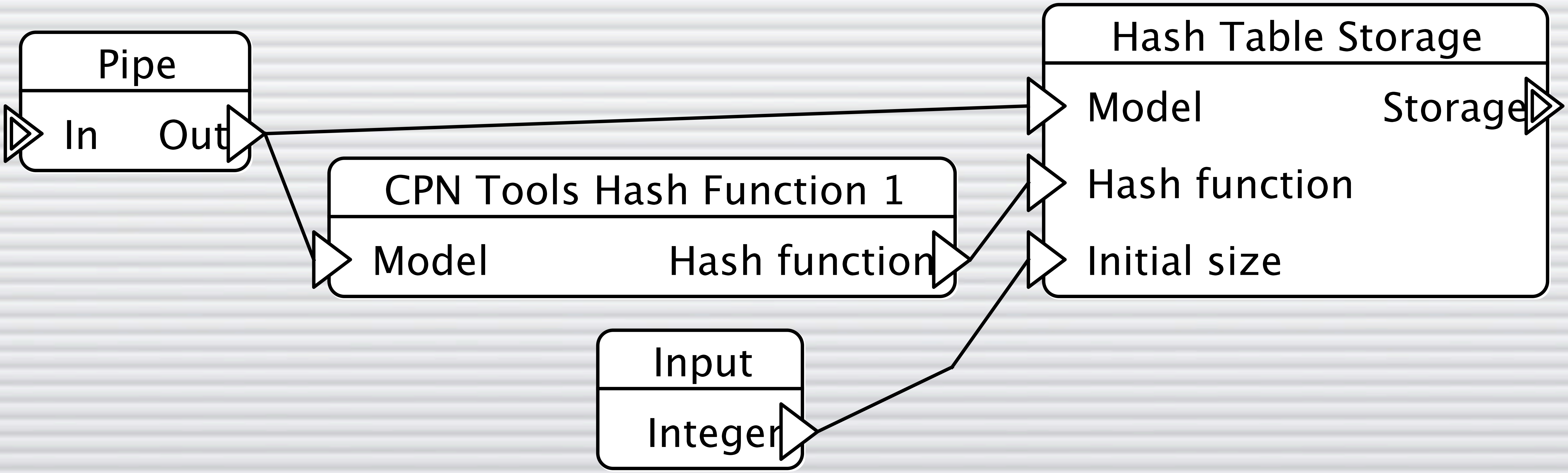


Temporary storage is a queue
Exploration algorithm
Technical – just make sure the letters match
Queue
Engine
Waiting set
Waiting Set Exploration
Model
Storage
Waiting set
Exploration T
Exploration T -> ST
Exploration T
Exploration ST
On-the-fly Safety Checker
Exploration ST
Property
Model
Storage
Build traces
Maximum of errors
Answer
Error Trace
Pipe
In
Out
Hash Storage
Model
Storage
Input
Boolean
Input
Integer
Permanent storage is a hash table
Stop after finding at most 10 errors
Build error-traces during exploration
Check the given property using the given exploration

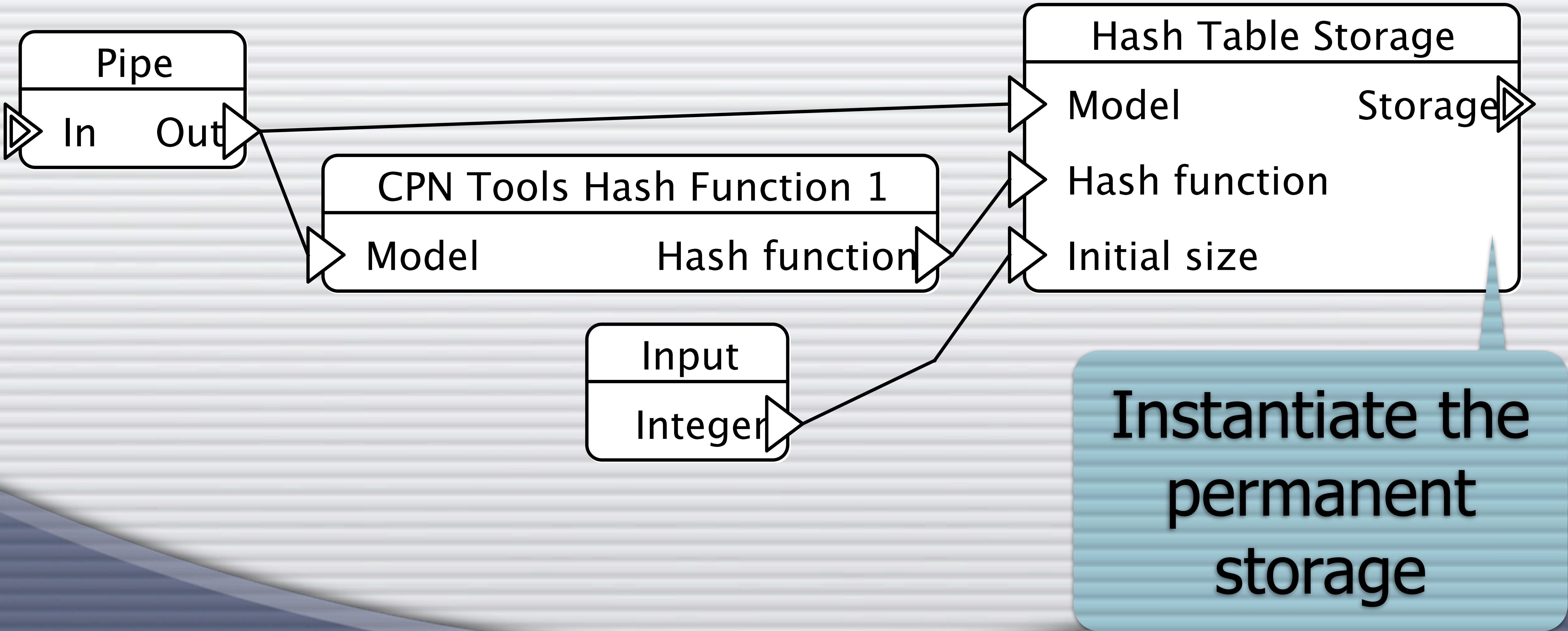

Temporary storage is a queue
Exploration algorithm
Technical – just make sure the letters match
Queue
Engine
Waiting set
Waiting Set Exploration
Model
Exploration T
Storage
Waiting set
Exploration T -> ST
Exploration T
Exploration ST
On-the-fly Safety Checker
Exploration ST
Answer
Property
Error Trace
Model
Storage
Build traces
Maximum of errors
Pipe
In
Out
Hash Storage
Model
Storage
Input
Boolean
Input
Integer
Permanent storage is a hash table
Technical – allows us to only specify the model once on the level above
Stop after finding at most 10 errors
Build error-traces during exploration
Check the given property using the given exploration



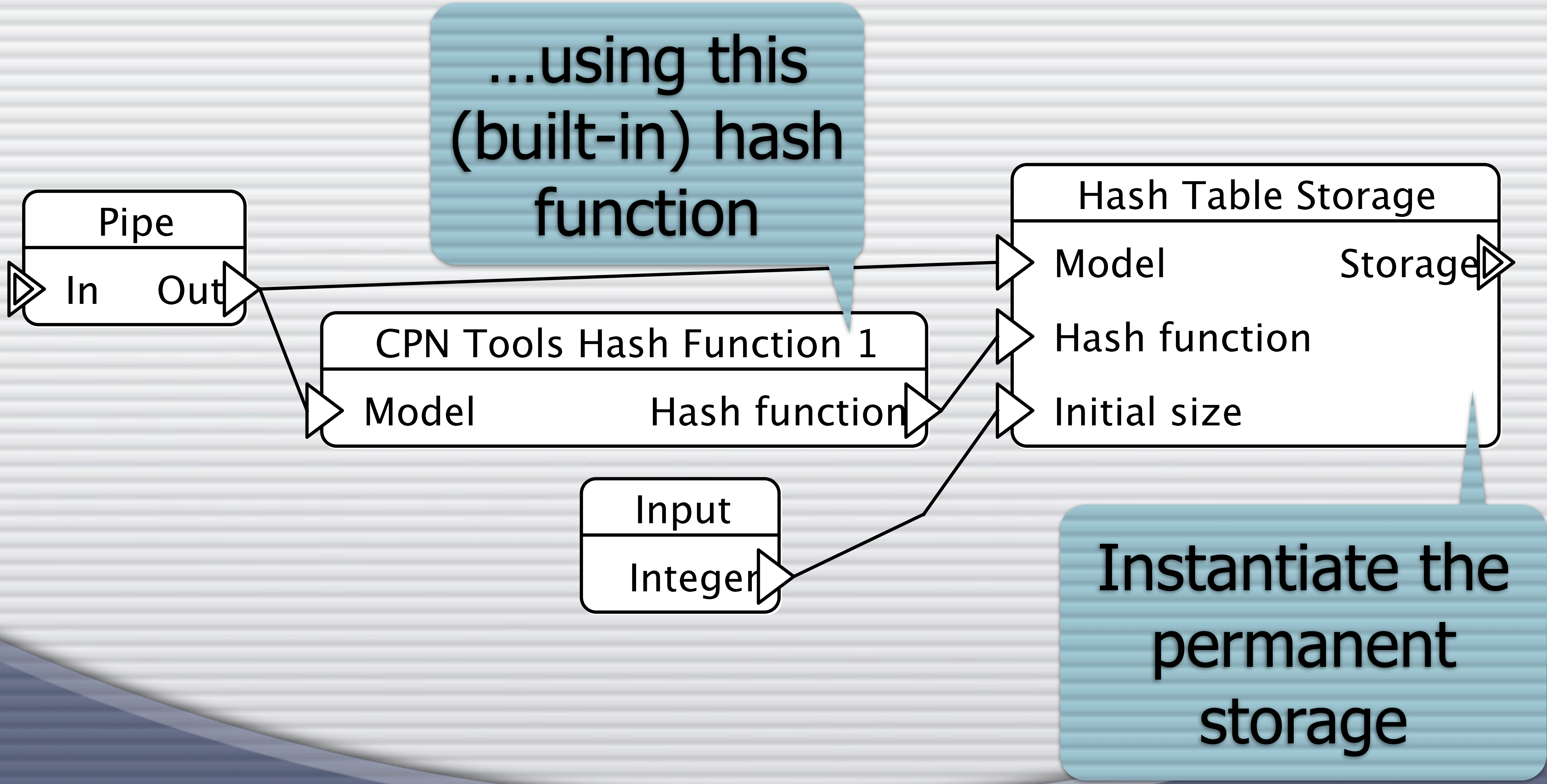
Safety Checker



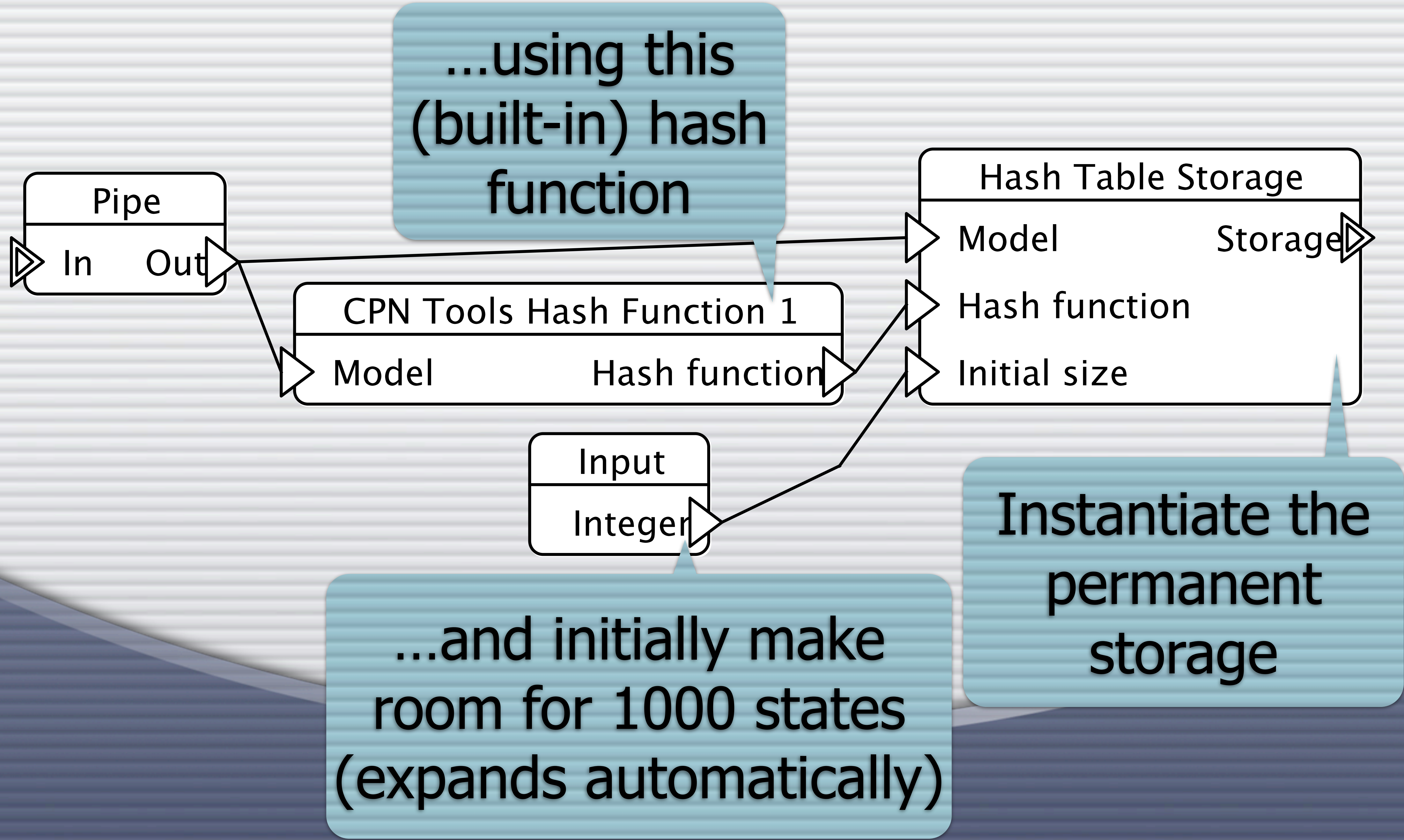
Hash Storage



Hash Storage

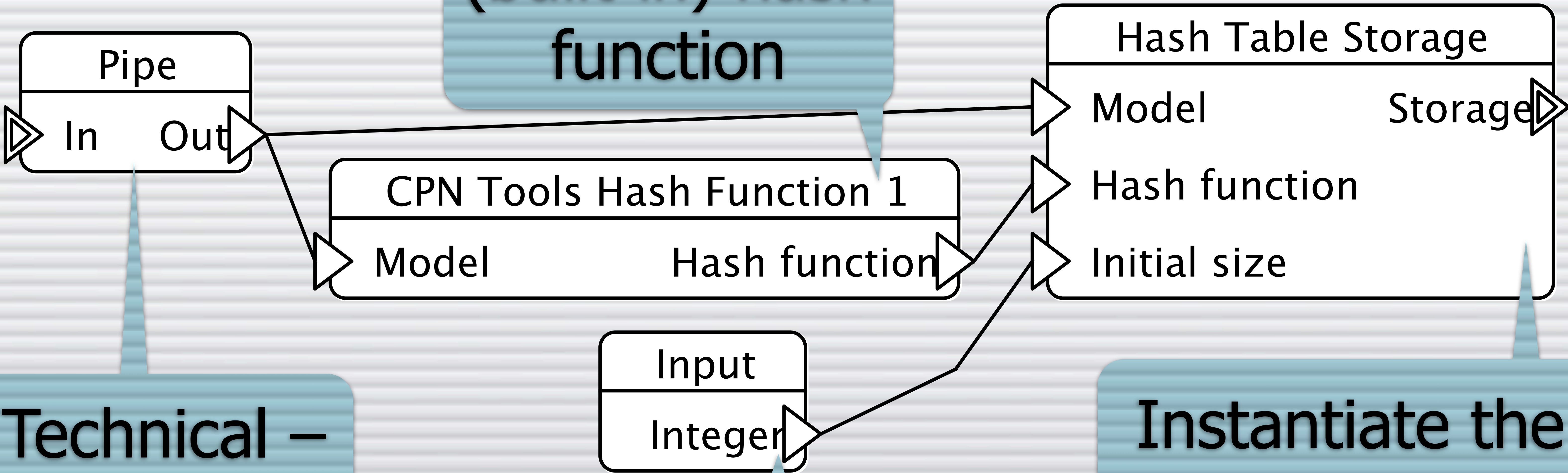


Hash Storage



Hash Storage

Hash Storage



...using this (built-in) hash function

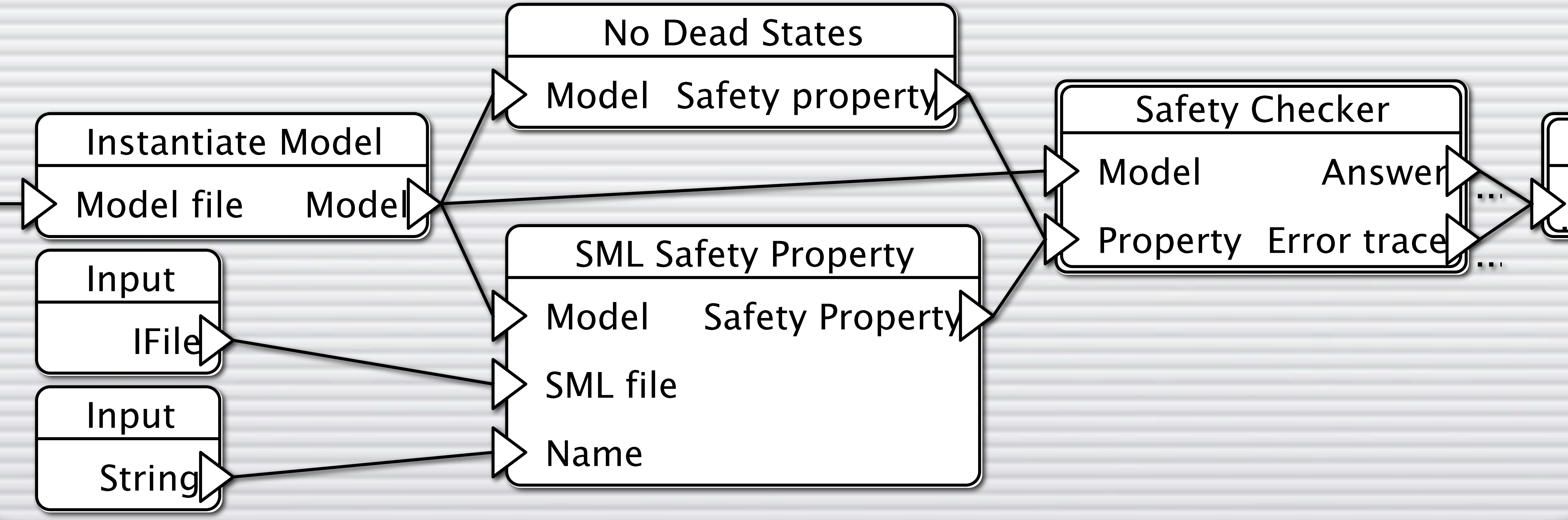
Technical – allows us to only specify the model once on the level above

...and initially make room for 1000 states (expands automatically)

Instantiate the permanent storage

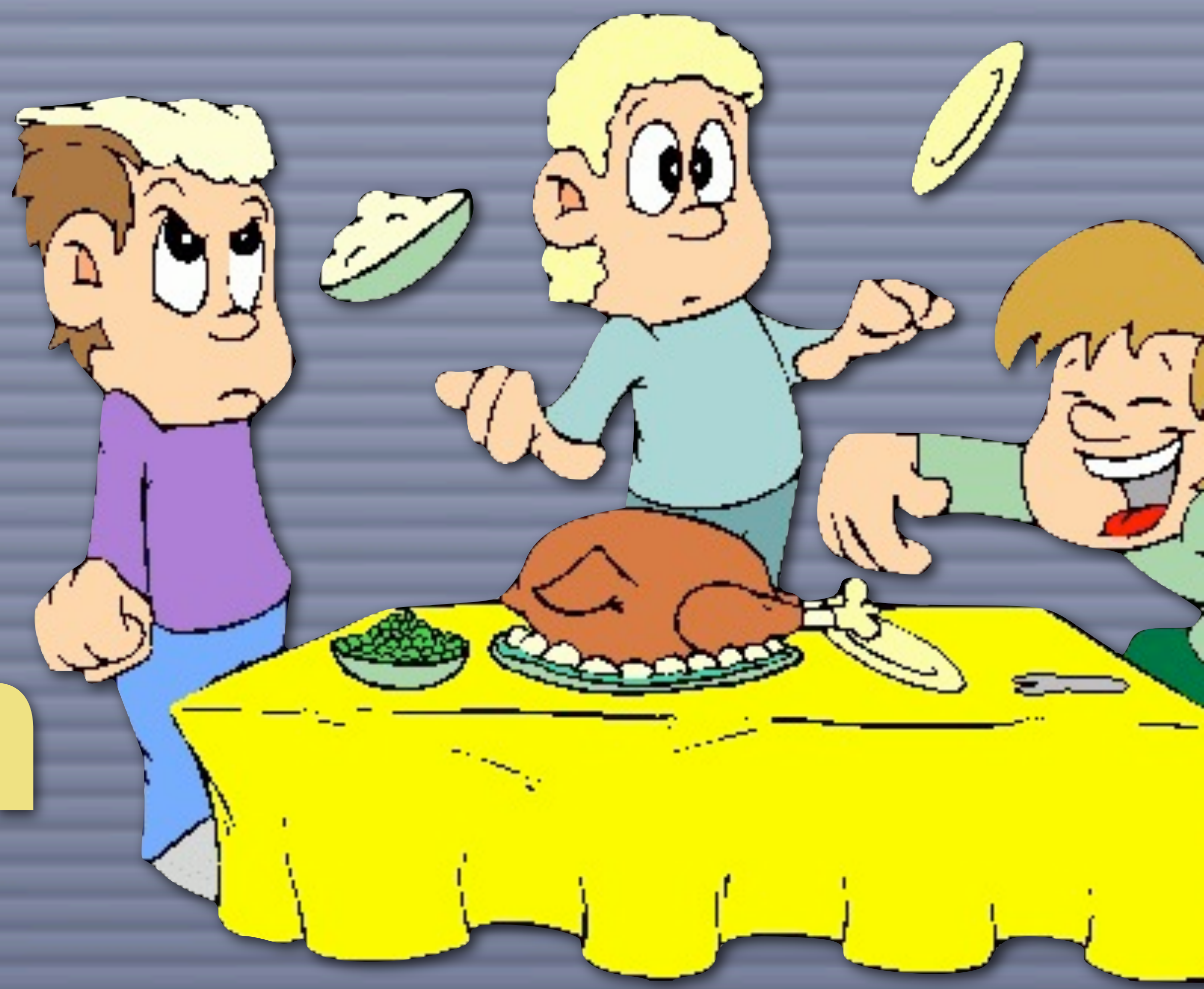
Safety Properties

- Sometimes we may want to check properties other than absence of deadlocks
- Custom properties are created using SML
- ASAP automatically generates a template formula tailored to a specific model



Example: Mutual Exclusion

Example: Mutual Exclusion



- We want to check that two adjacent philosophers cannot be eating at the same time
  - I.e., that they are not allowed access to a shared resource (chop-stick) at the same time
- This is equivalent to checking that if philosopher p is eating, then philosopher p+1 is not (mod n)

A Bit of SML

- Check if there is an element “p′ ” in “lst” that satisfies the predicate “f(p′)”:
```
List.exists (fn p’ => f(p’)) lst
```
- Check if “2 + 1 mod 7” belongs to a list, “lst”:
```
List.exists (fn p' => p’ = (2 + 1) mod 7) lst
```
- Check if “p + 1 mod n” belongs to a list, “lst”:
```
List.exists (fn p' => p’ = (p + 1) mod n) lst
```
- Check if there is an element “p” in “lst” such that “p + 1 mod n” belongs to “lst”:
```
List.exists (fn p => List.exists
  (fn p' => p' = (p + 1) mod n) lst) lst
```

Yes, this is inefficient; we can sort “lst” and only compare neighbors

Example: Mutual Exclusion

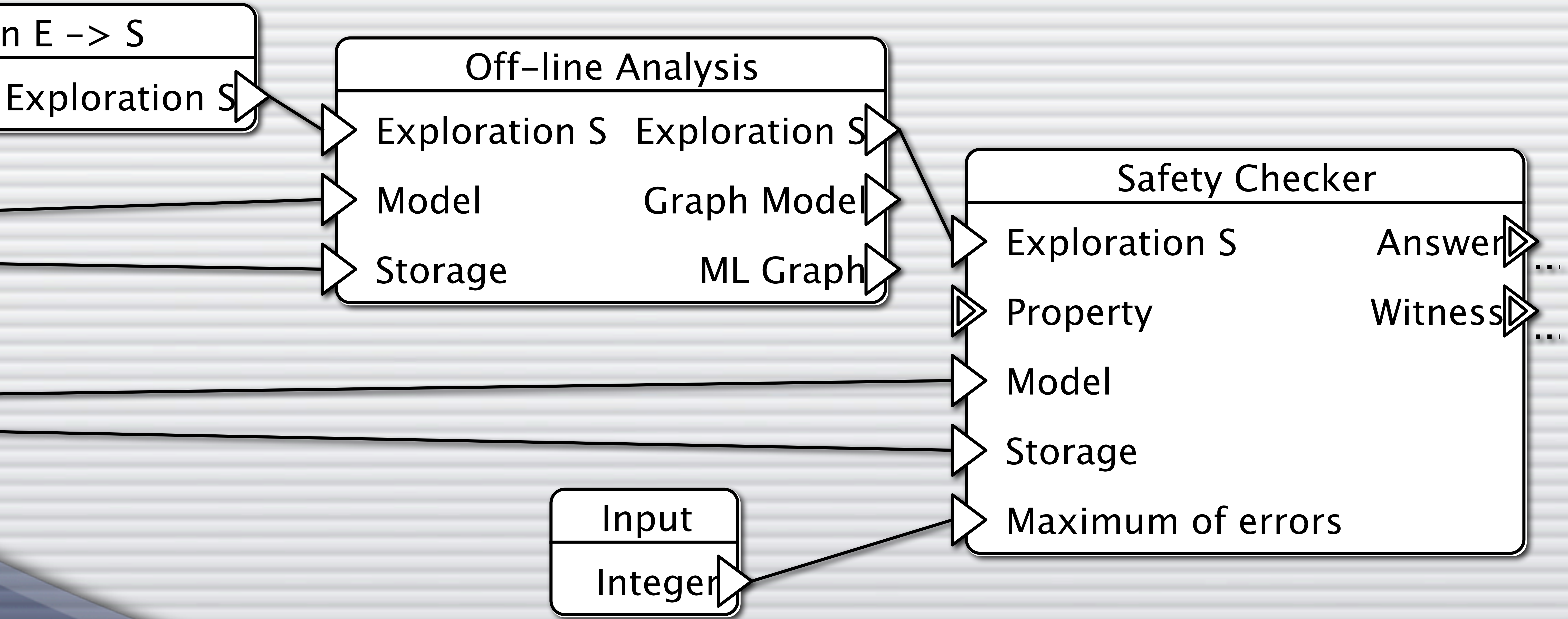
```
fun query (state, events) =
let
  fun query'New_Page { Waiting, Has_One, Eating,
                       Philosophers, Initialized,
                       Chopsticks } = true
  fun query'state { New_Page} = query'New_Page New_Page
in
  query'state state
end
```

Example: Mutual Exclusion

```
fun query (state, events) =
let
  fun query'New_Page { Waiting, Has_One, Eating,
                       Philosophers, Initialized,
                       Chopsticks } =
     not (List.exists (fn p => List.exists
                  (fn p' => p' =
                    (p + 1) mod (List.hd Philosophers)
                  ) Eating) Eating)
  fun query'state { New_Page} = query'New_Page New_Page
in
  query'state state
```

Demo: Mutual Exclusion

- Create property
- Edit JoSEL job
- Run checker



Example:
On-line vs. Off-line

Off-line Safety Checker

V := { s_0 }
W := { s_0 }
while W ≠ ∅ **do**
  Select an s ∈ W
  W := W ∖ { s }
  for all t, s‵
    such that s $\rightarrow^t$ s′ **do**
   if s′ ∉ V **then**
    V := V ∪ { s′ }
    W := W ∪ { s′ }

for all v ∈ V **do**
  if ¬I(v) **then**
    return false
return true

This is off-line analysis; we first generate the state space and then we analyze it.

On-line Safety Checker

$V := \{ s_0 \}$
$W := \{ s_0 \}$
while $W \neq \emptyset$ **do**
  Select an $s \in W$
  $W := W \setminus \{ s \}$
  if $\neg I(s)$ **then**
    return false
  for all t, s' **such that** $s \to^{t} s'$ **do**
    if $s' \notin V$ **then**
      $V := V \cup \{ s' \}$
      $W := W \cup \{ s' \}$
return true

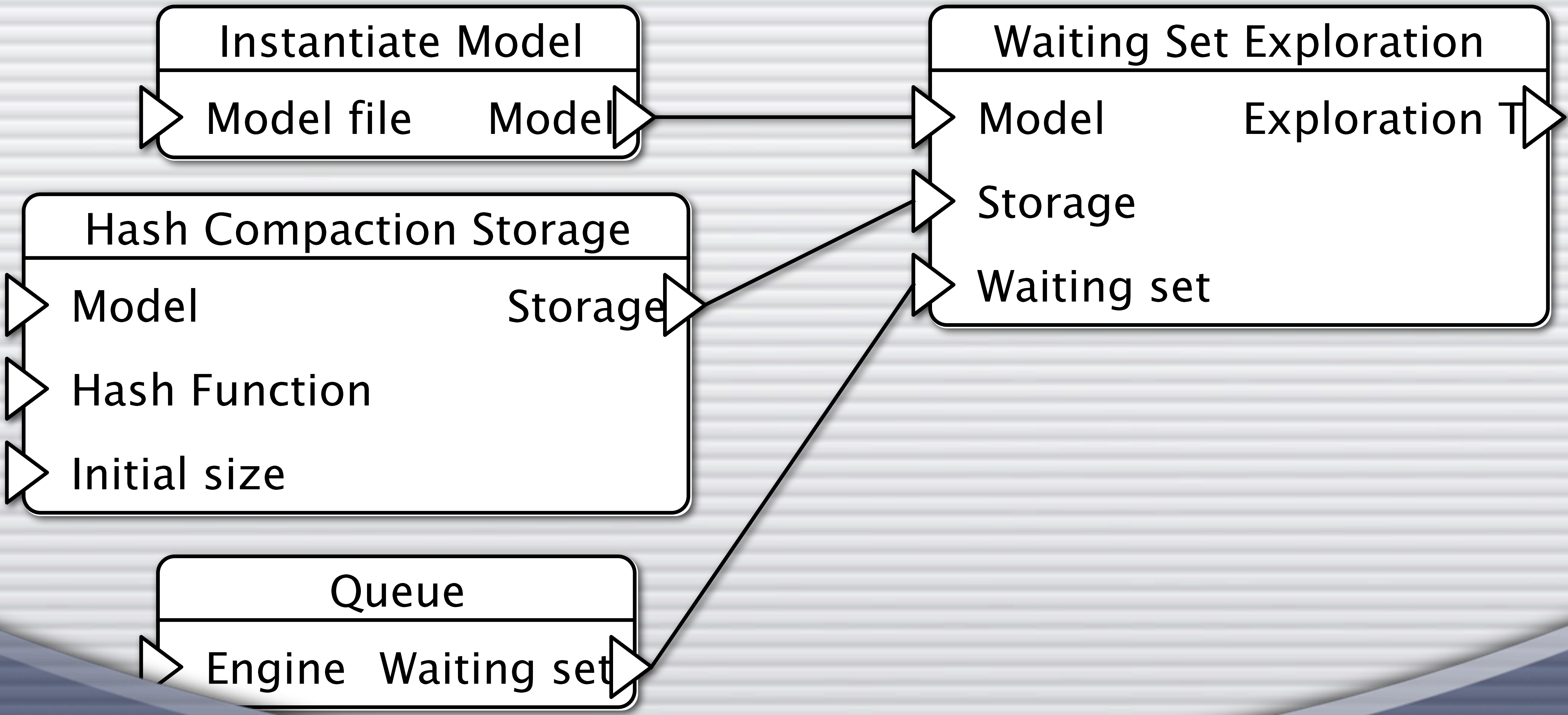
This is on-line analysis; we analyze the state space while we generate it.

On-line	Off-line
Finds errors faster Uses less memory Supported by ASAP	Can check additional properties subsequently Can (easier) provide error traces Can check more properties Supported by Design/CPN, CPN Tools, and ASAP

On-line vs. Off-line

Demo:
On-line vs. Off-line

- Show safety checker and time spent checking property (maybe crank up size)
- Change to off-line
- Note that top-level has not changed
- Show time spent checking property



Example: Hash-compaction

Hash-compaction

- A problem of the standard method is that we use 1000 bytes per state, and 4 GB / 1000 = $4 \bullet 10^6$ states
- What if we only use, say, 4 bytes per state; then we can store 4 GB / 4 = 10^9 states
- This is the rationale behind hash-compaction

Observation

- For a hash function h (any function, really) we have
  - $s = s' \Rightarrow h(s) = h(s')$
  - We use the terminology
    - s: **full state descriptor** (1000 bytes)
    - h(s): **compressed state descriptor** (4 bytes)
- We do not have that $h(s) = h(s') \Rightarrow s = s'$, but good hash functions ensure that this is mostly true
  - If $h(s) = h(s')$ but $s \neq s'$ we say we have a **hash collision**

Hash-compaction

$V := \{ s_0 \}$
$W := \{ s_0 \}$
while $W \neq \emptyset$ **do**
    Select an $s \in W$
    $W := W \setminus \{ s \}$
    if $\neg I(s)$ **then**
        return false
    for all t, s' **such that** $s \rightarrow^t s'$ **do**
        if $s' \notin V$ **then**
            $V := V \cup \{ s' \}$
            $W := W \cup \{ s' \}$
return true

We replace full state descriptors by compressed state descriptors in V

Hash-compaction

$V := \{\ h(s_0)\ \}$

$W := \{\ s_0\ \}$

while $W \neq \varnothing$ **do**

  Select an $s \in W$

  $W := W \setminus \{\ s\ \}$

  if $\neg I(s)$ **then**

    return false

  for all t, s' **such that** $s \to^{t} s'$ **do**

    if $h(s') \notin V$**en**

      $V := V \cup \{\ h(s')\ \}$

      $W := W \cup \{\ s'\ \}$

return true

We replace full state descriptors by compressed state descriptors in V

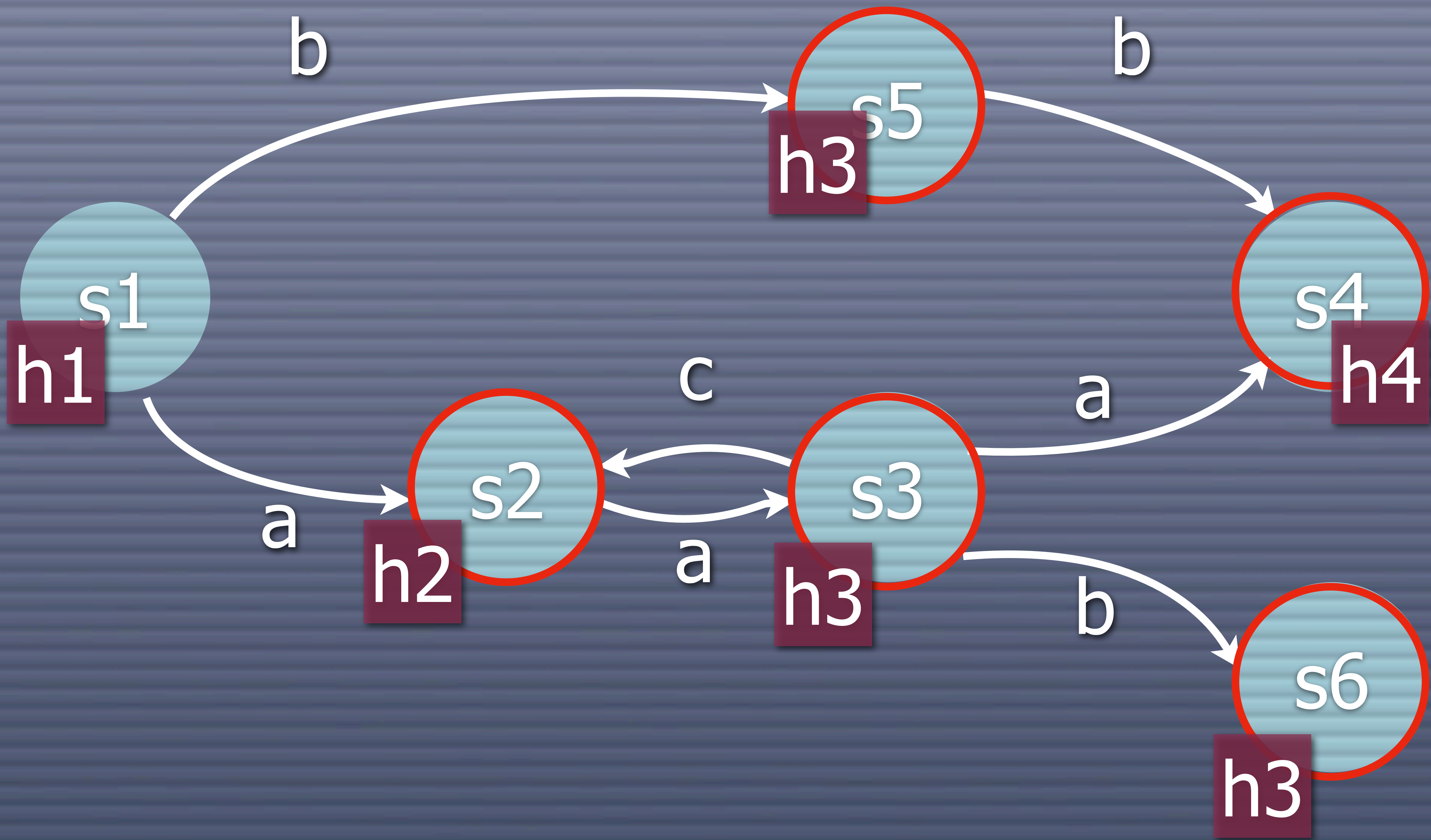
Hash-compaction

$V := \{ h(s_0) \}$
$W := \{ s_0 \}$
while $W \neq \emptyset$ **do**
  Select an $s \in W$
  $W := W \setminus \{ s \}$
  if $\neg I(s)$ **then**
    return false
  for all t, s' **such that** $s \rightarrow^{t} s'$ **do**
    if $h(s') \notin V$**en**
      $V := V \cup \{ h(s') \}$
      $W := W \cup \{ s' \}$
return true

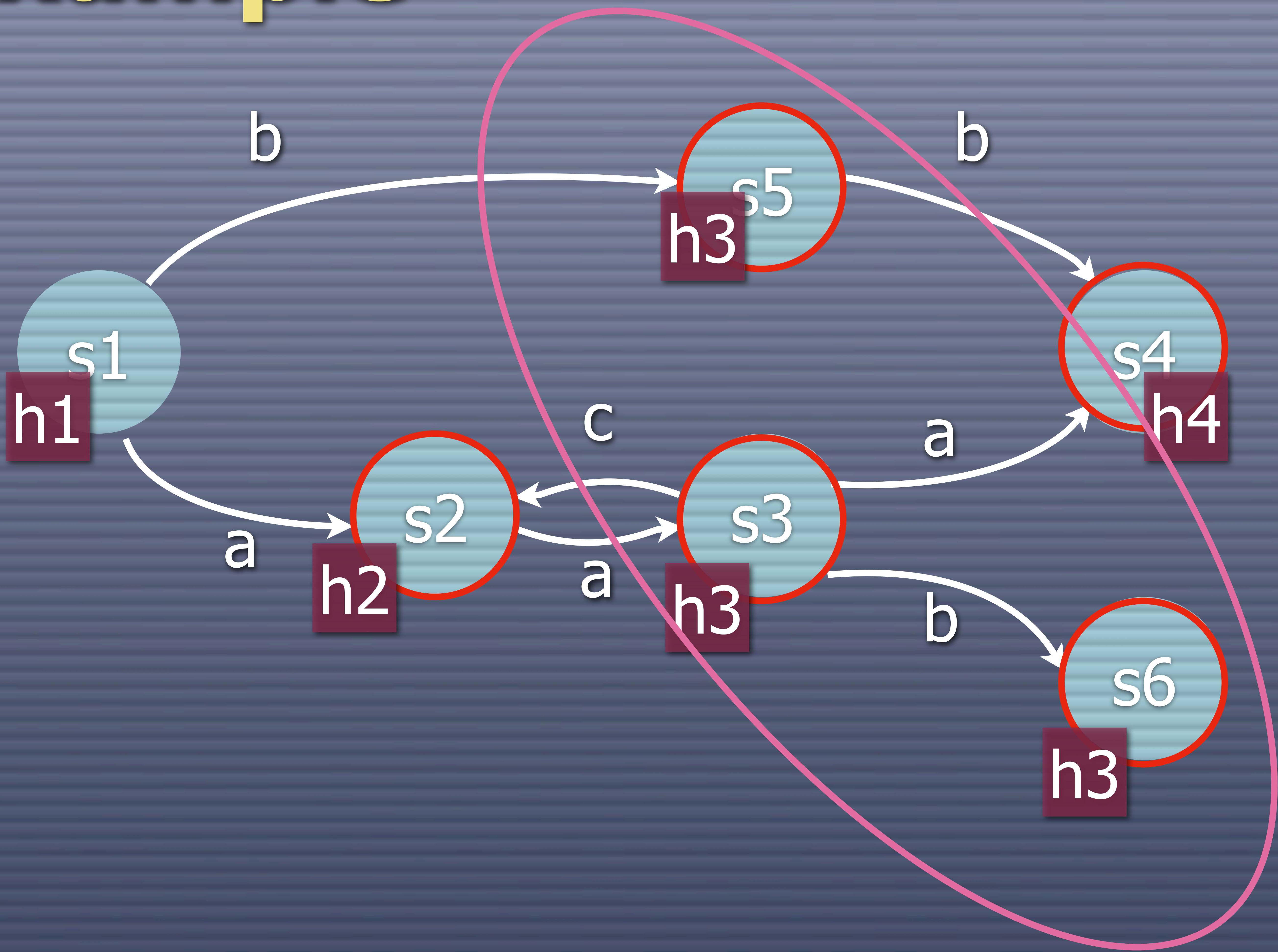
As long as we encounter no hash collisions, this algorithm works identically to the previous

We replace full state descriptors by compressed state descriptors in V

Example



Example



Example

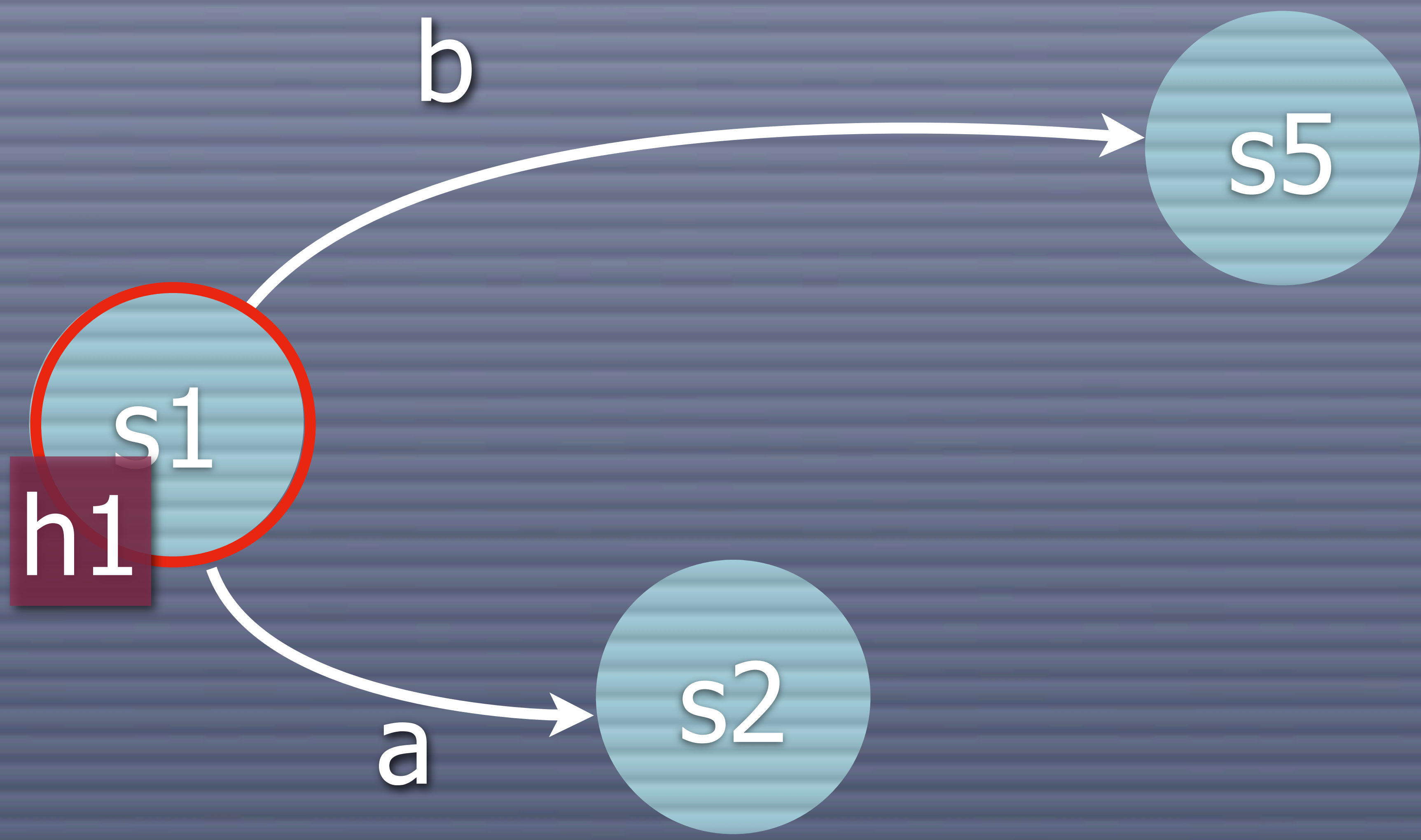
V: h1

W: s1

Example

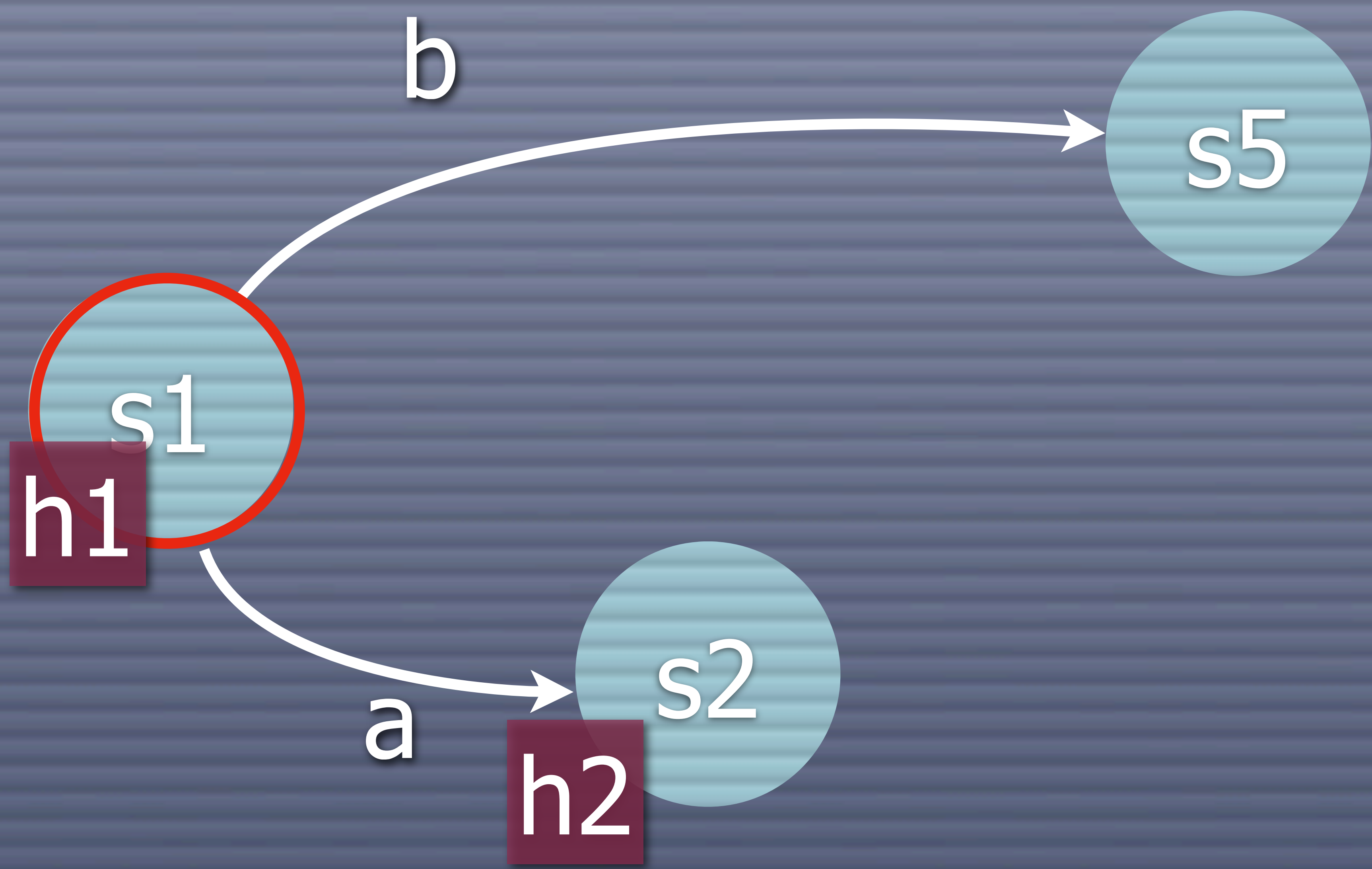
V: h1
W:

Example



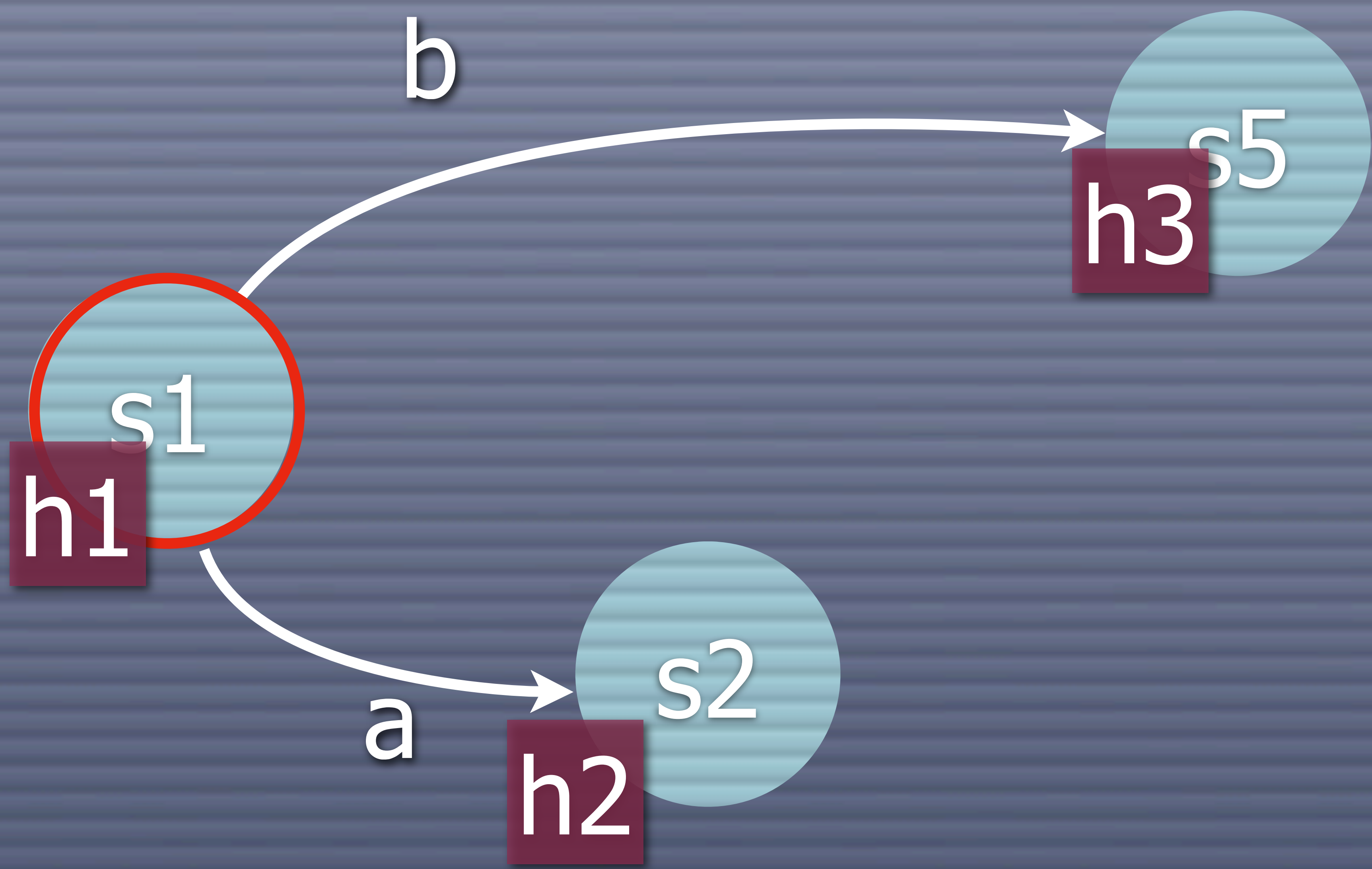
V: h1
W:

Example



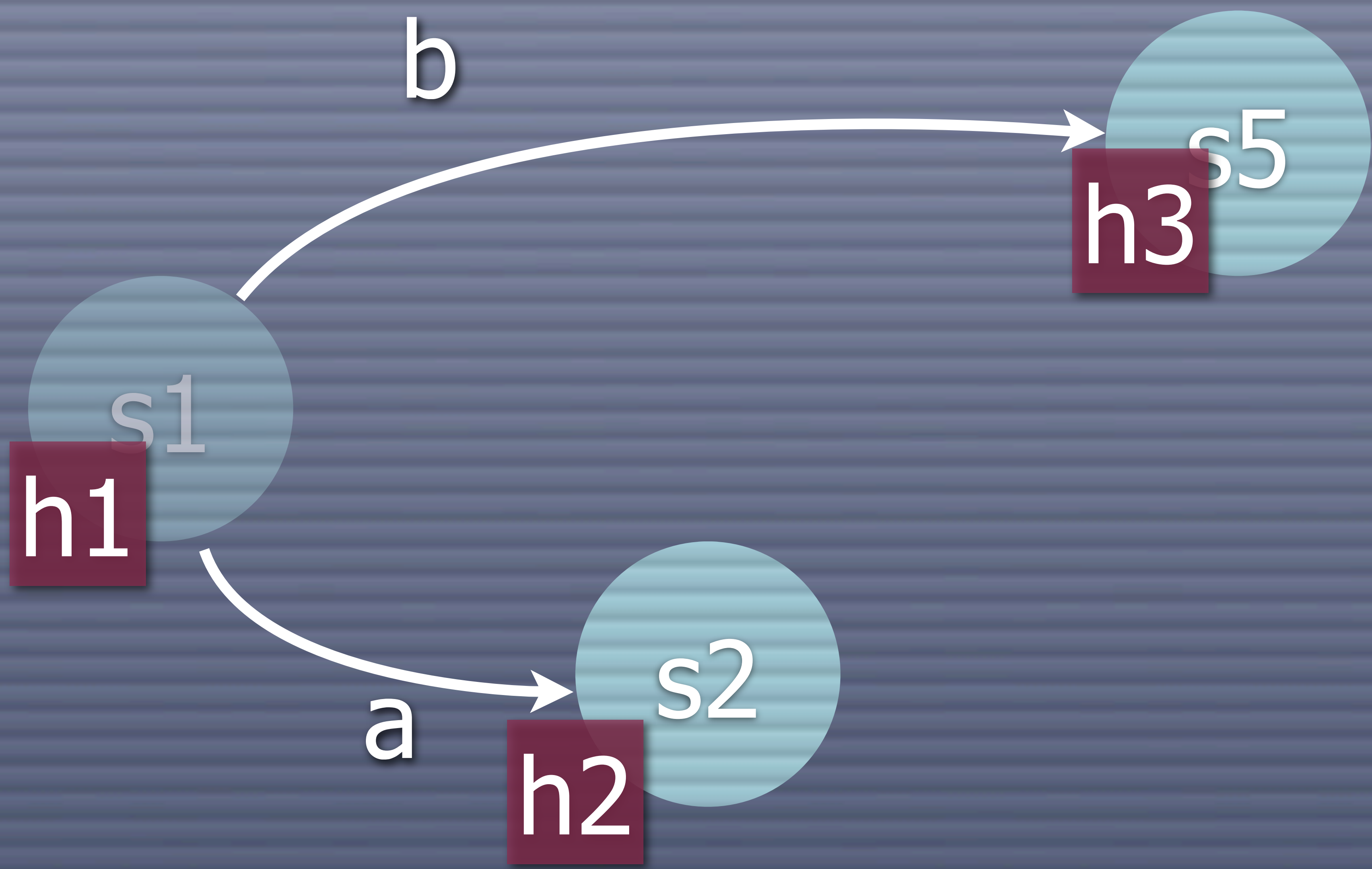
V: h1 h2
W: s2

Example



V: h1 h2 h3
W: s2 s5

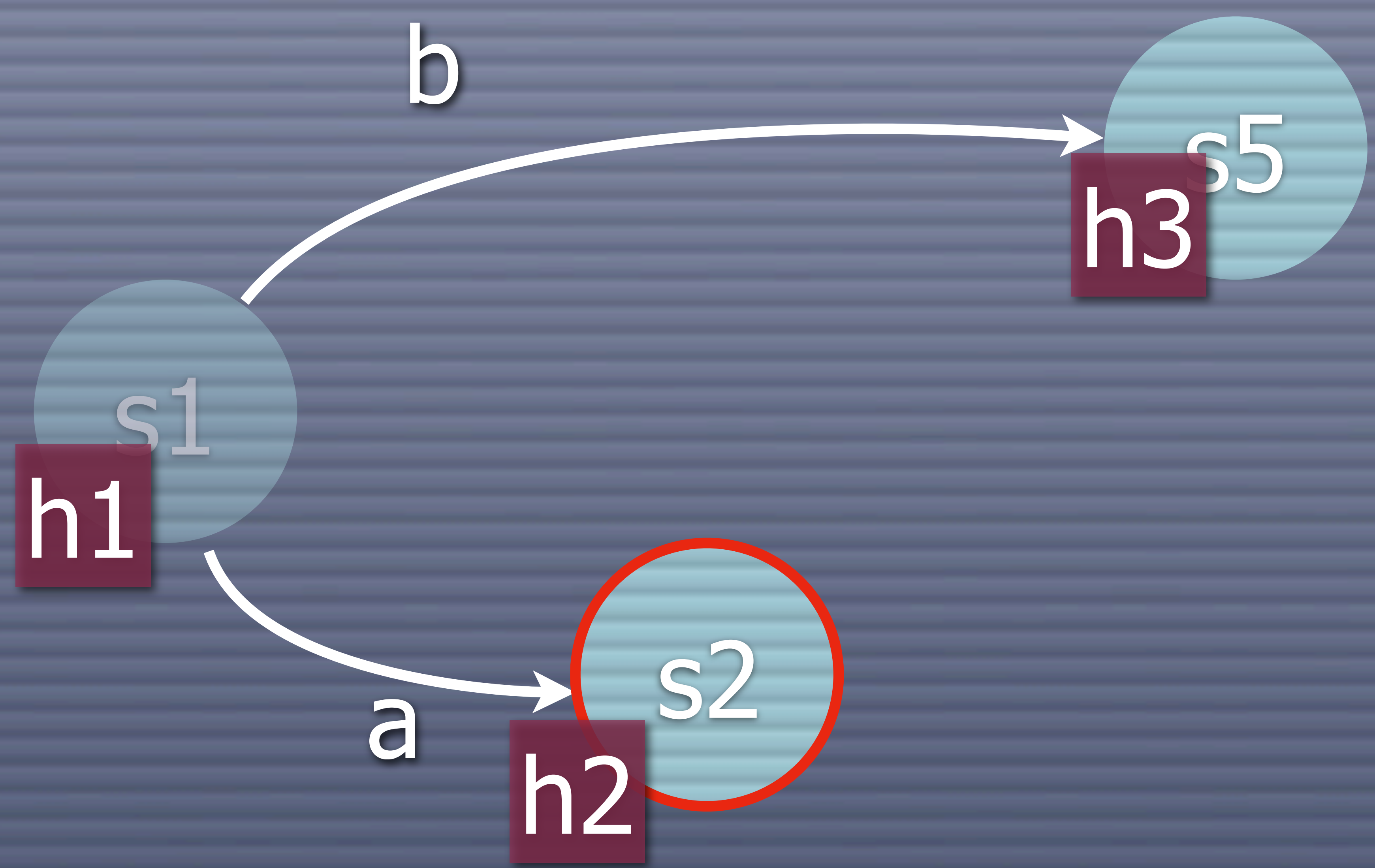
Example



V: h1 h2 h3

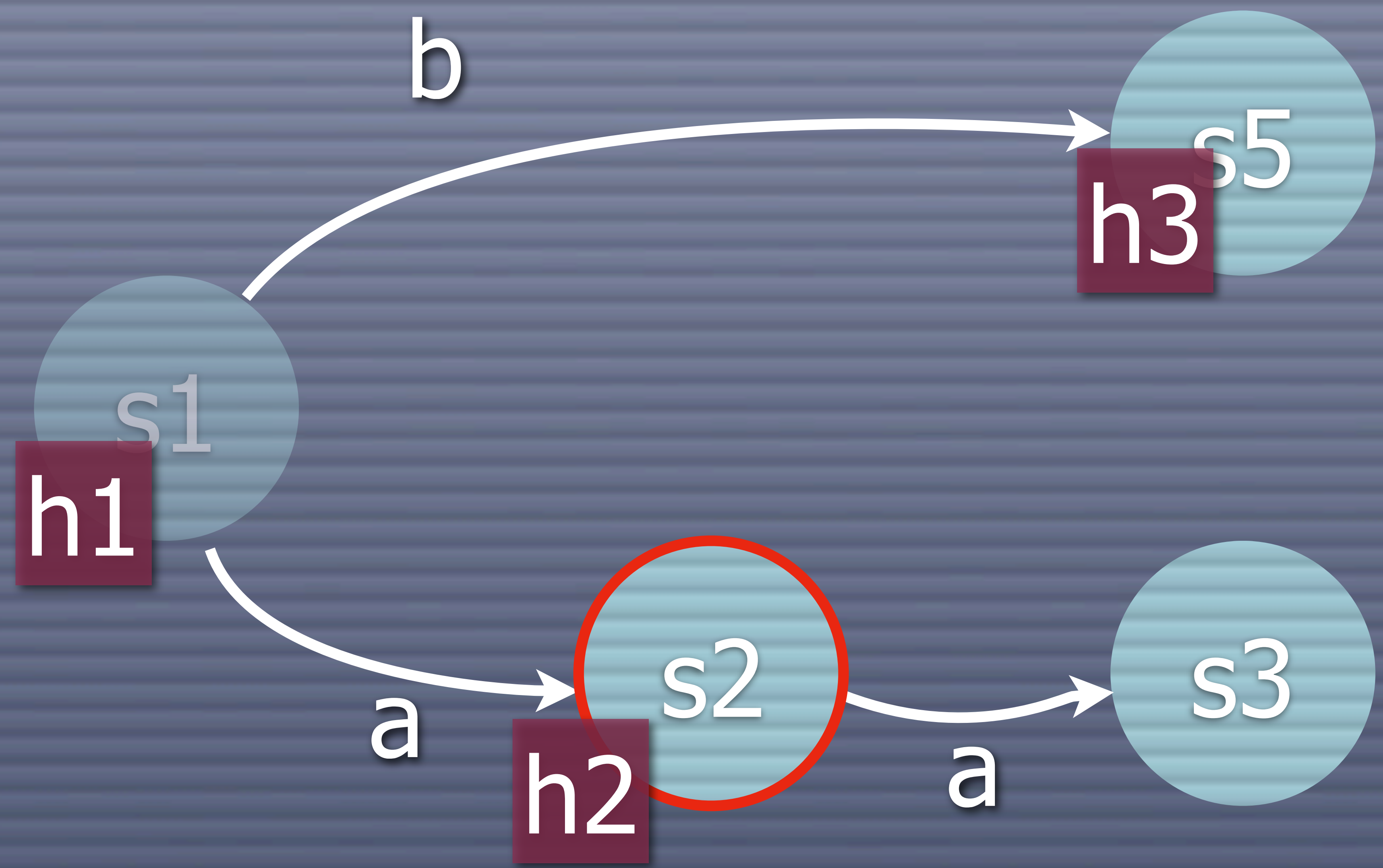
W: s2 s5

Example



V: h1 h2 h3
W: s5

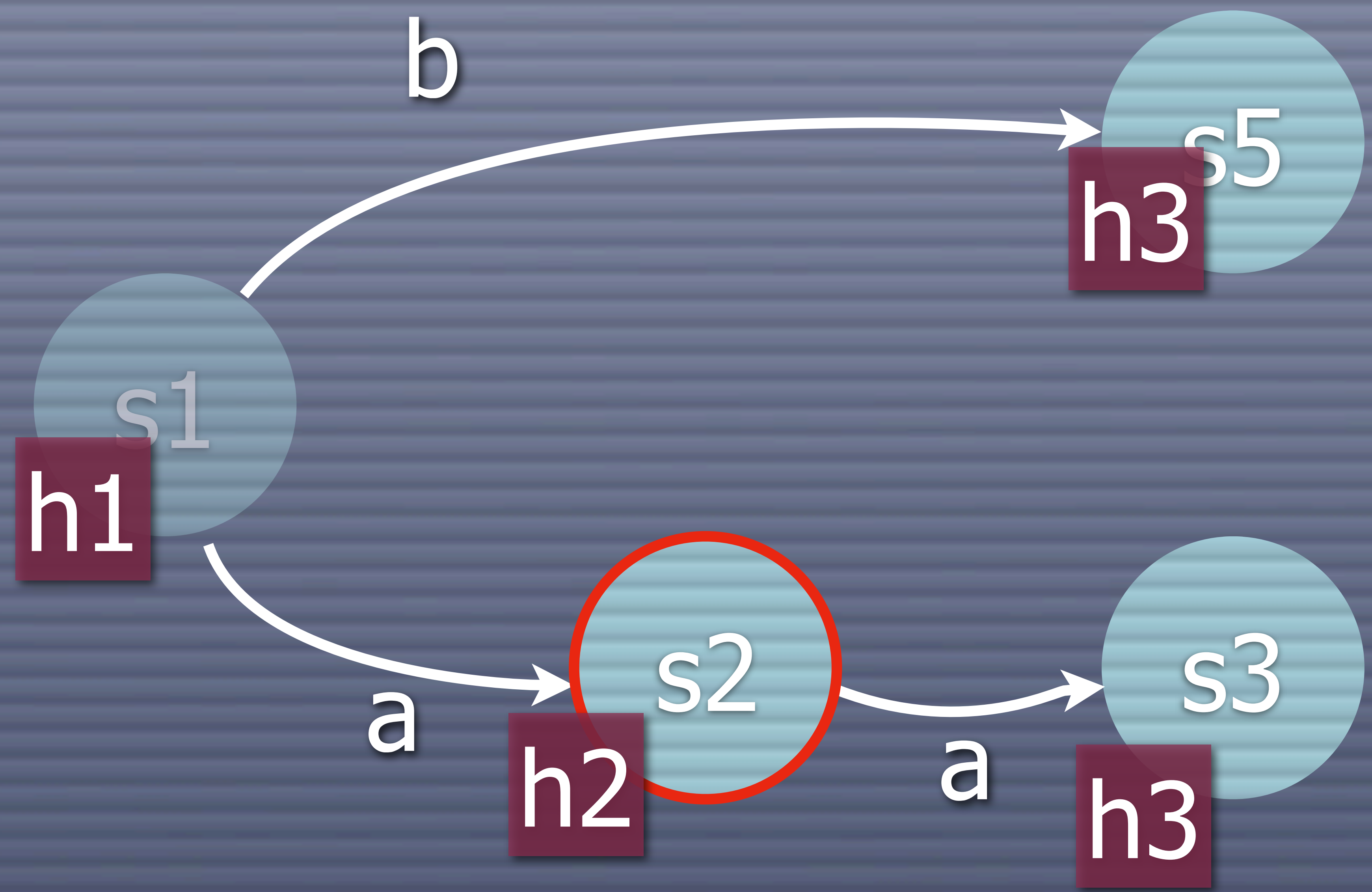
Example



V: h1 h2 h3

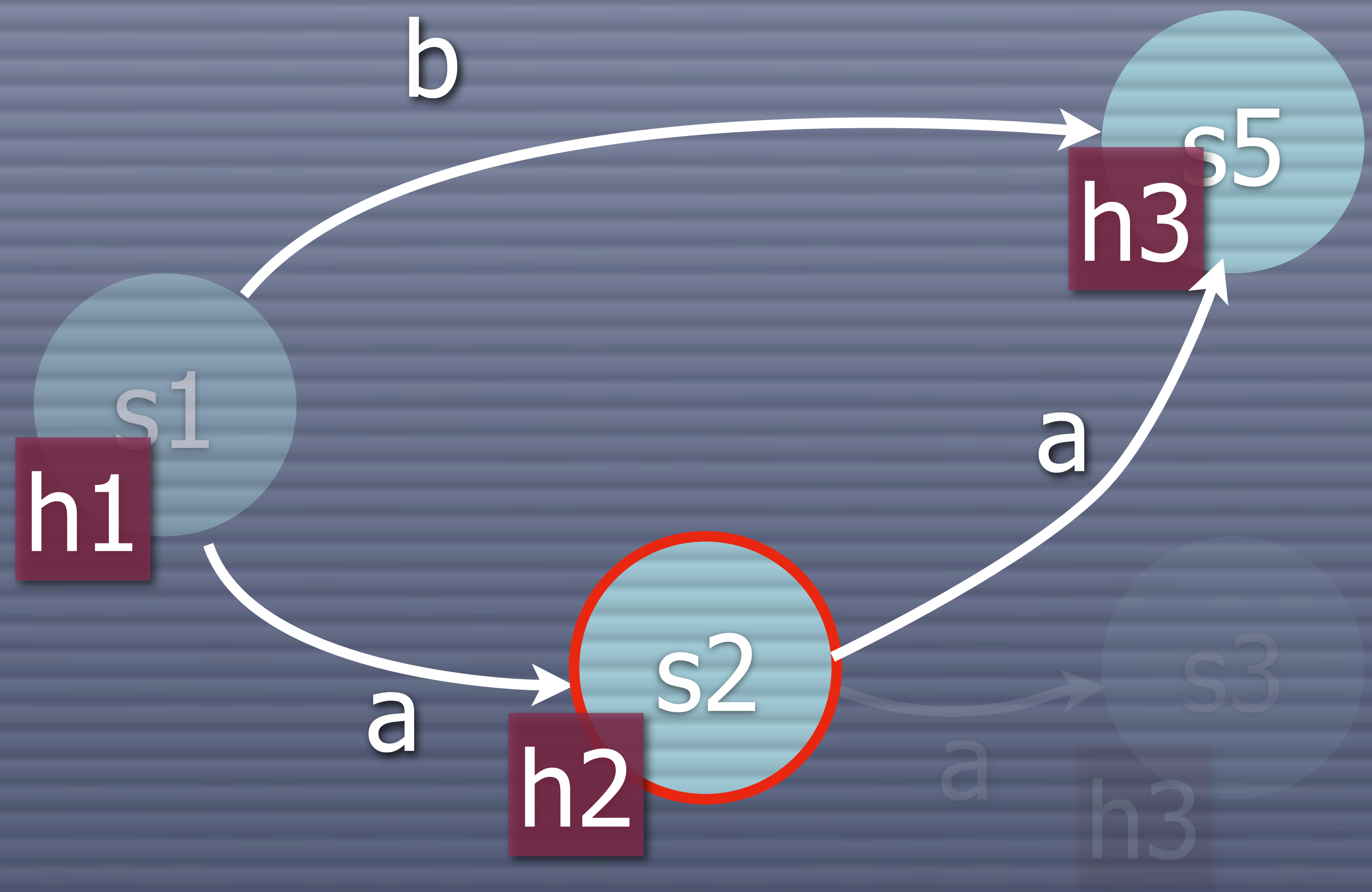
W: s5

Example



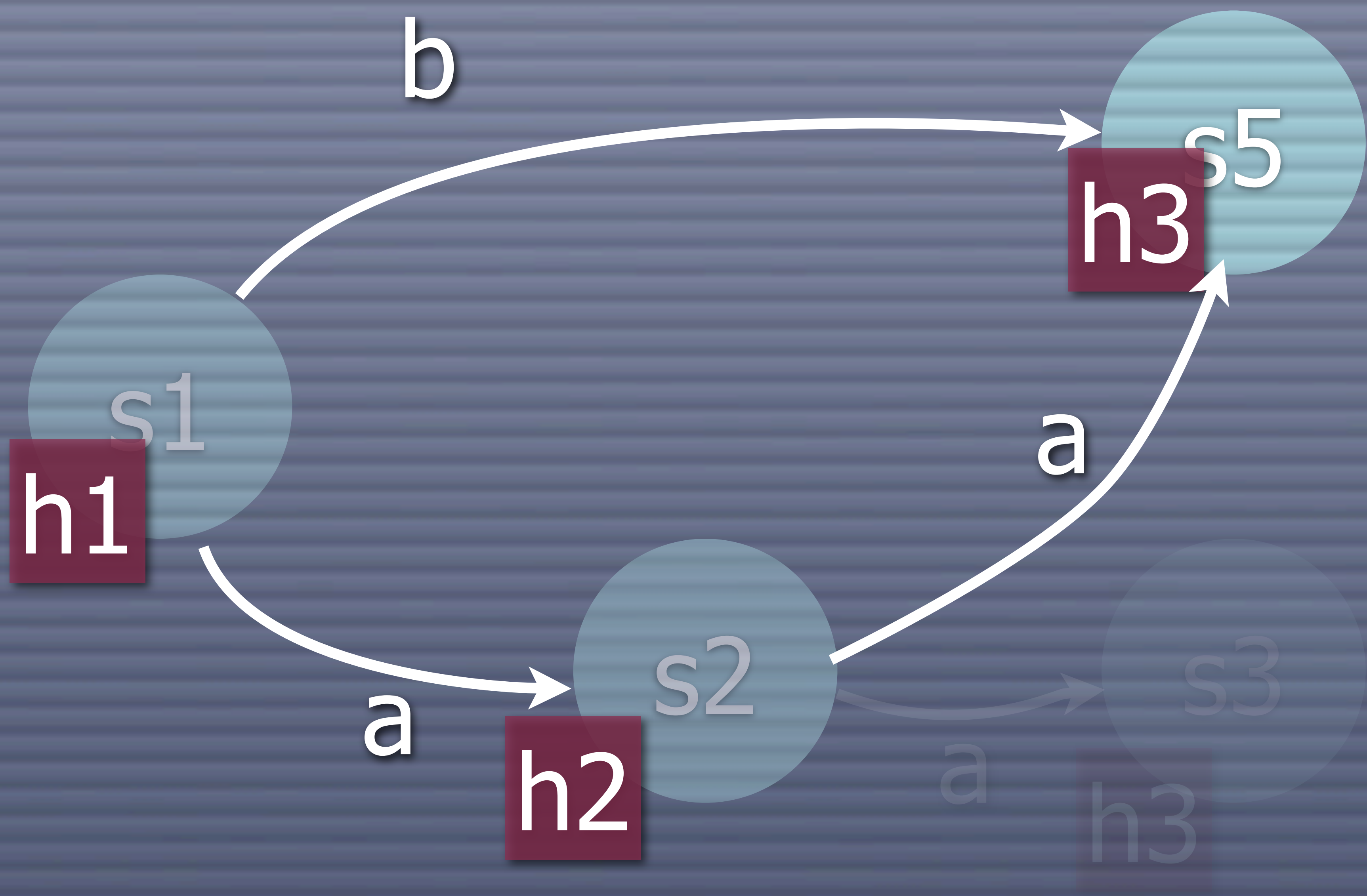
V: h1 h2 h3
W: s5

Example



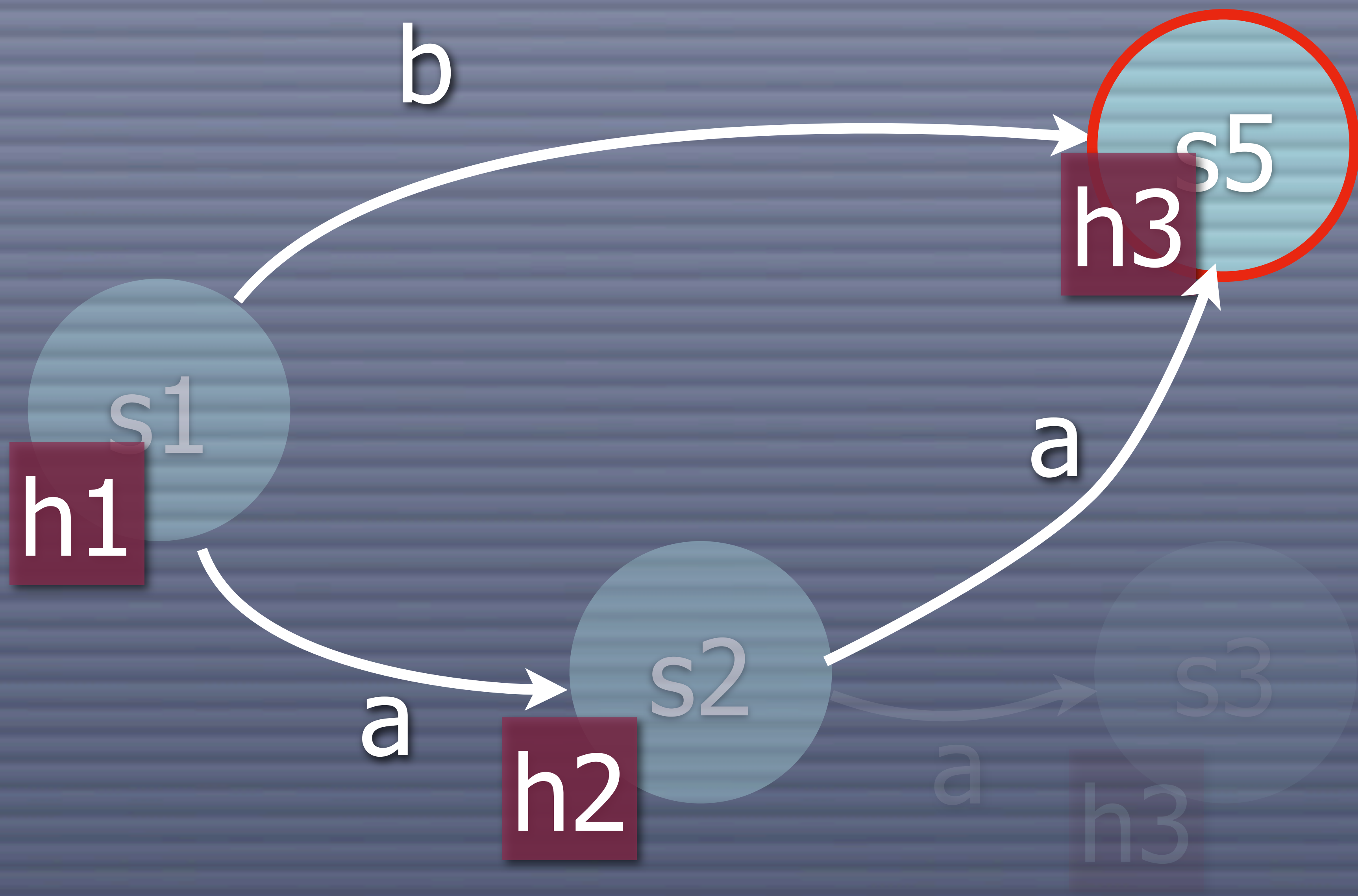
V: h1 h2 h3
W: s5

Example



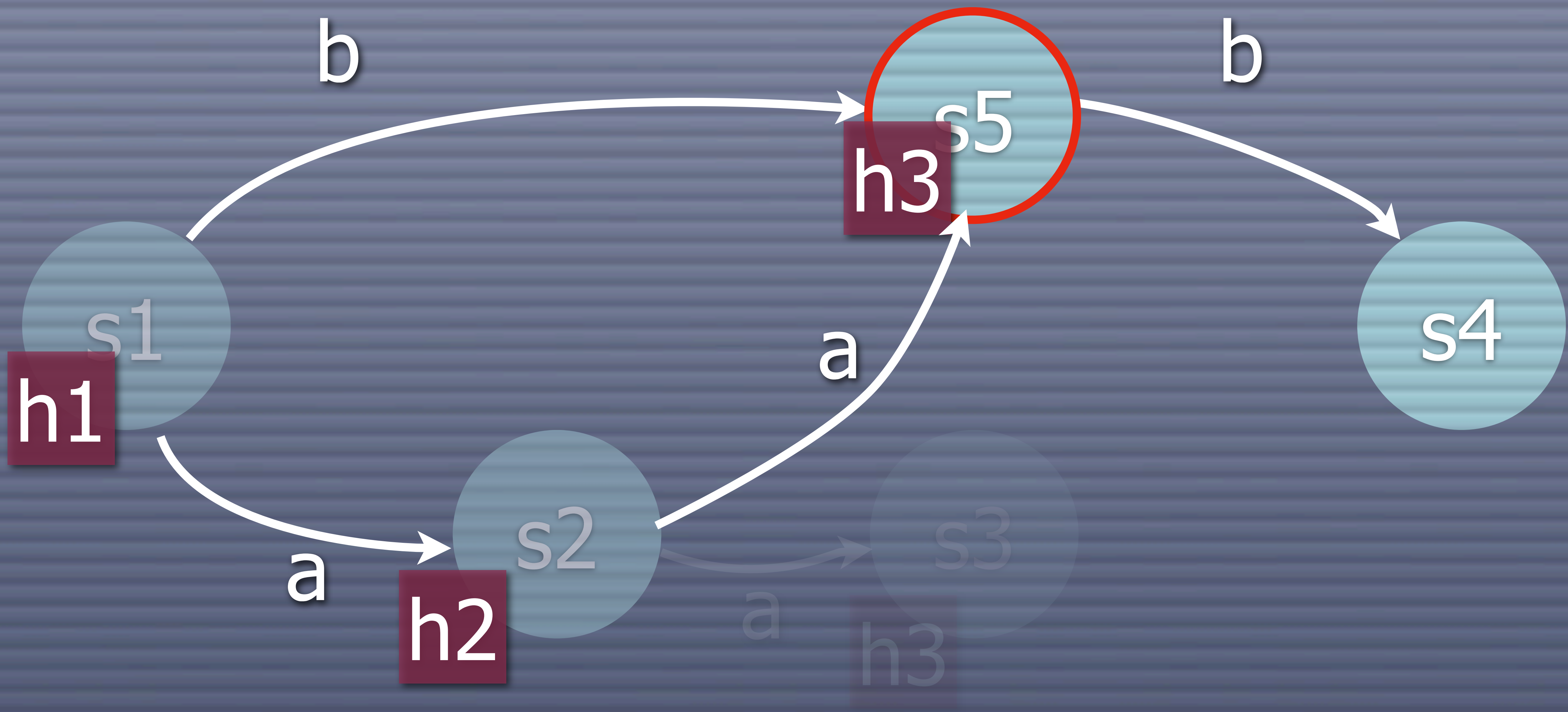
V: h1 h2 h3
W: s5

Example



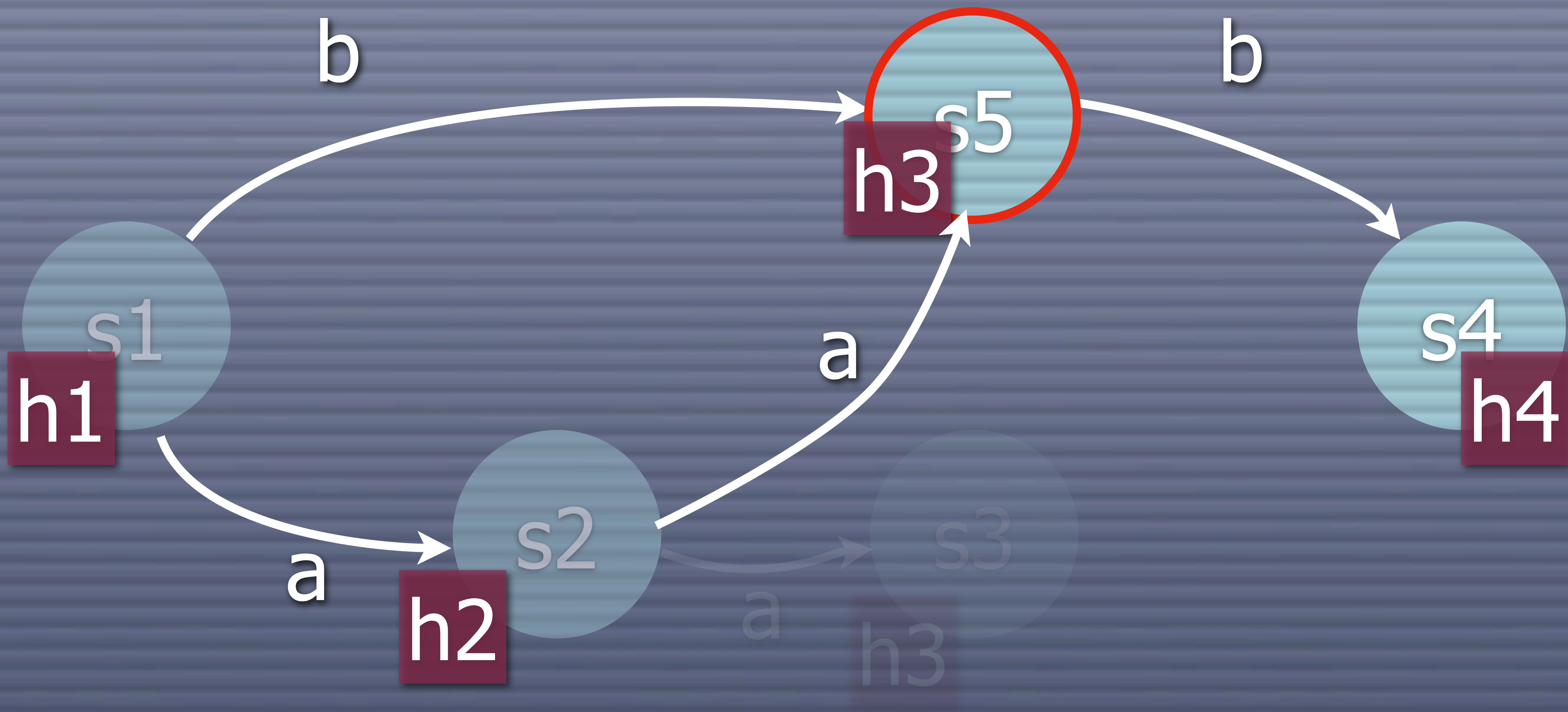
V: h1 h2 h3
W:

Example



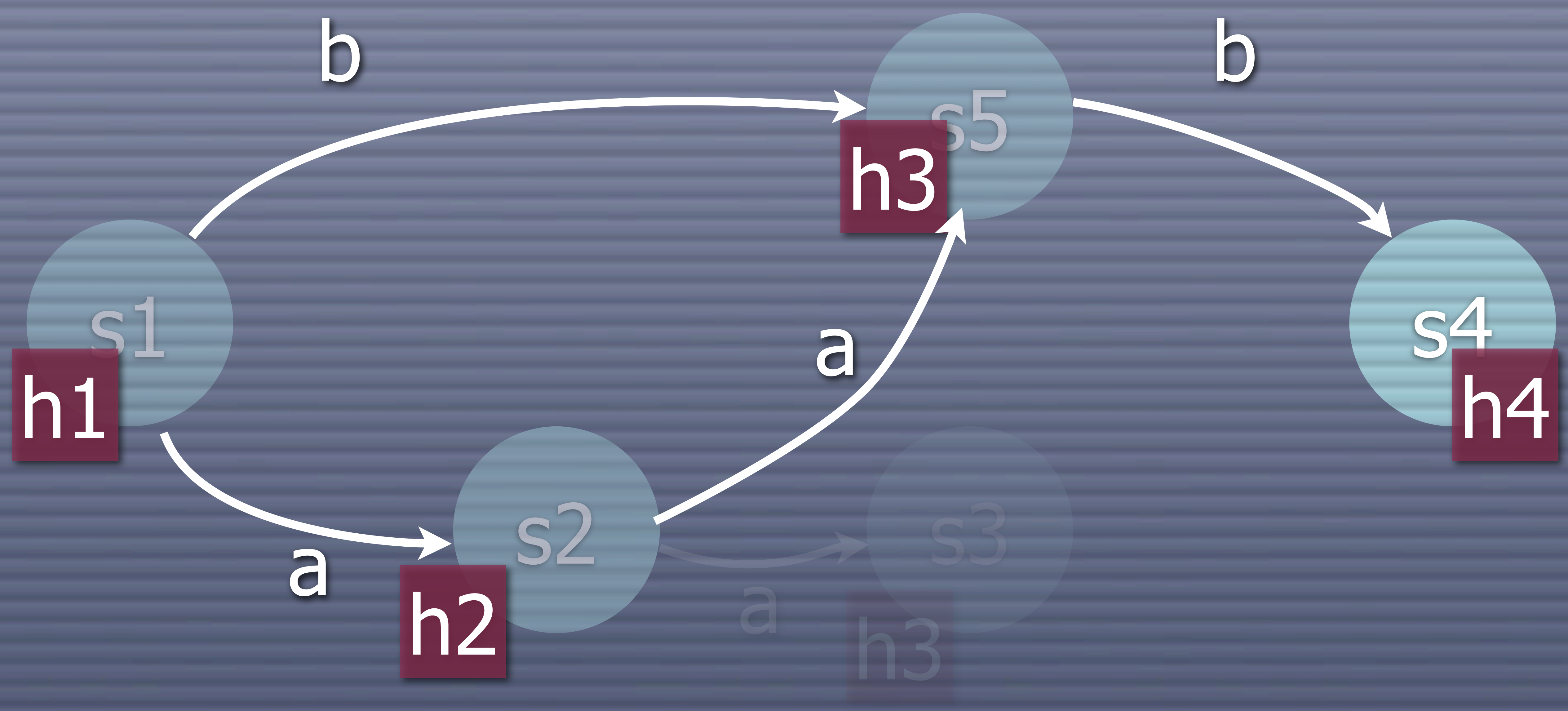
V: h1 h2 h3 h4
W: s4

Example



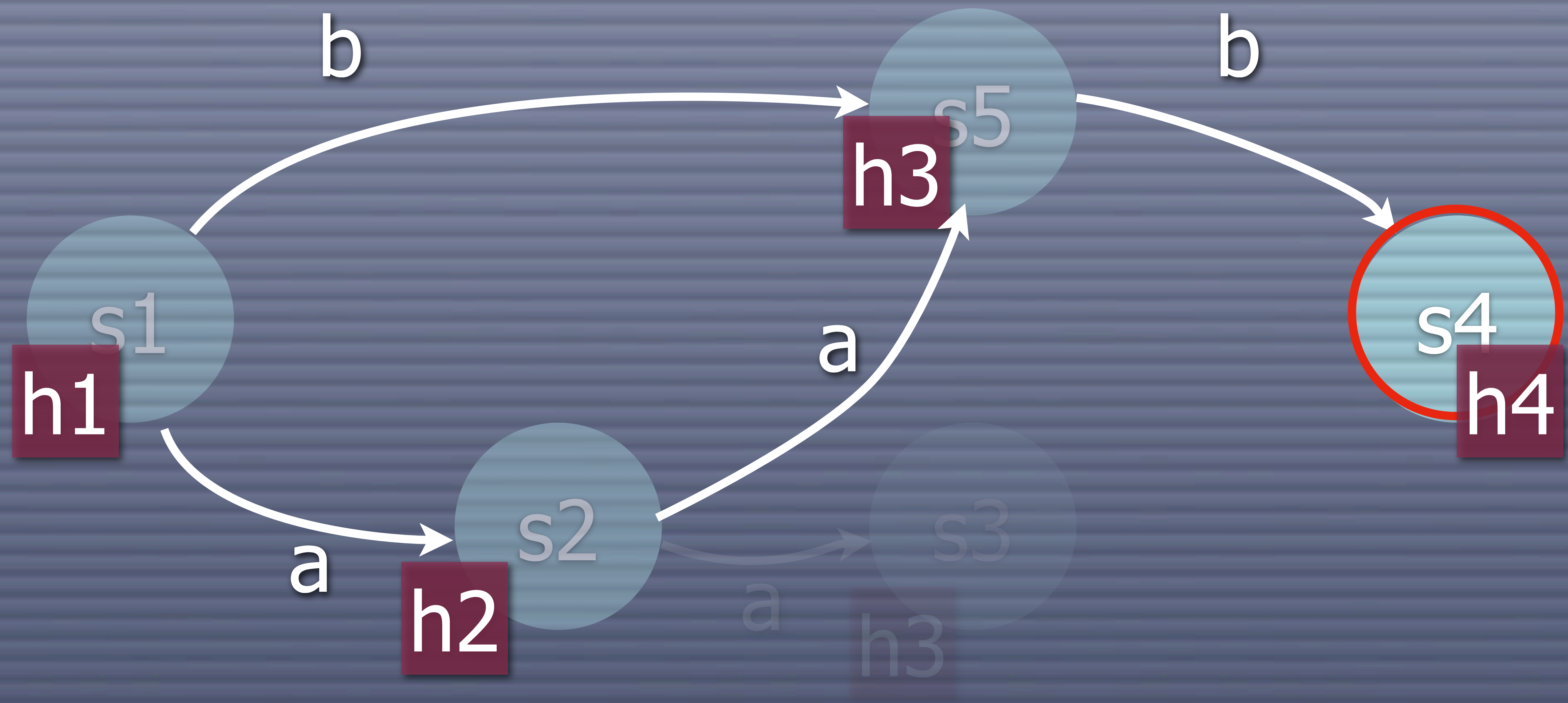
V: h1 h2 h3 h4
W: s4

Example



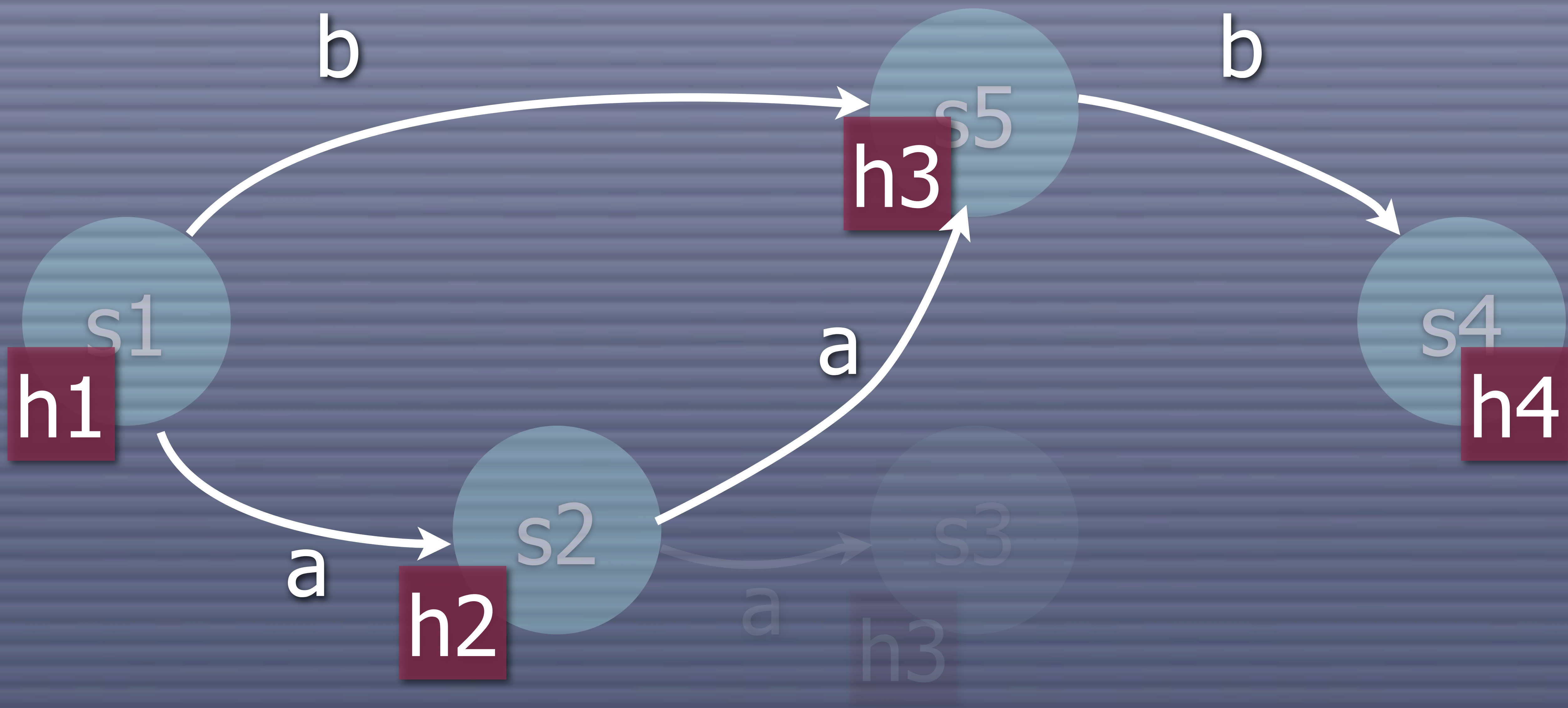
V: h1 h2 h3 h4
W: s4

Example



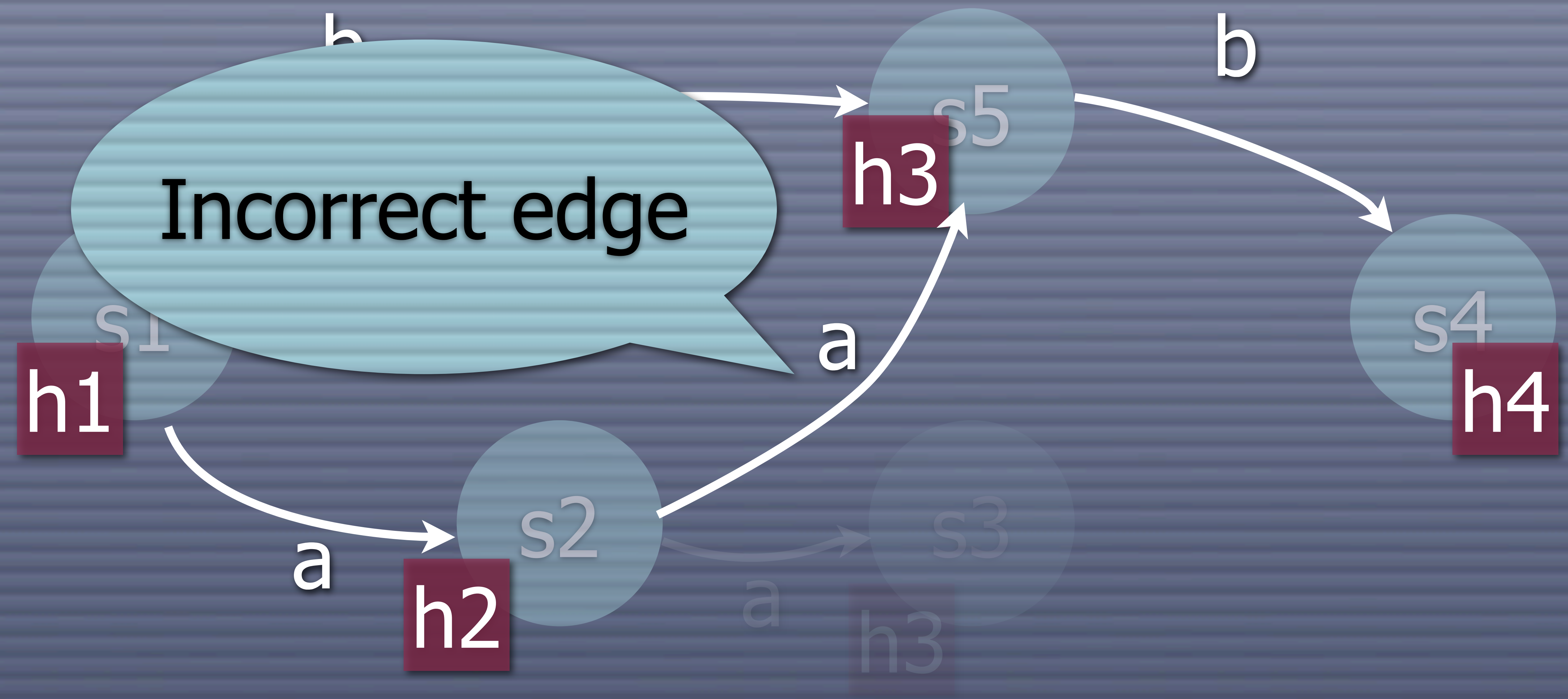
V: h1 h2 h3 h4
W:

Example



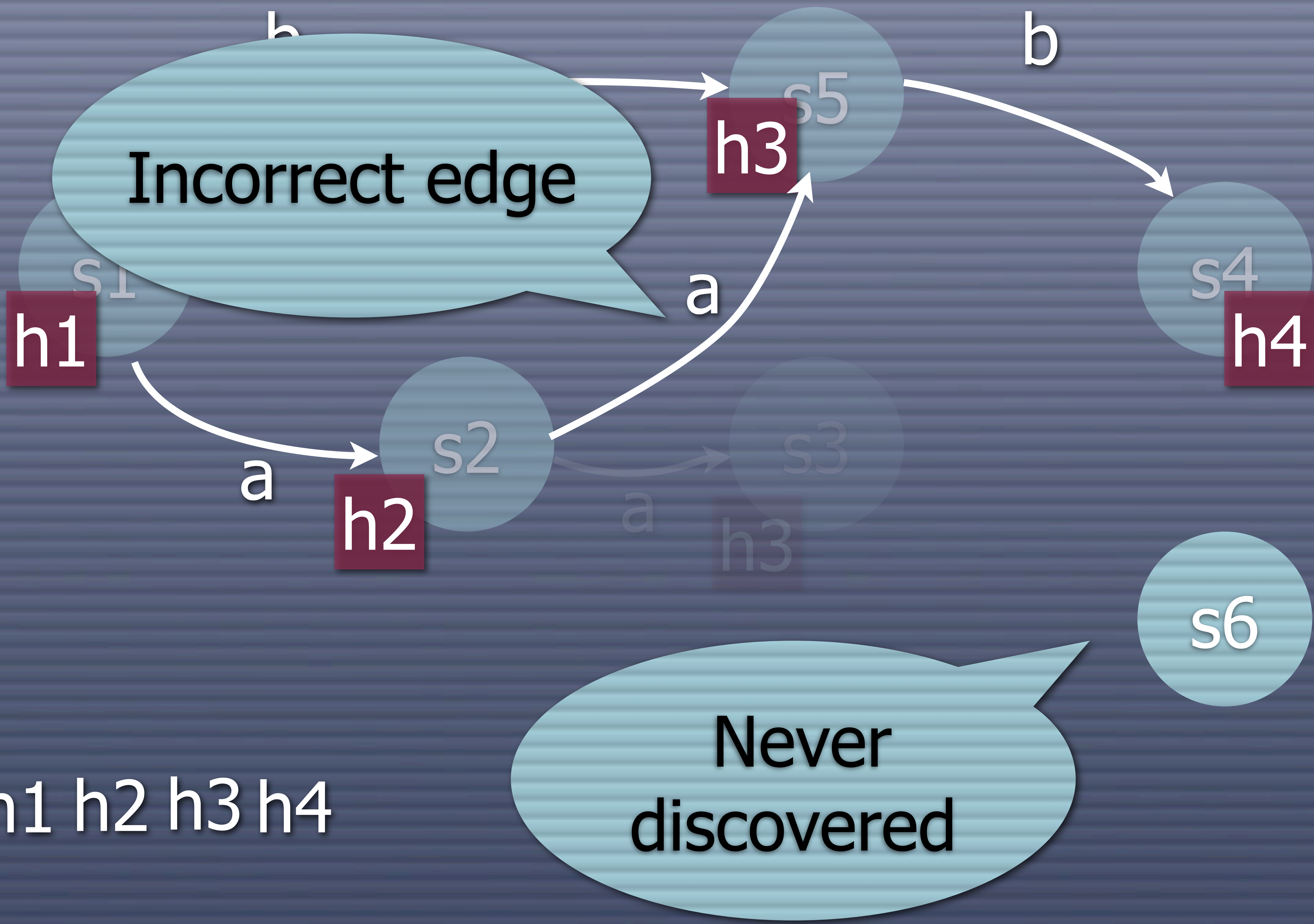
V: h1 h2 h3 h4
W:

Example



V: h1 h2 h3 h4
W:

Example



V: h1 h2 h3 h4
W:

Notes on Hash-compaction

- We find most but not all states
  - Improve coverage by using larger hash values
  - Improve coverage using more than one hash function
- SHA-1 uses 160 bits (20 bytes) per state and has no known collisions
- Uses around as much time as the standard algorithm and space is still O(# nodes) but with a smaller factor

Demo: Hash-compaction

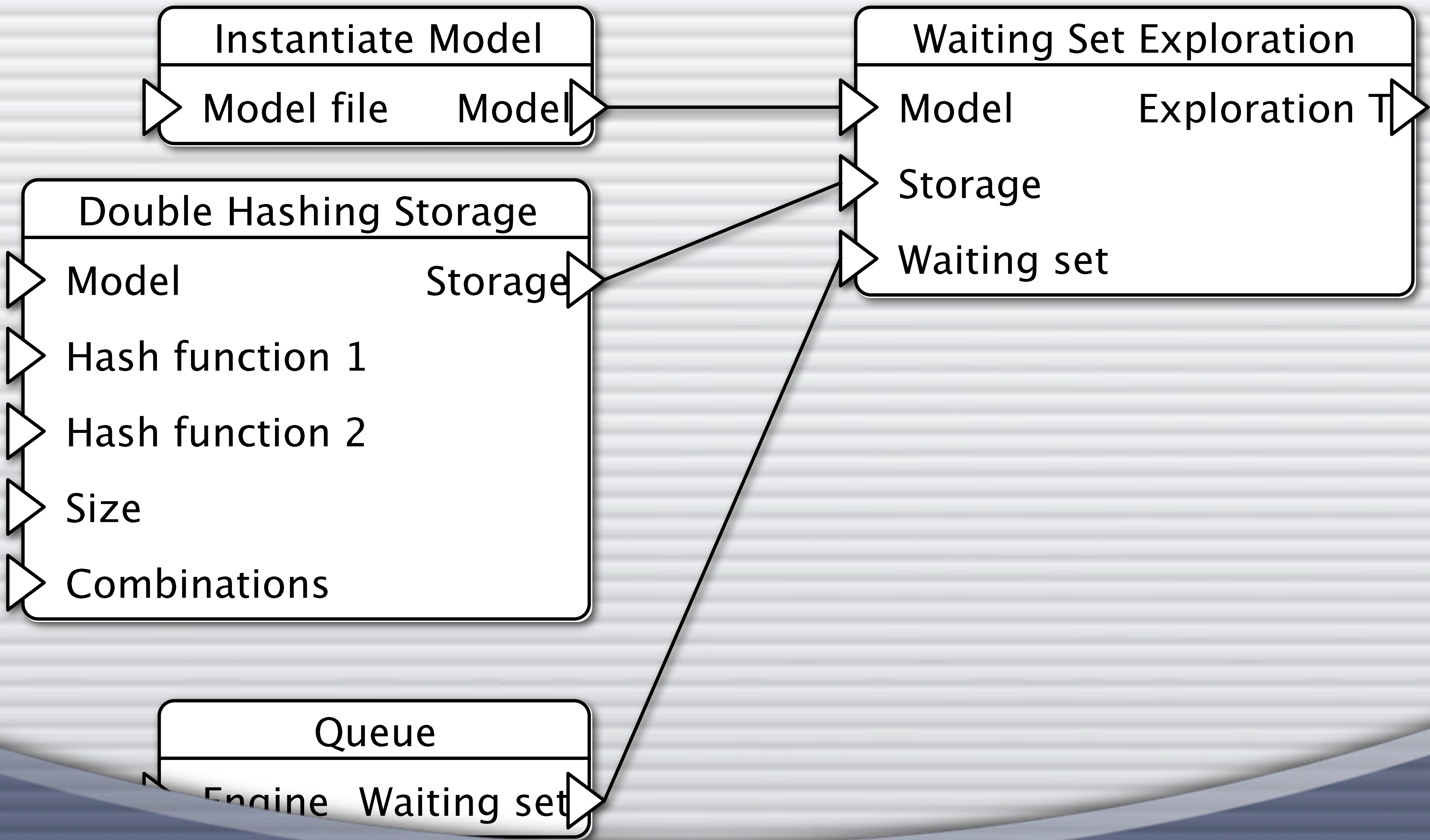
- Replace storage in standard method
  - We **can** but **should not** compute error traces
  - If we use DFS traversal, computing error traces is no problem

Numbers

Model	Nodes	NodesHC	Mem	MemHC	%	/st	/stHC
DP22	39604	39603	23.6	20.8	88	625	550
DB10	196832	196798	174.0	4.9	3	927	26
SW7,4	215196	214569	43.0	5.2	12	210	25
TS5	107648	107647	61.2	45.7	75	596	445
ERDP2	207003	206921	87.4	5.1	6	443	26
ERDP3	4277126	4270926	-	113.5	-	-	28

Numbers

Model	Nodes	NodesHC	Mem	MemHC	%	/st	/stHC
DP22	39604	39603	23.6	20.8	88		550
DB10	196832	196798	174.0	4.9	3	927	26
SW7,4	215196	214569	43.0	5.2	12	210	25
TS5	107648	107647	61.2	45.7	75		445
ERDP2	207003	206921	87.4	5.1	6	443	26
ERDP3	4277126	4270926	-	113.5	-	-	28



Example: Bit-state Hashing

Bit-state Hashing

- Hash-compaction uses a hash function to compress state descriptor and stores the compressed vectors
- Bit-state hashing instead uses a hash function to compute an index in an array and sets a bit if a corresponding state has been seen
- We need an array of size $2^{|h(s)|}/8$ bytes, e.g., $2^{32}/8$ = 500 Mb to get same coverage as hash compaction

Hash-compaction

$V := \{ s_0 \}$
$W := \{ s_0 \}$
while $W \neq \emptyset$ **do**
    Select an $s \in W$
    $W := W \setminus \{ s \}$
    if $\neg I(s)$ **then**
        return false
    for all t, s' **such that** $s \to^t s'$ **do**
        if $s' \notin V$ **then**
            $V := V \cup \{ s' \}$
            $W := W \cup \{ s' \}$
return true

We replace full state descriptors with bit-array access.

Hash-compaction

```
V := new bool[2^|h(s)|]; V[h(s0)] := true
W := { s0 }
while W ≠ ∅ do
    Select an s ∈ W
    W := W ∖ { s }
    if ¬I(s) then
        return false
    for all t, s′ such that s →t s′ do
        if ¬V[h(s′)] then
            V[h(s′)] := true
            W := W ∪ { s′ }
return true
```

We replace full state descriptors with bit-array access.

Hash-compaction

V := new bool[$2^{|h(s)|}$]; $V[h(s_0)]$:= true

$W := \{ s_0 \}$

while $W \neq \emptyset$ **do**

  Select an $s \in W$

  $W := W \setminus \{ s \}$

  if $\neg I(s)$ **then**

    return false

  for all t, s' **such that** $s \xrightarrow{t} s'$ **do**

    if $\neg V[h(s')]$**en**

      $V[h(s')]$:= true

      $W := W \cup \{ s' \}$

return true

This works exactly like hash-compaction with the same hash function.

We replace full state descriptors with bit-array access.

Bit-state Hashing vs. Hash-compaction

- Both allow us to increase the size of the compressed state descriptor to get better coverage, but for bit-state hashing each extra bit doubles memory usage
- Hash-compaction uses memory proportional to the size of the number of nodes, bit-state hashing uses a constant amount of memory
- Hash-compaction uses memory proportional to the number of hash functions we use, bit-state hashing uses a constant amount of memory

Bit-state Hashing vs. Hash-compaction

- Both allow us to increase the size of the compressed state descriptor to get better coverage, but for bit-state hashing each extra bit doubles memory usage
- Hash-compaction uses memory proportional to the size of the number of nodes, bit-state hashing uses a constant amount of memory
- Hash-compaction uses memory proportional to the number of hash functions we use, bit-state hashing uses a constant amount of memory

More Hash Functions

- Using 2 hash functions require that we have 2 collisions instead of just one
  - But we may have a new kind of collisions, $h_1(s_1) = h_2(s_2)$
- Using more hash functions improves coverage to a certain point where the bit-array gets “filled up”, so collisions become more common

Hash-compaction

$V := \text{new bool}[2^{|h(s)|}]$; $V[h(s_0)] := \text{true}$
$W := \{ s_0 \}$
while $W \neq \varnothing$ **do**
  Select an $s \in W$
  $W := W \setminus \{ s \}$
  if $\neg I(s)$ **then**
    return false
  for all t, s' **such that** $s \xrightarrow{t} s'$ **do**
    if $\neg V[h(s')]$ **then**
      $V[h(s')] := \text{true}$
      $W := W \cup \{ s' \}$
return true

We simply set and read bits for both (or all) hash functions.

Hash-compaction

$V := \text{new bool}[2^{|h(s)|}]$; $V[h(s_0)] := \text{true}$; $V[h_2(s_0)] := \text{true}$
$W := \{ s_0 \}$
while $W \neq \emptyset$ **do**
  Select an $s \in W$
  $W := W \setminus \{ s \}$
  if $\neg I(s)$ **then**
    return false
  for all t, s' **such that** $s \rightarrow^{t} s'$ **do**
    if $\neg V[h(s')]$ **or** $\neg V[h_2(s')]$
      $V[h(s')] := \text{true}$; $V[h_2(s')] := \text{true}$
      $W := W \cup \{ s' \}$
return true

We simply set and read bits for both (or all) hash functions.

Double Hashing

- Calculating hash functions is actually pretty expensive, so the time complexity grows with the number of hash functions
- Simply using $h_n(s) = n \bullet h_1(s)$ does **not** work!
- It turns out that using $h_n(s) = n \bullet h(s) + h'(s)$ does work; this is called double hashing
- Triple hashing works better but takes more time
- Experiments show that using 15-20 hash functions works well

Demo:
Bit-state Hashing

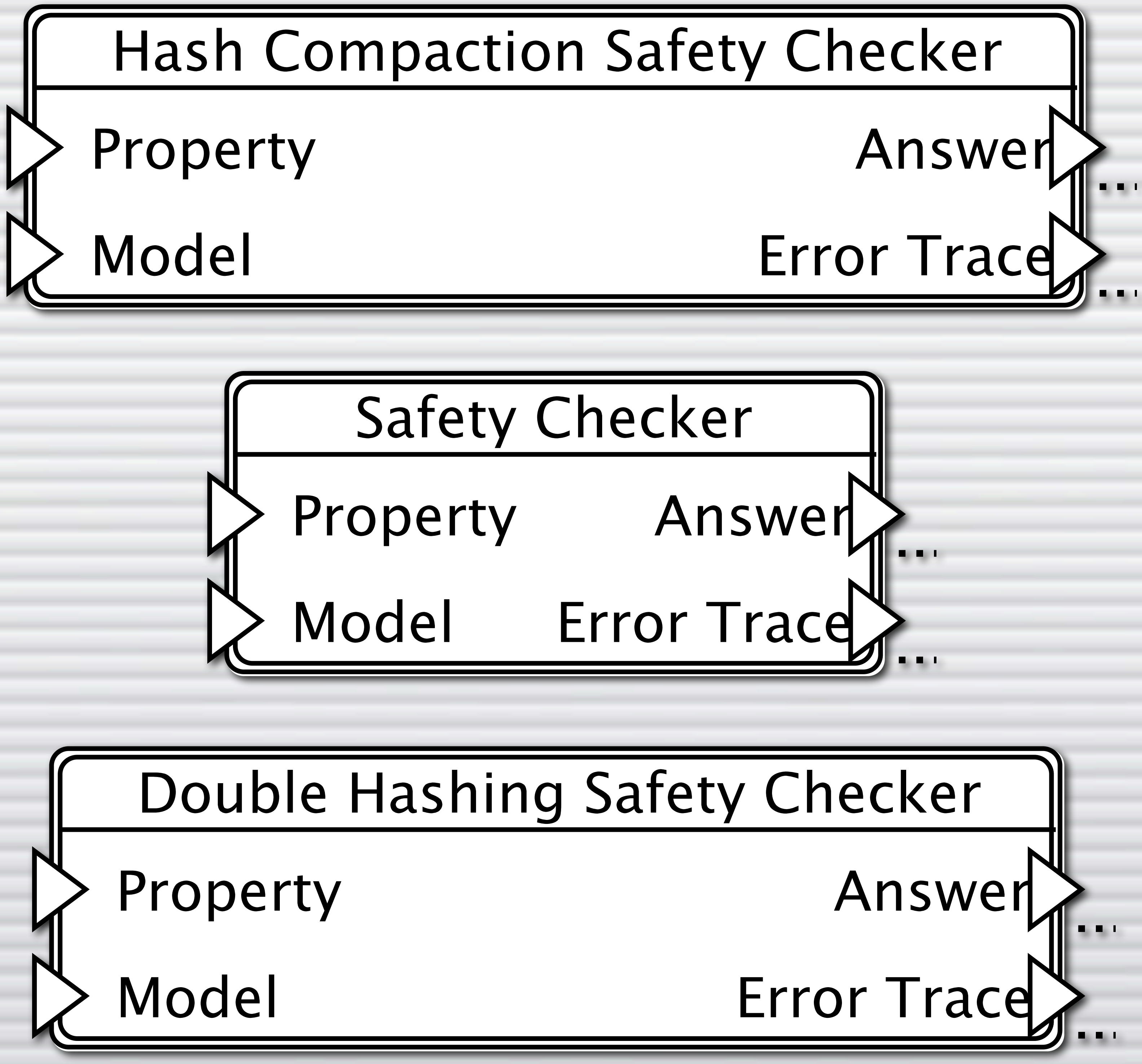
- Replace storage on standard example

Numbers

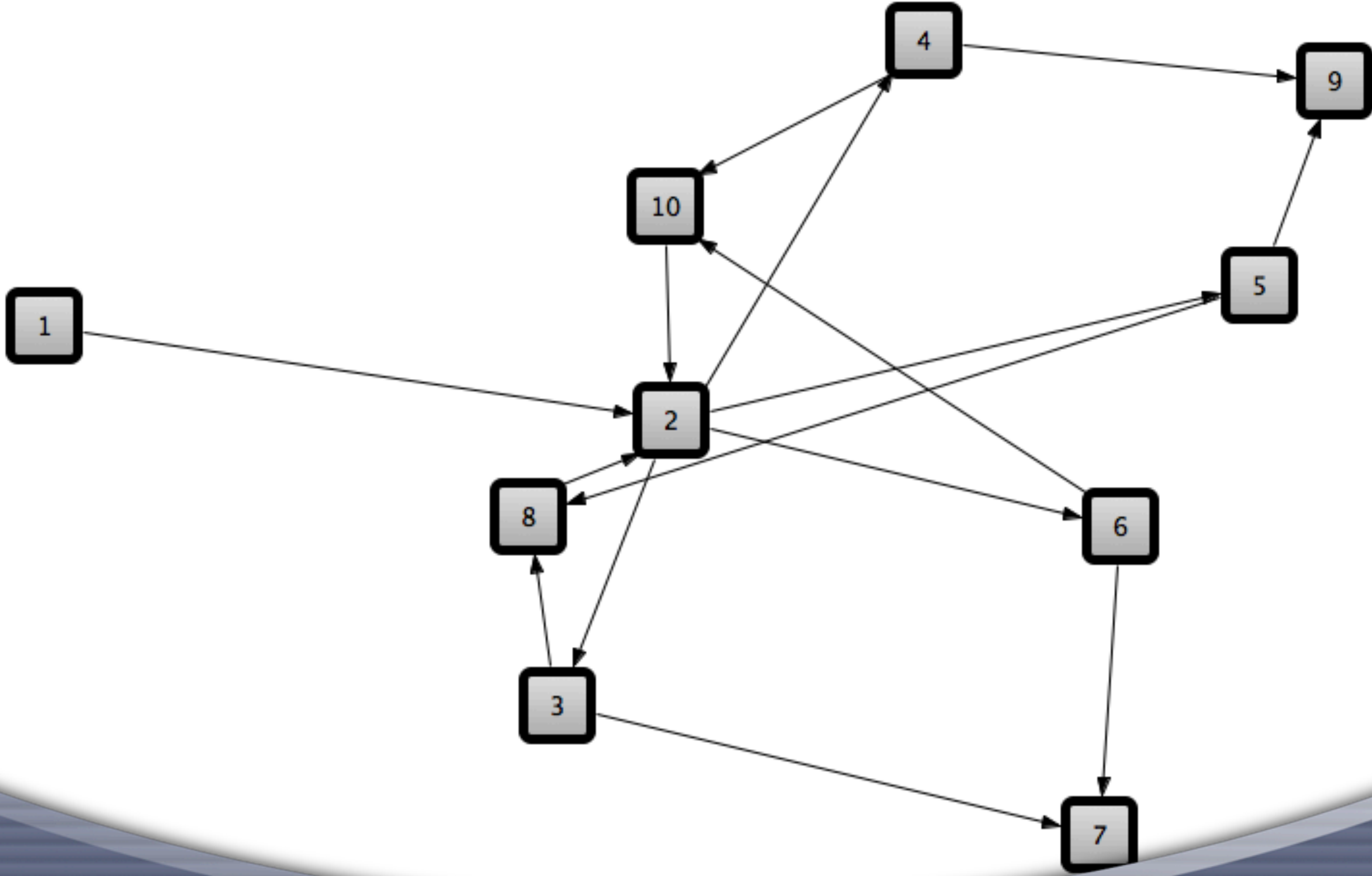
Model	Nodes	NodesDH	Mem	MemDH	%	/st	/stDH
DP22	39604	39604	23.6	32.0	135	625	846
DB10	196832	196832	174.0	12.3	7	927	66
SW7,4	215196	215196	43.0	12.3	28	210	60
TS5	107648	107648	61.2	55.4	90	596	540
ERDP2	207003	207003	87.4	12.3	14	443	62
ERDP3	4277126	4277125	-	12.1	-	-	3

More Numbers

Model	Nodes	MemHC	MemDH	/stateHC	/stateDH
DP22	39604	20.8	32.0	550	846
DB10	196832	4.9	12.3	26	66
SW7,4	215196	5.2	12.3	25	60
TS5	107648	45.7	55.4	445	540
ERDP2	207003	5.1	12.3	26	62
ERDP3	4277126	113.5	12.1	28	3



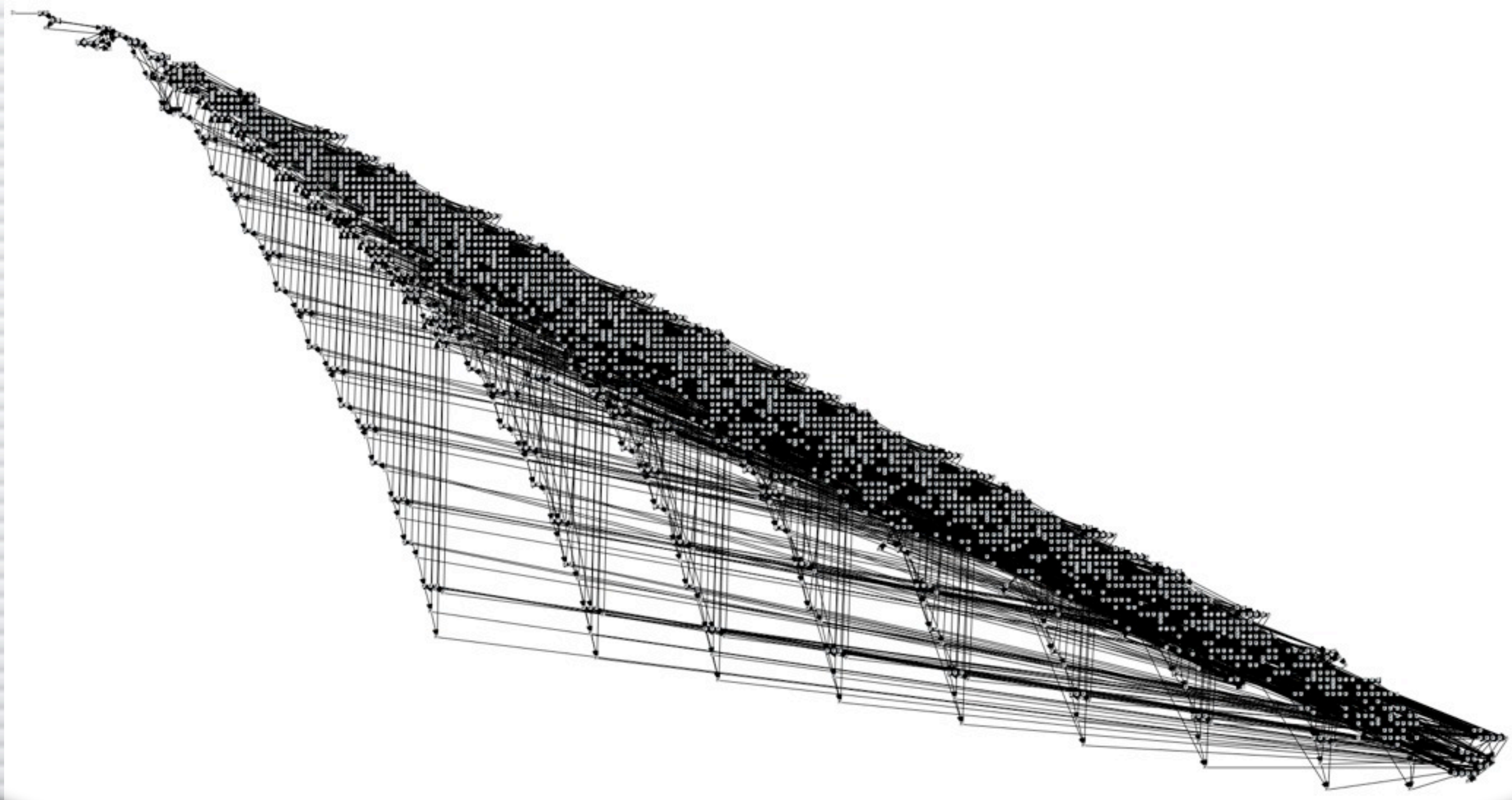
Comparing the Top-levels



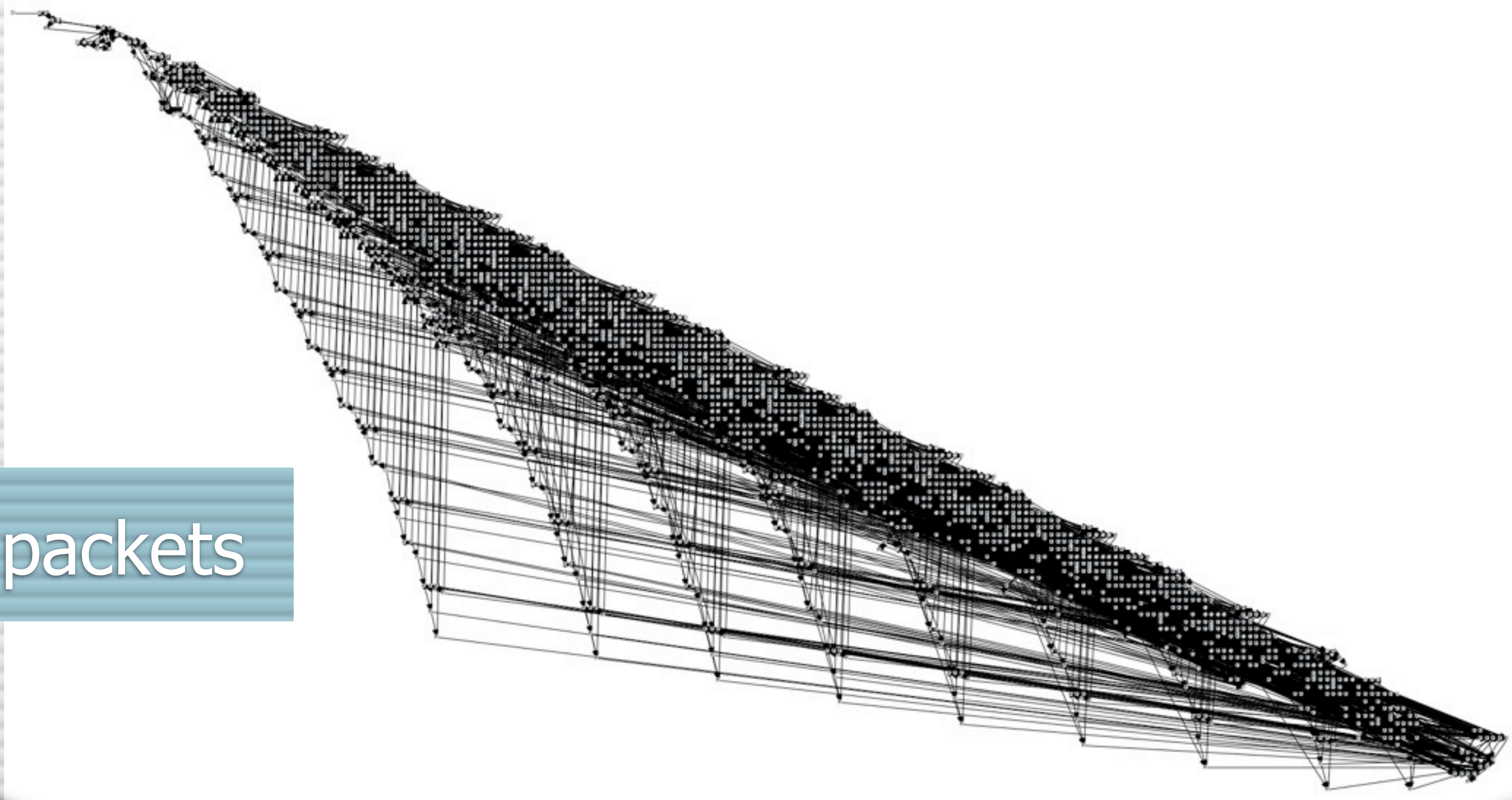
Example:
Drawing SS Graphs

Demo: Drawing SS Graphs

- Change safety checker to draw SS graph
- Change model size to 2 philosophers
- Play with layouts
- Export to DOT and GML

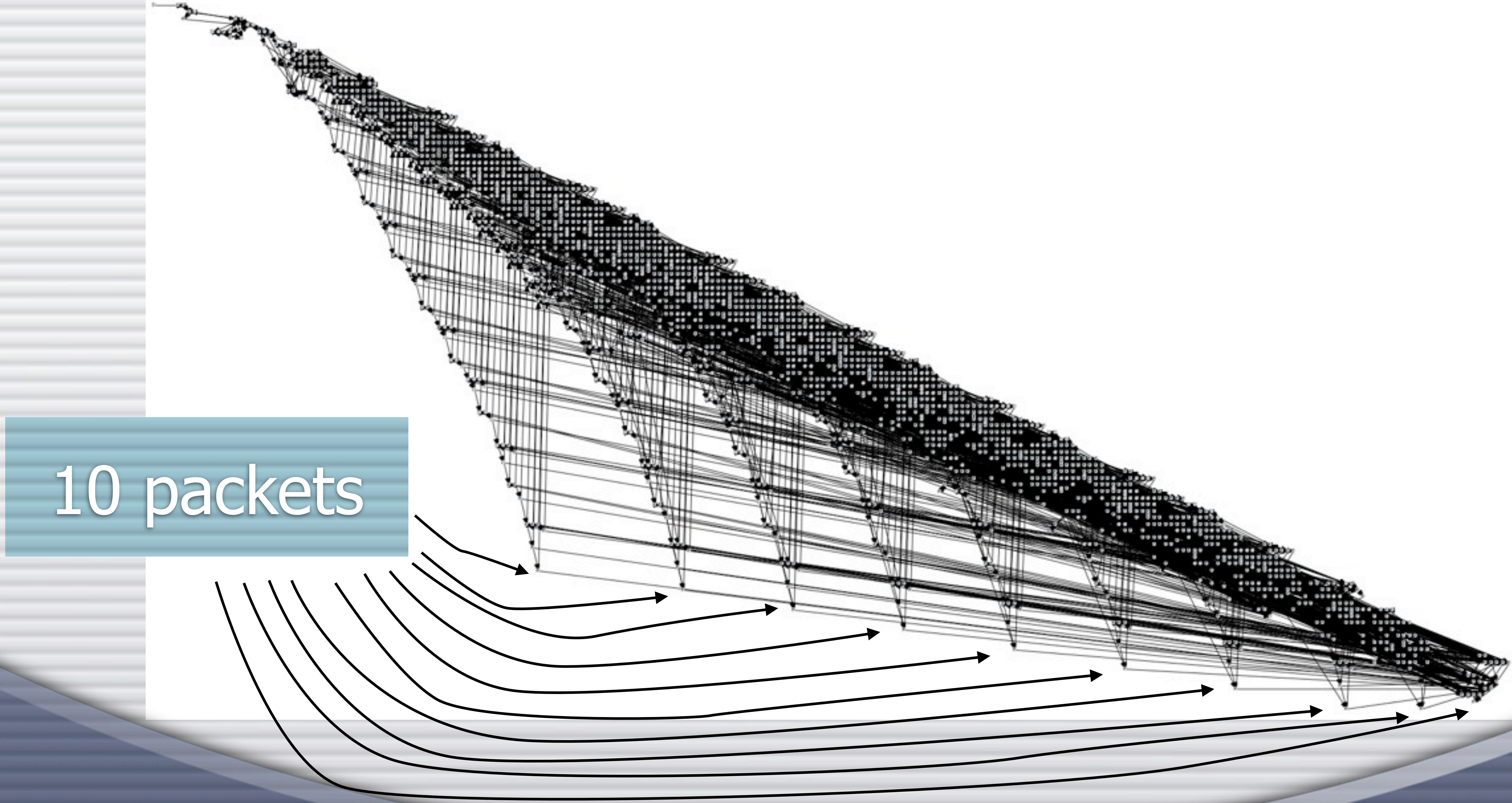


Example:
Simple Protocol

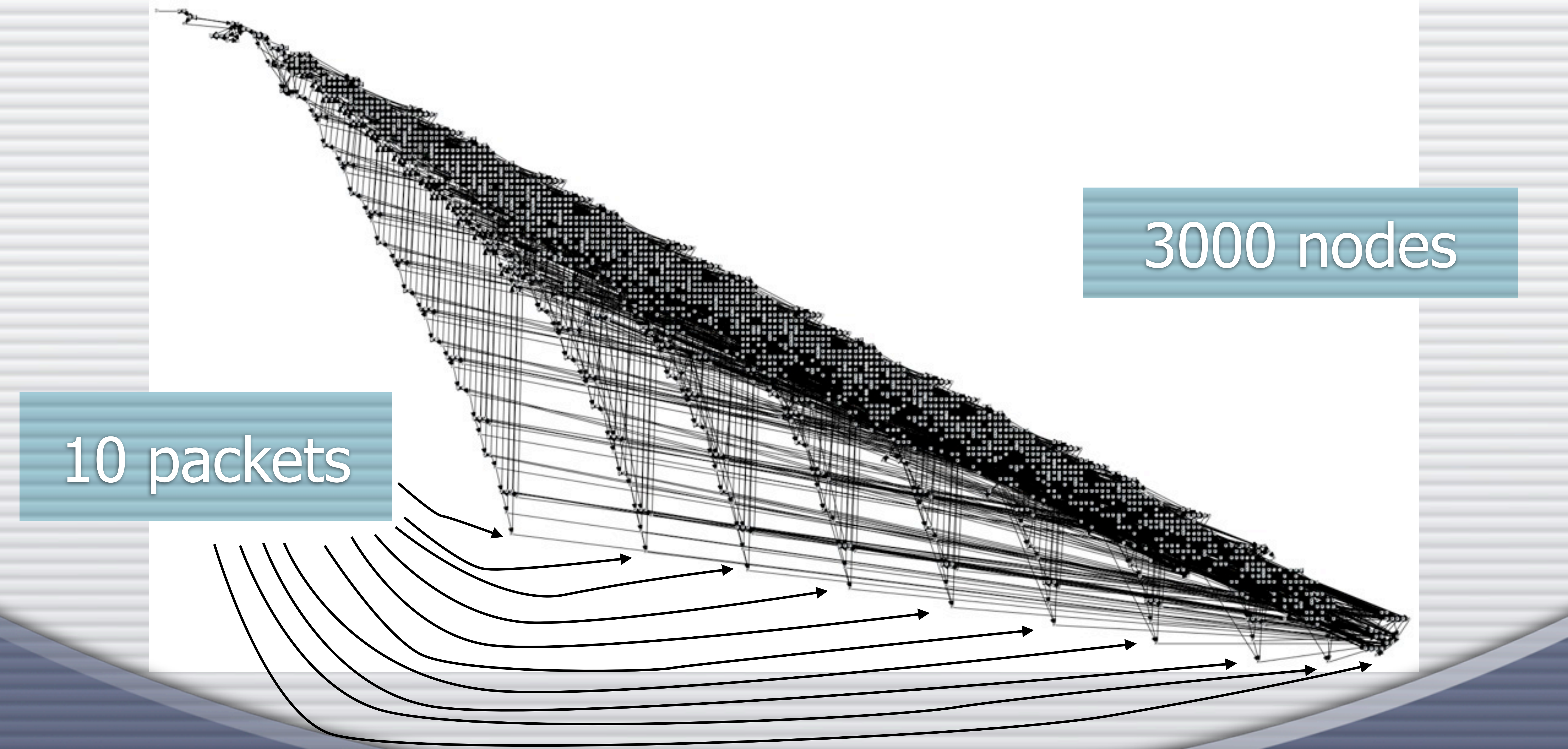


10 packets

Example: Simple Protocol



Example: Simple Protocol



Example: Simple Protocol

Demo:
Error Traces

- Displaying error trace
  - Displaying multiple error traces in a single window

Linear Temporal Logic

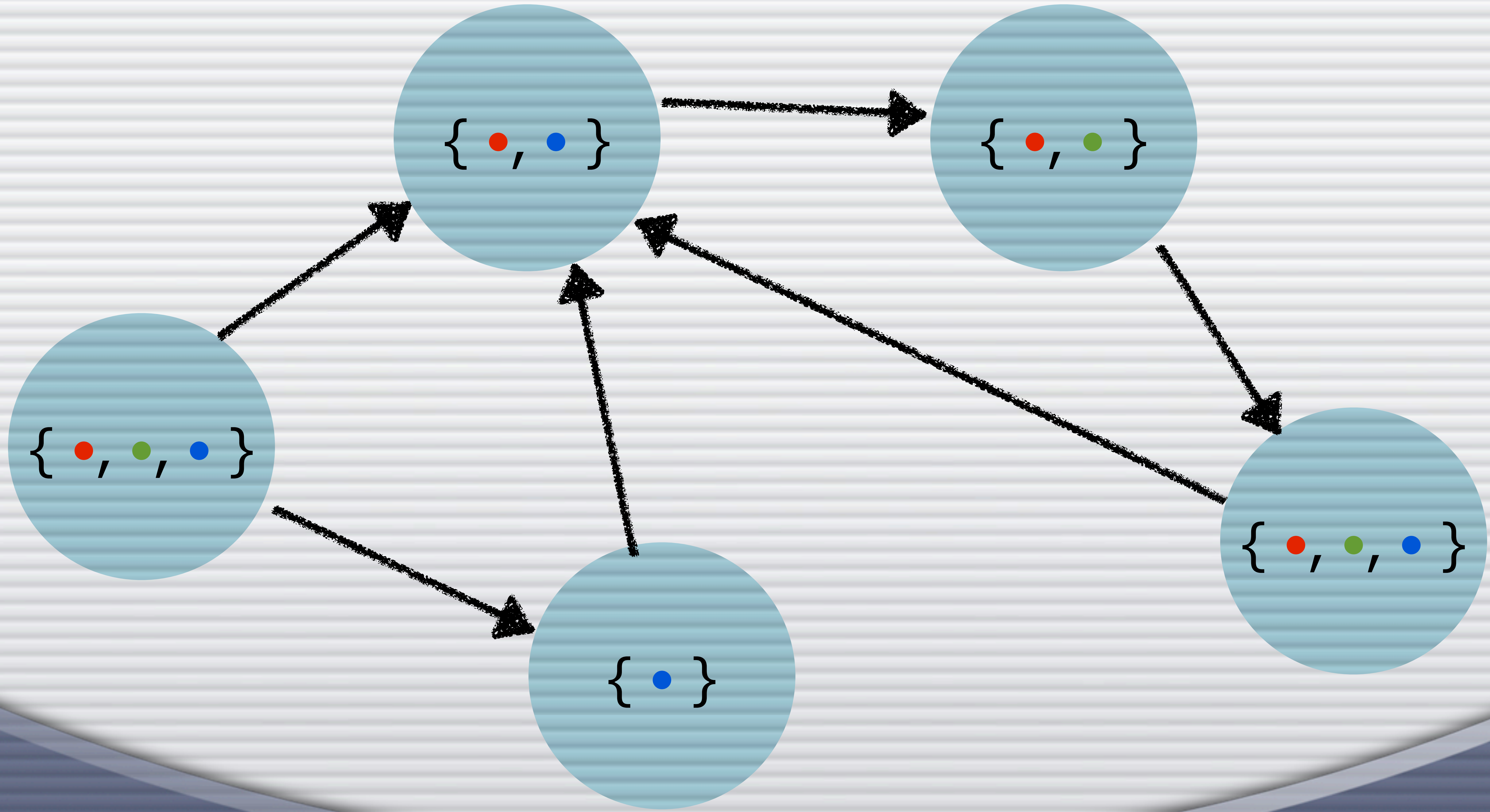
- Until now we have only dealt with safety properties (i.e., what happens in one state)
- Temporal logics allow us to talk about several states

Propositional Logic

- Atomic Propositions
  - AP = { p, q, r, … }
- Syntax
  - $\varphi ::= p \mid \neg\varphi \mid \varphi \to \psi$
- We call all such formulas Prop
- A formula φ holds for a system if it holds in all reachable states

We use SML functions for atomic propositions

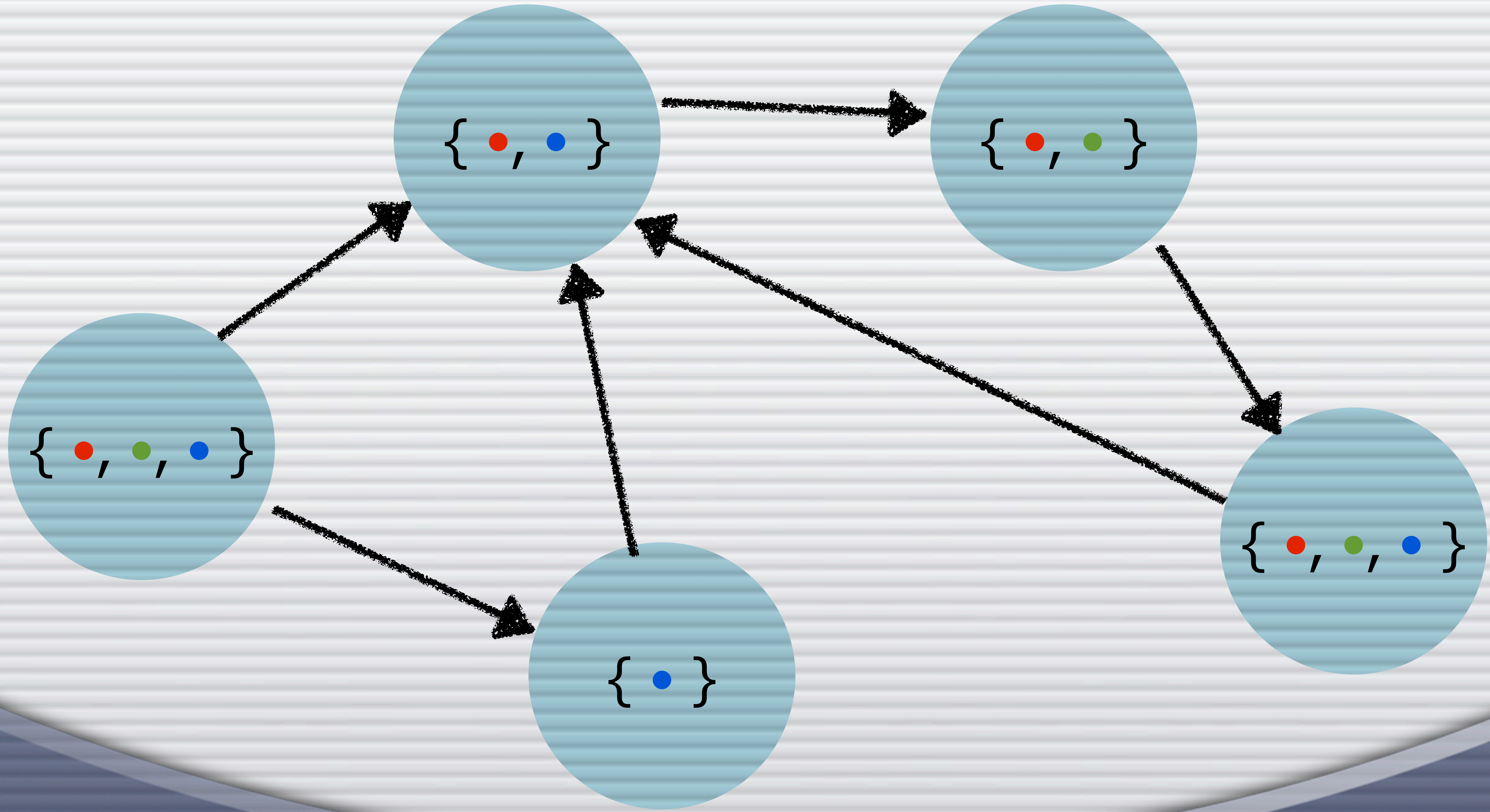
We have not really used connectors until now



Example

AP = { •, •, • }

APs holding in a state are shown inside the state

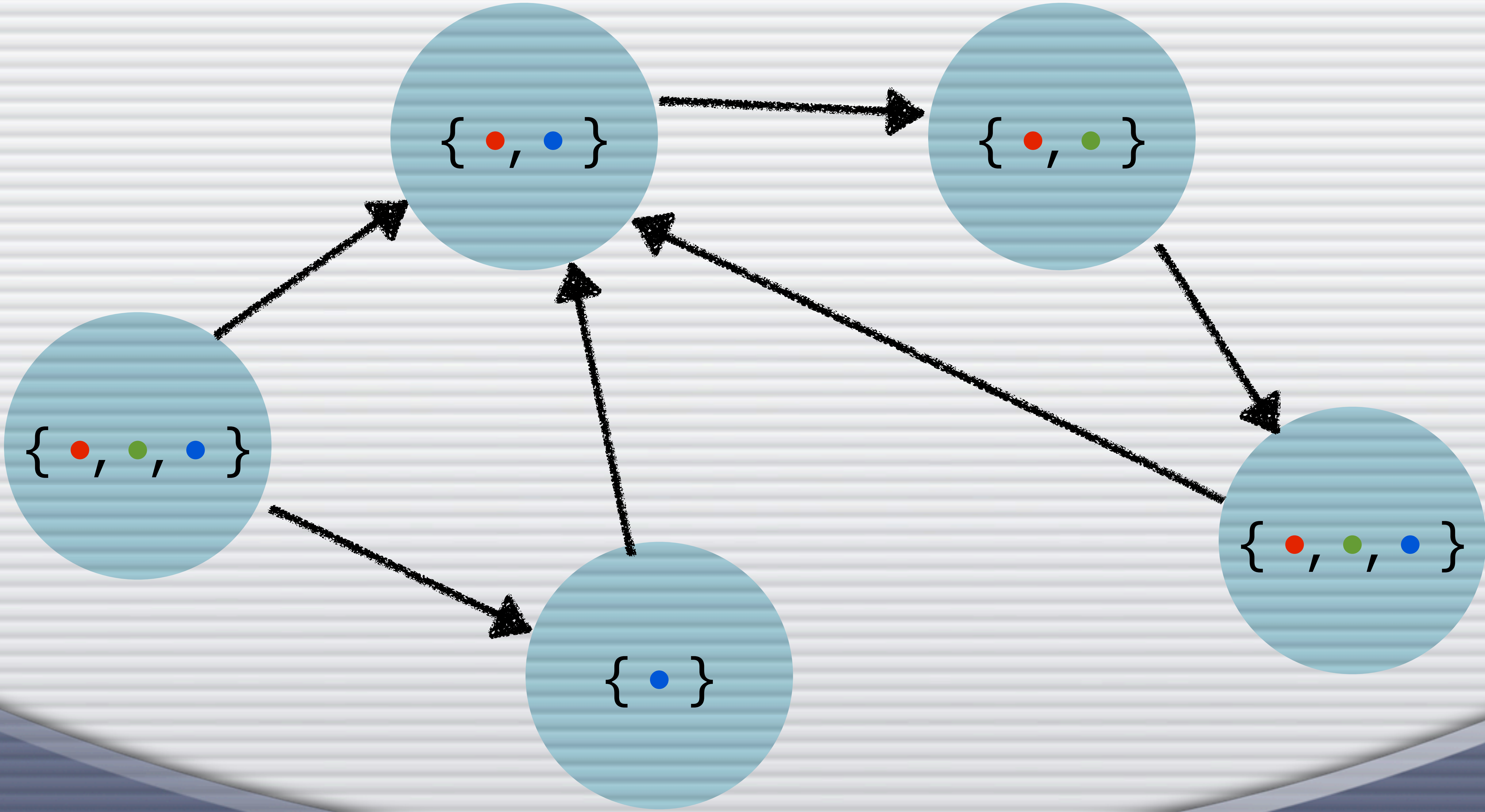


Example

AP = { •, •, • }

φ = • ∨ •

APs holding in a state are shown inside the state

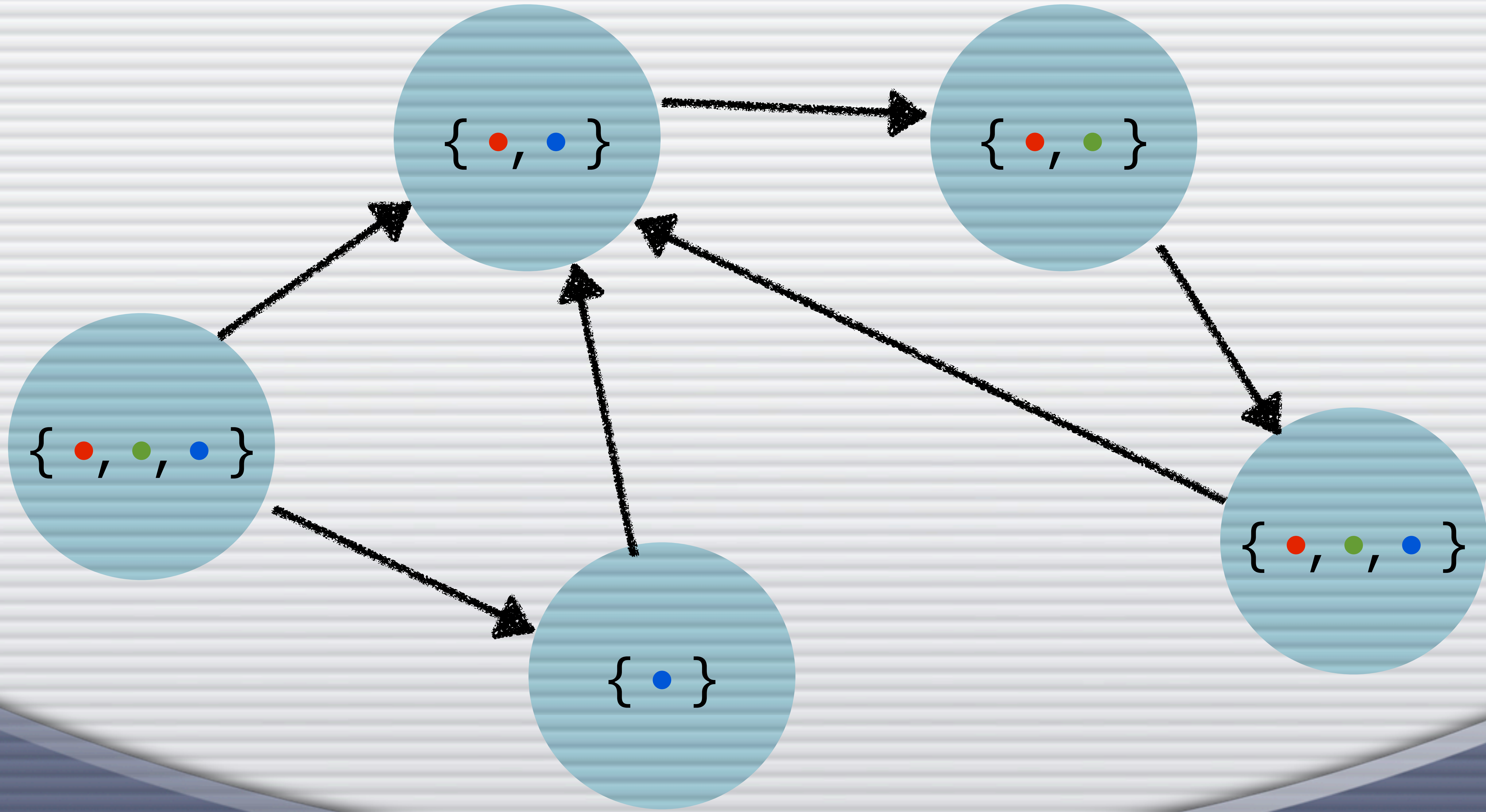


Example

AP = { •, •, • }

φ = • ∨ •

APs holding in a state are shown inside the state



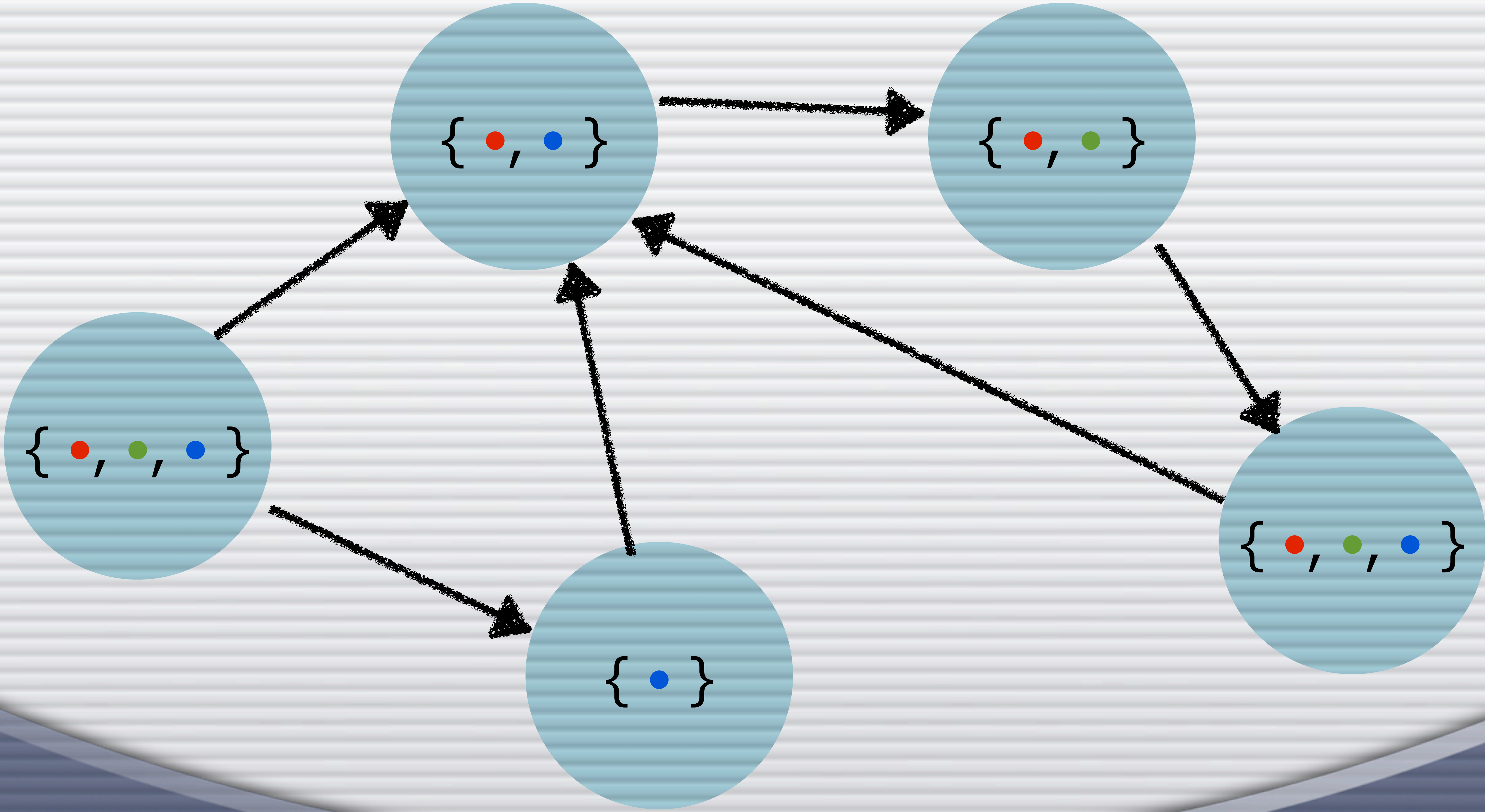
Example

AP = { •, •, • }

APs holding in a state are shown inside the state

φ = • ∨ •

φ′ = •



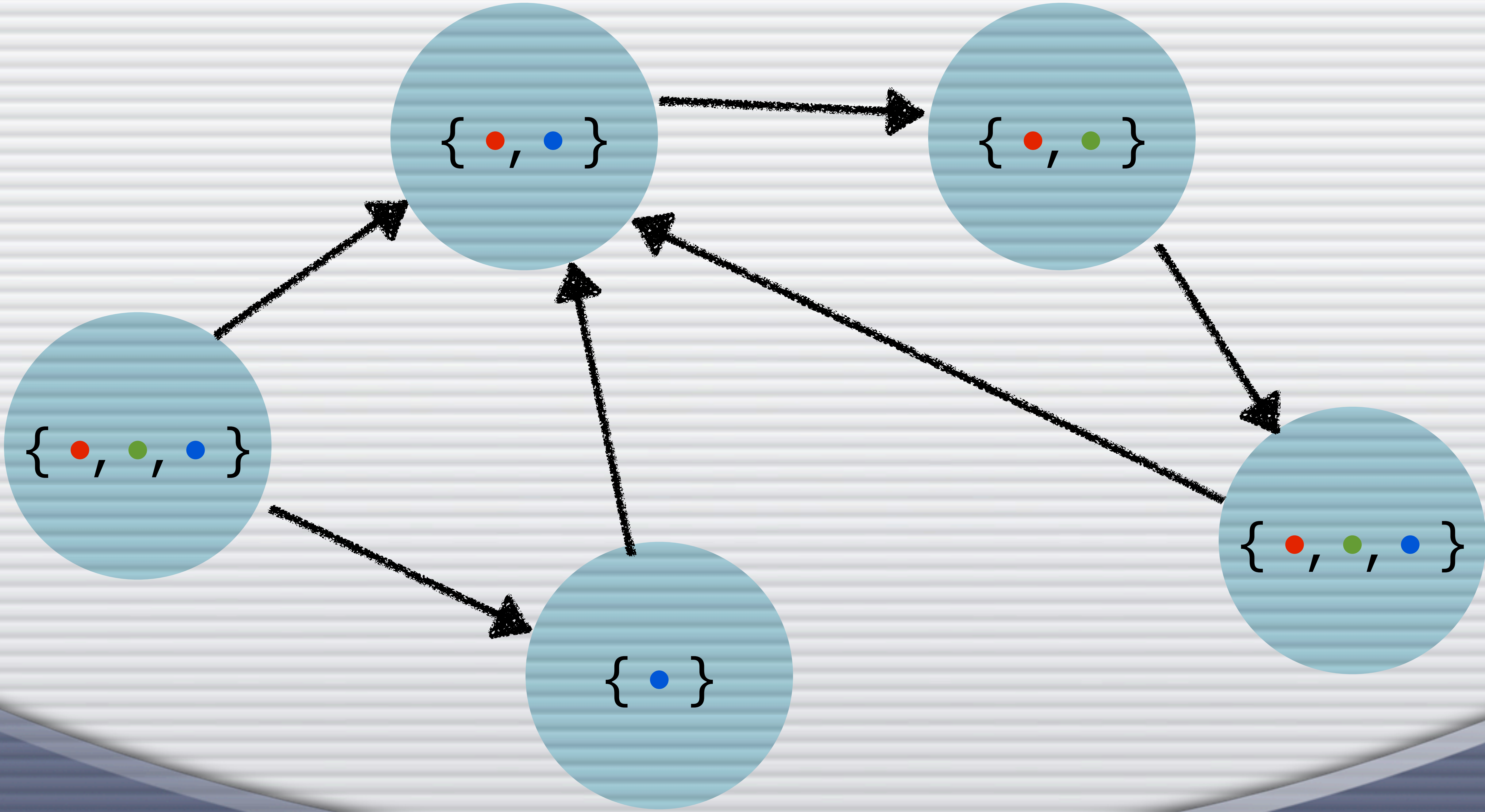
Example

AP = { •, •, • }

APs holding in a state are shown inside the state

$\varphi = \bullet \lor \bullet$

$\varphi' = \bullet$

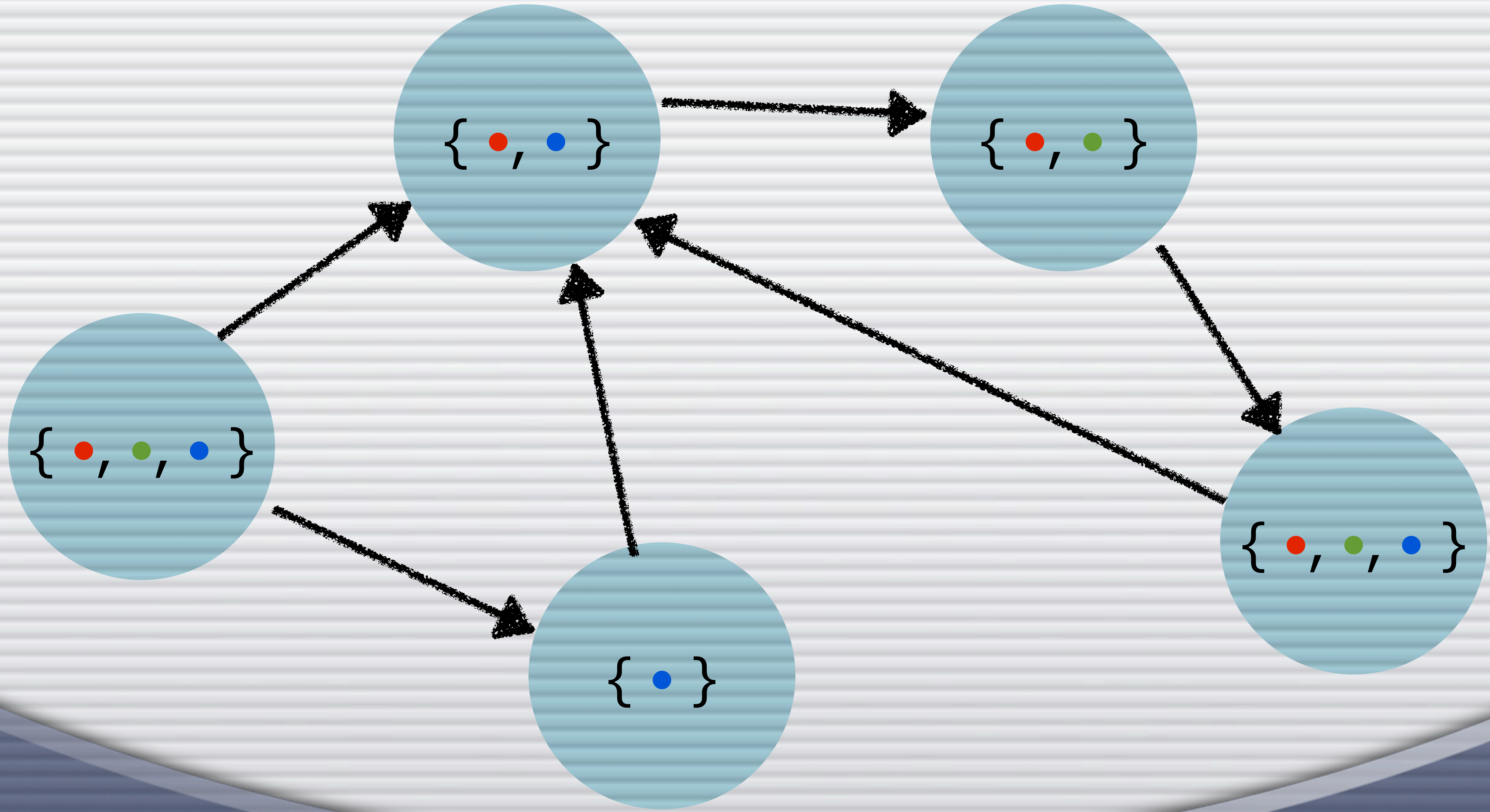


φ' = • does not hold

Example

AP = { •, •, • }

APs holding in a state are shown inside the state



Exam

AP = { •, •, • }

APs holding in a state are shown inside the state

φ′ = • does not hold

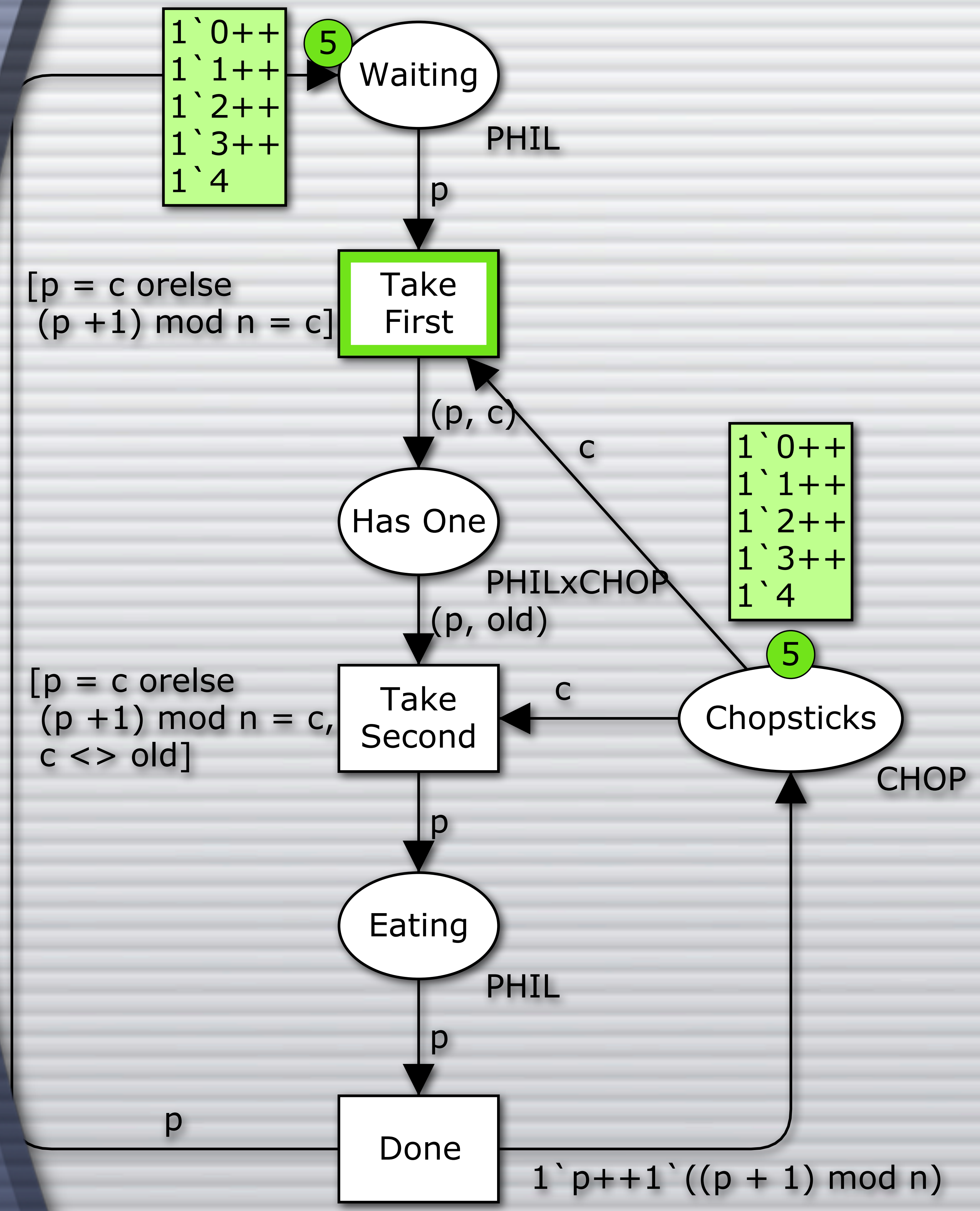
…but after executing "some" transitions, it does…

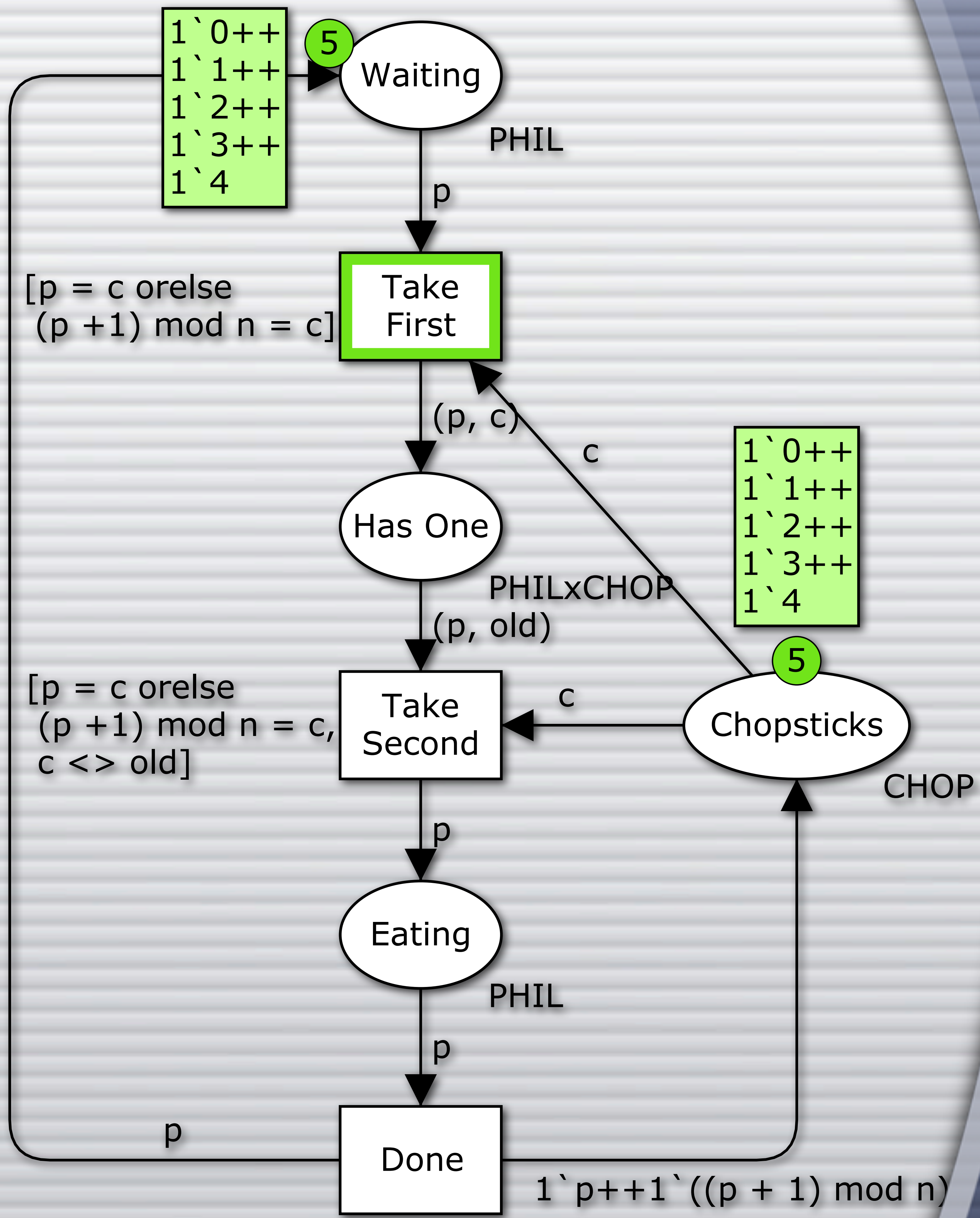
Example: Dining Philosophers

p = philisopher 1 eats

Philosopher 1 always eats: $\varphi = p$

Philosopher 1 never eats: $\varphi' = \neg p$



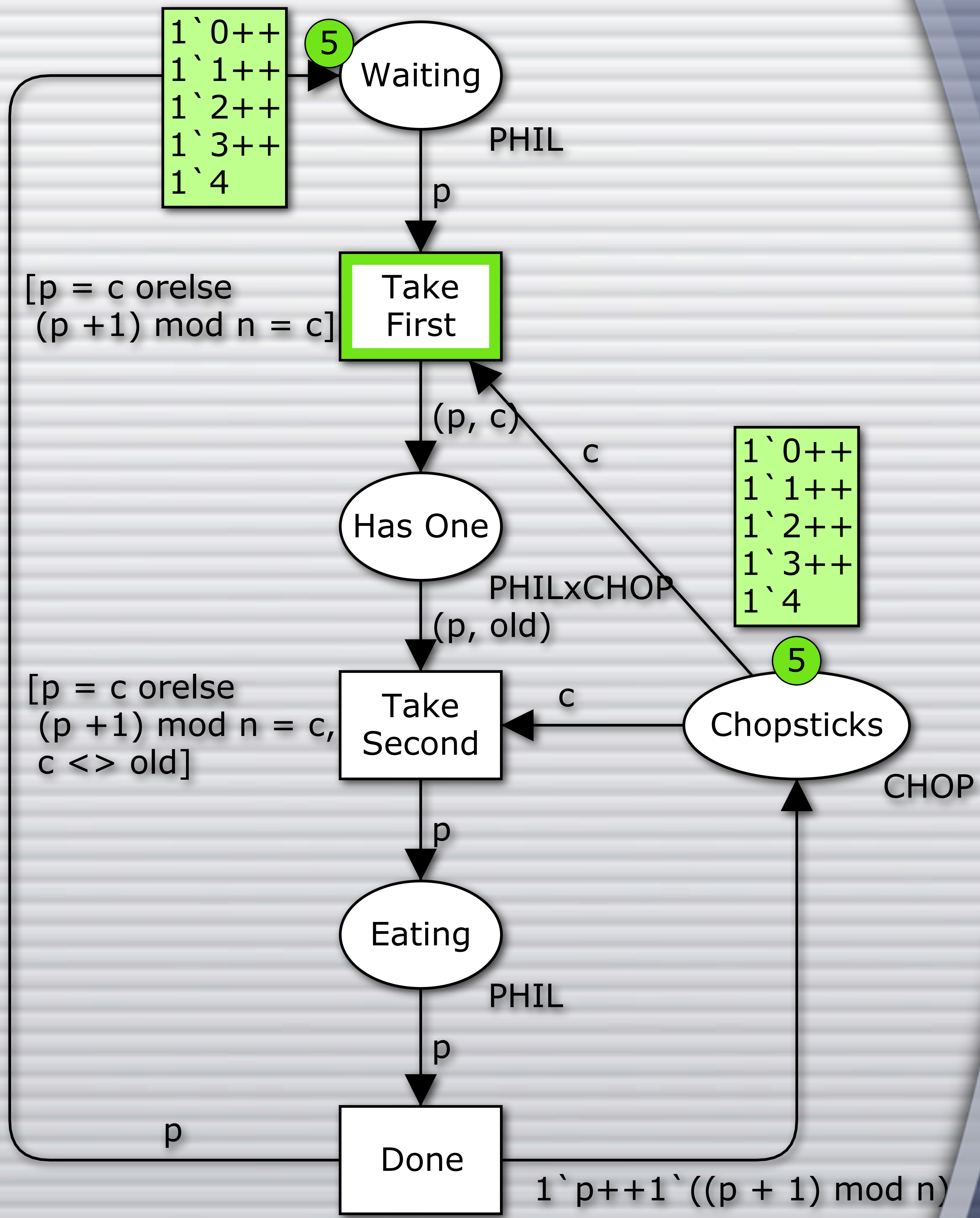


Example: Dining Philosophers

p = philisopher 1 eats

Philosopher 1 may eat?

Example: Dining Philosophers



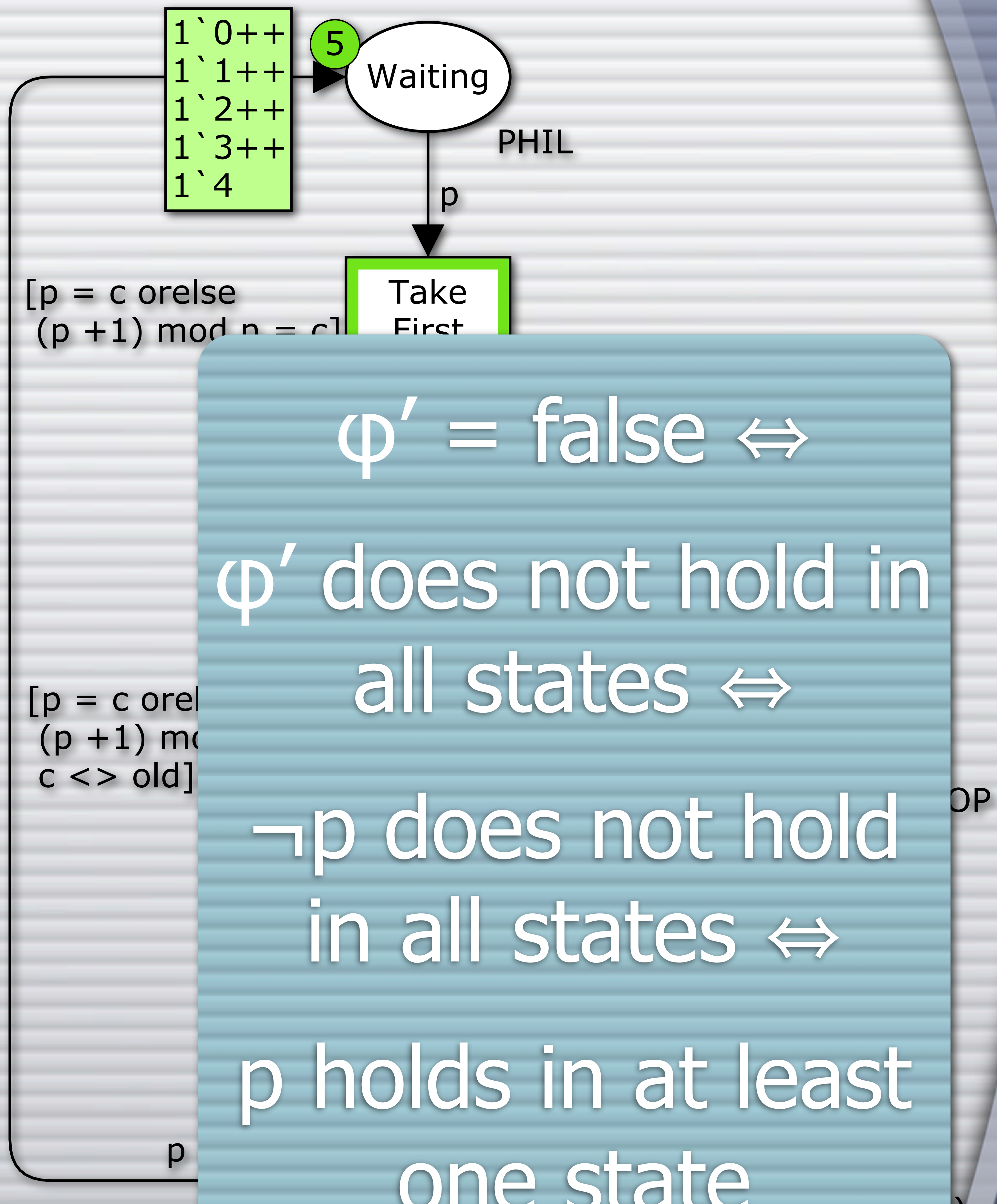
p = philisopher 1 eats

Philosopher 1 may eat?

…we **can** actually check this property:

- check $\varphi' = \neg p$
- answer is the opposite

Example: Dining Philosophers



$\varphi' = \text{false} \Leftrightarrow$

φ' does not hold in all states $\Leftrightarrow$

$\neg p$ does not hold in all states $\Leftrightarrow$

p holds in at least one state

p = philisopher 1 eats

Philosopher 1 may eat?

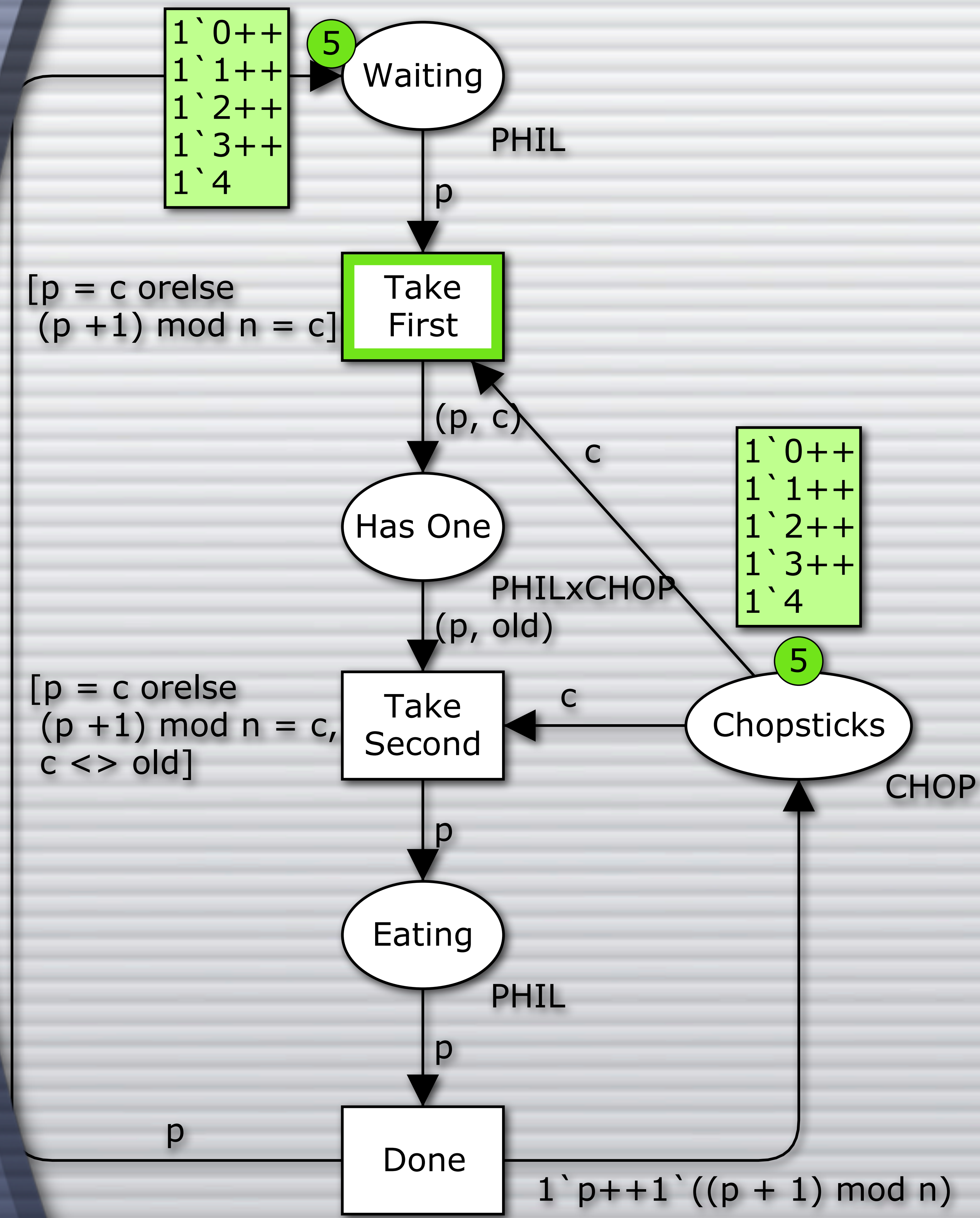
…we **can** actually check this property:

- check $\varphi' = \neg p$
- answer is the opposite

Example: Dining Philosophers

p = philisopher 1 eats

Philosopher 1 will always eat at some point

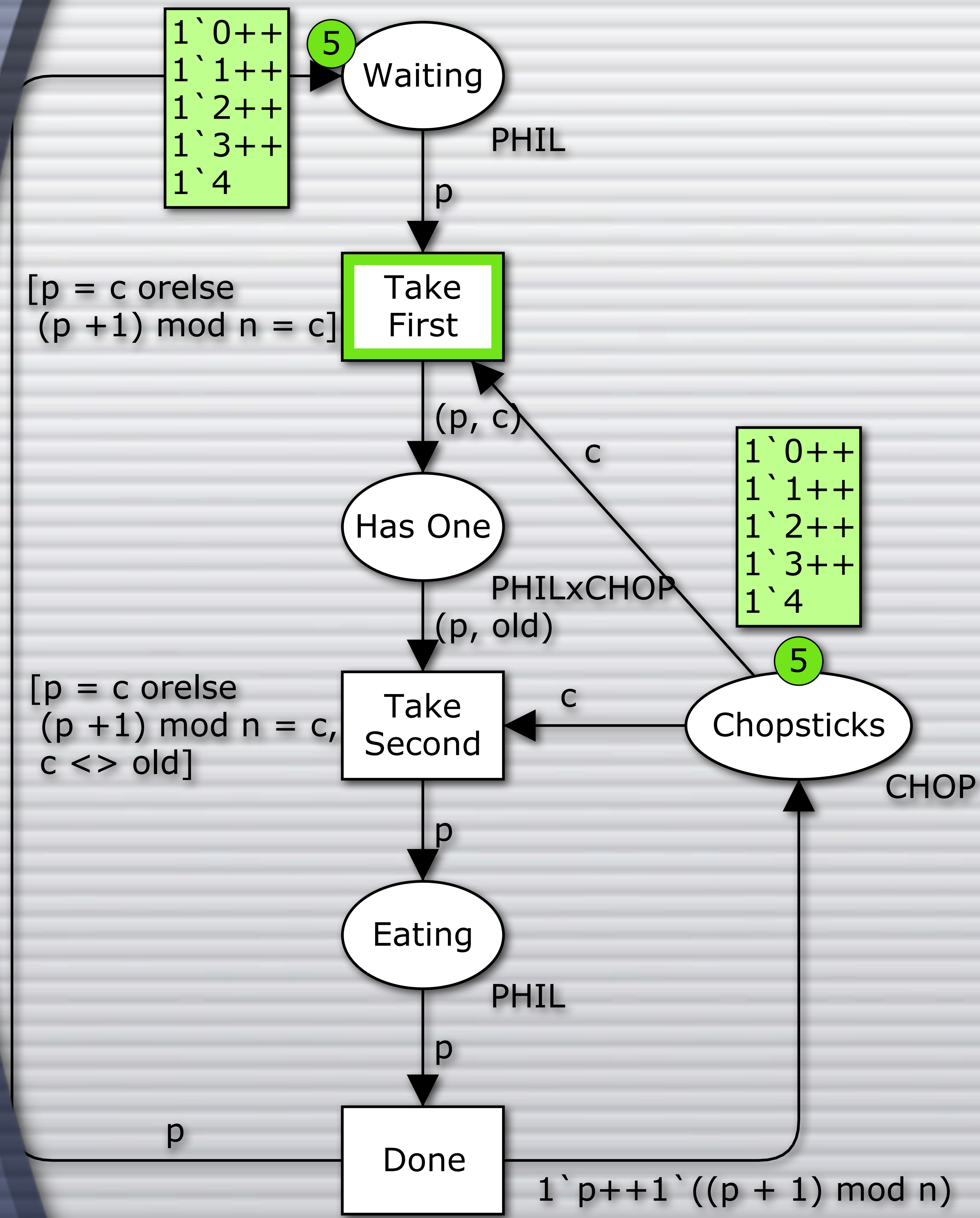


Example: Dining Philosophers

p = philisopher 1 eats

Philosopher 1 will always eat at some point

...we cannot check this (unless "eat at some point" is an atomic proposition)



PLTL Syntax

- Atomic Propositions
  - $AP = \{ p, q, r, \dots \}$
- Syntax
  - $\varphi ::= p \mid \neg\varphi \mid \varphi \rightarrow \psi \mid X\,\varphi \mid \varphi\, U\, \psi$
- Add some syntactical sugar
  - $F\varphi \equiv \text{true}\ U\ \varphi$ (also written $\Diamond\varphi$)
  - $G\varphi \equiv \neg F \neg \varphi$ (also written $\Box\varphi$)

PLTL Syntax

- Atomic Propositions
  - AP = { p, q
- Syntax
  - $\varphi ::= p \mid \neg\varphi \mid \varphi \rightarrow \psi \mid X\,\varphi \mid \varphi\, U\, \psi$
- Add some syntactical sugar
  - $F\varphi \equiv true\ U\ \varphi$ (also written $\Diamond\varphi$)
  - $G\varphi \equiv \neg F \neg\varphi$ (also written $\Box\varphi$)

In the next state φ holds

PLTL Syntax

- Atomic Propositions
  - AP = { p,
- Syntax
  - $\varphi ::= p \mid \neg\varphi \mid \varphi \rightarrow \psi \mid X\, \varphi \mid \varphi\, U\, \psi$

In the next state φ holds

φ holds until ψ holds (and ψ holds eventually)

- Add some syntactical sugar
  - $F\varphi \equiv true\ U\ \varphi$ (also written $\Diamond\varphi$)
  - $G\varphi \equiv \neg F \neg\varphi$ (also written $\Box\varphi$)

PLTL Syntax

- Atomic Propositions

  { p, q

  $p \mid \neg\varphi \mid \varphi \rightarrow \psi \mid X\,\varphi \mid \varphi\, U\, \psi$

φ holds at some point (future, eventually, possibly)

In the next state φ holds

φ holds until ψ holds (and ψ holds eventually)

- Add some syntactical sugar
  - $F\varphi \equiv \text{true}\ U\ \varphi$ (also written $\Diamond\varphi$)
  - $G\varphi \equiv \neg F \neg\varphi$ (also written $\Box\varphi$)

PLTL Syntax

- Atomic Propositions
  - { p, q

φ holds at some point (future, eventually, possibly)

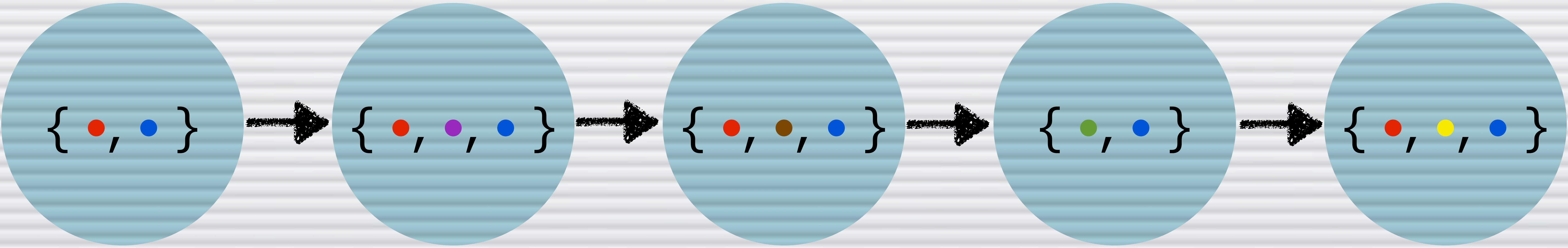
In the next state φ holds

φ holds until ψ holds (and ψ holds eventually)

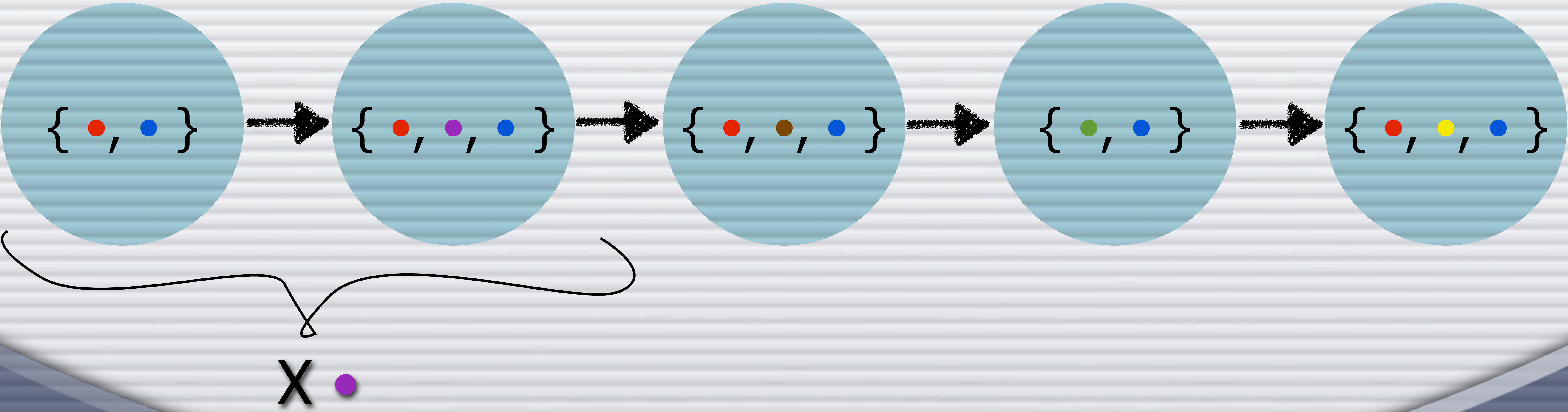
$p \mid \neg\varphi \mid \varphi \rightarrow \psi \mid X\,\varphi \mid \varphi\, U\, \psi$

- Add some syntactical sugar
  - $F\varphi \equiv true\ U\ \varphi$ (also writt
  - $G\varphi \equiv \neg F \neg \varphi$ (also written

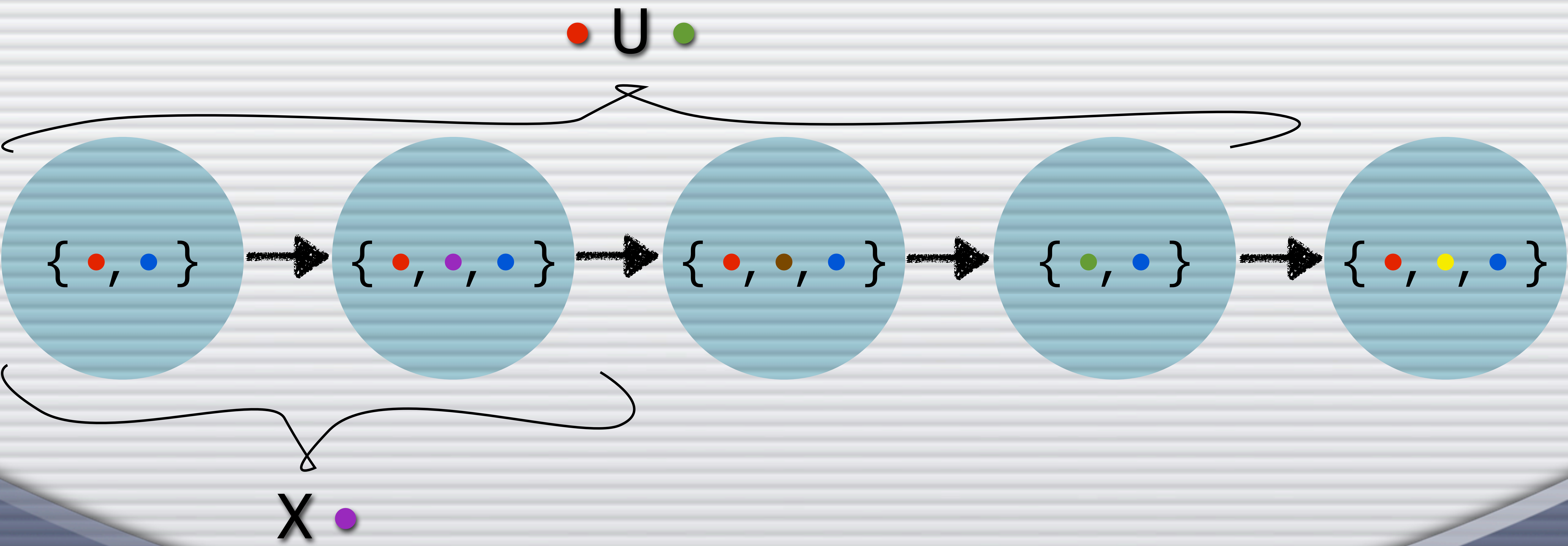
φ holds in all states (everywhere, globally, necessarily)



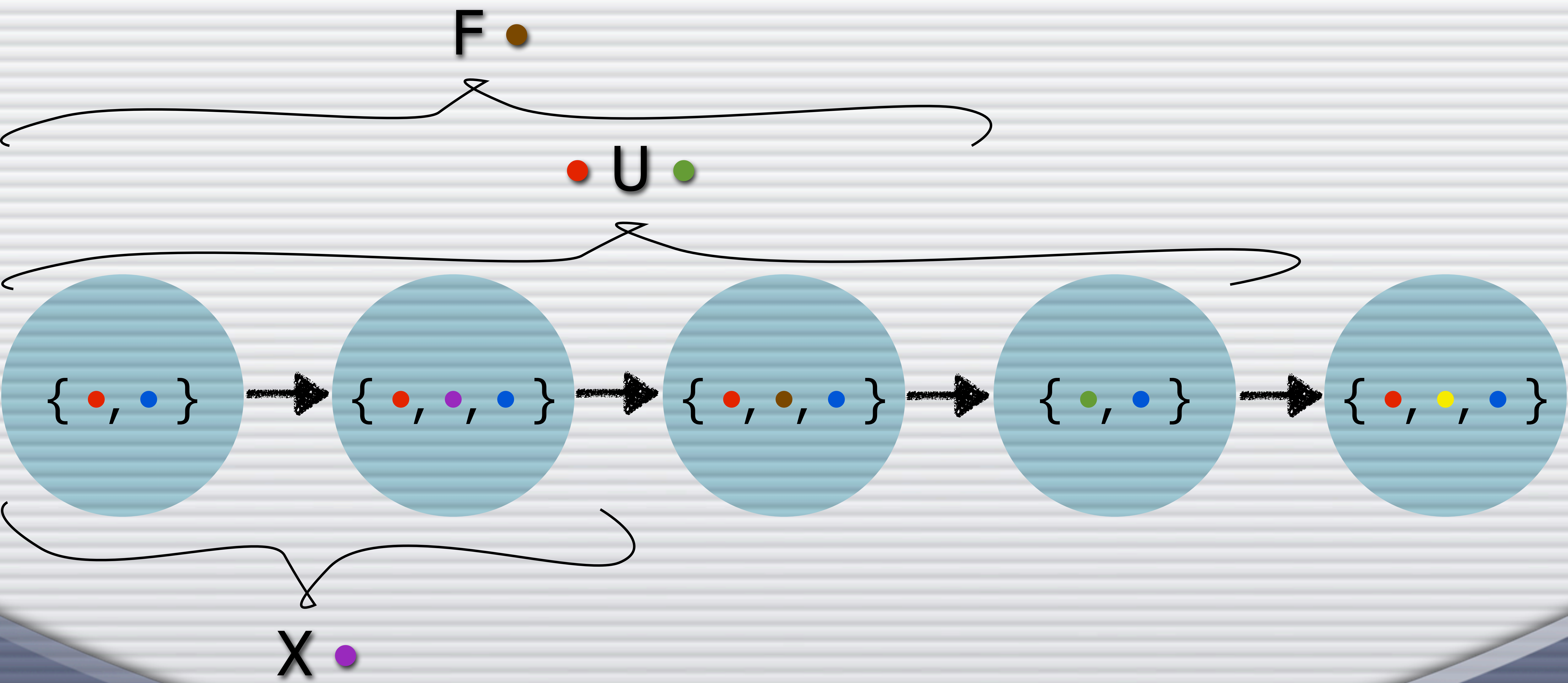
Example



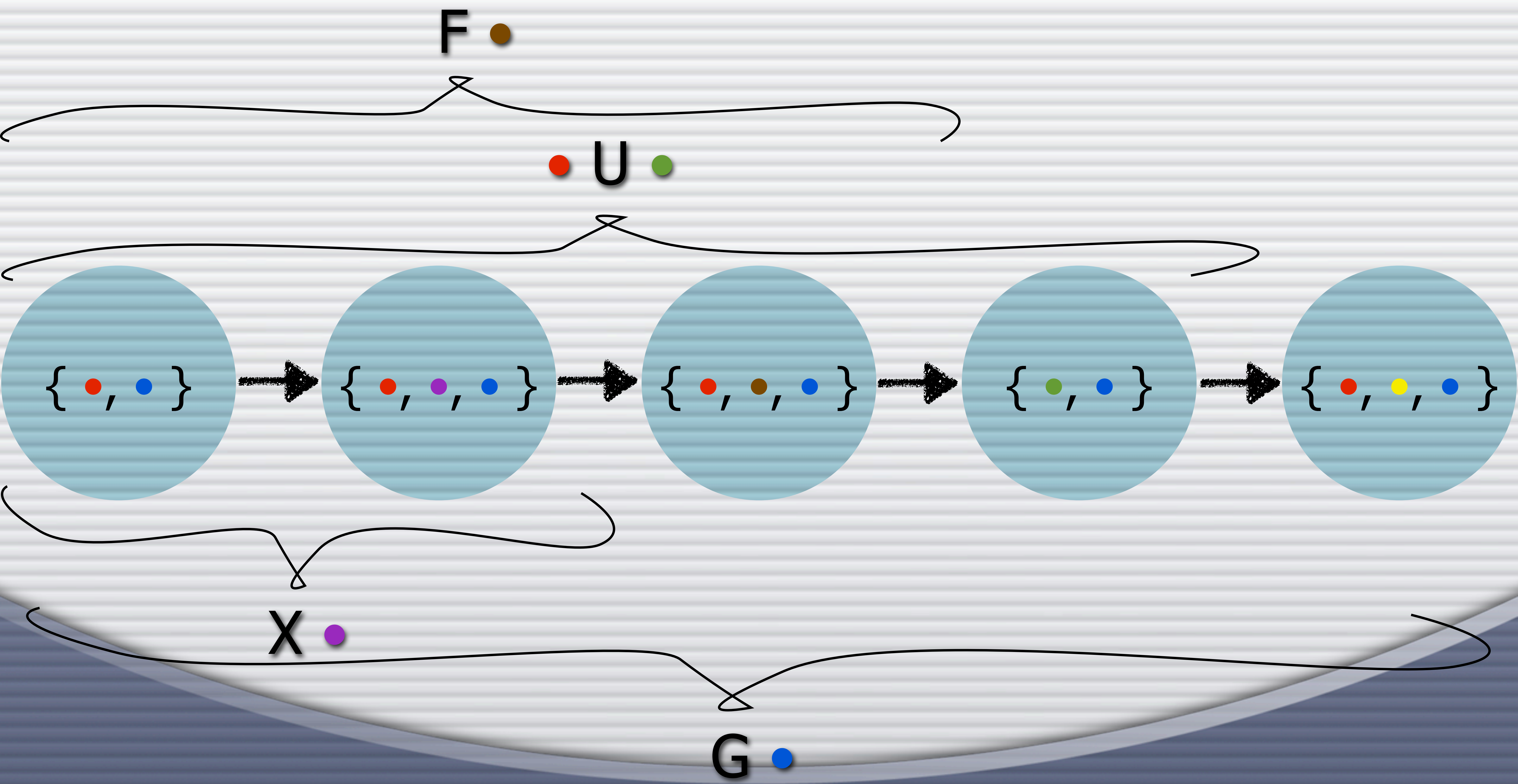
Example



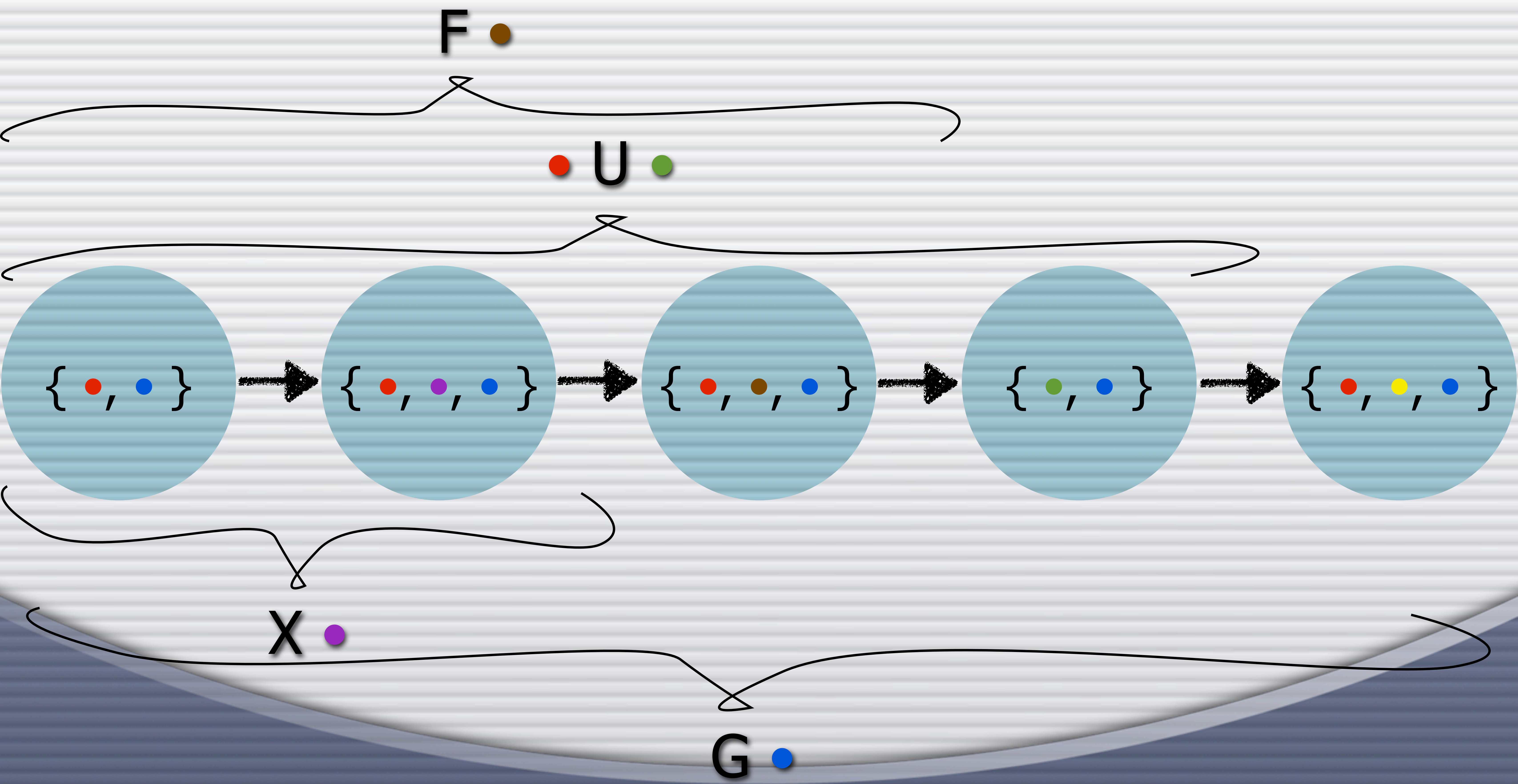
Example



Example

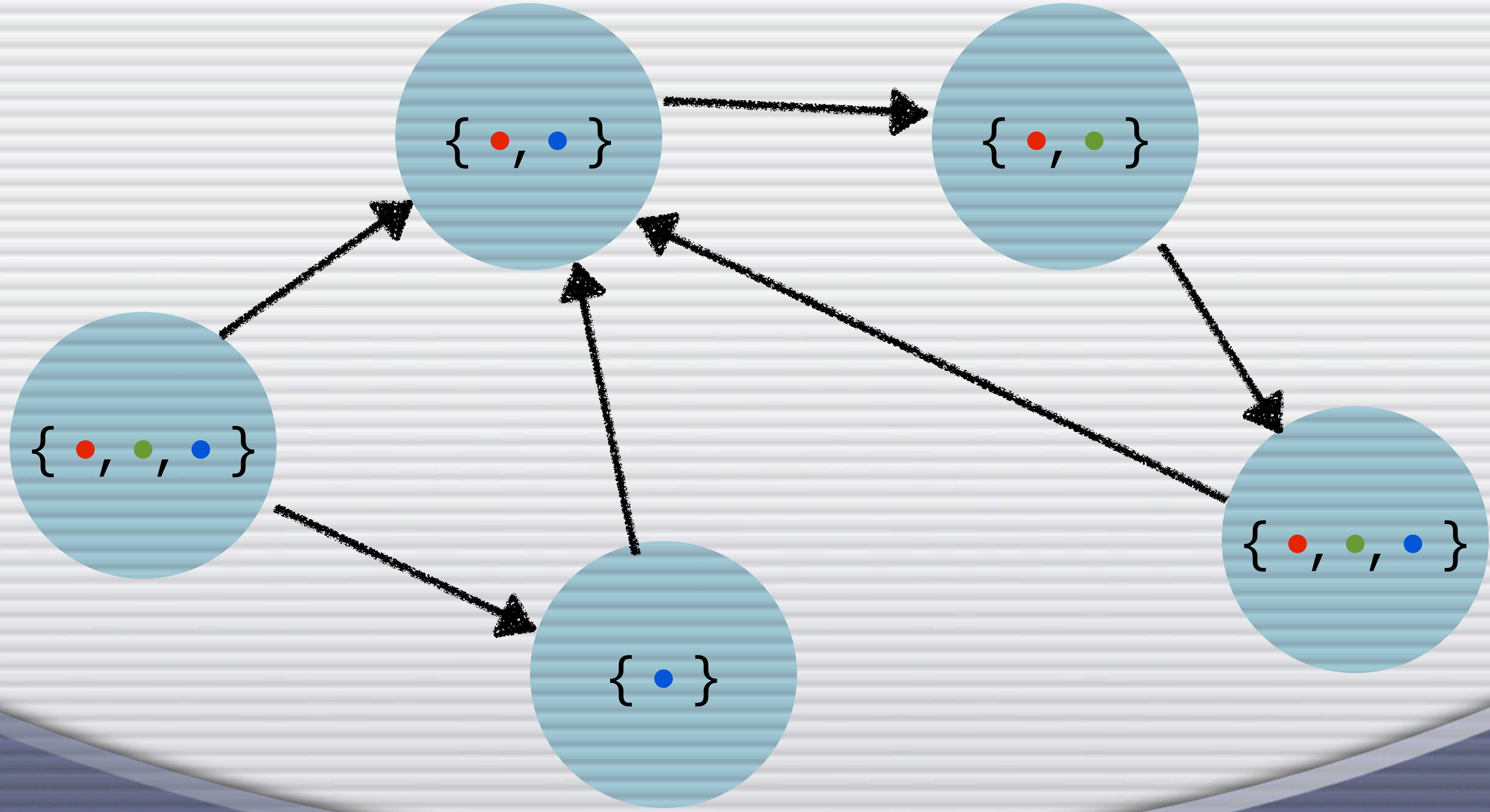


Example

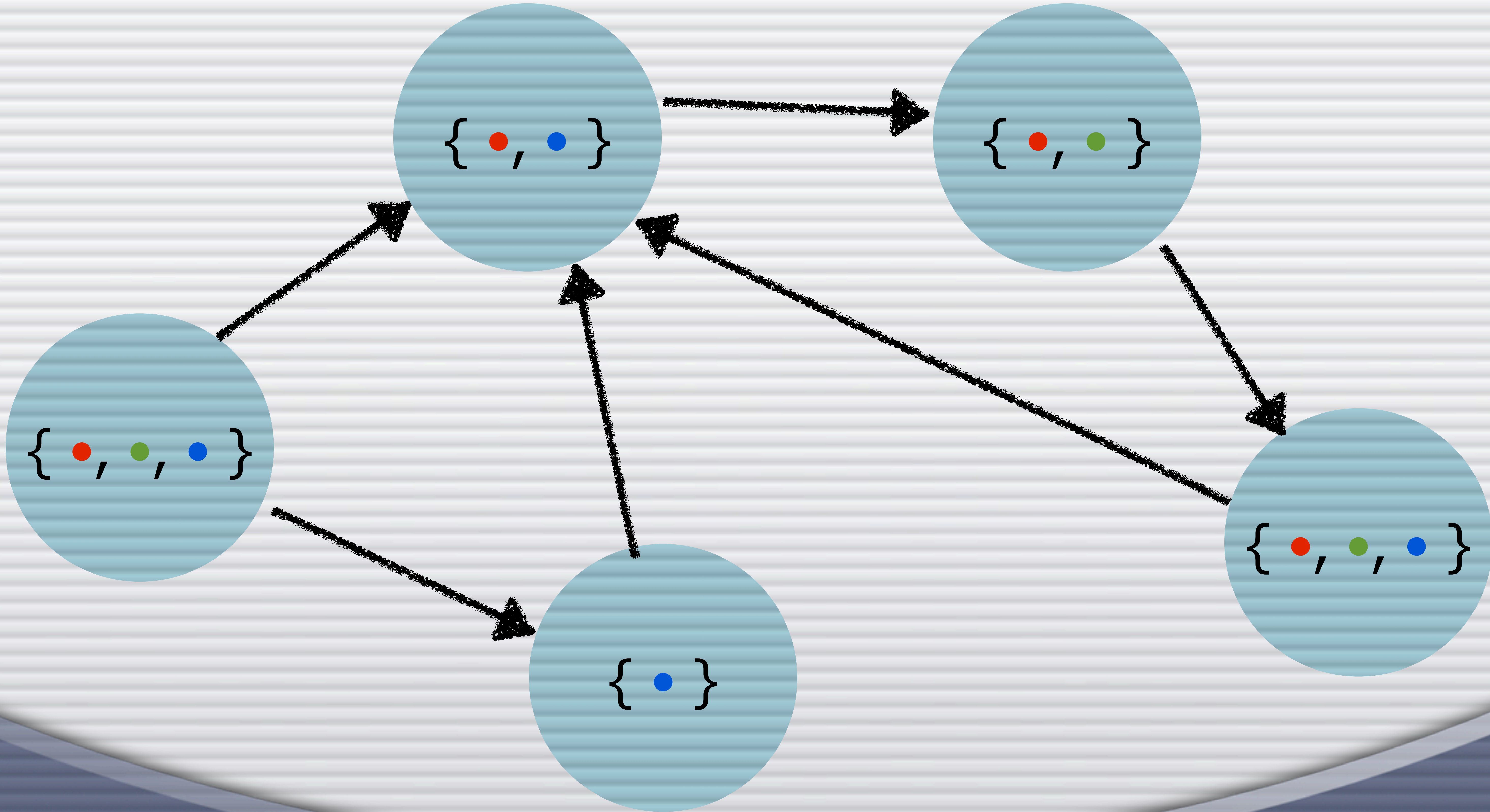


Example

For a property to hold for a state space, it must hold along all (infinite) paths



Example



Example

F ●

FG ●

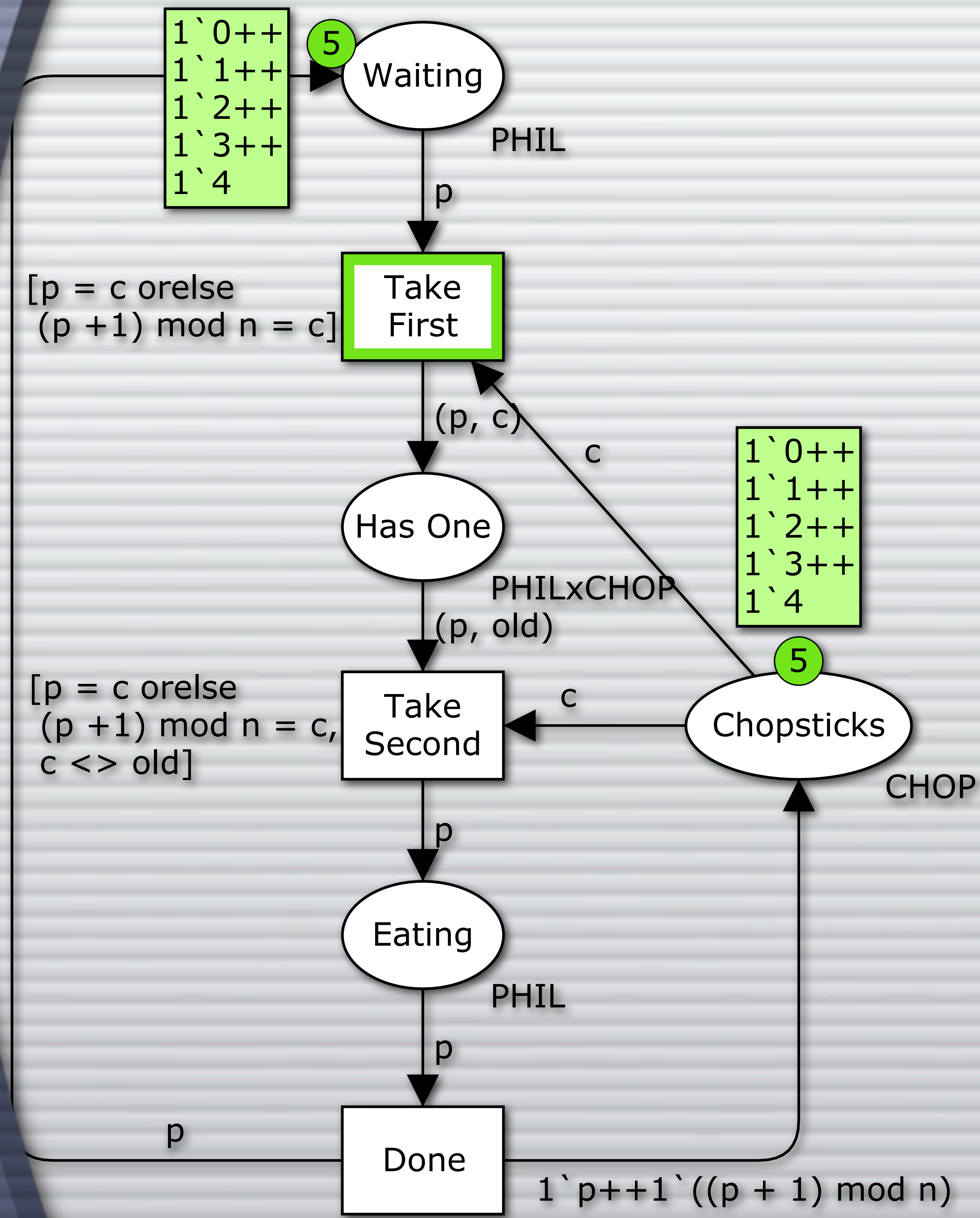
GF ●

● → X ●

Example: Dining Philosophers

p = philisopher 1 eats

Philosopher 1 will always eat at some point

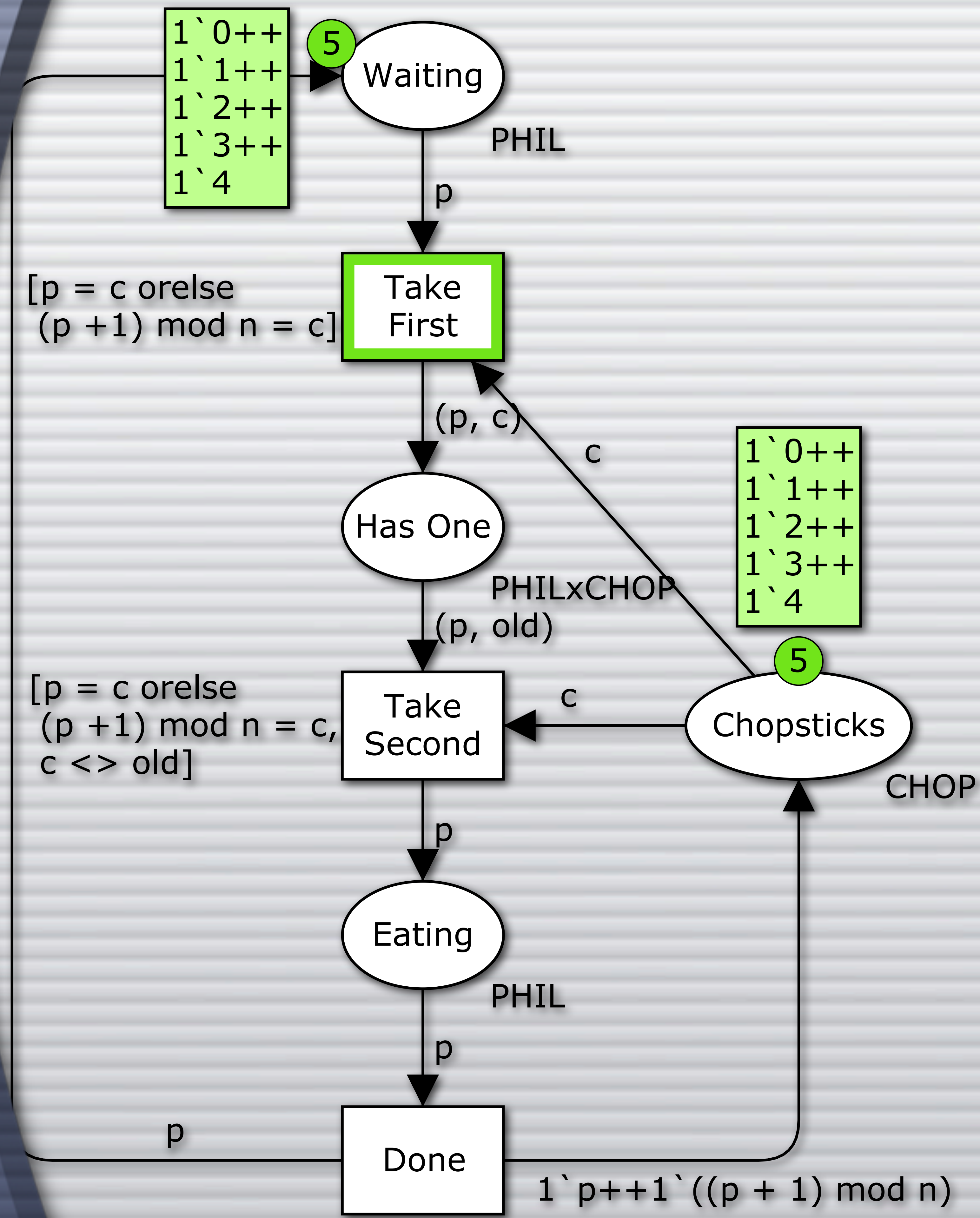


Example: Dining Philosophers

p = philisopher 1 eats

Philosopher 1 will always eat at some point

- φ″ = F p
- φ‴ = G F p



LTL Examples

- **Safety** (nothing bad happens): G ¬bad
- **Liveness** (something good happens): F good
- **Response** (requests are eventually serviced): G(request → F serviced)
- **Reactiveness** (infinite number of requests means an infinite number are serviced): GF sent → GF received

Checking LTL

- L(M) language of a model M (i.e., all possible executions of M)
- L(φ) language of a formula φ (i.e., all traces satisfying φ)
- We want to check that
  $L(M) \subseteq L(\varphi) \Leftrightarrow L(M) \cap L(\varphi)^c = \varnothing$

  $\Leftrightarrow L(M) \cap L(\neg\varphi) = \varnothing$

Checking LTL (2)

- We want to check that $L(M) \cap L(\neg\varphi) = \emptyset$
- We can construct a Büchi automaton $A_{\neg\varphi}$ such that $L(\neg\varphi) = L(A_{\neg\varphi})$
- The state space SS_M is essentially a Büchi automaton representing $L(M)$
- We thus check whether $L(SS_M \times A_{\neg\varphi}) = \emptyset$
  - The product is (essentially) equal to the product construction for finite automata

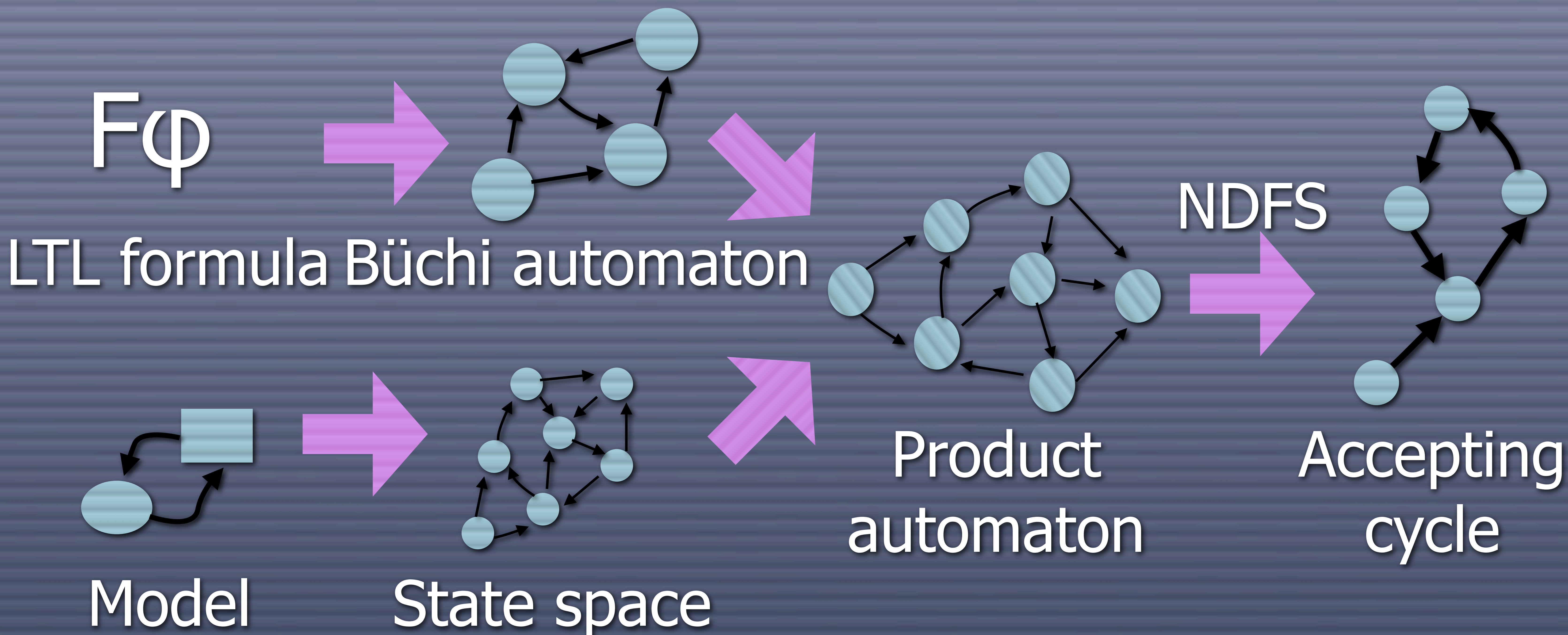
Checking LTL (3)

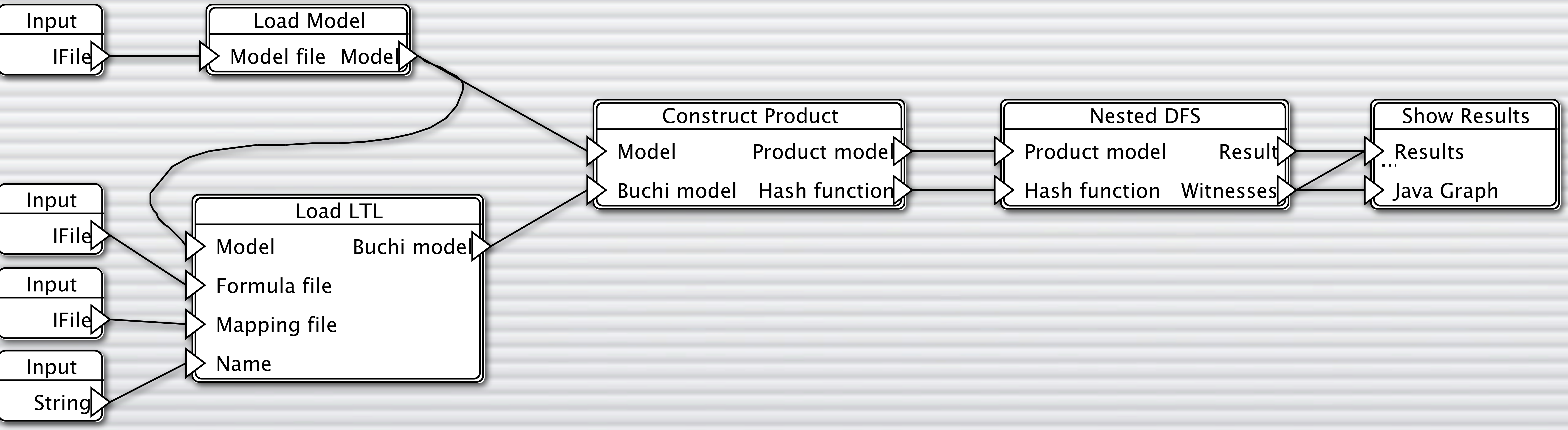
- A Büchi automaton is a finite automaton but in order for a word to be accepted, we must go thru an accept state infinitely often
- As it is finite, this means we must visit (at least) one accept state infinitely often
- This is only possible if we can find a loop containing the accept state from the initial state

Checking LTL (4)

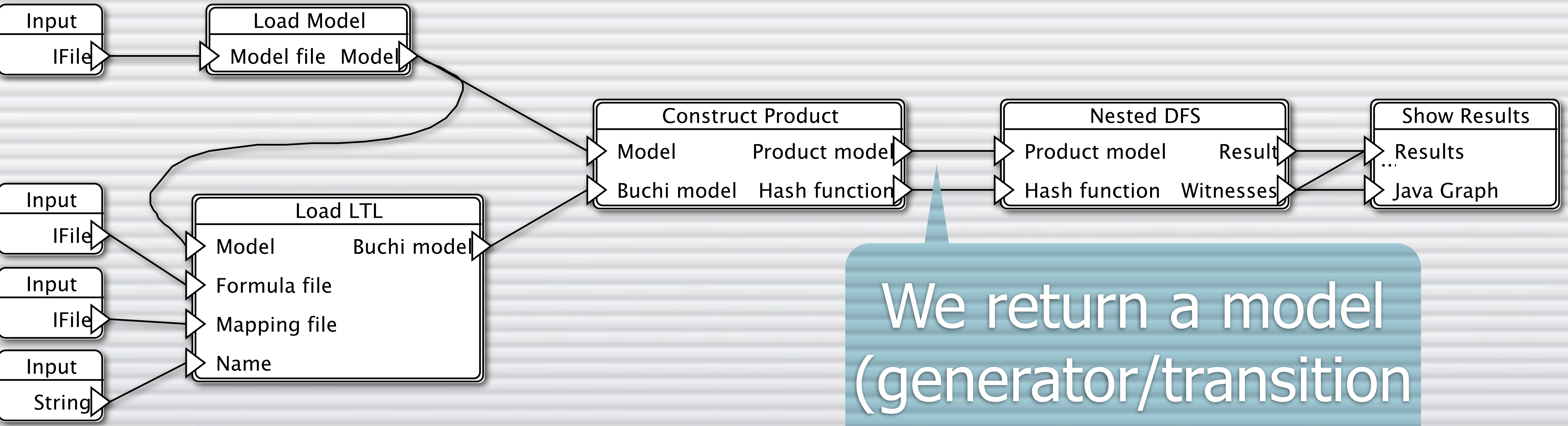
- We can find such “accepting” loops by nested depth-first search:
  - Do DFS from the initial state until an accepting state
  - Do DFS from the accepting state to see if we can reach the state again

Checking LTL (5)



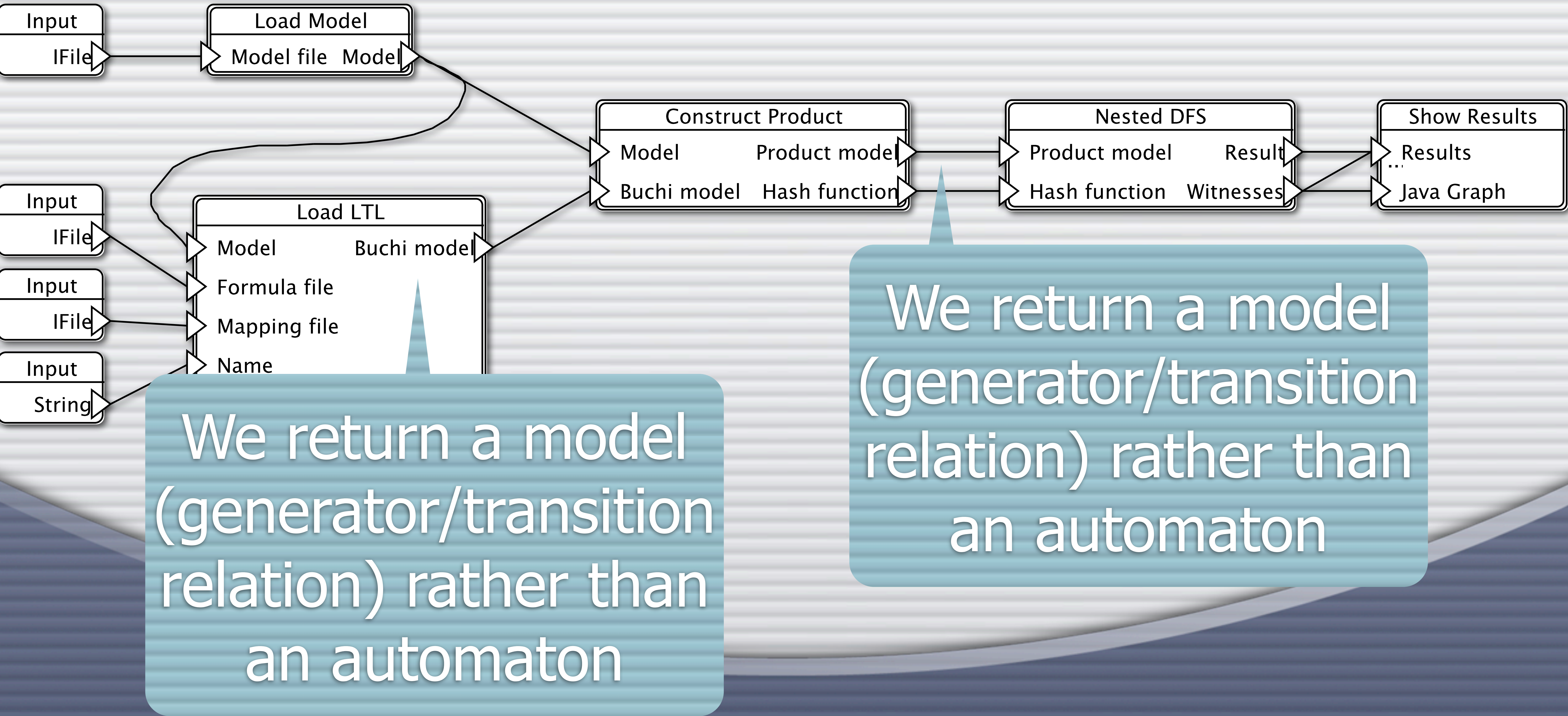


LTL Checking in ASAP



We return a model (generator/transition relation) rather than an automaton

LTL Checking in ASAP



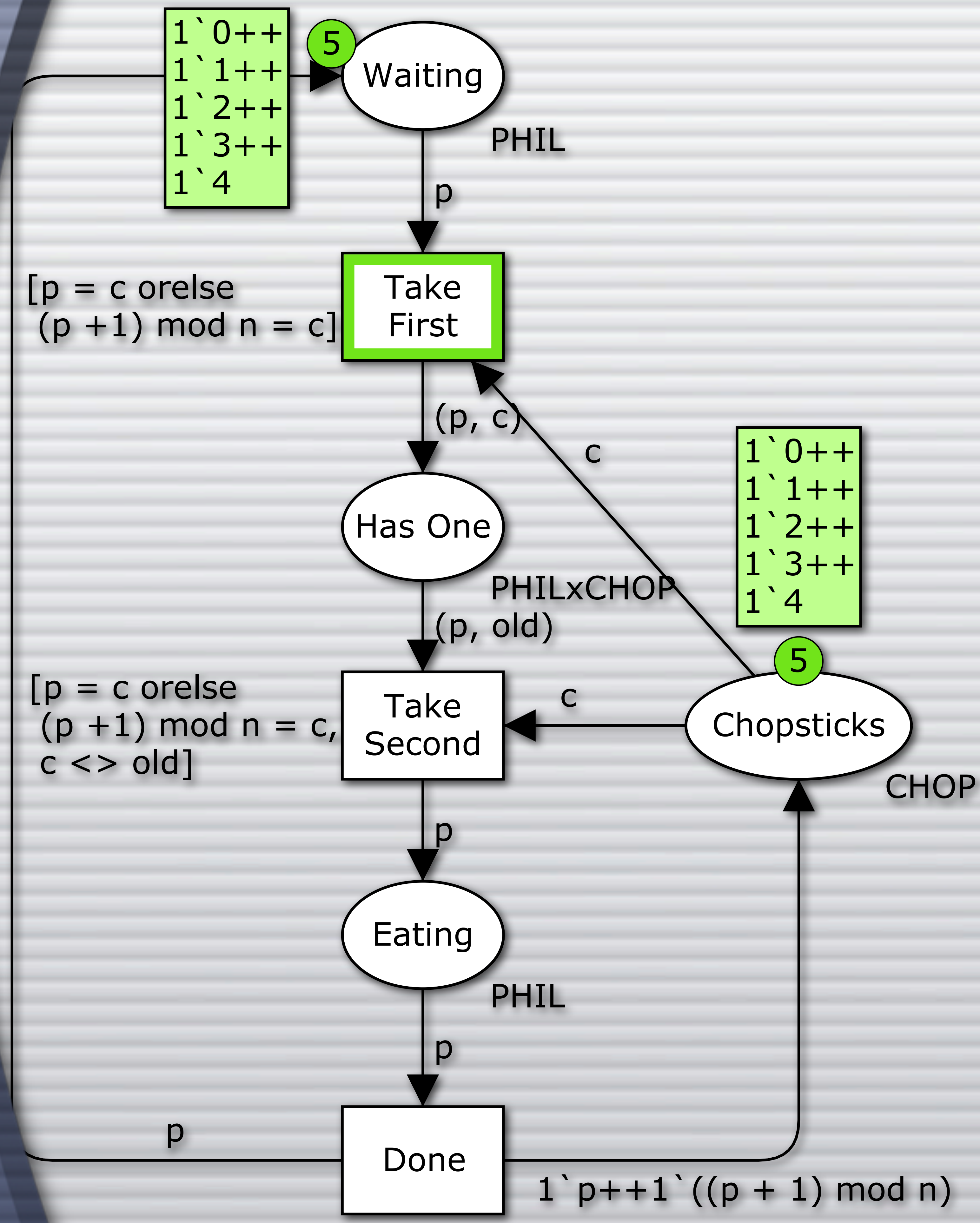
LTL Checking in ASAP

Example: Dining Philosophers

p = philisopher 1 eats

Philosopher 1 will always eat at some point

- $\varphi'' = F\ p$
- $\varphi''' = G\ F\ p$



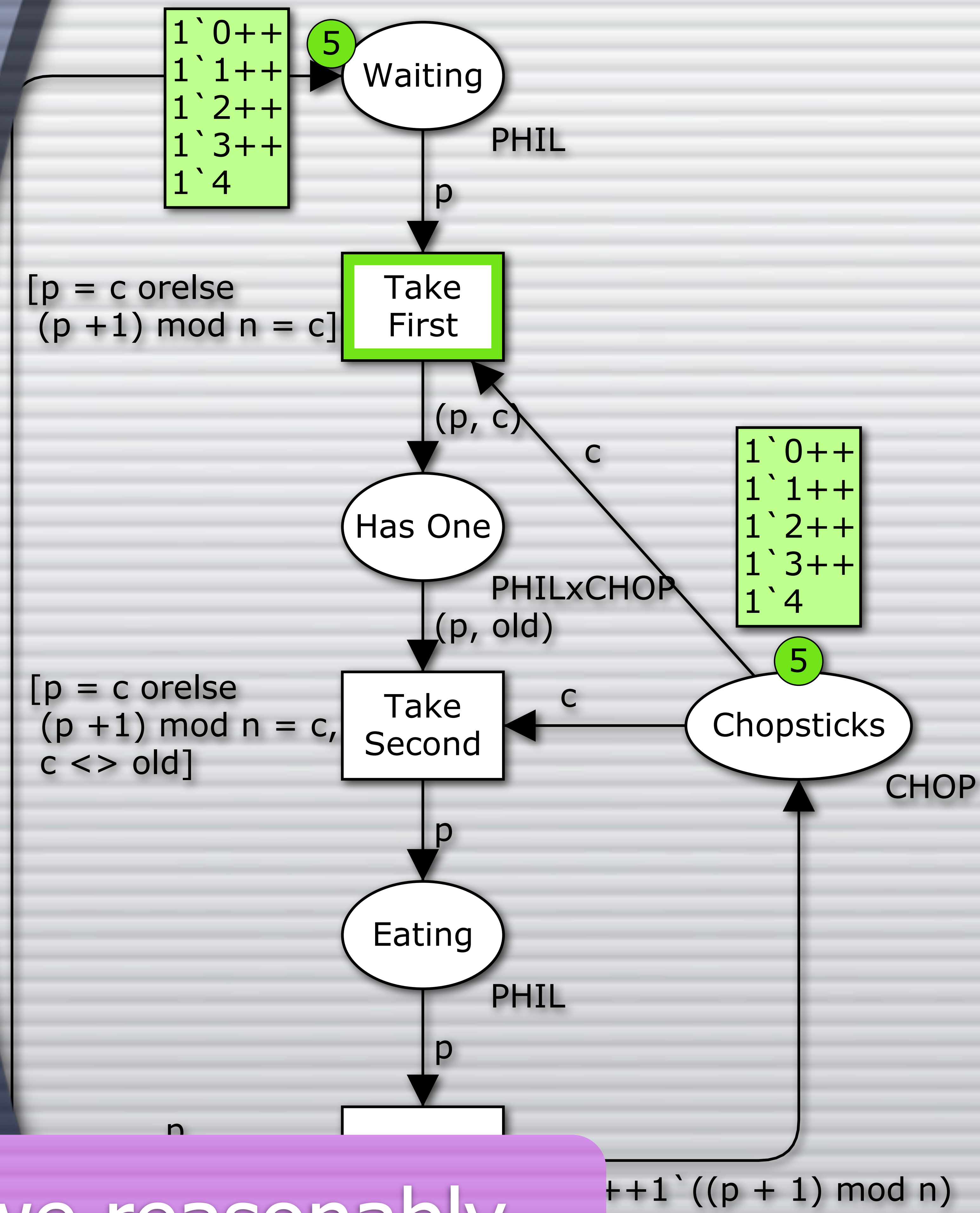
Example: Dining Philosophers

p = philisopher 1 eats

Philosopher 1 will always eat at some point

- φ'' = F p
- φ''' = G F p

How do we reasonably enter formulas in the tool?



Entering Formulas

- We already said that our atomic propositions are functions
- It is possible to build a complex data structure representing the formulas and APs as functions
- ...which we do internally and immediately hide from users :-)

Demo: LTL

- Show JoSEL task
- Create formulas F p and G F p
  - Mapping can be reused
- Show error traces
- (Draw Büchi automaton)