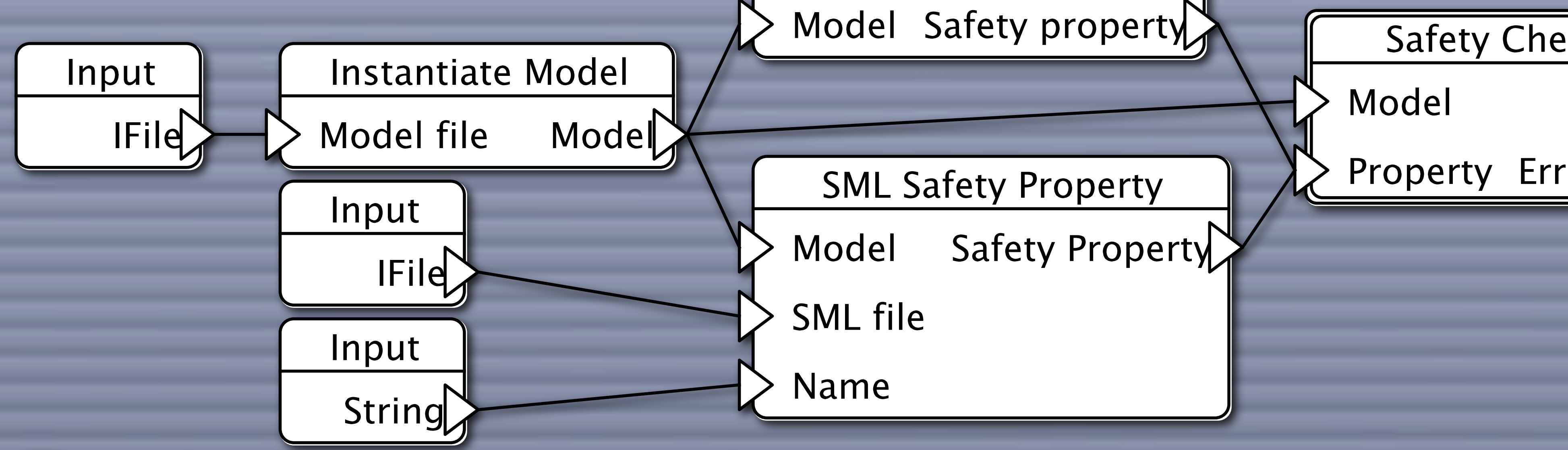
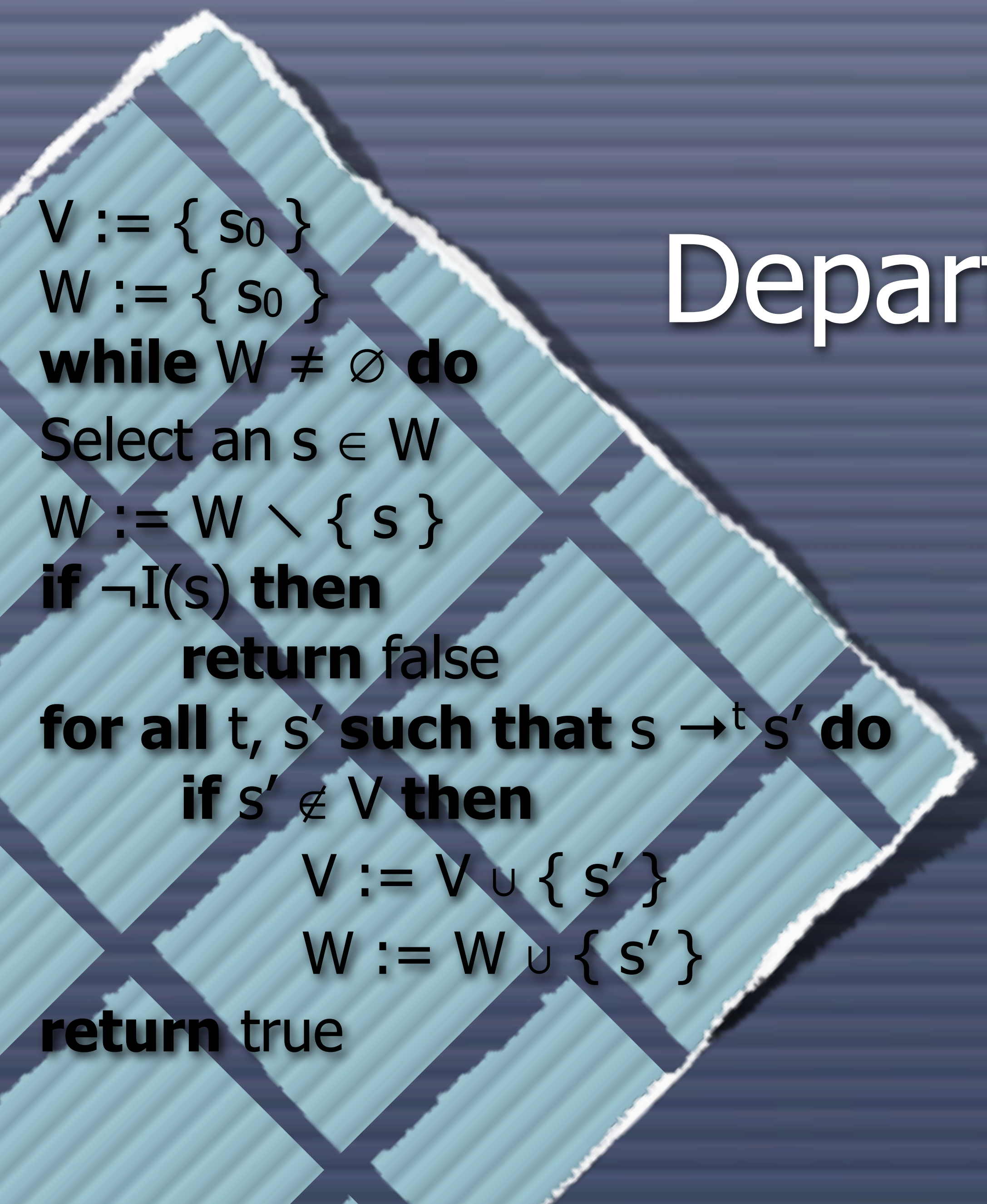
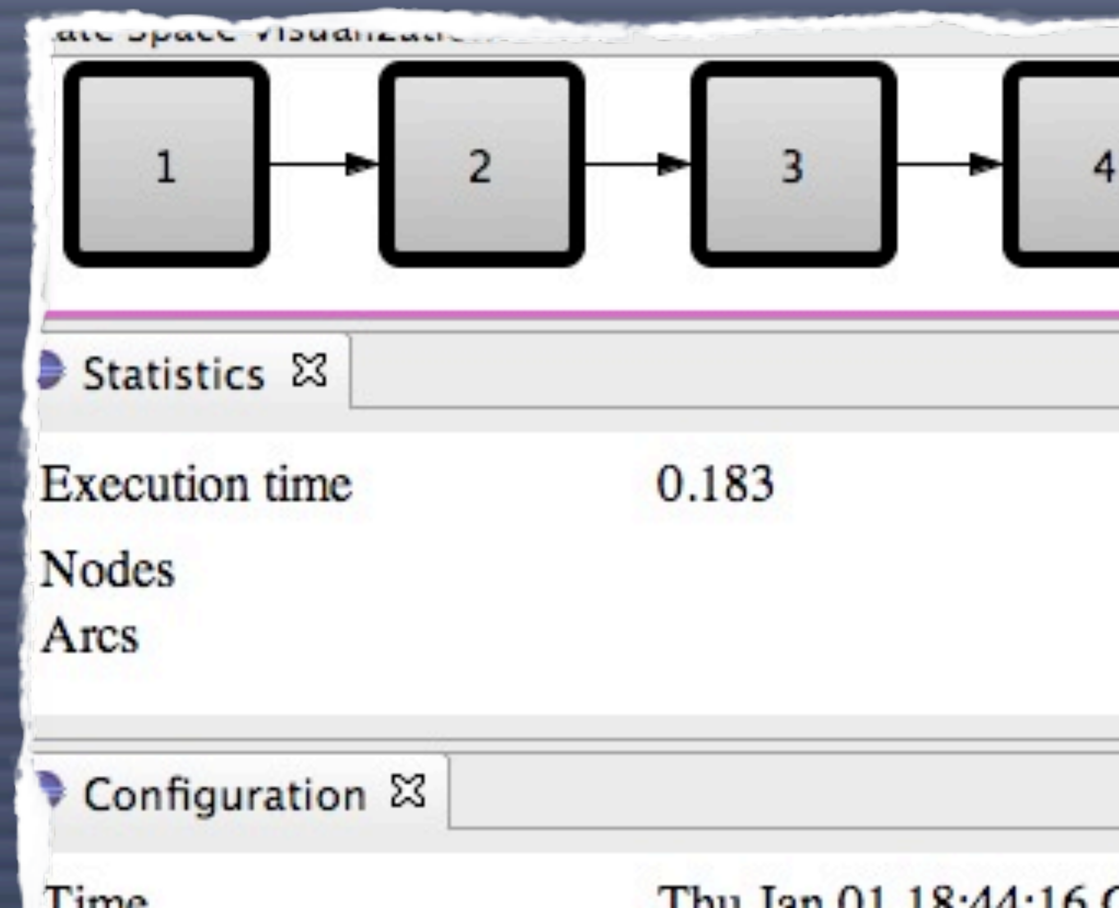


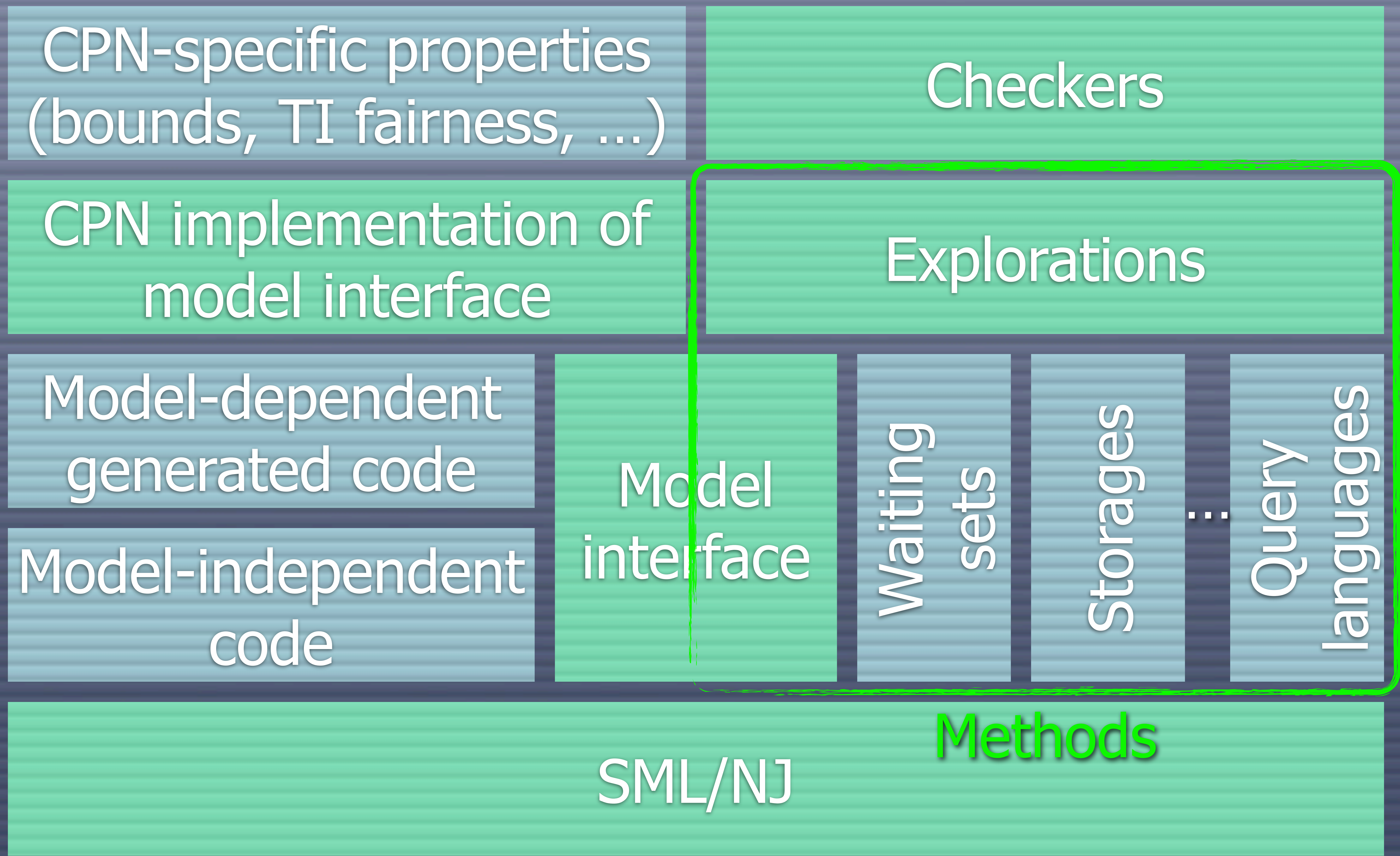
Advanced State Space Methods and ASAP: Simple Methods

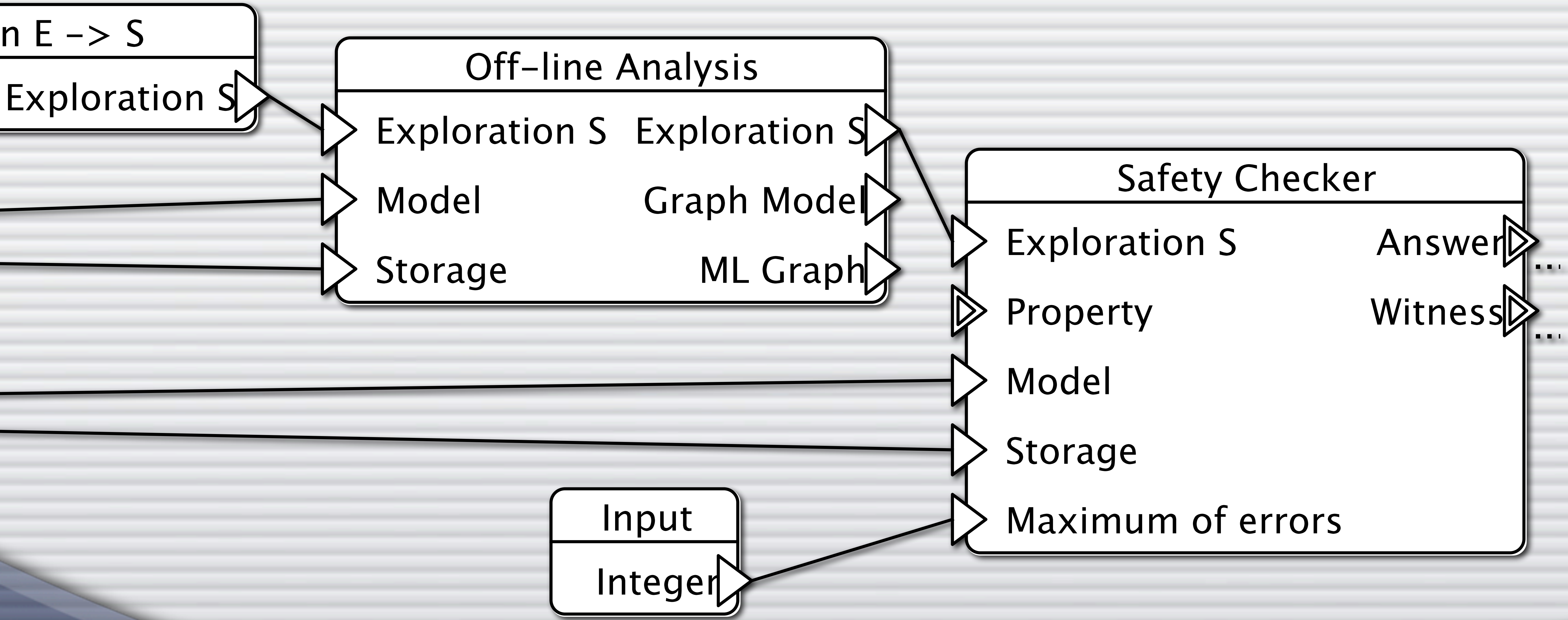
Michael Westergaard
Department of Computer Science
Aarhus University
mw@cs.au.dk





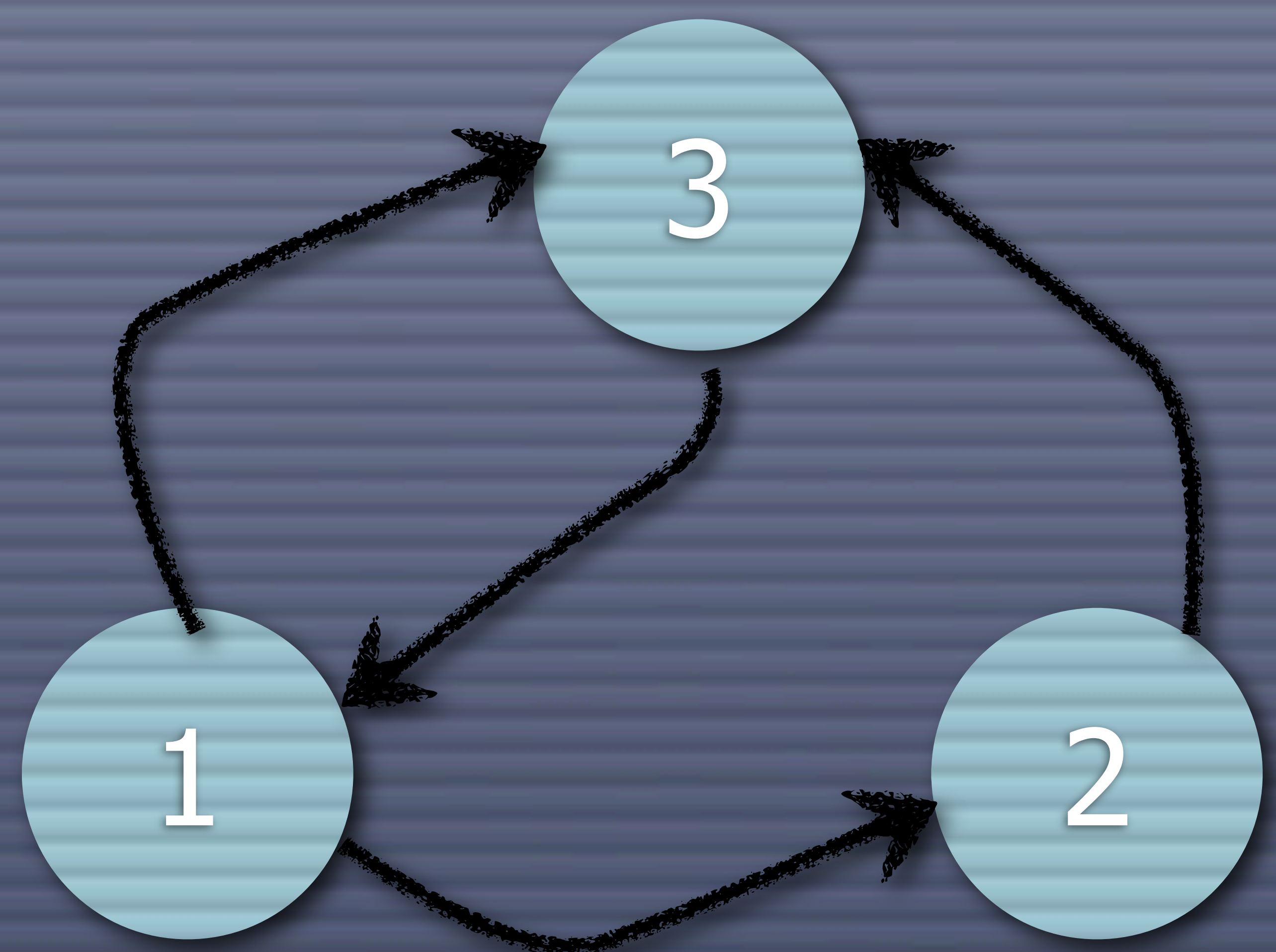
State-space Tool of ASAP





Example:
On-line vs. Off-line

Constructing a State Space



Constructing a State Space

V:
W:

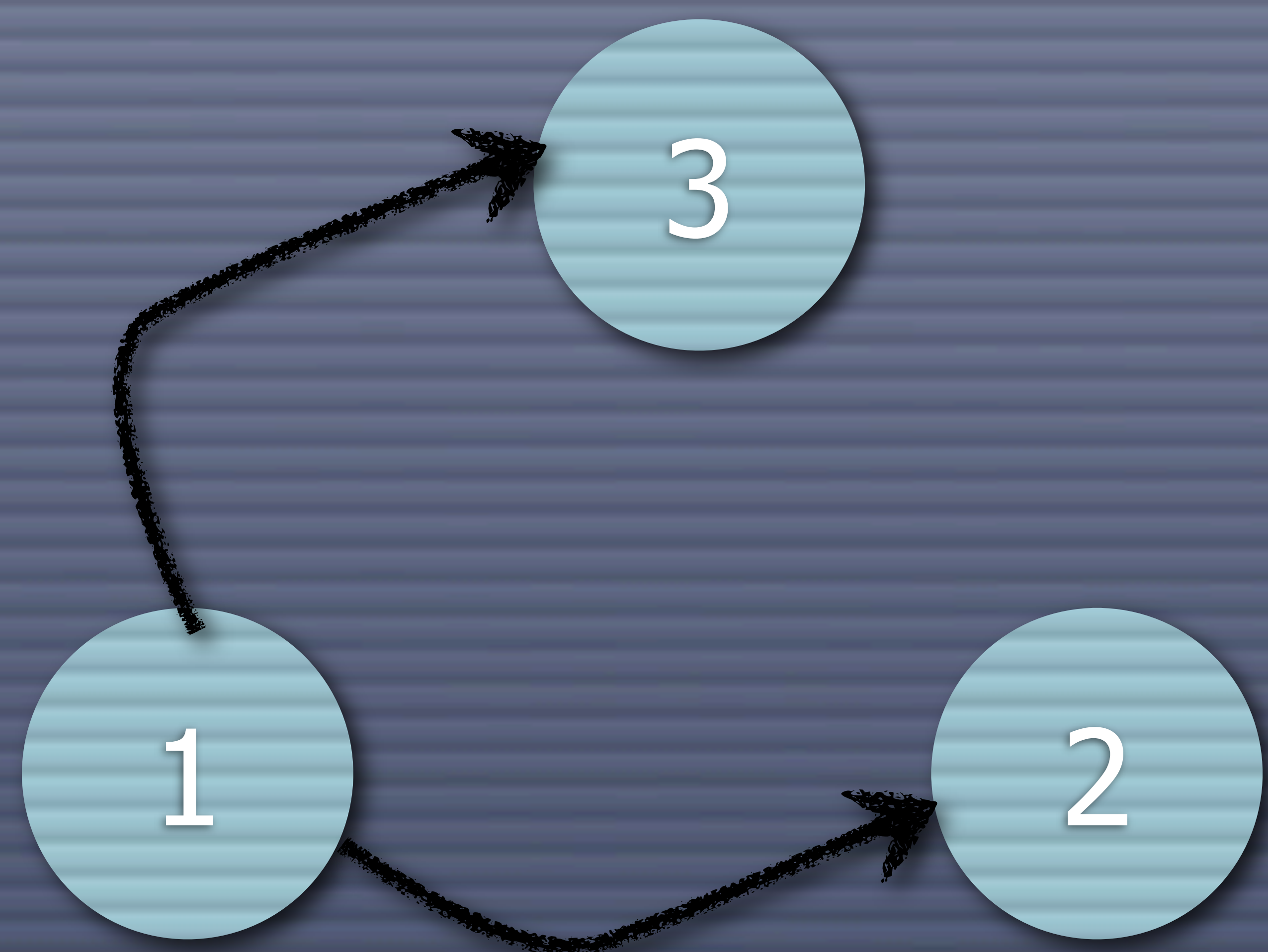
Constructing a State Space

V: 1
W: 1

Constructing a State Space

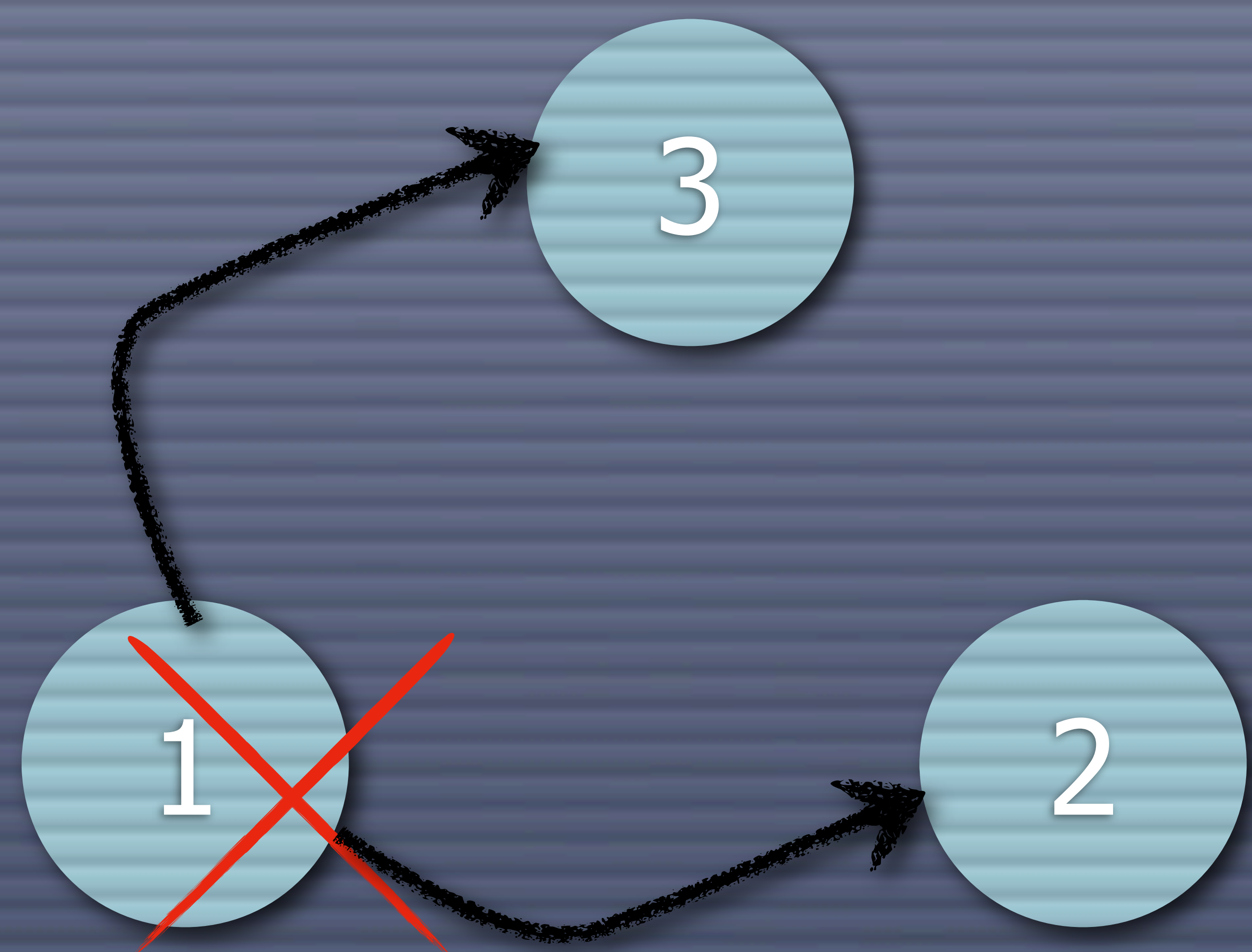
V: 1
W:

Constructing a State Space



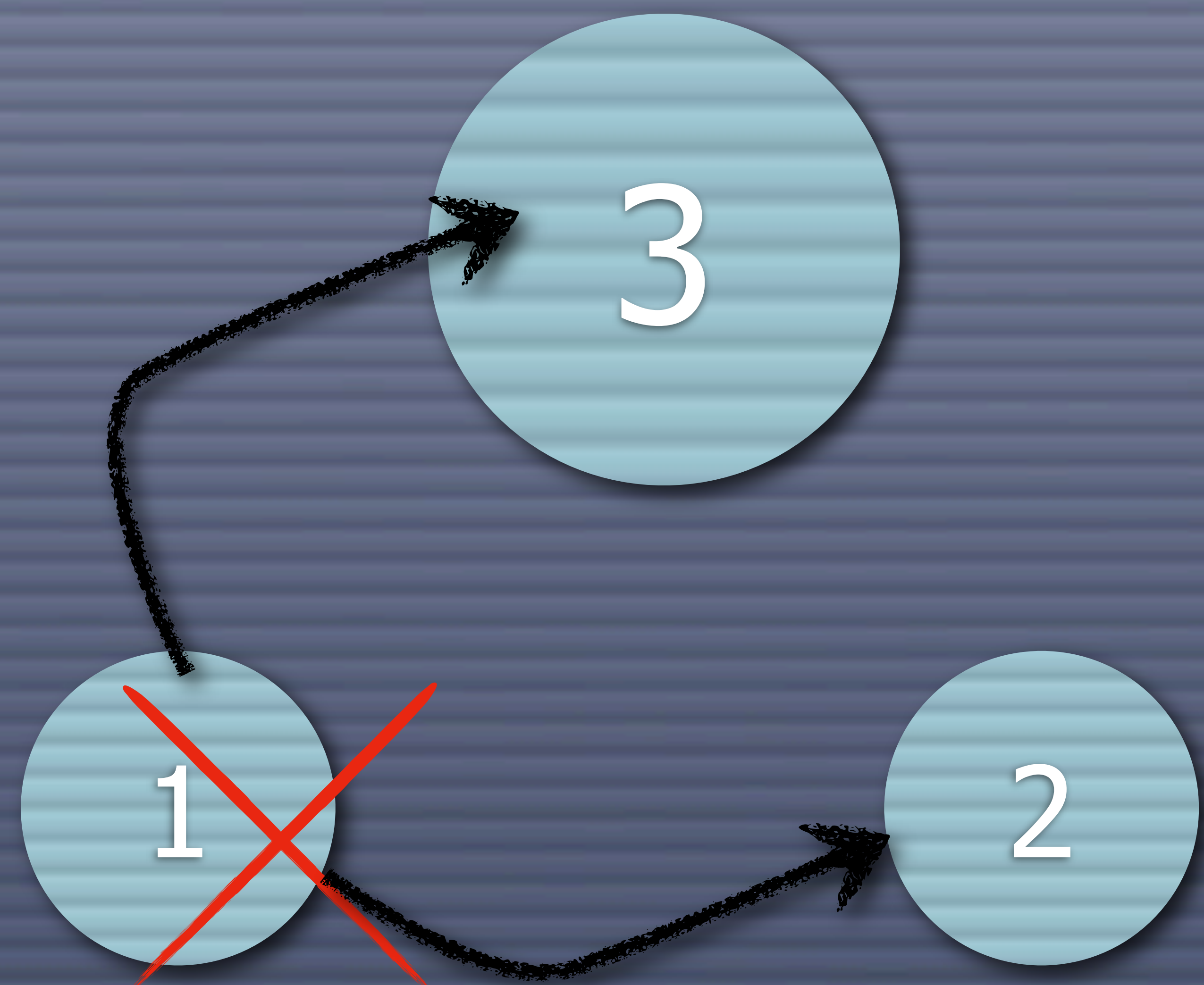
V: 1 2 3
W: 2 3

Constructing a State Space



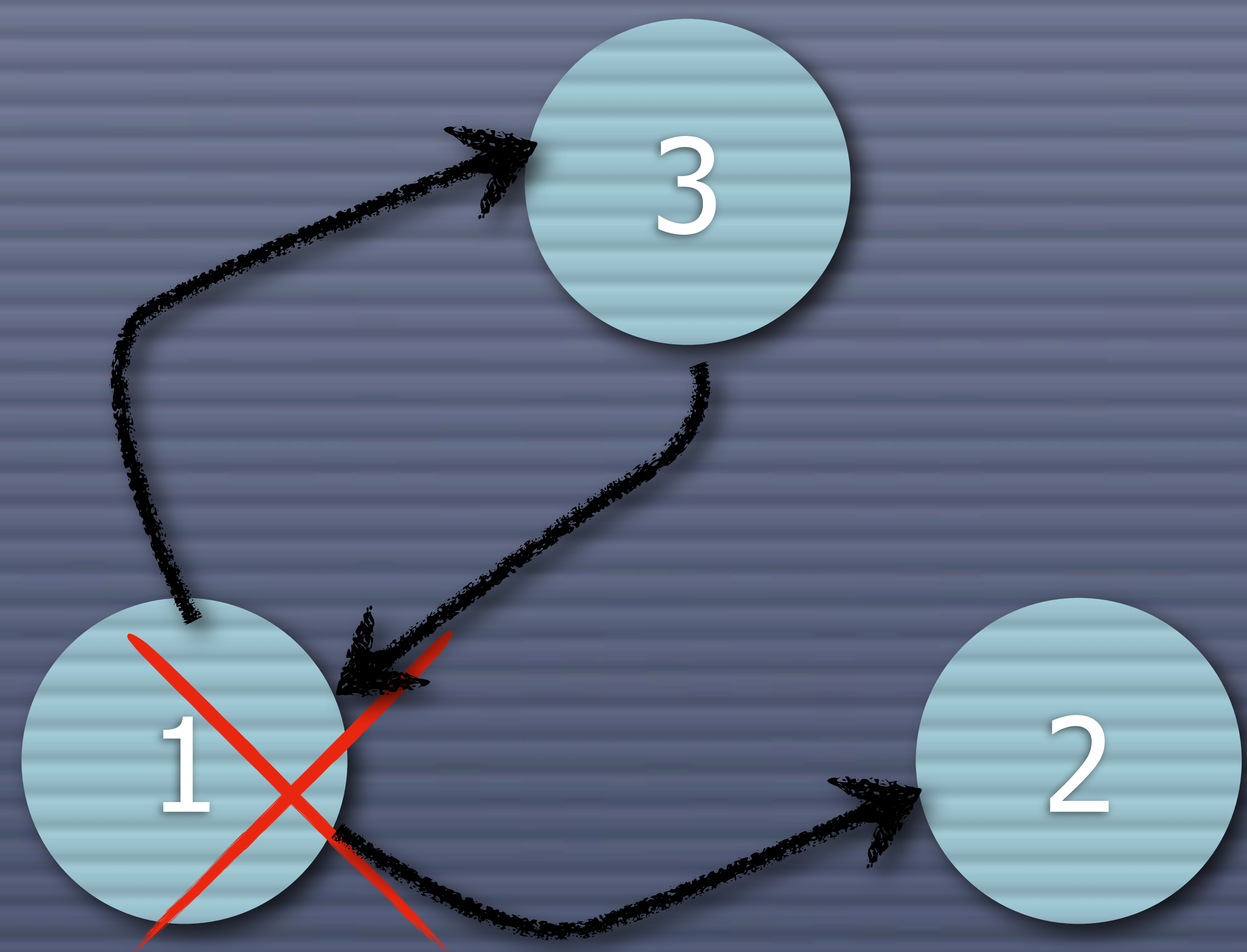
V: 1 2 3
W: 2 3

Constructing a State Space



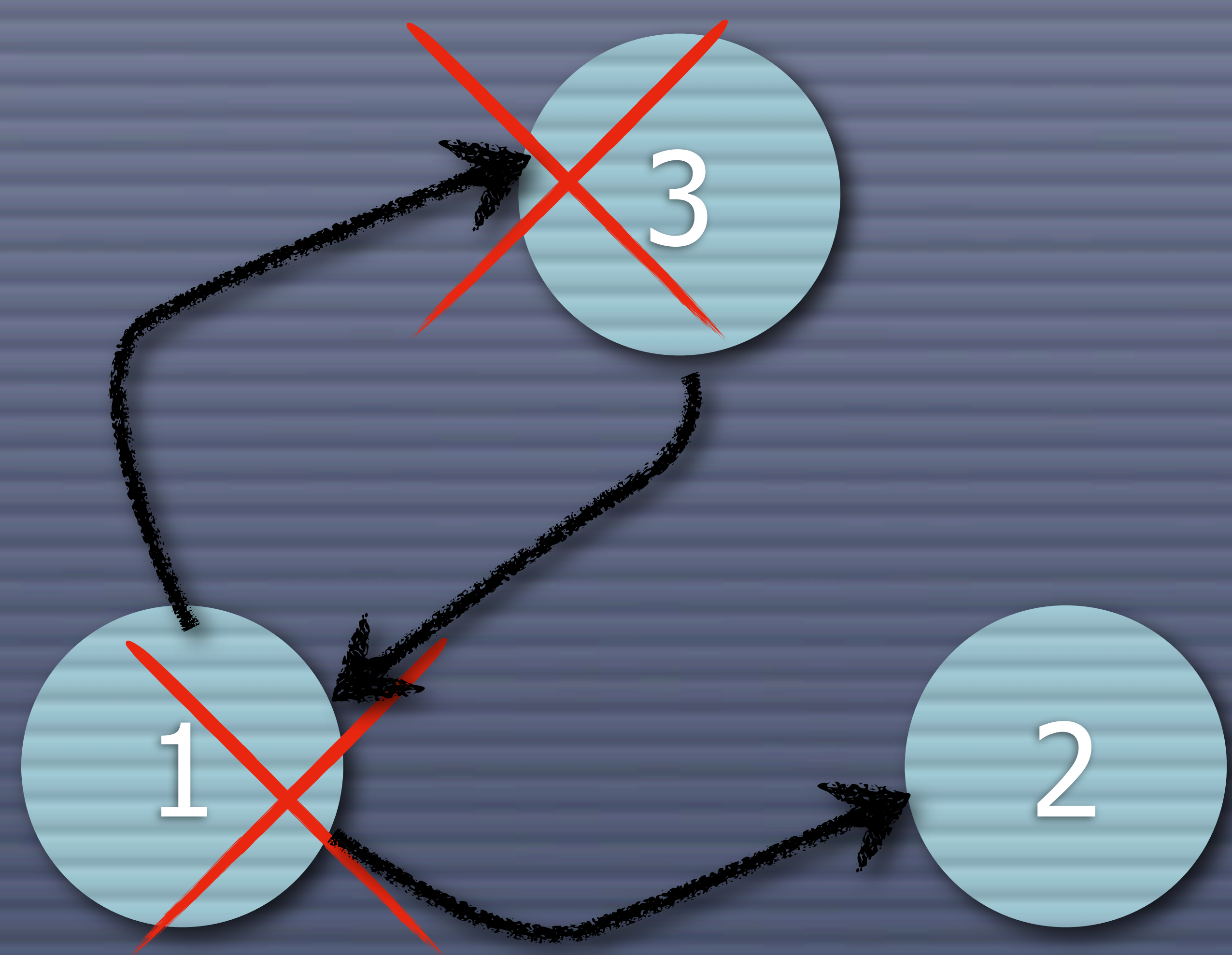
V: 1 2 3
W: 2

Constructing a State Space



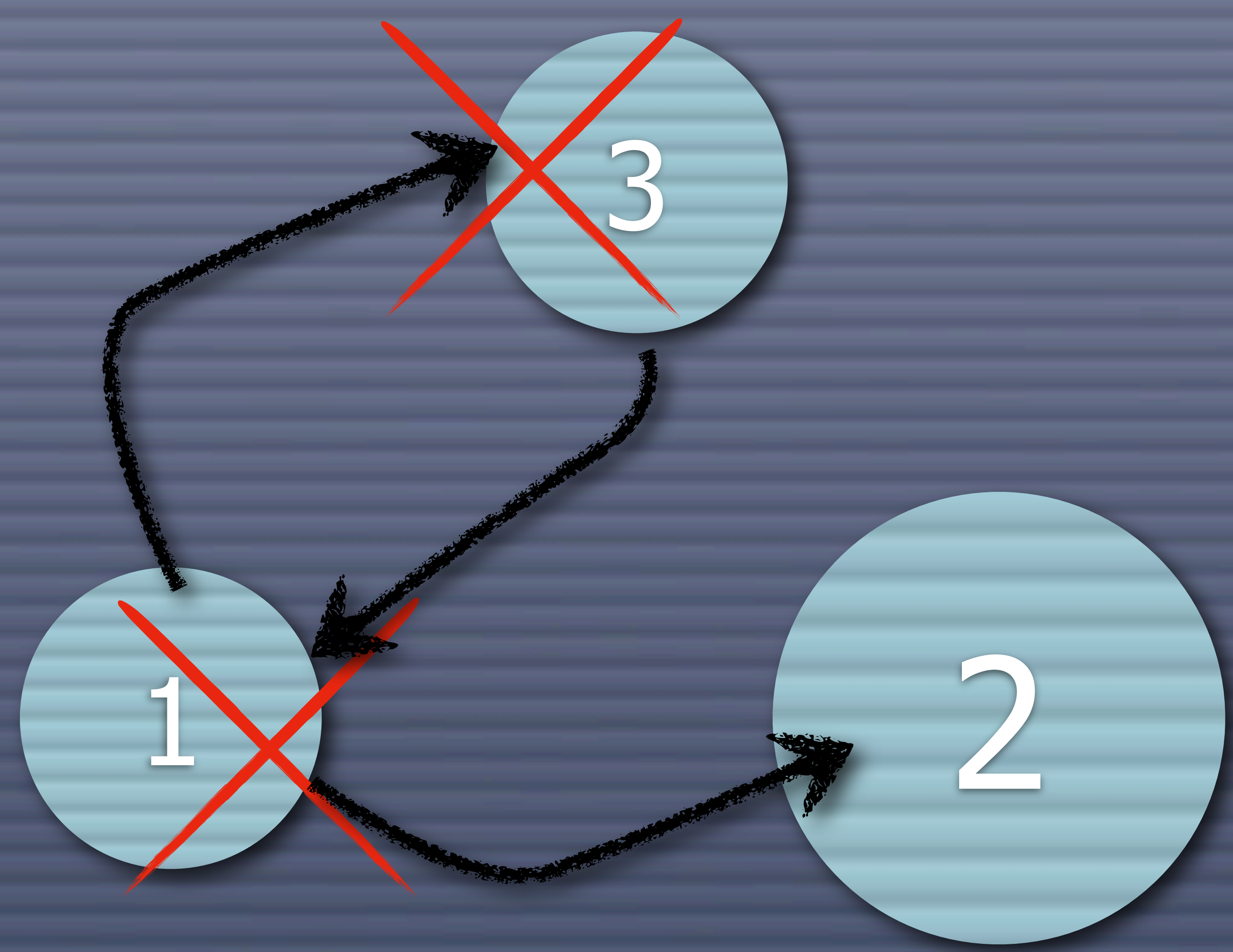
V: 1 2 3
W: 2

Constructing a State Space



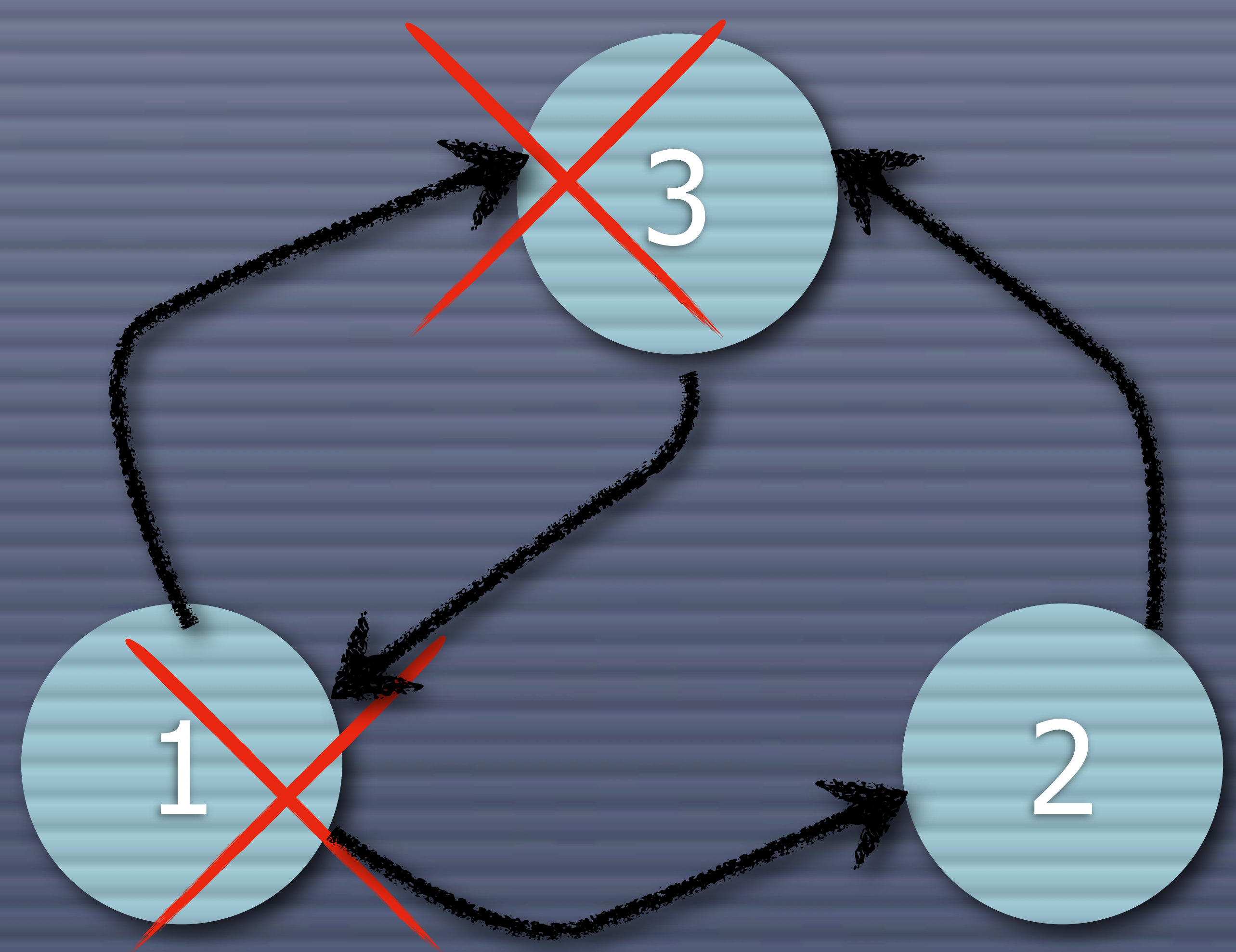
V: 1 2 3
W: 2

Constructing a State Space



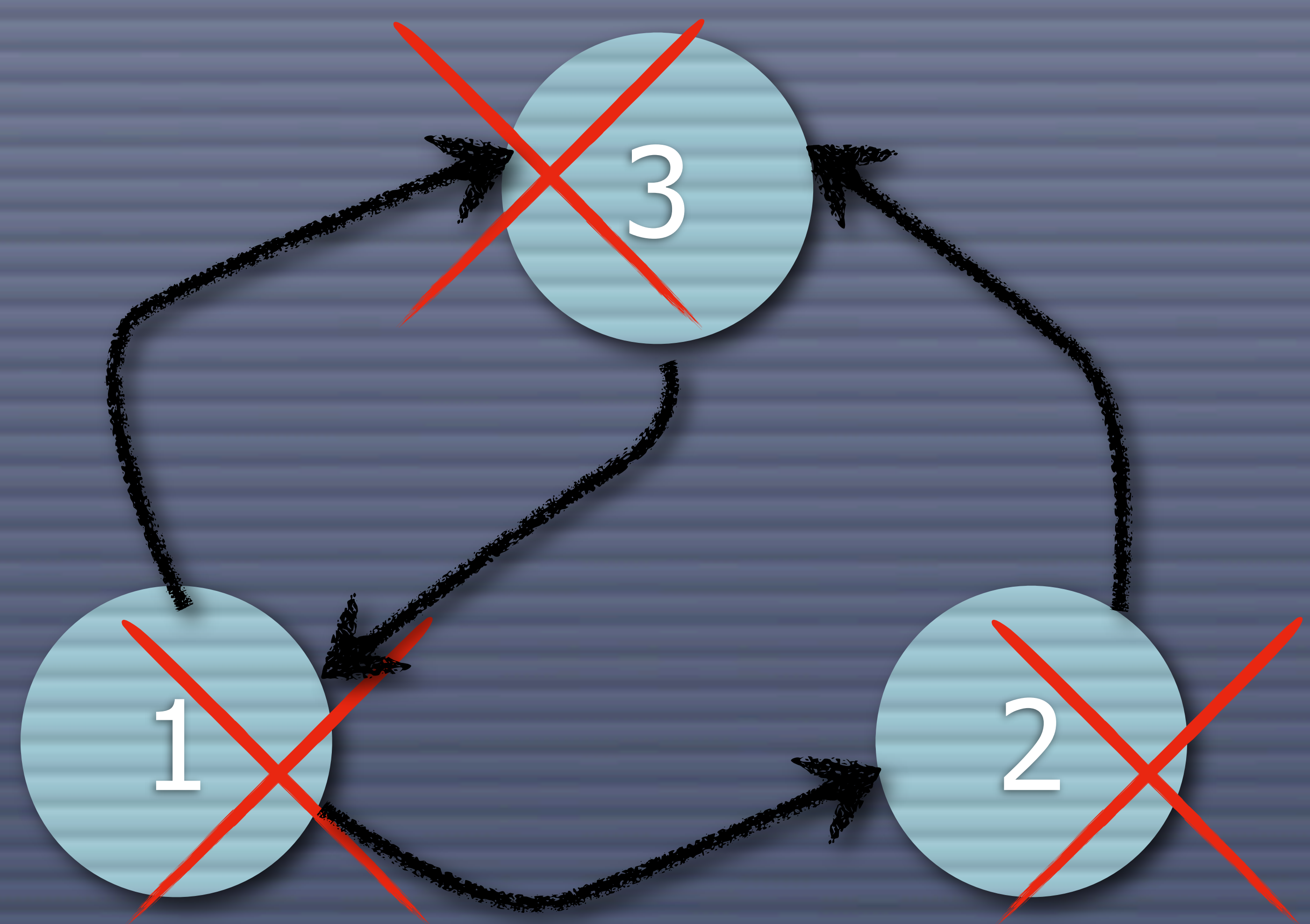
V: 1 2 3
W:

Constructing a State Space



V: 1 2 3
W:

Constructing a State Space



V: 1 2 3
W:

Off-line Safety Checker

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

for all $v \in V$ **do**
  if $\neg I(v)$ **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** $s \rightarrow^{t} s'$ **do**
        if $s' \notin V$ **then**
            $V := V \cup \{ s' \}$
            $W := W \cup \{ s' \}$
return true

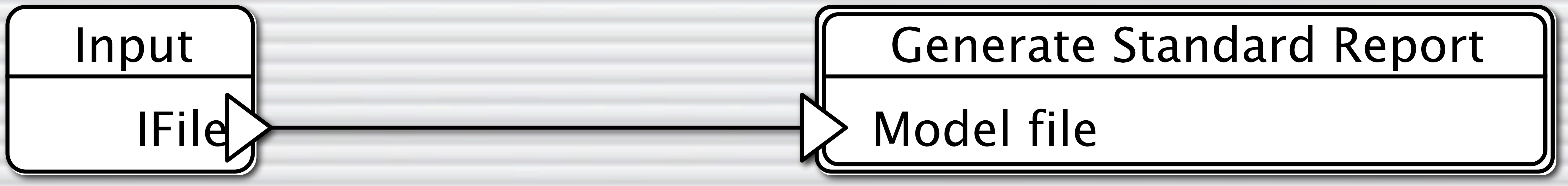
This is on-line analysis; we analyze the state space while we generate it.

On-line vs. Off-line

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

Demo:
On-line vs. Off-line (08)

- 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: Standard Report

The Standard Report

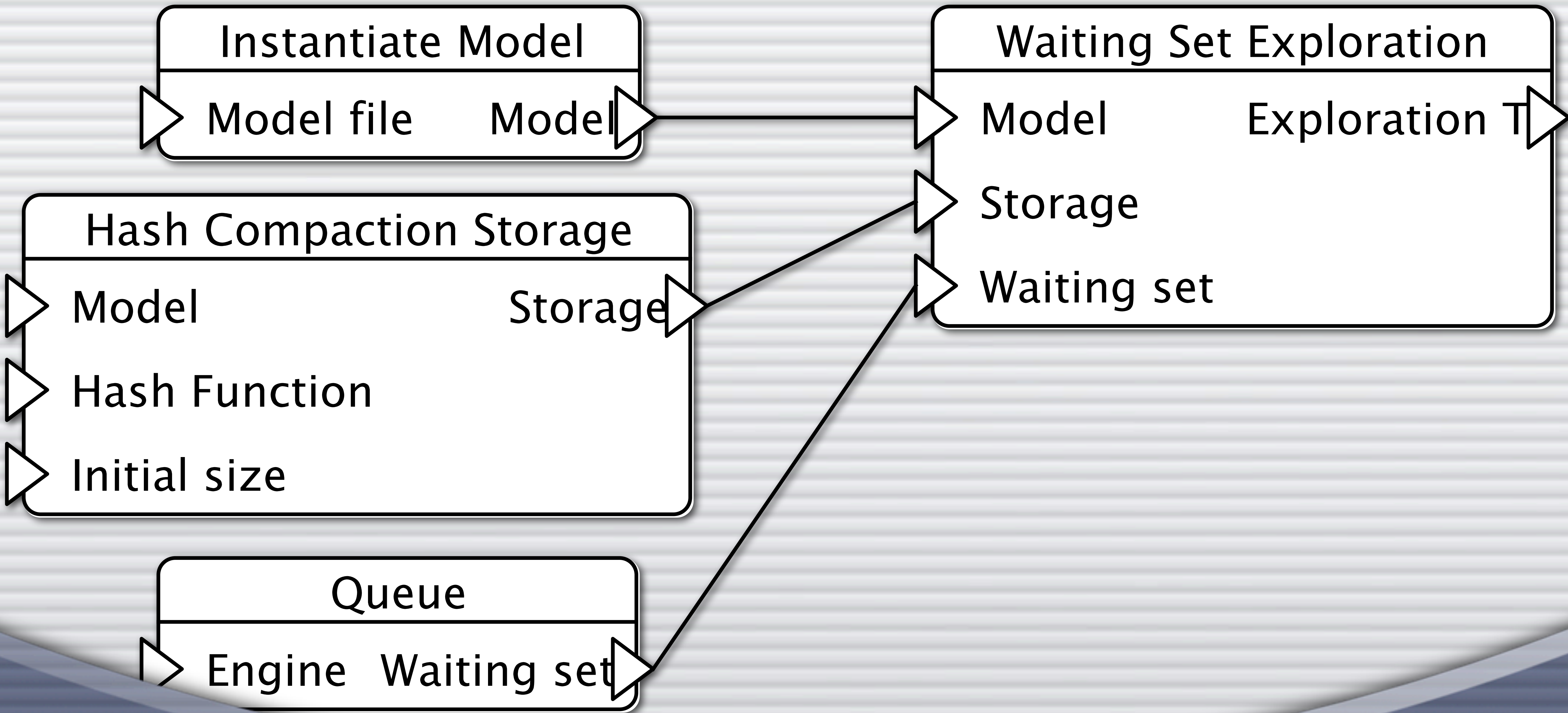
- CPN Tools (and DESIGN/CPN) creates a **standard report** with a set of standard properties
- It is possible to **remove** properties from the report
- It is not possible to **add** new properties to the report

The Standard Report in ASAP

- Is **very much** work in progress!
- Contains the same properties as the standard report in CPN Tools
- Is based on JoSEL

Demo:
Standard Report (09)

- Switch to standard report workspace
- Go thru the standard report JoSEL specification



Example: Hash-compaction

State Space Methods

- Store states compactly
- Delete states during exploration
- Store only some states
- Use external memory

State Space Methods

- Store states compactly
- Delete states during exploration
- Store only some states
- Use external memory

Hash-compaction

- A problem of the standard method is that we use 1000 bytes per state, and 4 GB / 1000 = $4 \cdot 10^6$ 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** $s \to^t s'$ **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 $W \neq \emptyset$ **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

We replace full state descriptors by compressed state descriptors in V

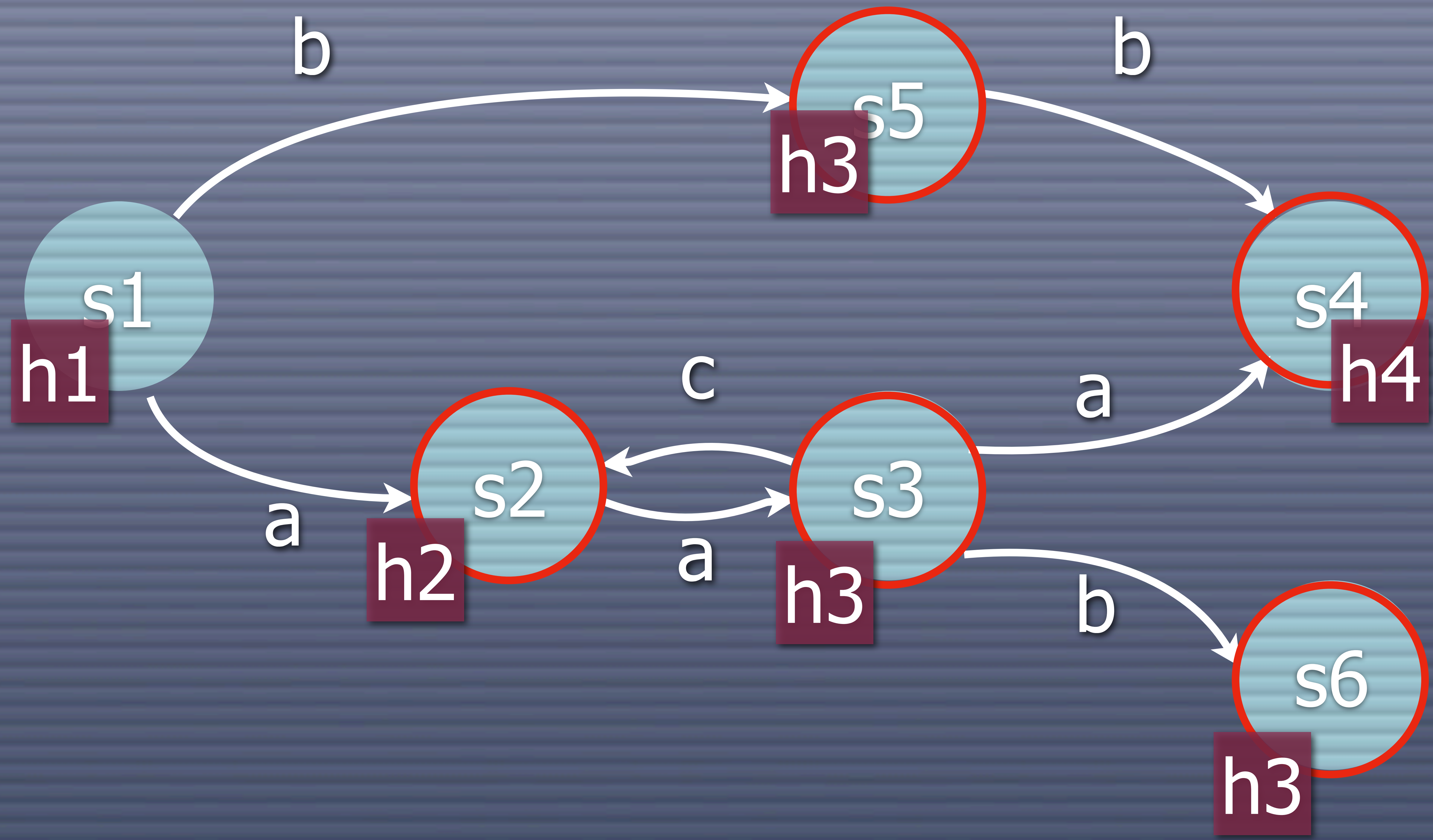
Hash-compaction

$V := \{ h(s_0) \}$
$W := \{ s_0 \}$
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 $h(s') \notin V$ **then**
      $V := V \cup \{ h(s') \}$
      $W := W \cup \{ s' \}$
return true

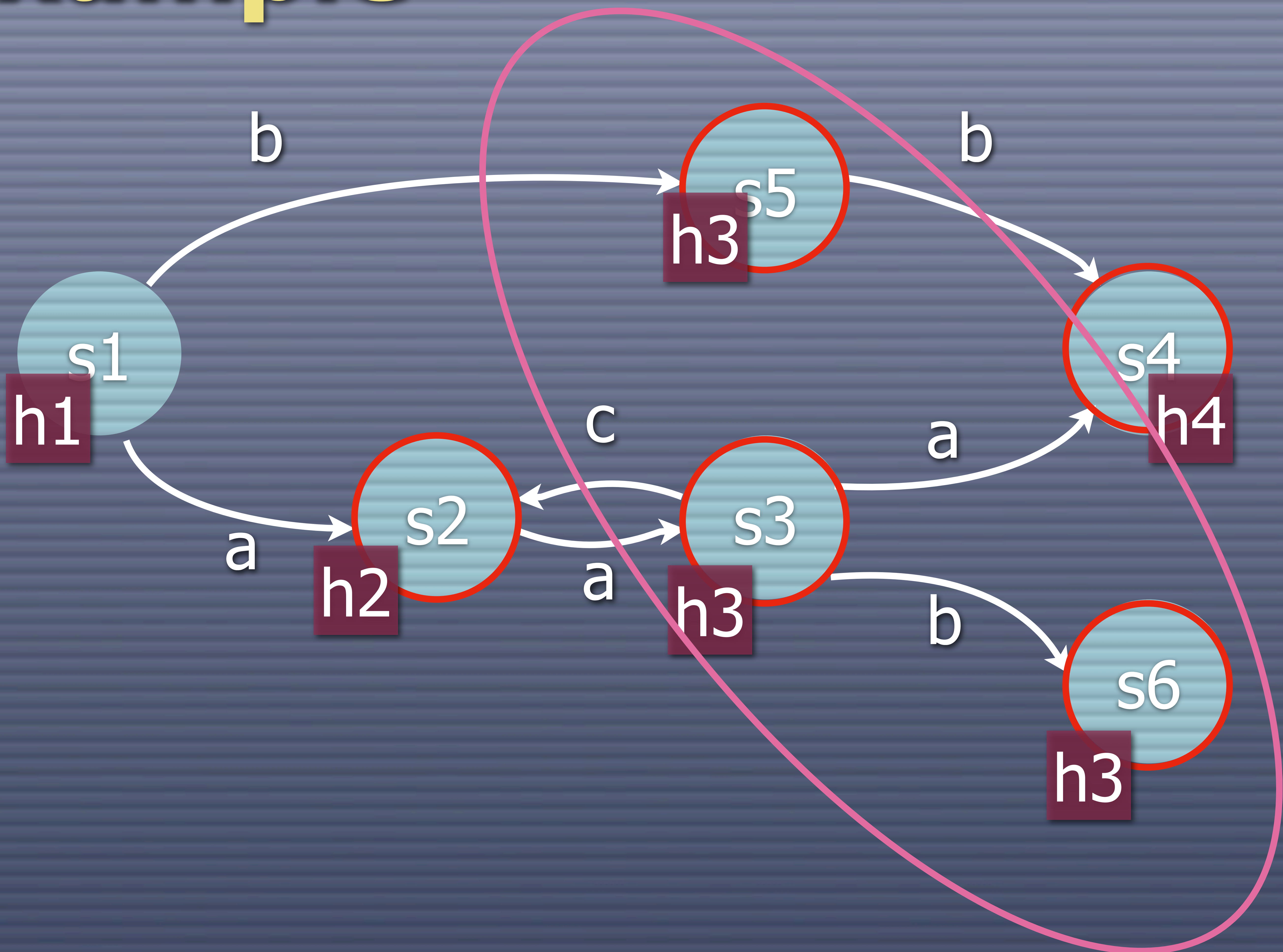
As long as we encounter no hash collisions, this algorithm works identically to the previous

We replace full state descriptors by compressed state descriptors in V

Example



Example



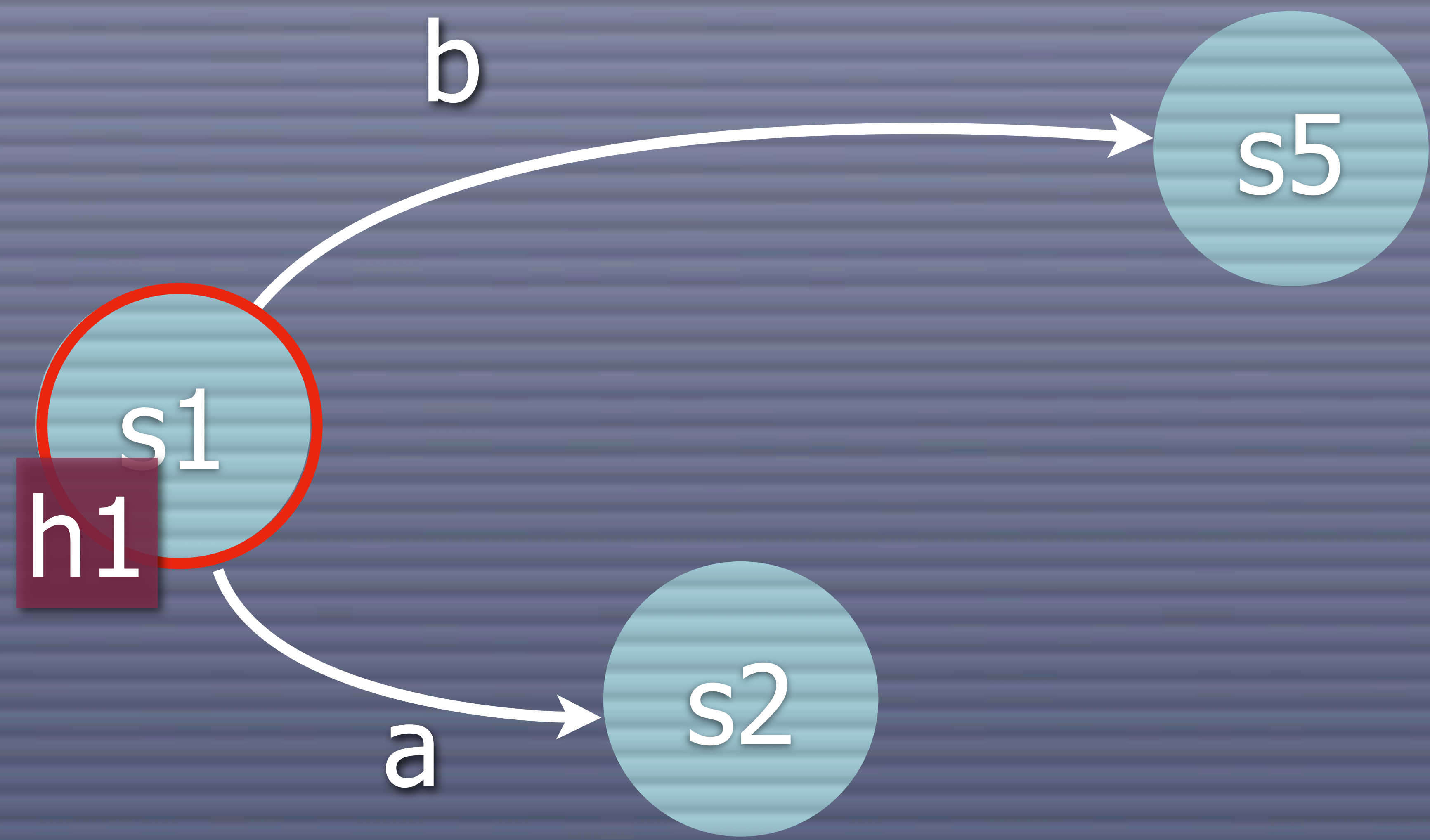
Example

V: h1
W: s1

Example

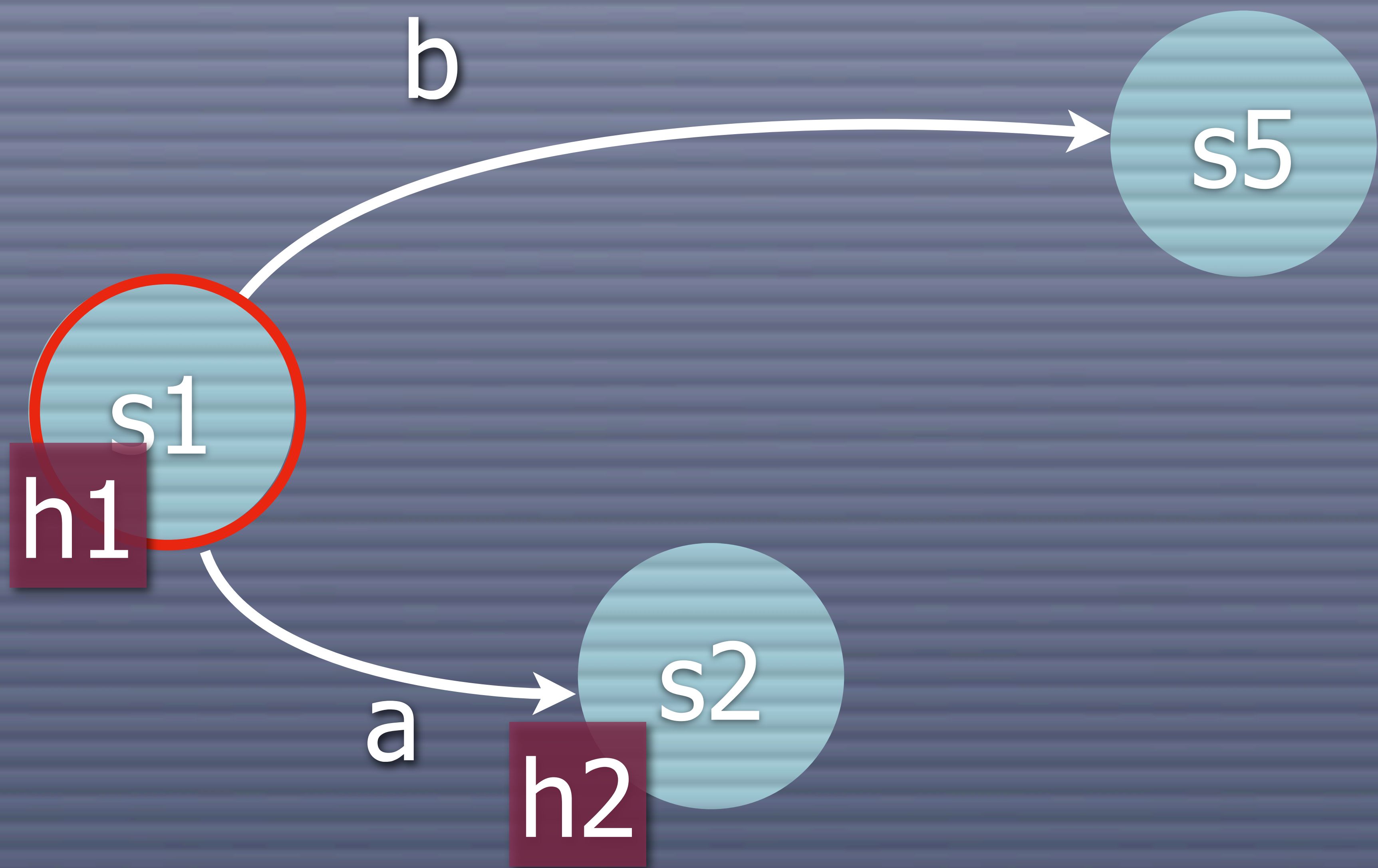
V: h1
W:

Example



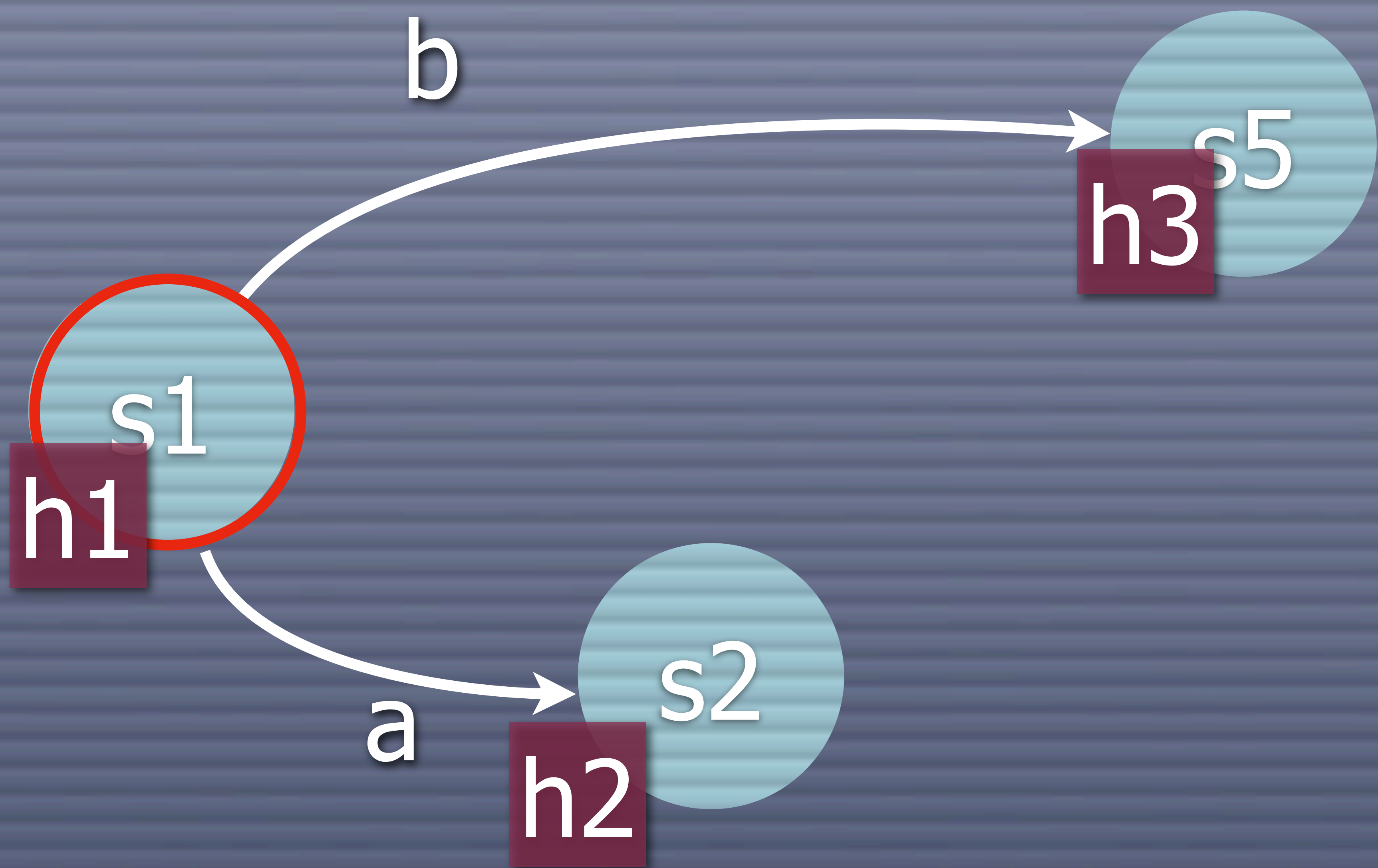
V: h1
W:

Example



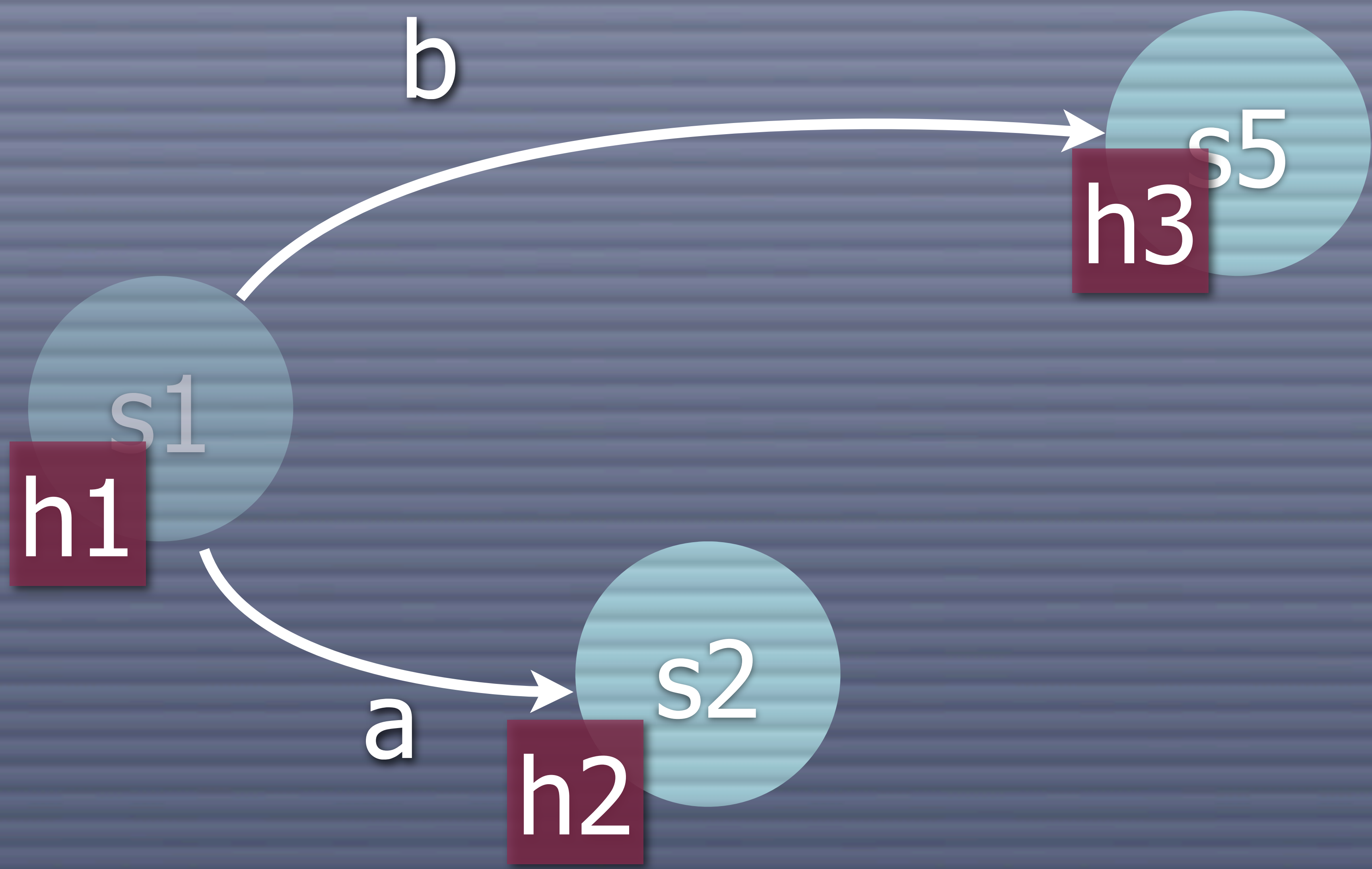
V: h1 h2
W: s2

Example



V: h1 h2 h3
W: s2 s5

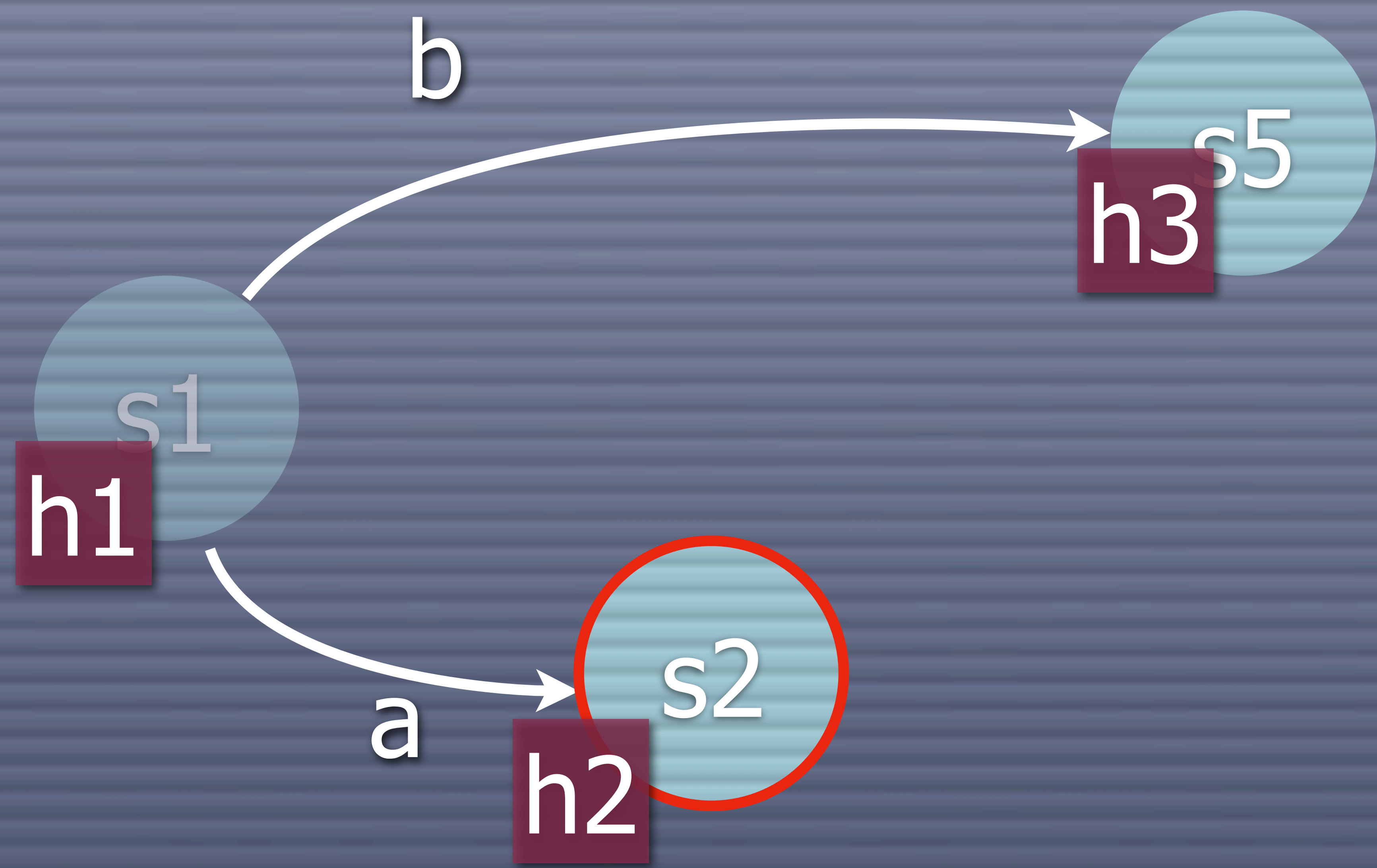
Example



V: h1 h2 h3

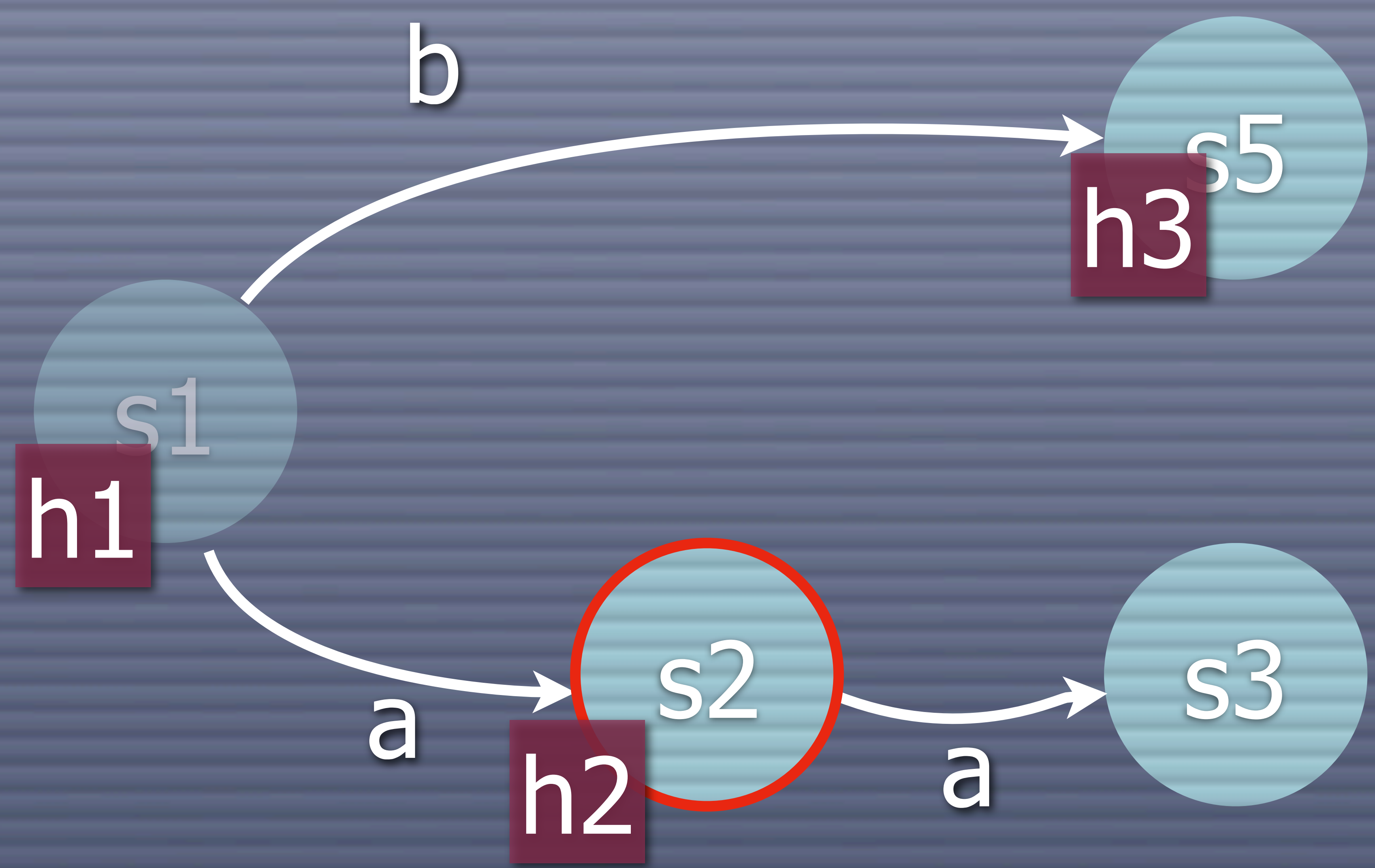
W: s2 s5

Example



V: h1 h2 h3
W: s5

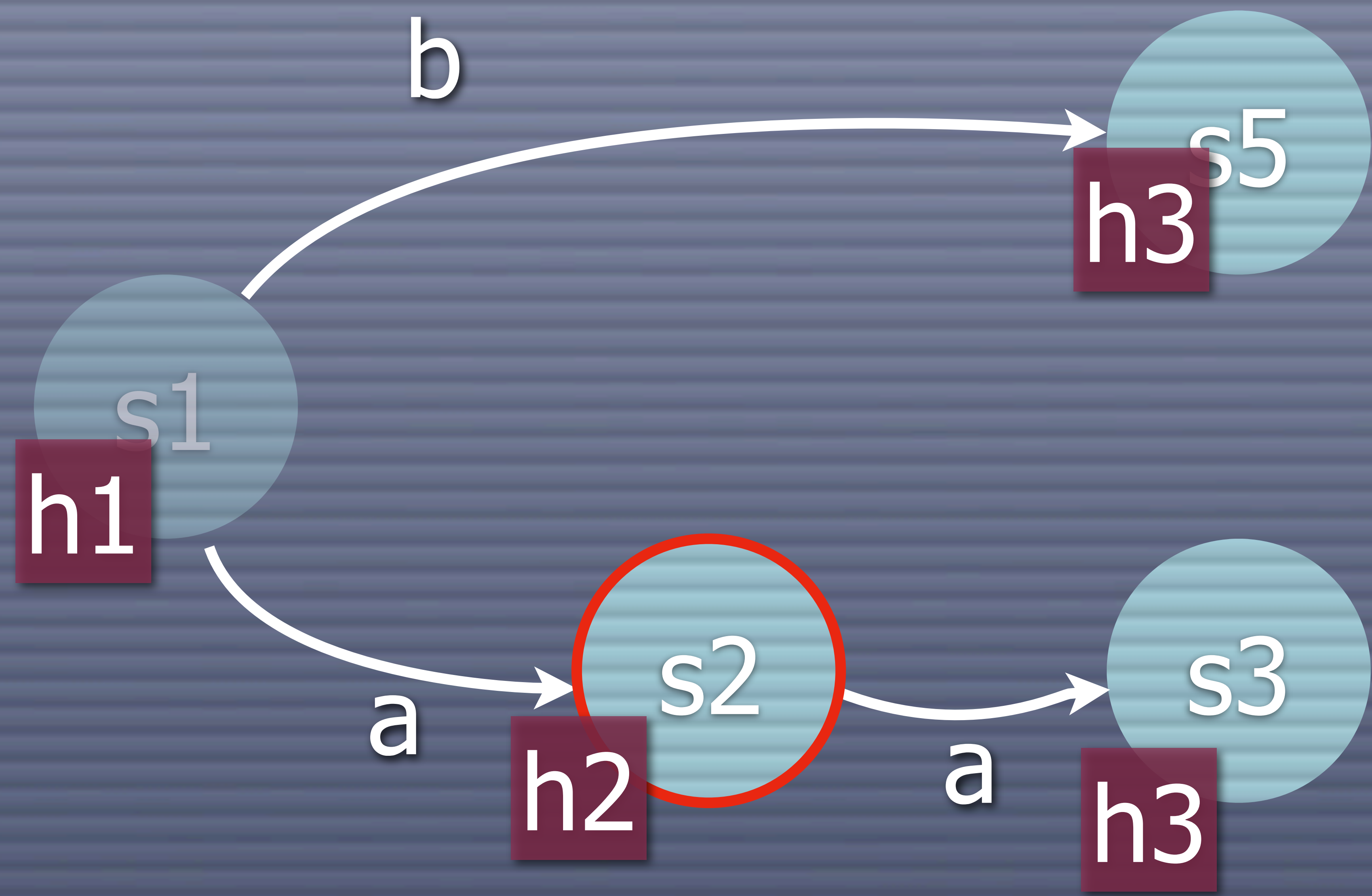
Example



V: h1 h2 h3

W: s5

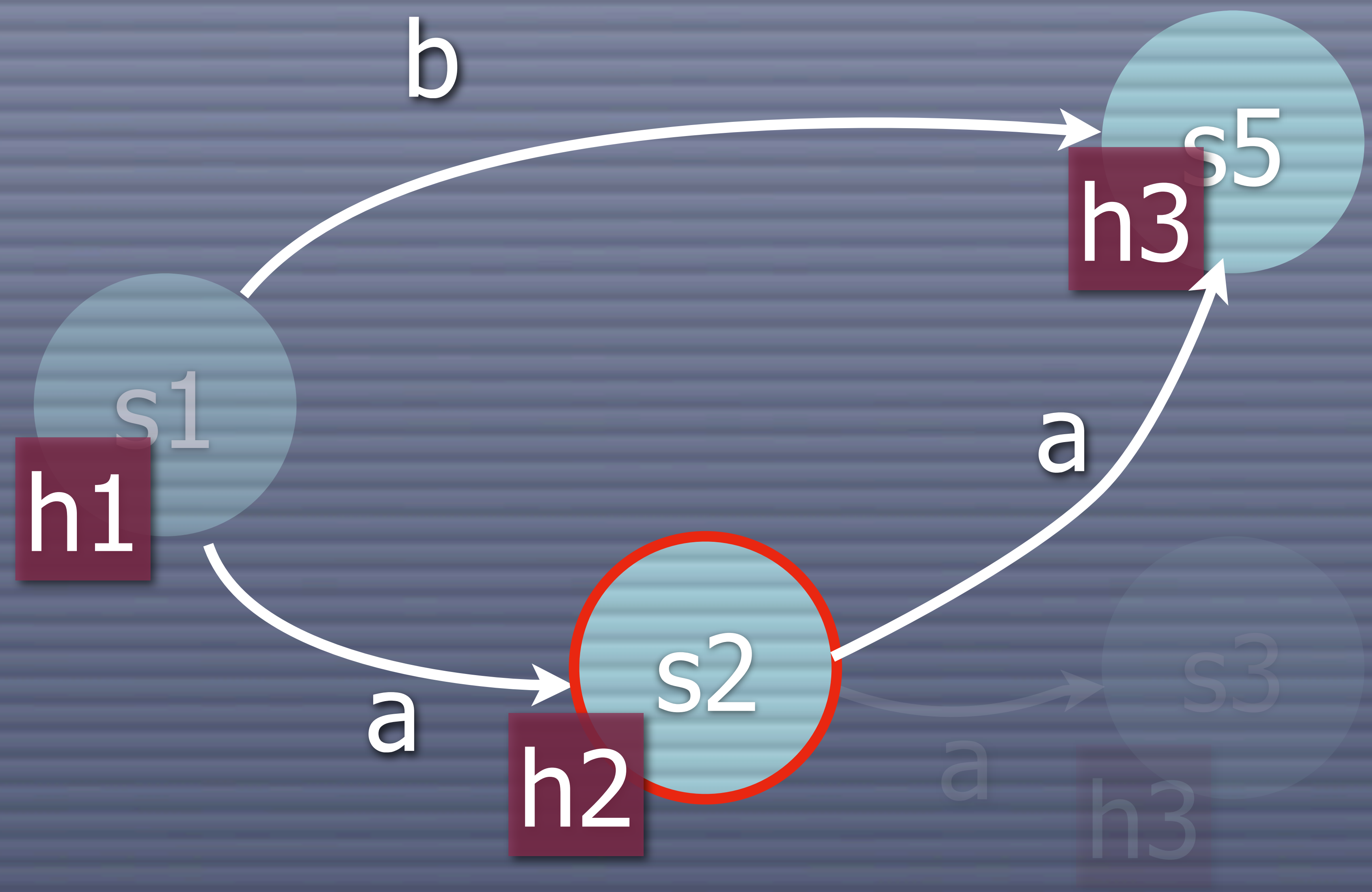
Example



V: h1 h2 h3

W: s5

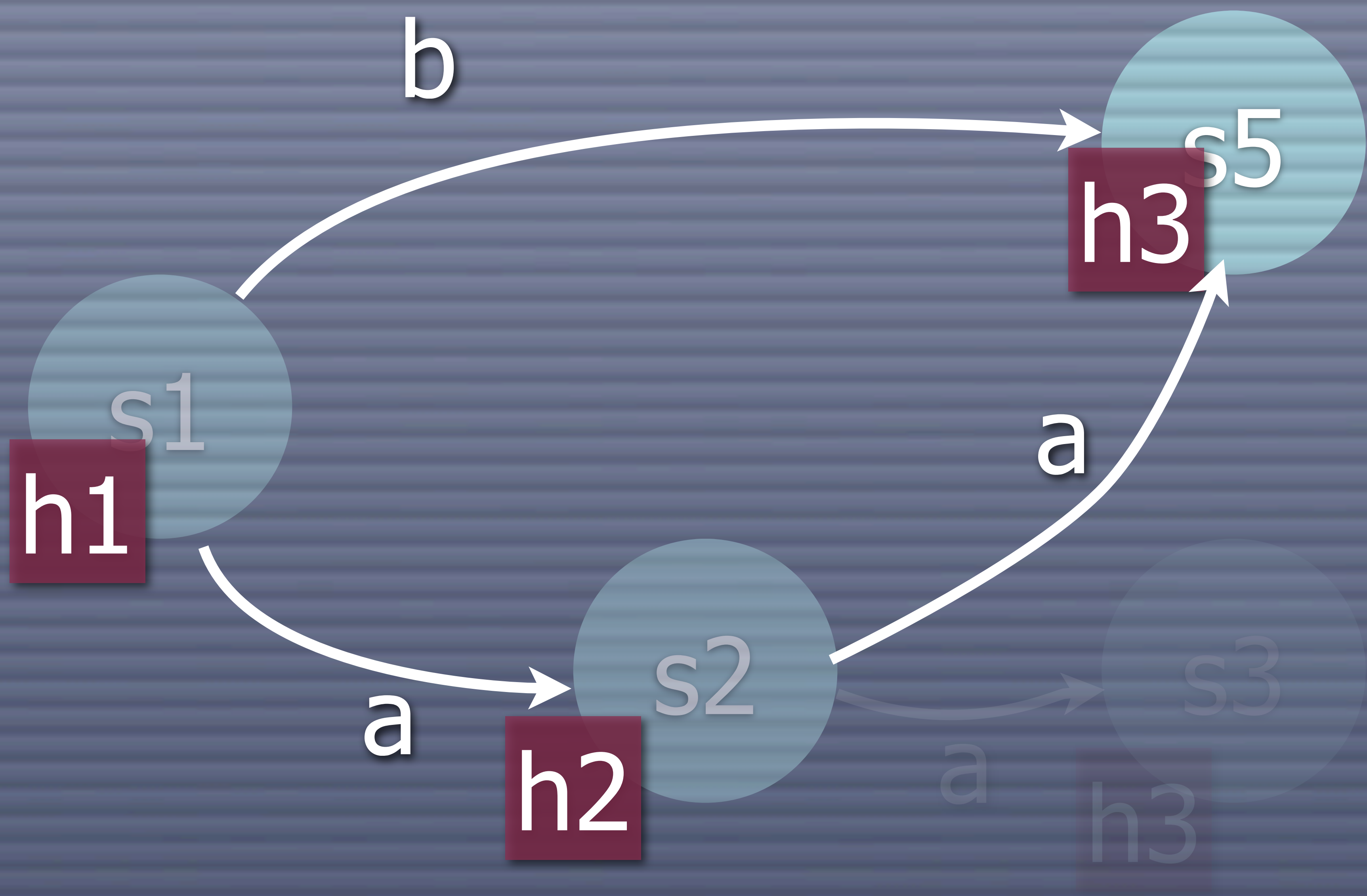
Example



V: h1 h2 h3

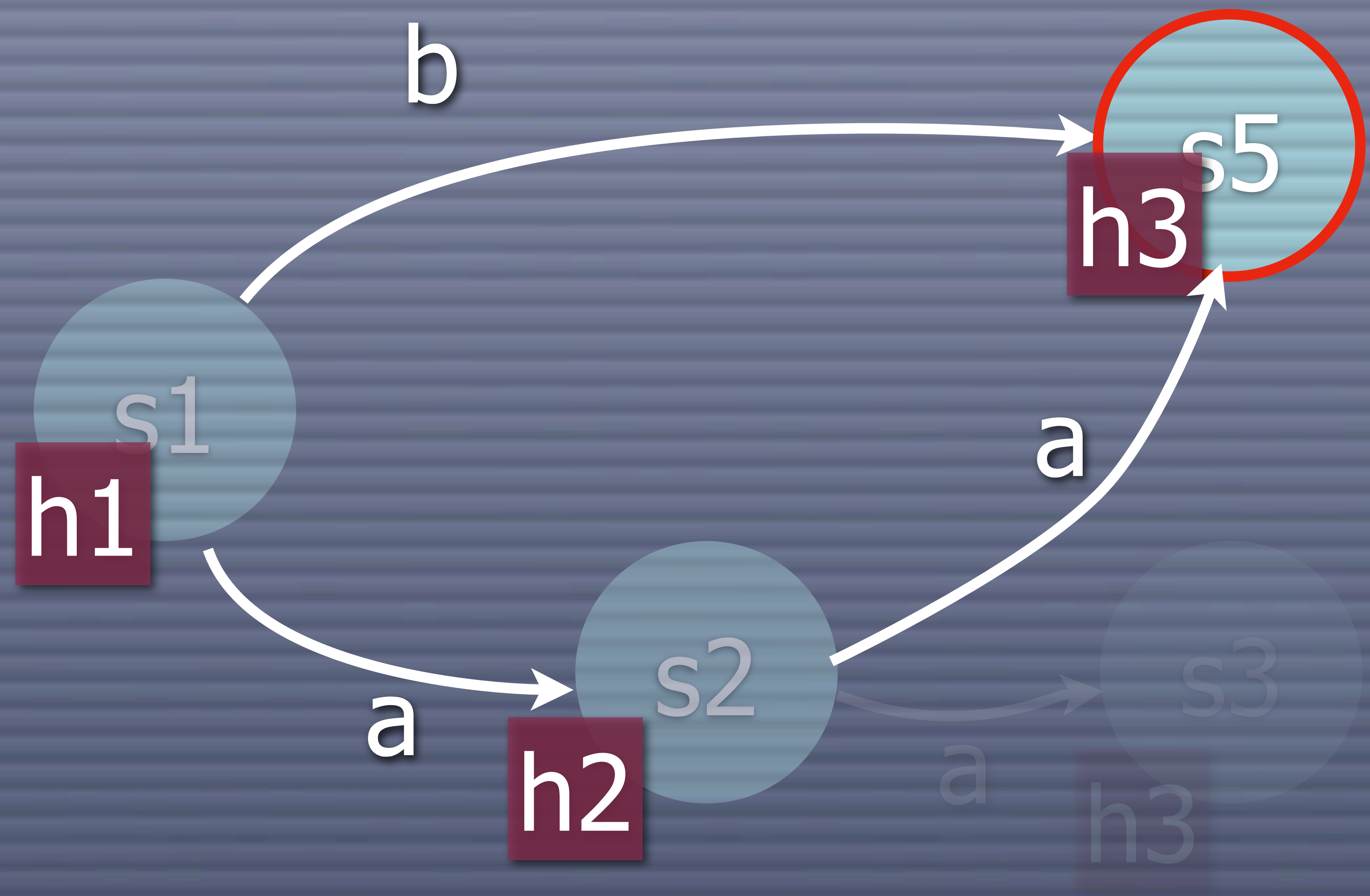
W: s5

Example



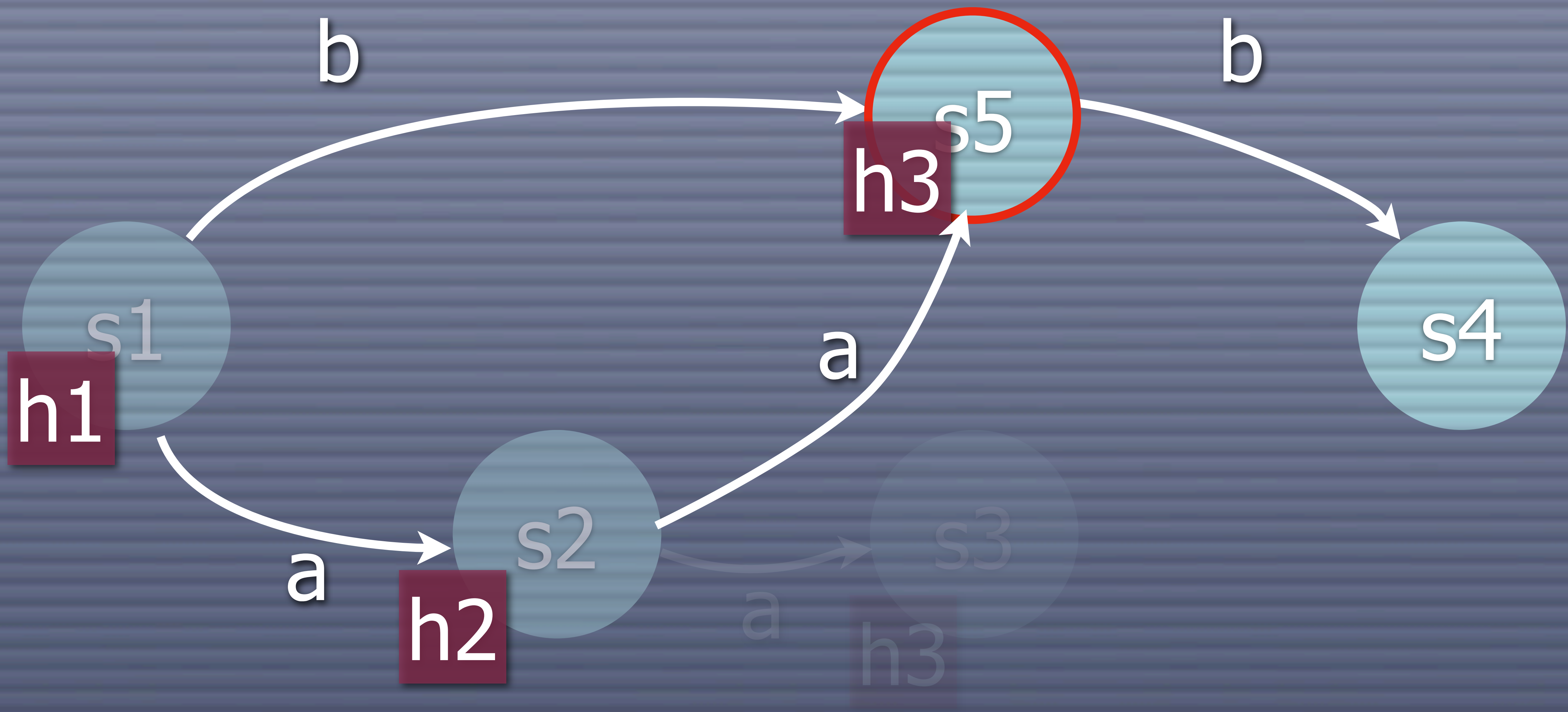
V: h1 h2 h3
W: s5

Example



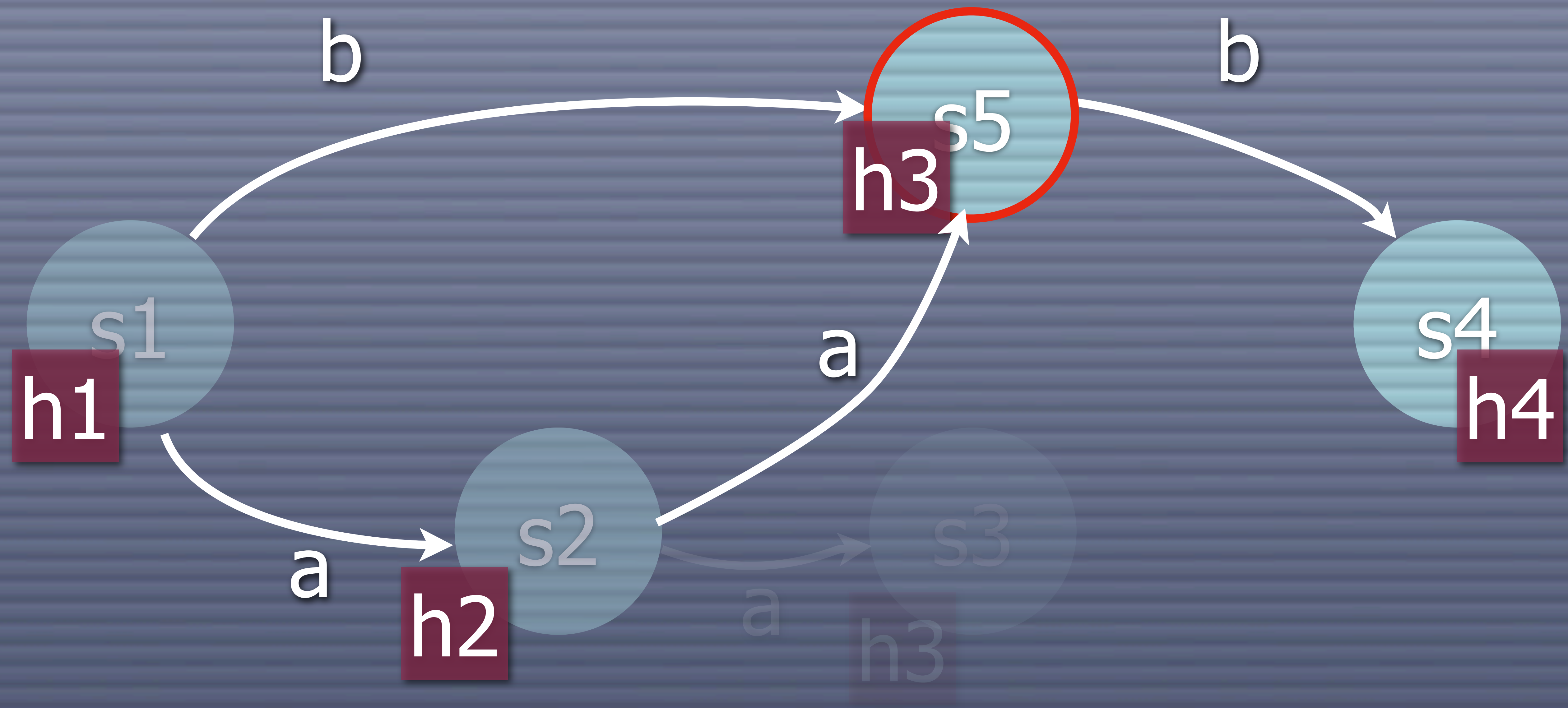
V: h1 h2 h3
W:

Example



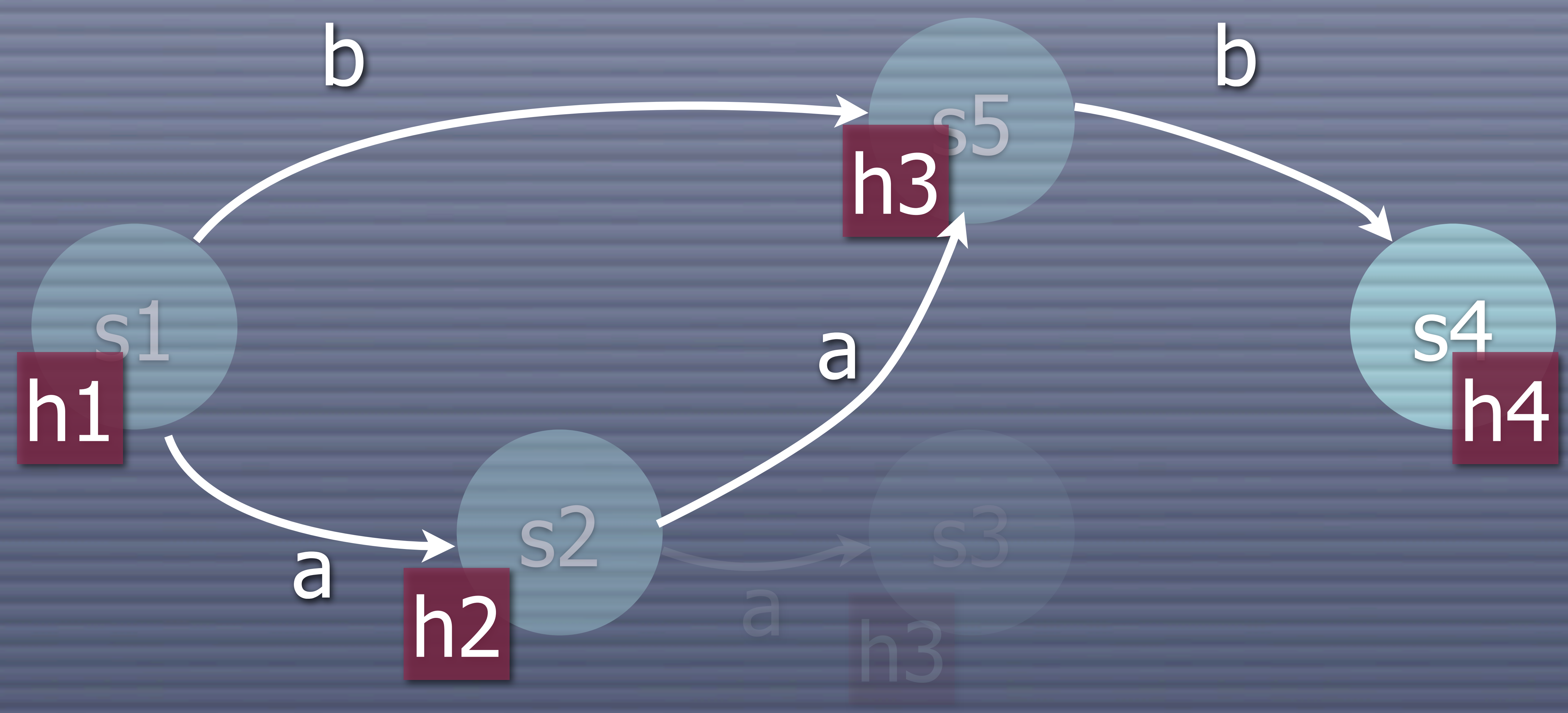
V: h1 h2 h3 h4
W: s4

Example



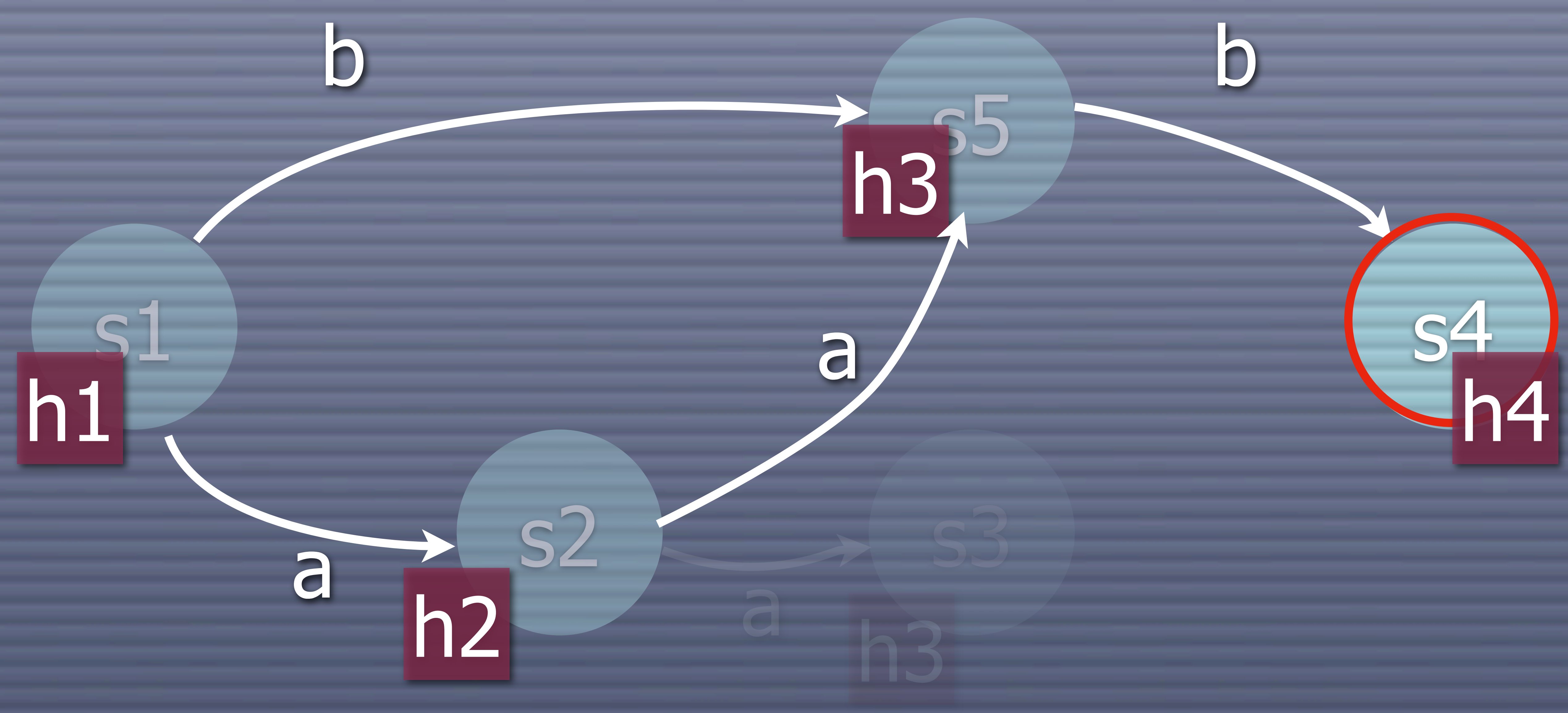
V: h1 h2 h3 h4
W: s4

Example



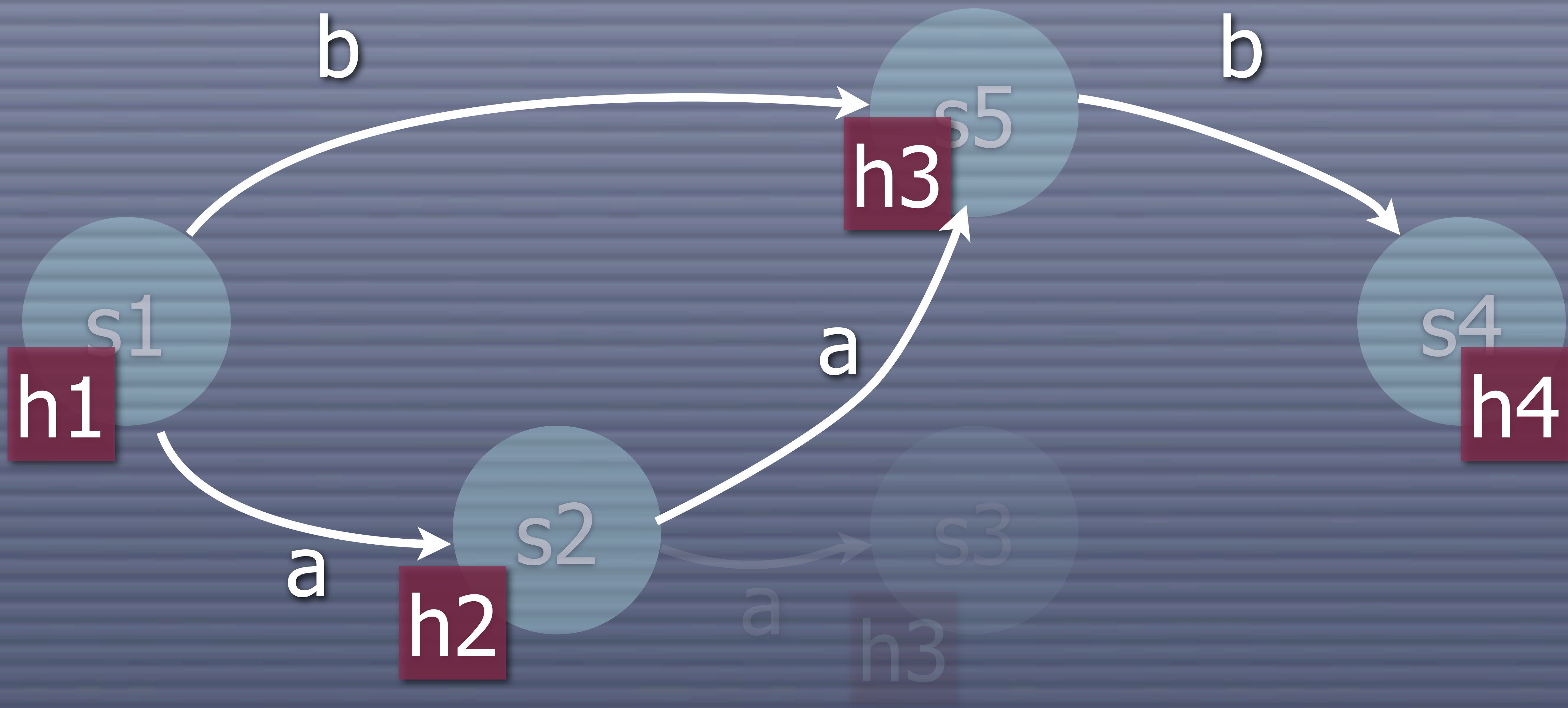
V: h1 h2 h3 h4
W: s4

Example



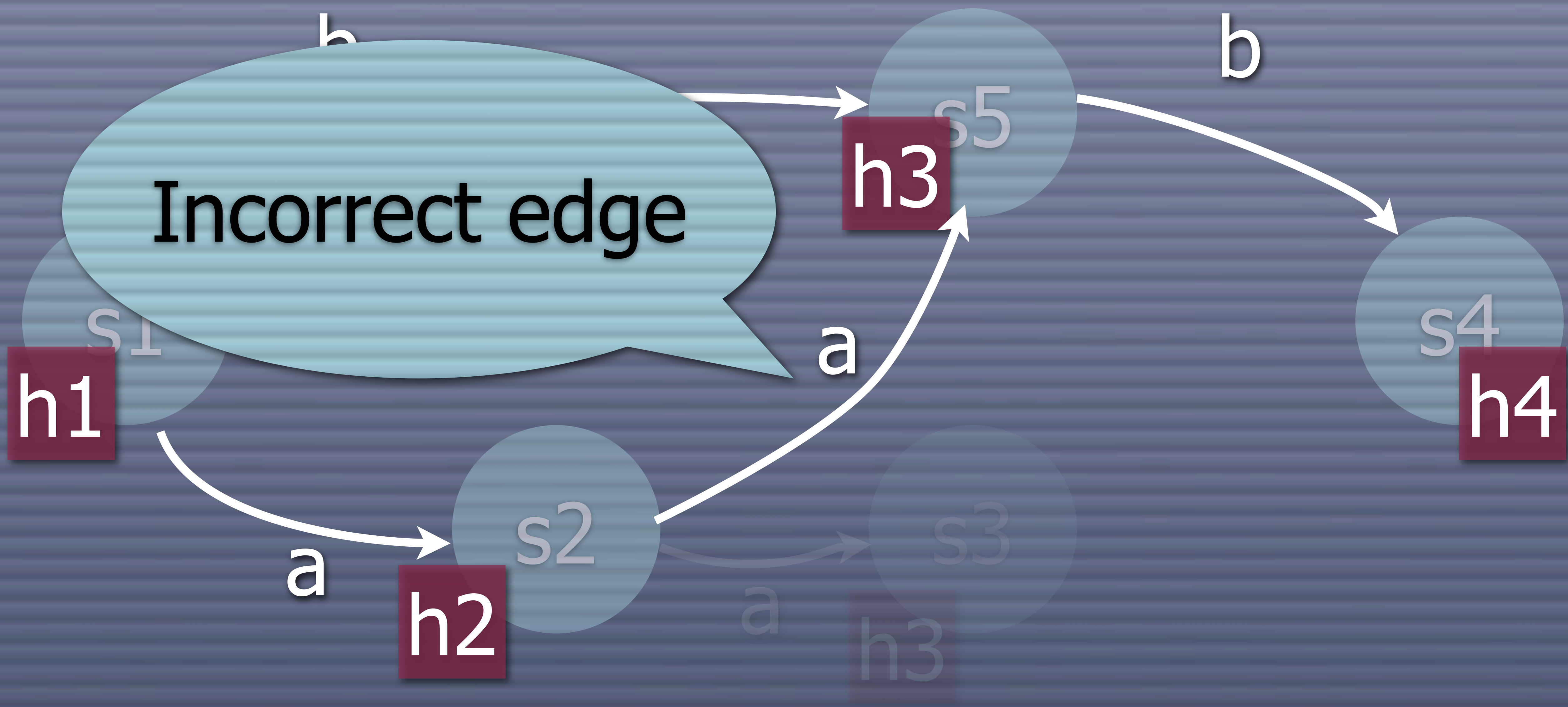
V: h1 h2 h3 h4
W:

Example



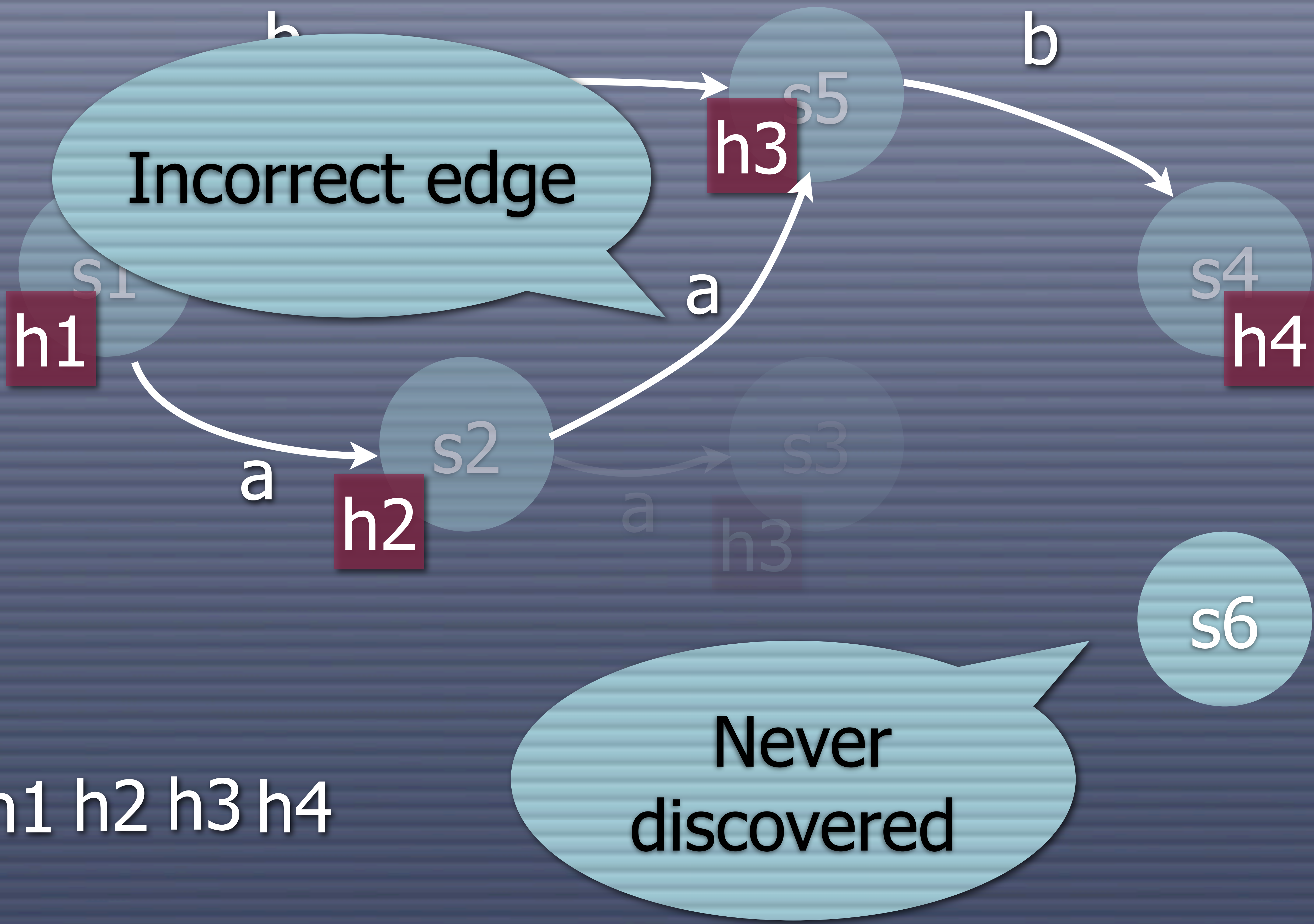
V: h1 h2 h3 h4
W:

Example



V: h1 h2 h3 h4
W:

Example



V: h1 h2 h3 h4
W:

Notes on Hash-compaction

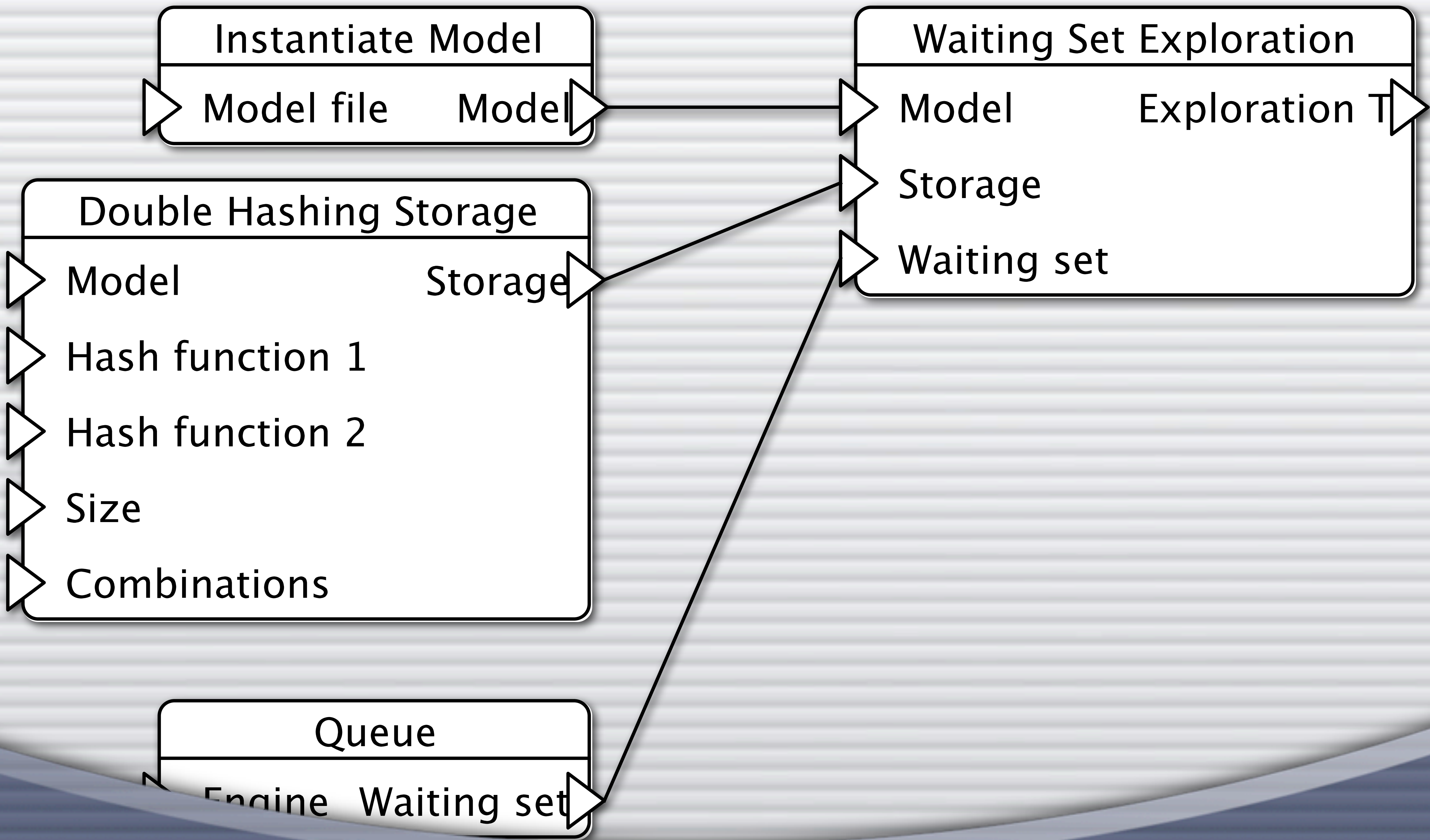
- We find most but not all states
  - Improve coverage by using larger hash values
  - Improve coverage using more than one hash function
- SHA-1 uses 160 bits (20 bytes) per state and has no known collisions
- Uses around as much time as the standard algorithm and space is still O(# nodes) but with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

Demo: Hash-compaction (10)

- Replace storage in standard method
  - We **can** but **should not** compute error traces
- Replace storage in sweep-line method – easy to combine methods

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



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** $s \rightarrow^t s'$ **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′)]en
            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** $s \rightarrow^t s'$ **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 := new bool[$2^{|h(s)|}$]; V[h(s_0)] := true
W := { s_0 }
while W ≠ ∅ **do**
    Select an s ∈ W
    W := W ∖ { s }
    if ¬I(s) **then**
        return false
    for all t, s′ **such that** s $\to^t$ s′ **do**
        if ¬V[h(s′)] **then**
            V[h(s′)] := true
            W := W ∪ { s′ }
return true

We simply set and read bits for both

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** $s \to^{t} s'$ **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

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 (11)

- Replace storage on standard example
- Try replacing storage on sweep-line example
  - JoSEL catches (most) illegal combinations on construction

Bit-state Hashing and the Sweep-line Method

- We can combine the hash-compaction method with the sweep-line method
- We cannot combine the double hashing method with the sweep-line method
  - The sweep-line method deletes states
  - We may have $h_n(s) = h_m(s')$ with $s \neq s'$
  - Thus, removing s may accidentally remove s′ as well

Bit-state Hashing and the Sweep-line Method

- The bit-state hashing/double hashing methods use a constant amount of memory regardless of number of states stored
- Can we win anything by removing entries using the sweep-line?
- We can reduce the probability of collisions

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