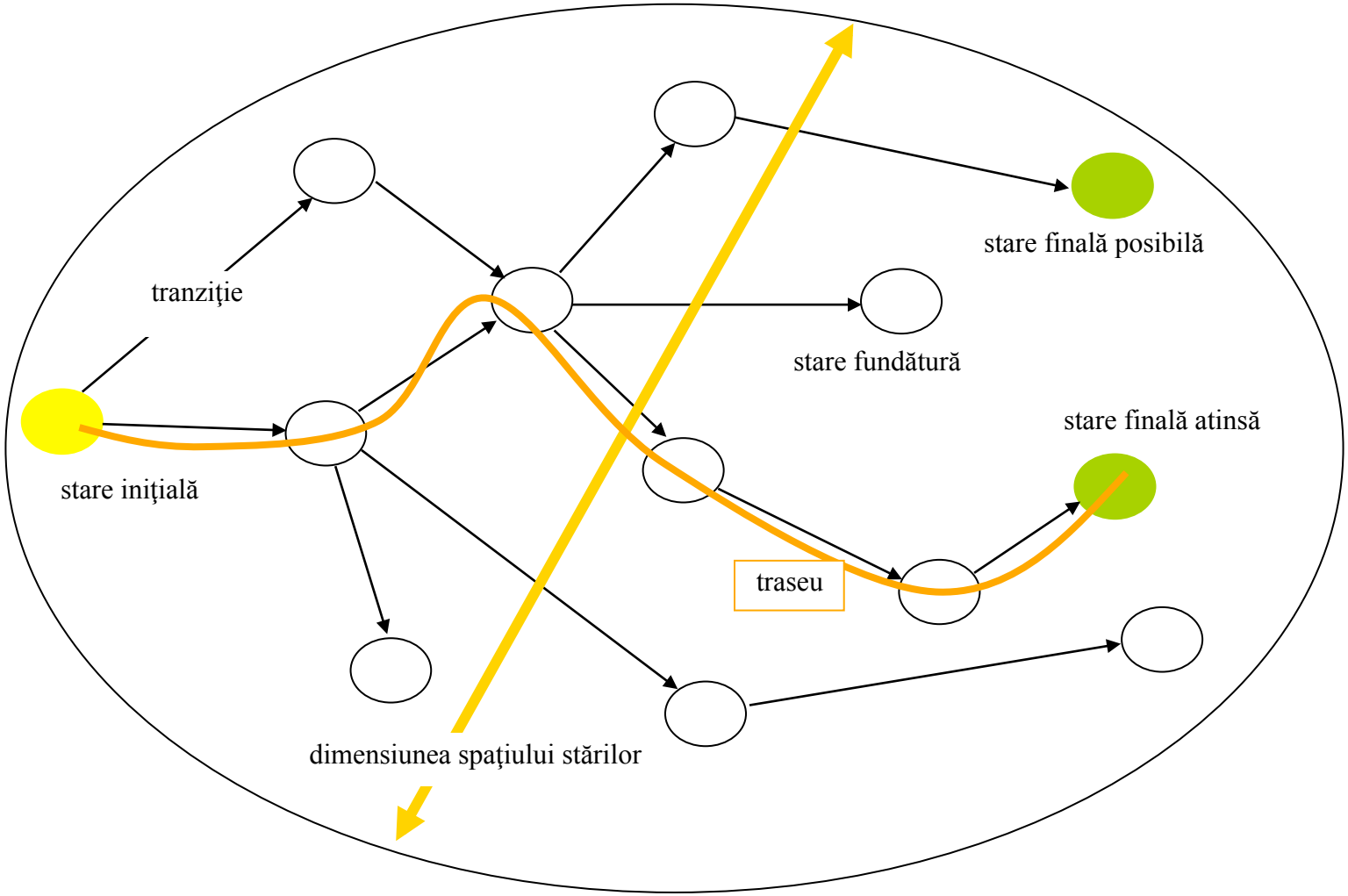
Curs 3-4
Agenda și
controlul rulării regulilor

Spațiul stărilor și căutarea soluției în SE



Filtrarea: confruntarea părților stângi ale regulilor cu faptele

- Două acțiuni:
  - Care sunt faptele din BF cu care se “potrivesc” părților stângi ale regulilor?
  - Dacă potrivirea reușește, ca efect derivat: cum se “leagă” variabilele la valori?

Execuția: tipuri de acțiuni

- Modificări ale faptelor din BF: *retract*, *assert*, *modify*
- Instrucțiuni *read*, *print*, *if*, *halt*…
- Legări de variabile
- …

Legarea variabilelor la valori

- Variabile legate explicit (la indecși de fapte)

  `?var <- (pattern…)`

- Variabile legate implicit (prin confruntarea dintre pattern-uri și fapte)

Tipuri de variabile

- Unicâmp

  `?var`

- Multicâmp

  `$?var`

Exploatarea legărilor

- Legare univocă (deterministă)
  - – un pattern se confruntă cu un unic fapt

  ➔ o singură instanță de regulă

```
(deffacts faptele-mele
   (alpha a b c)
)
(defrule rule1 ”legare neambigua”
   (alpha ?x $?)
   =>
   (printout t ”x=" ?x crlf)
)
```

Exploatarea legărilor

- Legare univocă (deterministă)
  - – un pattern se confruntă cu un unic fapt

  ➔ o singură instanță de regulă

```
(deffacts faptele-mele
   (alpha a b c)
)
(defrule rule2 ”legare neambigua”
   (alpha $? ?x)
   =>
   (printout t ”x=" ?x crlf)
)
```

Exploatarea legărilor

- Legare ambiguă (nedeterministă)
  - o regulă, un pattern
  - un fapt

  ➔ dar mai multe instanțe

```
(deffacts faptele-mele
   (alpha a b c)
)
(defrule rule3 ”legare ambigua”
   (alpha $? ?x $?)
   =>
   (printout t ”x=" ?x crlf)
)
```

Exploatarea legărilor

- Legare ambiguă (nedeterministă)
  - o regulă, mai multe pattern-uri
  - un fapt

  ➔ mai multe instanțe

  ➔ doar prima apariție a variabilei e nedeterministă

```
(deffacts faptele-mele
   (alpha a b c)
   (beta e b a)
)
(defrule rule4 ”legare ambigua”
   (alpha $? ?x $?)
   (beta $?beg ?x $?end)
   =>
   (printout t ”x= " ?x)
   (printout t ”$?beg= “ $?beg ”$?end= “ $?end crlf))
```

Exploatarea legărilor

- Legare ambiguă (nedeterministă)
  - o regulă, un pattern
  - mai multe fapte

  ➔ mai multe instanțe de reguli

```
(deffacts faptele-mele
  (alpha a b c)
  (alpha e b a)
)
(defrule rule5 ”legare ambigua”
  (alpha ?x $?)
=>
  (printout t ”x= " ?x))
```

Exploatarea legărilor

- Legare ambiguă (nedeterministă)
  - o regulă, mai multe pattern-uri
  - mai multe fapte

  ➔ mai multe instanțe de reguli

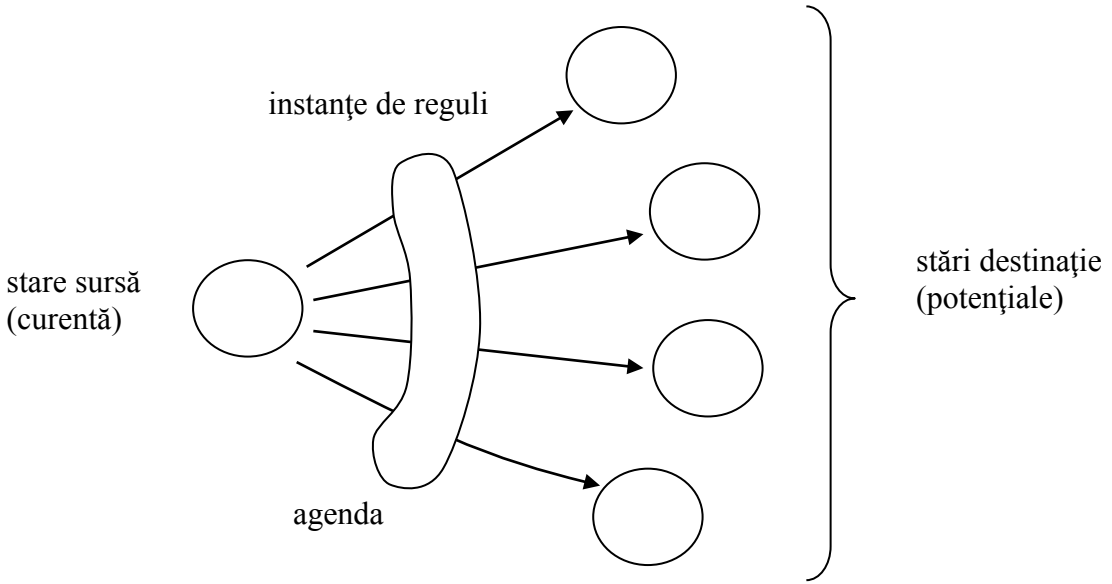
```
(deffacts faptele-mele
  (alpha a b c)
  (alpha e b a)
  (beta e b a)
)
(defrule rule6 ”legare ambigua”
  (alpha $? ?x $?)
  (beta $?beg ?x $?end)
  =>
  (printout t ”x= " ?x)
  (printout t ”$?beg= “ $?beg ”$?end= “ $?end crlf))
```

Exploatarea legărilor

- Legare ambiguă (nedeterministă)
  - exemplu: generarea mulțimii submulțimilor unei mulțimi

```
(deffacts faptele-mele
  (alpha a b c d e)
)
(defrule power-set ”genereaza multimea submultimilor”
  (alpha $?beg ?x $?end)
=>
  (assert (alpha $?beg $?end))
)
```

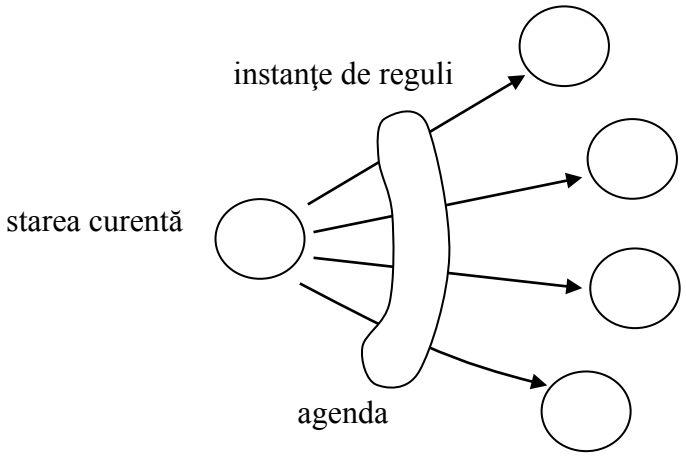
Tranziții posibile. Agenda



- La fiecare moment:
  - agenda e populată cu toate instanțele generate
  - doar cea mai prioritară e executată
  - dar ce se întâmplă cu celelalte? rămân în Agendă și se amestecă cu instanțele generate ulterior?...

Tranziții posibile. Agenda

- Agenda controlează nedeterminismul

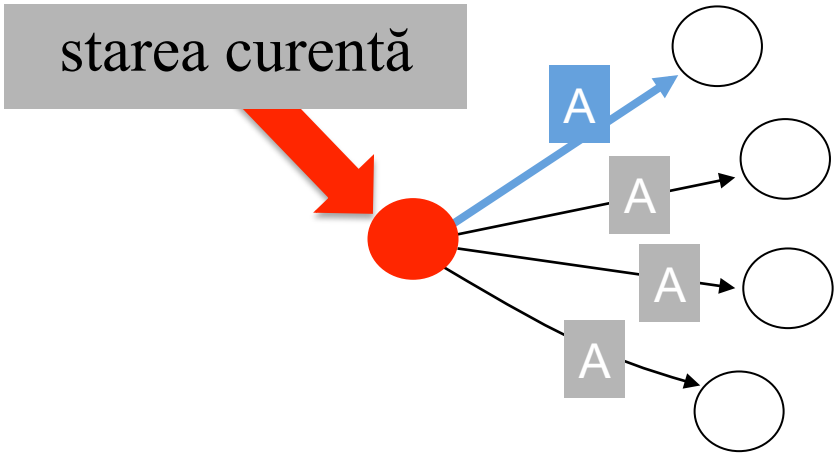



```
(deffacts faptele-mele
   (alpha a b c d)
)
(defrule A
   (alpha $? ?x $?)
=>
(printout t "A" crlf)
(assert (beta ?x))
)
```

```
(defrule B
   (beta ?x)
=>
   (printout t "B:" ?x crlf)
)
```

Agenda controlează nedeterminismul

- Control depth-first

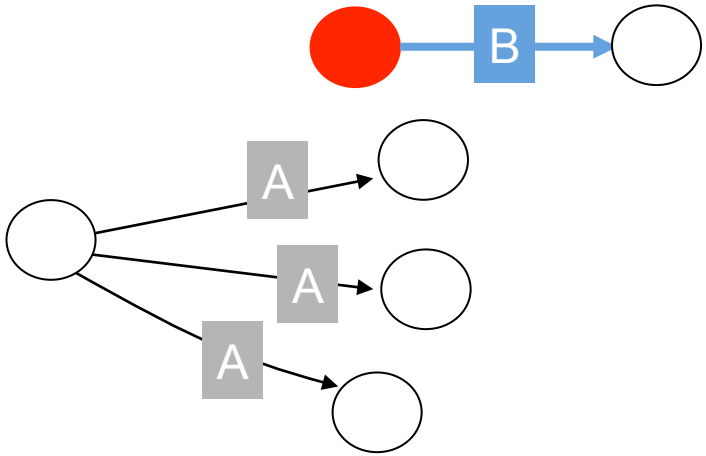



```
(deffacts faptele-mele
   (alpha a b c d)
)
(defrule A
   (alpha $? ?x $?)
=>
(printout t "A" crlf)
(assert (beta ?x))
)
```

```
(defrule B
   (beta ?x)
=>
   (printout t "B:" ?x crlf)
)
```

Agenda controlează nedeterminismul

- Control depth-first

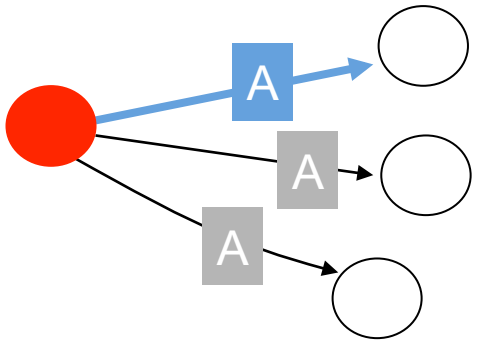



```
(deffacts faptele-mele
   (alpha a b c d)
)
(defrule A
   (alpha $? ?x $?)
=>
(printout t "A" crlf)
(assert (beta ?x))
)
```

```
(defrule B
   (beta ?x)
=>
   (printout t "B:" ?x crlf)
)
```

Agenda controlează nedeterminismul

- Control depth-first

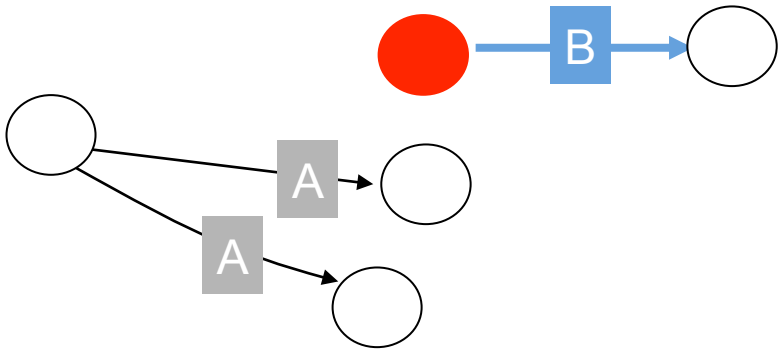



```
(deffacts faptele-mele
   (alpha a b c d)
)
(defrule A
   (alpha $? ?x $?)
=>
(printout t "A" crlf)
(assert (beta ?x))
)
```

```
(defrule B
   (beta ?x)
=>
   (printout t "B:" ?x crlf)
)
```

Agenda controlează nedeterminismul

- Control depth-first

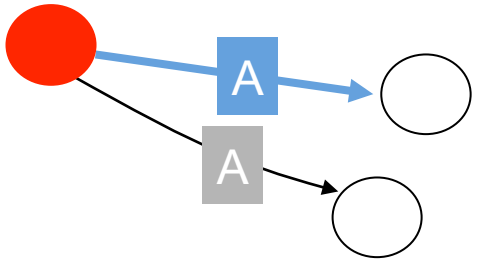



```
(deffacts faptele-mele
   (alpha a b c d)
)
(defrule A
   (alpha $? ?x $?)
=>
(printout t "A" crlf)
(assert (beta ?x))
)
```

```
(defrule B
   (beta ?x)
=>
   (printout t "B:" ?x crlf)
)
```

Agenda controlează nedeterminismul

- Control depth-first

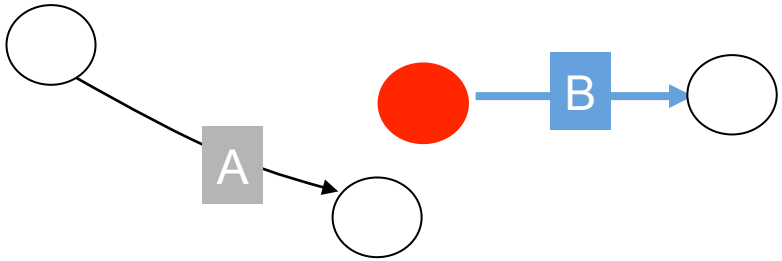



```
(deffacts faptele-mele
   (alpha a b c d)
)
(defrule A
   (alpha $? ?x $?)
=>
(printout t "A" crlf)
(assert (beta ?x))
)
```

```
(defrule B
   (beta ?x)
=>
   (printout t "B:" ?x crlf)
)
```

Agenda controlează nedeterminismul

- Control depth-first

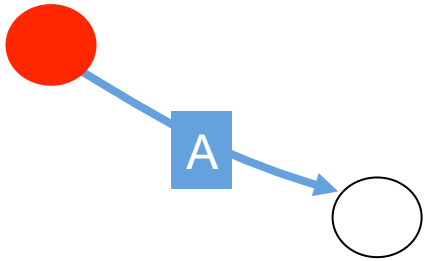



```
(deffacts faptele-mele
   (alpha a b c d)
)
(defrule A
   (alpha $? ?x $?)
=>
(printout t "A" crlf)
(assert (beta ?x))
)
```

```
(defrule B
   (beta ?x)
=>
   (printout t "B:" ?x crlf)
)
```

Agenda controlează nedeterminismul

- Control depth-first



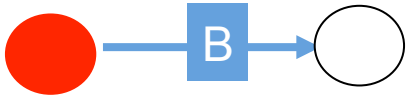

```
(deffacts faptele-mele
   (alpha a b c d)
)
(defrule A
   (alpha $? ?x $?)
=>
(printout t "A" crlf)
(assert (beta ?x))
)
```

```
(defrule B
   (beta ?x)
=>
   (printout t "B:" ?x crlf)
)
```

Agenda controlează nedeterminismul

- Control depth-first

O execuție de tip backtracking…

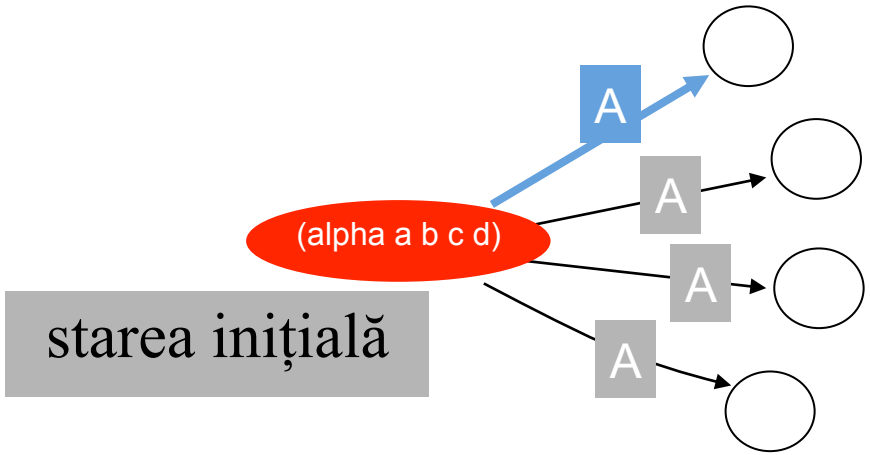



```
(deffacts faptele-mele
   (alpha a b c d)
)
(defrule A
   (alpha $? ?x $?)
=>
(printout t "A" crlf)
(assert (beta ?x))
)
```

```
(defrule B
   (beta ?x)
=>
   (printout t "B:" ?x crlf)
)
```

Și totuși…

- Lucrurile nu stau chiar așa

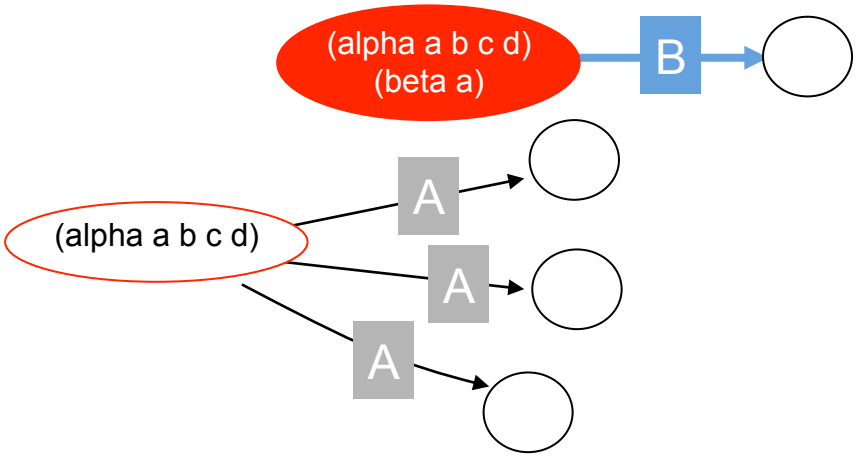



```
(deffacts faptele-mele
   (alpha a b c d)
)
(defrule A
   (alpha $? ?x $?)
=>
(printout t "A" crlf)
(assert (beta ?x))
)
```

```
(defrule B
   (beta ?x)
=>
   (printout t "B:" ?x crlf)
)
```

Și totuși…

- Lucrurile nu stau chiar așa

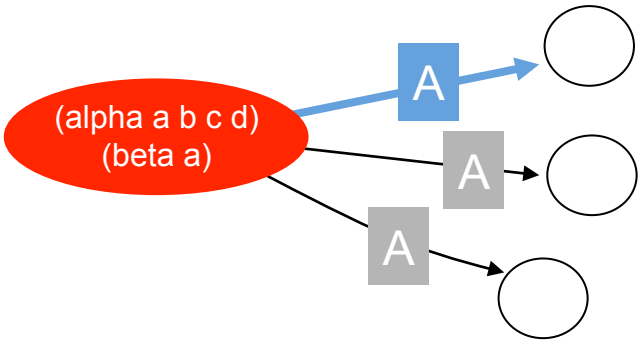



```
(deffacts faptele-mele
   (alpha a b c d)
)
(defrule A
   (alpha $? ?x $?)
=>
(printout t "A" crlf)
(assert (beta ?x))
)
```

```
(defrule B
   (beta ?x)
=>
   (printout t "B:" ?x crlf)
)
```

Și totuși…

- Lucrurile nu stau chiar așa

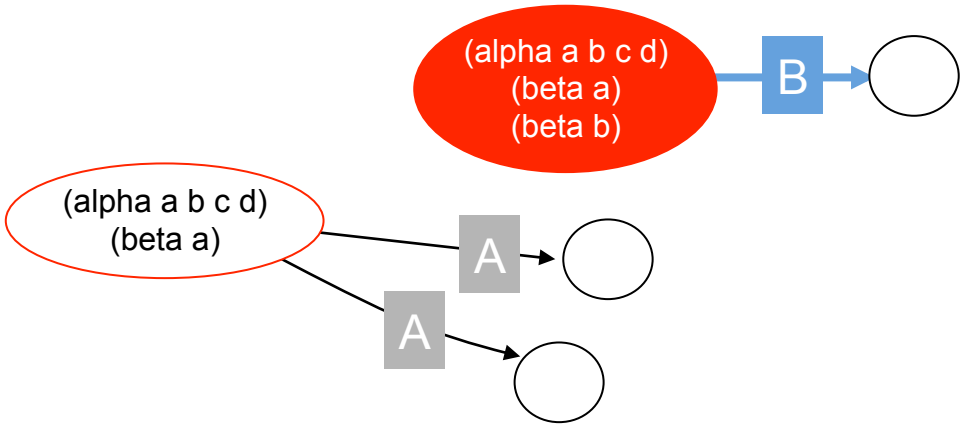



```
(deffacts faptele-mele
   (alpha a b c d)
)
(defrule A
   (alpha $? ?x $?)
=>
(printout t "A" crlf)
(assert (beta ?x))
)
```

```
(defrule B
   (beta ?x)
=>
   (printout t "B:" ?x crlf)
)
```

Și totuși…

- Lucrurile nu stau chiar așa

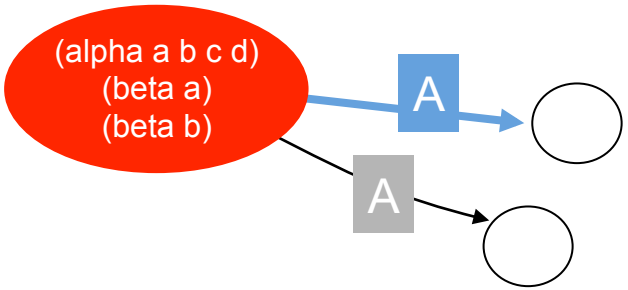



```
(deffacts faptele-mele
   (alpha a b c d)
)
(defrule A
   (alpha $? ?x $?)
=>
(printout t "A" crlf)
(assert (beta ?x))
)
```

```
(defrule B
   (beta ?x)
=>
   (printout t "B:" ?x crlf)
)
```

Și totuși…

- Lucrurile nu stau chiar așa

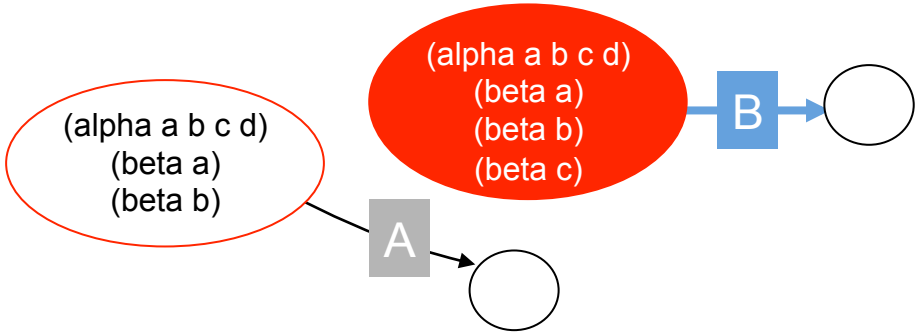



```
(deffacts faptele-mele
   (alpha a b c d)
)
(defrule A
   (alpha $? ?x $?)
=>
(printout t "A" crlf)
(assert (beta ?x))
)
```

```
(defrule B
   (beta ?x)
=>
   (printout t "B:" ?x crlf)
)
```

Și totuși…

- Lucrurile nu stau chiar așa

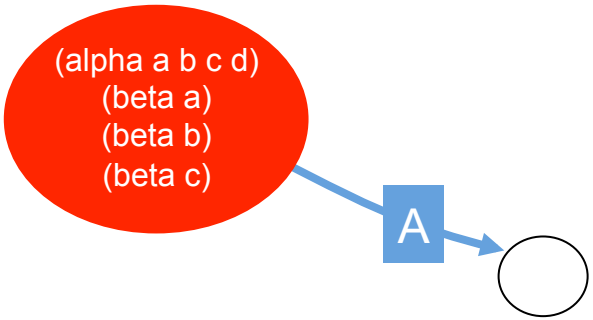



```
(deffacts faptele-mele
   (alpha a b c d)
)
(defrule A
   (alpha $? ?x $?)
=>
(printout t "A" crlf)
(assert (beta ?x))
)
```

```
(defrule B
   (beta ?x)
=>
   (printout t "B:" ?x crlf)
)
```

Și totuși…

- Lucrurile nu stau chiar așa



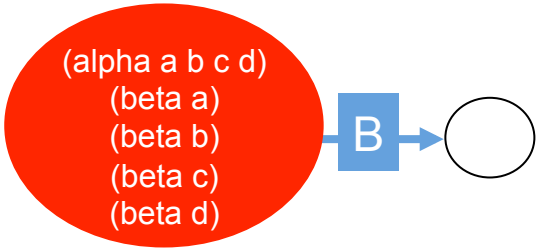

```
(deffacts faptele-mele
   (alpha a b c d)
)
(defrule A
   (alpha $? ?x $?)
=>
(printout t "A" crlf)
(assert (beta ?x))
)
```

```
(defrule B
   (beta ?x)
=>
   (printout t "B:" ?x crlf)
)
```

Și totuși…

- Lucrurile nu stau chiar așa

Altceva decât backtracking…

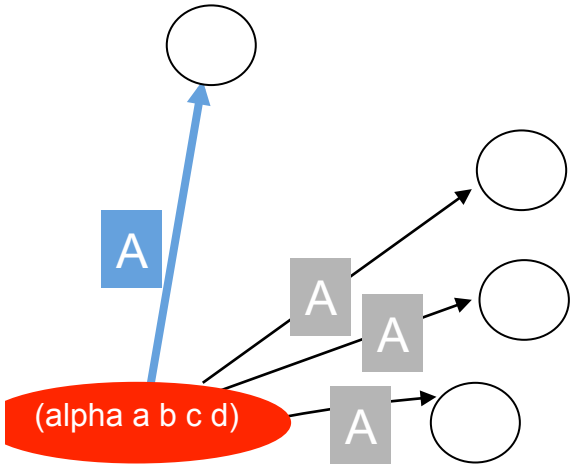



```
(deffacts faptele-mele
   (alpha a b c d)
)
(defrule A
   (alpha $? ?x $?)
=>
(printout t "A" crlf)
(assert (beta ?x))
)
```

```
(defrule B
   (beta ?x)
=>
   (printout t "B:" ?x crlf)
)
```

Stările nu rămân aceleași

- Niciodată nu se revine într-o stare anterioară

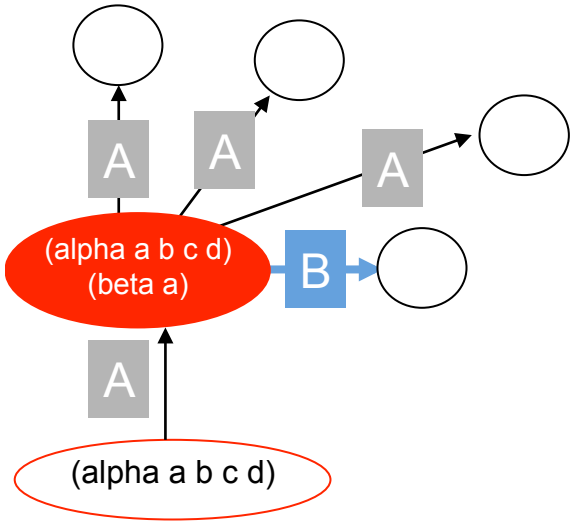



```
(deffacts faptele-mele
   (alpha a b c d)
)
(defrule A
   (alpha $? ?x $?)
=>
(printout t "A" crlf)
(assert (beta ?x))
)
```

```
(defrule B
   (beta ?x)
=>
   (printout t "B:" ?x crlf)
)
```

Stările nu rămân aceleași

- Niciodată nu se revine într-o stare anterioară

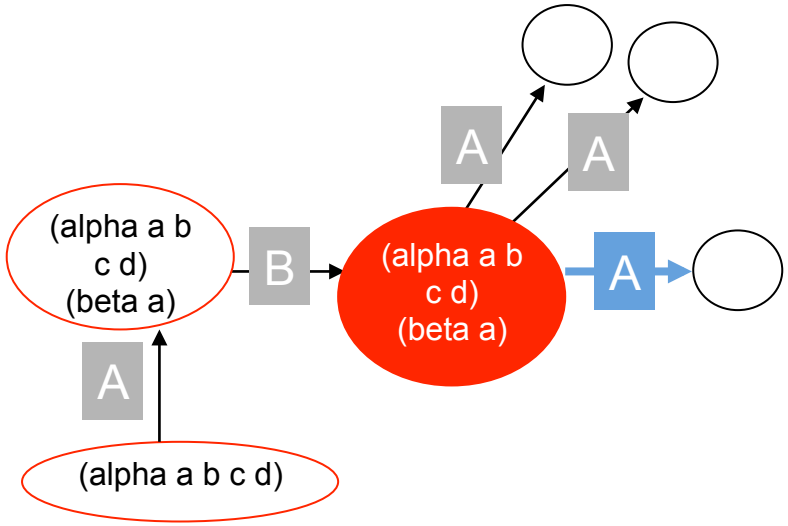



```
(deffacts faptele-mele
   (alpha a b c d)
)
(defrule A
   (alpha $? ?x $?)
=>
(printout t "A" crlf)
(assert (beta ?x))
)
```

```
(defrule B
   (beta ?x)
=>
   (printout t "B:" ?x crlf)
)
```

Stările nu rămân aceleași

- Niciodată nu se revine într-o stare anterioară

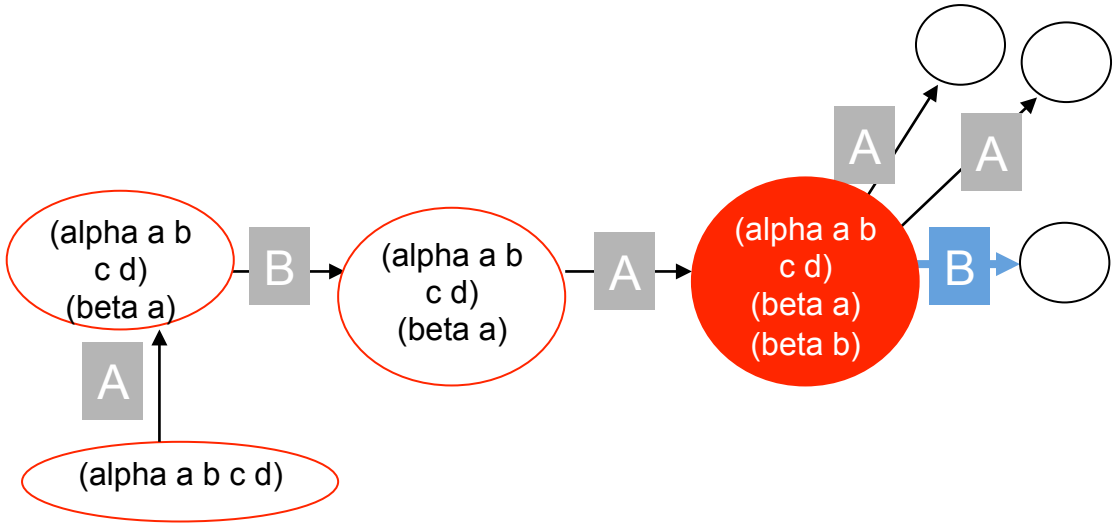



```
(deffacts faptele-mele
   (alpha a b c d)
)
(defrule A
   (alpha $? ?x $?)
=>
(printout t "A" crlf)
(assert (beta ?x))
)
```

```
(defrule B
   (beta ?x)
=>
   (printout t "B:" ?x crlf)
)
```

Stările nu rămân aceleași

- Niciodată nu se revine într-o stare anterioară

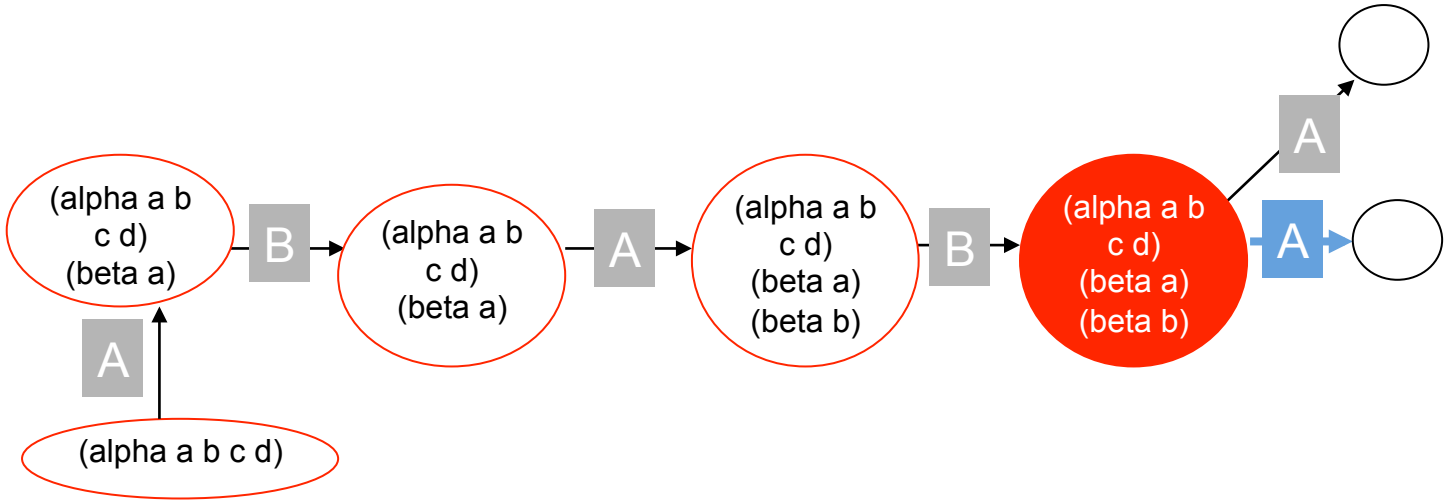



```
(deffacts faptele-mele
   (alpha a b c d)
)
(defrule A
   (alpha $? ?x $?)
=>
(printout t "A" crlf)
(assert (beta ?x))
)
```

```
(defrule B
   (beta ?x)
=>
   (printout t "B:" ?x crlf)
)
```

Stările nu rămân aceleași

- Niciodată nu se revine într-o stare anterioară

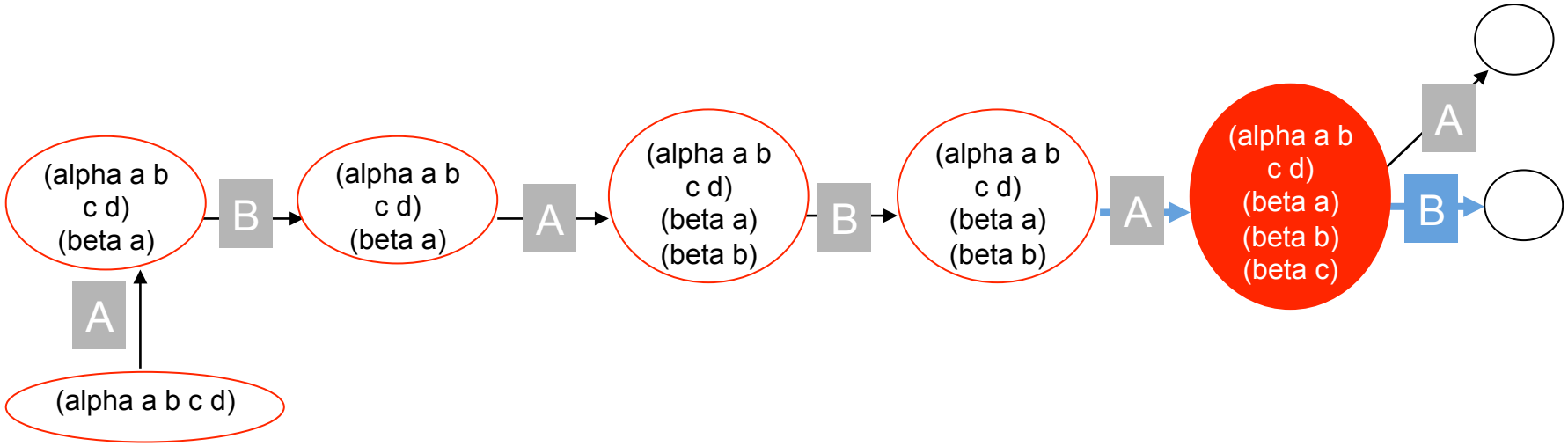



```
(deffacts faptele-mele
   (alpha a b c d)
)
(defrule A
   (alpha $? ?x $?)
=>
(printout t "A" crlf)
(assert (beta ?x))
)
```

```
(defrule B
   (beta ?x)
=>
   (printout t "B:" ?x crlf)
)
```

Stările nu rămân aceleași

- Niciodată nu se revine într-o stare anterioară

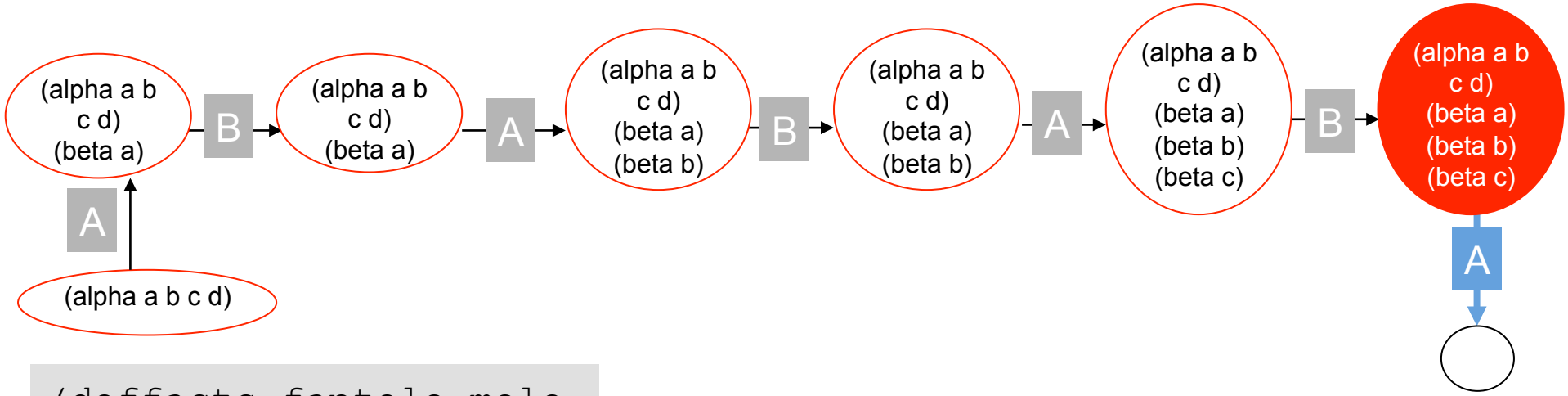



```
(deffacts faptele-mele
   (alpha a b c d)
)
(defrule A
   (alpha $? ?x $?)
=>
(printout t "A" crlf)
(assert (beta ?x))
)
```

```
(defrule B
   (beta ?x)
=>
   (printout t "B:" ?x crlf)
)
```

Stările nu rămân aceleași

- Niciodată nu se revine într-o stare anterioară

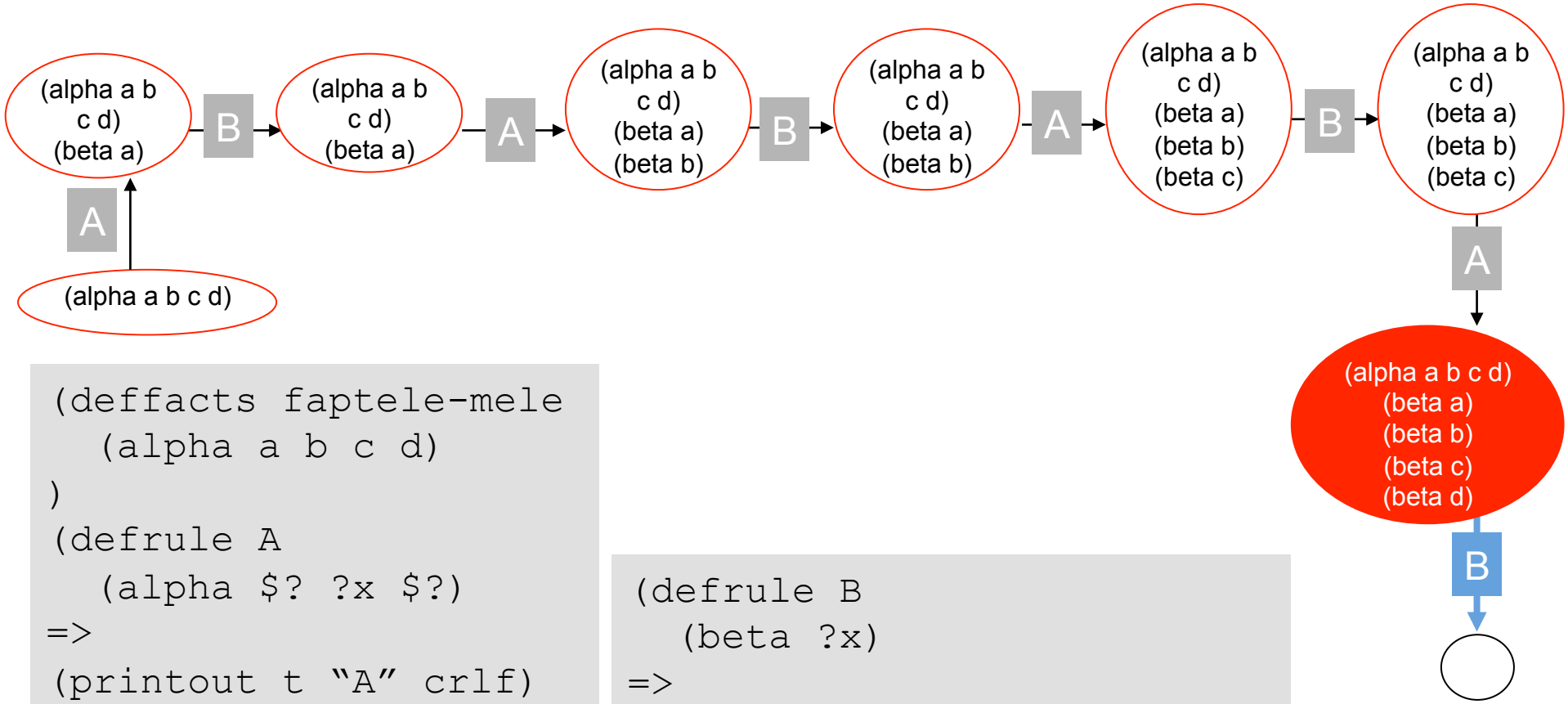



```
(deffacts faptele-mele
   (alpha a b c d)
)
(defrule A
   (alpha $? ?x $?)
=>
(printout t "A" crlf)
(assert (beta ?x))
)
```

```
(defrule B
   (beta ?x)
=>
   (printout t "B:" ?x crlf)
)
```

Stările nu rămân aceleași

- Niciodată nu se revine într-o stare anterioară

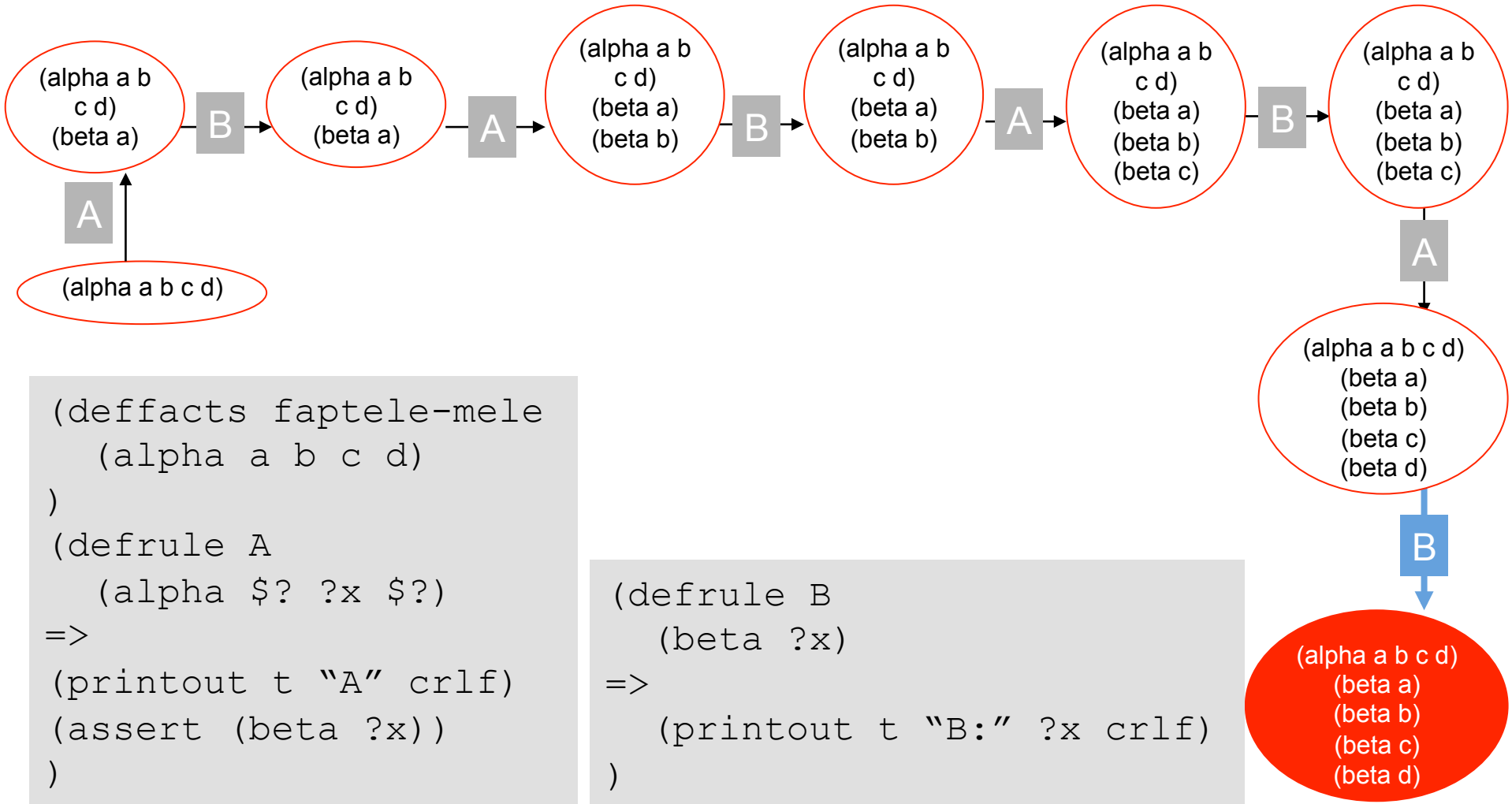



```
(deffacts faptele-mele
   (alpha a b c d)
)
(defrule A
   (alpha $? ?x $?)
=>
(printout t "A" crlf)
(assert (beta ?x))
)
```

```
(defrule B
   (beta ?x)
=>
   (printout t "B:" ?x crlf)
)
```

Stările nu rămân aceleași

- Niciodată nu se revine într-o stare anterioară




```
(deffacts faptele-mele
   (alpha a b c d)
)
(defrule A
   (alpha $? ?x $?)
=>
(printout t "A" crlf)
(assert (beta ?x))
)
```

```
(defrule B
   (beta ?x)
=>
   (printout t "B:" ?x crlf)
)
```

Grafuri Și-SAU

R1: **dacă** A, **atunci** E

R2: **dacă** B, E, **atunci** F

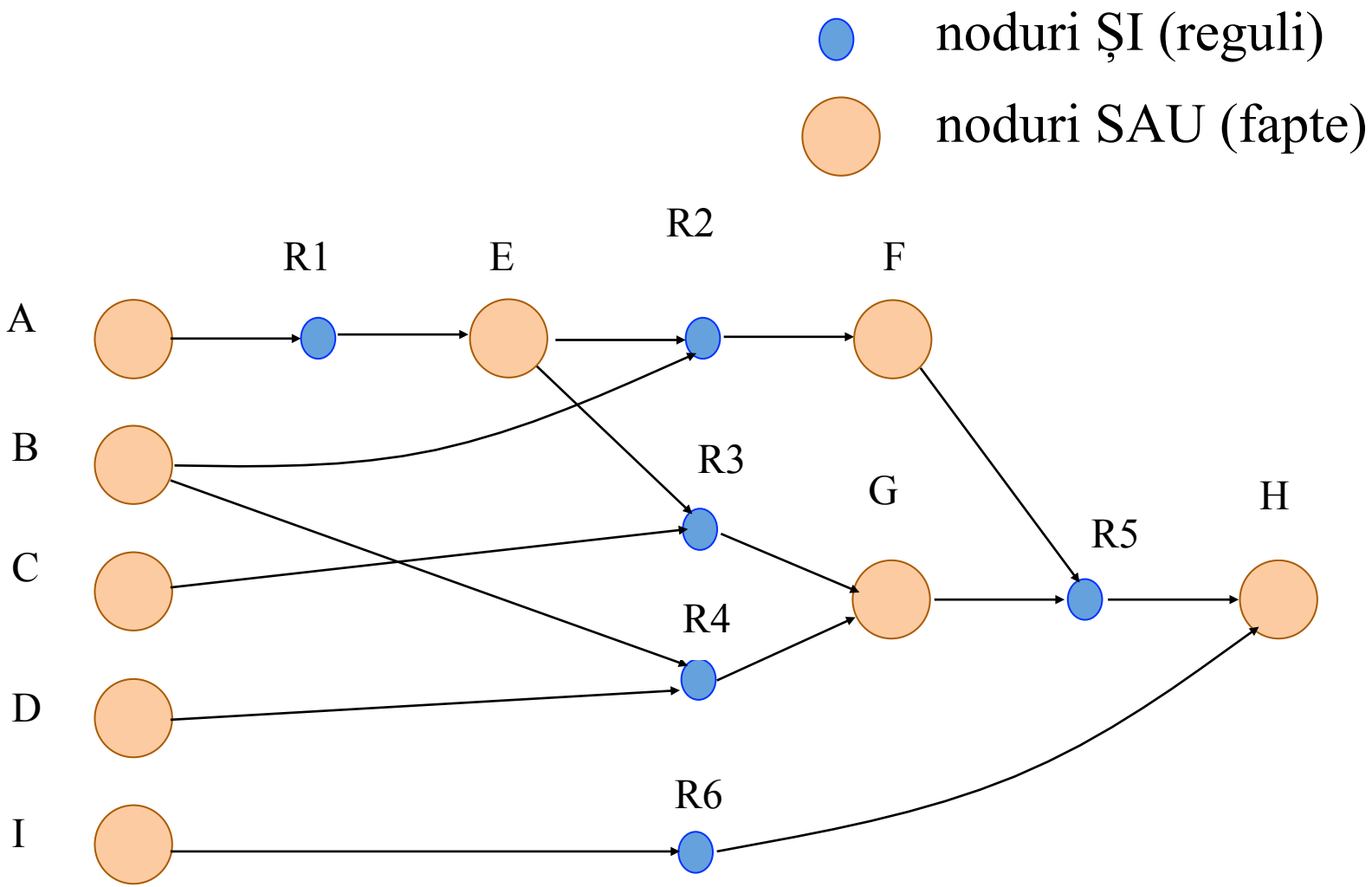
R3: **dacă** C, E, **atunci** G

R4: **dacă** B, D, **atunci** G

R5: **dacă** F, G, **atunci** H

R6: **dacă** I, **atunci** H

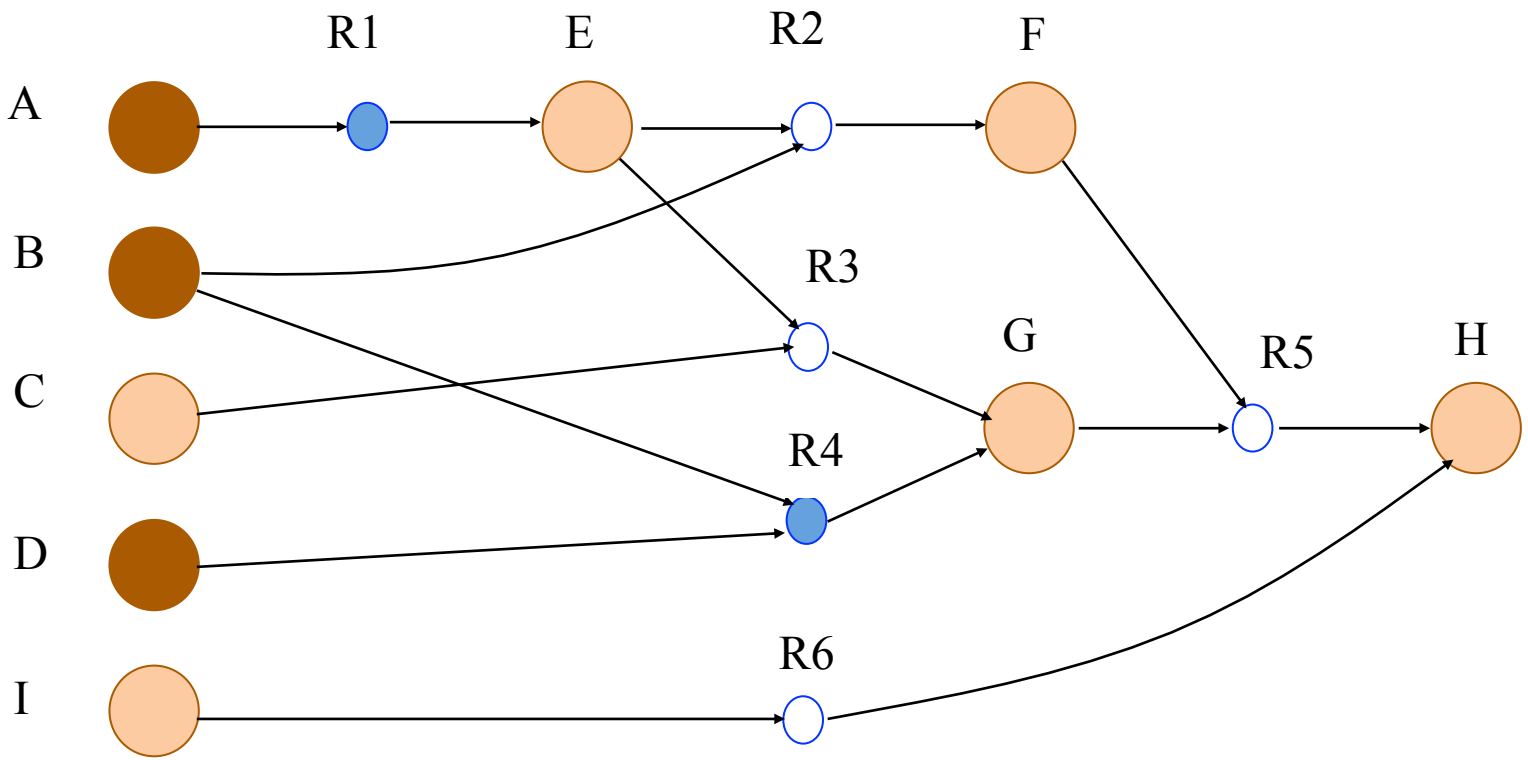
Un sistem de reguli poate fi reprezentat ca un graf ȘI-SAU



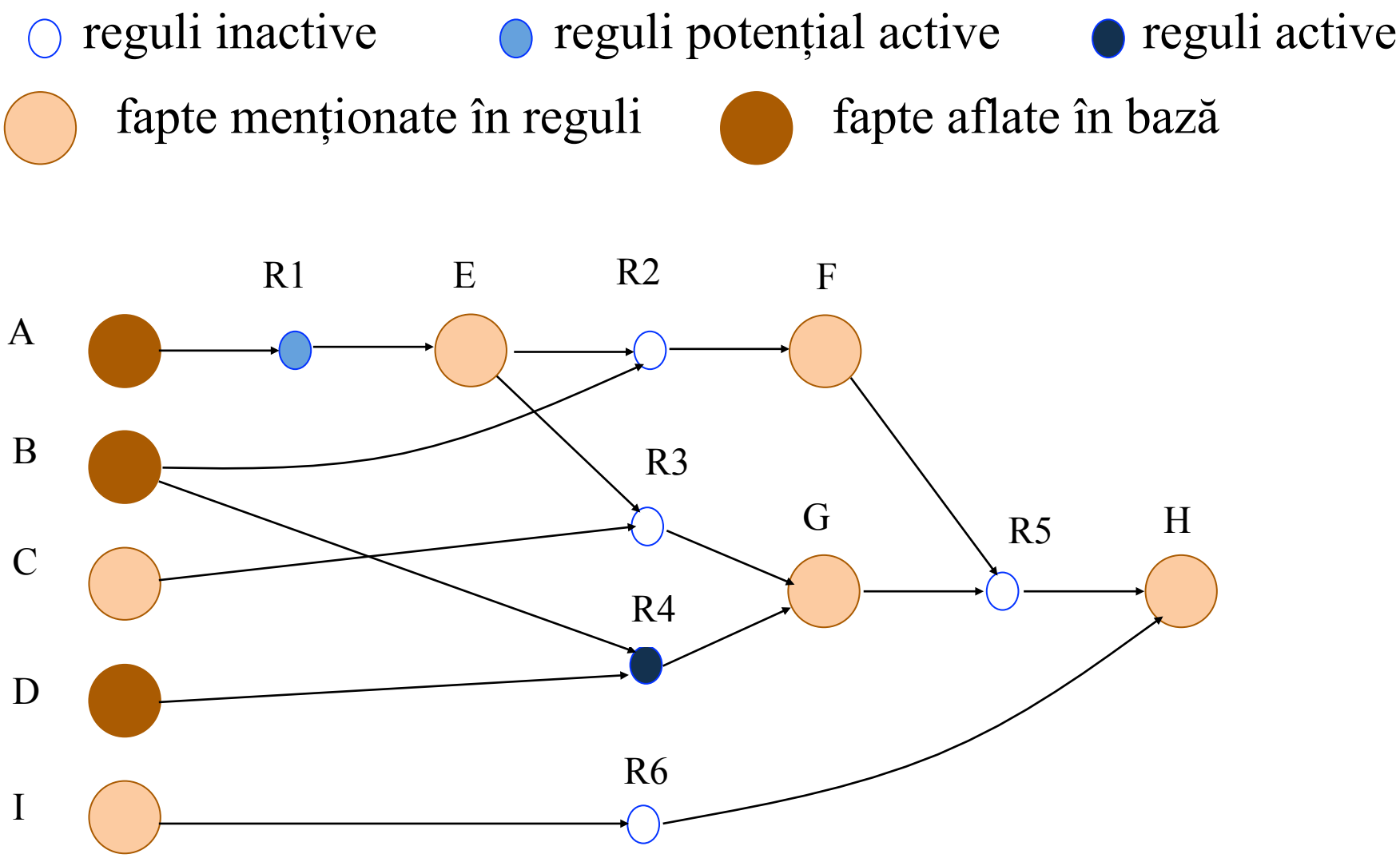
O rulare posibilă

reguli inactive reguli potențial active reguli active

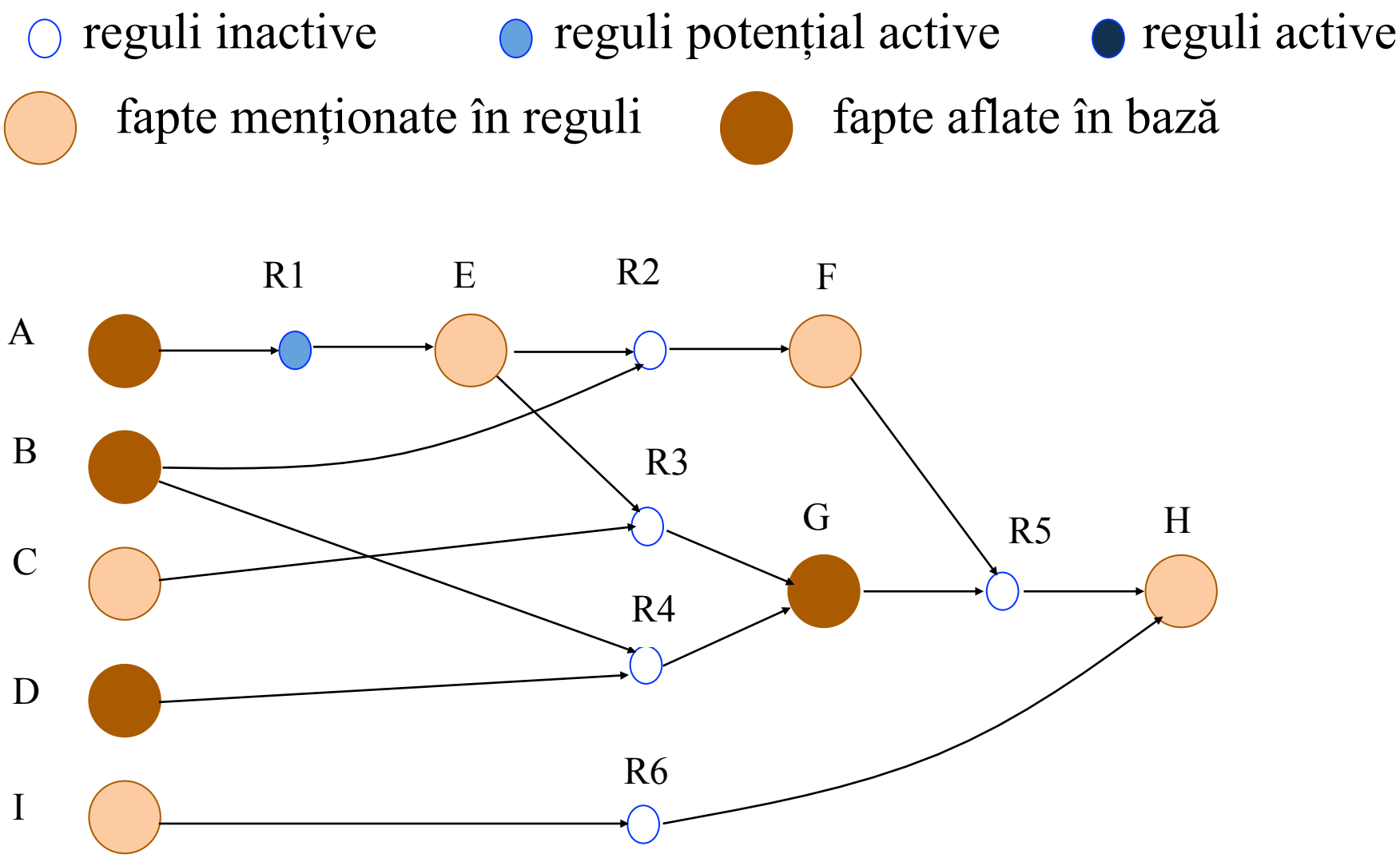
fapte menționate în reguli fapte aflate în bază



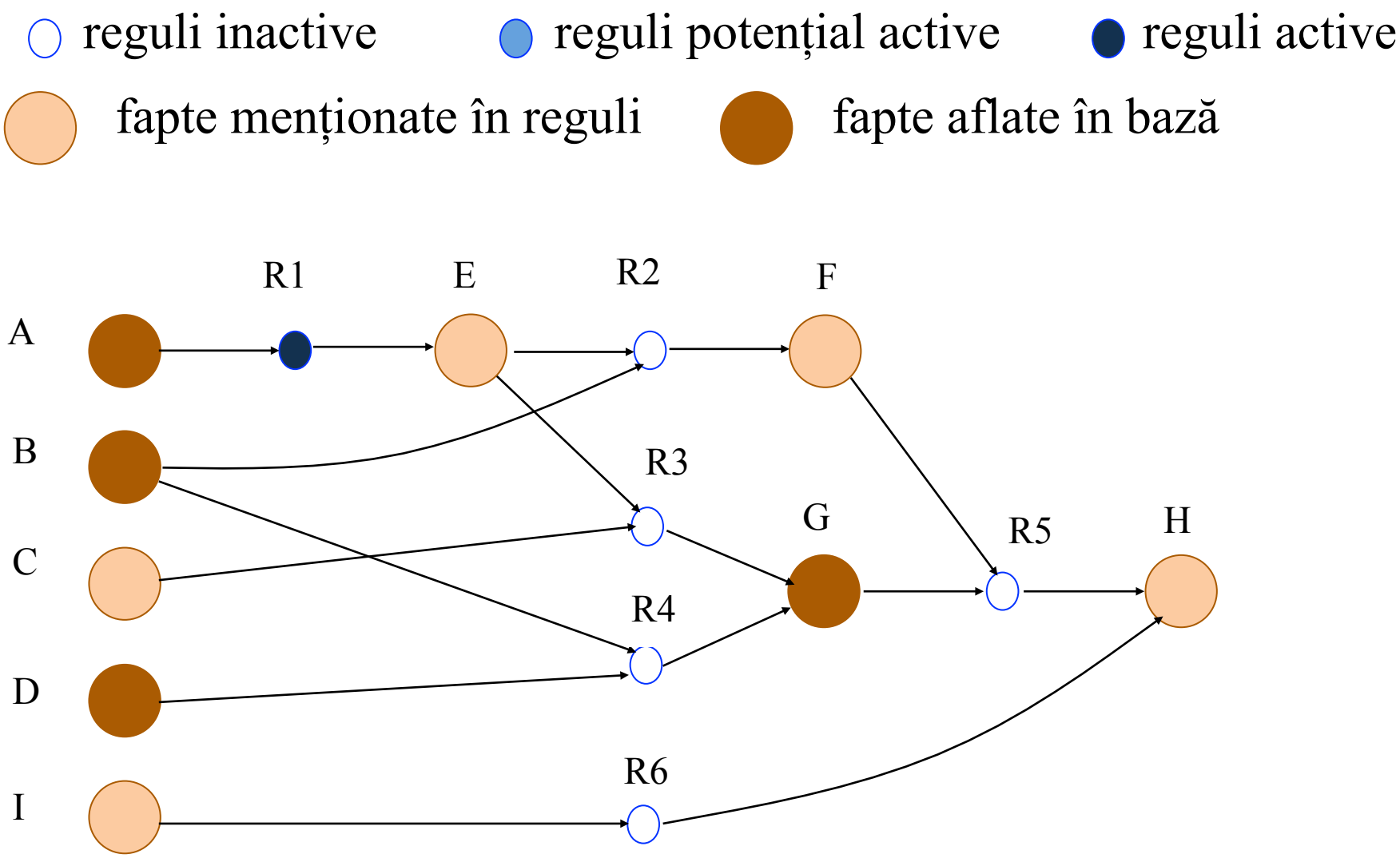
O rulare posibilă



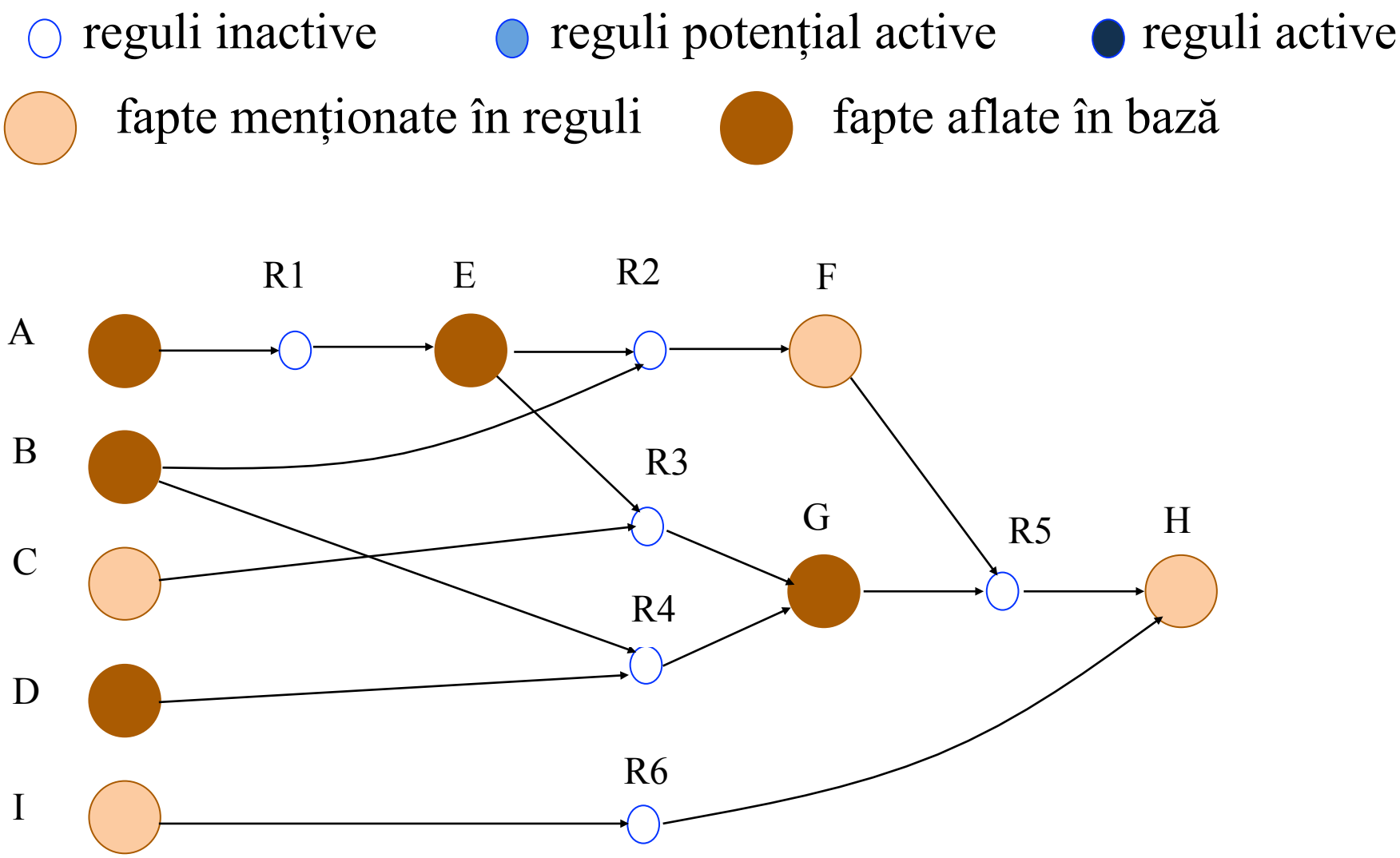
O rulare posibilă



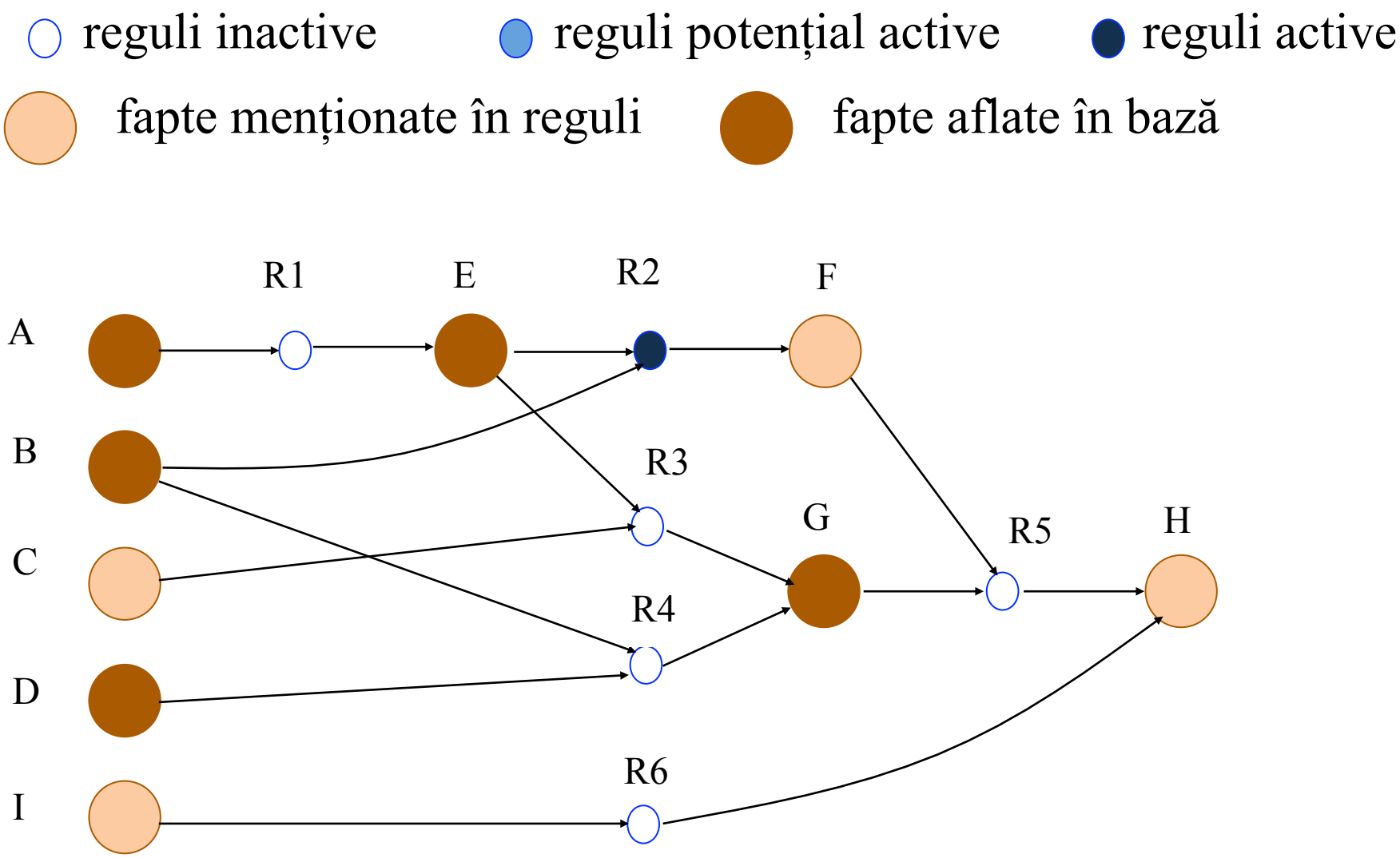
O rulare posibilă



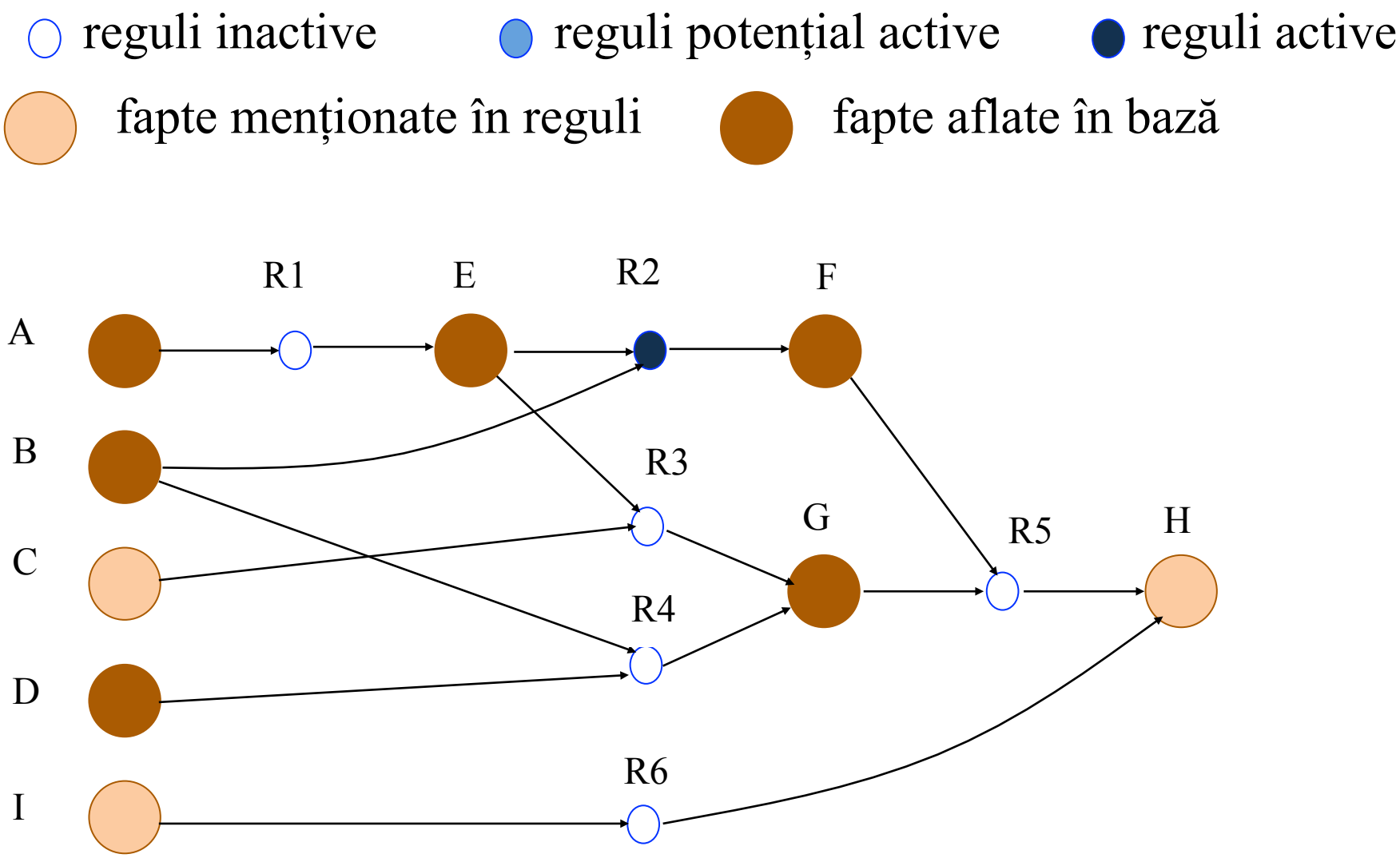
O rulare posibilă



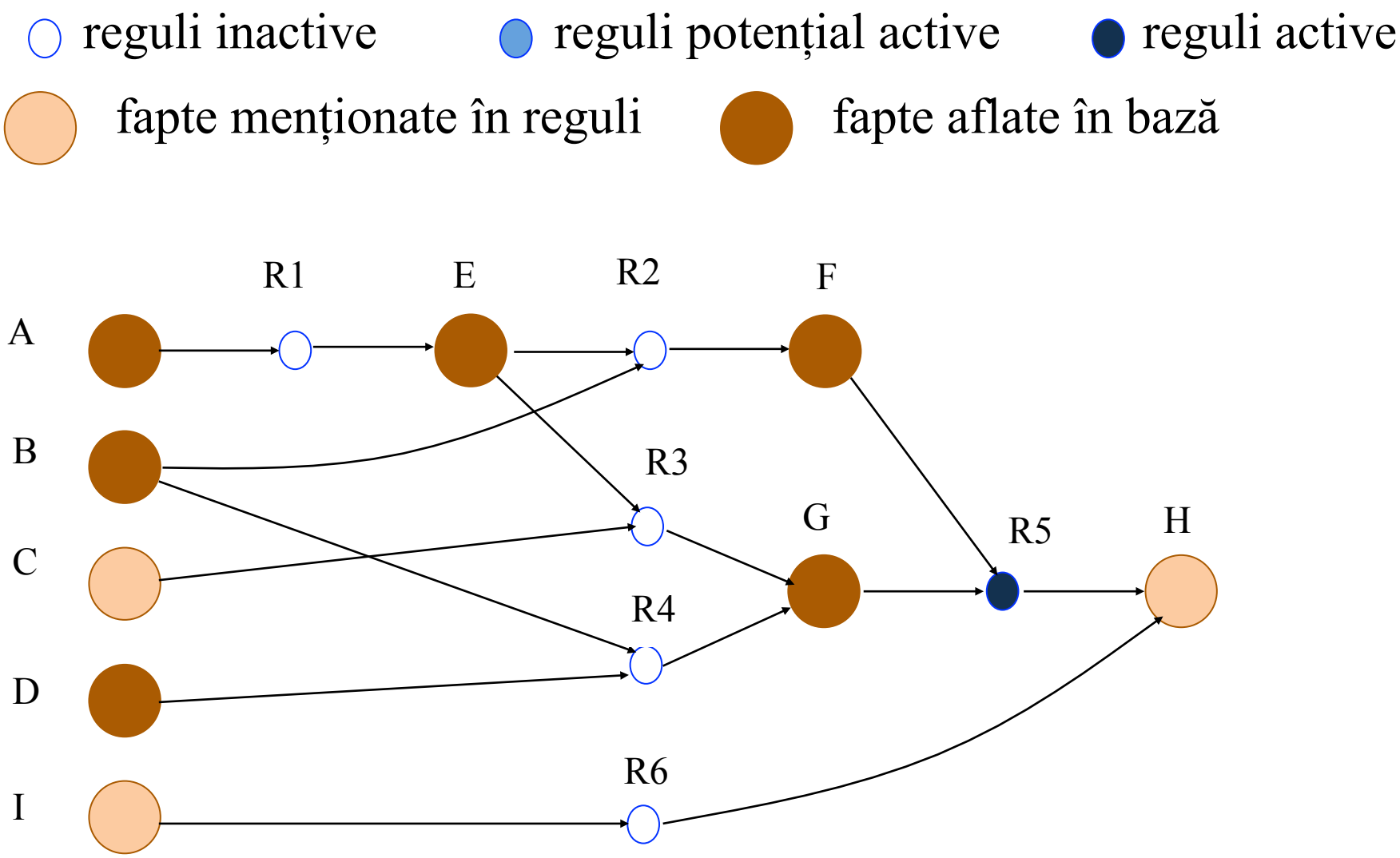
O rulare posibilă



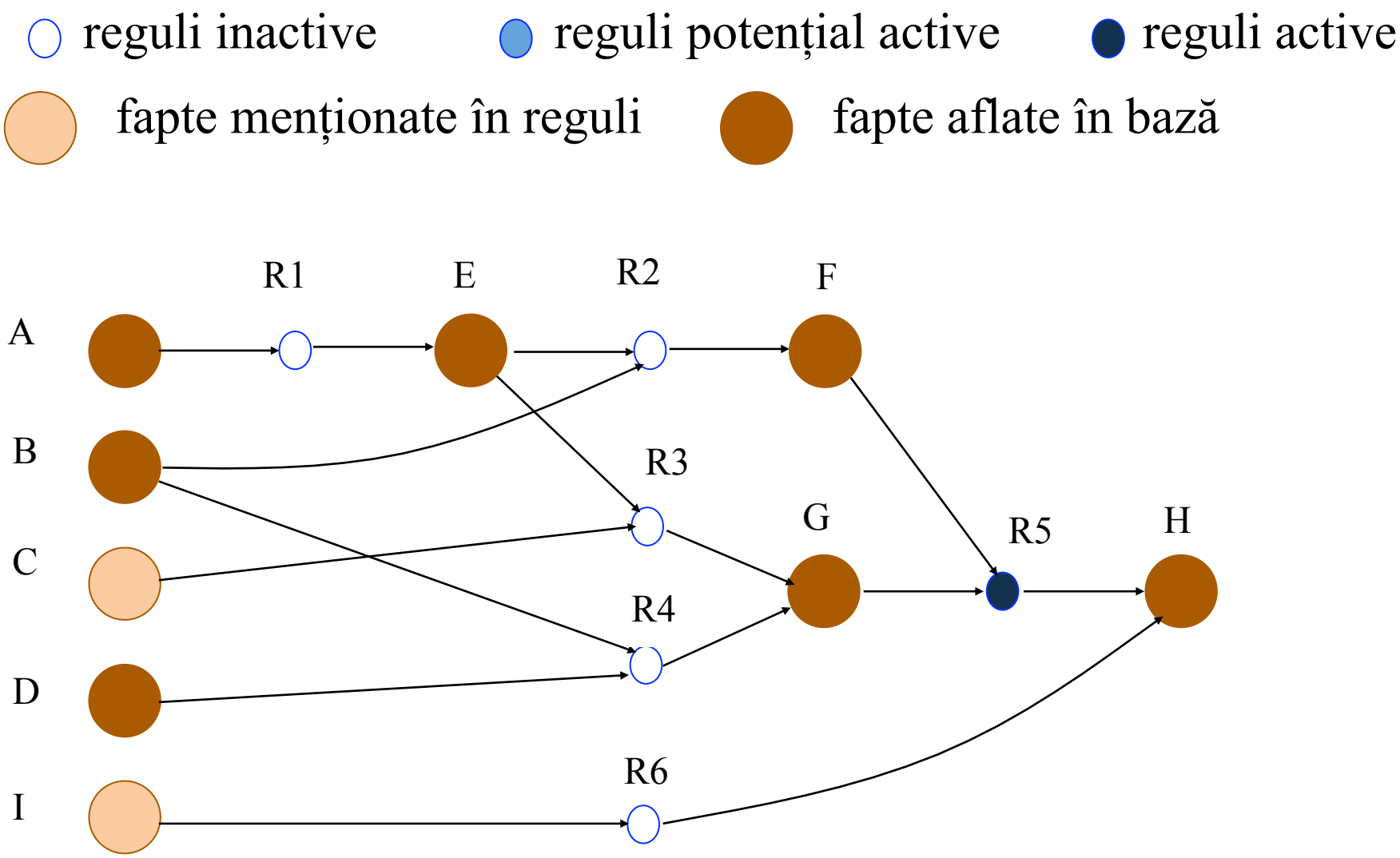
O rulare posibilă



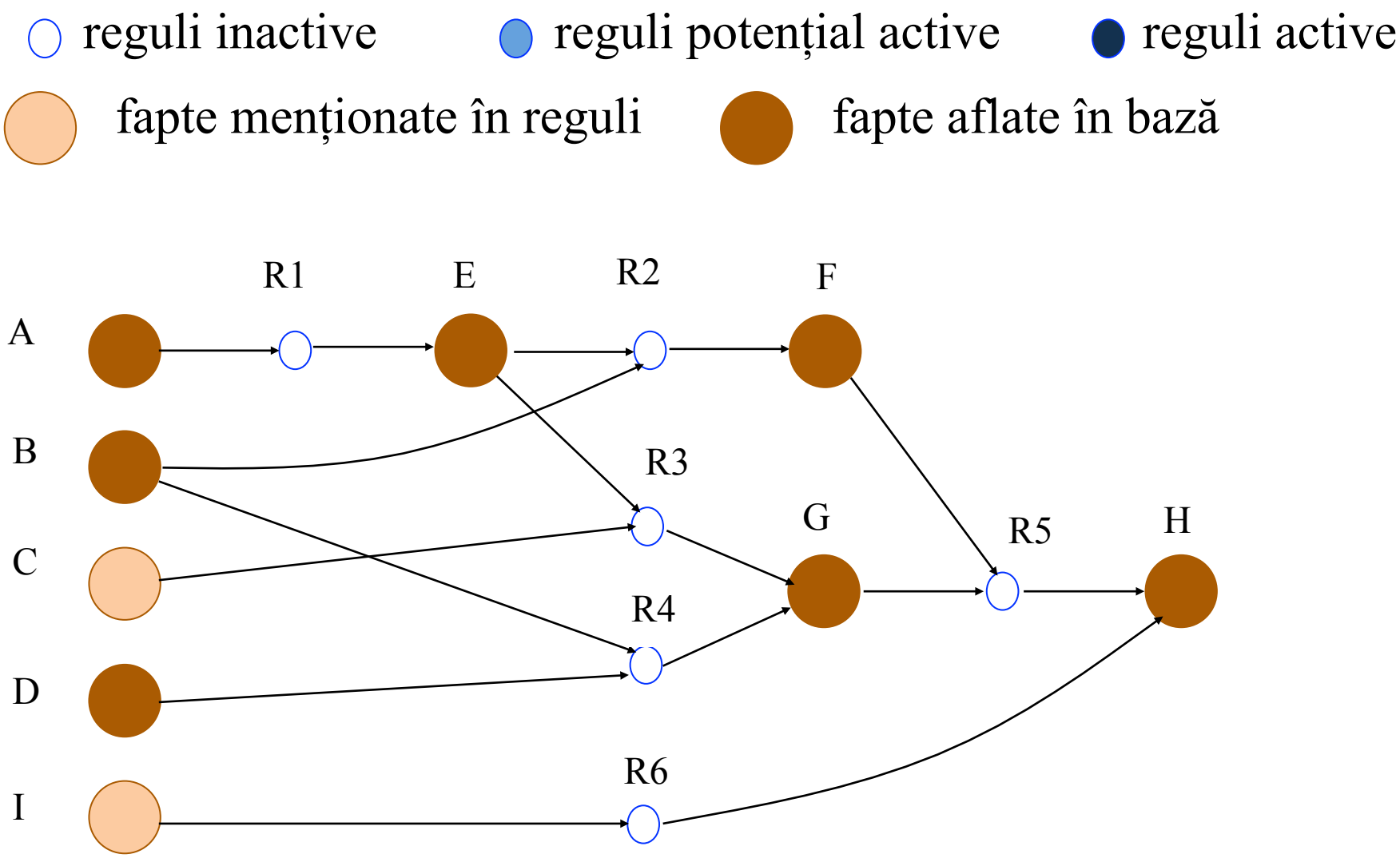
O rulare posibilă



O rulare posibilă



O rulare posibilă



Exemplu: selecții

- *O bază de date conține informaţii despre nişte piese de lego:*
  - *un identificator al pieselor,*
  - *forma,*
  - *materialul din care sunt făcute (plastic sau lemn)*
  - *suprafaţa,*
  - *grosimea*
  - *culoarea.*

Exemplu: selecții

- *Să se afle:*
  1. *toate piesele de altă culoare decât galben;*
  2. *toate piesele de culoare fie roşu fie galben;*
  3. *toate piesele care au grosimea strict mai mică de 5 unităţi, dar diferită de 2,*
  4. *toate piesele roşii din plastic şi verzi din lemn;*
  5. *toate piesele din plastic cu aceeaşi arie ca a unor piese din lemn dar mai subţiri decât acestea;*
  6. *toate piesele de aceeaşi grosime, pe grupe de grosimi;*
  7. *cele mai groase piese dintre cele de aceeaşi arie.*