Εισαγωγή στο εργαλείο SPIN

Στην ενότητα αυτή θα μελετηθούν τα εξής θέματα:

Ο model-checker SPIN

Η γλώσσα προδιαγραφής συστημάτων Promela

Εντολές, διεργασίες και μοντέλο επικοινωνίας

Σημασιολογία εκτέλεσης

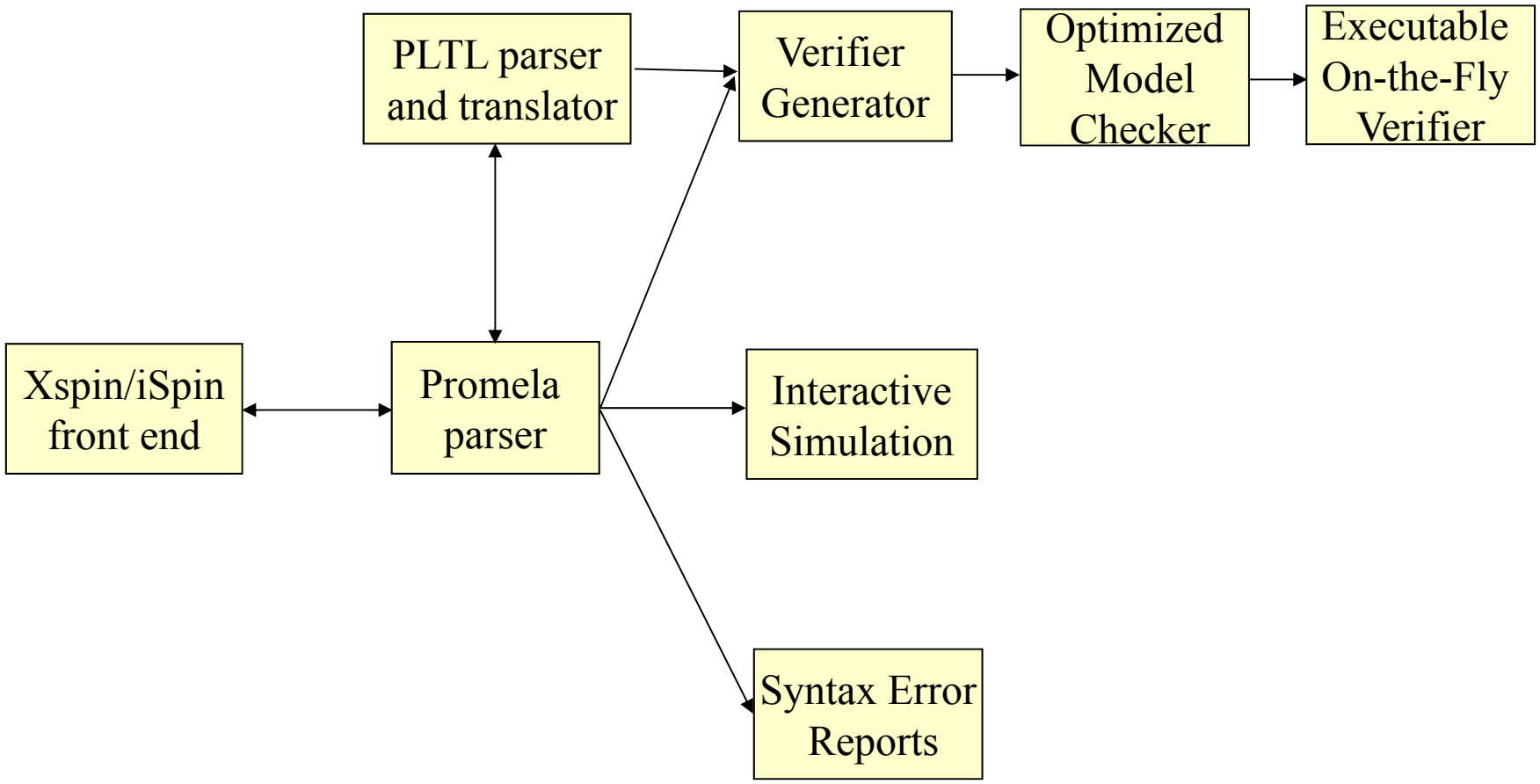
Ανάλυση με την SPIN

Παραδείγματα

Spin και μοντελοέλεγχος

- Ο μοντελοέλεγχος είναι ένα σύνολο από αυτοματοποιημένες τεχνικές οι οποίες ελέγχουν κατά πόσο μοντέλα συστημάτων ικανοποιούν επιθυμητές ιδιότητες.
- Ένας μοντελοελεγκτής (model-checker) επαληθεύει κατά πόσο, για ένα σύστημα M και μια ιδιότητα Φ, **M |= Φ**.
- Το εργαλείο SPIN είναι ένας από τους πιο αποδοτικούς μοντελο-ελεγκτές.
- Ελέγχει ιδιότητες της LTL και για την αυτοματοποιημένη επαλήθευση χρησιμοποιεί αλγόριθμους αυτομάτων.
- Αναπτύχθηκε στα Bell Laboratories.

Βασική δομή της Spin



Spin

- **SPIN** (**S**imple **P**romela **In**terpreter)

- Γλώσσα μοντελοποίησης συστημάτων: **Promela** (**Pr**otocol/Process **Me**ta **La**nguage)
  - Γλώσσα προδιαγραφής συστημάτων με πεπερασμένο αριθμό καταστάσεων (όχι γλώσσα προγραμματισμού)
  - χαλαρά βασισμένη στην άλγεβρα διεργασιών CSP
    - δυναμική δημιουργία παράλληλων διεργασιών
    - επικοινωνία ανάμεσα σε διεργασίες μπορεί να γίνει είτε συγχρονισμένα είτε ασύγχρονα (μέσω buffer)
  - στοιχεία από τη γλώσσα C

Spin

- Σημαντικότερα versions

1.0	Jan 1991	αρχικό version, Holzmann 1991
2.0	Jan 1995	partial-order reduction
3.0	Apr 1997	ελαχιστοποίηση του αυτομάτου του μοντέλου
4.0	2002	εισαγωγή κώδικα C
5.0	2007	Multi-core support
6.0	2012	Priority-based scheduling

ACM Software Systems Award, 2001
Gerard Holzmann

- Στοιχεία επιτυχίας
  - ανάλυση συστημάτων με το "πάτημα ενός κουμπιού"
  - πολύ αποδοτική υλοποίηση
  - καλή γραφική διασύνδεση
  - καλή υποστήριξη
  - περιέχει τα αποτελέσματα έρευνας 2+ δεκαετιών στην αυτοματοποιημένη επαλήθευση συστημάτων (πολλοί αλγόριθμοι βελτιστοποίησης)

Promela

- Ένα μοντέλο της Promela περιέχει
  - δηλώσεις τύπων
  - δηλώσεις καναλιών
  - δηλώσεις καθολικών μεταβλητών
  - δηλώσεις διεργασιών
  - δήλωση αρχικής διεργασίας

```
mtype = {OK, READY, ACK}
int name;
typedef type1 type2
```

```
channel ch = {s} of type…
```

```
proctype p (args)
      {statements}
```

```
[init process]
```

- Μοντέλα της Promela αντιστοιχούν σε δυνατόν μεγάλα αλλά πεπερασμένα συστήματα μεταβάσεων.

Τύποι δεδομένων

bit/bool	0,1
byte	0...255
short	$-2^{15}-1 \ldots 2^{15}-1$
int	$-2^{31}-1 \ldots 2^{31}-1$

- *Δηλώσεις μεταβλητών*

```
byte name1, name2=4, name3;
int arr1[5];
```

- *Δηλώσεις νέων τύπων*

```
mtype = {OK, READY, ACK}
mtype status = OK
```

- *Δομές*

```
typedef Msg{
        byte a[2], b;
        short c
}
```

Εκφράσεις και εντολές

- *Σταθερές*: `#define NAME 5`
- *Εκφράσεις*: αριθμητικές (+, -, *, /, %), συγκρίσεις (>, >=, <, <=, ==, !=) Boolean (&&, ||, !), ανάθεση (=) αύξηση/μείωση (++, --)
- Στην Promela οι συνθήκες μπορούν να χρησιμοποιηθούν ως εντολές:

  π.χ. η συνθήκη

  ```
  (a == b)
  ```

  εκτελείται όταν τα a και b εξισωθούν.
- /* Τα *σχόλια* μπαίνουν σε "παρενθέσεις" */
- Μία εντολή σε οποιοδήποτε σημείο του μοντέλου μπορεί να ονομαστεί από μία *ετικέτα*. Μία ετικέτα μπορεί να είναι μία ακολουθία χαρακτήρων ακολουθούμενη από ':'. Οι ετικέτες `end`, `progress`, `accept`, έχουν ειδική σημασία.

Διεργασίες

- Ένας τύπος διεργασίας (`proctype`) περιέχει
  - ένα όνομα
  - μία λίστα από τυπικές παραμέτρους
  - δηλώσεις τοπικών μεταβλητών
  - μία ακολουθία εντολών

```
proctype A () {
  byte state;
  state = 3
}
```

```
byte state;
proctype B () {
      state = state - 1
}
proctype C(){
    (state == 1) -> state = 3
}
```

Διεργασίες

- Μία δήλωση διεργασίας απλά ορίζει μια συμπεριφορά διεργασίας, δεν την εκτελεί. Εκτέλεση διεργασίας πρέπει να δηλωθεί ρητά μέσω του τελεστή `run`. Ο τελεστής `run` επιστρέφει τον αριθμό της διεργασίας που ξεκίνησε.

- Σε ένα μοντέλο της Promela, αρχικά εκτελείται μόνο η διεργασία με όνομα `init`.

- Διεργασίες μπορούν να δημιουργηθούν από οποιαδήποτε διεργασία σε οποιοδήποτε σημείο κατά την εκτέλεση ενός μοντέλου.

- Διεργασίες μπορούν επίσης να δημιουργηθούν όταν δηλωθούν ως `active` πριν από τη δήλωση `proctype`.

Παράδειγμα

```
byte state = 1;

proctype A(){
       byte tmp;
       (state==1) -> tmp = state;
                     tmp = tmp+1; state = tmp
}

proctype B(){
       byte tmp;
       (state==1) -> tmp = state;
                     tmp = tmp-1; state = tmp
}

init{
       run A(); run B()
}
```

Ο αλγόριθμος αμοιβαίου αποκλεισμού του Dekker

```
#define true    1
#define false   0
#define Aturn   false
#define Bturn   true

bool x, y, t;

proctype A(){
   x = true;
   t = Bturn;
   (y == false || t == Aturn);
   /* critical section */
   x = false
}

init{
   run A(); run B()
}
```

```
proctype B(){
  y = true;
  t = Aturn;
  (x == false || t == Bturn);
  /* critical section */
  y = false
}
```

Ατομικές ακολουθίες

- Η λέξη κλειδί `atomic` έχει ως αποτέλεσμα τη δημιουργία ατομικών εντολών, δηλαδή, εκτελεί την ακολουθία εντολών στην οποία αναφέρεται ως μία αδιάσπαστη οντότητα.

- Ποιες οι δυνατές εκτελέσεις του πιο κάτω μοντέλου;

```
byte state = 1
proctype A () {
      atomic{
       (state == 1) -> state = state - 1
      }
}
proctype B (){
 atomic{
       (state == 1) -> state = state + 1
 }
}
init {
  run A(); run B();
}
```

Κανάλια μηνυμάτων

- Στην Promela είναι δυνατή η δημιουργία *καναλιών* τα οποία χρησιμοποιούνται για μεταφορά μηνυμάτων ανάμεσα σε διεργασίες.

- Η δήλωση

  `chan Transfer = [x] of {type`$_1$`, type`$_2$`, …, type`$_n$`}`

  δημιουργεί το κανάλι `Transfer` που μπορεί να κρατήσει μέχρι `x` μηνύματα-πλειάδες n στοιχείων τύπου ($type_1$, $type_2$, …, $type_n$).

- Η εντολή

  `Transfer!expr`

  στέλνει την τιμή του expr στο κανάλι Transfer, ενώ η εντολή

  `Trasfer?y`

  λαμβάνει ένα μήνυμα από το κανάλι Transfer και τοποθετεί την τιμή αυτή στη μεταβλητή y.

Κανάλια μηνυμάτων

- Η επικοινωνία είναι FIFO.

- Κανάλια μπορούν να είναι αντικείμενα μηνυμάτων:

```
chan queue = [3] of {mtype,chan,int}
queue!READY, Transfer, 1
```

ή

```
queue!READY(Transfer, 1)
```

Παράδειγμα

```
proctype A(chan q1){
        chan q2;
        q1?q2;
        q2!123
}

proctype B(chan q3){
        int x;
        q3?x;
        printf("x = %d\n", x)
}

init {
        chan qname = [1] of { chan };
        chan qforb = [1] of { int };
        run A(qname);
        run B(qforb);
        qname!qforb
}
```

Κανάλια μηνυμάτων

- Ειδική περίπτωση όταν x (το μέγεθος του καναλιού) = 0. Για παράδειγμα

```
chan port = [0] of {byte}
```

είναι ένα συγχρονισμένο κανάλι.

- Το κανάλι μπορεί να διαβιβάσει αλλά όχι να αποθηκεύσει μηνύματα. Επομένως εξαναγκάζει τον αποστολέα να περιμένει μέχρις ότου κάποιος είναι έτοιμος να συγχρονιστεί.

Εντολές

- Μία εντολή μπορεί να βρίσκεται σε μια από τις καταστάσεις
  - *εκτελέσιμη*, αν η εντολή μπορεί να εκτελεστεί άμεσα
  - *blocked*, αν δεν μπορεί να εκτελεστεί
- Μια ανάθεση είναι πάντα εκτελέσιμη.
- Οι εκφράσεις είναι επίσης νόμιμες εντολές. Μια έκφραση είναι εκτελέσιμη αν η τιμή της είναι άνιση του μηδενός
  - $2 < 3$ πάντα εκτελέσιμη
  - $x < 27$ εκτελέσιμη μόνο αν η τιμή του x είναι μικρότερη του 27
  - $3 + x$ εκτελέσιμη μόνο αν η τιμή του x είναι άνιση με -3

Εντολές

- Η εντολή skip είναι πάντα εκτελέσιμη
  - δεν κάνει τίποτα, απλά αυξάνει το process counter της διεργασίας

- Ο τελεστής run είναι εκτελέσιμος μόνο αν είναι δυνατόν να δημιουργηθεί μια νέα διεργασία.

- Η εντολή printf είναι πάντα εκτελέσιμη.

Έλεγχος ροής

- Επιλογή if

```
if
:: x%2==1 -> z=z*y; x--
:: x%2==0 -> y=y*y; x=x/2
fi
```

- Γενικά, αν υπάρχουν περισσότερες από μια αληθείς συνθήκες η Spin διαλέγει τυχαία μια από τις εκτελέσιμες επιλογές.
- Αν δεν υπάρχει καμιά αληθής επιλογή η εντολή `if` μπλοκάρεται.
- Λέξη κλειδί `else`

  Μπορεί να χρησιμοποιηθεί ως συνθήκη. Γίνεται εκτελέσιμη αν καμιά από τις υπόλοιπες συνθήκες δεν είναι εκτελέσιμη.

```
if
:: x==1 -> x--
:: else -> skip
fi
```

Έλεγχος ροής

- Επανάληψη `do` και εντολή `goto`

```
do
:: x>y -> x=x-y
:: y>x -> y=y-x
:: else goto outside
od;
outside: …
```

- Η εντολή `break` επιτρέπει τη διαφυγή από ένα do-loop.

Ποια η διαφορά των πιο κάτω βρόχων

```
byte count;

proctype counter(){
   do
   :: count = count + 1
   :: count = count - 1
   :: (count == 0) -> break
   od
}
```

```
byte count;

proctype counter(){
   do
   :: (count != 0) ->
        if
        :: count = count + 1
        :: count = count - 1
        fi
   :: (count == 0) -> break
   od
}
```

Παράδειγμα

```
#define p       0
#define v       1

chan sema = [0] of { bit };

proctype dijkstra(){
  byte count = 1;
  do
  :: (count == 1) -> sema!p; count = 0
  :: (count == 0) -> sema?v; count = 1
  od
}
proctype user(){
  do
  :: sema?p;    /*     critical section begins */
     sema!v    /* non-critical section */
  od
}
init{
  run dijkstra(); run user(); run user(); run user()
}
```

Η εντολή timeout

- Πολλά πρωτόκολλα χρησιμοποιούν ρολόγια ή μηχανισμούς timeout για να ξαναστείλουν μηνύματα ή acknowledgements. Η Promela δεν παρέχει στοιχεία πραγματικού χρόνου, αλλά…
- Η λέξη κλειδί `timeout` επιτρέπει την απόδραση από καταστάσεις αδιεξόδου: παίρνει την τιμή 1 όταν καμιά άλλη εντολή δεν μπορεί να εκτελεστεί.
- Η πιο κάτω διεργασία κάνει reset το σύστημα όταν αυτό κολλήσει.

```
proctype watchdog(){
    do
    :: timeout -> guard!reset
    od
}
```

Ετικέτες στην Promela

- Για διευκόλυνση ανίχνευσης *αδιεξόδων* και *προόδου*, κατά την εκτέλεση ενός μοντέλου στην Promela, παρέχονται (διαδικασίες και) ειδικές ετικέτες οι οποίες μπορούν να χρησιμοποιηθούν για να ξεχωρίσουν
  - ανάμεσα σε φυσιολογική αδράνεια και αδιέξοδα, και
  - σημαντικά κομμάτια κώδικα που υποδηλώνουν πρόοδο κατά την εκτέλεση
- Για σήμανση ότι μια αδρανής κατάσταση δεν αποτελεί αδιέξοδο αλλά αναμονή για νέα δεδομένα, μπορεί να χρησιμοποιηθεί η ετικέτα `end` ή οποιαδήποτε συμβολοσειρά αρχίζει με τους χαρακτήρες 'end' (π.χ. `end2`). Η ίδια ετικέτα μπορεί να χρησιμοποιηθεί για να δείξει ότι μια κατάσταση μπορεί να θεωρηθεί ως τελική (αν όλες οι υπόλοιπες διεργασίες έχουν τερματίσει).
- Για σήμανση ότι κάποιο κομμάτι κώδικα πρέπει να εκτελεσθεί για να σημειωθεί πρόοδος στο μοντέλο μπορεί να χρησιμοποιηθεί η ετικέτα `progress` ή οποιαδήποτε συμβολοσειρά αρχίζει με τους χαρακτήρες 'progress' (π.χ. `progresstwo`).

Ετικέτες στην Promela

```
proctype dijkstra(){

              byte count = 1;
end:          do
              :: (count == 1) ->
progress:                sema!p; count = 0
              :: (count == 0) ->
                       sema?v; count = 1
               od
}
```

Σημασιολογία παρεμβαλλόμενης διάταξης

- Διεργασίες στην Promela εκτελούνται ταυτόχρονα με *σημασιολογία παρεμβαλλόμενης διάταξης*.

- Εντολές των διεργασιών θεωρούνται ως ατομικές.

- Αν σε κάποια στιγμή υπάρχουν περισσότερες από μια εκτελέσιμες εντολές σε ενεργές παράλληλες διεργασίες, μία από αυτές επιλέγεται τυχαία για εκτέλεση.

- Οι μόνες εντολές που εκτελούνται ταυτόχρονα, αφορούν τις εντολές που απαρτίζουν μια συγχρονισμένη επικοινωνία.

Ιδιότητες

- Με τη Spin μπορούμε να ελέγξουμε τους πιο κάτω τύπους ιδιοτήτων
  - ισχυρισμούς
  - ύπαρξη αδιεξόδων
  - ύπαρξη νεκρού κώδικα
  - ιδιότητες ζωτικότητας
    - Κύκλοι έλλειψης προόδου
    - Αποδεκτοί κύκλοι
  - ιδιότητες γραμμικής λογικής (LTL)

Ισχυρισμοί

- Η εντολή `assert(expr)` χρησιμοποιείται συχνά σε μοντέλα της Promela για έλεγχο κατά πόσο ιδιότητες ικανοποιούνται σε κάποιες καταστάσεις.

- Αν η τιμή της έκφρασης `expr` υπολογισθεί ως 0 η Spin θα τερματίσει την εκτέλεση με μήνυμα λάθους, αφού η πρόταση expr παραβιάζεται.

- Η εντολή `assert(expr)` είναι πάντα εκτελέσιμη.

Αμοιβαίος Αποκλεισμός – Χρήση ισχυρισμού

```
bool busy;
byte mutex;

proctype P(int i){
        (!busy) -> busy = true;
        mutex++;
        printf("P%d in critical section\n", i);
        mutex--;
        busy = false;
}
active proctype invariant(){
        assert (mutex <= 1)
}

init{
        run P(0); run P(1)
}
```

Ισχυρισμός – μαρκάρουμε την επιλογή "Assertions" στα "Verification Options"

Ιδιότητες

- Με τη Spin μπορούμε να ελέγξουμε τους πιο κάτω τύπους ιδιοτήτων
  - ισχυρισμούς
  - **ύπαρξη αδιεξόδων**
  - ύπαρξη νεκρού κώδικα
  - ιδιότητες ζωτικότητας
    - Κύκλοι έλλειψης προόδου
    - Αποδεκτοί κύκλοι
  - ιδιότητες γραμμικής λογικής (LTL)

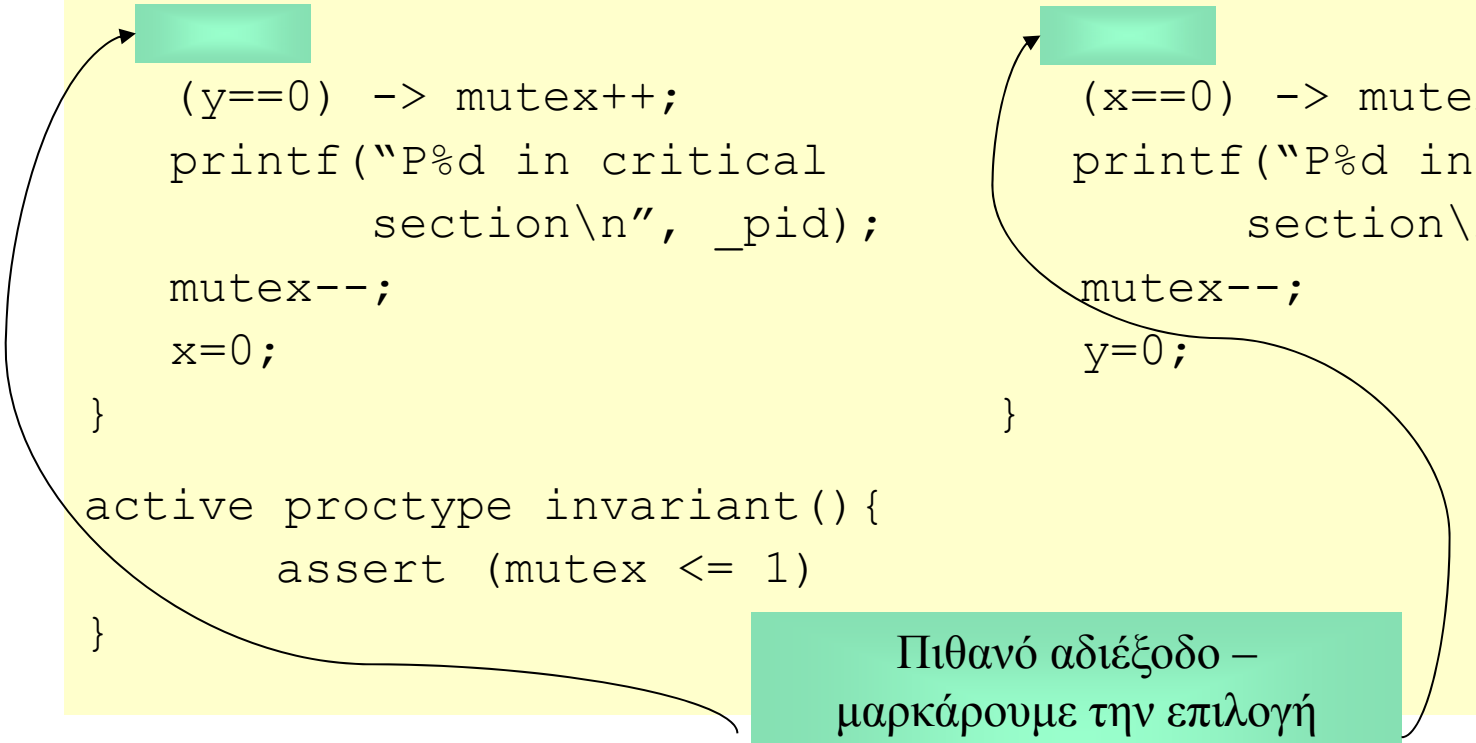
Αμοιβαίος Αποκλεισμός – Εντοπισμός αδιεξόδου

```
bit x, y;
byte mutex;

active proctype A(){
    (y==0) -> mutex++;
    printf("P%d in critical
            section\n", _pid);
    mutex--;
    x=0;
}

active proctype B(){
    (x==0) -> mutex++;
    printf("P%d in critical
            section\n",_pid);
    mutex--;
    y=0;
}

active proctype invariant(){
        assert (mutex <= 1)
}
```



Πιθανό αδιέξοδο – μαρκάρουμε την επιλογή "Invalid Endstates" στα "Verification Options"

Παράδειγμα

```
#define p       0
#define v       1

chan sema = [0] of { bit };

proctype dijkstra(){
end:    do
         :: sema!p; sema?v
        od
}

proctype user(){
     sema?p;
      /*     critical section */
     sema!v
}

init{
  run dijkstra(); run user(); run user(); run user()
}
```

Αποδεκτό “αδιέξοδο”

Ιδιότητες

- Με τη Spin μπορούμε να ελέγξουμε τους πιο κάτω τύπους ιδιοτήτων
  - ισχυρισμούς
  - ύπαρξη αδιεξόδων
  - **ύπαρξη νεκρού κώδικα**
  - ιδιότητες ζωτικότητας
    - Κύκλοι έλλειψης προόδου
    - Αποδεκτοί κύκλοι
  - ιδιότητες γραμμικής λογικής (LTL)

Η SPIN ενημερώνει για μη εφικτές καταστάσεις

Ιδιότητες

- Με τη Spin μπορούμε να ελέγξουμε τους πιο κάτω τύπους ιδιοτήτων
  - ισχυρισμούς
  - ύπαρξη αδιεξόδων
  - ύπαρξη νεκρού κώδικα
  - **ιδιότητες ζωτικότητας**
    - **Κύκλοι έλλειψης προόδου**
    - **Αποδεκτοί κύκλοι**
  - ιδιότητες γραμμικής λογικής (LTL)

Ο αλγόριθμος αμοιβαίου αποκλεισμού του Dekker

```
byte mutex;

bool x, y;
mtype t = Aturn;

active proctype A(){
do
:: x = true;
   t = Bturn;
   (!y || t == Aturn);
    mutex++;
    /* critical section */
    mutex--;
    x = false
od
}

active proctype invariant(){
   assert (mutex <= 1)
}
```

```
active proctype B(){
do
::  y = true;
    t = Aturn;
   (!x || t == Bturn);
progress:  mutex ++;
    /* critical section */
    mutex--;
    y = false
}
```

Υπάρχει κύκλος όπου δεν σημειώνεται πρόοδος στη διεργασία B – μαρκάρουμε την επιλογή “Non-progress cycles” στα “Verification Options”

Ιδιότητες

- Με τη Spin μπορούμε να ελέγξουμε τους πιο κάτω τύπους ιδιοτήτων
  - ισχυρισμούς
  - ύπαρξη αδιεξόδων
  - ύπαρξη νεκρού κώδικα
  - ιδιότητες ζωτικότητας
    - Κύκλοι έλλειψης προόδου
    - Αποδεκτοί κύκλοι
  - **ιδιότητες γραμμικής λογικής (LTL)**

LTL στην SPIN

- Η SPIN επιτρέπει τον μοντελοέλεγχο LTL ιδιοτήτων.
- Η σύνταξη που χρησιμοποιείται έχει ως εξής:

```
f ::= true, false
      propositional symbols (p, q, κλπ)
      ( f )
      unary f
      f1 binary f2

unary ::= []  (G)
      |   <>  (F)
      |   X
      |   !   (λογική άρνηση)

binary ::= U
       |  &&  (λογικό και)
       |  ||  (λογικό ή)
       |  ->  (λογική συνεπαγωγή)
       |  <->  (λογική ισοδυναμία)
```

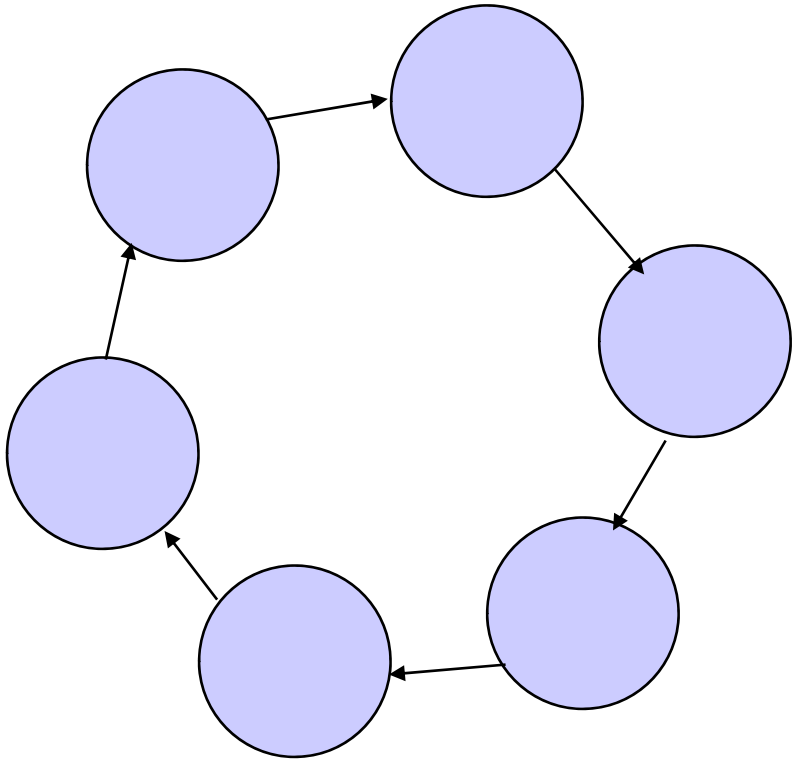
Spin

- Η Spin επιτρέπει στο χρήστη τα εξής:
  - Σύνταξη μοντέλων στην Promela
  - Προσομοίωση μοντέλων της Promela
    - τυχαιοποιημένες προσομοιώσεις
    - προσομοιώσεις με αλληλεπίδραση
  - Επαλήθευση μοντέλων της Promela
    - έλεγχος LTL ιδιοτήτων
    - έλεγχος ισχυρισμών και εύρεση αδιεξόδων, κλπ
  - Επιπλέον προσφέρει
    - Εισηγήσεις για απλοποίηση μοντέλων
    - γραφικό user interface με γραφική παράθεση των συστημάτων μεταβάσεων που αντιστοιχούν σε διεργασίες
    - Property manager για LTL ιδιότητες

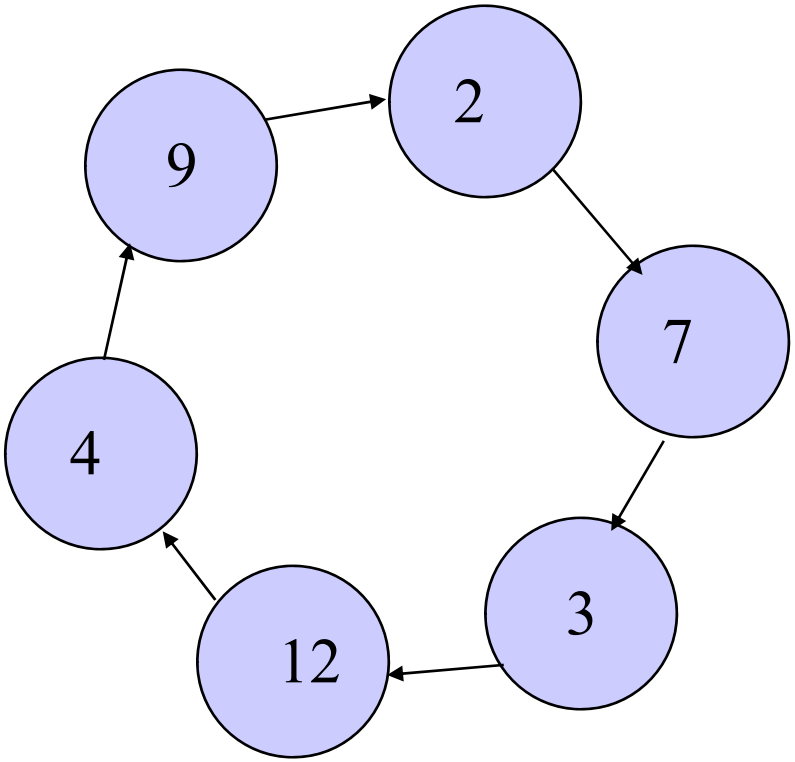
Εκλογή αρχηγού

Ένας κατευθυνόμενος δακτύλιος από επεξεργαστές. Κάθε ένας έχει μια μοναδική τιμή. Επικοινωνία γίνεται στη φορά των δεικτών του ρολογιού.

Ο αρχηγός εκλέγεται ως ο επεξεργαστής με τη μέγιστη τιμή.



Παράδειγμα



Βασική ιδέα αλγορίθμου

- Αρχικά όλες οι διεργασίες είναι ενεργές.
- Αν κάποια διεργασία αντιληφθεί πως η τιμή της δεν είναι η μέγιστη γίνεται αδρανής.
- Μια αδρανής διεργασία απλά μεταφέρει τιμές σε δεξιόστροφη φορά.
- Ο αλγόριθμος εκτελείται σε φάσεις.
- Σε κάθε φάση κάθε διεργασία στέλνει κάποια τιμή προς τα δεξιά.

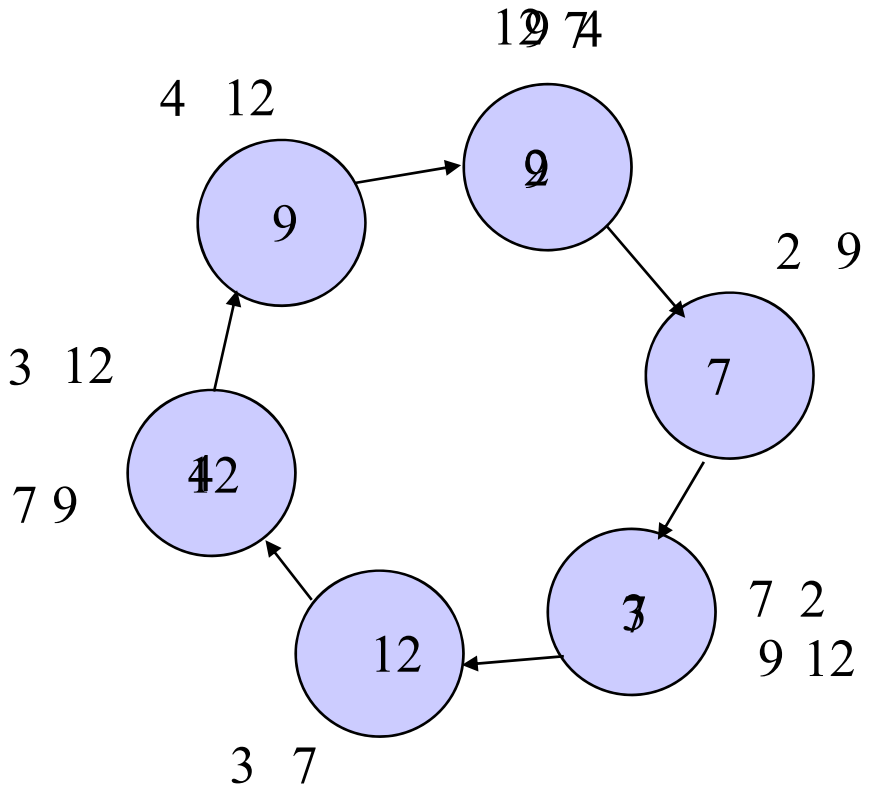
Βασική ιδέα (συν.)

- Βήμα 1: Κάθε διεργασία, λαμβάνοντας κάποια τιμή από τα αριστερά τη συγκρίνει με την τωρινή της τιμή
  - Αν είναι η ίδια, αυτή είναι και η μέγιστη.
  - Αν δεν είναι η ίδια στέλνει την τιμή που μόλις έλαβε στα δεξιά.

- Βήμα 2: Όταν λάβει τη δεύτερη τιμή συγκρίνει τις τρεις τιμές, δηλαδή την τιμή της ίδιας της διεργασίας και τις τιμές των δύο ενεργών διεργασιών στα αριστερά.

- Αν η πρώτη αριστερή ενεργή διεργασία έχει τη μέγιστη τιμή, τότε κρατά αυτή την τιμή και επιστρέφει στο Βήμα 1. Διαφορετικά γίνεται αδρανής.

Παράδειγμα



Ψευδοκώδικας

```
active:
  d = my_number;
  do forever
        send(d);
        receive(e);
        if (e = d) then send to all processes and STOP
        send(e);
        receive(f);
        if e>=max(d,f) then d = e
        else goto relay
  end

relay:
  do forever
        receive(d);
        send(d);
  end
```

Ανάλυση αλγορίθμου

Αφού μοντελοποιήσουμε τον αλγόριθμο στη Promela, μπορούμε να ελέγξουμε ιδιότητες όπως:

- Η μέγιστη τιμή τελικά θα βρεθεί.
- Ο αλγόριθμος βρίσκει ακριβώς μια μέγιστη τιμή.
- Από τη στιγμή που βρίσκεται η μέγιστη τιμή εξακολουθεί να υπάρχει και δεν αλλάζει η τιμή της.
- Η μέγιστη τιμή είναι πάντα ίση με 5.

Δείτε την υλοποίηση (αρχείο leader) και ιδιότητες για ανάλυση (αρχείο leader.ltl) στο distribution της SPIN.

Το πρόβλημα με τους hippies

- Τέσσερα άτομα (Νικόλας, Βίκη, Μαρία and Γιώργος) βρίσκονται μέσα σε μια σπηλιά κρατώντας μια λάμπα πετρελαίου.
- Πρέπει να διασχίσουν ένα στενό πέρασμα όπου μόνο ένα ή δύο άτομα μπορούν να περάσουν την ίδια στιγμή. Επίσης δεν είναι ασφαλές να περάσουν χωρίς φως.
- Τα τέσσερα άτομα μπορούν να περάσουν το πέρασμα σε διαφορετικούς χρόνους: ο Νικόλας σε 5 λεπτά, η Βίκη σε 10, η Μαρία σε 20 και ο Γιώργος σε 25.
- Όταν δύο άτομα περνούν μαζί από το πέρασμα, ο χρόνος που χρειάζονται είναι ο μεγαλύτερος από τους χρόνους τους.
- Το λάδι στη λάμπα μπορεί να παρέχει φως μόνο για 60 λεπτά.
- Είναι δυνατή η διάσχιση σε χρόνο <= 60;

Μοντέλο Spin (1)

```
chan out-of-cave = [0] of {hippie, hippie} ;
chan into-cave = [0] of {hippie} ;
chan stopwatch = [0] of {hippie} ;
byte time ;

proctype Out()
{
bit here[N] ;
hippie h1, h2 ;
here[0]=1; here[1]=1; here[2]=1; here[3]=1;
do
:: select_hippie(h1) ;
   select_hippie(h2) ;
   out-of-cave! h1, h2 ;
   if all_gone -> break fi;
   into-cave? h1 ;
   here[h1] = 1 ;
   stopwatch ! h1 ;
od
}
```

Μοντέλο Spin (2)

```
proctype In()
{
bit here[N] ;
hippie h1, h2 ;

do
:: out-of-cave? h1, h2 ;
   stopwatch ! max(h1,h2) ;
    here[h1] = 1 ;
    here[h2] = 1 ;
   if all_here -> break fi;
    select_hippie(h1);
   into-cave! h1 ;

od
}
```

Μοντέλο Spin (3)

```
#define select_hippie(x) \
if                       \
:: here[0] -> x=0        \
:: here[1] -> x=1        \
:: here[2] -> x=2        \
:: here[3] -> x=3        \
fi ; here[x] = 0
```

```
#define all_here \
  (here[0]+here[1]+ \
   here[2]+here[3]==4)
```

```
#define all_gone \
  (here[0]+here[1]+ \
   here[2]+here[3]==0)
```

Μοντέλο Spin (4)

```
proctype Timer()
{
end:
do
:: stopwatch ? 0 -> time=time+5 ;
:: stopwatch ? 1 -> time=time+10
:: stopwatch ? 2 -> time=time+20;
:: stopwatch ? 3 -> time=time+25;
od
}

init {
atomic { run Out(); run In(); run Timer(); }
}
```

Ελέγχουμε την ιδιότητα
<> (time>60)

Αλγόριθμος Προσομοίωσης (simulation)

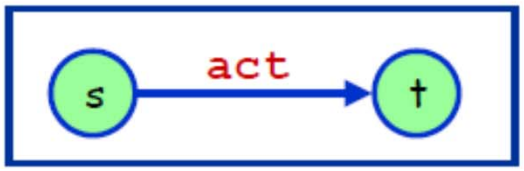
deadlock ≡ allBlocked

```
while (!error & !allBlocked) {
   ActionList menu = getCurrentExecutableActions();
   allBlocked = (menu.size() == 0);
   if (! allBlocked) {
     Action act = menu.chooseRandom();
     error = act.execute();
   }
}
```

interactive simulation:
act is chosen by the user

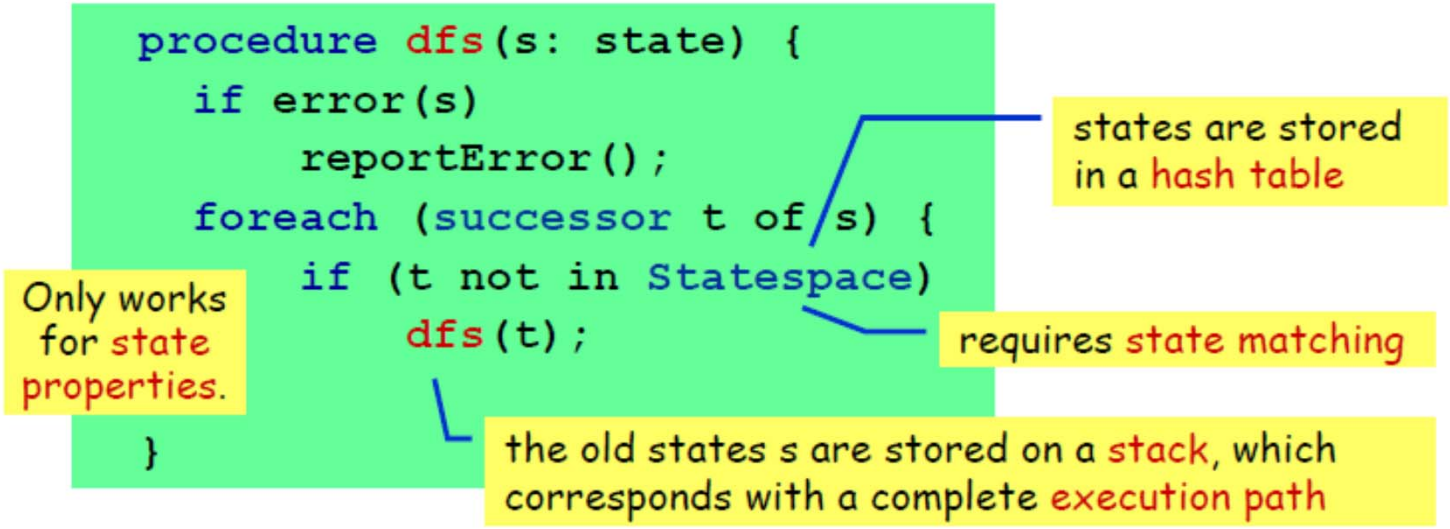
act is executed and the system enters a new state

Visit all processes and collect all executable actions .



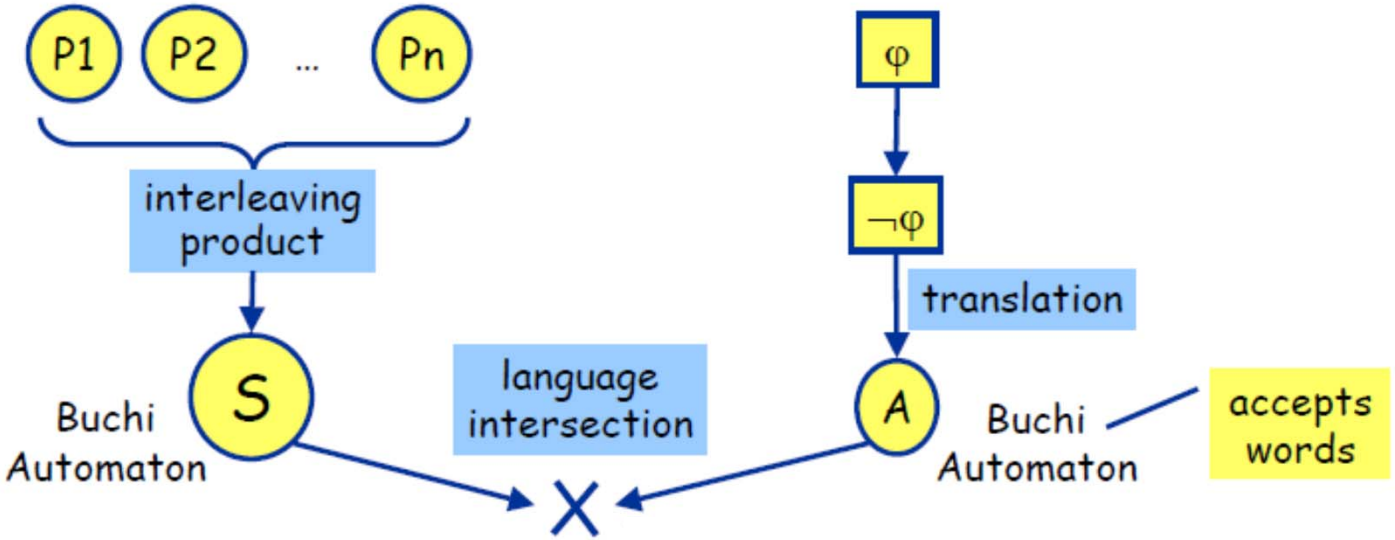
Αλγόριθμος Επαλήθευσης – ιδιότητες κατάστασης

- Η Spin χρησιμοποιεί κατά βάθος διερεύνηση για να διασχίσει το σύστημα καταστάσεων ενός μοντέλου.
- Ο έλεγχος της όποιας ιδιότητας γίνεται ταυτόχρονα με την κατασκευή του μοντέλου (επαλήθευση on-the-fly)



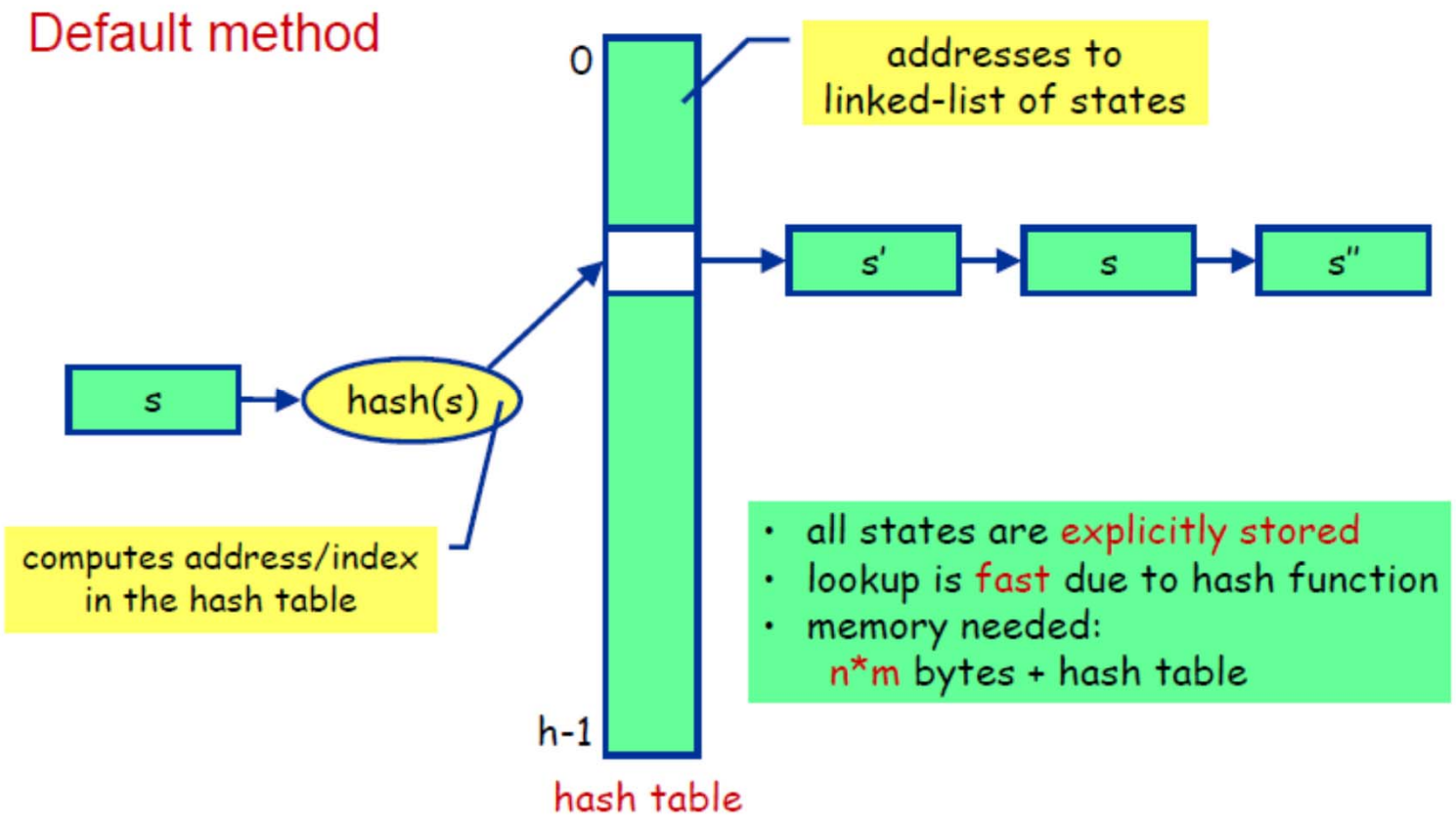
Αλγόριθμος Επαλήθευσης – ιδιότητες εκτέλεσης

- Το μοντέλο και η *άρνηση* της ιδιότητας μοντελοποιούνται ως: αυτόματα.

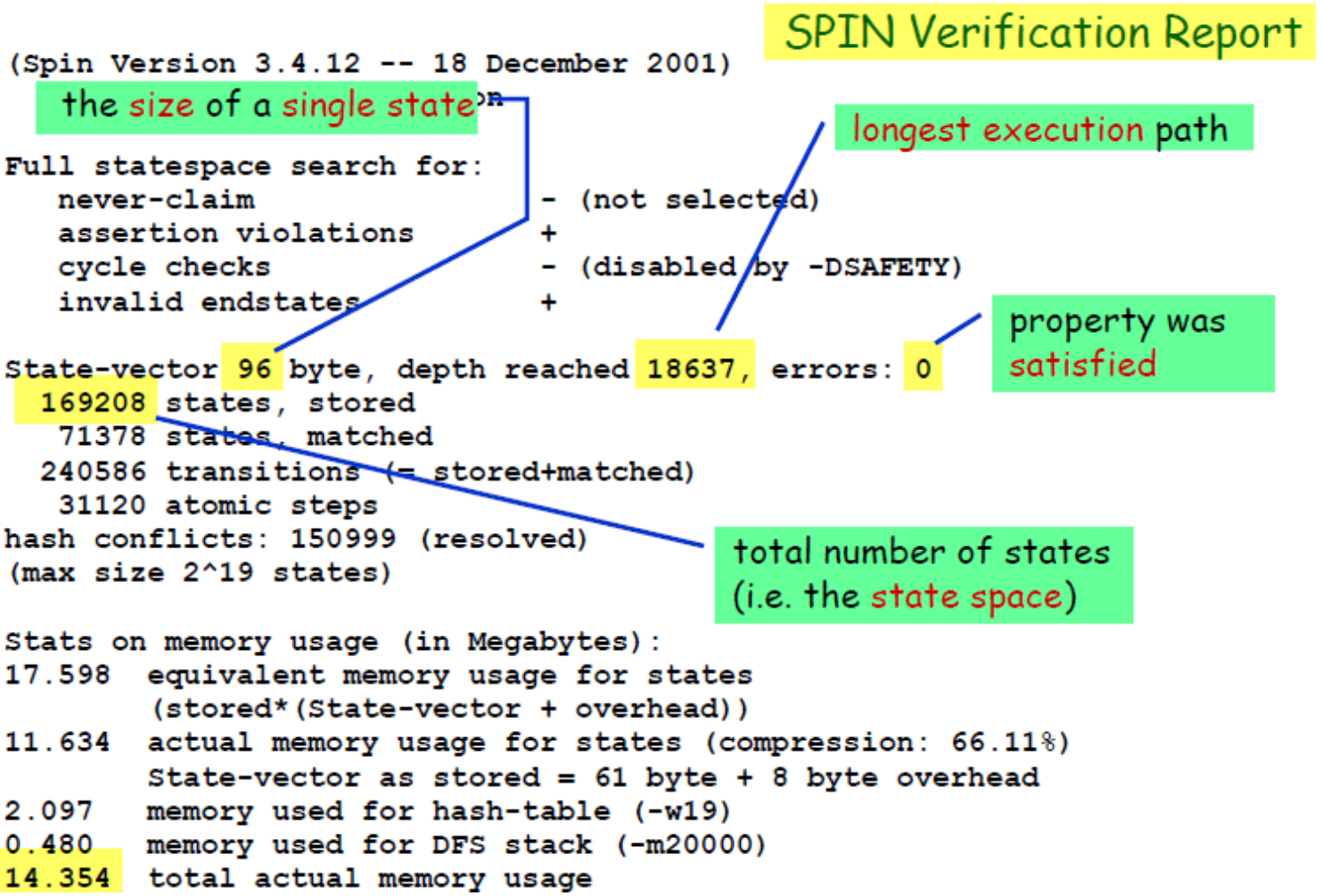


- Αν τομή των γλωσσών των δύο αυτομάτων, X, είναι κενή η ιδιότητα ικανοποιείται από το μοντέλο.

Αποθήκευση καταστάσεων



Αναφορές από SPIN



Αλγόριθμοι Ελαχιστοποίησης (1)

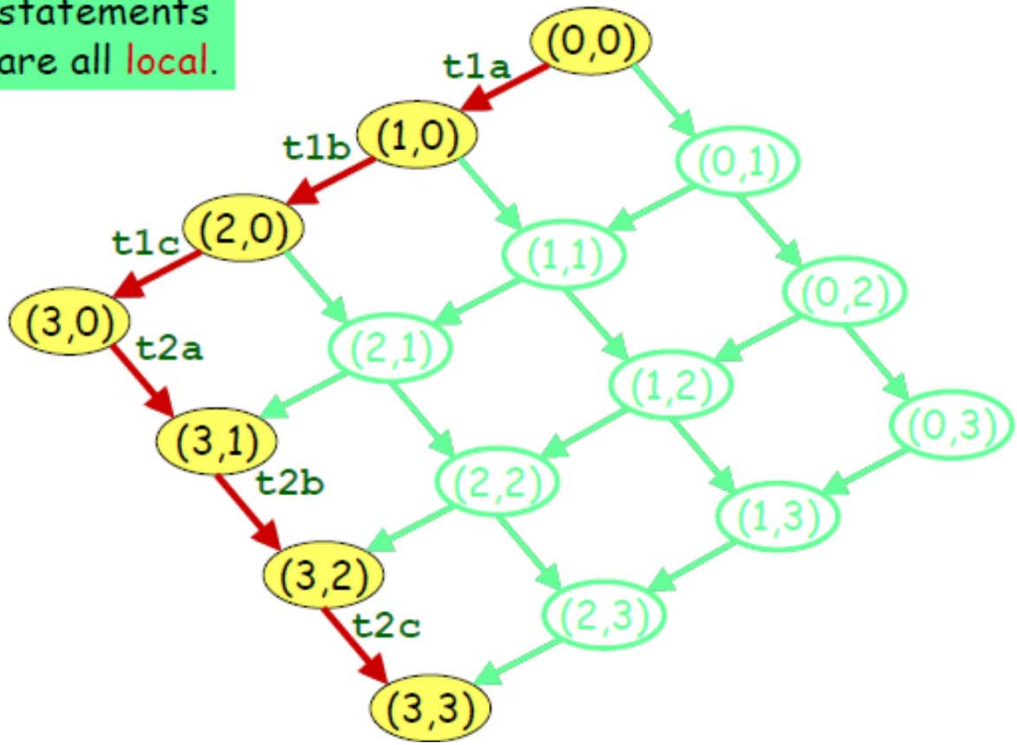
- Η Spin υποστηρίζει διάφορους αλγόριθμους ελαχιστοποίησης του συστήματος καταστάσεων:
  - partial order reduction
  - bitstate hashing
  - minimised automaton encoding of states
  - state vector compression
  - dataflow analysis
  - slicing algorithm

Αλγόριθμοι Ελαχιστοποίησης (2)

- Partial Order Reduction
  - Βασίζεται στην παρατήρηση ότι η ικανοποίηση κάποιας ιδιότητας συχνά δεν εξαρτάται από τη σειρά με την οποία εκτελούνται διάφορες εντολές => είναι αρκετό να θεωρήσουμε μόνο ένα υποσύνολο των δυνατών εκτελέσεων.
  - Ο αλγόριθμος εκτελεί ατομικά τις «τοπικές» μεταβάσεις των διεργασιών όπου τοπικές μεταβάσεις περιλαμβάνουν:
    - Αναθέσεις και άλλες προσβάσεις σε τοπικές μεταβλητές
    - Επικοινωνίες σε κανάλια τα οποία χρησιμοποιούνται από αποκλειστικά δύο διεργασίες.
  - Οι υπόλοιπες εκτελέσεις καλύπτονται από μια μορφή *συρροής*.

Αλγόριθμοι Ελαχιστοποίησης (3)

Suppose the statements of P1 and P2 are all local.

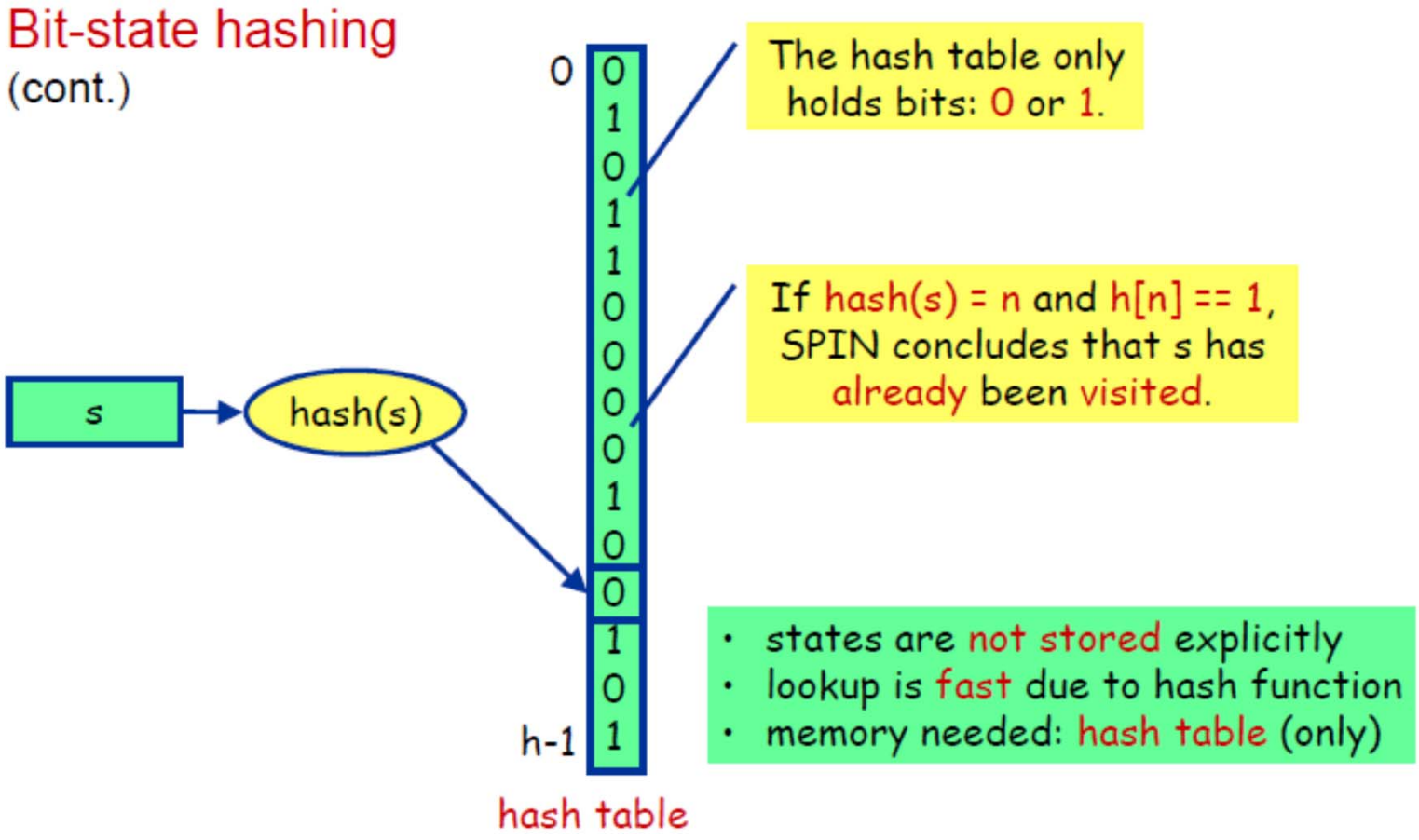



```
proctype P1() { t1a; t1b; t1c }
proctype P2() { t2a; t2b; t2c }
init { run P1(); run P2() }
```

Αλγόριθμοι Ελαχιστοποίησης (4)

- Bit-state hashing
  - Μόνο ένα δυφίο χρησιμοποιείται για την αποθήκευση μιας προσβάσιμης κατάστασης.
  - Για οποιαδήποτε κατάσταση, μια συνάρτηση κατακερματισμού χρησιμοποιείται για υπολογισμό της διεύθυνσης στον πίνακα κατακερματισμού
  - Δεν γίνεται έλεγχος για συγκρούσεις
  - Συντελεστής φορτίου = # existing bits / # reached states
  - Στόχος: συντελεστής φορτίου > 100

Αλγόριθμοι Ελαχιστοποίησης (5)



Αμετάβλητες συνθήκες

- []P, όπου η P είναι μια ιδιότητα κατάστασης.

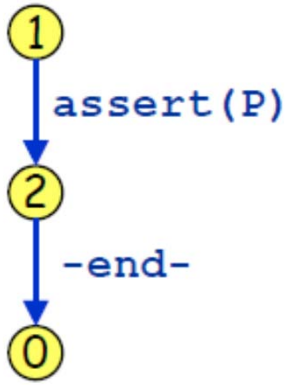
- Παραδείγματα:
  - [] mutex ! = 2
  - [] !aflag

- Η Spin προσφέρει (τουλάχιστον) 7 τρόπους ανάλυσης τέτοιων ιδιοτήτων.

Προτάσεις 1 + 2: monitor διεργασία

- Προσθέτουμε την πιο κάτω διεργασία στον κώδικά μας.

```
active proctype monitor()
{
  assert(P);
}
```

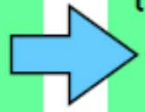


- Δύο παραλλαγές:
  - Η διεργασία δημιουργείται πρώτη
  - Η διεργασία δημιουργείται τελευταία

Πρόταση 3: διεργασία με φρουρό

- Η πιο κάτω πρόταση αποφεύγει την συνεχή ενεργοποίηση του assert το οποίο γίνεται εκτελέσιμο μόνο όταν η P γίνει ψευδής.

```
active proctype monitor()
{
  assert(P) ;
}
```

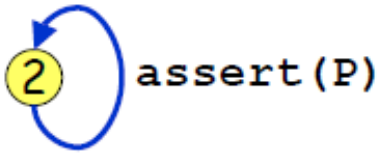


```
active proctype monitor()
{
  atomic {
    !P -> assert(P) ;
  }
}
```

Πρόταση 4: διεργασία με do

- Η πιο κάτω διεργασία φαίνεται λιγότερο αποδοτική από λειτουργική άποψη, αλλά ο αριθμός των καταστάσεων είναι μειωμένος.

```
active proctype monitor()
{
  do
  :: assert(P)
  od
}
```



Πρόταση 5 – never claim

```
never {
   do
   :: assert(P)
   od
}
```

SPIN will synchronise the never claim automaton with the automaton of the system. SPIN uses never claims to verify LTL formulae.

Πρόταση 6 - LTL

- Ελέγχουμε την αμετάβλητη συνθήκη ως [] P.

- Η Spin μεταφράζει την ιδιότητα στο πιο κάτω never claim.

```
never { ![]P
TO_init:
  if
  :: (!P) -> goto accept_all
  :: (1)  -> goto TO_init
  fi;
accept_all:
  skip
}
```

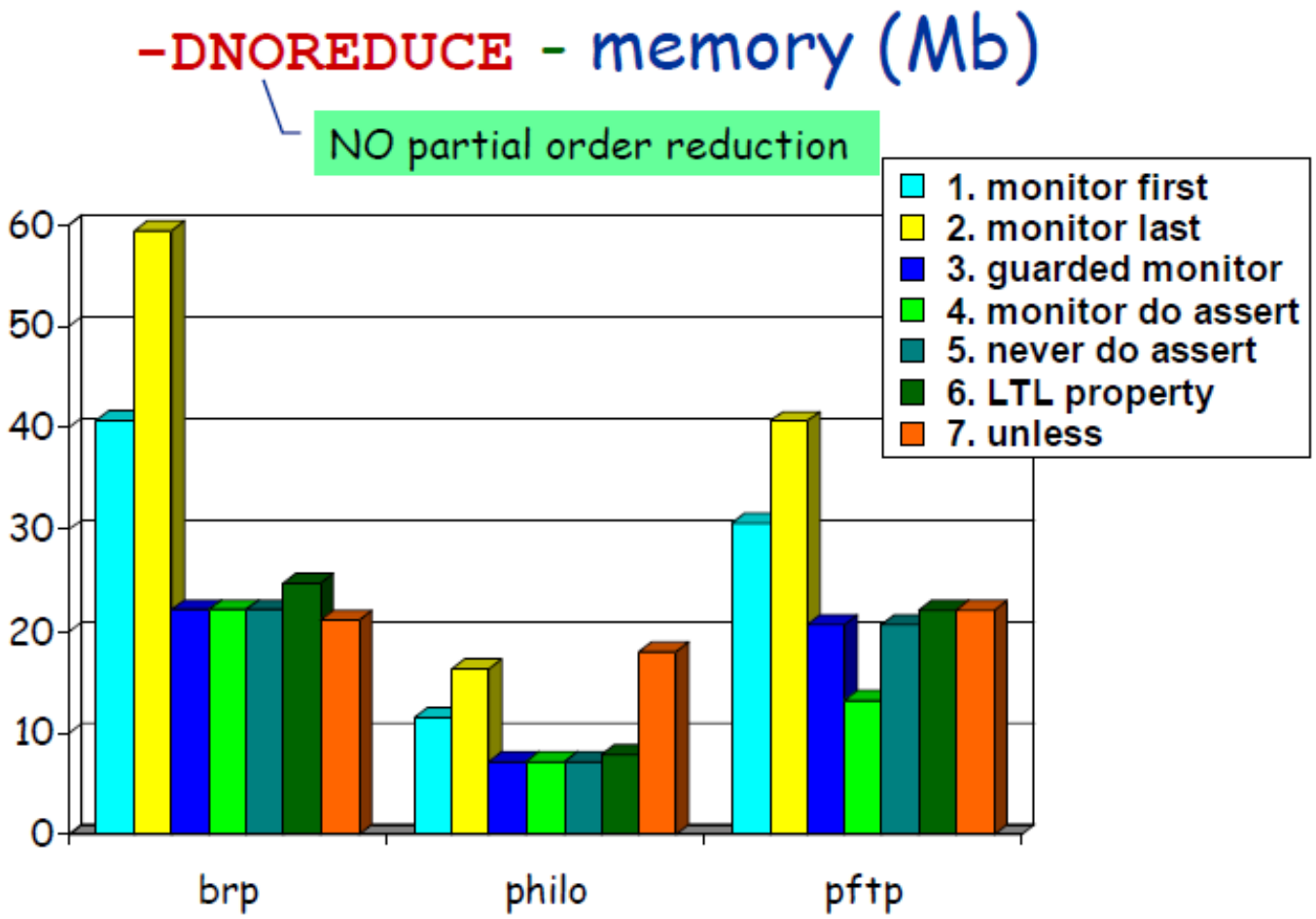
Πρόταση 7 - unless

- Εσωκλείουμε τον κορμό τουλάχιστον μιας διεργασίας ως εξής:

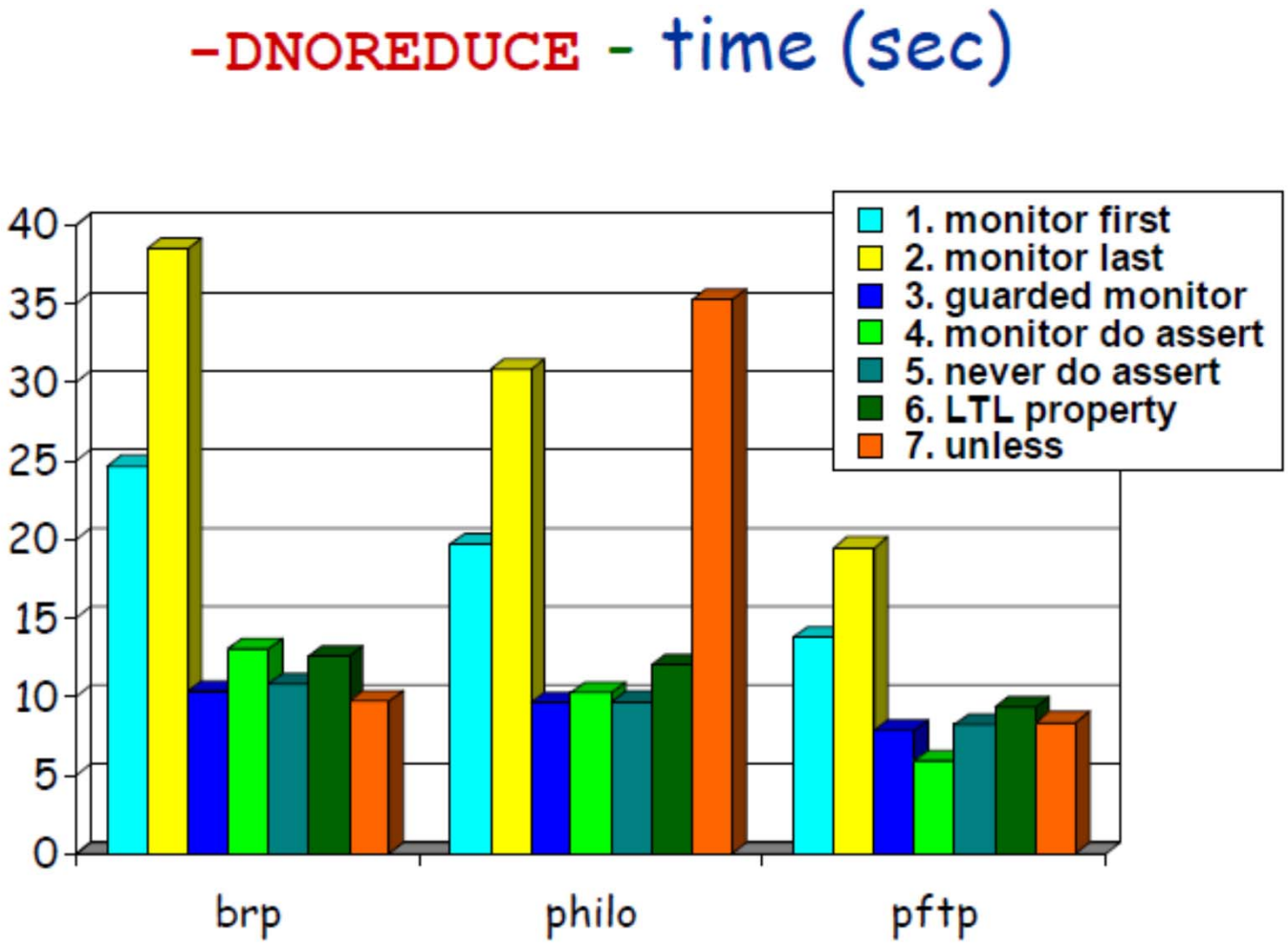
```
{ body } unless { atomic { !P -> assert(P) ; } }
```

+ Η ιδιότητα P μπορεί να χρησιμοποιήσει τοπικές μεταβλητές.
+ Δεν χρειάζεται να ορίσουμε καινούρια διεργασία
- Απαιτεί την αλλαγή του κώδικα
- Πιθανόν να έρθει σε σύγκρουση με το partial-order reduction
- Η διεργασία δεν τερματίζει

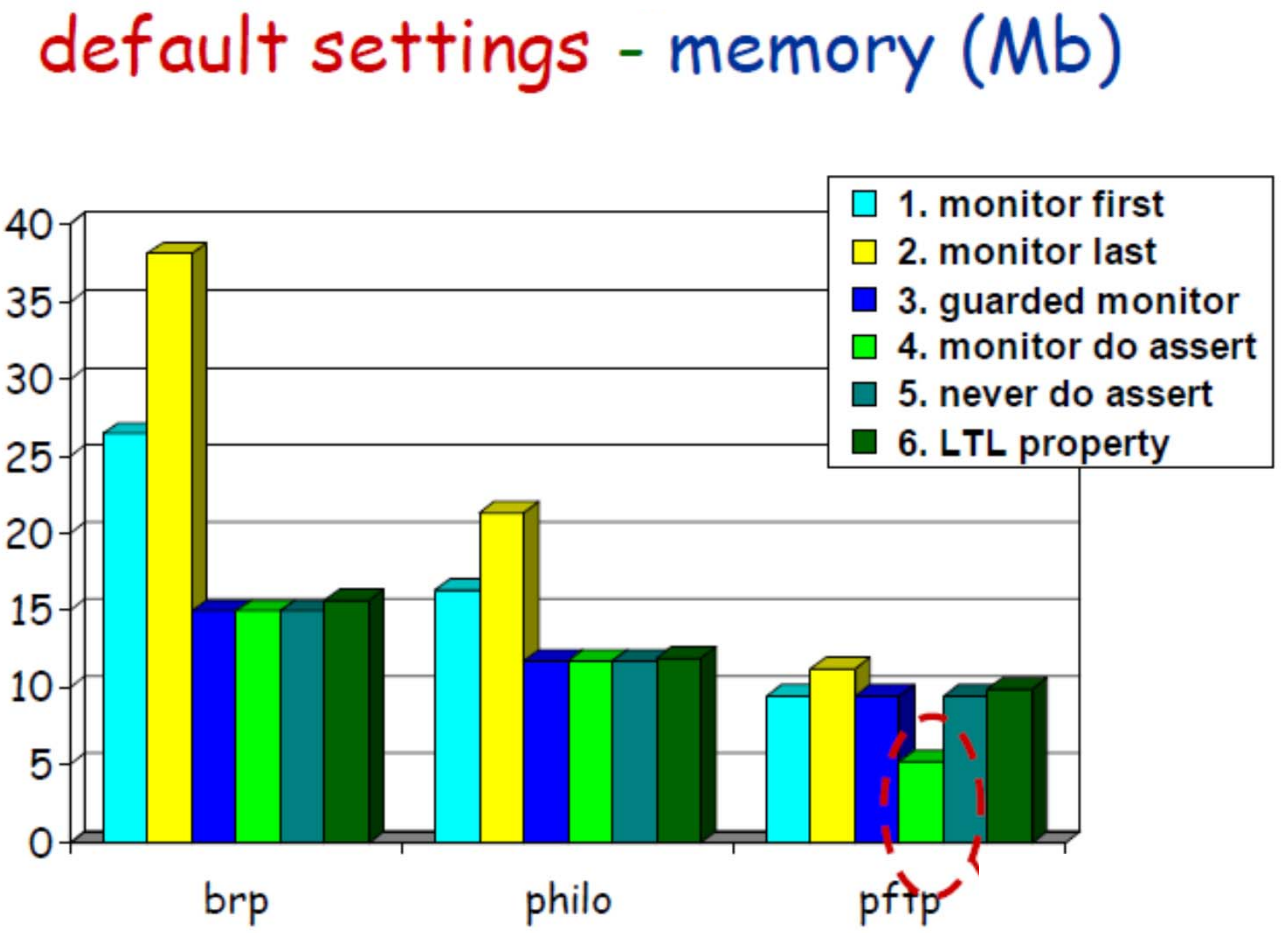
Αποτελέσματα Πειραμάτων



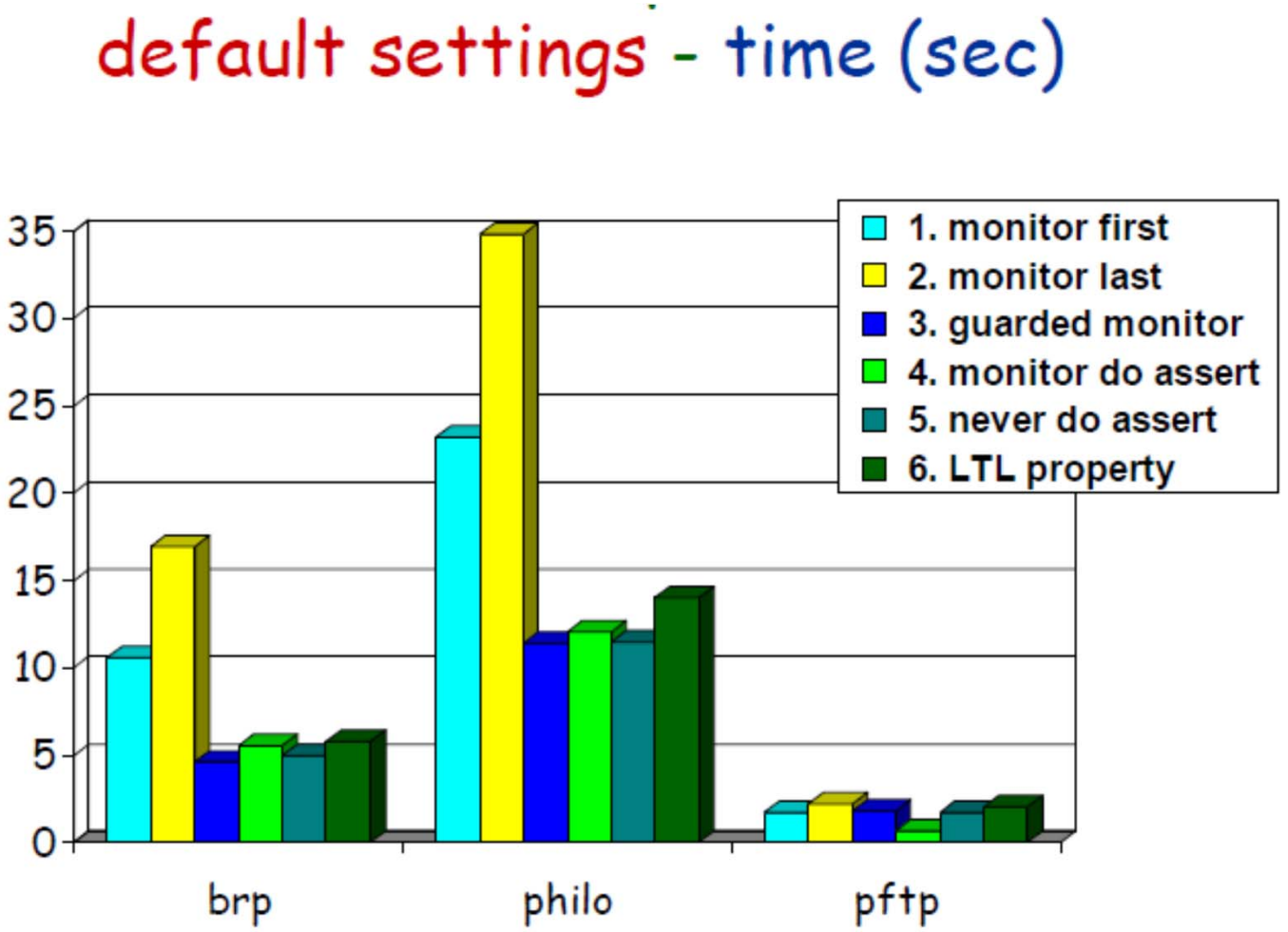
Αποτελέσματα Πειραμάτων



Αποτελέσματα Πειραμάτων



Αποτελέσματα Πειραμάτων



Further Reading

- Basic *Spin* Manual: http://spinroot.com/spin/Man/Manual.html
- Guidelines for verification with *Xspin* :
- http://spinroot.com/spin/Man/GettingStarted.html
- A sample set of exercises with *Spin:* http://spinroot.com/spin/Man/Exercises.html
- Spin Online References: http://spinroot.com/spin/Man/index.html