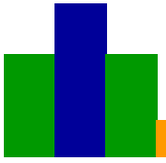
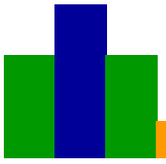


Lecţia 13:

Model checking II: Algorithmi de model checking; SMV

v1.0 (06.05.07)
Gheorghe Stefanescu — Universitatea Bucureşti

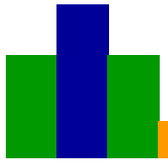
Metode de Dezvoltare Software, Sem.2
Februarie 2007— Iunie 2007



Model checking II

Cuprins:

- *Generalitati*
- Algorithmul de etichetare
- Corectitudine
- SMV - Symbolic Model Verifier
- Concluzii, diverse, etc.



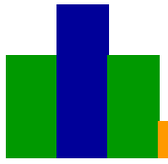
Problema de model checking

Reamintim că *problema de rezolvat cu model checking* este de tipul

(A) *Dat un model $\mathcal{M}$, o formulă CTL ϕ, şi* o stare *s, este adevărat că $\mathcal{M}$, $s \models \phi$?*

unde

- $\mathcal{M}$ este un model al sistemului şi s este o stare din model;
- ϕ este o formulă CTL pe care sistemul ar trebui să o satisfacă



Model

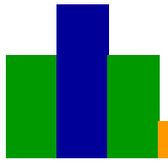
Intrebare: Ce model de sistem se foloseşte?

După cum am spus, dar accentuând faptul că modelul este *finit*:

> *un sistem este reprezentat printr-un sistem finit de tranziţii*

Comentarii:

- uzual, avem un *graf* direcţionat, etichetat, *uriaş*, adesea cu milioane de stări
- *arborii infiniţi* obţinuţi prin desfăşurarea acestor grafuri sunt buni pentru a ne dezvolta intuiţia, de a obţine rezultate teoretice, dar nu pentru a fi folosiţi în calculator



Rezultatul model checking-ului

Date $\mathcal{M}, s, \phi$, rezultatul returnat de model checker este

(1) *da*: $\mathcal{M}, s \models \phi$; ori
(2) *nu*: $\mathcal{M}, s \not\models \phi$

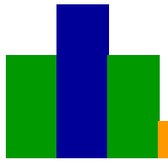
dar, adesea, în ultimul caz multe model checkere returnează şi un *drum care invalidează* ϕ.

Formulare alternativă:

(B) *Date un model* $\mathcal{M}$ *şi o formulă CTL* ϕ*, găsiţi* toate stările *s din model care satisfac* ϕ.

Note:

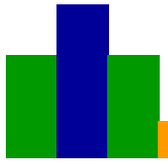
- Aceste două formulări A,B sunt *echivalente*: dacă avem algorithmi pentru una, avem şi pentru cealaltă.
- In cele ce urmează ne ocupăm de *varianta B*.



Model checking II

Cuprins:

- Generalitati
- *Algorithmul de etichetare*
- Corectitudine
- SMV - Symbolic Model Verifier
- Concluzii, diverse, etc.



Set redus de conectori CTL

Plecăm la drum cu următoarea versiune redusă de conectori CTL

$$\Gamma = \{\bot,\ \neg,\ \wedge,\ \mathrm{AF},\ \mathrm{EU},\ \mathrm{EX}\}$$

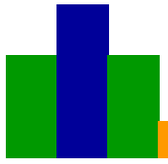
unde:

- $\bot, \neg$, şi $\wedge$ sunt utilizaţi pentru partea propoziţională
- $\mathrm{AF},\ \mathrm{EU}$, şi EX sunt utilizaţi pentru partea temporală

Folosim o procedură (neexplicitată aici) de *preprocesare* care:

1. verifică corectitudinea *sintactică* a formulei ϕ;
2. *translatează* formula într-una $\mathrm{TRANSLATE}(\phi)$ scrisă numai cu conectori din Γ.

In cele ce urmează, presupunem că ϕ este formulă CTL în formatul Γ.



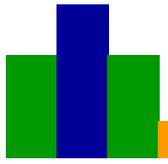
Algorithmul de etichetare

Ideea algorithmului este următoarea:

1. descompunem formula ϕ în bucăţi (sub-formule) şi aplicăm inducţia structurală pentru a eticheta graful cu sub-formulele lui ϕ (intuitiv, *o formulă etichetează un nod din graf dacă şi numai dacă este adevărată în acel nod*)

2. pentru orice astfel de sub-formulă, trecem prin graf spre a găsi valoarea într-o stare în funcţie de semnificaţia conectorului şi de valorile de adevăr ale sub-formulelor din care este compusă

Pentru punctul 2, poate fi necesar să ştim valorile sub-formulelor în diferite stări (acest lucru este necesar în cazul operatorilor temporali, dar nu şi pentru cei propoziţionali).

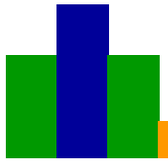


..Algorithmul de etichetare

Input: un model CTL $\mathcal{M} = (S, \rightarrow, L)$ şi o formulă CTL ϕ (în formatul Γ)

Output: mulţimea stărilor din $\mathcal{M}$ care satisfac ϕ

1. $\bot$: nici o stare nu este etichetată cu $\bot$
2. p: etichetăm cu p toate stările s cu $p \in L(s)$
3. $\neg\phi_1$: etichetăm s cu $\neg\phi_1$, dacă s nu este deja etichetată cu ϕ_1
4. $\phi_1 \wedge \phi_2$: etichetăm s cu $\phi_1 \wedge \phi_2$, dacă s este deja etichetată atât cu ϕ_1 cât şi cu ϕ_2
5. EX ϕ_1: etichetăm s cu EX ϕ_1, dacă unul din succesori este deja etichetat cu ϕ_1



...Algorithmul de etichetare

6 AF ϕ_1:

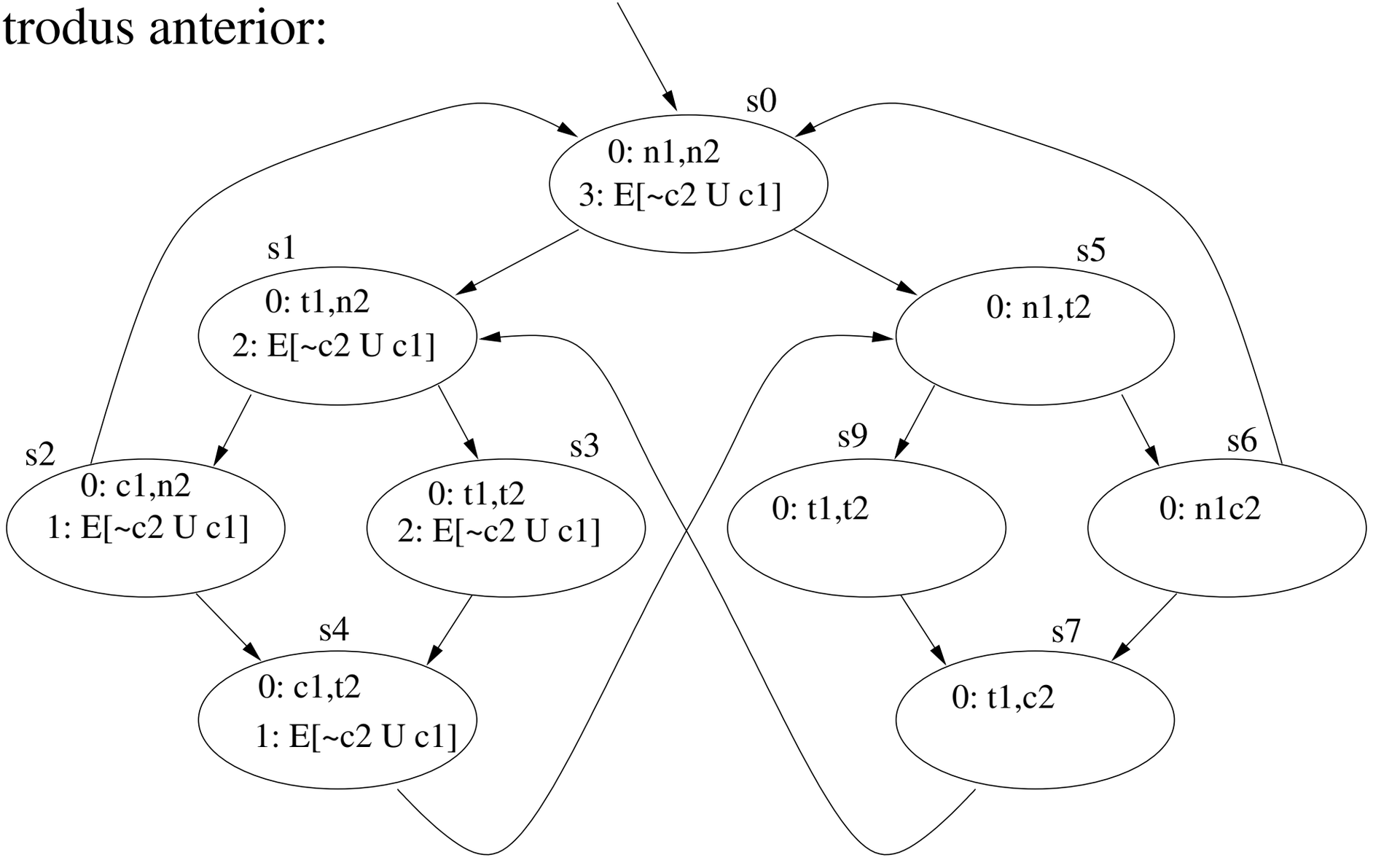
1. (marcaj iniţial) etichetăm orice *s* cu AF ϕ_1, dacă *s* este deja etichetat cu ϕ_1
2. (marcaj repetat) etichetăm orice *s* cu AF ϕ_1, dacă toţi succesorii lui *s* sunt deja etichetaţi cu AF ϕ_1
3. repetăm (2) până nu mai sunt modificări

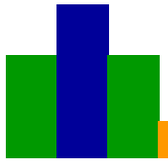
7 E$[\phi_1$ U $\phi_2]$:

1. (marcaj iniţial) etichetăm orice *s* cu E$[\phi_1$ U $\phi_2]$, dacă *s* este deja etichetat cu ϕ_2
2. (marcaj repetat) etichetăm orice *s* cu E$[\phi_1$ U $\phi_2]$, dacă *s* este deja etichetat cu ϕ_1 şi cel puţin unul din succesorii săi este deja etichetat cu E$[\phi_1$ U $\phi_2]$
3. repetăm (2) până nu mai sunt modificări

Excluderea mutuală

Verificăm $\mathrm{E}[\neg c_2 \ \mathrm{U} \ c_1]$ în modelul MUT2 de excludere mutuală introdus anterior:





EG **direct**

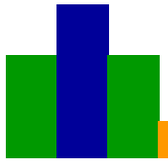
Operatorul EG poate fi tratat direct astfel:

6' EG ϕ_1:

0. etichetăm *toate* stările s cu EG ϕ_1
1. (de-marcaj iniţial) dacă ϕ_1 nu este satisfăcută în s atunci *şstergem* eticheta EG ϕ_1
2. (de-marcaj repetat) *şstergem* eticheta EG ϕ_1 din orice stare s, dacă nici unul din succesori nu este marcat cu EG ϕ_1
3. repetăm (2) până nu mai sunt modificări

Această tratare diferită se bazează pe următoarea caracterizare a operatorului EG ca punct fix maximal

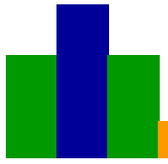
$$\text{EG}\ \phi \equiv \phi \wedge \text{EX}\ \text{EG}\ \phi$$



O varianta imbunatatita

Varianta mai performantă:

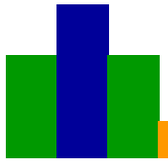
- Utilizăm EX, EU, şi EG în loc de EX, EU, şi AF
- Tratăm EX şi EU ca înainte
- Pentru EG ϕ
  - ne restrângem la stările care satisfac ϕ
  - găsim componentele tare-conexe maximale SCC (i.e., regiuni în care orice nod este conectat cu orice alt nod din regiune)
  - folosim graful restricţionat dat de SCC-uri



Pseudo-cod

function $\text{SAT}(\phi)$:
/* precondition: ϕ este o formulă CTL arbitrară */
/* postcondition: $\text{SAT}(\phi)$ returnează stările care satisfac ϕ */
begin function
case
ϕ este $\top$: **return** S
ϕ este $\bot$: **return** $\emptyset$
ϕ este formulă atomică: **return** $\{s \in S: \phi \in L(s)\}$
ϕ este $\neg\phi_1$: **return** $S \setminus \text{SAT}(\phi_1)$
ϕ este $\phi_1 \wedge \phi_2$: **return** $\text{SAT}(\phi_1) \cap \text{SAT}(\phi_2)$
ϕ este $\phi_1 \vee \phi_2$: **return** $\text{SAT}(\phi_1) \cup \text{SAT}(\phi_2)$
ϕ este $\phi_1 \rightarrow \phi_2$: **return** $\text{SAT}(\neg\phi_1 \vee \phi_2)$

(...cont.)



..Pseudo-cod

(...cont.)

ϕ este $\mathrm{AX}\phi_1$: **return** $\mathrm{SAT}(\neg\mathrm{EX}\neg\phi_1)$

ϕ este $\mathrm{EX}\phi_1$: **return** $\mathrm{SAT}_{EX}(\phi_1)$

ϕ este $\mathrm{A}[\phi_1 \mathrm{U} \phi_2]$: **return** $\mathrm{SAT}(\neg(\mathrm{E}[\neg\phi_2 \mathrm{U}(\neg\phi_1 \wedge \neg\phi_2)] \vee \mathrm{EG}\neg\phi_2))$

ϕ este $\mathrm{E}[\phi_1 \mathrm{U} \phi_2]$: **return** $\mathrm{SAT}_{EU}(\phi_1, \phi_2)$

ϕ este $\mathrm{EF}\phi_1$: **return** $\mathrm{SAT}(\mathrm{E}[\top \mathrm{U}\ \phi_1])$

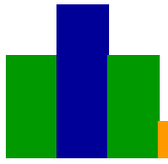
ϕ este $\mathrm{EG}\phi_1$: **return** $\mathrm{SAT}(\neg\mathrm{AF}\neg\phi_1])$

ϕ este $\mathrm{AF}\phi_1$: **return** $\mathrm{SAT}_{AF}(\phi_1)$

ϕ este $\mathrm{AG}\phi_1$: **return** $\mathrm{SAT}(\neg\mathrm{EF}\neg\phi_1)$

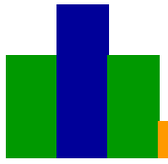
end case

end function



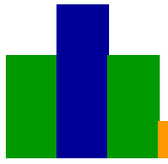
..Pseudo-cod

function $\text{SAT}_{EX}(\phi)$:
/* pre: ϕ este o formulă CTL arbitrară */
/* post: $\text{SAT}_{EX}(\phi)$ returnează stările care satisfac EX ϕ */
local var X, Y
begin
  $X := \text{SAT}(\phi)$;
  $Y := \{s_0 \in S\colon s_0 \rightarrow s_1$, pentru un $s_1 \in X\}$;
  return Y
end



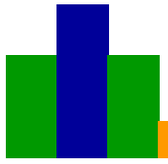
..Pseudo-cod

function $\text{SAT}_{AF}(\phi)$:
/* pre: ϕ este o formulă CTL arbitrară */
/* post: $\text{SAT}_{AF}(\phi)$ returnează stările care satisfac AF ϕ */
local var X, Y
begin
  $X := S$;
  $Y := \text{SAT}(\phi)$;
  repeat until $X = Y$
    begin
      $X := Y$;
      $Y := Y \cup \{s \in S:$ pentru toţi s' cu $s \rightarrow s'$, avem $s' \in Y\}$;
    end
  return Y
end



...(pseudo-code)

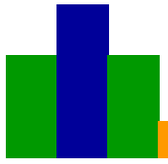
function $\text{SAT}_{EU}(\phi)$:
/* pre: ϕ este o formulă CTL arbitrară */
/* post: $\text{SAT}_{EU}(\phi, \psi)$ returnează stările care satisfac $\text{E}[\phi \text{U} \psi]$ */
local var W, X, Y
begin
  $W := \text{SAT}(\phi)$;
  $X := S$;
  $Y := \text{SAT}(\psi)$;
  repeat until $X = Y$
    begin
      $X := Y$;
      $Y := Y \cup (W \cap \{s \in S:$ există s' cu $s \rightarrow s'$ şi $s' \in Y\}$;
    end
  return Y
end



Problema “exploziei starilor”

Comentarii:

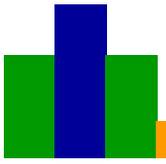
- algorithmul de etichetare este destul de eficient [linear în mărimea modelului]
- ... dar modelul însuşi poate fi larg, exponenţial în numărul de componente (rulând în paralel 10 thread-uri, fiecare cu 10 stări obţinem un sistem cu $10^{10} = 10,000,000,000$ stări!)
- problema *exploziei stărilor* se referă la tendinţa spaţiului stărilor de a deveni foarte mare
- curent, problema este *nerezolvată* (în cazul general)



Tehnici de tratare a “exploziei starilor”

Comentarii: Există tehnici puternice pentru a trata anumite cazuri particulare, e.g.:

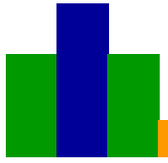
1. *structuri de date eficiente* - exemplu: *OBDD-uri* (diagrame de decizie ordonate binare); OBDD-urile sunt utilizate pentru a reprezenta mulţimi de stări, nu stări individuale)
2. *abstracţie* - se poate face abstracţie de variabile din model irelevante pentru formula verificată
3. *reducere parţial ordonată* - execuţii diferite pot fi echivalente din punctul de vedere al formulei verificate; aceasta tehnică verifică formula pe o singură execuţie dintr-o astfel de clasă
4. *inducţie* - se foloseşte dacă avem un număr mare de procese *identice*
5. *compunere* - se încearcă spargerea problemei în bucăţi mici, care să se trateaze separat



Model checking II

Cuprins:

- Generalitati
- Algorithmul de etichetare
- *Corectitudine*
- SMV - Symbolic Model Verifier
- Concluzii, diverse, etc.



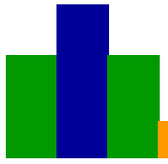
Puncte fixe

Definiţii, convenţii:

- Fie S o mulţime de stări şi $F : \mathcal{P}(S) \to \mathcal{P}(S)$ o funcţie.
- F se numeşte *monotonă* dacă: $X \subseteq Y$ implică $F(X) \subseteq F(Y)$.
- Un $X \in \mathcal{P}(S)$ se numeşte *punct fix* dacă: $F(X) = X$.
- Notăm $F^k(X) = F(F(\ldots F(X)\ldots))$, unde F se aplică de k ori.
- F se numeşte *continuă* dacă

$$F(\bigcup X_i) = \bigcup F(X_i)$$

pentru orice secvenţă crescătoare $X_0 \subseteq X_1 \subseteq X_2 \ldots$.



teorema lui Kleene

Teorema Kleene:

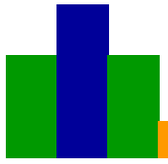
- *O funcţie monotonă şi continuă are un cel mai mic punct fix, notat* $\mu Z.F(Z)$*, şi un cel mai mare punct fix, notat* $\nu Z.F(Z)$*.*
- *Următoarele formule pot fi folosite spre a le calcula:*

$$\mu Z.F(Z) = \emptyset \cup F(\emptyset) \cup F(F(\emptyset)) \cup \ldots$$

şi

$$\nu Z.F(Z) = S \cap F(S) \cap F(F(S)) \cap \ldots$$

In particular, dacă S este finită, să zicem cu n elemente, continuitatea nu este necesară. Intr-adevăr,



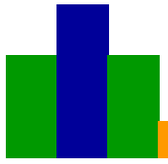
Puncte fixe in multimi finite

Teoremă: Dacă S are n elemente şi F este monotonă, atunci

$$\mu Z.F(Z) = F^n(\emptyset) \quad şi \quad \nu Z.F(Z) = F^n(S)$$

Dem.:

(1)
- Clar, $\emptyset \subseteq F^1(\emptyset)$
- Aplicând F obţinem $F^1(\emptyset) \subseteq F^2(\emptyset)$
- Repetând, obţinem : $\emptyset \subseteq F^1(\emptyset) \subseteq F^2(\emptyset) \subseteq \ldots \subseteq F^{n+1}(\emptyset)$
- Lanţul de incluziuni nu poate fi mereu strict, deci la un pas '$\subseteq$' este egalitate.
- Concluzie: există $0 \leq i_0 \leq n$ cu $F^{i_0}(\emptyset) = F(F^{i_0}(\emptyset))$, adică $F^{i_0}(\emptyset)$ este punct fix

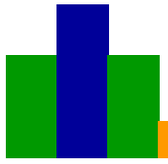


..Puncte fixe in multimi finite

(2) $F^i(\emptyset)$ este mai mic decât orice alt punct fix:

- Fie X punct fix
- Cum $\emptyset \subseteq X$, aplicând F obţinem $F(\emptyset) \subseteq F(X) = X$
- Repetând, obţinem $F^k(\emptyset) \subseteq X$ penru orice k, deci $F^{i_0}(\emptyset) \subseteq X$.

(3) Cazul celui mai mare punct fix este similar, dar se începe cu S (nu cu $\emptyset$) şi se inversează incluziunile.



Corectitudinea lui SAT_{EU}

Notăm cu $[\![\phi]\!]$ mulţimea stărilor care satisfac ϕ şi cu F aplicaţia

$$Z \mapsto [\![\psi]\!] \cup ([\![\phi]\!] \cap \{s\colon \text{există } s' \text{ cu } s \to s' \text{ şi } s' \in Z\}$$

Teoremă: Dacă F este ca mai sus şi $n = |S|$, atunci:

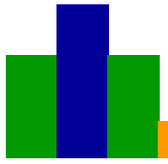
(1) F este monotonă

(2) $[\![\mathrm{E}[\phi \mathrm{U} \psi]]\!]$ este cel mai mic punct fix al lui F

(3) $[\![\mathrm{E}[\phi \mathrm{U} \psi]]\!] = F^{n+1}(\emptyset)$

Dem:

(1) Aplicaţia $H(Z) = \{s\colon \text{există } s' \text{ cu } s \to s' \text{ şi } s' \in Z\}$ este monotonă. F se obţine din H prin intersecţii şi reuniuni, deci este şi ea monotonă.



..Corectitudinea lui SAT_{EU}

(2) Analizând stările $F^k(\emptyset)$ observăm că

— $F^0(\emptyset)$ conţine stările din $[\![\psi]\!]$;

— $F^1(\emptyset)$ conţine stările din $[\![\psi]\!]$, ori pe cele din $[\![\phi]\!]$ care au tranziţii (într-un pas) la strări din $[\![\psi]\!]$;

— ...

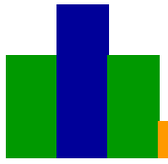
In general,

$F^k(\emptyset)$ conţine acele stări care au un drum de lungime cel mult k la o stare din $[\![\psi]\!]$ trecând numai prin stări din $[\![\phi]\!]$

Deci reuniunea tuturor $F^k(\emptyset)$ dă $[\![\mathrm{E}[\phi \ \mathrm{U} \ \psi]]\!]$.

Stim că lanţul $F^k(\emptyset)$ este crescător şi $F^{n+1}(\emptyset)$ este punct fix, deci reuniunea tuturor $F^k(\emptyset)$ este $F^{n+1}(\emptyset)$.

(3) Este deja demonstrat în (2).



..Corectitudinea lui SAT_{EU}

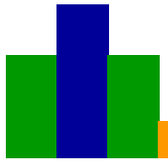
Observaţia finală este că SAT_{EU} foloseşte un proces iterativ echivalent, mai simplu şi mai rapid:

In loc de

$$F^{k+1}(\emptyset)$$
$$= [\![\psi]\!] \cup ([\![\phi]\!] \cap \{s\colon \text{există } s' \text{ cu } s \rightarrow s' \text{ şi } s' \in F^k(\emptyset)\}$$

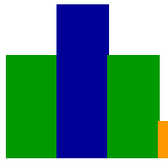
se foloseşte procesul iterativ

$$F_1^{k+1}(\emptyset)$$
$$= F_1^k(\emptyset) \cup ([\![\phi]\!] \cap \{s\colon \text{există } s' \text{ cu } s \rightarrow s' \text{ şi } s' \in F_1^k(\emptyset)\}$$



SAT_{EG}

function $\text{SAT}_{EG}(\phi)$:
/* pre: ϕ este o formulă CTL arbitrară */
/* post: $\text{SAT}_{EG}(\phi)$ returnează stările care satisfac EG ϕ */
local var X, Y
begin
  $X := \emptyset$;
  $Y := \text{SAT}(\phi)$;
  repeat until $X = Y$
    begin
      $X := Y$;
      $Y := Y \cap \{s \in S:$ există s' cu $s \rightarrow s'$ şi $s' \in Y\}$;
    end
  return Y
end



Corectitudinea lui SAT_{EG}

Notăm cu $[\![\phi]\!]$ mulţimea stărilor care satisfac ϕ şi cu G aplicaţia

$$Z \mapsto [\![\phi]\!] \cap \{s\colon \text{există } s' \text{ cu } s \longrightarrow s' \text{ şi } s' \in Z\}$$

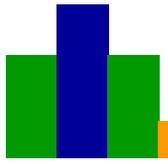
Teoremă: *Dacă F este ca mai sus şi $n = |S|$, atunci:*

(1) G este monotonă

(2) $[\![\mathrm{EG}\ \phi]\!]$ este cel mai mare punct fix al lui G

(3) $[\![\mathrm{EG}\ \phi]\!] = G^{n+1}(S)$

Demonstraţia este similară cu cea din cazul precedent. $\triangle$



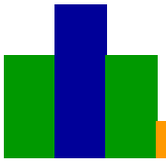
..Corectitudinea lui SAT_{EG}

In fine, în loc de procesul iterativ

$$G^{k+1}(S) = [\![\phi]\!] \cap \{s\colon \text{există } s' \text{ cu } s \rightarrow s' \text{ şi } s' \in G^k(S)\}$$

algoritmul pentru SAT_{EG} foloseşte un proces iterativ mai simplu, anume

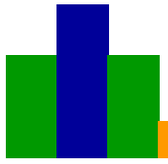
$$G_1^{k+1}(S) = G^k(S) \cap \{s\colon \text{există } s' \text{ cu } s \rightarrow s' \text{ şi } s' \in G^k(S)\}$$



Model checking II

Cuprins:

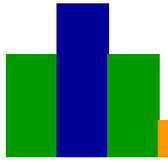
- Generalitati
- Algorithmul de etichetare
- Corectitudine
- *SMV - Symbolic Model Verifier*
- Concluzii, diverse, etc.



SMV - Symbolic Model Verifier

SMV (Symbolic Model Verifier) a fost unul din primele model checkere. Se bazează pe CTL, a fost introdus la începepului anilor 1990 şi a avut un mare impact în domeniul verificării

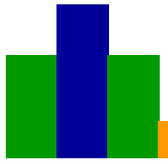
- SMV a fost dezvoltat la CMU, vezi www.cs.cmu.edu/~modelcheck/smv.html
- conţine un limbaj pentru a descrie modele (diagrame)
- poate verifica validitatea formulelor CTL în astfel de modele
- rezultatul este fie *true*, fie *un trace* care arată *de ce este formula falsă*



SMV - Sintaxa

SMV - Sintaxa (informal)

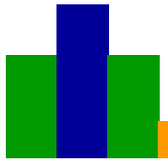
- programele SMV constă din unul ori mai multe module (unul din ele trebuie sa fie **main**)
- fiecare modul declară variabile şi le asignează valori
- asignările folosesc două cuvinte cheie:
  - **initial** (spre a indica starea iniţială) şi
  - **next** (spre a indica următoare stare din diagrma de tranziţii)
- asignările pot fi nedeterministe - acest lucru este indicat folosind notaţia de mulţime $\{\ldots\}$ (se alege un element din această mulţime)



..SMV - Sintaxa

(...cont.)

- se poate folosi cunstrucţia **case**
  - condiţiile din faţa lui ‘:’ se parsează de sus în jos şi prima găsită adevărată se execută
  - se poate folosi o variantă *default* (care este mereu adevarată, notată cu 1) uzual pusă ultima în instrucţiunea **case**
- un modul poate avea specificaţii proprii care trebuiesc verificate, scrise în sintaxa CTL (dar cu `&`, `|`, `->`, `!` în loc de $\wedge, \vee, \rightarrow, \neg$)



SMV, exemplul 1

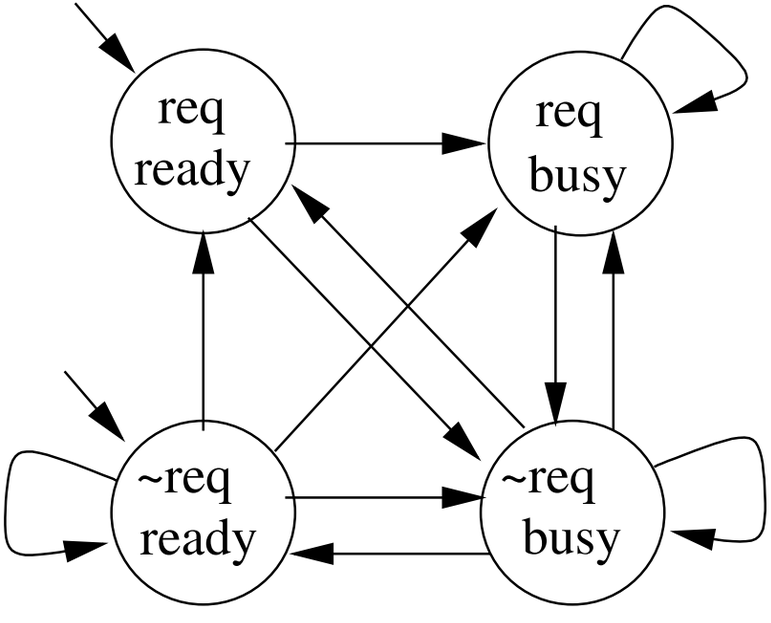
Exemplul 1, destul de tipic, este următorul:

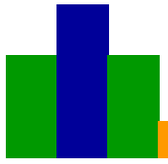
- modelează o parte a unui sistem care trece de la `ready` la `busy`, fie din cauze interne (invizibile în model), fie din cauza unei cereri `request`;
- sistemul trece din `busy` în `ready` în mod nedeterminist (fară un motiv vizibil)
- în acest model, verificăm formula

```
AG(request -> AF status = busy)
```

...(SMV, 1st example)

```
MODULE main
VAR
    request :  boolean;
    status :  {ready,busy};
ASSIGN
    init(status) := ready;
    next(status) :=
      case
        request :  busy;
        1 :  {ready,busy};
      esac;
SPEC
    AG(request -> AF status = busy)
```



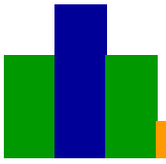


SMV, exemplul 2

Al doilea program, folosind mai multe module, este următorul:

- programul modelează un contor de la 000 la 111
- un modul `counter_cell` este instanţiat de trei ori cu numele `bit0, bit1, bit2`
- `counter_cell` are un parametru formal
- punctul '.' se foloseşte pentru a accesa o valoare particulară dintr-o instanţă (`m.v` denotă variabila `v` din modulul `m`)
- verificăm formula

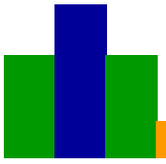
```
AG AF bit2.carry_out
```



..SMV, exemplul 2

```
MODULE main
VAR
    bit0 :  counter_cell(1);
    bit1 :  counter_cell(bit0.carry_out);
    bit2 :  counter_cell(bit1.carry_out);
SPEC
    AG AF bit2.carry_out

MODULE counter_cell(carry_in)
VAR
    value :  boolean;
ASSIGN
    init(value) := 0;
    next(value) := value + carry_in mod 2;
DEFINE
    carry_out := value & carry_in;
```

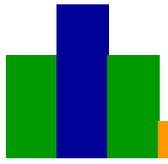


..SMV, exemplul 2

Notă:

- instrucţiunea `define` se foloseşte pentru a evita creştea spaţiului stărilor
- efectul său se poate obţine şi cu:

```
VAR
    carry_out :  boolean;
ASSIGN
    carry_out := value & carry_in;
```



Compunere sincrona vs. asincrona

Execuţii:

- In lipsa altor specificări, modulele SMV se compun *sincron*:

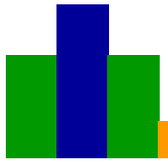
  la fiecare ciclu de ceas, fiecare modul execută o tranziţie

  (folosită în special pentru verificarea de hardware)

- Modulele din SMV se pot compune şi *asincron*

  la fiecare ciclu de ceas, SMV alege aleator un modul şi execută o tranziţie de acolo

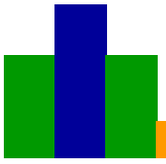
  (folosită în special pentru verificarea de programe paralele ori protocoale de comunicare)



SMV, exemplul 3 - Excluderea Mutuala

Am văzut anterior un model CTL pentru ‘excluderea mutuala’ - aici dăm o implementare SMV. Noile caracteristici sunt:

- există un modul `main` cu (1) o variabilă `turn` care determină care proces intră în secţiunea sa critică şi (2) două instanţieri ale modulului `prc`
- datorită variabilei `turn` sistemul de tranziţii este un pic mai complicat
- o caracteristică importantă este prezenţa instrucţiunii **fairness**: ea conţine o formulă CTL ϕ şi restricţionează căutarea la drumurile în care ϕ este true de o infinitate de ori (`running` este un cuvânt cheie SMV care indică faptul că respectivul modul este selectat de o infinitate de ori)



..SMV, exemplul 3

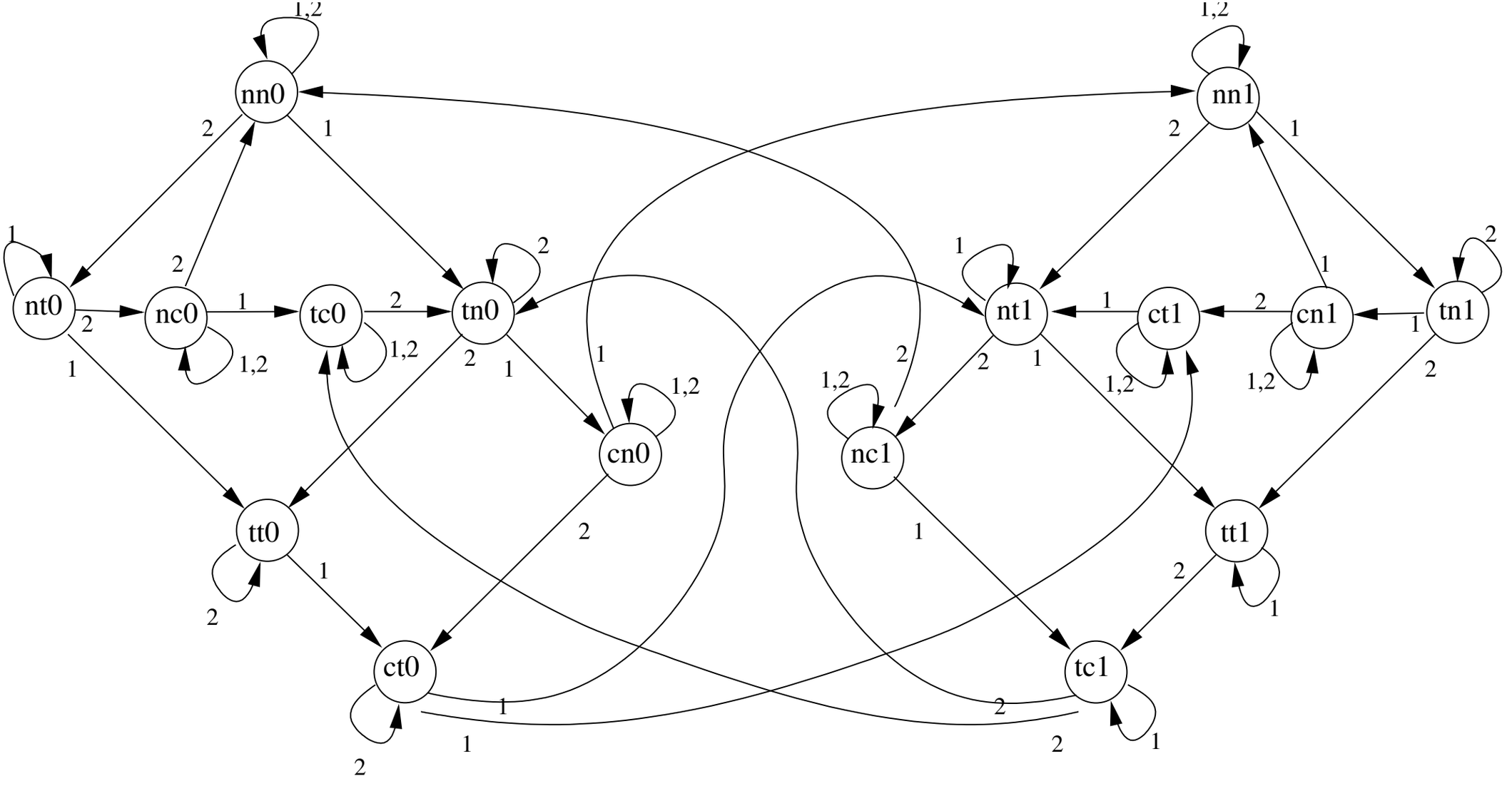
```
MODULE main
    VAR
      pr1 :  process prc(pr2.st, turn, 0);
      pr2 :  process prc(pr1.st, turn, 1);
      turn :  boolean;
    ASSIGN
      init(turn) := 0;
    --safety
    SPEC AG!((pr1.st = c) & (pr2.st = c))
    --liveness
    SPEC AG((pr1.st = t) -> AF (pr1.st = c))
    SPEC AG((pr2.st = t) -> AF (pr2.st = c))
    --no strict sequencing
    SPEC EF(pr1.st = c & E[pr1.st = c U
           (!pr1.st = c & E[!  pr2.st = c U pr1.st = c ])])
```

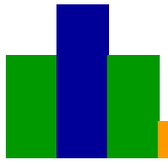
..SMV, exemplul 3

```
MODULE prc(other-st, turn, myturn)
    VAR
       st :  {n, t, c};
    ASSIGN
       init(st) := n;
       next(st) :=
          case
             (st = n) :  {t, n};
             (st = t) & (other-st = n) :  c;
             (st = t) & (other-st = t) & (turn = myturn) :  c;
             (st = c) :  {c, n};
             1 :  st;
          esac;
       next(turn) :=
          case
             turn = myturn & st = c :  !turn;
             1 :  turn;
          esac;
    FAIRNESS running
    FAIRNESS !(st = c)
```

..SMV, exemplul 3

Exculdere mutuală în SMV:





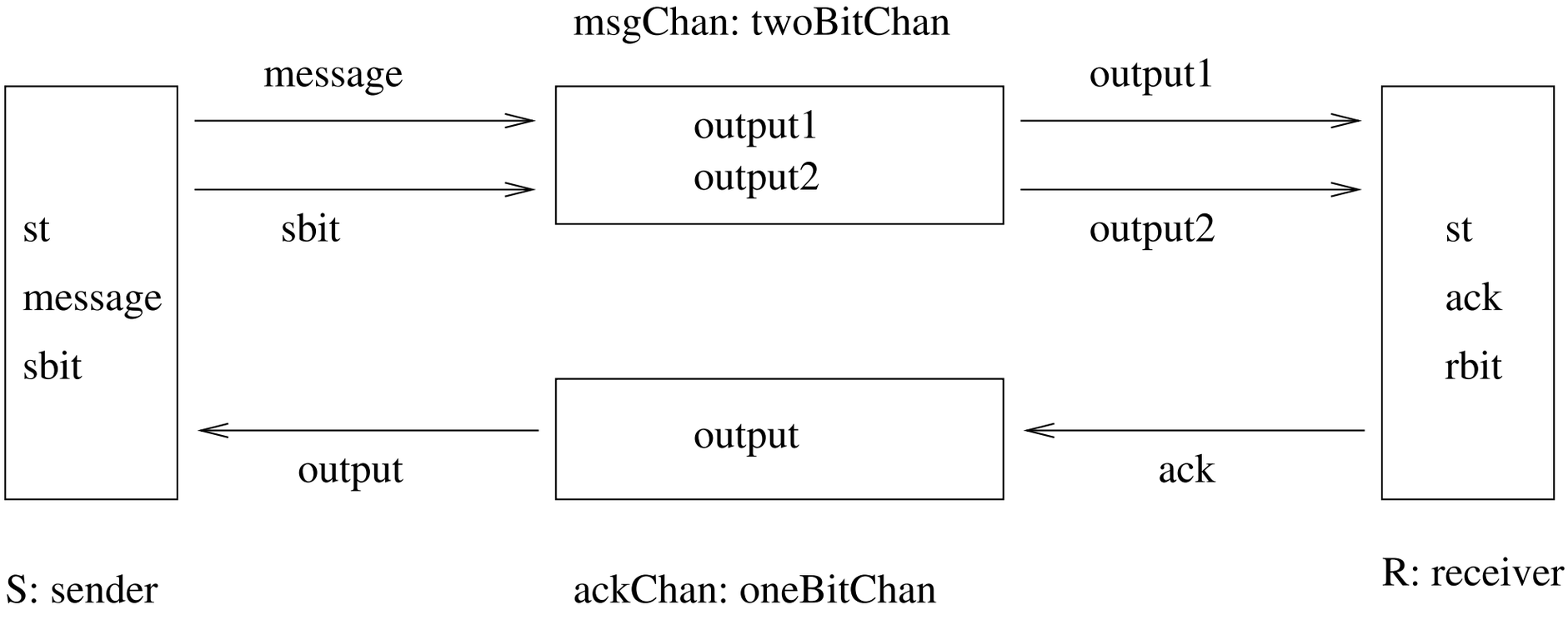
ABP: Alternating Bit Protocol

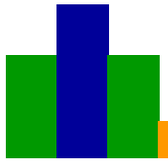
Prezentare:

- *ABP (Alternating Bit Protocol)* este un protocol care *transmite corect* date prin *canale defecte* (ce pot pierde ori duplica datele)
- ABP foloseşte două astfel de canale defecte între expeditor şi destinatar: unul pentru a trimite *datele*, celălalt pentru *confirmări*
- în cazul unei transmisii eronate, data se *retransmite*;
- pentru a-şi atinge scopul, APB ţine socoteala acestor transmisii repetate folosind un *bit de control* care se complementează când se trece de la o dată la alta
- expeditorul anexează bitul său de control la dată şi o *retirimite* până ce primeşte *bitul înapoi* pe canalul de confirmare

...(ABP)

Fogura descrie structura protocolului ABP:



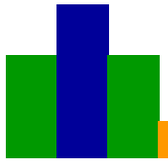


...(ABP)

```
00 MODULE sender(ack)
01 VAR
02    st :  {sending, sent};
03    message :  boolean;
04    sbit :  boolean;
05 ASSIGN
06    init(st) := sending;
07    next(st) :=
08       case
09          ack = sbit & !(st = sent) :  sent;
10          1 :  sending;
11       esac;
12    next(message) :=
13       case
14          st = sent :  {0, 1};
15          1 :  message;
16       esac;
17    next(sbit) :=
18       case
19          st = sent :  !sbit;
20          1 :  sbit;
21       esac;
22 FAIRNESS running
23 SPEC AG AF st = sent
```

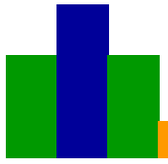
...(ABP)

```
24 MODULE receiver(message, sbit)
25 VAR
26    st :  {receiving, received};
27    ack :  boolean;
28    rbit :  boolean;
29 ASSIGN
30    init(st) := receiving;
31    next(st) :=
32       case
33          sbit = rbit & !(st = received) :  received;
34          1 :  receiving;
35       esac;
36    next(ack) :=
37       case
38          st = received :  sbit;
39          1 :  ack;
40       esac;
41    next(rbit) :=
42       case
43          st = received :  !rbit;
44          1 :  rbit;
45       esac;
46 FAIRNESS running
47 SPEC AG AF st = received
```



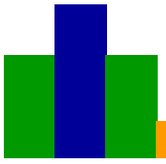
...(ABP)

```
48 MODULE oneBitChan(input)
49 VAR
50    output :  boolean;
51 ASSIGN
52    next(output) := {input, output};
53 FAIRNESS running
54 FAIRNESS (input = 0 -> AF output = 0) & (input = 1
55    -> AF output = 1)
56
57 MODULE twoBitChan(input1, input2)
58 VAR
59    output1 :  boolean;
60    output2 :  boolean;
61 ASSIGN
62    next(output2) := {input2, output2};
63    next(output1) :=
64       case
65          input2 = next(output2) :  input1;
66          1 :  {input1, output1};
67       esac;
68 FAIRNESS running
69 FAIRNESS (input1 = 0 -> AF output1 = 0) & (input1 = 1
70    -> AF output1 = 1) & (input2 = 0 -> AF output2 = 0)
71    & (input2 = 1 -> AF output2 = 1)
```



...(ABP)

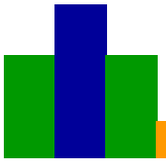
```
72 MODULE main
73 VAR
74    S : process sender(ackChan.output);
75    R : process receiver(msgChan.output1, msgChan.output2);
76    msgChan :  process twoBitChan(S.message, S.sbit);
77    ackChan :  process oneBitChan(R.ack);
78 ASSIGN
79    init(S.sbit) := 0;
80    init(R.rbit) := 0;
81    init(R.ack) := 1;
82    init(msgChan.output2) := 1;
83    init(ackChan.output) := 1;
84 SPEC AG(S.st = sent & S.message = 1 -> msgChan.output1 = 1)
```



Model checking II

Cuprins:

- Generalitati
- Algorithmul de etichetare
- Corectitudine
- SMV - Symbolic Model Verifier
- *Concluzii, diverse, etc.*



Concluzii, diverse, etc.

a se insera...