

# Formalne Metode u oblikovanju sustava

FER

drugi ciklus predavanja  
ver. 0.1.7  
nadm.zadnje.rev.: 17. travnja 2009.

# Ponavljjanje

- 1 ciklus razvoja programa i formalna metode
- 2 verifikacija, provjera modela
- 3 komunicirajući automati (CFSM)

$I \models S$  ili  $M \models \varphi$

- 1 provjera modela za programe
- 2 provjera modela za sklopovlje

# Uvod

- 1 formalna verifikacija komunikacijskih protokola
- 2 ciljani sustav: komunikacijski protokoli u raspodijeljenim sustavima (*eng. distributed systems*)
- 3 modeliranje implementacije: Promela (**Pro**cess **meta** language)
- 4 sustav za verifikaciju: SPIN (**S**imple **P**rotocol **I**nterpreter)

## U nastavku:

Cilj je analiza: modeliranje i verifikacija ponašanja **raspodjeljenih sustava (konkurentnih reaktivnih sustava)** pomoću provjere modela (*eng. model checking*)

Što je konkurentni reaktivni sustav ? U kakvoj je vezi s protokolima i distribuiranim sustavima? ...

# Općenito o konkurentnim i reaktivnim sustavima

Sustav promatramo kao skup komunicirajućih **procesa**.

**Procesi** komuniciraju pomoću:

- 1 izmjene poruka (*eng. message passing*)
- 2 preko repova—dijeljenje resursa (*eng. resource sharing*)

## Primjer za vježbu:

- o na \*WIN platformi pokrenuti TASK MANAGER ili na \*nux iz konzole npr. naredbu `ps -ax`
- o Skicirajte međusobno povezane procese! Uočite procese karakteristične za operativni sustav odnosno pojedinu aplikaciju!

## Distribuirani sustav

**Distribuirani sustav** procesi su raspoređeni u prostoru.  
(od više procesa na jednom procesoru preko *multi-core procesora* do mreže procesora)

## Konkurentni sustav

**Procesi** se izvode sa različitim varijantama paralelnosti  
(istovremenosti)

## Reaktivni sustav

**Reaktivni sustav** mora **odmah** reagirati na signale, poruke . . . okoline

- 1 komunikacijski protokol u *raspodjeljenom, konkurentnom reaktivnom sustavu* je podrška za izmjenu informacija/podataka među procesima.
- 2 kritičnost: protokol mora biti kvalitetan i robustan: **analiza, modeliranje i verifikacija**
- 3 kod distribuiranih aplikacija (web, klijent–server, dretve . . . ) nestaje precizne granice između protokola i korisničke aplikacije

# Kako naučiti verificirati protokol/raspodjeljenu aplikaciju ?

... ili kako primjeniti *provjeru modela* na zadani problem ?

- 1 Modeliranje: definirati model u `Promela` jeziku
- 2 Analiza–*provjera modela*: pozvati `SPIN` sa datotekom–modelom opisanim jezikom `Promela` i provjeriti valjanost uvjeta formulama *LTL* logike
- 3 **naučiti** teorijsku podlogu radi efikasnog i optimalnog korištenja `SPIN` programa

## Za one koji hoće više ...

### Komunikacijski protokol → aplikacija

Poželjno je svaku aplikaciju koju oblikujete—razvijate verificirati (*model-checking*).

Trenutačni trendovi u razvoju programa uključuju *skrivenne formalne metode* u kojima "model-checking" postaje dio modela razvoja programa ...



## A sada formalno: teorijska podloga

**Teorijska podloga** sadržana je u slijedećoj formuli

$$M \models \varphi$$

sa slijedećim značenjem

(da li Vam je nešto slično poznato od ranije ?)

$M$  je model odnosno Promela program

$\varphi$  su LTL formule (**LTL** - **L**inear **T**emporal **L**ogic) kojima provjeravamo uvjete

## Sve zajedno ...

- Promela program (model) se sastoji od mreže komunicirajućih automata. Sintaksa je slična jeziku C, Dijkstrine "guarded" komande i CSP algebra čine teorijske temelje (detalji kasnije)
- Spin pronalazi sve moguće interakcije (Kripké struktura) kao sinkroni ili asinhroni produkt automata, efikasno ih kodira (*bit-state-hashing*)
- sastavni dio Spin-a je i konverzija *LTL* logičke formule u Büchi automat

## U nastavku:

- 1 SPIN/Promela sustav sa uvodnim primjerom "Hello world" (tzv. snake preview ...)
- 2 konačni automat (FSM), sinkroni ili asinkroni produkt komunicirajućih automata (CFSM), *LTL* logika, pretvorba *LTL* u Büchi automat, Dijkstra "guarded" komande ili što sve *SPIN* uključuje ...

## SPIN instalacija

- 1 Instalacija se svodi na kopiranje već pripremljenih izvršnih verzija sa <http://spinroot.com/spin/Man/README.html>.
- 2 Potreban je i C prevodioc (preporuka: *gcc*)
- 3 Za one koje hoće više: pogledati *SPIN newsletter* i *SPIN* simpozije

## Primjer: "Hello world" ili kao opisujemo CFSM

```
/* A "Hello World" Promela model for SPIN. */
active proctype Hello() {
printf("Hello process, my pid is: %d\n", _pid);
}

init {
int lastpid;
printf("init process, my pid is: %d\n", _pid);
lastpid = run Hello();
printf("last pid was: %d\n", lastpid);
}
```

## Napomena:

*Promela* ima uvijek barem jedan `init{}` proces  
svaki *Promela* proces je jedan automat (FSM) iz CFSM (mreže komunicirajućih automata)

`_pid` "broj" procesa

predefinirane ili "unutarnje" varijable počinju s "\_"

`run` pokreće druge procese

*Promela* proces  $\neq$  ranije spomenutim procesima

slično sintaksi jezika CC ali oprez, **semantika** je drukčija

## SPIN: prema Kripke strukturi

```
spin -n2 hello.prm
init process, my pid is: 1
  last pid was: 2
Hello process, my pid is: 0
Hello process, my pid is: 2
3 processes created
running SPIN in
random simulation mode
random seed
```

### Za vježbu:

Pokušajmo zajedno skicirati automate FSM za "Hello world" !  
Što određuje broj `Promela` procesa ?

Hello world je izveden u tzv. simulacijskom modu  
slijede preporučeni koraci primjene *Promela/SPIN...*  
ili

Kako iz **CFSM** dobiti Kripke strukturu ?

Kako provesti analizu primjenom *LTL* logike nad *Kripke* strukturom ?

Za one koji hoće više:

Kripke struktura i *dostupnost*

Kolika je *kompleksnost* ?

## Roadmap ili postupak verifikacije

- (1) definirati prototip ili model za verifikaciju  
(*Promela* program  $\equiv$  CFSM)
- (2) prekontrolirati sintaksu modela: `spin -A`, `spin -c` ili `spin -p`  
prevesti model/*Promela* program: `spin -a hello.prm`
- (3) početi sa serijom *random* simulacija: `spin hello.prm` ili npr.  
`spin -p -u10 hello.prm`
- (4) kreirati **verifikator**: `spin -a hello.prm`  
načiniti izvršnu verziju gcc ili cc `-o hello pan.c`  
izvesti *hello* tj. **verificirati**
- (5) po "tragu" (*eng.trail*) do grešaka: `spin -t -p hello.prm`
- (6) redefinirati (ako treba) model tj. `hello.prm` i ponoviti sve korake  
do željene kvalitete



## Teorijska podloga Promela modela-jezika

Teorijska podloga Promela modela-jezika su:

- (i) Dijkstrine "*guarded*" komande
- (ii) CSP Hoare algebra (**C**ommunicating **S**equential **P**rocesses) komunikacijska algebra kao posebna vrsta procesnih algebri

## Dijkstrine "guarded" komande

"guarded" komande su oblika  $G \rightarrow S$ , gdje je:

- $G$  je **propozicija**, koju nazivamo **guard**
- $S$  je izvršna naredba naredba koju "guard" može blokirati.

### Semantika:

Semantika *Promele* i originalna *Dijkstrina* semantika nisu potpuno iste. (Vidjeti primjer za *Promelu*)

- u trenutku kada  $G$  postane istinit ...
- ... izvodi se naredba  $S$ ,  
ako  $G$  nije istinit nastupa "blokada"
- nije greška u *Promela* jeziku kada "guard" privremeno blokira!  
(npr. kada modeliramo čekanje prijema signala ili poruke!)
- kod Dijkstre kod neistinite propozicije  $G$  *kontekst* odlučuje o daljnjoj akciji (nije od značaja za nas)

## Dijkstrine "*guarded*" komande–nastavak

Primjer (u Promela sintaksi):

.....

$(A == \text{msgOK}) \longrightarrow G$

naredbe iza  $\longrightarrow S$

.....

naredbe  $S$  iza "*guarded*" komande  $G$  mogu biti izvedene samo ako varijabla  $A$  poprimi vrijednost  $\text{msgOK}$ .

Tko hoće više:

Dijkstra, Edsger W. "EWD472: Guarded commands, non-determinacy and formal. derivation of programs." (PDF)

<http://www.cs.utexas.edu/users/EWD/ewd04xx/EWD472.PDF>

# FSM–konačni automat

## Definicija FSM

Konačni automat  $A_i$  je 5 – *torka*  $(S, s_0, L, T, F)$ , gdje je:

- **S** konačni skup *stanja* ( $S$ ) (eng. *states*),
- **$s_0$**  *inicijalno* (početno) stanje,  $s_0 \in S$ ,
- **L** konačni skup *labela*,
- **T** skup *prijelaza*,  $T \subseteq (S \times L \times S)$ ,
- **F** skup konačnih (*finalnih-završnih*) stanja,  $F \subseteq S$ .

## Za vježbu:

Nacrtajte FSM (automat) prema primjeru na ploči i označite svaku od sastavnica u  $(S, s_0, L, T, F)$ .

Koja je veza prema UML ili SDL (MSC) i *Promela* procesu?

Kako ga možete programski realizirati ?

## *LTL* logika i Büchi automat

- logičku formula kojom *verificiramo* (provjeravamo) zadana svojstva *SPIN* pretvara u posebnu vrstu automata – Büchi automat
- Büchi automat je posebna vrsta FSM koja prihvaća beskonačne sekvence labela *L*. Büchi automat interpretiramo nad Kripke strukturom
- kažemo: Büchi automat ima konačan broj stanja i svojstvo  $\omega$  prihvaćanja:

$\alpha_0, \alpha_1, \dots, \alpha_i, \alpha_{i+1}, \dots, \alpha_n$  gdje  $n \longrightarrow \infty$  i  $\alpha_i \in L$

- u praktičnoj realizaciji Büchi automat je pridodan *CFSM* mreži automata
- oprez s *LTL* formulama: preporuča se upotreba do maksimalno tri temporalna *LTL* operatora zbog memorijskih ograničenja
- *SPIN* pridodaje u *Promela* model dodatne instrukcije za *LTL* formulu (never claim no o tome više kasnije ...)

# LTL logika

**LTL** – Linearna Temporalna Logika

## Sintaksa:

**LTL** sadrži propozicijske varijable  $p_1, p_2, \dots$ , uobičajene logičke konektore  $\neg, \vee, \wedge, \rightarrow$  i slijedeće temporalne modalne operatore:

- \* (next) (X, N,  $\bigcirc$ )
- \* (always, globally) (G,  $\Box$ ) npr.  $\Box p$
- \* (eventually, finally) (F,  $\Diamond$ ) npr.  $\Diamond p$
- \* (until) (U,  $\mathcal{U}$ ) npr.  $p \mathcal{U} q$

## LTL logika – nastavak

Semantiku donosimo za operatore koji se koriste u *SPIN*-u.

### Semantika

- \*  $\Box\phi$ :  $\phi$  je istinit na *cijelom* putu (u Kripke strukturi)
- \*  $\Diamond\phi$ :  $\phi$  je *na kraju*, u *konačnici* istinit (istinit je negdje na putu)
- \*  $\psi\mathcal{U}\phi$ :  $\phi$  je istinit u trenutnoj i budućim pozicijama, a  $\psi$  mora biti istinit do te pozicije

### Za vježbu:

Skicirati dijagram za svaki od operatora !

Neka je zadan *FSM* automat  $A=(S, s_0, L, T, F)$ .

Uvode se tri posebna tipa labela *L*:

- 1  $A.A$  je skup stanja označenih kao **stanja prihvatanja**  
(eng. *accept-state labels*)
- 2  $A.E$  je skup stanja označenih kao **konačna stanja**  
(eng. *end-state labels*)
- 3  $A.P$  je skup stanja označenih kao **stanja napredovanja**  
(eng. *progress-state labels*)

### Zašto posebne labele $A.A$ , $A.P$ i $A.E$

Posebne labele su dio *Promele*.

Na osnovu njihovog položaja *SPIN* može utvrditi neizvedene prijelaze, nemogućnost završetka ...

... mogu se usporediti sa *BREAK-points* i *WATCH-points* tijekom "debugiranja" programa



## Asinkroni produkt

Asinkroni produkt konačnog skupa automata  $A_1, A_2, \dots, A_n$  je također novi  $FSM$   $(S, s_0, L, T, F)$ :

- $A.S$  je kartezijev produkt  $A_1.S \times \dots \times A_n.S$ ,
- $A.s_0$  je  $n$ -torka  $(A_1.s_0, \dots, A_n.s_0)$ ,
- $A.L$  je unija skupova  $A_1.L \cup \dots \cup A_n.L$ ,
- $A.T$  je skup  $n$ -torki,  $((x_1, \dots, x_n), |, (y_1, \dots, y_n))$  takvi da  
 $\exists i, 1 \leq i \leq n, (x_i, |, y_i) \in A_i.T$  i  $\forall j, 1 \leq j \leq n, i \neq j \rightarrow (x_i \equiv y_j)$
- $A.F$  je podskup  $A.S$  takav da  $\forall (A_1.s, \dots, A_n.s) \in A.F$ ,  
 $\exists i, A_i.s \in A_i.F$

## Za vježbu:

- 1 Precrtati FSM sa ploče i odrediti asinkroni produkt.
- 2 U kakvoj su vezi asinkroni produkt i graf dostupnosti ?
- 3 U kakvoj su vezi graf dostupnosti i Kripke struktura ?
- 4 U kakvoj su vezi asinkroni produkt Kripke struktura ?

## Asinkroni produkt i konkurentnost

- \* Asinkroni produkt opisuje ponašanje sustava opisanog kao *CFSM*
- \* **Važno:** dozvoljen je samo **jedan prijelaz** po automatu
- \* na taj način *asinkroni produkt* opisuje "*interleaving*" semantiku konkurentnih procesa

## Sinkroni produkt i *LTL* formule

Neka je sustav *Sys* opisan *Promela* modelom koji se sastoji od Büchi automata *B* i asinkronog produkta automata *A<sub>i</sub>* iz *CFSM*:

$$Sys = B \oplus \prod_{i=1}^n A_i$$

- Operator  $\oplus$  predstavlja **sinkroni** produkt.

## Sinkroni produkt

Sinkroni produkt je automat  $A=(S, s_0, L, T, F)$ :

- $A.S$  je kartezijev produkt  $P'.S \times B.S$ , gdje  $P'$  ima pridodane *nil* (prazne) prijelaze u svakom stanju u  $P$  koje nema napretka (progres)
- $A.s_0$  je  $(P.s_0, B.s_0)$ ,
- $A.L$  je  $P'.L \times B.L$ ,
- $A.T$  je skup parova  $(t_1, t_2)$  takvi da  $t_1 \in P'.T$  i  $t_2 \in B.T$ ,
- $A.F$  je skup parova  $(s_1, s_2) \in A.S$  gdje  $s_1 \in P.F \vee s_2 \in B.F$

## Procesne algebre: CSP i SPIN

Spomenimo najvažnije poveznice između *CSP* algebri (**C**ommunicating **S**equential **P**rocesses) i *SPIN*/Promele

- $\mathcal{A} = \alpha_1, \alpha_2, \dots, \alpha_n \equiv A.L$   
(alfabet  $\mathcal{A}$  je predstavljen labelama u *FSM*)
- operatori *CSP* algebre (deterministički i nedeterministički izbor, kompozicija *CSP* procesa)
  - ⇒ realizirani kroz sinkronu i asinkronu kompoziciju automata

## ...i na kraju teme

- kvaliteta *SPIN* programskog alata spada u klasu "*industrial strength*" odnosno koristi se kao stabilan, pouzdan i upotrebljiv "model-checker" sustav pogodan za industriju i istraživanje
- pravilna upotreba ovisi o poznavanju svih mogućnosti i opcija
- mnogi alati na ključnim mjestima koriste *SPIN*: (*JavaPathFinder*, *Bandera* ...)
- ... slijedi praktična primjena

# Formalne Metode u oblikovanju sustava

FER

drugi ciklus predavanja, drugo predavanje  
ver. 0.1.8  
nadm.zadnje.rev.: 24. travnja 2009.



# Ponavljjanje

- 1 Teorijska podloga: automati (*CFSM*, *FSM* i logika (*LTL*))
- 2 Modeliranje: procesi i *Promela* jezik
- 3 Protokoli i procesi u raspodjeljenim, konkurentnim i reaktivnim sustavima

## ... slijede detalji o *Promela* jeziku

*Promela* model ili program se sastoji od:

- 1 deklaracije *tipova* podataka (eng. *type declaration*)
- 2 deklaracije *globalnih* varijabli (eng. *global variable declaration*)
- 3 deklaracije *komunikacijskih kanala* (eng. *channel declaration*)
- 4 deklaracije *procesa* (eng. *process declaration*)
- 5 deklaracije *početnog*, zajedničkog procesa (eng. *init process declaration*)

### *Promela* i *FSM*

- 1 Poželjno je uvijek uočiti istoznačnost *FSM* i *Promela* procesa
- 2 Često se koristi kao sinonim *Promela* model ili *Promela* program

## A što je u jeziku *Promela* izvršno ... ?

- ... upravo smo naveli samo **deklaracije** u jeziku *Promela*
- koje nalazimo i u npr. jeziku C (npr. `short int i=1, float a;` i sl.)
- gdje su izvršne naredbe npr. `i++; a=a*1+2; func10(a,i);`
- u jeziku *Promela* osnovna izvršna jedinica je **proces**

- 1 Programski alat *Spin* opisuje ponašanje sustava kao skupa potencijalno interaktivnih, asinkronih, komunicirajućih dretvi, niti, tredova (eng. *threads*)
- 2 deklaracija procesa (`proctype` konstrukt) opisuje ponašanje ali **izvršivost** (eng. *executability*) možemo postići jedino eksplicitnim pozivom procesa (preko `active proctype` ili `sa run` konstruktom)

## Za vježbu:

### *Hello primjer i procesi*

Modificirajte *Hello* primjer sa i bez *init* naredbe.

- 1) Što znači `active proctype` ?
- 2) Koliko ima ukupno procesa sa i bez *init* naredbe ?
- 3) Što znači `active proctype` bez *init* naredbe?
- 4) Pokrenite `./hello` i `./hello -d`. Što uočavate ?

## Značenje ";"

- u jeziku C: ";" **završava** naredbu
- (eng. *statement terminator*)
- u jeziku *Promela* ";" **razdvaja, odvaja** naredbu  
(eng. *statement separator*)

## Važno:

Umjesto ";" možemo koristiti i "->" kao separator naredbi u jeziku *Promela*

# Specifičnosti jezika Promela

## Blokirajuće ili izvršne naredbe

Sve promela naredbe su *izvršne* ili *blokirajuće*

Blokirajuće naredbe su implementacija Dijkstrinih *guarded* komandi: one blokiraju samo do trenutka kada je uvjet **G** zadovoljen, a *nakon* toga se izvode slijedeće naredbe

## Primjer:

```
(turn == P) -> printf("Produce") ...
```

tek onda i samo onda kada je varijabla `turn` jednaka `P` ispisuje se "Produce"

sve dok ta jednakost ne vrijedi ili *guard* – propozicija ne postane istinita `proces` je (privremeno) blokiran

## Tipovi podataka

- ⇒ varijable u jeziku *Promela* su lokalne ili globalne
- ⇒ Osnovni tipovi podataka:

bit	$0 \dots 1$	bit OK=1;
bool	false...true	bool flag = false;
byte	$0 \dots 255$	byte foo;
chan	$1 \dots 255$	chan AtoB;
mtype	$1 \dots 255$	mtype msg;
pid	$0 \dots 255$	pid p;
short	$-2^{15} \dots s^{15} - 1$	short a = 137;
int	$-2^{31} \dots s^{31} - 1$	int i = 13;
unsigned	$0 \dots 2^n - 1$	unsigned u:3;

# Tipovi podataka

## Nema ...

`real, float, pointer`

kao tipovi podataka ne postoje u jeziku *Promela*

modelira se **koordinacija** među procesima a ne izvode se numerički proračuni

## Napomene o podacima:

- 1 inicijalne vrijednosti svih varijabli (lokalnih i globalnih) su jednake 0
- 2 sve varijable moraju biti deklarirane prije upotrebe
- 3 deklaracija se može nalaziti bilo gdje u programu



# Tipovi podataka

## Polja

- U jeziku *Promela* moguće je definirati jednodimenzionalna polja
- Vrijednosti indeksa polja kreću od nule kao i kod jezika C

## Primjer polja:

```
bit a[11];  
byte tr224[99];  
...
```

## Tipovi podataka

U jeziku *Promela* korisnik može definirati vlastite tipove podataka (sintaksa slijedi jezik C):

*typedef* primjer:

```
typedef adtStruct {  
    short foo29;  
    byte vxcount = 12;  
}
```

```
adtStruct serverstatus;  
serverstatus.vxcount = 159;
```

## Napomena:

Definiranje *vlastitih podatkovnih struktura* u jeziku *Promela* znatno proširuje dosege upotrebe programskog alata *Spin* (u velikoj mjeri podržani su apstraktni tipovi podataka)

Osim toga u jeziku *Promela* moguće je ubaciti i dijelove pisane u jeziku *C* (eng. *C embedded code*)

## O komunikaciji. . .

### . . . preko kanala

*Promela* procesi komuniciraju preko kanala (chan)

Kanale je potrebno deklarirati

Kanali su **globalnog** karaktera

Problem: odrediti kapacitet kanala u općem slučaju

### . . . i preko globalnih varijabli

*Promela* procesi komuniciraju i preko globalnih varijabli

Problem: prava dostupnosti globalnim varijablama ("mutual exclusion")

## Kanali s porukama

### Primjer:

Precrtajte u bilježnicu primjer UML sekvencnog (ili MSC dijagrama)

*Promela* procesi komuniciraju slanjem/prijemom poruka kroz kanale:

```
chan AtoB = [10] of {int, short, bit}
```

- **chan** deklaracija komunikacijskog kanala
- **AtoB** ime kanala
- **[10]** kapacitet kanala: maksimalni broj poruka kapacitet **[0]** znači sinkronu izmjenu poruka
- **{int, short, bit}** struktura poruke koja se šalje kroz kanal

## Kanali s porukama

### Primjer:

Precrtajte u bilježnicu primjer UML sekvencnog (ili MSC dijagrama)

*Promela* procesi komuniciraju slanjem/prijemom poruka kroz kanale:

```
chan AtoB = [10] of {int, short, bit}
```

- **chan** deklaracija komunikacijskog kanala
- **AtoB** ime kanala
- **[10]** kapacitet kanala: maksimalni broj poruka kapacitet **[0]** znači sinkronu izmjenu poruka
- **{int, short, bit}** struktura poruke koja se šalje kroz kanal

## Kanali s porukama

### Primjer:

Precrtajte u bilježnicu primjer UML sekvencnog (ili MSC dijagrama)

*Promela* procesi komuniciraju slanjem/prijemom poruka kroz kanale:

```
chan AtoB = [10] of {int, short, bit}
```

- **chan** deklaracija komunikacijskog kanala
- **AtoB** ime kanala
- **[10]** kapacitet kanala: maksimalni broj poruka kapacitet **[0]** znači sinkronu izmjenu poruka
- **{int, short, bit}** struktura poruke koja se šalje kroz kanal

## Kanali s porukama

### Primjer:

Precrtajte u bilježnicu primjer UML sekvencnog (ili MSC dijagrama)

*Promela* procesi komuniciraju slanjem/prijemom poruka kroz kanale:

```
chan AtoB = [10] of {int, short, bit}
```

- `chan` deklaracija komunikacijskog kanala
- `AtoB` ime kanala
- `[10]` kapacitet kanala: maksimalni broj poruka kapacitet `[0]` znači sinkronu izmjenu poruka
- `{int, short, bit}` struktura poruke koja se šalje kroz kanal



## mtype deklaracija

deklaracija tipa poruke omogućuje pojednostavljeno rukovanje porukama:

osim standardnih tipova `mtype` je ugrađeni tip koji se tipično koristi unutar kanala:

```
chan toServer = [2] of { mtype, data, adress0 }
```

### Primjer:

- **mtype** = ack, req, setFGL;
- **mtype** m; – neinicijalizirana poruka ima vrijednost 0
- **mtype** mblockA = wsdp; – inicijalizirana poruka, ima vrijednost različitu od 0

Dozvoljeno je do 255 različitih poruka.

# Prijem i predaja poruka

## Sintaksa

Sintaksa simbola predaje i prijema je preuzeta iz *CSP* algebre:

- simbol "!" se koristi za predaju
- simbol "?" se koristi za prijem
- prijem/predaja znače stavljanje/uzimanje poruke u kanal koji opisuje komunikaciju između dva procesa
- naredba sa prijemom/predajom je izvršna ako kanal nije prazan/pun
- ponašanje kanala slično je ponašanju repa (eng. *queue*)

# Prijem i predaja

## Prijem

ch?*const*<sub>1</sub> ili *var*<sub>1</sub> ...*const*<sub>n</sub> ili *var*<sub>n</sub>

*const*<sub>i</sub> i *var*<sub>i</sub> moraju odgovarati poljima u poruci

## Predaja

ch! *expr*<sub>1</sub> ... *expr*<sub>n</sub>

*expr*<sub>i</sub> mora po tipu odgovarati poljima u poruci

## Primjeri

```
mtype = req;    chan chn = [N] of mtype, bit;  
bit nmsg;  
chn?req,nmsg;  
chn!ack,1;
```

### Za vježbu:

Da li je moguće komunikaciju riješiti  
bez korištenja `chan` te prijema/ predaje ?  
Obrazložite mogućnosti !

## Prijelazi u *FSM* i *Promela* naredbe

Svaki *Promela* proces predstavlja *FSM*.

Prijelaze (*T* prema *FSM* definiciji) možemo definirati:

- uvijek izvršne  
npr.: (`printf`, `assert`, "assertions" kao `x++`, `y=x-3`)
- izvršne kada su istiniti uvjeti ("*guard*")  
npr.: (`x == 2`), (`N < 4`)
- izvršne kada kanal nije pun (predaja `send`)
- izvršne kada kanal nije prazan (prijem `receive`)

## Sinkronost vs. asinkronost

- ⇒ U svojoj suštini **svi** procesi su *asinkroni*
- ⇒ *Sinkronost* uvodimo zbog potrebe modeliranja: često je potrebno analizirati samo bitno, zato se *asinkrone* pojave apstrahiraju
- ⇒ kod analize zahtjeva (eng. requirements analysis) često ne promatramo asinkrone popratne pojave
- ⇒ *Spin* preko `chan Chan = [0] of msg1, msg2 ...` podržava *sinkroni* način rada

# Struktura procesa

Strukturu *Promela* procesa definiramo preko strukture *FSM* sa slijedecim konstruktima:

- `;` , `goto` i *labele*
- nedeteriministička selekcija (*Promela* `if`)
- nedeteriministička iteracija (*Promela* `do` petlja)
- *promela* `"unless": { } unless { }`
- atomske (nedjeljive) sekvence (`atomic { } id_step { }`)

## nedeterministička selekcija (*if*)

*if*

*guard*<sub>1</sub>  $\longrightarrow$  *stmtnt*<sub>1,1</sub>; *stmtnt*<sub>1,2</sub>; *stmtnt*<sub>1,3</sub>;

*guard*<sub>1</sub>  $\longrightarrow$  *stmtnt*<sub>2,1</sub>; *stmtnt*<sub>2,2</sub>; *stmtnt*<sub>2,3</sub>;

...

*guard*<sub>1</sub>  $\longrightarrow$  *stmtnt*<sub>*n*,1</sub>; *stmtnt*<sub>*n*,2</sub>; *stmtnt*<sub>*n*,3</sub>;

*fi*

- ako je barem jedan "guard" izvršan, *if* je izvršan
- ako je više od jedan "guard" izvršan, izvodi se "guard" po slučajnom odabiru
- ako niti jedan "guard" nije izvršan, *if* *blokira*

### Za vježbu:

Precrtajte dio pripadnog automata (*FSM*) za *Promela if* naredbu !



## nedeterministička iteracija (do)

do

$guard_1 \longrightarrow stmt_{1,1}; stmt_{1,2}; stmt_{1,3};$

$guard_1 \longrightarrow stmt_{2,1}; stmt_{2,2}; stmt_{2,3};$

...

$guard_1 \longrightarrow stmt_{n,1}; stmt_{n,2}; stmt_{n,3};$

od

→ do u *Promeli* je *if* u beskonačnoj petlji ...

→ ... iz koje se izlazi sa *break* ili *goto* naredbom

### Za vježbu:

Precrtajte dio pripadnog automata (FSM) za *Promela* *do* naredbu !

Koja je semantika *Pomela* *goto* naredbe (iz većine jezika izbačene) ?

## Atomske (nedjeljive sekvence)

```
atomic { }
```

`atomic { }` sekvencu ili blok naredbi *Spin* u simulaciji/verifikaciji promatra kao da su nedjeljive

```
d_step { }
```

`d_step { }` je rigorozniji oblik `atomic { }` direktive.

unutar `d_step { }` nisu dozvoljeni `goto`, nedeterminizam i naredbe koje mogu "blokirati".

### Za vježbu:

Koja je glavna namjena `atomic { }` i `d_step { }` direktiva ?  
Kako utječe na memorijske i vemenske resuse *Spin* alata ?

## Primjer: asinkroni produkt

```
#define N 4
#define p (x < N)

int x = N;

active proctype A1()
{
do
:: x%2 -> x = 3*x+1
od
}
```

## Primjer: asinkroni produkt

```
active proctype A2()  
{  
do  
:: !(x%2) -> x = x/2  
od  
}
```

*LT*L ili never-claim u *Promeli*:

```
never {      /* <>[]p */
T0_init:
    if
    :: p -> goto accept_S4
    :: true -> goto T0_init
    fi;
accept_S4:
    if
    :: p -> goto accept_S4
    fi;
}
```

## Analiza $A_1$ i $A_2$

Za vježbu:

- a) što je predikat  $p$  u *LTL* formuli
- b) da li je potreban `init { }` dio programa
- c) generirati `never { }` "claim" sa spin -f <>[]p
- d) nacrtati Büchi automat  $B$
- e) nacrtati  $A_1$  i  $A_2$  direktno iz koda u *Promeli*
- f) nacrtati  $A_1$  i  $A_2$  preko `pan -d` naredbe
- g) usporediti i provjeriti dobivene grafove automata  $A_1$  i  $A_2$
- h) provesti analizu: preko simulacije i verifikacije

## Za one koji hoće više:

### Službeni SPIN tutori

- najbolji način za pretvaranje *Promele*/*SPIN* u još moćan i pouzdan programski alat za svakodnevnu upotrebu je samostalno modeliranje
- poželjno je proučiti što više (rješениh) primjera
- pri tome se može pomoći *tutorima* i člancima dostupnim na [www](http://www.fer.hr) stranicama

## Zaključak:

- (1) modeli sa **konačnim** brojem stanja (ali sa  $\omega$ -prihvatljivosti)
- (2) **asinkronost**: nema unaprijed definiranog mehanizma za sinkronizaciju kao ni sistemskog "sata" (clock)
- (3) **nedeterministička** upravljačka struktura: prijelazi u *FSM* su nedeterministički
- (4) **izvršivost** preko blokirajućih naredbi ("*guards*")
- (5) mogućnost **dodavanja** koda u jeziku *C* i **vlastite strukture** podataka
- (6) **proširenje osnovne namjene**: osim analize konkurentnih reaktivnih programa *Spin* se primjenjuje i u testiranju, planiranju, ... kao sastavni dio raznih programskih alata ...



# Formalne Metode u oblikovanju sustava

FER

drugi ciklus predavanja, treće predavanje  
ver. 0.1.8  
nadm.zadnje.rev.: 2. svibnja 2009.

# Ponavljjanje

- 1 *Promela* jezik
- 2 LTL formule
- 3 Instalacija i primjer (*Hello*)

## Napomena:

- zadanje predavanje u ovom ciklusu ima naglasak na praktičnoj upotrebi
- primjere pokušati samostalno rješavati sa računalom ...

# Spin i XSpin

## Spin

- *Spin* nema grafičko sučelje
- poziva se iz komandne linije:
- `spin --help` pokazuje sve dostupne opcije

## XSpin

- *Xspin* je grafičko sučelje za *Spin*
- *Xspin* je preprocesor i vizualizator za *spin*
- *Xspin* je napisan u TclTk skriptnom jeziku

## Korištenje alata *Spin*

- 1 programskom alatu *Spin* pristupamo preko jezika *Promela*.
- 2 *Spin* koristimo samostalno za analizu . . .
- 3 *Spin* koristimo kao dio nekog programskog alata
- 4 osnovna namjena: *Spin* služi za verifikaciju konkurentnih reaktivnih procesa
- 5 osim verifikacije *Spin* nalazi primjenu i u mnogim ostalima područjima razvoja programske potpore

## Kako uključiti Spin u ciklus razvoja programske potpore ?

- 1 kao analizator–verifikator: potrebno je razviti i analizirati modele
- 2 kao alat koje je već dio programskog alata ili eng. *model extractor* (npr. *JavaPathfinder* ...)
- 3 aplikacija / problem od interesa koristi algoritme unutar *Spin* alata. Tada **proširujemo** vlastiti programski alat ili aplikaciju (eng. *embedding*).
- 4 apstraktne strukture podataka su podržane preko *Promela* `typedef` konstrukta.

## Ograničenja *Spin Promela* alata:

- 1 konačni broj procesa
- 2 konačni broj stanja po procesu
- 3 varijable moraju biti ograničene
- 4 eksplozija stanja: (algoritmi za izračun grafova dostupnosti nisu polinomno kompletni!)
- 5 nemogućnost dinamičke deklaracije novih procesa

Spomenuta ograničenja odnose se na svaku analizu/verifikaciju provjerom modela  
(na svaki *model checking* program). . .

→ Programski alat *Spin* je industrijski relevantan proizvod (eng. *industrial strength tool*).

→ Do sada postoji mnogo značajnih primjena  
*Kako uspješno koristiti Spin programski alat ?*

- 1 **konačni** broj stanja, procesa i ograničenost varijabli: *modeliranje* je, načelno govoreći proces apstrakcije koji nije egzaktn. Pažljivim izborom apstrahiranja–modeliranja može se rješavati veliki broj problema
- 2 **explozija stanja**: *Spin* stanja kodira preko posebne tehnike (*bit–state–hashing*, koristi naprednu metodologiju za smanjenje broja stanja zbog pravilnosti u modelima, a i korisnik preko opcija može utjecati na kompleksnost analize)

Za one koji hoće više:

Na *Spin* stranici ([spinroot.comspin/whatispin.html](http://spinroot.comspin/whatispin.html)) proučite primjere industrijske primjene

Da li možete vlastite zadatke ili projekte opisati i analizirati

*Spin/Promela* modelima ?



## Spin u analizi modela

- Neka je problem za verifikaciju ili analizu opisan i iskazan u jeziku *Promela* ...
- problem se nalazi u datoteci `model.prm`...
- nakon uobičajenog uvodnog `hello.prm` modela ...
- slijede dodatne opcije koje se koriste prilikom verifikacije

### Primjer:

precrtajte sa ploče `model.prm` !  
na računalu sami pokrenite zadane opcije

## O radnom primjeru koji se rješava

Zadani primjer je poopćeni model konkurentnih procesa koje nalazimo u mnogim praktičnim primjenama:

- raspodjeljeni, konkurentni sustavi: komunikacija među procesima
- klijent server aplikacije
- *"mutex"* protokoli
- komunikacijski protokoli
- raspodjeljene *web* aplikacije
- komponente i oblikovni obrasci (eng. *"design patterns"*)

⇒ Pažnju usmjerite na **analizu** `modela.prm`

### Napomena:

→ **Modeliranje i apstrakcija** realnih problema je poseban problem i nije dobro istovremeno učiti modeliranje i korištenje

→ **Modeliranje i apstrakcija**  $\equiv$  definiranje *Promela modela* (tj. `modela.prm`)

## Spin opcije

```
spin -A model.prm
```

- To je prva opcija nakon što je definiran model.
- Provjerava se sintaksa i eventualni nekonzistentni konstrukti

## Spin opcije

```
spin -p model.prm
```

- pokreće simulaciju modela
- simulacija se vrši po slučajnom izboru
- opcija `-nN` sa npr. `N = 12` inicijalizira generator slučajnih brojeva
- ⇒ simulacijom se dobiva osnovni uvid o ponašanju modela

## Spin opcije

```
spin -p -c -u200 -j10 model.prm
```

- simulacija se zaustavlja nakon 200 koraka
- prvih 10 koraka u simulaciji se preskače

## Spin opcije

```
spin -l -g -r -s -c model.prm
```

- opcije pokazuju lokalne i globalne varijable kao i prijem odnosno predaju poruka
- simulacija pokazuje konzistentnost: ponašanje modela prema očekivanjima
- simulacija ukazuje na greške zbog nepoznavanja *Promela* semantike i sintakse

## Spin opcije

```
spin -a model.prm
```

```
spin -a -f 'ltl-formula' model.prm ili
```

```
spin -a -F LTL-file model.prm
```

- opcija `spin -a` generira u jeziku C analizator (`pan.[bchmt]`)
- `spin -a -f` i `spin -a -F` pridodaju **LTL** formule analizatoru
- *Spin* transformira **LTL** formule u Büchi automat
- Büchi automat je u *Promela* modelu kodiran sa `never { }` konstruktom
- moguće je i kreirati vlastite Büchi automate editiranjem `never { }` konstrukta



## Spin opcije

```
spin -a model.prm  
gcc -o pan pan.c  
./pan ili pan.exe
```

- generiranje analizatora kao `pan` ili `pan.exe` izvršnog programa
- spomenimo i dodatnu opciju `spin -m` kao intervenciju u *Promela* semantiku: predaja je uvijek izvršna, (nije blokirajuća) iako je rep pun

## Izlaz *pan* analizatora: rezultati

```
(Spin Version 5.1.7 -- 23 December 2008)
```

```
+ Partial Order Reduction
```

```
Full statespace search for:
```

```
never claim          - (none specified)
```

```
assertion violations +
```

```
acceptance  cycles - (not selected)
```

```
invalid end states +
```

```
State-vector 20 byte, depth reached 49, errors: 0
```

```
    148 states, stored
```

```
    129 states, matched
```

```
    277 transitions (= stored+matched)
```

```
      0 atomic steps
```

```
hash conflicts:          0 (resolved)
```

```
    2.501 memory usage (Mbyte)
```

```
unreached in proctype mutex
```

```
line 26, state 23, "-end-"
```

```
(1 of 23 states)
```

```
pan: elapsed time 0 seconds
```

## Izlaz *pan* analizatora: rezultati

- Full statespace search for: pokazuje kao se provjerava model (npr. pogrešna završna stanja i pogrešne tvrdnje)
- State-vector: opisuje dubinu i veličinu grafa
- **errors: 0** – ukazuje na odsutnost grešaka. Pojava greške generira datoteku sa "tragom" (eng. *error trail*), tako da je moguće precizno utvrditi kojom sekvencom instrukcija dolazi do greške.
- **unreached**: pokazuje djelove modela koji su nedostupni

### Napomena:

Analizator pokaže koje su analize provedene

- Što zaista znače rezultati odnosno izlazne poruke analizatora ?

## Spin opcije

```
spin -t -p model.prm
```

→ ako analizator javi grešku opcija `-t` ispisuje sekvencu koja vodi prema greški

```
spin --help i pan --help
```

## Sve opcije

→ pokazuje sve opcije: preporuka je koristiti samo opcije koje za koje se detaljno razumije semantika

## Pan opcije

```
gcc -o pan pan.c
```

- generiranje analizatora (*pan*) svodi se na upotrebu programa prevodioca
- preporuča se upotreba *gcc* prevodioca

## Spin direktive prevodiocu

```
gcc -Dxxx -o pan pan.c
```

→ ponekad je potrebno prevoditi sa dodatnim opcijama:

- \* **-DNP** otkriva cikluse koje nemaju progressa (napretka)
- \* **-DBFS** generiranje stabla dostupnosti po širini (*breadth-first*) umjesto po dubini (*depth-first*)

## Spin direktive prevodiocu (primjeri)

```
gcc -DMEMLIM=512 -o pan pan.c
```

```
gcc -DHC4 -o pan pan.c
```

```
gcc -DBITSTATE -o pan pan.c
```

### Kako *Spin* maksimalno iskoristi postojeće ...

- problem svakog analizatora su memorijska ograničenja, opcije optimiziraju korištenje memorije
- potrebno je kroz niz iteracija pronaći optimalnu upotrebu s obzirom na model i dostupne resurse
- *Spin* kodira (preko  $10^{22}$ ) stanja primjenom BITSTATE-hash kodiranja



## Pan opcije

pan -w23

*Spin* pokušava sve izračunati unutar  $10^{18}$  stanja, ako nije dovoljno, opcija -w23 povećava na  $10^{23}$  stanja

## *Pan* opcije

```
pan -m100000
```

*Spin* može ograničiti i dubinu pretraživanja, u ovom slučaju na 100000.

## Pan opcije

$\text{pan} \neg a$  i  
 $\text{pan} \neg 1$

analiza za provjeru istinitosti *LTL* formula ( $\text{pan} \neg a$ )  
odnosno "petlji bez napretka" ( $\text{pan} \neg 1$ )

## Zaključak i nastavak ...

- (1) analizirajte gotove primjere modela koji slijede !
- (2) nakon toga pokušajte kreirati svoje *Promela* modele !
- (3) procijenite upotrebljivost programskog alata *Spin/Promela* na vašim primjerima !
- (4) planirajte daljnje učenje oko razvoja programske potpore ...
- (5) stalno pratite literaturu: verifikacija, modeliranje je područje koje se razvija i u kome slijedi još rezultata ...
- (6) pročitajte zadnje dvije folije: one ilustriraju sadašnju situaciju oko razvoja programske potpore ...
- (7) povežite gradivo ovog ciklusa sa ostalim ciklusima predavanja ili odgovorite na pitanje: **"How designer designs ?"**

## Za vježbu:

Slijede tri primjera protokol (*Bartlett*), "*Produce–Consumer*" i *Dekker* mutex protokol

- a) nacrtajte pripadne *FSM* !
- b) pokrenite simulacijski mod `spin -c -p ...`

## Za one koje žele više:

- generirajte verifikator (`pan`) !
- Kako bi pokazali odsutnost npr. "deadlocka" ?

# Bartlett protokol

```
#define MAX 4                      /* file ex.2 */
proctype A(chan in, out)
{
  byte mt; /* message data */
  bit  vr;

S1: mt = (mt+1)%MAX;
   out!mt,1;
   goto S2;

S2: in?v;
   if
   :: (vr == 1) -> goto S1
   :: (vr == 0) -> goto S3
   :: printf("MSC: AERROR1\n") -> goto S5
   fi;

S3: out!mt,1;
   goto S2;

S4: in?v;
   if
   :: goto S1
   :: printf("MSC: AERROR2\n"); goto S5
   fi;

S5: out!mt,0;
   goto S4
}
```

# Primjer - Bartlett protokol

```
proctype B(chan in, out)
{
  byte mr, lmr;
  bit ar;
  goto S2; /* initial state */
S1: assert(mr == (lmr+1)%MAX);
  lmr = mr;
  out!1;
  goto S2;
S2: in?mr,ar;
  if
  :: (ar == 1) -> goto S1
  :: (ar == 0) -> goto S3
  :: printf("MSC: ERROR1\n"); goto S5
  fi;
S3: out!1;
  goto S2;
S4: in?mr,ar;
  if
  :: goto S1
  :: printf("MSC: ERROR2\n"); goto S5
  fi;
S5: out!0;
  goto S4
}
}
```

## Primjer - Bartlett protokol

```
init {  
    chan a2b = [2] of { bit };  
    chan b2a = [2] of { byte, bit };  
    atomic {  
        run A(a2b, b2a);  
        run B(b2a, a2b)  
    }  
}
```



# Primjer - "Producer Consumer"

```
mtype = { P, C };

mtype turn = P;

active proctype producer()
{
do
:: (turn == P) ->
printf("Produce\n");
turn = C
od
}

active proctype consumer()
{
do
:: (turn == C) ->
printf("Consume\n");
turn = P
od
}
```

## Primjer - Dekker mutex

```
bit turn; bool flag[2]; byte cnt;
  active [2] proctype mutex() /* Dekker's 1965 algorithm */
  { pid i, j;
    i = _pid;
    j = 1 - _pid;
    again:
    flag[i] = true;

do
  :: flag[j] ->
  if
  :: turn == j ->
  flag[i] = false;
  !(turn == j);
  flag[i] = true
  :: else
  fi
  :: else ->
  break
  od;

  cnt++;
  assert(cnt == 1); /* critical section */
  cnt--;
  turn = j;
  flag[i] = false;
  goto again
}
```

## C.A.R. Hoare

CACM, 03/2009 Vol.52 No3.3 interview pp.41

*As far as the fundamental science is concerned, we still certainly do not know how to prove programs correct. We need a lot of steady progress in this area, which one can foresee, and a lot of breakthroughs where people suddenly find there is a simple way to do something that everybody hitherto has thought to be far too difficult.*

## Fortune Magazine: BrainstormTech.

IEEE Spectrum INT, September/2008 pp.05

**The Future of code** quote from user guide that comes with Your (. . . \*) computer

*This computer is not intended for use in the operation of nuclear facilities, aircraft navigation or communication systems, or air traffic control machines, or for any other uses where the failure of your computer system could lead to death, personal injury, or severe environmental damage.*

Pomaže li uvijek **Cntrl–Alt–Del** ili **Esc** ?