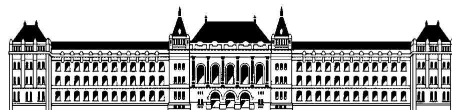
BUDAPESTI MÛSZAKI EGYETEM
Méréstechnika és Információs Rendszerek Tanszék



SPIN
Mérési ú tmutató

Készfette:
Já vorszky Judit

Tartalomjegyzék

1. Bevezetés 2
1.1. Általános leírás 2
1.2. Módszertan 3
1.3. Felépítés 3
2. PROMELA 4
2.1. Általános jellemzõk 4
2.2. Adattípusok 4
2.3. Processzek és modellek 5
2.4. Kijelentések 7
2.5. Példa 9
3. SPIN 10
3.1. Szimuláció 10
3.2. Verifikáció 10
3.2.1. Standard elvárások 10
3.2.2. Elvárások megfogalmazása 11

1. Bevezetés

1.1. Á ltalá nos leírá s

A SPIN elosztott rendszerek formális verifikációjára alkalmas szoftver rendszer, melyet a Bell Labs fejlesztett. A célja, hogy egy hatékony verifikációs lehetõséget adjon szoftver rendszerek (nem hardver) számára. A program egy magas szintû nyelvet használ a verifikálandó rendszer specifikálására, a PROMELA-t (PROcess MEta LAnguage).

A SPIN fõ alkalmazási területe elosztott rendszerek logikai tervezési hibáinak detektálása, mint operációs rendszerek, adat kommunikációs protokollok, konkurens algoritmusok, vasúti jelzõ rendszerek, és így tovább. Ezen rendszereknél a SPIN lehetõséget ad a rendszer specifikációjának logikai konzisztenciájának

A SPIN szoftver standard ANSI C nyelven íródott, és futtatható a különbözõ verziójú UNIX operációs rendszereken. Ugyanígy futtatható PC-n futó Linux, Windows95/98, vagy WindowsNT rendszereken is.

---

http://netlib.bell-labs.com/netlib/spin/whatisspin.html
http://cm.bell-labs.com/cm/cs/what/spin/Man/index.html

1.2. Módszertan

használhatjuk:
alkalmas arra, hogy a tervezett rendszer specifikáljuk, illetve, hogy ellenõrizzük a specifikációt. Részletesebben:

1. Specifikáció: a tervezés specifikációja, azaz egy olyan specifikáció melyben csak a lényeges tervezési választások szerepelnek, implementációs részletek nélkül. Ez a viselkedés modellje.
2. Specifikáció: az elvárások specifikációja, azaz azon kijelentések megfogalmazása, melyeket szeretnénk ha a tervezett rendszerünk teljesítene.
3. Verifikáció: a tervezés logikai konzisztenciájának (1) ellenõrzése, illetve hogy a tervezett rendszer specifikációja kielégíti-e az elvárásainkat, vagyis a specifikált elvárások teljesülnek-e (2).

Így a SPIN-nel lehetõségünk van:

1. a viselkedés modelljének specifikálására, és ebbõl kiindulva a
    - a rendszer viselkedésének szimulálására és a rendszer állapotainak megfigyelésére,
    - az állapot jellemzõk verifikálására, ciklusok kimutatására, elérhetetlen kódok megtalálására,

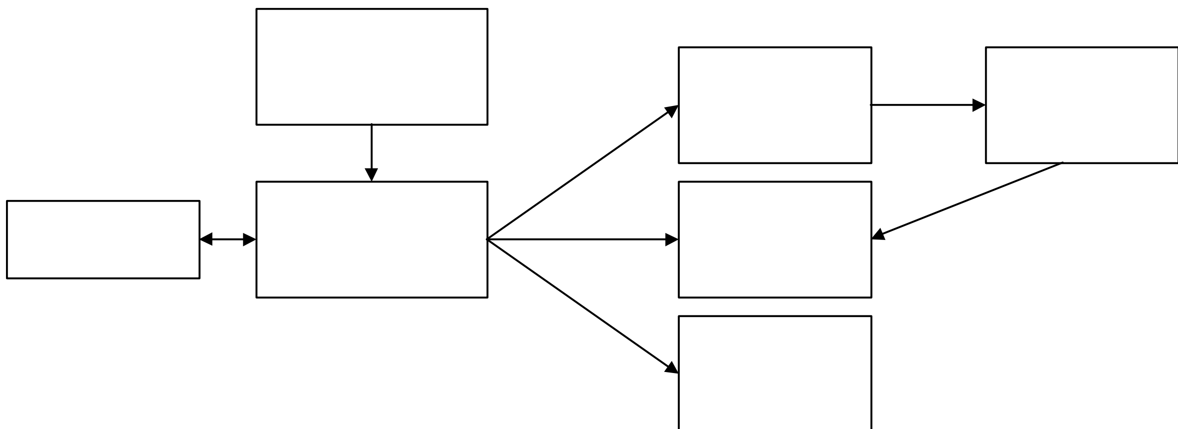
VAGY

2. a viselkedés modelljének illetve az elvárásoknak együttes specifikációjára, és annak ellenõrzésére, hogy a viselkedés modellje kielégíti-e az elvárásokat.

A SPIN minden esetben amikor valamilyen ellentmondást, problémát talál, egy példán keresztül bemutatja, azaz ha egy verifikációt indítunk SPIN-nel, és az leáll, akkor például a SPIN a probléma okáig

1.3. Felépfés

A SPIN általános felépítése:



Megjegyzés: Az LTL a Lineáris Temporális Logika rövidítése, ennek segítségével fogalmazhatunk meg a logika szabályainak megfelelõen elvárásokat, melyeket szeretnénk, hogy ha a rendszerünk teljesítene.

2. PROMELA

A PROMELA (PROcess MEta LAnguage) a SPIN bemeneti nyelve, egy verifikáció modellezõ nyelv. Lehetõséget ad arra, hogy megfogalmazzunk absztrakt módon egy protokollt (avagy általánosságban egy elosztott rendszert) csak a lényegre figyelve, vagyis a processz interakció szempontjából irreleváns részleteket kihagyva. Így egy adott processznek csak a lényegét modellezzük PROMELA-ban, és a SPIN-nel ezt

sek sorozatából tevõdik össze, így konstruálva az egyre részletesebb PROMELA modelleket. Mindegyik modell verifikálható a SPIN-nel különbözõ megszorításokat téve a környezetre, pl.: üzenet elvesztés, üzenet duplikáció, stb. Amint egy modell helyessége a SPIN által bizonyítva van, ez a tény felhasználható a késõbbi modellek megalkotásához, illetve verifikálásához.

A viselkedés modelljét fogalmazzuk meg egy PROMELA programmal, amely *processzekbõl*, üzenet *csatornákból* és *változókból* épül fel. A processzek globális objektumok, és ezek szolgálnak a modellezni kívánt rendszer viselkedésének specifikálására. A csatornák kommunikációra szolgálnak, és lehetnek globálisak, illetve processzen belül lokálisak is. Ugyanígy a változók is lehetnek globálisak, illetve processzen belül lokálisak. Míg a processzek specifikálják a viselkedést, a csatornák és globális változók definiálják azt a

A PROMELA nyelvre, mint programozási nyelvre jellemzõ, hogy egy C-szerû nyelv, kibõvítve a kommunikáció lehetõségével, és nemdeterminizmussal.

2.2. Adattípusok

- **Alap adattípusok** : hossza (bit)

bit	1
bool	1
byte	8
short	16 (elõjeles)
int	32 (elõjeles)
chan	

A felsorolásban az elsõ öt típus magától értetõdõ, esetleg megjegyezendõ, hogy a bit és bool típusok szinonimái egy biten tárolt információnak.

```
int x, y, z;
bit b;
```

A chan típussal tudunk csatornát definiálni.

```
chan c = [5] of {bit, int}
```

Ebben az utolsó esetben egy 5 hosszú pufferelt csatornát definiáltunk, melyben 1 üzenet egy bit és egy int érték lehet. Ha szinkron csatornát szeretnénk létrehozni, vagyis olyat, ahol nincs puffer, akkor a következõképpen tehetjük meg:

```
chan d = [0] of {bit, int, short}
```

Egy mtype változó használható arra az esetre, ha szimbolikus üzenet értékeket szeretnénk használni:

```
mtype = {control, data, error}
```

- **Strukturá lt adattípusok**

Tömbök:

```
int x[10];
chan c[3] = [6] of {bit, int, chan};
```

A tömb elemeire x[0] … x[9] -ként, illetve c[0] … c[2] -ként hivatkozhatunk.

```
typedef MSG { bit control[5];
              int data}
MSG m;
```

A struktúra elemeire m.bit[0] … m.bit[4] -ként, illetve m.data -ként hivatkozhatunk.

A PROMELA-ban felépített rendszer modellek alapja a processzek. Vagyis a modellezni kívánt rendszert kommunikáló processzekkel írjuk le, ahol a kommunikációt csatornákkal valósítjuk meg.

- **Processz:**

Egy változó vagy csatorna állapotát csak egy processzen belül változtathatjuk illetve figyelhetjük meg. Process viselkedését proctype deklarációval definiálhatjuk a következõ módon:

proctype *procname* (*formal_parameters*) {*local_declarations*; *statements*}

A pontosvesszõ a kijelentések közti szeparátor (nem kijelentés befejezésére utaló jel, mint például a C-ben, ezért nincs szükség pontosvesszõre az utolsó kijelentés után). A PROMELA kétféle kijelentés szeparátort ismer: a pontosvesszõt: “ ; ”, és a nyilat: “ -> ”. Ez a két szeparátor ekvivalens. A nyíl szeparátort gyakran használják arra, hogy informális módon jelöljék az okozati összefüggést két kijelentés között.

B processz egy kijelentést tartalmaz, amely a **state** változó értékét csökkenti 1-vel, és mivel ez a kijelentés mindig **B** típusú processzek késleltetés nélkül teljesülnek. Az **A** típusú processzek végrehajtása viszont addig késleltetõdik, amíg a feltétel nem teljesül, vagyis a **state** változó értéke el nem éri a kívánt

- **PROMELA modell:**

A proctype definíció csak a processz viselkedését deklarálja, végrehajtásra nem alkalmas. Kezdetben a PROMELA modellben csak egy processz hajtódik végre, ez az init processz, ezt explicit deklarálni kell minden PROMELA specifikációban. Ezek alapján a legkisebb PROMELA specifikáció:

```
init { skip }
```

A skip utasítás az az utasítás ami nem csinál semmit. A kezdõ processz tud globális változókat inicializálni, és processzeket példányosítani. A processz példányosításra szolgál a run utasítás:

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

A run utasítás akkor nem hajtható végre, ha a processz nem példányosítható, például egy ilyen eset, ha már túl sok processz fut. A run utasítással alapadattípusú (csatorna is) paraméter értéket adhatunk át az új processzeknek, viszont adat tömbök, illetve processz típusok nem adhatok át. A run utasítás nem csak a kezdõ processzben, hanem bármelyik processzben alkalmazható. Egy processz eltûnik, amint befejezõdik, ez akkor történik meg, hogyha a kódja végére ér, viszont ha az adott processz indított más processzeket, elõbb

A run utasítással egy processz típusból akárhány processzt létrehozhatunk. Ha több, mint egy konkurens processz olvashatja, illetve írhatja ugyanazt a globális változót, akkor a már jól ismert probléma halmazzal találkozunk, a kölcsönös kizárás problémájával, amely megoldására több, a szakirodalomban ismert lehetõség van, például kritikus szekció programban történõ megvalósítása. A PROMELA-ban még egy nyelvi lehetõség van a kölcsönös kizárás megvalósítására, ez az atomic kifejezés. Kijelentések sorozatát atomic kifejezéssel egybe zárva a felhasználó kijelentheti, hogy az egybe zárt kijelentéseket egy oszthatatlan egységként kell végrehajtani. Ilyenkor az egybe zárt utasítások közé nem ékelõdhet be más processz utasítása. A következõ példa az atomic használatát mutatja be, amikor két konkurens processz ugyanazt a globális state változót akarja használni:

```
byte state = 1;

proctype A ( )
{       atomic {
                (state==1) -> state = state + 1
               }
}

proctype B ( )
{       atomic  {
                (state==1) -> state = state - 1
               }
}

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

Így a state
végre. A másik processz örökre blokkolódik.

Végeredményben egy általános PROMELA specifikáció a következõképpen néz ki:

global_declarations;
proctype *procname*1 (*formal_parameters*1) {*local_declarations*1; *statements*1};
…
proctype *procname*n (*formal_parameters*n) {*local_declarations*n; *statements*n};
init { … run(*procname*j) … run(*procname*k) … }
never { … }

A never kulcsszó után adhatjuk meg az igényeinket, hogy mik azok, amiket elvárunk, hogy a specifikált rendszer teljesítsen. Ugyanezt megtehetjük lineáris temporális logikában leírt formulával is. Ezekrõl

2.4. Kijelentések

Egy kijelentés *engedélyezett* vagy *blokkolt*. Ha *blokkolt*, akkor a kijelentés végrehajtása azon a ponton ahol éppen tart, megragad, amíg a kijelentés nem lesz.

Kijelentés formája:

kijelentés ::= *kijelentés* vagy *kijelentés; kijelentés* vagy *kijelentés* -> *kijelentés*

_____ végrehajtódik, majd az ciklus kiértékelése elölrõl kezdõdik. Ha egyszerre több opció is engedélyezett, akkor ezek közül az egyik ***nemdeterminisztikus*** módon kiválasztódik, és végrehajtódik. Ha egyik _____ se engedélyezett, akkor az else õr engedélyezõdik, ha van. A ciklusból break vagy goto utasítással lehet kilépni. Formája:

do
:: *kijelentések*
…
:: *kijelentések*
od

- **Csatorná ba írá s**illetve **olvasá s**mûveletének formája:

  Írás: q!*vâtozó1*,*konstans*,*vâtozó2*, …
  Olvasás: q?*vâtozó1*,*konstans*,*vâtozó2*, …

  Itt q a csatorna, és q-nak inicializáltnak kell lennie.
  Ha q pufferelt:
  - q! … (írás) engedélyezett, ha q nincs tele. A q! … hatására a csatorna , *konstans*, *vâtozó2*, …).
  - q? … (olvasás) engedélyezett, ha q nem üres, és a csatorna legfelsõ elemének: (*v1*, *c*, *v2*, …) *c* konstansa megegyezik az olvasni akart *konstans* *v1*, *c*, *v2*, …) kivesszük a csatornából, és *vâtozó1*-nek megfeleltetjük *v1*-t, *vâtozó2*-nek megfeleltetjük *v2*-t, stb…

  Ha q nem pufferelt (szinkron csatorna):
  - q! … (q? …) engedélyezett, ha létezik egy evvel a mûvelettel összhangban levõ q? … (q! …) mûvelet, és ezek szimultán végrehajthatóak, és a konstansok megegyeznek. A hatására a kimenõ értékek hozzárendelõdnek

2.5. Példa

Mit csinál a következõ PROMELA program?

```
chan RQ[2] = [1] of {bit};

proctype C(byte j)
{
        do
        :: RQ[j-1] ! 1
        od
}

proctype S( )
{
        bit x;
        do
        :: RQ[0] ? x;
        :: RQ[1] ? x
        od
}

init { atomic {run C(1); run S( ); run C(2) } }
```

3. SPIN

A program grafikus felületét elindítva (Xspin) egy könnyen kezelhetõ PROMELA editort kapunk, szintaxis ellenõrzési lehetõséggel. Lehetõségünk van bármilyen szöveg file megnyitására is.

3.1. Szimulá ció

A szintaktikailag helyes PROMELA programot a SPIN segítségével "lejátszhatjuk". Erre a szimulációra három különbözõ módot kínál a SPIN:

- **Random**, azaz véletlenszerû: a SPIN elkezdi végrehajtani a programot, ha nemdeterminisztikus
- **Interaktív** : a SPIN a program végrehajtása során minden nemdeterminisztikus választáskor a felhasználótól megkérdezi, hogy melyik úton menjen tovább.
- **Guided**, azaz vezetett: a végrehajtás egy ellenpélda, amely automatikusan generálódott egy

A szimulációt nyomon lehet követni, erre is több lehetõség van:

- **MSC**, azaz Message Sequence Chart: a SPIN kirajzolja az idõben az egyes processzek közti üzenet küldéseket. Minden processznek (beleértve az init processzt is) egy szál felel meg, és a szálak között láthatóak az üzenetek értékeikkel, illetve, hogy melyik csatornán küldték õket.
- **Vá ltozók**
  is.

3.2. Verifiká ció

valamilyen módon meg kell adni, hogy mit tekintünk helyesnek, azaz meg kell fogalmaznunk az elvárásainkat.

Vannak standard elvárások, amelyek külön megfogalmazásához nincs szükség, a SPIN automatikusan ellenõrzi ezeket. Ilyen például a deadlock (holtpont) hiányának ellenõrzése, az olyan PROMELA kód részletek ellenõrzése amelyeket a program sose használ, puffer túlcsordulása stb.

Megjegyzés
végeztével kapunk egy listát, melyben a felsorolt nem elért PROMELA kódrészletek között találjuk ezen processzek végét jelölõ kódot. Mivel célunk volt, hogy ezen processzek ezeket ne is érjék el, ez ebben az

3.2.2. Elvá rá sok megfogalmazá sa

Elvárások megfogalmazásának három módja van:

- **Követeléseket** fogalmazhatunk meg PROMELA-ban, amelyek mindig engedélyezettek, és a PROMELA modellben bárhol elhelyezhetõek. Formája:

  assert (*feltétel*)

  A *feltétel*-nek egy Boolean kifejezésnek kell lennie. Ha a igaz, akkor a követelésnek nincs hatása. Ha a a végrehajtás során egyszer is hamis lesz, amikor a követelés kiértékelésére kerül sor, a PROMELA modell végrehajtása megszakad. Például:

  ```
  bit X, Y;

  proctype C( )
  {
          do
          :: X=0; Y=X; assert (X==Y)
          :: X=1; Y=X
          od
  }

  init { run C( ) }
  ```

  Ebben a példában azt a követelést fogalmaztuk meg, hogy elvárjuk, hogy az Y=X értékadás után a két változó értéke megegyezzen. Ha ez nem teljesülne a SPIN hibát jelezne.

- **Címkéket** adhatunk meg egyes PROMELA kódrészletekhez, hogy a SPIN számára jelezzük, hogy az adott kódrészletre valamilyen szempontból fordítson figyelmet. Többféle címke típus van:
  - Egy véges állapotú rendszerben minden végrehajtás véges számú állapotátmenet után vagy befejezõdik egy végállapotban, vagy visszatér egy elõzõleg már meglátogatott állapotba. Nem minden befejezõ állapot érvényes, például a deadlock állapotok, illetve lehetnek a modellben különbözõ felvett hiba állapotok is, ezért a PROMELA-ban megkülönböztethetjük, hogy mely végállapotok érvényesek, és melyek érvénytelenek. Ugyanígy egy processz végrehajtása lehet ciklikus, amikor a végállapotát soha nem éri el, viszont a processz mûködése így helyes, és valamely belsõ állapota felel meg egy elvi végállapotnak. Ekkor képesnek kell lennünk rá, hogy ezt jelöljük, és erre szolgál az ***end state label***. Formája: a kívánt kódrészlet elé az end: címkét kell helyezni.
  - A SPIN-t utasítani lehet, hogy megtalálja az összes érvénytelen ciklikus utasítás sorozatot. A kérdés csak az, hogy mi tesz egy ciklust érvénytelenné vagy érvényessé. Az érvénytelen ciklust úgy definiálhatjuk, hogy egy olyan véges számú utasításból álló kódrészlet, ami végtelen gyakran ismétlõdik, anélkül, hogy a modellben “haladás” történne. A felhasználó a ***progress state label*** segítségével definiálhatja, hogy melyek azok az utasítások, melyeknek “haladniuk” kell. Formája: a kívánt kódrészlet elé az progress: címkét kell helyezni, illetve, ha több helyre is szeretnénk ilyen címkét tenni, bármilyen progress-sel kezdõdõ címke érvényes, például: progress15.

acceptance state label segítségével tehetjük meg. Egy ilyen címkével jelölhetjük azt a kódrészletet, amelynek nem kéne részének lennie egy végtelenül gyakran ismétlõdõ utasítás sorozatnak. Formája: a kívánt kódrészlet elé az accept: címkét kell helyezni, illetve, ha több helyre is szeretnénk ilyen címkét tenni, bármilyen accept-tel kezdõdõ címke érvényes.

- Az elvárásainkat megfogalmazhatjuk **Lineáris Temporális Logika** segítségével is. Erre a SPIN egy külön editort kínál, mely az LTL logikában megadott formulát PROMELA-ra fordítja, és ezt a init processz után kell helyeznünk, és a never kulcsszóval kezdõdik. Ez az úgynevezett ***never claim*** egy speciális processz, amely a PROMELA modell többi részével párhuzamosan fut, a benne levõ kijelentések feltételek, a modell futására nincs hatásuk.

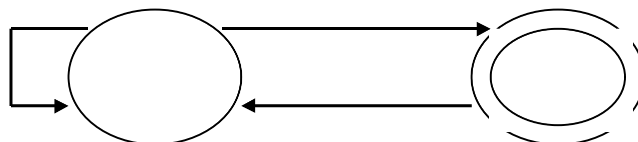
Az LTL formulák formája a következõ lehet:

F ::= p | true | false
| !F | F && F | F || F | F -> F | F < -- > F
| [] F
| <> F
| F **U** F

ahol:
p ::= *bármilyen boolean kifejezés* | proctype [pid] @label

[] F jelentése, hogy az F formula az <u>idõ során</u> **mindig** igaznak kell lennie. <>F jelentése, hogy az F formula az <u>idõ során</u> valamikor **végül is** igaz lesz F1 **U** F2 jelentése, hogy **amíg** F1 formula igaz, **addig** F1 formulának is igaznak kell lennie.

ul azt szeretnénk megfogalmazni, hogy a p boolean kifejezés az idõ során végül is ig , és ez mindig, azaz többször is bekövetkezzen. Ez LTL formulával m zva: []<>p. Ezt a következõ véges automata írja le:



Így a ***never claim***

```
never {
        T0      :  if
                   :: ( true ) -> goto T0
                   :: ( p )     -> goto accept
                   fi
        accept:  if
                   :: ( true ) -> goto T0
                   fi
        }
```

Megjegyzés: SPIN-nel mikor verifikációt készülünk futtatni, a verifikációs opciók beállításánál az “advanced” beállításoknál célszerû a fizikai memóriát a valós értékre beállítani.