Logika dla informatyków

Andrzej M. Borzyszkowski

Instytut Informatyki
Uniwersytet Gdański

inf.ug.edu.pl/~amb

SPIN i Promela

SPIN i Promela – wprowadzenie

Promela = Process meta language

- język do modelowania procesów współbieżnych
- programy napisane w języku Promela można zarówno wykonywać/symulować, jak i weryfikować
- *symulacja* to pojedyncze wykonanie
- *weryfikacja* bierze pod uwagę wszystkie możliwe wykonania

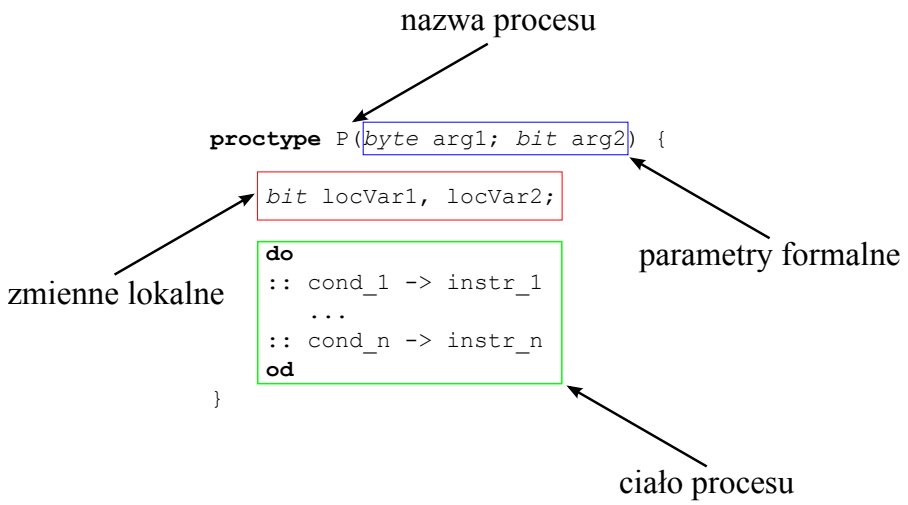
SPIN = Simple Promela Interpreter

- weryfikator modelowy opracowany w Bell Labs, laureat *Software System Award* (2002) – podobnie jak *Unix*, TEX, *SmallTalk*, *Postscript*, *TCP/IP*, *Tcl/Tk*.
- `http://spinroot.com/spin/whatispin.html`

SPIN i Promela jako narzędzie modelowania

- Możemy opisywać skończone systemy tranzycji (modelujące zachowanie zbioru procesów).
- Procesy
  - wykonują się współbieżnie
  - posiadają własny stan lokalny
  - mogą komunikować się za pośrednictwem kanałów (synchronicznie i asynchronicznie) oraz współdzielonej pamięci
  - ich zachowanie zdefiniowane jest poprzez typ procesu (w modelu może działać kilka procesów tego samego typu)
- Pewne ograniczenia:
  - procesy nie mogą być definiowane rekurencyjnie
  - zakres danych oraz liczba procesów muszą być ograniczone
- SPIN „przegląda" system tranzycji, konstruując jego stany w miarę ich odwiedzania

Promela – typ procesu



Promela – HelloWorld

hello.pml:

```
active proctype main () {
    printf("Hello world!\n")
}
```

- **active** – tworzy instancję (bezparametrowego!) procesu
- **proctype** – określa, że `main` jest procesem
- język *podobny* składniowo do (podzbioru) języka C – są różnice, np. „brak średnika" po instrukcji `printf` powyżej (średnik oddziela, a nie kończy instrukcje)
- `main` to nazwa (typ) procesu, a nie słowo kluczowe

SPIN i Promela – typy danych

- Pięć podstawowych typów liczbowych
  - `bit` oraz `bool` – zakres [0..1]
  - `byte` – zakres [0..255]
  - `short` – zakres $[-2^{15}..2^{15}-1]$
  - `int` – zakres $[-2^{31}..2^{31}-1]$ albo $[-2^{63}..2^{63}-1]$
- Domyślną wartością zmiennych typów liczbowych jest 0
- Typy strukturalne: rekordy i tablice
- Ewentualne konflikty typów wykrywane są podczas działania (SPIN to interpreter)

SPIN i Promela – instrukcje

- Instrukcja może być
  - wykonalna – jeśli da się ją natychmiast wykonać, albo
  - zablokowana – w przeciwnym wypadku
- Podstawienie, `skip`, `printf` są zawsze wykonalne
- Wyrażenie (jest także instrukcją) jest wykonalne, jeśli jego wartość jest niezerowa, np.
  - `1 < 3` jest zawsze wykonalne
  - `x < 15` jest wykonalne jedynie, jeśli w aktualnym stanie zmienna `x` ma wartość mniejszą niż 15
  - `x++` jest wykonalne, jeśli w aktualnym stanie zmienna `x` ma wartość różną od -1
- Instrukcja `run` jest wykonalna, jeśli można utworzyć wymagany przez nią nowy process

SPIN i Promela – instrukcje c.d.

- Instrukcja `assert` jest zawsze wykonalna; jeśli wyrażenie będące jej argumentem ma wartość zerową, SPIN przerwie wykonywanie programu z komunikatem o „naruszeniu asercji”
- Instrukcja `if...fi` (podobnie jak `do...od`) jest wykonalna, jeśli co najmniej jedna z instrukcji „strażników” jest wykonalna; strażnik `else` staje się wykonalny, jeśli wszystkie inne instrukcje-strażnicy są zablokowane

```
if/do
 :: choice1 -> stat1.1; stat1.2; …
 :: choice2 -> stat2.1; stat2.2; …
 :: …
 :: choicen -> statn.1; statn.2; …
fi/od
```

jeśli więcej niż jeden strażnik jest wykonalny SPIN dokonuje niedeterministycznego wyboru

SPIN – przeplotowa semantyka równoległości

```
byte n = 0;

active proctype P() {
    n = 1;
    printf("Process P, n=%d\n", n)
}

active proctype Q() {
    n = 2;
    printf("Process Q, n=%d\n", n)
}
```

6 możliwych „przeplotów wykonania”:

1	2	3	4	5	6
n = 1 printf(P) n = 2 printf(Q)	n = 1 n = 2 printf(P) printf(Q)	n = 1 n = 2 printf(Q) printf(P)	n = 2 printf(Q) n = 1 printf(P)	n = 2 n = 1 printf(Q) printf(P)	n = 2 n = 1 printf(P) printf(Q)

W trybie interakcyjnym, SPIN, pozwala „ręcznie sterować” przeplotem:
`spin -i interleave.pml`

SPIN – atomowość akcji

- instrukcje w języku Promela są wykonywane jako „akcje atomowe” (niepodzielne)
- wyrażenia w języku Promela traktowane są jak instrukcje
  - przeplot może nastąpić np. pomiędzy obliczeniem instrukcji strażnika w pętli **do-od**, a wykonaniem następujących po niej instrukcji
- atomowość możemy wymusić otaczając dany ciąg instrukcji/wyrażeń frazą `atomic` lub `d_step`:

```
atomic {
    stat_1;
     …
    stat_n
}
```

lub

```
d_step {
    stat_1;
     …
    stat_n
}
```

(między `atomic` a `d_step` istnieją pewne różnice w sposobie traktowania „niepodzielności akcji” oraz tego, co może znaleźć się w obrębie frazy – patrz dokumentacja)

SPIN – równoległość bywa kłopotliwa...

```
byte n = 0;

active proctype P() {
    byte temp;
    temp = n + 1;
    n = temp;
    printf("Process P, n=%d\n", n)
}

active proctype Q() {
    byte temp;
    temp = n + 1;
    n = temp;
    printf("Process Q, n=%d\n", n)
}
```

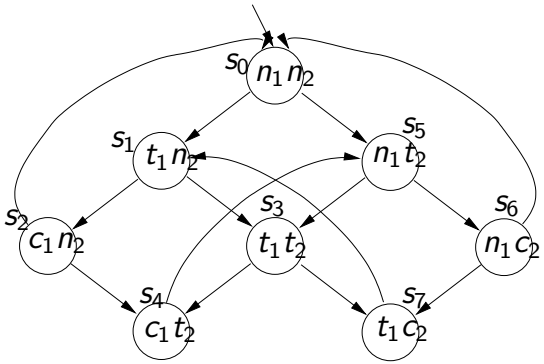
Zbiór procesów

```
byte n = 0;

active [2] proctype P() {
    byte temp;
    temp = n + 1;
    n = temp;
    printf("Process P%d, n=%d\n", _pid, n)
}
```

W żadnym z powyższych programów zmienna `n` **nie musi** przybrać wartości 2.

Wzajemne wykluczanie – *mutual exclusion*



Dwa procesy, 1 i 2 stany

- neutralny
- *trying*: zgłasza żądanie wejścia do obszaru krytycznego
- *critical*: znajduje się w obszarze krytycznym

Obszar krytyczny: część kodu wykonywana z wykorzystaniem wspólnych zasobów.

SPIN – problem „sekcji krytycznej”

```
bool wantP = false, wantQ = false;

active proctype P() {
  do
  :: printf("Noncritical section P\n");
     wantP = true;
     !wantQ;
       printf("Critical section P\n");
     wantP = false
  od
}
```

```
active proctype Q() {
  do
  :: printf("Noncritical section Q\n");
     wantQ = true;
     !wantP;
       printf("Critical section Q\n");
     wantQ = false
  od
}
```

Czy powyższy program jest poprawny, tzn. zapewnia wzajemne wykluczanie i brak zakleszczenia?

SPIN – problem „sekcji krytycznej”

NIE
zakleszczenia są możliwe/pewne
zamiana kolejności operacji prowadzi do niewybaczalnego błędu wejścia dwóch procesów w sekcję krytyczną.

Rozwiązaniem jest transakcyjność, tzn. potraktowanie pewnej liczby operacji jako jednej operacji atomowej wykonywanej w oderwaniu od innych.

SPIN – problem „sekcji krytycznej” c.d.

```
bool wantP = false, wantQ = false;

active proctype P() {
  do
  :: printf("Noncritical section P\n");
     atomic {
        !wantQ;
        wantP = true
     }
     printf("Critical section P\n");
     wantP = false
  od
}
```

```
active proctype Q() {
  do
  :: printf("Noncritical section Q\n");
     atomic {
        !wantP;
        wantQ = true
     }
     printf("Critical section Q\n");
     wantQ = false
  od
}
```

Powyższy program zapewnia wzajemne wykluczanie, oraz brak zakleszczenia wykorzystując sekwencje atomowe.

SPIN – problem „sekcji krytycznej” c.d.

Rozwiązanie ma nadal pewne problemy pragmatyczne.
Mianowicie każdy z procesów musi wyraźnie odnieść się do wszystkich innych procesów. Czyli po dodaniu lub usunięciu jednego procesu trzeba zmieniać kod programu dla innych procesów również. Nie zapewnia to skalowalności rozwiązania.

Rozwiązaniem jest możliwość współdzielenia pamięci, wszystkie procesy obserwują (i modyfikują) jedną zmienną – semafor.

SPIN – problem „sekcji krytycznej” c.d.

Rozwiązanie semaforowe:

```
byte sem = 1, critical = 0;

active [3] proctype P() {
  do
  :: printf("Noncritical section P%d\n",_pid);
     atomic { /* wait(sem) */
       sem > 0;
       sem--
     }
     critical++;
     assert(critical <= 1);
     critical--;
     printf("Critical section P%d\n",_pid);
     sem++    /* signal(sem) */
  od
}
```


SPIN i Promela – kanały

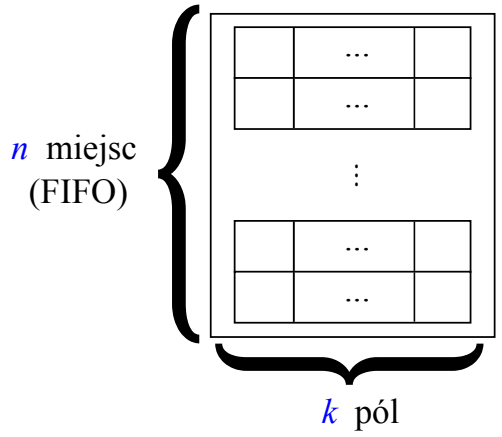
- Służą do przesyłania danych pomiędzy procesami
- Każdy kanał ma (utypowiony) bufor ustalonej, skończonej pojemności, np.

  ```
  chan to_sndr = [2] of byte;
  ```

  deklaruje kanał o nazwie `to_sndr`, z którym kojarzymy bufor typu `byte` o pojemności 2
- specjalny typ `mtype` służy do wprowadzania enumeracji wiadomości (począwszy od wartości 1), np.

  ```
  mtype = { msg, ack };
  chan to_rcvr = [2] of { mtype, bit }
  ```


SPIN i Promela – kanały i bufory




```
chan ch = [n] of { T1, ... , Tk };
```

SPIN i Promela – kanały i bufory

- **Wysyłanie**

  `out_q!ack,sendVal` lub `out_q!ack(sendVal)`

  wykonalne, jeśli w buforze jest wolne miejsce(!)
- **Odbiór**

  `in_q?msg,recVal` lub `in_q?msg(recVal)`

  wykonalny, jeśli bufor jest niepusty(!)
- Kanały mogą być współdzielone przez procesy; na ogół są używane jednokierunkowo
- Jeśli zadeklarowana pojemność bufora kanału wynosi 0, to jest to kanał do komunikacji synchronicznej

SPIN – weryfikacja prostych własności (asercje)

```
active proctype P() {
    int dividend = 15, divisor = 4, res, rem;

    assert(dividend >= 0 && divisor > 0);

    res = 0; rem = dividend;
    do
    :: rem >= divisor -> res++; rem = rem - divisor
    :: else -> break
    od;
    printf("%d divided by %d=%d, rem=%d",
           dividend,divisor,res,rem);

    assert(0 <= rem && rem < divisor);
    assert(dividend == res * divisor + rem)

}
```


SPIN – weryfikacja prostych własności (asercje)

```
active proctype P() {
    int dividend = 15, divisor = 4, res, rem;

    assert(dividend >= 0 && divisor > 0);   // pre-warunek

    res = 0; rem = dividend;
    do
    :: rem >= divisor -> res++; rem = rem - divisor
    :: else -> break
    od;
    printf("%d divided by %d=%d, rem=%d",
           dividend,divisor,res,rem);

    assert(0 <= rem && rem < divisor);      // post-warunek
    assert(dividend == res * divisor + rem)

}
```

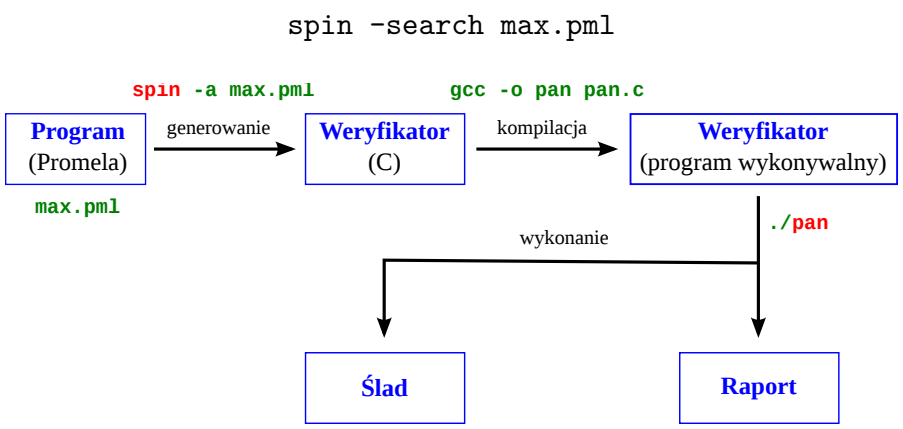

SPIN – weryfikacja własności

Nieco ciekawszy przykład:

```
active proctype P() {
    int a = 5, b = 5, max;
    if
    :: a >= b -> max = a
    :: b >= a -> max = b + 1
    fi;
    assert(a >= b -> max == a : max == b)
}
```

Z powodu niedeterministycznego wyboru wewnątrz instrukcji **if**...**fi** konieczna jest weryfikacja wszystkich możliwych ścieżek obliczeń programu.

SPIN – weryfikacja własności



SPIN – weryfikacja własności

Uruchamiamy „analizator ścieżek” dla przykładu z maksimum:

```
./pan
pan: assertion violated ( ((a>=b)) ? ((max==a)) : ((max==b)) ) (at depth 0)
pan: wrote max.pml.trail

(Spin Version 6.2.3 -- 24 October 2012)
Warning: Search not completed
  ...
State-vector 24 byte, depth reached 0, errors: 1
  ...
```

SPIN znalazł ścieżkę prowadzącą do naruszenia asercji!

SPIN – weryfikacja własności c.d.

Stosując opcję „**-e**” możemy wymusić kontynuację przeszukiwania zbioru ścieżek mimo napotkania naruszenia asercji:

```
./pan -e
pan: assertion violated ( ((a>=b)) ? ((max==a)) : ((max==b)) ) (at depth 0)
pan: wrote max.pml.trail

(Spin Version 6.2.3 -- 24 October 2012)

Full statespace search for:
  ...
  assertion violations +
  ...
State-vector 24 byte, depth reached 2, errors: 1
  ...
```

SPIN nie znalazł więcej ścieżek prowadzących do naruszenia asercji.

SPIN – weryfikacja własności, komunikaty

Stosując odpowiednie opcje możemy prześledzić na czym polagało naruszenie asercji:

```
spin -t -g -p -l max.pml
Starting P with pid 0
  1: proc  0 (P) line   4 "max.pml" (state 3) [((b>=a))]
  1: proc  0 (P) line   4 "max.pml" (state 4) [max = (b+1)]
  P(0):max = 6
spin: line   5 "max.pml", Error: assertion violated saw '-2''
spin: text of failed assertion: assert(( ((a>=b)) -> ((max==a)) : ((max==b)) ))
  1: proc  0 (P) line   5 "max.pml" (state 7) [assert(( ((a>=b)) -> ((max==a)) : ((max==b)) ))]
  P(0):max = 6
spin: trail ends after 1 steps
#processes: 1
  1: proc  0 (P) line   6 "max.pml" (state 8) <valid end state>
  P(0):max = 6
1 process created
```

Promela i LTL

Gramatyka (w postaci Backusa-Naura)

```
ltl ::= opd | (ltl) | uop ltl | ltl bop ltl
opd ::= true | false | nazwa
uop ::= [] | <> | !
bop ::= U | V | && | || | -> | <->
```

Odpowiedniość operatorów: Promela $\leftrightarrow$ LTL

- `[]` $\equiv G$, `<>` $\equiv F$, `!` $\equiv \neg$
- `U` $\equiv U$, `V` $\equiv R$, `&&` $\equiv \wedge$, `||` $\equiv \vee$, `->` $\equiv \rightarrow$

Uwaga

Operator next (`X`)*, chociaż (opcjonalnie) obecny w wariancie logiki LTL dostępnym w Promeli, ma ograniczoną użyteczność i raczej nie należy z niego korzystać w systemie SPIN.*

SPIN: weryfikacja własności wyrażonych w LTL

Aby sprawdzić, czy

$$M \models \varphi$$

gdzie M jest (budowanym przez SPIN) *automatem* reprezentującym dany program w języku Promela, a φ jest własnością wyrażoną w logice LTL, trzeba

- zbudować automat $A_{\neg\varphi}$ kodujący potencjalne naruszenia własności φ
- rozważyć synchroniczne wykonanie M oraz $A_{\neg\varphi}$

Automat $A_{\neg\varphi}$ „monitoruje" działanie M. Jeśli istnieje obliczenie akceptowane przez „złożenie" M oraz $A_{\neg\varphi}$, wówczas demonstruje ono naruszenie własności φ.

Promela: sygnalizator świetlny

`lights.pml`:

```
mtype { red, yellow, green };
mtype light = green;

active proctype TrafficLights() {
    do
    :: if
       :: light == red -> light = green
       :: light == yellow -> light =  red
       :: light == green -> light = yellow
       fi;
       printf("The light is now %e\n", light)
    od
}
```


SPIN – symulacja

```
> spin lights.pml

The light is now yellow
The light is now red
The light is now green
The light is now yellow
The light is now red
The light is now green
The light is now yellow
The light is now red
The light is now green
...
```

SPIN: weryfikacja własności wyrażonych w LTL

- chcemy wyrazić i zweryfikować własność:

  „*zawsze kiedyś zapali się zielone światło*"
- zaczynamy od zdefiniowania „zdania atomowego"

```
#define g_light (light==green)
```

  które zapisujemy w pliku o nazwie `lights.ltl`
- formuła, której prawdziwość chcemy zweryfikować ma postać:

$$GF(g\_light)$$

  w notacji spin-owej

$$\texttt{frm} = \texttt{[]<> g\_light}$$

SPIN: weryfikacja własności wyrażonych w LTL c.d.

- generujemy *automat* (program) $A_{\neg\texttt{frm}}$

```
spin -a -f '!frm' >> lights.ltl
```

  lub

```
spin -a -F light.frm >> lights.ltl
```

- tworzymy „analizator ścieżek"

```
spin -a -N lights.ltl lights.pml
```

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

- i wywołujemy go z linijki rozkazowej

```
./pan
```


Definiowanie własności LTL razem z modelem

- SPIN w obecnej wersji pozwala definiować własności LTL w prostszy sposób

```
ltl willBecomeGreen {[]<> (light==green)}
```

- w specyfikacji modelu możemy umieścić wiele (nazwanych) formuł
- odwołujemy się do nich z pomocą opcji `-ltl`:

```
spin -search -ltl willBecomeGreen spec.pml
```


SAFETY i LIVENESS

Bezpieczeństwo

- „nic złego nie może się zdarzyć"
- przykład – „niezmienniki":
  - „wartość x jest zawsze większa od wartości y"
- najczęściej wyrażane z pomocą asercji
- można wykorzystywać też formuły LTL

SAFETY i LIVENESS

Żywotność

- „w końcu zdarzy się coś pożądanego”
- przykład – „reaktywność”:
  - „po pojawieniu się żądania, w końcu zostanie wygenerowana odpowiedź”
- najczęściej wyrażane z pomocą formuł LTL