ALR, weryfikacja

Weryfikacja czyli "model checking" ...

- weryfikujemy algorytmy ASYNCHRONICZNE (lub systemy, urządzenia, ...)
  "message passing", "shared memory", ...
- sprawdzamy czy pewien WARUNEK jest spełniony dla KAŻDEJ możliwej egzekucji algorytmu (konfiguracja początkowa + ciąg zdarzeń obliczeniowych)
- WARUNEK jest wyrażony w logice temporalnej (LTL= Linear TL); np:
  "<>[] X" w każdej egzekucji, od pewnego momentu spełniony jest warunek X,
  "[] X" w każdej egzekucji, zawsze jest spełniony X
- "model checking", a nie "program checking"
  chodzi o "model" algorytmu (uproszczony!)
- omawiamy dwa skrajne podejścia:
  - Spin/Promela - C podobny j. programowania/modelowania
  - Sieci Petriego - model buduje się graficznie, z "miejsc" i "tranzycji"

ALR, weryfikacja, Spin/Promela

http://spinroot.com/

http://spinroot.com/spin/Man/promela.html

"Promela" to uproszczony j. C wyposażony w dodatkowe konstr./idee:

1. *typy zmiennych:* tylko całkowite (bit, bool, byte, short, int) oraz tablice; trzeba używać OSZCZĘDNIE !

2. w zasadzie NIE MA tu procedur !? (bo chodzi o modelowanie, nie programowanie)

3.
```
// tworzy N współbieżnie działających procesów:
active [N] proctype mojProces() {
  // kod procesu
}
```

4.
```
// kanały (do "message passing")
chan mojKanal = [max_liczba_kom] of {lista_typów}
      // każdy komunikat to ciąg liczb, plus ew. typ "mtype" (=enum)
// oraz instrukcje do wysyłania/odbierania komunikatów:
mojKanal ! wart1, wart2, ...; // wysyłanie, może blokować
mojKanal ? zm1, zm2, ...; // odczyt, może blokować
```

5.
```
// wyrażenie LTL z etykietą w kodzie programu
ltl L1 {<>[] (a==6)}
  // od pewnej chwili jest spełniony warunek a==6; a - zm. globalna
ltl L2 {[] (a1==a2)}
  // zawsze jest spełniony warunek a1==a2
```

6.
```
// non-determinizm (modelowanie, nie programowanie (?))
if
:: wyr1 -> instr1a; instr1b; ...; // wykonują się jeśli warunek "wyr1" spełniony
:: wyr2 -> instr2a; instr2b; ...;
:: instr3; ...; // może być bez warunku
:: else -> instr; ...; // jeśli żaden wyrX nie jest spełniony (bez instr3 (?))
fi
```

7.
```
// pętla; najlepiej jeśli zawsze 1 z wyrX jest spełniony
do
:: wyr1 -> instr1a; instr1b; ...; // wyjście z pętli przez "break"!
:: wyr2 -> instr2a; instr2b; ...;
:: else -> instr; ...;
od
```

8.
```
// "dziwne" konstrukcje:
wyr -> instr_a; instr_b; ...; // BLOKUJE, jeśli wyr NIE jest spełniony
wyr; instr_a; instr_b; ...;    // ... to samo (!)
zm = (war -> wart1 : wart2); // trochę inaczej niż w C
```

"Spin" to program, który interpretuje programy w promeli...

1. "spin" potrafi uruchomić program w Promeli (non-det jest losowy):
   spin mojprog1.pml

2. "spin" potrafi wygenerować program w j. C,
   który bada wszystkie możliwe egzekucje,
   i sprawdza czy warunek LTL jest spełniony ...
   to jest właśnie **weryfikacja** ...
   UWAGA na eksplozję kombinatoryczną - główne niebezpieczeństwo !!

3. *weryfikacja spin-em/ zasada działania:*
   każdy proces ma skończoną liczbę stanów,
   konfiguracja to stany wszystkich procesów
   + wart. zm. globalnych + zawartośc kanałów,
   możemy przejść z danej konfiguracji do innej poprzez **zdarzenie obliczeniowe**

4. *zdarzenie obliczeniowe:*
   wykonanie pojedynczej instrukcji programu w Promeli,
   zdarzenie obl. to "id procesu"; *ale co z non-det ?!?!? trzeba podać "opcję" ...*

5. nakładka graficzna na program "spin": ispin.tcl;
   ważne opcje: "Verification", "Simulate / Reply";
   jeśli podczas weryfikacji prog np01.pml wystąpi błąd,
   to jest tworzony plik np01.pml.trail
   który można potem "prześledzić" ...

Przykład: 5 procesów zwiększających zm. globalną "a" o 1, 3x...

```
byte a= 0; // wydaje się, że po jakimś czasie powinno być a==15

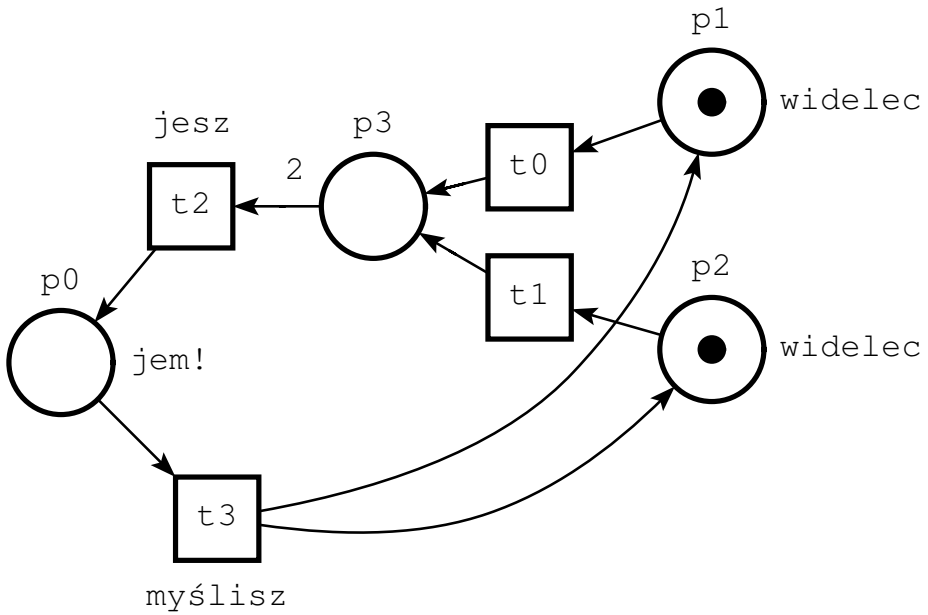
active [5] proctype P() {
  byte tmp, i;
  i=0;
  do
  :: i>=3 -> break;
  :: else -> tmp=a; tmp++; a=tmp;
      // tak robią "a++" prymitywne procesory (na rejestrach typu AX itp)
    i++;
  od
}

ltl L1 {<>[] (a==15)}
ltl L2 {! <>[] (a==4)}
ltl L3 {! <>[] (a==2)}
  /*
    L1 - klasyczne sprawdzenie, że algorytm nie działa...
    L2 - jest po to aby pokazać, że na końcu może być a==4
    L3 - nie dziala, wlacza sie OOM killer !!
  */
```

ALR, weryfikacja, Sieci Petriego

Sieć Petriego składa się z **miejsc** (kółka) zawierających pionki,
oraz z **tranzycji** (prostokąty), które mają krawędzie we i wy,
krawędzie łączą miejsca i tranzycje,
tranzycja jest **aktywna** jeśli wszystkie miejsca we maja ≥ 1 pionków,
wykonanie aktywnej tranzycji usuwa po 1 pionku z miejsc we,
i dodaje po 1 pionku do miejsc wy...,
liczba pionków w każdym miejscu to **konfiguracja, stan, marking**

Przykład: *uwaga na inny sposób podnoszenia i odkładania widelców !*



Właściowści sieci petriego (pn = petri net), które się bada:

- Niech N oznacza pn, M_0 początkową konfigurację,
  a $R(N, M_0)$ zbiór konfiguracji osiągalnych (poprzez egzekucję N od M_0);
  Uwaga: egzekucja w pn to ciąg aktywnych tranzycji !!

- **Ograniczoność, bounded pn**
  pn jest ograniczona jeśli dla wszystkich konfig. z $R(N, M_0)$ liczba pionków w każdym miejscu jest ograniczona

- **Żywotność, live pn**
  konfig. jest "live" jeśli da się z niej uaktywnić każda tranzycje,
  pn jest "live" jeśli każda konfiguracja z $R(N, M_0)$ jest "live"

- **Odwracalność, reversible pn**
  pn jest odwracalna, jeśli dla każdej konfig. z $R(N, M_0)$ da się wrócić do M_0

Jak się analizuje sieci petriego?

- **Reachability tree/graph**
  tree: drzewo skierowane, wierz/konfiguracje, kraw/aktywne tranzycje,
  konfig "dead-end", konfig "old" (powtórzenie konfig w ścieżce od korzenia);
  graf: "old" zastępujemy przez pętle ...
  r.tree/graph ma sens dla ograniczonych pn!

- **Coverability tree/graph**
  ω = nieskończoność, $\omega + a = \omega$, $\omega - a = \omega$, $\omega > a$, a - liczba;
  m_1, m_2 - konfig, $m_1 \geq m_2$ (rozumiane po składowych);
  tree: budujemy reachability tree...
  jeśli natrafimy na konfig większą niż inna konfig w ścieżce od korzenia, to
  większe składowe zastępujemy przez ω
  graf: analogicznie jak r.graph...
  czasami c.graph jest skończony a r.graph nie!

- *przykłady na tablicy...*

Oprogramowanie dla sieci petriego: np. TINA

http://projects.laas.fr/tina
http://projects.laas.fr/tina//papers.php

"Tina" obsługuje time pn (= tpn) ...

- jest to zbiór programów graficznych i tekstowych,
  takze std. formatów plików przechowujących pn i ts (=transition system), ...
- program "nd" jest graficznym edytorem do
  budowania, symulowania i analizowania pn ...
- można zbudować grafy reachability lub coverability,
  oraz zbadać czy pn jest: bunded, live, reversible,
  oraz wiele innych rzeczy ...
- ```
  // program tina buduje "reachability graf" dla pn
  tina plik.ndr plik.ktz
      // .ndr to graficzny zapis pn, .ktz to ts (=transiton system)
  // program "selt" sprawdza warunek LTL dla pn:\\
  selt np01.ktz -f '<>([](p0<=3))'
      // w warunku występują nazwy miejsc (l. pionków) i tranzycji (czy aktywna)
  ```
  ```

ALR, weryfikacja, Logika Temporalna, LTL

operatory logiki temporalnej oraz przykłady:

1. [](war), w każdej egzekucji zawsze war jest spełniony (tzn w każdej konfig)
2. <>(war), w każdej egzekucji choć raz war jest spełniony
3. <>[](war), ... od pewnej chwili war jest spełniony (b. przydatne !)
4. []<>(war), ... nieskończenie wiele razy war jest spełniony
5. ($war1$)U($war2$), "until", ... war1 jest spełniony, aż war2 bedzie spełniony