Weryfikacja modelowa
Wprowadzenie

Marcin Szpyrka

Katedra Informatyki Stosowanej
AGH w Krakowie

2015/16

Literatura

1. Christel Baier, Joost-Pieter Katoen: Principles of Model Checking. The MIT Press, 2008
2. Michael Huth, Mark Ryan: Logic in Computer Science. Modelling and Reasoning about Systems. Cambridge University Press, 2004.
3. Marcin Szpyrka: Sieci Petriego w modelowaniu i analizie systemów współbieżnych. WNT, Warszawa, 2008
4. Tomasz Szmuc, Marcin Szpyrka: Metody formalne w inżynierii oprogramowania systemów czasu rzeczywistego. WNT, Warszawa, 2010
5. Marcin Szpyrka: Modelowanie systemów współbieżnych w języku Alvis. Wydawnictwa AGH, Kraków, 2013
6. Strony internetowe wskazane na stronie www wykładu.

Kontekst

- Rozszerzający się rynek zastosowań systemów informatycznych powoduje wzrost zapotrzebowania na systemy, które powinny być rozwijane szybko, jak najmniejszym kosztem, przy jednoczesnym zagwarantowaniu wysokiej jakości produktu końcowego.
- Dla znacznej części wytwarzanych systemów informatycznych, podstawowym wymaganiem jest ich jakość (tj. brak błędów), a nie wydajność np. systemy krytyczne ze względu na bezpieczeństwo (safety critical systems).
- Znaczna ilość systemów to systemy wbudowane, które są połączone i współpracują z innymi komponentami i/lub systemami. Typowe zjawiska w takich systemach to współbieżność i niedeterminizm, które są trudne do sprawdzania z użyciem tradycyjnych technik testowania oprogramowania.
- Usuwanie błędów odkrytych w fazie użytkowania systemu może być zadaniem trudnym i bardzo kosztownym, np. w przypadku projektowania sprzętu.
- Przy ocenie bezpieczeństwa systemów teleinformatycznych stosowane są normy, które w zależności od poziomu bezpieczeństwa zalecają lub wymagają zastosowania metod formalnych: ITSEC (Information Technology Security Evaluation Criteria), Common Criteria (norma ISO 15408).

Dramatyczne przykłady błędów systemów informatycznych

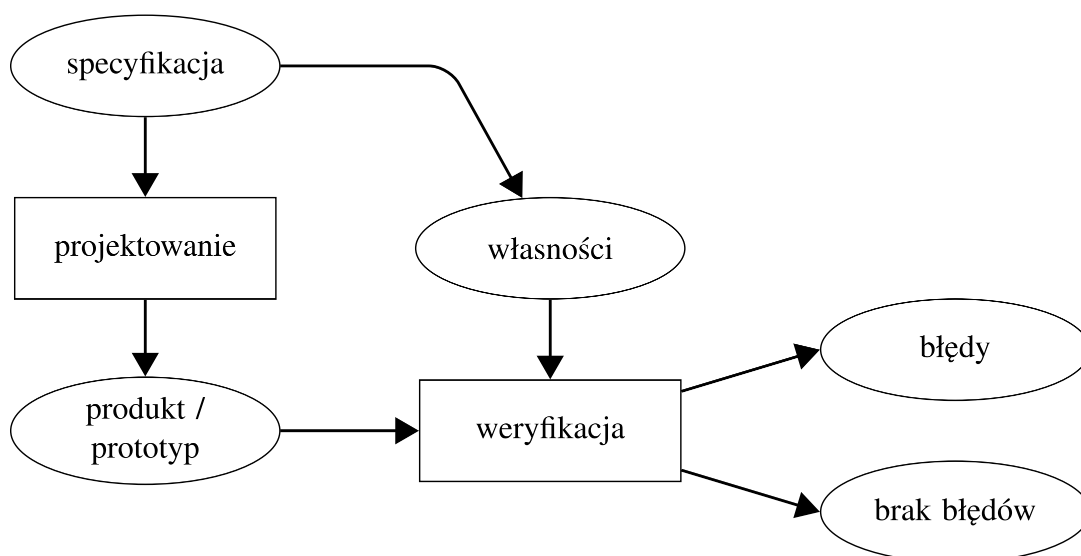
- Intel Pentium FPDI (Floating Point Divide Instructions) bug – 1994 r., istniało ryzyko wystąpienia błędów przy operacji dzielenia liczb zmiennoprzecinkowych, koszt wymiany wadliwych procesorów – 475 milionów USD, nieoszacowane straty na reputacji firmy Intel;
- Zniszczenie rakiety Arian-5 – 1996 r., wybuch rakiety 40 s po starcie (w pierwszym locie po dekadzie przygotowań), przyczyna: błąd konwersji 64 bitowej liczby zmiennoprzecinkowej na 16 bitową liczbę całkowitą (przekroczenie zakresu), strata ok. 500 milionów USD;
- Katastrofa lotu Lufthansa 2904 – 1993 r. Okęcie, system hamowania zadziałał za późno, ponieważ lewe koło dotknęło ziemi 9 sekund później niż prawe – z powodu działania silnego wiatru oraz tafli wody na płycie lotniska koła nie uzyskały tej samej prędkości kątowej, więc system uznał, że nie dotknęły podłoża; skutki: śmierć drugiego pilota i 1 pasażera, 51 osób zostało ciężko rannych, 5 lekko rannych;
- Mars Polar Lander – 1998 r., sonda uległa zniszczeniu z powodu nadmiernego zbliżenia się do powierzchni planety, przyczyna: zespół angielski inżynierów pracujących przy projekcie zastosował anglosaskie jednostki zamiast metrycznych, co spowodowało, że sonda w rzeczywistości była niżej niż widział to system nadzorujący lot; skutki: strata 125 milionów USD;

Współbieżne procesy – przykład

```
int x = 0;

// Inc
while(1)            // 1
  if(x < 200)       // 2
    x = x + 1;      // 3

// Dec
while(1)            // 1
  if(x > 0)         // 2
    x = x - 1;      // 3

// Reset
while(1)            // 1
  if(x == 200)      // 2
    x = 0;          // 3
```

Wymagania: $x \in [0, 200]$

Weryfikacja systemu (posteriori)



Poprawność systemu rozważa się zawsze w odniesieniu do specyfikacji.
Weryfikacja modelowa dotyczy weryfikacji, w której specyfikacja systemu jest wyrażona formalnie.

Weryfikacja oprogramowania

- Peer reviewing – Jest to statyczna analiza kodu źródłowego. Pozwala na wychwycenie między 31% a 93% błędów (średnio 60%). Jest stosowana w ok. 80% projektów informatycznych. Nie pozwala np. na wychwycenie błędów dotyczących współbieżności.
- Testowanie – Jest to dynamiczna analiza oprogramowania. Pochłania od 30% do 50% kosztów projektu. Nie pozwala na kompletne przeanalizowanie wszystkich możliwych ścieżek.
- Przy wyszukiwaniu błędów w działaniu systemu obowiązuje zasada; im szybciej tym lepiej. Usunięcie błędu w wdrożonym systemie jest około 500 razy droższe niż usunięcie tego samego błędu we wczesnej fazie projektowania.
- Około 50% błędów jest wynikiem fazy programowania – średnio ok. 20 błędów na 1000 linii kodu (bez komentarzy). Na etapie udostępniania typowego oprogramowania, akceptowalna liczba błędów to ok. 1 błąd na 1000 linii kodu.
- Przy rozwijaniu złożonych systemów informatycznych więcej czasu i wysiłku poświęca się na weryfikacje systemu niż na jego projektowanie.

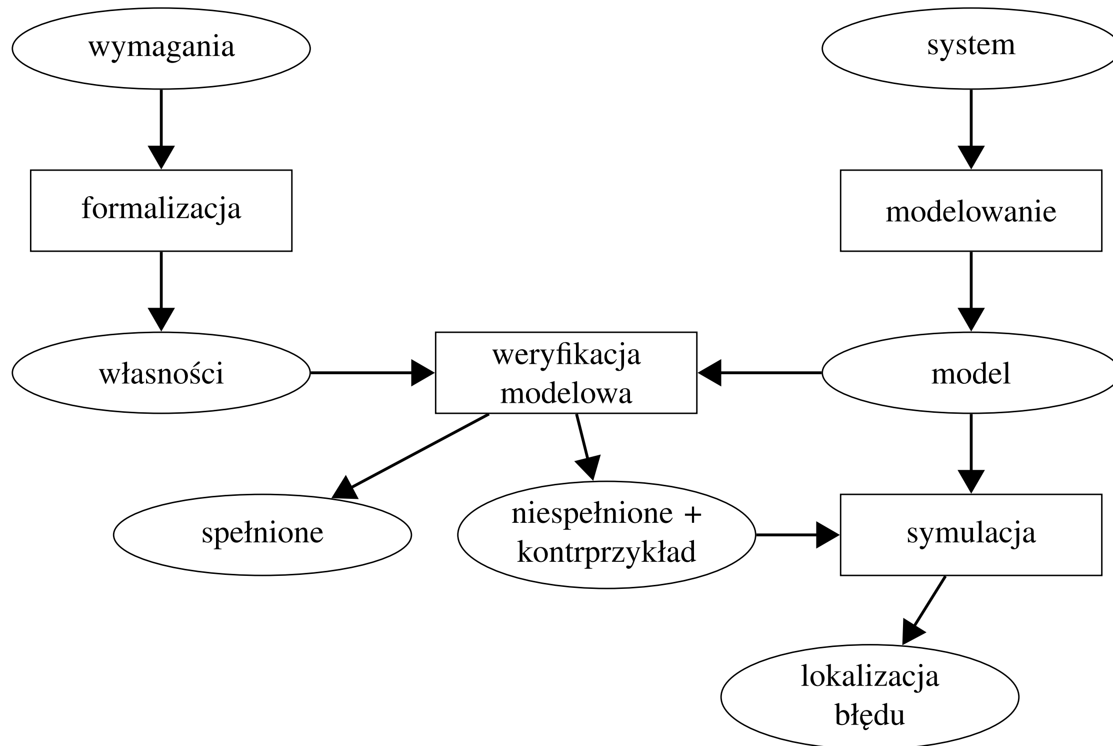
Weryfikacja modelowa (1)

> Formal methods should be part of the education of every computer scientist and software engineer, just as the appropriate branch of applied maths is a necessary part of the education of all other engineers. (z raportu Federal Aviation Authority i NASA dotyczącego zastosowania metod formalnych)

- Metody formalne mogą być traktowane jako matematyka stosowana do modelowania i analizy systemów informatycznych.
- Weryfikacja modelowa opiera się na analizie modelu przygotowanego z użyciem wybranej metody formalnej (model ma jednoznaczną semantykę). Dla przygotowanego modelu analizowany jest zbiór wszystkich możliwych jego stanów. Jednocześnie formalne wyrażenie specyfikacji systemu pozwala wyeliminować z niej błędy takie jak: niekompletność, dwuznaczność itp.
- W ostatnich dwóch dekadach nastąpił znaczny rozwój metod formalnych i narzędzi komputerowych wspierających ich zastosowanie. Badania wykazały, że użycie metod formalnych pozwoliłoby na wykrycie błędów pokazanych na slajdzie 5.

> Any verification using model-based techniques is only as good as the model of the system.

Weryfikacja modelowa – schemat



Weryfikacja modelowa (2)

Model checking is an automated technique that, given a finite-state model of a system and a formal property, systematically checks whether this property holds for that model.

- Termin model checking pochodzi z pracy:
  Edmund M. Clarke, E. Allen Emerson: *Design and Synthesis of Synchronization Skeletons Using Branching-Time Temporal Logic*. In: Logic of Programs, LNCS, vol. 131, pp. 52-71, 1981.
- Technika weryfikacji modelowej została zapoczątkowana niezależnie pracą Clarke'a i Emersona oraz pracą:
  Jean-Pierre Queille, Joseph Sifakis: *Specification and verification of concurrent systems in CESAR*. In: 5th International Symposium on Programming, LNCS, vol. 137, pp. 337-351, 1982.

Weryfikacja i walidacja:
Weryfikacja – sprawdzenie, czy model jest zgodny ze specyfikacją – check that we are building the thing right.
Walidacja – sprawdzenie, czy model jest zgodny z jego nieformalnym opisem – check that we are building the right thing.

Możliwe wyniki

1) Własność jest spełniona – Po pozytywnym sprawdzeniu wszystkich własności mamy pewność, że model jest zgodny ze specyfikacją.

2) Własność nie jest spełniona – Kontrprzykładem w takiej sytuacji jest stan, w którym badana własność nie jest spełniona oraz ścieżka pokazująca w jaki sposób można uzyskać ten stan zaczynając od stanu początkowego.
W przypadku narzędzi komputerowych umożliwiających symulację krokową możliwe jest powtórzenie takiej ścieżki i znalezienie odpowiedzi na pytanie dlaczego możliwe jest uzyskanie stanu, który nie spełnia danej własności.

3) Model jest za duży, by możliwe było jego przeanalizowanie – Problem eksplozji stanów może uniemożliwić wyznaczenie zbioru wszystkich możliwych stanów ze względu na możliwości techniczne sprzętu komputerowego.
Rozwiązaniem w takim przypadku może być budowa modelu na wyższym poziomie abstrakcji (pomijamy szczegóły nieistotne z punktu widzenia analizowanych własności).

Weryfikacja modelowa – zalety

- Jest to uniwersalna technika, którą można stosować dla szerokiego spektrum systemów informatycznych.
- Możliwa jest weryfikacja tylko wybranych własności (nie jest konieczna znajomość kompletnej specyfikacji).
- W przeciwieństwie do symulacji i testowania sprawdzane są wszystkie możliwości.
- W przypadku, gdy własność nie jest spełniona, dostarcza istotnych informacji pozwalających znaleźć przyczynę niepowodzenia.
- Jej stosowanie nie wymaga dużego zaangażowania użytkownika, ani też czasochłonnego przygotowania teoretycznego.
- Wzrasta zainteresowanie weryfikacją modelową ze strony dużych korporacji, czego efektem są m.in. komercyjne narzędzia wspierające jej stosowanie.
- Stosunkowo łatwo można zintegrować weryfikację modelową z stosowanymi cyklami rozwoju oprogramowania. Jej zastosowanie może skrócić czas wytwarzania oprogramowania i poprawić jego jakość.
- Jest to metoda matematyczna.

Weryfikacja modelowa – wady

- Weryfikacja modelowa lepiej sprawdza się dla systemów zorientowanych na przepływ sterowania niż na przepływ danych (np. problem nieskończonych dziedzin).
- Nie może być stosowana dla systemów z nieskończoną liczbą stanów i w przypadku wnioskowania o abstrakcyjnych typach danych.
- Weryfikacja dotyczy modelu, a nie samego systemu.
- Eksplozja stanów może uniemożliwić weryfikację modelu.
- Wymaga umiejętności i doświadczenia w budowaniu modeli na odpowiednim poziomie abstrakcji i umiejętności zapisu wymagań jako formul logicznych.
- Narzędzia komputerowe stosowane do sprawdzania spełniania formuł logicznych mogą same w sobie zawierać błędy.
- Nie może być stosowana do systemów o nieokreślonej liczbie komponentów (bądź liczbie komponentów, która jest parametrem).

System tranzycyjny (system przejść)

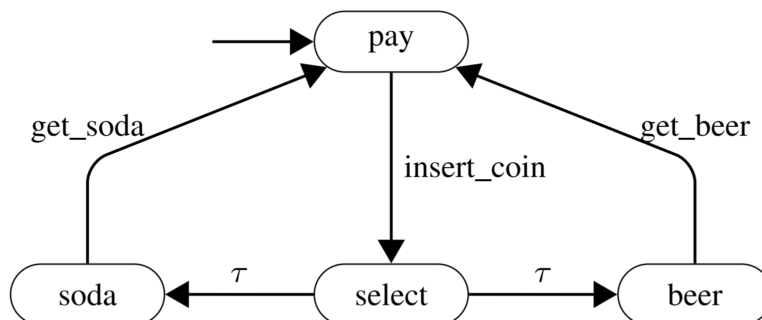
Systemem tranzycyjnym (transition system) nazywamy krotkę $TS = (S, Act, \to, I, AP, L)$, gdzie:

- S jest zbiorem stanów (states);
- Act jest zbiorem akcji (actions), $\alpha, \beta, \dots \in Act$;
- $\to \subseteq S \times Act \times S$ jest relacją przejść (transition relation);
- $I \subseteq S$ jest zbiorem stanów początkowych (initial states);
- AP jest zbiorem formuł atomowych (atomic propositions), $a, b, \dots \in AP$;
- $L \colon S \to 2^{AP}$ jest funkcją etykietującą stany (labeling function).

System tranzycyjny TS nazywamy skończonym, jeżeli S, Act i AP są zbiorami skończonymi.

- Zdania atomowe służą do sformalizowania charakterystyk temporalnych. Wyrażają podstawowe informacje o stanach rozważanego systemu.
- Jeżeli $I = \emptyset$ to rozważany system nie wykazuje żadnej dynamiki. Jeżeli $|I| > 1$, to wybór stanu początkowego jest niedeterministyczny, podobnie jak wybór kolejnego stanu, przy kilku możliwych przejściach z danego stanu s.
- $L(s)$ oznacza zbiór wszystkich formuł atomowych, które są prawdziwe dla stanu s.

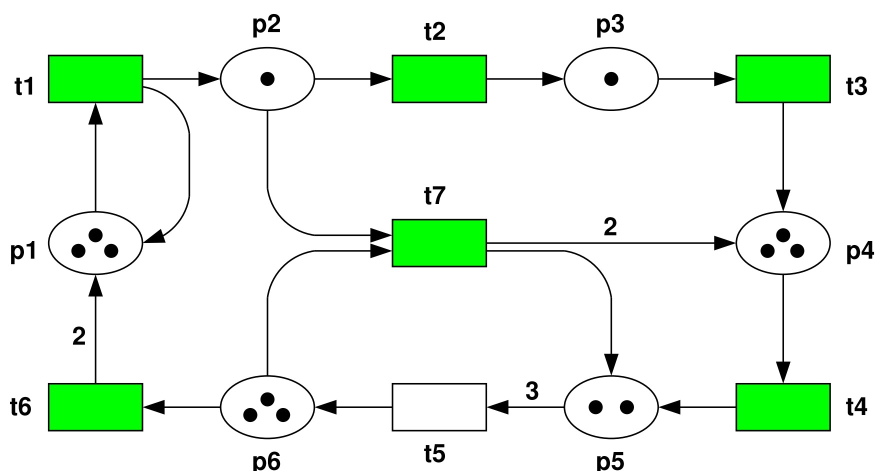
System tranzycyjny – przykład



- $S = \{pay, select, soda, beer\}$
- $Act = \{insert_coin, get_soda, get_beer, \tau\}$, τ reprezentuje wewnętrzne aktywności systemu, które nas nie interesują
- $I = \{pay\}$
- przykłady przejść: $pay \xrightarrow{insert_coin} select$, $beer \xrightarrow{get_beer} pay$
- $AP = \{paid, drink\}$, *paid* – klient zapłacił, *drink* – maszyna wydała napój
- $L(pay) = \emptyset$, $L(soda) = L(beer) = \{paid, drink\}$, $L(select) = \{paid\}$

Uwaga: Pokazana maszyna po wrzuceniu monety wydaje (niedeterministycznie) wodę lub piwo.

Sieci Petriego niskiego poziomu



Sieć miejsc i przejść definiujemy jako piątkę postaci $\mathcal{N} = (P, T, A, W, M_0)$, gdzie

(1) P jest niepustym zbiorem miejsc.

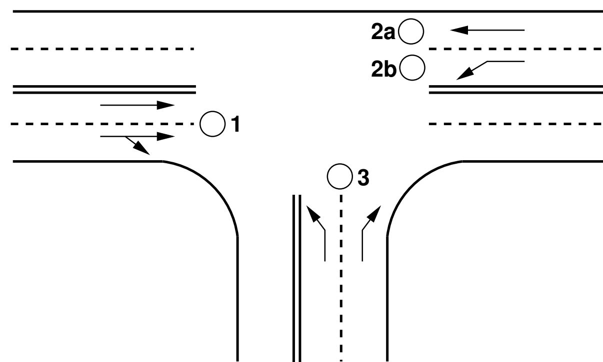
(2) T jest niepustym zbiorem przejść (*tranzycji*) takim, że $P \cap T = \emptyset$.

(3) $A \subseteq (P \times T) \cup (T \times P)$ jest zbiorem łuków.

(4) $W: A \to \mathbb{N}$ jest funkcją wag łuków.

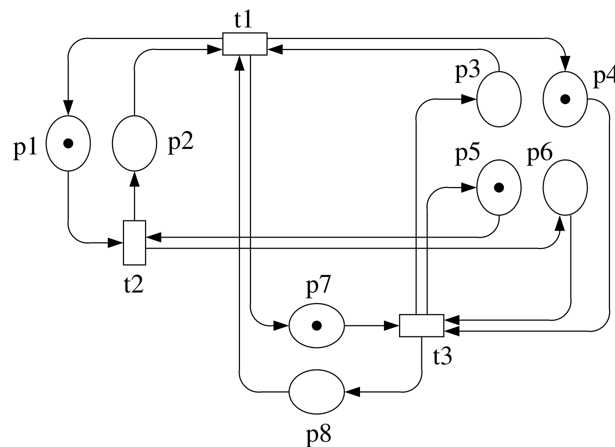
(5) $M_0: P \to \mathbb{Z}_+$ jest znakowaniem początkowym sieci $\mathcal{N}$.

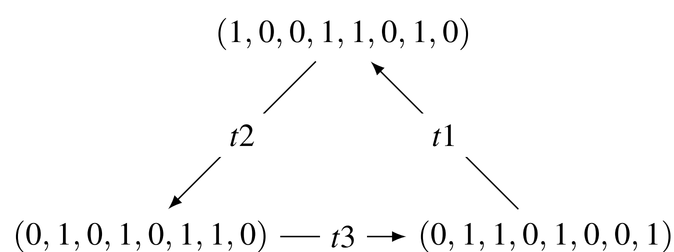
Model sygnalizacji świetlnej – sieć znakowana (1)



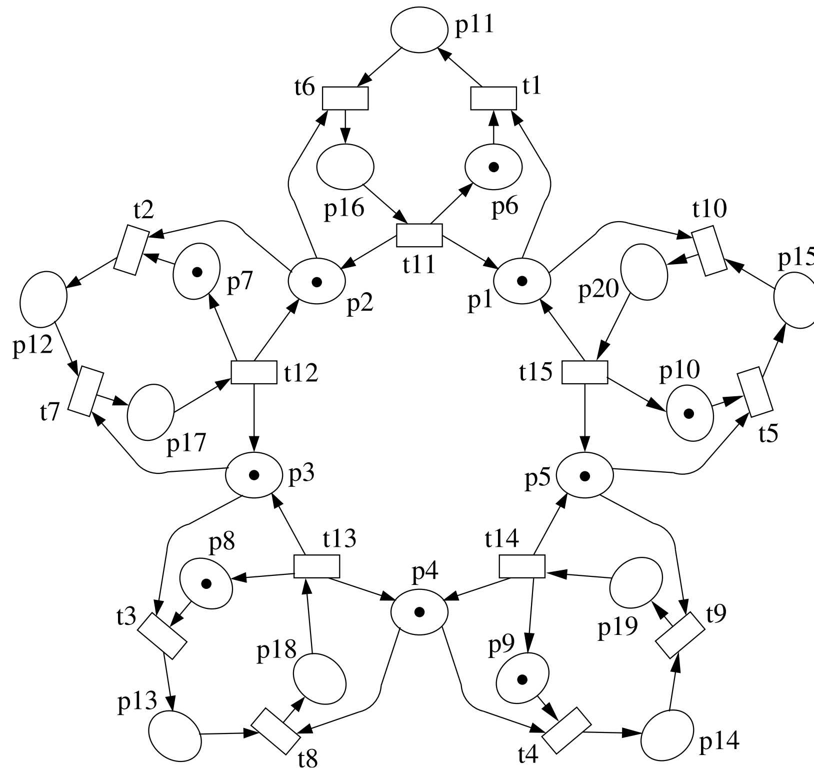
Stan	1	2a	2b	3
1	zielony	zielony	czerwony	czerwony
2	czerwony	zielony	zielony	czerwony
3	czerwony	czerwony	czerwony	zielony

Model sygnalizacji świetlnej – sieć znakowana (2)

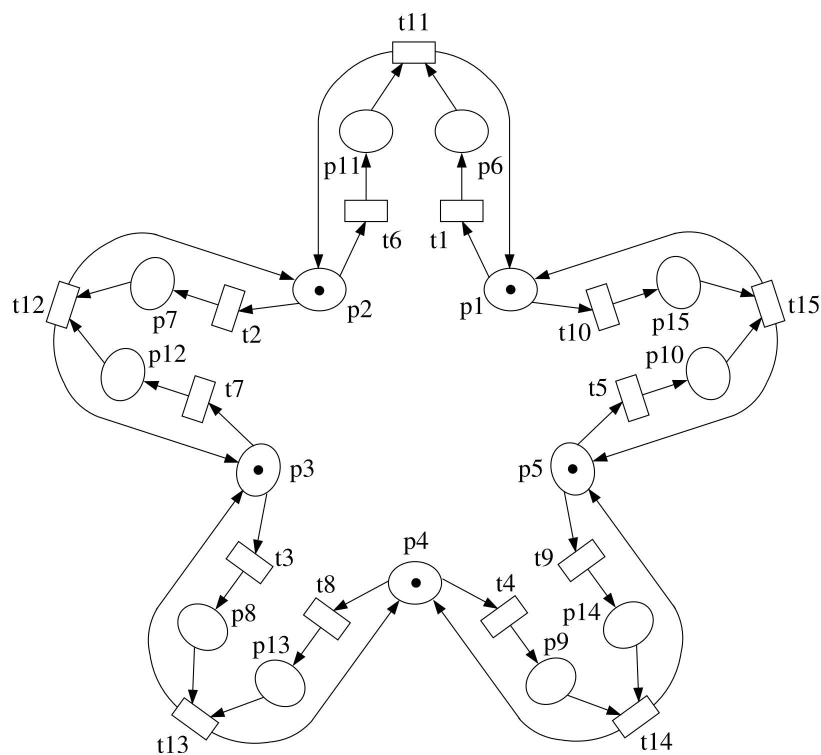




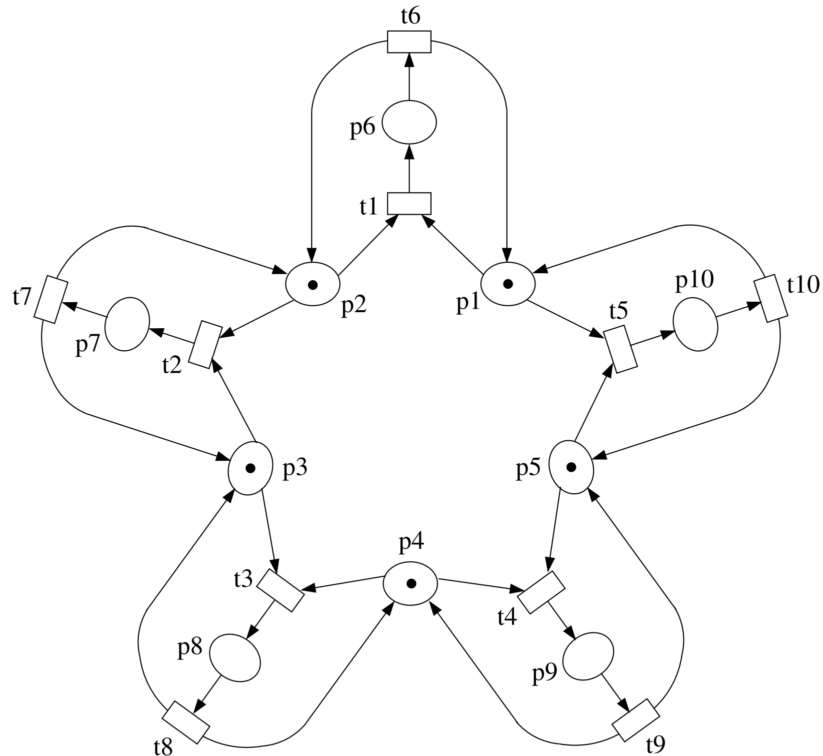
Model problemu 5 filozofów (wersja 1)



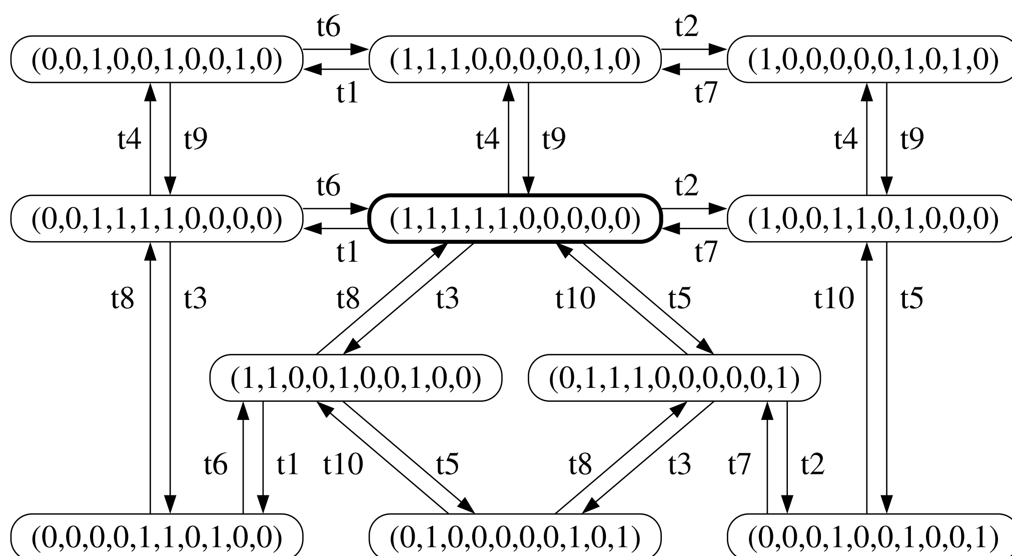
Model problemu 5 filozofów (wersja 2)



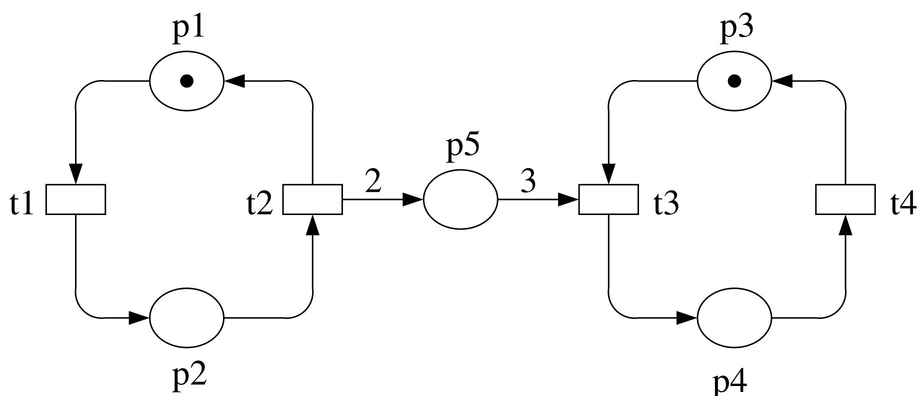
Model problemu 5 filozofów (wersja 3)



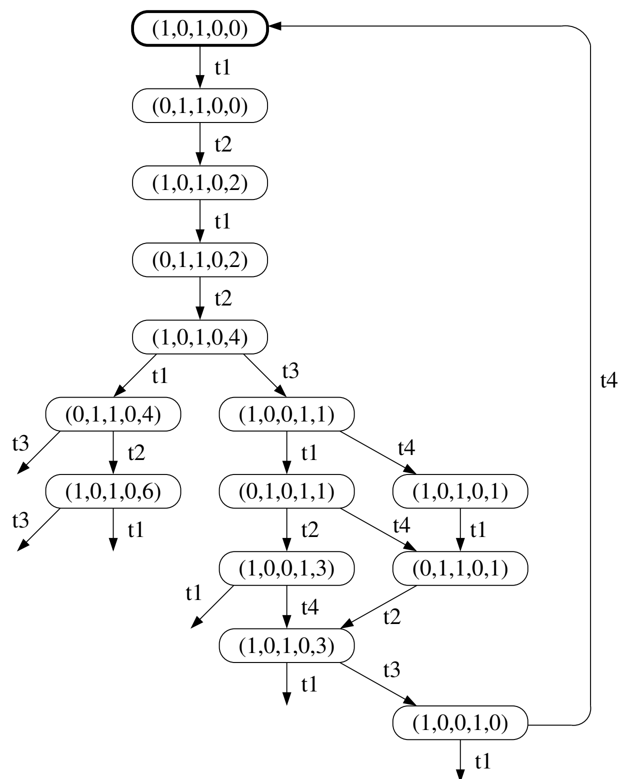
Graf osiągalności – problemu 5 filozofów (wersja 3)



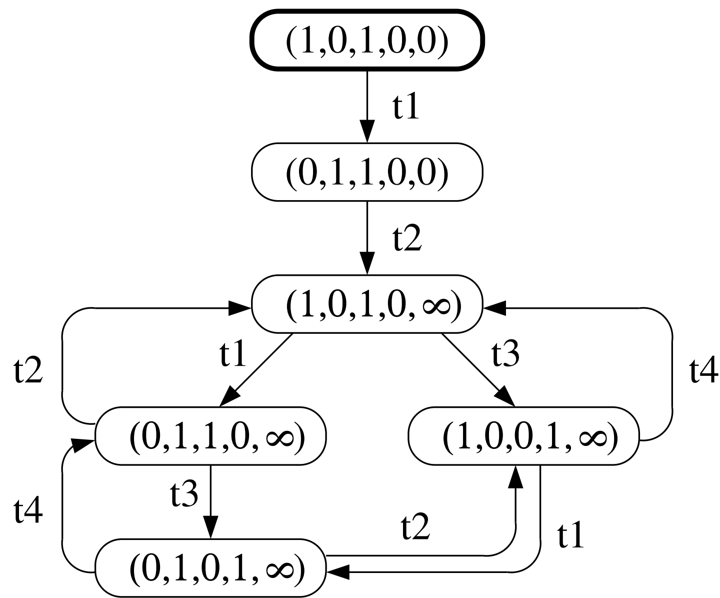
Model systemu *producent – konsument*



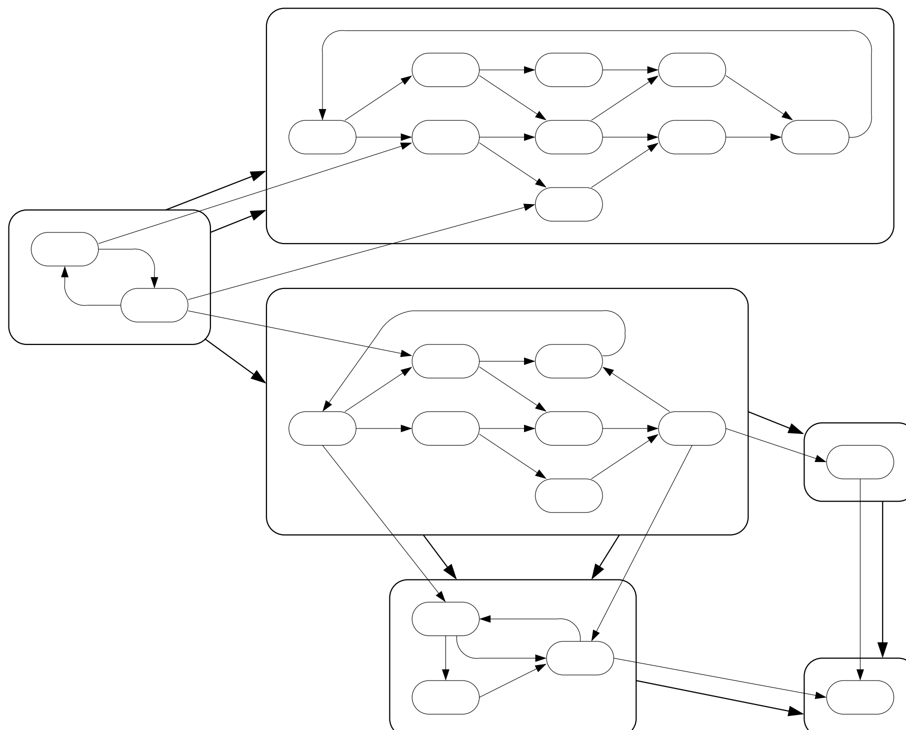
Fragment grafu osiągalności – *producent – konsument*



Graf pokrycia – *producent – konsument*



Graf silnie spójnych składowych



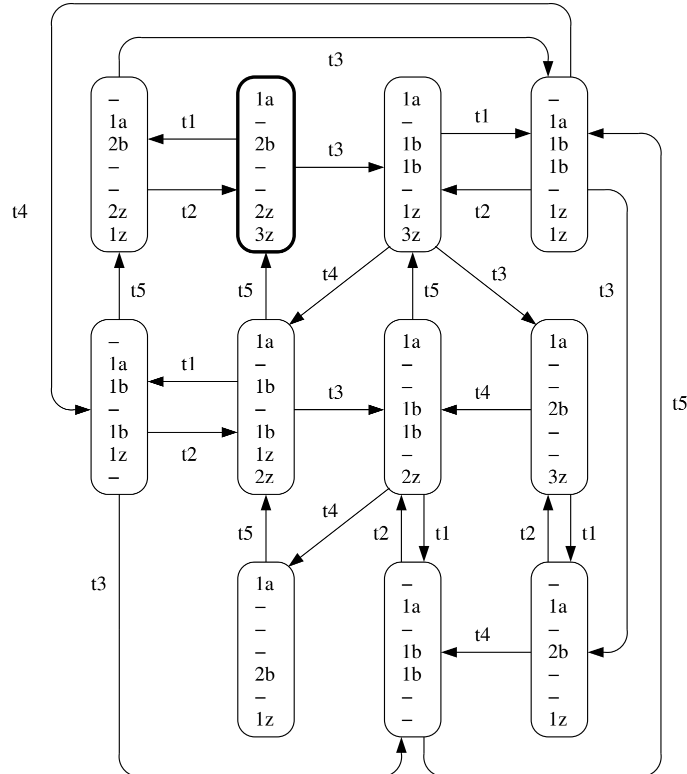
Sieci Petriego – sieci kolorowane

```
color P = with p | q;
color I = int with 1..20;
color E = with e;
color U = product P * I  timed;
var x : P;
var i : I
```

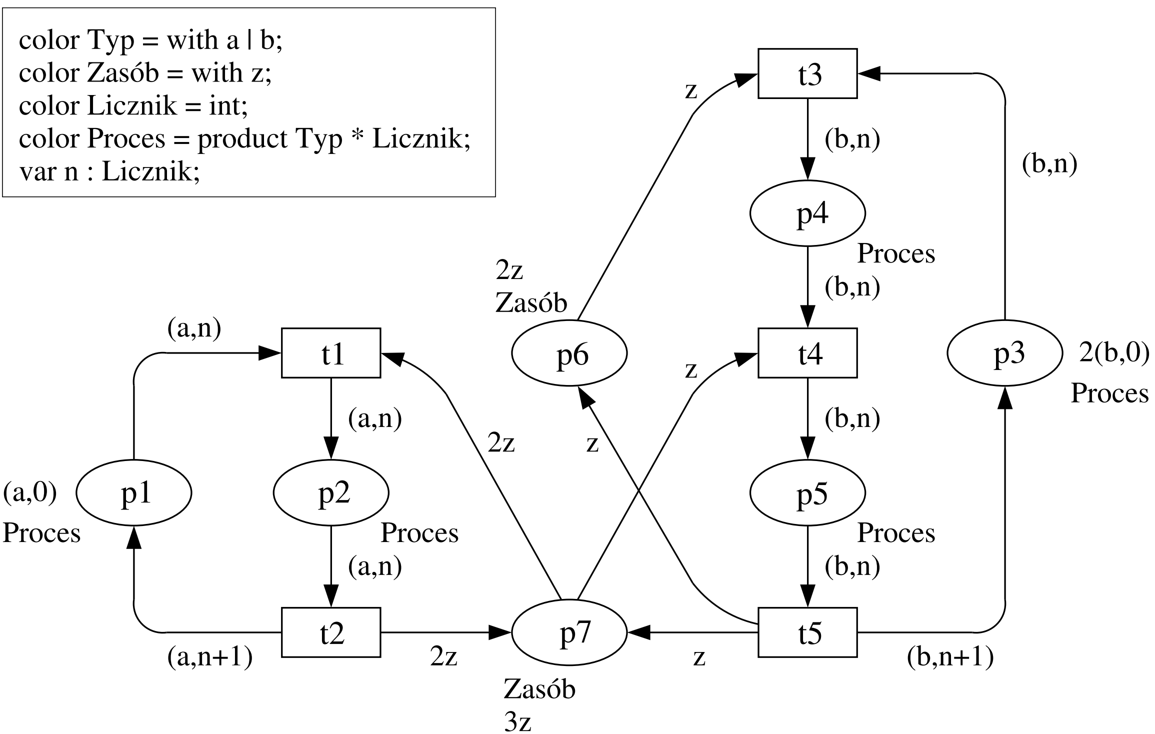

Model współzawodnictwa procesów (wersja 1)

```
color Proces = with a | b;
color Zasób = with z;
```

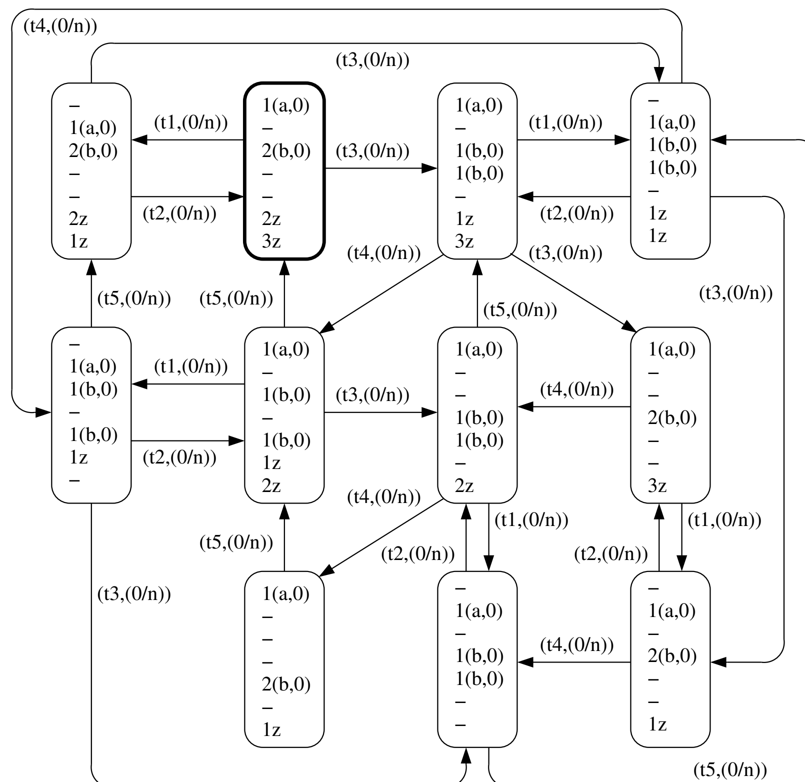
Graf osiągalności dla modelu współzawodnictwa procesów (wersja 1)



Model współzawodnictwa procesów (wersja 2)



Graf relacyjny dla modelu współzawodnictwa procesów (wersja 2)



Algebra procesów CCS

Podstawowym pojęciem algebry CCS jest agent rozumiany jako dowolna część modelowanego systemu, która ma swoją własną tożsamość trwającą w czasie. Agent ma zdefiniowany interfejs do komunikacji z otoczeniem, tj. zbiór portów komunikacyjnych. Para portów o komplementarnych nazwach tworzy kanał komunikacyjny poprzez który dwa agenty mogą się komunikować. Pojedyncze porty mogą być również użyte do komunikacji z otoczeniem modelowanego systemu. Rachunek CCS dostarcza również szereg operatorów, które pozwalają łączyć ze sobą agenty budując w tej sposób agenty (procesy) o bardziej złożonym zachowaniu i możliwej komunikacji wewnętrznej.

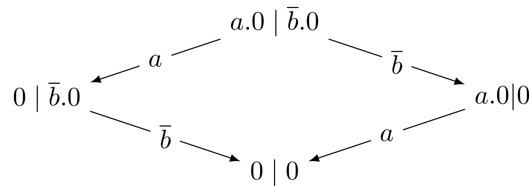
- Prefiks (kropka) określa kolejność wykonywania akcji, np. $\alpha.A$ – najpierw akcja α, a później zachowanie zgodne z definicją agenta A.
- Sumy $+$ pozwala na wybranie jednej z kilku możliwych akcji, np. $\alpha.A + \beta.B$ najpierw akcja α, a później zgodnie z definicją agenta A lub najpierw akcja β, a później zgodnie z definicją B.
- Złożenie równoległego $|$ pozwala łączyć ze sobą agenty. Połączone agenty mogą komunikować się ze sobą, jeżeli istnieją pary portów komplementarnych.
- Obcięcia $\backslash$ pozwala ukryć wskazane porty przed otoczeniem. Obcięte porty mogą być używane wyłącznie do komunikacji wewnętrznej między agentami.
- Przeetykietowanie [ ] służy do zmiany nazw portów agentów. Stosuje się go w celu uzyskania pary portów komplementarnych, lub wyeliminowania takiej pary.

Graf pochodnych – przykłady (1)

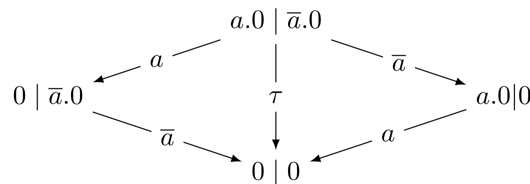
$a.\tau.\overline{b}.0$ $\quad a.\tau.\overline{b}.0 \xrightarrow{a} \tau.\overline{b}.0 \xrightarrow{\tau} \overline{b}.0 \xrightarrow{\overline{b}} 0$

$a.0 + b.0$ $\quad 0 \xleftarrow{a} a.0 + b.0 \xrightarrow{b} 0$

$a.0 \mid \overline{b}.0$

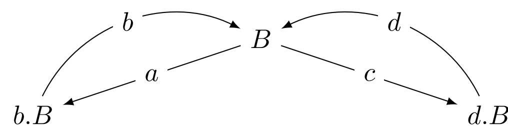


$a.0 \mid \overline{a}.0$



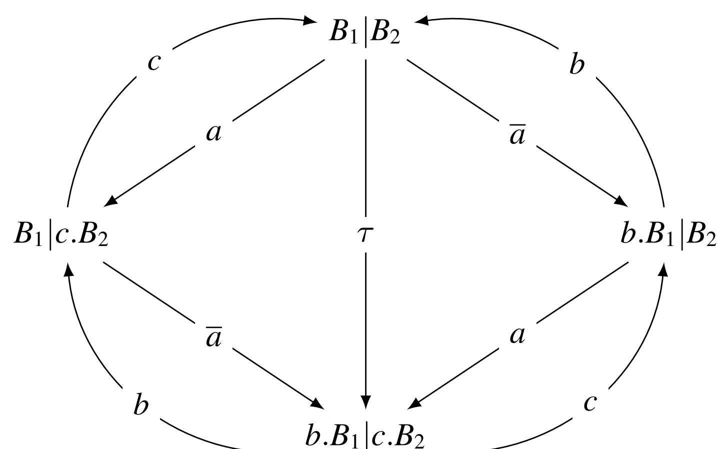
$(a.0 \mid \overline{a}.0)\backslash\{a\}$ $\quad (a.0 \mid \overline{a}.0)\backslash\{a\} \xrightarrow{\tau} (0 \mid 0)\backslash\{a\}$

$B = a.b.B + c.d.B$



Graf pochodnych – przykłady (2)

$$B_1 = \overline{a}.b.B \qquad B_2 = a.c.B_2$$



Model *producent – konsument*

Bardzo często systemy współbieżne przedstawiane są jako obcięte złożenie przeetykietowanych komponentów, tzn. w postaci $(E_1[f_1]|\ldots|E_n[f_n])\backslash L$.
Postać tą nazywamy standardową formą współbieżną.

```
agent Producer = produce.'put.Producer;
agent Consumer = get.consume.Consumer;

agent Buffer = put.Buffer1;
agent Buffer1 = put.Buffer2 + 'get.Buffer;
agent Buffer2 = 'get.Buffer1;

set S = {put, get};
agent System = (Producer | Buffer | Consumer)\S;
```

CWB

```
input "prodcon.ccs";
print;
graph System;
sim System;
size System;
quit;
```


Model *czytelników i pisarzy*

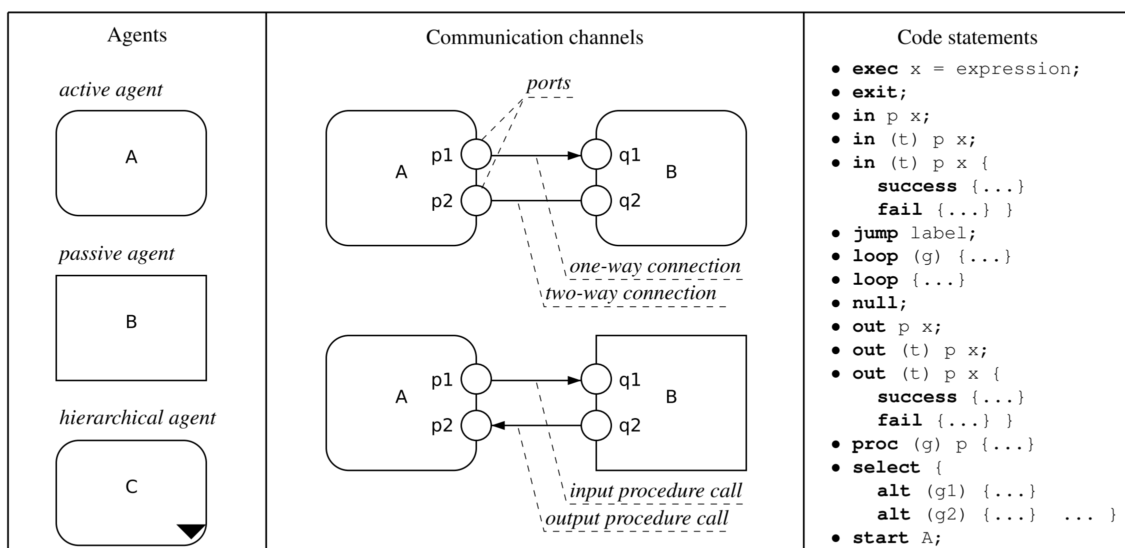
```
agent Reader1 = 'in.read.out.Reader1;
agent Reader2 = 'in.read.out.Reader2;
agent Reader3 = 'in.read.out.Reader3;

agent Writer1 = 'lock.write.unlock.Writer1;
agent Writer2 = 'lock.write.unlock.Writer2;

agent Reading = in.In1;
agent In1 = 'out.Library + in.In2;
agent In2 = 'out.In1 + in.In3;
agent In3 = 'out.In2;

agent Writing = lock.'unlock.Library;

agent Library = Reading + Writing;

set S = {in, out, lock, unlock};

agent RW1 = (Library | Reader1 | Reader2 |
             Reader3 | Writer1 | Writer2)\S;
```

Model 5 *filozofów*

```
agent Fork = up.down.Fork;
agent Phil = 'lup.'rup.think.'ldown.'rdown.Phil;

agent Phil1 = Phil['u12/'lup,'u51/'rup,'d12/'ldown,'d51/'rdown];
agent Phil2 = Phil['u23/'lup,'u12/'rup,'d23/'ldown,'d12/'rdown];
agent Phil3 = Phil['u34/'lup,'u23/'rup,'d34/'ldown,'d23/'rdown];
agent Phil4 = Phil['u45/'lup,'u34/'rup,'d45/'ldown,'d34/'rdown];
agent Phil5 = Phil['u51/'lup,'u45/'rup,'d51/'ldown,'d45/'rdown];

agent Fork1 = Fork[u12/up,d12/down];
agent Fork2 = Fork[u23/up,d23/down];
agent Fork3 = Fork[u34/up,d34/down];
agent Fork4 = Fork[u45/up,d45/down];
agent Fork5 = Fork[u51/up,d51/down];

set Res = {u12,u23,u34,u45,u51,d12,d23,d34,d45,d51};

agent S = (Phil1|Phil2|Phil3|Phil4|Phil5|
           Fork1|Fork2|Fork3|Fork4|Fork5)\Res;
```


Alvis – składnia

Alvis = **AL**gebra + **VIS**ual
Despite of preliminary plans Alvis **is not** a process algebra!



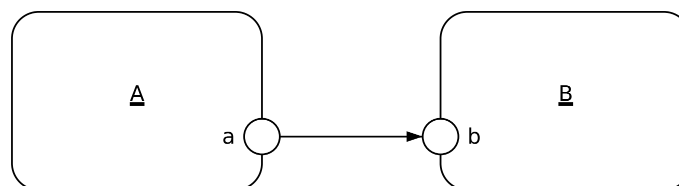
Cele projektu Alvis

- Łatwy w modelowaniu język formalny („dla inżynierów”).
- Wizualne modelowanie struktury połączeń komunikacyjnych z systemie.
- Język wysokiego poziomu do definiowania dynamiki komponentów systemu.
- Możliwość weryfikacji z użyciem technik weryfikacji modelowej.
- Wygodne modelowania systemów: współbieżnych, czasu rzeczywistego, wbudowanych i rozproszonych.

Prace dyplomowe

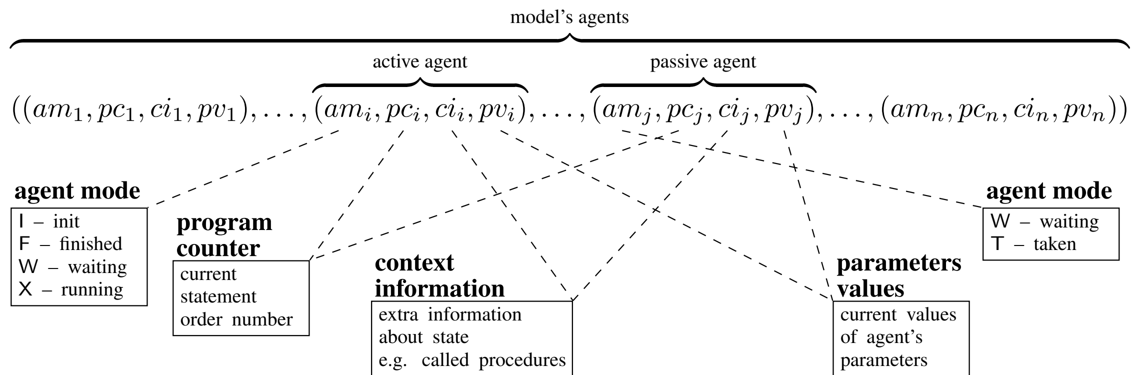
Sporo zrobiliśmy, ale horyzont „ciągle ucieka” ;). Kolejne kroki w tej „podróży” możecie zrobić Wy w ramach prac dyplomowych. Zapraszamy!

Alvis – przykład komunikacja między agentami

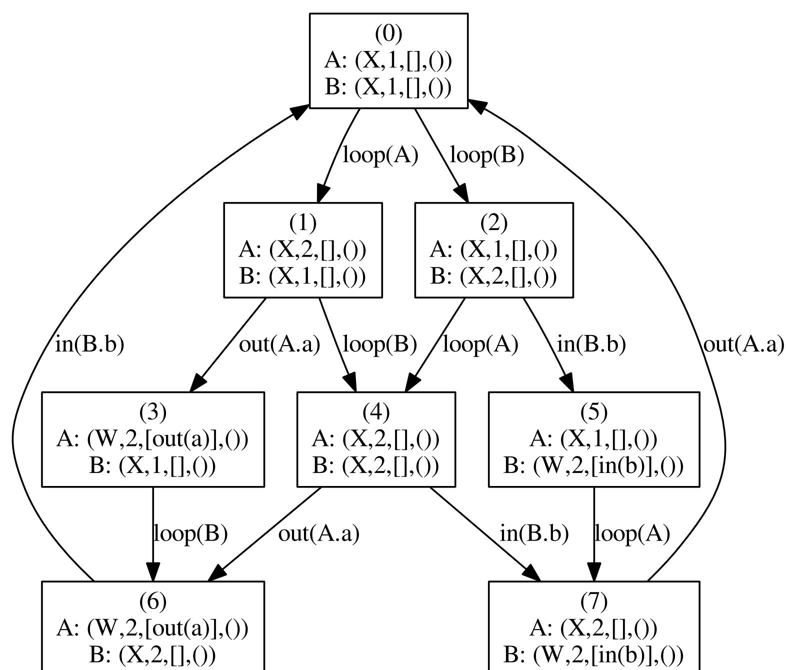



```
agent A {
  loop {       -- 1
    out a;     -- 2
  }
}

agent B {
  loop {       -- 1
    in b;      -- 2
  }
}
```

Stan modelu



Alvis – graf LTS



Workflow