Opracowanie nt. algebr procesów
Metody formalne - Kolokwium 2

Spis treści

1 Zagadnienia od Szymkata

1. porównanie LOTOS-a z CCS (wykład)
2. porównanie CCS z CSP (wykład)
3. analogiczne pojęcia w CCS, CSP i LOTOS-ie (wykład)
4. LOTOS: etykietowany graf przejść (LTS) dla zdefiniowanej specyfikacji
5. LOTOS: złożenie równoległe procesów (współbieżne) z synchronizacją pełną, częściową lub bez synchronizacji, wybór procesu aktywnego z wielu procesów możliwych, ukrywanie jednego procesu w drugim, rekursywna (nieskończona) iteracja procesu
6. LOTOS: przekazywanie/uzgadnianie danych miedzy procesami za pośrednictwem określonych bram (akcji synchronizujących), postaci ofert komunikacyjnych
7. redukcja etykietowanych systemów przejść (LTS) w pakiecie CADP, różnego typu równoważności bisymulacyjne grafów LTS
8. LOTOS: akcje wewnętrzne (nieobserwowalne), proces bezczynności, zakończenie procesu, złożenie sekwencyjne, wyzwalanie (aktywowanie) i blokowanie (przerwanie) procesu, ukrywanie procesu, zmianę nazwy procesu (reetykietowanie), wykonanie jednego procesu pod warunkiem wykonania drugiego procesu
9. LOTOS: ogólna składnia specyfikacji behawioralnej (podstawowe słowa kluczowe i format definicji poszczególnych obiektów)

2 Źródła

Opracowanie na podstawie:

- Szmuc, Szpyrka - Metody formalne w inżynierii oprogramowania systemów czasu rzeczywistego
- Wykłady Szmuca
- L. Logrippo, M. Faci, M. Haj-Hussein - An Introduction to LOTOS: Learning by Examples
- Kenneth J. Turner - The Formal Specification Language LOTOS A Course for Users
- Calculi an Automata for Modelling Untimed and Timed Concurrent Systems - Bowman, Howard, Gomez, Rodolfo

3 Pojęcia wstępne

System

Szerokie pojęcie, np. oprogramowanie, organizacja, firma, osoba. W kontekście algebr procesów przez system rozumiane są jednak tylko zachowania (oprogramowania, organizacji, itd.) oraz informacje dotyczące tych zachowań (kolejność, czas wykonania, priorytety, wzajemne zależności). System składa się z procesów.

Proces

Proces to pewien logicznie spójny aspekt zachowania systemu, podzbiór zachowań systemu.

Algebry procesów

Algebry procesów należą do metod formalnych, służą do opisu systemów współbieżnych, opierają się na rachunku algebraicznym.

Communicating Sequential Processes (CSP)

Pierwsza z algebr procesów (1978), opiera się na modelu komunikacji międzyprocesowej opartym o przekazywanie wiadomości (message passing). Umożliwia modelowanie komunikacji synchronicznej.

Calculus of Communicating Systems (CCS)

Kolejna z algebr procesów (1980) z innymi definicjami i twierdzeniami. W tej algebrze dużą rolę odgrywa możliwość dowodzenia równoważności pomiędzy modelami.

Język LOTOS

Kolejna algebra procesów, która powstała na bazie CSP i CCS. LOTOS został stworzony (1989) przez ISO w celu standaryzacji specyfikacji usług i protokołów sieciowych.

4 Zagadnienie 3

Zagadnienie 3

Analogiczne pojęcia w CCS, CSP i LOTOS-ie (wykład)

Algebry procesów korzystają z różnych nazw dla tych samych pojęć. Zestawienie w tabeli poniżej.

CCS	CSP	LOTOS
akcje: $a_1, a_2, ...$	zdarzenia: $e_1, e_2, ...$	akcje lub bramy $g_1, g_2, ...$
agenci: $A_1, A_2, ...$	procesy: $P_1, P_2, ...$	wyrażenia behavioralne: $B_1, B_2, ...$
agent 0	proces: STOP	proces: STOP
rodzaj syntaktyczny: $\mathcal{L}(A)$	alfabet: αP	zdarzenia lub akcje
porty: $p_1, p_2, ...$	kanały: $c_1, c_2, ...$	bramy: $g_1, g_2, ...$
etykiety	symbole	bramy (*gates*)
zbiór wartości	dziedzina	sorts: $t_1, t_2, ...$

Rysunek 1: Różne nazwy dla tych samych pojęć w każdej z algebr.

Chcąc wyrazić proces w algebrze procesów posługujemy się akcjami i operatorami, które mają go opisywać. Każda algebra definiuje swoje nazewnictwo jak widać w tabeli powyżej, jednak poszczególne pojęcia znaczą zazwyczaj to samo w każdej z algebr:

Akcje
Odpowiadają niepodzielnym (atomowym) czynnościom wykonywanym przez proces (w CCS akcje, w CSP zdarzenia, w LOTOS akcje lub bramy). Akcje oznacza się małymi literami.

Operatory
Właściwe dla danej algebry operacje algebraiczne

Wyrażenie algebraiczne
Połączenie akcji i operatorów tworzy wyrażenie algebraiczne.

Proces
Wyrażenia algebraiczne po spełnieniu odpowiednich warunków syntaktycznych nazywane są procesami. Procesy oznacza się dużymi literami. (w CCS agenci, w CSP procesy, w LOTOS wyrażenie behawioralne)

Alfabet procesu (rodzaj procesu) Danemu procesowi przyporządkowane są akcje, które może on wykonywać. Pełny zbiór akcji, które może wykonać dany proces nazywany jest alfabetem tego procesu (w CSP, natomiast w CCS rodzajem, a w LOTOS to po prostu zdarzenia lub akcje).

Komunikacja pomiędzy procesami Istnieje możliwość komunikacji pomiędzy procesami (w CCS przez porty, w CSP przez kanały, w LOTOS przez bramy).

5 Zagadnienia 1 i 2

Zagadnienie 1

Porównanie LOTOS-a z CCS (wykład)

Najważniejsze to:

1. Sposób synchronizacji procesów - LOTOS używa multi-synchronizacji, a CSS używa synchronizacji poprzez akcje komplementarne
2. Działania operatora ukrywania (w LOTOS) i jego odpowiednika Restriction w CSS jest inne - w LOTOS operator ten generuje akcje wewnętrzne, w CCS usuwa akcje
3. Akcja wewnętrzna w CCS (τ), oprócz tego, że wprowadza niedeterminizm jak w LOTOS, jest także używana do synchronizacji
4. Poza tym wiele operatorów z CCS ma prawie identyczne znaczenie jak w LOTOS
5. w LOTOS istnieje więcej operatorów niż w CCS

Zagadnienie 2

Porównanie CCS z CSP (wykład)

1. (Najważniejsze) Inne podejście do problemu równoważności procesów
   - W CSP problem równoważności jest definiowany z użyciem pojęcia śladu lub przy użyciu modelu niepowodzeń
   - W CCS równoważność procesów jest zdefiniowana w kontekście relacji bisymulacji

   Bisymulacja jest semantyką równoważności procesów, która bardziej precyzyjnie pozwala porównywać procesy.
2. Inne podejście do synchronizacji procesów, w CCS synchronizacja z użyciem akcji komplementarnych, w CSP synchronizacja bliższa tej z LOTOS-a
3. Zostały pomyślane jako narzędzia do innych celów
   - CSP została pomyślana jako narzędzie do modelowania i analizy systemów współbieżnych.
   - CCS została pomyślana jako narzędzie do analizy obserwowanego z zewnątrz systemu oraz dowodzenia równoważności między modelami
4. Inna terminologia dla tych samych pojęć
   - Przykładowo w CSP atomowe czynności wykonywane przez proces to akcje
   - Atomowe czynności wykonywane przez proces w CCS nazywane są zdarzeniami
5. Inne operatory w każdej z algebr
   - W CSP istnieją inne operatory, których nie ma w CCS, a także w CCS istnieją operatory, których nie ma w CSP.

Zagadnienie Bonus

Porównanie LOTOS-a z CSP

Najważniejsze to

1. Operator wyboru - CSP posiada wiele operatorów wyboru, które stosowane są zależnie od sytuacji, LOTOS ma jeden taki operator
2. Akcje wewnętrzne - w CSP nie można się jawnie odwołać do akcji wewnętrznej, a w LOTOS można

6 Zagadnienie 4

Zagadnienie 4

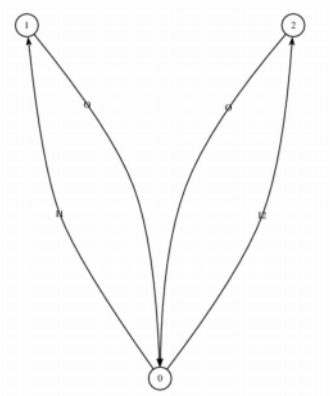
LOTOS: etykietowany graf przejść (LTS) dla zdefiniowanej specyfikacji

Chyba chodzi o to żeby narysować LTS-a dla zadanego programu w LOTOS. Przykłady:

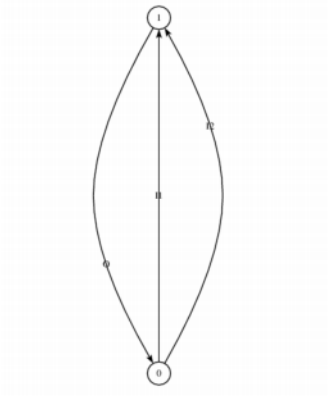
Przykład

```
specification XOR[I1,I2,O]: noexit
behaviour
  XOR[I1,I2,O]
where
  process XOR[I1,I2,O]: noexit :=
    I1; O; XOR[I1,I2,O]
    []
    I2; O; XOR[I1,I2,O]
  endproc
endspec
```

Etykietowany graf przejść (LTS) dla powyższej specyfikacji wygląda następująco



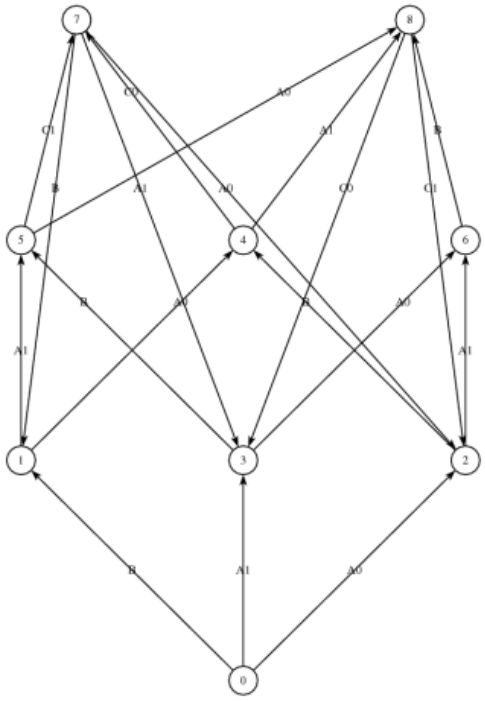
a) graf pełny



b) graf po redukcji rozgałęzieniowej

Przykład

```
specification sequencer[a0,a1,b,c0,c1]: noexit
behaviour
   (S1[a0,c0] ||| S2[a1,c1]) |[c0,c1]| S3[b,c0,c1]
where
  process S1[a0,c0]: noexit :=
    a0; c0; S1[a0,c0]
  endproc
  process S2[a1,c1]: noexit :=
    a1; c1; S2[a1,c1]
  endproc
  process S3[b,c0,c1]: noexit :=
    b; (c0; S3[b,c0,c1] [] c1; S3[b,c0,c1])
  endproc
endspec
```



Przypomnienie definicji:
LTS (Labelled Transition System) to krotka LTS = (S, Act, $\rightarrow$, s_0), gdzie:

- S to skończony zbiór stanów
- Act to skończony zbiór akcji
- $\rightarrow$ to przejścia pomiędzy stanami
- $s_0 \in S$ to stan początkowy

7 Zagadnienia 5 i 8

Zagadnienie 5

LOTOS: złożenie równoległe procesów (współbieżne) z synchronizacją pełną, częściową lub bez synchronizacji, wybór procesu aktywnego z wielu procesów możliwych, ukrywanie jednego procesu w drugim, rekursywna (nieskończona) iteracja procesu

Zagadnienie 8

LOTOS: akcje wewnętrzne (nieobserwowalne), proces bezczynności, zakończenie procesu, złożenie sekwencyjne, wyzwalanie (aktywowanie) i blokowanie (przerwanie) procesu, ukrywanie procesu, zmiana nazwy procesu (reetykietowanie), wykonanie jednego procesu pod warunkiem wykonania drugiego procesu

Czym się różnią: złożenie sekwencyjne, wyzwalanie (aktywowanie) i wykonanie jednego procesu pod warunkiem wykonania drugiego procesu - tego nie wiem.

Przypominamy, że w LOTOS używamy pojęć:

LOTOS
akcje lub bramy $g_1, g_2, ...$
wyrażenia behavioralne: $B_1, B_2, ...$
proces: STOP
zdarzenia lub akcje
bramy: $g_1, g_2, ...$
bramy (*gates*)
sorts: $t_1, t_2, ...$

Rysunek 2: Nazwy stosowane w LOTOS.

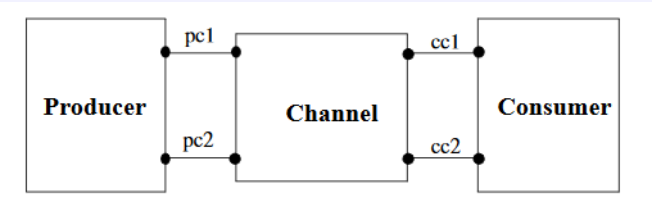
7.1. Operatory w LOTOS

Operatory będą przedstawione na przykładzie problemu Producent-Konsument(P-K).

7.1.1. Proces bezczynności stop i zakończenia exit

W LOTOS istnieją dwa predefiniowane wyrażenie behawioralne - stop i exit.
stop - reprezentuje proces zupełnie nieaktywny, oznacza deadlock lub brak możliwości wykonania akcji
exit - oznacza normalne (pozytywne) zakończenie procesu

Przykład Producent-Konsument



Rysunek 3: Problem producent-konsument.

- Producent i konsument komunikują się przez kanał.
- Kanał działa tak jak FIFO, nie może zmieniać kolejności wiadomości ani ich tracić. Kanał synchronizuje się z konsumentem dopiero gdy zakończy synchronizacje z producentem.
- Producent musi utworzyć dwa elementy, a później kończy działanie
- Konsument musi odebrać oba elementy i kończy działanie

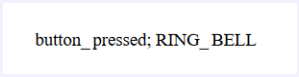
W przykładzie łatwo jest wyróżnić trzy procesy (producent, kanał, konsument).

7.1.2. Operator prefix ; (operator następstwa sekwencyjnego)

Operator prefix, czyli średnik **;** służy do sekwencyjnego złożenia akcji i wyrażenia behawioralnego.

Przykład

$$a; B$$



Rysunek 4: Po naciśnięciu guzika, dzwoni dzwonek.

To wyrażenie behawioralne, które oznacza, że po wykonaniu akcji a proces przechodzi w stan B (reprezentowany przez wyrażenie behawioralne B).

Producent jako proces może być widziany jak czarna skrzynka komunikująca się ze światem przez port pc1(tworzy jeden element) oraz pc2(tworzy drugi element). Proces producenta można więc zapisać jako:

```
process Producer [pc1, pc2] : exit :=
    pc1;   pc2;   exit
endproc
```

Właściwe bramy z których korzysta proces są jego parametrami i muszą być określone na etapie tworzenia instancji procesu. (podrozdział psanie programów z użyciem operatorów).
Analogicznie można opisać proces konsumenta i kanału:

```
process Consumer [cc1, cc2] : exit :=
    cc1;   cc2;   exit
endproc
```

```
process Channel [pc1, pc2, cc1, cc2] : exit :=
    pc1; pc2; cc1; cc2; exit
endproc
```

Taki opis procesów jest dobry jeśli analizujemy każdy proces oddzielnie, jednak aby złożyć trzy procesy razem potrzebne będzie jeszcze użycie operatorów składania równoległego (parallel composition operators), które będą omówione później.

Zauważmy w tym przykładzie, że wszystkie wykonywane akcje są obserwowalne (zewnętrzne). Przykładowo wykonanie akcji pc1 przez producenta powoduje również wykonanie w tym samym momencie wykonanie tej samej akcji w procesie Kanału. Analogicznie dla pozostałych akcji.

Warto jeszcze zwrócić na notację

process ***Producer*** [pc1, pc2] : **exit** :=

Konkretnie na końcówkę ": **exit**".
Taka końcówka oznacza, że proces jest w stanie wykonać zachowanie **exit** na końcu działania. Końcówka ta może być ustawiona na jedną z dwóch wartości: *exit* jeśli proces może pomyślnie zakończyć działanie albo wartość *noexit* w innym wypadku.

7.1.3. Operator wyboru [] (choice operator)

Aby zilustrować działanie operatora wyboru modyfikujemy rozważany problem producent-konsument w ten sposób, że:

- Kanał może dostarczyć konsumentowi pierwszy element od producenta, zanim producent wyprodukuje drugi.

Aby zamodelować to zachowanie potrzebny jest nam operator wyboru, który jest oznaczany **[]**. Operator ten oznacza wybór pomiędzy dwoma lub większą ilością zachowań.
W przypadku problemu P-K Kanał musi najpierw się synchronizować z Producentem na bramce pc1, a później ma dwie możliwości:

1. Może się synchronizować z Producentem na bramce pc2
2. Może się synchronizować z Konsumentem na bramce cc1

Wybór jednego z zachowań eliminuje możliwość wykonania drugiego. Można więc zapisać:

```
process Channel [ pc1, pc2, cc1, cc2] : exit  :=
    pc1;
    (
        pc2;    cc1;    cc2;    exit
    []
        cc1;    pc2;    cc2;    exit
    )
endproc
```

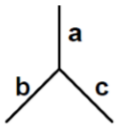
Operator wyboru umożliwia modelowanie niedeterministycznych zachowań, np. tak jak poniżej dla automatu sprzedającego kawę i mleko w linii numer 2.

```
insert_quarter;  get_coffee; stop []  insert_dime;  get_milk; stop     (* 1 *)

insert_quarter; get_coffee; stop [] insert_quarter; get_milk; stop     (* 2 *)

insert_quarter; (get_coffee; stop [] get_milk; stop)                   (* 3 *)
```

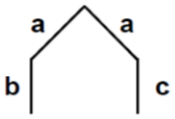
Zwróćmy uwagę na ważną różnicę pomiędzy 2 i 3. W przypadku drugim po wrzuceniu monety (insert_quarter) nie mamy już wyboru (środowisko procesu nie ma wyboru) czy chcemy kawę czy mleko, bo automat niedeterministycznie sam wybiera kawę lub mleko. W przykładzie 3 po wrzuceniu monety możemy wybrać (środowisko procesu może wybrać) czy chcemy kawę czy mleko.

a ; **(B$_1$ [] B$_2$)** and a ; **B$_1$ []** a ; **B$_2$** model different behaviours

process one [a,b,c]
a; (b; **stop**
[]
c; **stop**)
endproc



process two [a,b,c]
a; b; **stop**
[]
a; c; **stop**
endproc



The choice of 'a' is not determined

Examples of nondeterministic behaviours:

a; B$_1$ [] a; B$_2$

The internal action **i** is also a source of nondeterminism

i; B$_1$ [] i; B$_2$

a; B$_1$ [] i; B$_2$

Jaka jest różnica pomiędzy dwoma rysunkami ?
Chodzi o to, że środowisko ma wybór albo go nie ma (tak jak przy kupowaniu kawy lub mleka w powyższym przykładzie).

Na Rysunku z prawej środowisko może wybrać a, ale nie ma już wyboru co będzie następne (niedeterminizm). Na rysunku z lewej środowisko może wybrać a, a później może jeszcze wybrać czy chce b czy c (po wrzuceniu monety do automatu możemy jeszcze wybrać czy chcemy kawę czy mleko).

7.1.4. Operator aktywowania » (enable)

Operator aktywowania jest podobny do **;**, ale służy do sekwencyjnego łączenia dwóch wyrażeń behawioralnych (a nie akcji i wyrażenia behawioralnego jak w przypadku operatora **;**).

for example, if process SHOP is:

visit_ shop; buy_ food; come_ home; **exit**

cook_ food; eat_ food; **stop**

then the process DINE, defined by:

SHOP >> EAT

denotes the possible sequence of events:

visit_ shop; buy_ food; come_ home; cook_ food; eat_ food

- **exit** (i.e. the *¡* event) is absorbed by a following '>>'
- if the left-hand behaviour expression does not terminate successfully, the right-hand behaviour expression will not apply; for example:

(prime_ 5; prime_ 7; (prime_ 9; **stop** [] **exit**)) >> (prime_ 11; **exit**)

can successfully terminate after the sequence of events:

*prime_ 5; prime_ 7; prime_* 11

but will deadlock after the sequence of events:

prime_ 5; prime_ 7; prime_ 9

- '>>' is associative, but is not of course commutative

Wracając do przykładu problemu P-K, moglibyśmy zapisać, że:

```
process Channel [ pc1, pc2, cc1, cc2] : exit  :=
    pc1;
    (
        pc2;    cc1;    exit
        []
        cc1;    pc2;    exit
    )
    >>      cc2;    exit
endproc
```

7.1.5. Zdarzenie (akcje) wewnętrzne *i*

W LOTOS wyrażenie behawioralne może wykonywać dwa rodzaje akcji.

- Akcje nieobserwowalne (wewnętrzne) - to akcje, które dany proces może wykonywać niezależnie od innych procesów lub jakichś innych zdarzeń, w LOTOS są oznaczane przez **i**.
- Akcje obserwowalne (zewnętrzne) - to akcje, które wymagają synchronizacji ze środowiskiem działania procesu, te akcje mogą być wykonywane w punktach synchronizacji, czyli bramkach (gates).

Znów dla celów pokazania działania zdarzenia wewnętrznego modyfikujemy rozważany problem P-K.

- Teraz chcemy zamodelować sytuację w której kanał może samoistnie stracić pierwszy oraz drugi element (np. jakiś błąd wewnętrzny). Zauważmy, że taka akcja nie jest zależna od żadnego innego procesu, ale jest akcją wewnątrz kanału (nieobserwowalną).

Do tej pory w procesie używaliśmy tylko akcji obserwowalnych, przykładowo pc1, oznaczało w procesie producenta wyprodukowanie pierwszego elementu, a w procesie kanału odebranie pierwszego elementu.

Teraz chcemy dodać do procesu kanału akcję, która dzieje się tylko w tym procesie (wewnętrzną). Akcję wewnętrzną w procesie oznaczamy przez **i**.
Chcemy więc mieć sytuację opisaną nieformalnie jako:

```
process Channel [ . . . ] : exit  :=
    ( Get the first element;
        (   Get the second element ;      Put the first element;      exit    (* 1 *)
            []
            Get the second element;       Lose the first element;     exit    (* 2 *)
            []
            Put the first element;        Get the second element;     exit    (* 3 *)
            []
            Lose the first element;       Get the second element;     exit     (* 4  *)
        )
    )
    >>     ( Put the second element; exit  [] Lose the second element; exit )
endproc
```

Zauważamy, że 2 i 4 są równoważne, bo występuje w nich akcja wewnętrzna, co prowadzi do formalnej implementacji:

```
process Channel [pc1, pc2, cc1, cc2] : exit  :=
    pc1;  ( pc2 ;  cc1;  exit  [] cc1;  pc2;  exit     []    i;   pc2;  exit   )
          >> ( cc2;  exit   []   i;   exit)
endproc
```

Użycie akcji wewnętrznej wprowadza niedeterminizm, ponieważ nie wiadomo czy wykona się synchronizacja pc2, cc1 czy też zdarzenie wewnętrzne i. Gdyby nie było w wyborze trzeciego przypadku z i, to mielibyśmy determinizm, bo środowisko mogłoby wybrać pomiędzy pc2 lub cc1.

Oczywiście trzeba teraz dostosować działanie konsumenta:

```
process Consumer [ cc1, cc2] : exit  :=
    cc1; (cc2; exit [] exit)
    []
    cc2;    exit
    []
    exit
endproc
```

7.1.6. Kilka ciekawszych przykładów z wydarzeniami wewnętrznymi

Przykład

coffe; exit [] milk; exit

To proces, który może synchronizować się zarówno na akcji coffe jak i milk.

Przykład

i; coffe; exit [] milk; exit

To proces, który może nie być w stanie synchronizować się na milk, ale zawsze będzie w stanie synchronizować się na coffe.

- Jeśli środowisko proponuje milk, a w procesie zaszło już wydarzenie i, to nie może się synchronizować przez milk
- Jeśli środowisko proponuje coffe, a w procesie nie zaszło jeszcze wydarzenie i, to można założyć, że wydarzenie i kiedyś zajdzie.

Przykład

i; coffe; exit [] i; milk; exit

Przez analogię do poprzedniego przypadku, to proces, który może nie być w stanie synchronizować się na coffee lub na milk.

7.1.7. Operator przerwania, deaktywowania (disable) [>

- a frequent occurrence in specifications is the need to specify behaviour which may be interrupted by something else (e.g. disconnection may terminate data transfer)
- the '[>' operator (pronounced *disabled by*) allows the right-hand behaviour expression to interrupt the left-hand behaviour expression; if this happens, the future behaviour is that of the right-hand behaviour expression only
- if the left-hand behaviour expression terminates successfully, then the combination also terminates successfully (i.e. disabling is no longer possible); this is to allow for contingencies

  for example:

  (send_data; reset_timer; receive_acknowledgement; **exit**)
  [>
  (timer_expired; sound_alarm; **stop**)

  may terminate successfully if an acknowledgement is received to a message, but may sound an alarm if no acknowledgement is received within some time period

- beware that behaviour may be disabled between its 'last' event and **stop** or **exit**

  thus, in the example above, **timer_expired** may occur *before* **reset_timer** and *after* **receive_acknowledgement**; this could be realistic in an actual implementation
- the left-hand behaviour expression is often non-terminating (iterative or recursive), and the right-hand one is often a closing-down behaviour expression; for example:

  DATA_TRANSFER [> DISCONNECTION

- '[>' is associative, but is not of course commutative

Wracając do przykładu P-K:

As a final requirement, let us assume that the channel may also fail at any time.

To model this fact, we put a disable [> at the end of the channel's behavior expression. We use the internal action **i**, to express an internal decision by the channel.

```
process Channel [ pc1, pc2, cc1, cc2] : exit :=
(
        (* same behavior expression as in Section 2.6 *)
)
[>  (   i; (* channel goes down *)
        exit
    )
endproc
```

Also, to ensure that the ***Consumer*** terminates successfully when the ***Channel*** goes down, we must disable the existing behavior of the ***Consumer*** with an **exit** which is always ready to synchronize with the **exit** after the [> in the ***Channel***. The ***Consumer*** then becomes:

```
process Consumer [ cc1, cc2 ] : exit :=
    (
       (*  same behavior expression as in Section 2.6  *)
     )
     [> exit
endproc
```

Note that the ***Consumer*** behavior is still equivalent to the one of Section 2.6. The difference would appear if the ***Consumer*** enabled itself recursively.

A corresponding change is necessary in the ***Producer.***

- Jak działa i jak wygląda operator przerwania ?

7.1.8. Operatory złożenia równoległego

Są trzy operatory złożenia równoległego w LOTOS:

- **Operator złożenia równoległego bez synchronizacji (interleaving) |||**
  Operator ten służy do opisu równoległości pomiędzy zachowaniami gdy żadna synchronizacja nie jest wymagana.

Przykład

(out1; out2; exit) ||| (in1; in2; exit)

Taki zapis nie oznacza jednak, że akcje out1 i in1 mogą wykonać się jednocześnie. W LOTOS wszystkie operacje są atomiczne i wszystkie procesy wykonywane są jakby na jednym procesorze (tzw. interleaving concurrency).
Jeśli wykonywane są więc dwa zdarzenia a i b równolegle, to oznacza to, że a może wykonać się przed b lub b przed a. Konsekwencją takiego działania jest to, że każda formuła wykorzystująca równoległość może być przepisana przy pomocy operatora wyboru. Ten przykład można równoważnie przepisać jako:

Przykład

```
out1; (out2;  in1;  in2;  exit
      []
      in1;    (out2;    in2 ;  exit   []  in2;  out2;  exit)  )
[]
in1;  (in2;   out1;  out2;  exit
      []
      out1;   (in2;     out2;  exit   []  out2;  in2;   exit   )
```

Czyli wszystkie możliwe kombinacje kolejności wykonać zdarzeń z obu zachowań.

- **Operator złożenia równoległego z częściową synchronizacją (selective) |[L]|**
  Tutaj procesy wykonują wybrane akcje równocześnie, akcje które oba procesy mają wykonać równolegle w tym samym czasie ujmowane są w miejsce litery L.

Przykład

```
    a; b; c; exit        (* subprocess 1 *)
  |[a]|
    d;  a; c; exit       (* subprocess 2 *)

is equivalent to
                         d; a; (b; c; exit  ||| c; exit)
or to
                         d; a; (b; (c; c; exit [] c; c; exit) [] c; b; c; exit)
or of course to
                         d; a; (b; c; c; exit [] c; b; c; exit)
```

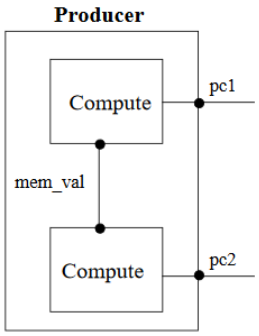
Proces 1 musi czekać aż proces drugi wykona d, ponieważ zdarzenie a musi być wykonane równocześnie przez dwa procesy. Pozostałe zdarzenia mogą wykonać się w dowolnej kolejności i nie wykonują się w jednej chwili.

- **Operator złożenia równoległego z pełną synchronizacją (full) ||**
  Wszystkie zdarzenia w procesach się synchronizują, a nie tylko niektóre wybrane, czyli tak jakby lista w operatorze częściowej synchronizacji zawierała wszystkie bramki.

7.1.9. Operator ukrywania (hiding)

W operatorze ukrywania chodzi o to, że chcemy zamodelować zdarzenie, które się synchronizuje, ale tylko wewnątrz pewnego procesu.

Przykładowo mamy proces Producenta, który składa się z dwóch procesów:




```
process Producer [ pc1, pc2 ] : exit :=
    hide mem_val in
    (
        Compute [pc1, mem_val]
        |[mem_val]|
        Compute [mem_val, pc2]
    )
    where
        process Compute [v1, v2] : exit :=
            v1; v2; exit
        endproc
endproc
```

Chcemy aby procesy z których składa się Producent synchronizowały się między sobą na zdarzenia mem_val, ale nie chcemy aby synchronizacja ta wymagała synchronizacji z otoczeniem procesu Producenta tak jak jest to dla zdarzeń pc1 i pc2. Wystarczy zatem dodać operator hide tak jak w przykładzie powyżej, wówczas zdarzenie mem_val nie musi się synchronizować z otoczeniem, a powoduje synchronizację procesów Compute.

7.1.10. Rekurencja procesów

Proces może odwoływać się w swojej definicji do samego siebie.

Przykład

```
process PRESS_UP [up, down] : noexit :=
   up; down; PRESS_UP [up, down]
endproc
```

Warto zwrócić uwagę na **noexit** w deklaracji procesu.

Rekurencja jest w LOTOS osiągana przez tworzenie instancji procesu - proces tworzy instancję samego siebie.

Pamiętać jednak należy, że o rekurencji w LOTOS nie należy myśleć jak o rekurencji w języku programowania. Nie ma czegoś takiego jak stos wywołania procesu itp. Rekurencja w LOTOS odnosi się do iteracji, a więc tak jakbyśmy kilkukrotnie wykonywali ten sam kod.

7.1.11. Reetykietowanie

Operator reetykietowania nie służy do specyfikowania zachowania procesu, jest to operator, który jest używany do formalnej definicji tego jak bramy formalne są zastępowane przez właściwe bramy przy tworzeniu instancji procesu. Operator ten jest zapisywany jako:

$$[g_1/g'_1, ..., g_n/g'_n]$$

8 Zagadnienie 9

Zagadnienie 9

LOTOS: ogólna składnia specyfikacji behawioralnej (podstawowe słowa kluczowe i format definicji poszczególnych obiektów)

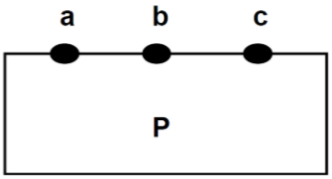
W języku lotos aby zdefiniować proces i wyrażenia behawioralne korzystamy z kilku słów kluczowych.

Process <process_id> <parameter_part> **:=**

<behaviour_expression>

endproc

Ex. : **process** P [a,b,c] := … **endproc**



process_id : an identifier which designates a process, that is a **process name**

parameter_part : in Basic LOTOS, a list of interaction points or gates

behaviour_expression : an expression defining the behaviour of the process, i.e. the allowed orderings of events

Such expressions are built up from more elementary expressions by using LOTOS operators.

Rysunek 5: Definiowanie procesu w LOTOS.

W LOTOS tworzone są instancje procesów, najpierw definiowany jest proces (co odpowiada napisaniu metody w języku programowania), a następnie tworzona jest jego instancja (co odpowiada wywołaniu danej metody w języku programowania).

Wracając do przykładu P-K, zdefiniowaliśmy już procesy osobno, teraz chcemy stworzyć system przy pomocy operatorów składania równoległego.

Przykład

```
1   specification Producer_Consumer [ pc1, pc2, cc1 cc2 ] : exit
2
3   behavior
4       (
5       Producer [ pc1, pc2 ]
6         |||
7       Consumer [ cc1, cc2 ]
8       )
9         ||
10          Channel [pc1, pc2, cc1, cc2]
11
12      where
13        process Producer [ out1, out2 ] : exit := . . .      (*As defined previously*)
14        process Consumer [ in1, in2 ] : exit := . . .        (*As defined previously *)
15        process Channel [ le1, le2, , re1, re2 ] : exit := . . . (*As defined previously *)
16      endspec
```

Before introducing the rest of LOTOS operators, we must explain some of the notions that we have just introduced. Line 1 introduces the **specification** ***Producer_Consumer***, which synchronizes with the environment through four gates *pc1, pc2, cc1, cc2*. Line 3, **behavior,** indicates the beginning of ***Producer_Consumer***'s behavior expression. Any global data abstractions would have to be declared on line 2. The behavior expression of this specification is the composition of three instances of three processes. Line 12, the **where** clause, defines the processes that are used in the behavior expression of the specification. Lines 13, 14 and 15 introduce the definitions of the three processes ***Producer, Consumer*** and ***Channel.*** Note that each process definition has a list of formal gates, which must be relabelled with actual gates.

It is important to know that relabelling is done dynamically as each action is executed, rather than statically as the process is instantiated. For example, let:

```
process     P [a, b, c] : noexit :=
            a; b; stop |[a]| a; c; stop
endproc
```

The static relabelling of instantiating ***P*** with ***P****[c, c, a]* would be equivalent to *c; c;* ***stop*** *|[c]| c; a;* ***stop***, i.e. to *c; a;* ***stop***. In LOTOS' dynamic relabelling instead, the substitution is done before the actions are offered to the environment of the relabelled behavior. So, the execution of *a; b;* ***stop*** *|[a]| a; c;* ***stop*** results in *a; (b;* ***stop*** *||| c;* ***stop)***, which is equivalent to *a; (b; c;* ***stop*** *[] c; b;* ***stop)***. This , of course, becomes *c; (c; a;* ***stop*** *[] a; c;* ***stop)*** after the relabelling.

9 Zgadanienie 6

Zagadnienie 6

LOTOS: przekazywanie/uzgadnianie danych miedzy procesami za pośrednictwem określonych bram (akcji synchronizujących), postaci ofert komunikacyjnych

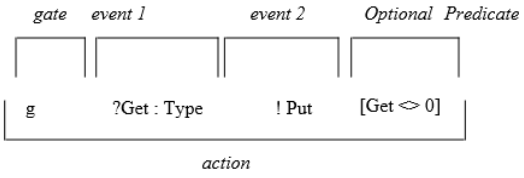
Dane i przekazywanie danych są dostępne w pełnej wersji LOTOS, tzw. full LOTOS.
W podstawowym LOTOS akcja była synonimem bramy. W full LOTOS akcja składa się z trzech komponentów:

- nazwa bramy
- lista zdarzeń
- opcjonalny predykat

Procesy się synchronizują pod warunkiem, że

- Bramy nazywają się tak samo
- Lista zdarzeń pasuje do siebie
- Predykat jest spełniony

Przykład akcji w full LOTOS:



Zdarzenie może oferować (!) lub akceptować wartość (?). Możliwe typy komunikacji:

Table 5.1 - Types of interaction

Process A	**Process B**	**synchron. condition**	**type of interaction**	**effect**
g **!**E_1	g **!**E_2	value(E_1) = value(E_2)	value matching	synchronization
g **!**E	g **?**x:t	value(E) is of sort t	value passing	after synchronization x = value(E)
g **?**x:t	g **?**y:u	t = u	value generation	after synchronization x = y = v, where v is some value of sort t

10 Zagadnienie 7

Zagadnienie 7

Redukcja etykietowanych systemów przejść (LTS) w pakiecie CADP, różnego typu równoważności bisymulacyjne grafów LTS

W pakiecie CADP dla danego systemu możemy wygenerować LTS, a następnie zredukować go według wskazanego typu równoważności przy użyciu narzędzia REDUCTOR. Grafy LTS mogą być porównywane przy użyciu wieli równoważności bisymulacyjnych, m.in.:

- Strong equivalence - stany są równoważne gdy akcje możliwe do wykonania w jednym są możliwe do wykonania w drugim oraz stany wynikowe są również powiązane.

  Definition 2.3.7 (Bisimulation). Let $A{=}(S, Act, \longrightarrow, s, T)$ be a labelled transition system. A binary relation $R \subseteq S \times S$ is called a *strong bisimulation relation* iff for all $s, t \in S$ such that sRt holds, it also holds for all actions $a \in Act$ that:

  1. if $s \xrightarrow{a} s'$, then there is a $t' \in S$ such that $t \xrightarrow{a} t'$ with $s'Rt'$,
  2. if $t \xrightarrow{a} t'$, then there is a $s' \in S$ such that $s \xrightarrow{a} s'$ with $s'Rt'$, and
  3. $s \in T$ if and only if $t \in T$.

- Branching equivalnce

  ### 2.4.3 (Rooted) Branching bisimulation

  The definition of branching bisimulation is very similar to that of strong bisimulation. But now, instead of letting a single action be simulated by a single action, an action can be simulated by a sequence of internal transitions, followed by that single action. See the diagram at the left of figure 2.13. It can be shown that all states that are visited via the τ-actions in this diagram are branching bisimilar.

- Weak equivalence

  A slight variation of branching bisimulation is weak bisimulation. We give its definition here, because weak bisimulation was defined well before branching bisimulation was invented and therefore weak bisimulation is much more commonly used in the literature.

  The primary difference between branching and weak bisimulation is that branching bisimulation preserves 'the branching structure' of processes. For instance the last pair

Stosowanie tych oraz innych równoważności dostępne jest w pakiecie CADP.