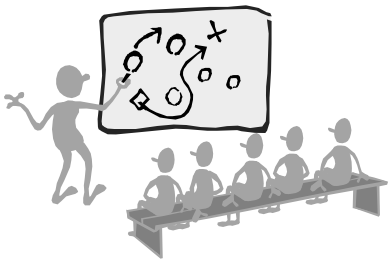


Podstawy przetwarzania rozproszonego

Detekcja zakończenia

Spis algorytmów

- Detekcja zakończenia dla modelu synchronicznego
- Detekcja zakończenia dla dyfuzyjnego modelu przetwarzania
- Jednofazowy algorytm detekcji zakończenia dla modelu atomowego
- Wektorowy algorytm detekcji zakończenia dla modelu atomowego
- Detekcja zakończenia statycznego
- Detekcja zakończenia dynamicznego

Problem detekcji zakończenia – przykład

Przykład 1: Rozproszone sortowanie

Każdy z procesów P_{i+1} ma za zadanie uporządkować (posortować) przypisany mu na wstępie zbiór $\mathcal{X}_{i+1}$ i wyznaczyć element min_{i+1}. Następnie procesy wysyłają elementy min_{i+1} do swoich lewych sąsiadów i oczekują na odpowiedź zawierającą max_i. Po otrzymaniu wiadomości z wartością min_{i+1}, a przed wysłaniem odpowiedzi, proces P_i wyznacza nowy element max_i.

W ogólności, nowo wyznaczony max_i może być równy otrzymanemu ostatnio min_{i+1}. Następnie każdy z procesów wysyła odpowiedź ze swoim elementem maksymalnym do prawego sąsiada. Po otrzymaniu odpowiedzi, procesy znów sortują zbiory $\mathcal{X}_i$ i jeśli w wyniku tego sortowania wartość min_{i+1} różnić się będzie od poprzednio wysłanego elementu minimalnego, to proces wysyła ten nowy element min_{i+1} do lewego sąsiada.

Problem detekcji zakończenia – przykład

Cel sortowania rozproszonego zostaje osiągnięty, gdy uporządkowany zostanie każdy zbiór $\mathcal{X}_i$, a ponadto dla każdego i: $1 \leqslant i \leqslant n\text{-}1$, $max_i < min_{i+1}$. Problem jednak w tym, że każdy proces ma tylko wiedzę lokalną, dotyczącą jego lokalnego zbioru i częściowo zbiorów bezpośrednich sąsiadów. Na tej podstawie procesy nie mogą jednak wnioskować o zakończeniu całego przetwarzania. Potrzebny jest zatem dodatkowy mechanizm pozwalający stwierdzić, że globalne warunki zakończenia sortowania rozproszonego zostały spełnione.

Problem detekcji zakończenia – przykład

Przykład 2: Algorytm Matterna konstrukcji spójnego obrazu stanu globalnego w środowisku z kanałami *nonFIFO*.

Ze względu na możliwe zmiany uporządkowania wiadomości w kanałach otrzymanie wiadomości koloru *Red* nie przesądzało o tym, że w kanałach nie ma już wcześniej wysłanych wiadomości koloru *White*. Ponieważ jednak zbiór otrzymanych przez proces koloru *Red* wiadomości *White* określa stan kanału w wyznaczanym obrazie stanu globalnego, niezbędne jest niezależne od procesu konstrukcji obrazu stanu globalnego sprawdzenie, czy w kanałach nie ma jeszcze wiadomości koloru *White*. Dopiero bowiem wówczas, gdy stwierdzimy, że wszystkie wiadomości koloru *White* zostały odebrane, wyznaczone stany kanałów odpowiadają spójnemu obrazowi stanu globalnego.

Wprowadzenie

Definicja nieformalna:

Nieformalnie problem detekcji zakończenia przetwarzania rozproszonego polega na sprawdzeniu, czy wszystkie procesy przetwarzania są w stanie pasywnym oraz czy żadna wiadomość będąca w kanale (transmitowana lub dostępna) nie uaktywni któregokolwiek z tych procesów.

Oznaczenia

W formalnej definicji tego problemu wykorzystane są następujące oznaczenia:

- $passive_i$ – zmienna logiczna (predykat) przyjmująca wartość *True* [wtedy tylko wtedy], gdy proces P_i jest pasywny
- $available_i$ – tablica [1..*n*] zmiennych logicznych procesu P_i skojarzona z wiadomościami dostępnymi
- $available_i[j]$ – *j*-ty element tablicy $available_i$ przyjmujący wartość *True*, gdy dla P_i jest dostępna wiadomość wysłana przez P_j
- $in\text{-}transit_i$ – tablica [1..*n*] zmiennych logicznych procesu P_i skojarzona z wiadomościami transmitowanymi
- $in\text{-}transit_i[j]$ – *j*-ty element tablicy $in\text{-}transit_i$ przyjmujący wartość *True*, gdy wiadomość wysłana przez P_j do P_i należy do $L^T_{j,i}$, a więc jest transmitowana i nie jest jeszcze dostępna

$\mathcal{AV}_i = \{P_j : available_i[j] = True\}$

$\mathcal{IT}_i = \{P_j : intransit_i[j] = True\}$

Zakończenie dynamiczne

Przetwarzanie rozproszone Π jest w danej chwili w stanie *zakończenia dynamicznego*, gdy spełniony jest predykat:

$$\mathbf{Dterm}(\mathcal{P}) \equiv \forall P_i :: P_i \in \mathcal{P} :: (passive_i \wedge \neg\, activate_i(\mathcal{AV}_i \cup \mathcal{IT}_i))$$

Żaden proces składowy przetwarzania rozproszonego nie będzie już nigdy uaktywniony. Stan ten będzie utrzymywany pomimo, że pewne wiadomości są wciąż transmitowane ($\mathcal{IT}_i \neq \varnothing$), a pewne wiadomości są już dostępne ($\mathcal{AV}_i \neq \varnothing$).

Definicja zakończenia dynamicznego uwzględnia rzeczywistą aktywność procesów wyrażoną przez zmienną $passive_i$ oraz potencjalną aktywność wyrażoną przez predykat $activate_i(\mathcal{AV}_i \cup \mathcal{IT}_i)$.

Predykat $Dterm(\mathcal{P})$ jest predykatem stabilnym.

Zakończenie statyczne

Przetwarzanie rozproszone jest w danej chwili w stanie *zakończenia statycznego*, gdy spełniony jest predykat:

$$\boldsymbol{Sterm}(\mathcal{P}) \equiv \forall P_i :: P_i \in \mathcal{P} :: (passive_i \wedge (\mathcal{IT}_i = \varnothing) \wedge \neg\, activate_i(\mathcal{AV}_i))$$

Predykat zakończenia statycznego przyjmuje wartość *True*, gdy wszystkie procesy są pasywne, wszystkie wiadomości znajdujące się w kanałach są dostępne ($\mathcal{IT}_i = \varnothing$) i dla żadnego procesu nie jest spełniony warunek uaktywnienia ($activate_i(\mathcal{AV}_i)=False$). Definicja ta uwzględnia zatem zarówno stany procesów jak i stany kanałów.

Porównując z $Dterm(\mathcal{P})$, predykat $Sterm(\mathcal{P})$ odpowiada detekcji nieco późniejszej, gdyż dodatkowo wymaga się, by wiadomości nie były już transmitowane ($\mathcal{IT}_i = \varnothing$).

Zakończenie dynamiczne ↔ statyczne

Niech $\vartheta \rightsquigarrow \vartheta'$ oznacza, że zajście predykatu ϑ prowadzi w skończonym choć nieprzewidzianym czasie do zajścia predykatu ϑ'.

$$Dterm(\mathcal{P}) \rightsquigarrow Sterm(\mathcal{P})$$

Dowód

W chwili τ, gdy $Dterm(\mathcal{P}) = True$, wszystkie procesy P_i są pasywne, zachodzi $\neg activate_i(\mathcal{AV}_i \cup \mathcal{IT}_i)[\tau]$, a część wiadomości może znajdować się w kanałach. Jednakże, wszystkie wiadomości transmitowane są brane pod uwagę, a ich dotarcie do węzłów docelowych i w konsekwencji ich dostępność, jest uwzględniona w wartości predykatu $\neg activate_i(\mathcal{AV}_i \cup \mathcal{IT}_i)$. Wobec niezawodności kanałów, wiadomości transmitowane osiągną węzły docelowe po skończonym choć nieprzewidywalnym czasie, w pewnym momencie $\tau' > \tau$. Wówczas $\mathcal{IT}_i[\tau'] = \varnothing$. Wobec stałej pasywności wszystkich procesów od chwili τ, w każdej chwili $\tau'' \geqslant \tau'$, $\mathcal{AV}_i[\tau''] = \mathcal{AV}_i[\tau] \cup \mathcal{IT}_i[\tau]$ oraz $\mathcal{IT}_i[\tau''] = \varnothing$.

Stąd otrzymujemy:

$$\neg activate_i(\mathcal{AV}_i[\tau'']) = True, \quad \mathcal{IT}_i[\tau''] = \varnothing$$

Predykat $Sterm(\mathcal{P})$ przyjmuje zatem wartość *True* . □

Klasyczna definicja zakończenia

W klasycznej definicji zakończenia przyjmowano, że przetwarzanie rozproszone jest w stanie zakończenia, jeżeli w danej chwili wszystkie procesy są pasywne i wszystkie kanały są puste, a więc gdy zachodzi następujący predykat:

$$Cterm(\mathcal{P}) \equiv \forall P_i :: P_i \in \mathcal{P} :: (passive_i \wedge (\mathcal{IT}_i = \varnothing) \wedge (\mathcal{AV}_i = \varnothing).$$

Klasyczna definicja zakończenia

Rozważmy teraz relację między $Sterm(\mathcal{P})$ a $Cterm(\mathcal{P})$.

Jeżeli procesy są uaktywnione w sposób natychmiastowy w chwili zajścia predykatu $activate_i(\mathcal{AV}_i)$ przez każdą dostępną wiadomość, to przetwarzanie rozproszone obejmujące zbiór procesów $\mathcal{P}$ jest statycznie zakończone wtedy i tylko wtedy, gdy zachodzi predykat $Cterm(\mathcal{P})$.

Dowód.

Zgodnie z założeniem, procesy stają się aktywne natychmiast w chwili zajścia predykatu $activate_i(\mathcal{AV}_i)$. W konsekwencji $passive_i$ jest równe *True* tylko wówczas, gdy $\neg activate_i(\mathcal{AV}_i)$. Stąd, w wypadku procesów uaktywnianych każdą wiadomością:

$$passive_i \wedge \neg activate_i(\mathcal{AV}_i) \equiv passive_i \wedge (\mathcal{AV}_i = \varnothing).$$

W konsekwencji:

$$Sterm(\mathcal{P}) \equiv \forall P_i :: P_i \in \mathcal{P} :: (passive_i \wedge (\mathcal{IT}_i = \varnothing) \wedge \neg activate_i(\mathcal{AV}_i))$$
$$\equiv \forall P_i :: P_i \in \mathcal{P} :: (passive_i \wedge (\mathcal{IT}_i = \varnothing) \wedge (\mathcal{AV}_i = \varnothing) \equiv Cterm(\mathcal{P}) \ . \ \square$$

Problem detekcji zakończenia

Problem detekcji zakończenia przetwarzania rozproszonego obejmującego zbiór procesów, sprowadza się do sprawdzenia czy przetwarzanie osiągnęło określony stan zakończenia, a więc – czy zachodzi odpowiedni predykat: $Dterm(\mathcal{P})$, $Sterm(\mathcal{P})$ lub $Cterm(\mathcal{P})$.

Problem detekcji zakończenia – przykład

Rozważmy problem sortowania rozproszonego zbioru $\mathcal{X}$ składającego się z v różnych liczb naturalnych, w środowisku rozproszonym o n węzłach (procesorach), $n < v$.

Przyjmijmy, że zbiór $\mathcal{X}$ zostaje wstępnie podzielony na podzbiory $\mathcal{X}_i$ w taki sposób, że:

$$\mathcal{X} = \mathcal{X}_i, \text{ oraz}$$

$$\forall i,j :: (1 \leqslant i, j \leqslant n) \wedge (i \neq j) :: (\mathcal{X}_i \cap \mathcal{X}_j = \varnothing) \wedge (\forall i :: 1 \leqslant i \leqslant n :: \mathcal{X}_i \neq \varnothing)$$

Niech:

- v_i – liczba elementów zbioru $\mathcal{X}_i$
- min_i – minimalny element zbioru $\mathcal{X}_i$
- max_i – maksymalny element zbioru $\mathcal{X}_i$
- P_i – procesy tworzące przetwarzanie rozproszone o topologii łańcucha skojarzone ze zbiorami $\mathcal{X}_i$.

Pary procesów składowych P_i, P_{i+1}, $1 \leqslant i \leqslant n$-1, połączone są kanałami dwukierunkowymi.

Detekcja zakończenia dla synchronicznego modelu przetwarzania (1)

W modelu *przetwarzania synchronicznego* przyjmuje się, że transmisje są natychmiastowe. Stąd kanały mogą być uznane za puste przez cały czas i problem zakończenia sprowadza się do sprawdzenia czy wszystkie procesy są jednocześnie pasywne.

Stan zakończenia opisuje następujący predykat:

$$Iterm(\mathcal{P}) \equiv \forall P_i :: P_i \in \mathcal{P} :: passive_i$$

Algorytm detekcji zakończenia dla modelu przetwarzania synchronicznego, wykorzystuje koncepcję ciągu cykli detekcyjnych i zakłada, że wszystkie monitory procesów aplikacyjnych połączone są w logiczny pierścień i obserwują stany procesów aplikacyjnych.

Detekcja zakończenia dla synchronicznego modelu przetwarzania (2)

- Monitorom (procesom) przypisany jest kolor *White* albo *Black*.
- Monitory przesyłają wzdłuż pierścienia wiadomość kontrolną – znacznik typu TOKEN, który również może mieć kolor *White* albo *Black.*
- Początkowo monitory mają kolor *White*, a zmieniają kolor na *Black*, gdy odpowiadający im proces aplikacyjny wyśle wiadomości do procesu o indeksie większym.
- Monitor inicjujący detekcję zakończenia $Q_\alpha = Q_1$ wysyła znacznik koloru *White* do swego następnika w pierścieniu Q_n jeżeli obserwowany przez niego proces aplikacyjny jest pasywny.
- Każdy kolejny monitor Q_i odbierający znacznik czeka aż obserwowany przez niego proces stanie się pasywny i wówczas wysyła znacznik o kolorze zgodnym z kolorem monitora.
- Po wysłaniu znacznika monitorowi przypisywany jest kolor *White*.

Algorytm kończy się, gdy znacznik koloru *White* dotrze do inicjatora. Dla uproszczenia prezentacji, w algorytmie wykorzystano funkcje:

$$succ(i) = (i)\ mod_n + 1$$
$$pred(i) = (i+n-2)\ mod_n + 1$$

Detekcja zakończenia dla synchronicznego modelu przetwarzania – algorytm

- Wstęp
- Typy komunikatów
- Deklaracje
- Procedury
- Akcje

- Wstęp
- Typy komunikatów
- Deklaracje
- Procedury
- Akcje

TYPY KOMUNIKATÓW

type `PACKET` **extends** `FRAME` **is record of**
data : MESSAGE
end record

type `TOKEN` **extends** `FRAME` **is record of**
colour : **enum** { *White, Black* }
end record

Detekcja zakończenia dla modelu synchronicznego (deklaracje)

➲ Wstęp
➲ Typy komunikatów
➲ Deklaracje
➲ Procedury
➲ Akcje

DEKLARACJE

msgIn : `MESSAGE`
pcktOut : `PACKET`
tokenOut : `TOKEN`
$tokenPresent_i$: `BOOLEAN` := *False*
$procColour_i$: **enum** { *White, Black* } := *White*
$terminationDetected_i$: `BOOLEAN` := *False*

Detekcja zakończenia dla modelu synchronicznego (procedury)

➲ Wstęp
➲ Typy komunikatów
➲ Deklaracje
➲ Procedury
➲ Akcje

PROCEDURY

procedure `InitProc( )`
tokenOut.colour := *White*
send(Q_α, $Q_{pred(\alpha)}$, *tokenOut*)
$tokenPresent_\alpha$:= *False*
$procColour_\alpha$:= *White*
end procedure

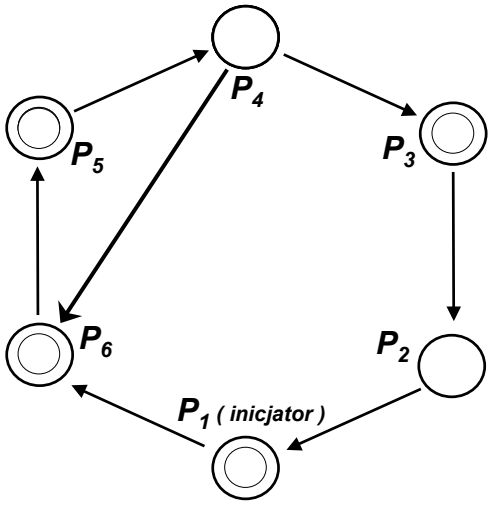
Detekcja zakończenia dla modelu synchronicznego (akcje)

- Wstęp
- Typy komunikatów
- Deklaracje
- Procedury
- Akcje

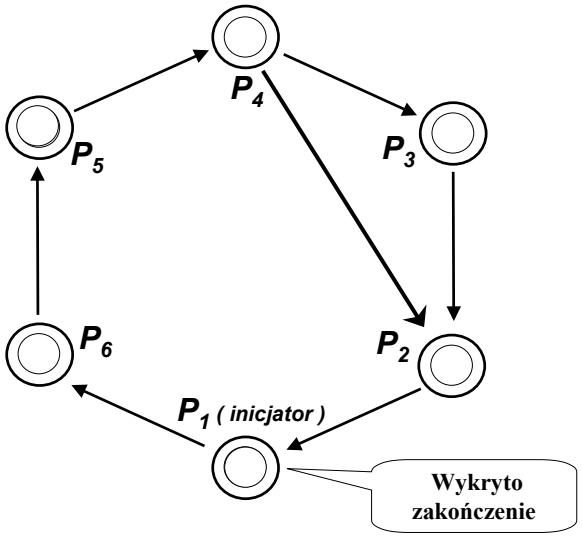
AKCJE

```
1.  when e_start( Q_α, TerminationDetection ) do
2.    wait until passive_α
3.    InitProc( )
4.  end when

5.  when e_send( P_i, P_j, msgOut : MESSAGE) do
6.    if  i < j
7.    then
8.       procColour_i := Black
9.    end if
10.   pcktOut.data := msgOut
11.   send( Q_i, Q_j, pcktOut )
12. end when

13. when e_receive( Q_j, Q_i, pcktIn : PACKET ) do
14.   msgIn := pcktIn.data
15.   deliver( P_j, P_i, msgIn )
16. end when

17. when e_receive( Q_succ(i), Q_i, tokenIn : TOKEN ) do
18.   tokenPresent_i := True
19.   wait until passive_i
20.   if i = α then
21.     if procColour_i = White ∧ tokenIn.colour = White
22.     then
23.       terminationDetected_i := True
24.       decide( terminationDetected_i )
25.     else
26.       InitProc( )
27.     end if
28.   else
29.     tokenOut.colour := tokenIn.colour
30.     if procColour_i = Black
31.     then
32.       tokenOut.colour := Black
33.     end if
34.     send( Q_i, Q_pred(i), tokenOut )
35.     tokenPresent_i := False
36.     procColour_i := White
37.   end if
38. end when
```


Przykład detekcji zakończenia w środowisku synchronicznym
– runda zakończona niepowodzeniem –



Przykład detekcji zakończenia w środowisku synchronicznym – runda zakończona sukcesem –



(c) Zakład Systemów Informatycznych Slajd 26

Detekcja zakończenia dla dyfuzyjnego modelu przetwarzania (1)

Przetwarzanie dyfuzyjne (ang. *diffusing computation*) jest specyficznym przetwarzaniem rozproszonym, w którym wyróżnia się:

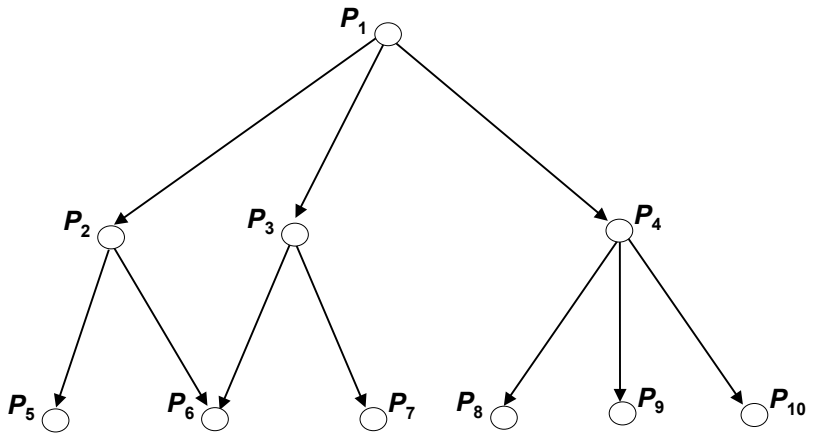
- *inicjatora* (środowisko) – może w dowolnej chwili rozpocząć przetwarzanie dyfuzyjne wysyłając wiadomość aplikacyjną do jednego lub wielu procesów kooperujących (zakłada się, że inicjacja taka zachodzi tylko raz),
- hierarchię kooperującymi procesów.

Każdy proces po uzyskaniu pierwszej wiadomości aplikacyjnej, nadawcę tej wiadomości traktuje jako proces *angażujący* (ang. *engager*) i realizuje dalsze przetwarzanie wysyłając wiadomości do innych procesów, w tym ewentualnie do inicjatora.

Proces aplikacyjny może w dowolnej chwili stać się pasywny i oczekiwać na uaktywniającą go wiadomość aplikacyjną od dowolnego innego procesu. Warunek uaktywnienia procesu definiuje zatem model żądań *OR* i zbiór warunkujący $\mathcal{D}_i = \mathcal{P} \setminus \{ P_i \}$.

(c) Zakład Systemów Informatycznych Slajd 27

Graf przetwarzania dyfuzyjnego



(c) Zakład Systemów Informatycznych Slajd 28

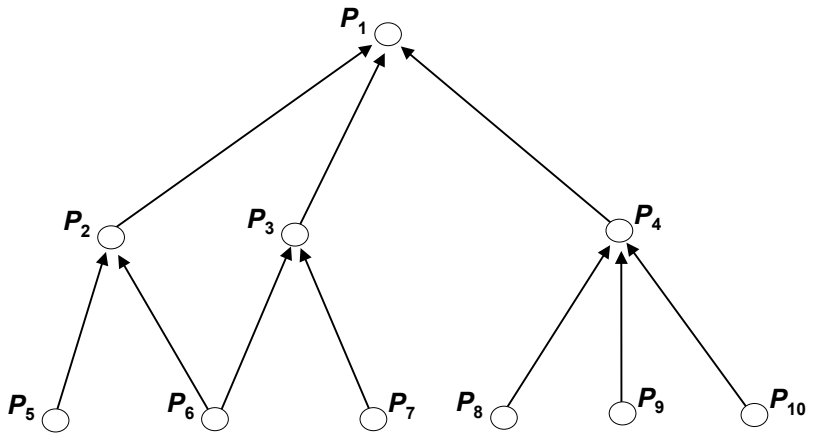
Detekcja zakończenia dla dyfuzyjnego modelu przetwarzania (2)

Koncepcja algorytmu detekcji zakończenia przetwarzania dyfuzyjnego:

- Monitory procesów aplikacyjnych przesyłają wiadomości kontrolne typu `SIGNAL` jako pewnego rodzaju odpowiedzi na wiadomości aplikacyjne.
- Monitor pasywnego inicjatora może stwierdzić zakończenie całego przetwarzania aplikacyjnego, gdy odbierze wiadomości `SIGNAL` od wszystkich monitorów związanych z procesami uaktywnionymi przez inicjatora.

(c) Zakład Systemów Informatycznych Slajd 29

Graf przetwarzania dyfuzyjnego



(c) Zakład Systemów Informatycznych Slajd 31

Detekcja zakończenia dla dyfuzyjnego modelu przetwarzania – algorytm

- Wstęp
- Typy komunikatów
- Deklaracje
- Procedury
- Akcje

(c) Zakład Systemów Informatycznych Slajd 32

Detekcja zakończenia dla dyfuzyjnego modelu przetwarzania
(typy komunikatów)

- Wstęp
- Typy komunikatów
- Deklaracje
- Procedury
- Akcje

TYPY KOMUNIKATÓW

type PACKET **extends** FRAME **is record of**
data : MESSAGE
end record

type SIGNAL **extends** FRAME

(c) Zakład Systemów Informatycznych Slajd 33

Detekcja zakończenia dla dyfuzyjnego modelu przetwarzania
(deklaracje)

- Wstęp
- Typy komunikatów
- Deklaracje
- Procedury
- Akcje

DEKLARACJE

msgIn	: MESSAGE
pcktOut	: PACKET
signalIn	: SIGNAL
$engager_i$	: PROCESS_ID
$notEngager_i$	: **set of** PROCESS_ID
$recvNo_i$	: INTEGER := 0
$sentNo_i$	: INTEGER := 0
$terminationDetected_i$	: BOOLEAN := False

(c) Zakład Systemów Informatycznych Slajd 34

Detekcja zakończenia dla dyfuzyjnego modelu przetwarzania (akcje)

➲ Wstęp
➲ Typy komunikatów
➲ Deklaracje
➲ Procedury
➲ Akcje

AKCJE

1. **when** e_start(P_α, $\mathcal{P}_\alpha^R$, *msgOut* : MESSAGE, *DiffusingComputation*) **do**
2. $\mathcal{Q}_\alpha^R := \{ Q_j : P_j \in \mathcal{P}_\alpha^R \}$
3. $pcktOut.data := msgOut$
4. $sentNo_\alpha := | \mathcal{Q}_\alpha^R |$
5. **send**(Q_α, $\mathcal{Q}_\alpha^R$, *pcktOut*)
6. **end when**

7. **when** e_send(Q_i, Q_j, *signalOut* : SIGNAL) **do**
8. **if** $recvNo_i = 1 \wedge sentNo_i = 0 \wedge passive_i$
9. **then**
10. $Q_j := engager_i$
11. **send**(Q_i, Q_j, *signalOut*)
12. **else**
13. **for** $Q_j \in notEngager_i$ do
14. $notEngager_i := notEngager_i \setminus Q_j$
15. **send**(Q_i, Q_j, *signalOut*)
16. **end for**
17. **end if**
18. $recvNo_i := recvNo_i - 1$
19. **end when**

20. **when** e_receive(Q_j, Q_i, *signalIn* : SIGNAL) **do**
21. $sentNo_i := sentNo_i - 1$
22. **if** $Q_i = Q_\alpha \wedge sentNo_i = 0$ **then**
23. **wait until** $passive_i$
24. $terminationDetected_i := True$
25. **decide**($terminationDetected_i$)
26. **end if**
27. **end when**

28. **when** e_send(P_i, P_j, *msgOut* : MESSAGE) **do**
29. $pcktOut.data := msgOut$
30. $sentNo_i := sentNo_i + 1$
31. **send**(Q_i, Q_j, *pcktOut*)
32. **end when**

33. **when** e_receive(Q_j, Q_i, *pcktIn* : PACKET) **do**
34. **if** $recvNo_i = 0$ **then**
35. $engager_i := Q_j$
36. **else**
37. $notEngager_i := notEngager_i \cup \{Q_j\}$
38. **end if**
39. $recvNo_i := recvNo_i + 1$
40. $msgIn := pcktIn.data$
41. **deliver**(P_j, P_i, *msgIn*)
42. **end when**

Detekcja zakończenia dla atomowego modelu przetwarzania (1)

W *atomowym modelu przetwarzania,* czasy przetwarzania lokalnego są zaniedbywane (zerowe), a opóźnienia są związane tylko z transmisją.

Można przyjąć, że procesy (uaktywniane natychmiastowo) są przez cały czas pasywne i sprowadzić problem detekcji zakończenia do wyznaczania stanów kanałów. Przyjmijmy dla uproszczenia, model żądań typu OR. Oznacza to, że dowolna wiadomość w kanale uaktywni proces.

Wówczas zakończenie przetwarzania opisuje następujący predykat:

$$Aterm(\mathcal{P}) \equiv \forall P_i :: P_i \in \mathcal{P} :: (\mathcal{IT}_i = \varnothing)$$

Detekcja zakończenia dla atomowego modelu przetwarzania (2)

Rozważmy dla modelu atomowego rozwiązanie problemu detekcji zakończenia przetwarzania rozproszonego z zastosowaniem liczników wiadomości.

- ❑ $sc_i(\tau)$ – określający liczbę wiadomości wysłanych do chwili τ przez P_i
- ❑ $rc_i(\tau)$ – określający liczbę wiadomości odebranych do chwili τ przez P_i
- ❑ $SC(\tau)$ – sumaryczna liczba wiadomości wysłanych przez wszystkie procesy przetwarzania rozproszonego do chwili τ
- ❑ $RC(\tau)$ – sumaryczna liczba wiadomości wysłanych przez wszystkie procesy przetwarzania rozproszonego do chwili τ

Dla modelu atomowego, równość $SC(\tau) = RC(\tau)$ oznacza, że każda wiadomość wysłana została odebrana. Tym samym wszystkie kanały są w chwili τ puste, a więc osiągnięty został stan zakończenia.

Problem polega na tym, że wyznaczenie $SC(\tau)$ i $RC(\tau)$ nie jest w systemie rozproszonym proste.

Detekcja zakończenia dla atomowego modelu przetwarzania (3)

Celem wysłania wiadomości kontrolnej (znacznika) do wszystkich monitorów jest wyznaczenie sumarycznej liczby wiadomości wysłanych i odebranych przez wszystkie procesy aplikacyjne.

Wiadomość kontrolna wysłana przez inicjatora Q_α przesyłana jest między monitorami, a po dotarciu do wszystkich monitorów wraca do Q_α.

Każdy z monitorów Q_i dodaje do znacznika stany liczników sc_i i rc_i odpowiadające bieżącej chwili τ_i.

Ponieważ kanały wprowadzają opóźnienia, zebrane liczniki odpowiadają więc różnym momentom τ_i, $1 \leqslant i \leqslant n$.

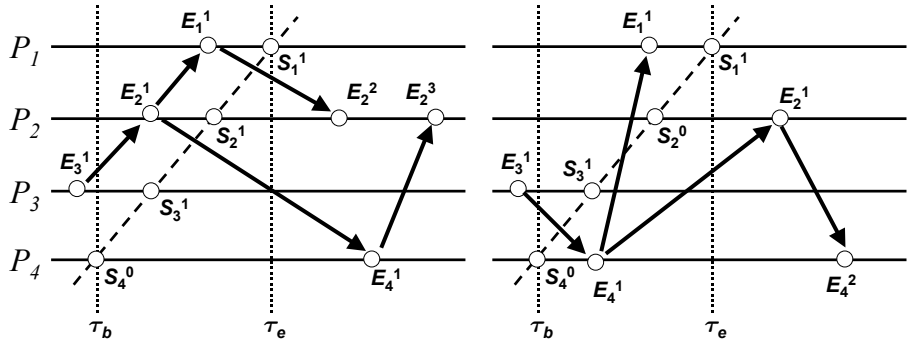
Oznaczmy przez SC^* i RC^* odpowiednio sumy zebranych przez znacznik liczników, a więc:

$$SC^* = sc_i(\tau_i),\ RC^* = rc_i(\tau_i),$$

pamiętając, że:

$$SC(\tau) = sc_i(\tau),\ RC(\tau) = rc_i(\tau),$$

Przykłady wyznaczania liczników *RC** i *SC**



$SC^* = RC^*$, ale $\forall \tau : SC(\tau) \neq RC(\tau)$

Detekcja zakończenia dla atomowego modelu przetwarzania twierdzenie

W algorytmie detekcji zakończenia rozważane są dwie fazy detekcji. Pierwsza rozpoczyna się o chwili τ_b^1 i kończy w chwili τ_e^1, a druga odpowiednio w chwilach τ^2_b i τ^2_e. Przyjęto ponadto, że: $\tau_b^1 < \tau_e^1 < \tau_b^2 < \tau_e^2$.

Oznaczmy teraz przez *SC** i *RC** liczniki wyznaczone w pierwszej fazie, a przez *SC*** i *RC*** liczniki oznaczone w drugiej fazie

Jeżeli

$$RC^* = SC^* = RC^{**} = SC^{**}$$

to monitorowane przetwarzanie aplikacyjne osiągnęło stan zakończenia przed zakończeniem procesu detekcji

Dowód

W celu udowodnienia powyższego twierdzenia wykażemy, że jeżeli $RC^* = SC^{**}$, to przetwarzanie jest w stanie zakończenia.

Detekcja zakończenia dla atomowego modelu przetwarzania
Lemat 1

Lokalne liczniki wiadomości są monotoniczne.
Tak więc, jeżeli $\tau \leqslant \tau'$, to $sc_i(\tau) \leqslant sc_i(\tau')$ i $rc_i(\tau) \leqslant rc_i(\tau')$.

Dowód

Dowód tego lematu wynika wprost z definicji liczników rc_i i sc_i.

Detekcja zakończenia dla atomowego modelu przetwarzania
Lemat 2

Sumaryczna liczba wiadomości wysłanych lub odebranych jest monotoniczna.
Jeżeli zatem $\tau \leqslant \tau'$, to $SC(\tau) \leqslant SC(\tau')$ i $RC(\tau) \leqslant RC(\tau')$.

Dowód

Dowód wynika wprost z definicji SC i RC.

Detekcja zakończenia dla atomowego modelu przetwarzania
Lemat 3

$$RC^* \leqslant RC(\tau_e^1)$$

Dowód

Z definicji,

$RC^* = rc_i(\tau_i)$, przy czym dla każdego i $\in \{1, 2, \ldots, n\}$, $\tau_b^1 \leqslant \tau_i \leqslant \tau_e^1$.

Z kolei,

$RC(\tau_e^1) = rc_i(\tau_e^1)$.

Ponieważ liczniki są monotoniczne i dla każdego i, $i \in \{1, 2, \ldots, n\}$, $\tau_i \leqslant \tau_e^1$, więc tym samym $rc_i(\tau_e^1) \geqslant rc_i(\tau_i)$ dla każdego i, a w konsekwencji:

$RC^* \leqslant RC(\tau_e^1)$.

Detekcja zakończenia dla atomowego modelu przetwarzania
Lemat 4

$$SC^{**} \geqslant SC(\tau_b^2)$$

Dowód

Z definicji,

$SC^{**} = sc_i(\tau'_i)$, przy czym dla każdego $i \in \{1,2, \ldots, n\}$, $\tau_b^2 \leqslant \tau'_i \leqslant \tau_e^2$.

Z kolei,

$SC(\tau_b^2) = sc_i(\tau_b^2)$.

Ponieważ liczniki są monotoniczne i dla każdego i, $i \in \{1,2, \ldots, n\}$, $\tau_b^2 \leqslant \tau'_i$, więc tym samym $sc_i(\tau_b^2) \leqslant sc_i(\tau'_i)$ dla każdego i,

a w konsekwencji:

$SC(\tau_b^2) \leqslant SC^{**}$.

Detekcja zakończenia dla atomowego modelu przetwarzania
Lemat 5

Dla wszystkich τ: $RC(\tau) \leqslant SC(\tau)$

Dowód

Relacja jest prawdziwa, gdyż kanały są z założenia niezawodne.

Detekcja zakończenia dla atomowego modelu przetwarzania
dowód twierdzenia

Dowód

Z założenia wynika, że:

(1) $RC^* = SC^{**}$

Z kolei z przedstawionych lematów otrzymujemy, że:

(2) $SC^{**} \geqslant SC(\tau_b^2) \geqslant SC(\tau_e^1) \geqslant RC(\tau_e^1) \geqslant RC^*$

Relacje (1) i (2) zachodzą jednocześnie tylko wówczas, gdy:
$SC^{**} = SC(\tau_b^2) = SC(\tau_e^1) = RC(\tau_e^1) = RC^*$

Tym samym w chwili τ_e^1, $SC(\tau_e^1) = RC(\tau_e^1)$ a to oznacza, że wszystkie kanały są puste, a więc wystąpiło zakończenie przetwarzania. □

Jednofazowy algorytm detekcji zakończenia (1)

Istnieją propozycje realizacji detekcji zakończenia w jednym przebiegu, czyli *algorytmy jednofazowe*.

Załóżmy, że monitory $Q_i : i \in \{1,2, \ldots , n\}$ połączone są w logiczny pierścień, i posiadają:

- ❖ $SRBalance_i$ – lokalne liczniki (ang. send-receive balance) o początkowej wartości 0. Wartość licznika $SRBalance_i$ w każdej chwili jest równa sc_i - rc_i.
- ❖ $detectNo_\alpha$ – numer sekwencyjny cyklu detekcji zakończenia, zainicjowanego przez monitor Q_α
- ❖ $maxDetectNo_i$ – zmienna określającą numer sekwencyjny cyklu detekcji skojarzony z wiadomością wysłaną najpóźniej, spośród wszystkich wiadomości odebranych przez Q_i.

Jednofazowy algorytm detekcji zakończenia (2)

W algorytmie detekcji zakończenia przetwarzania rozproszonego przesyłana jest jest wiadomość kontrolna (znacznik) typu `TOKEN` między kolejnymi monitorami. Wiadomość ta zawiera:

- ❖ identyfikator inicjatora Q_α – *initId*
- ❖ numer sekwencyjny *detectNo* cyklu detekcji zainicjowanego przez Q_α,
- ❖ Pole *SRAccu,* sumy liczników $SRBalance_i$ monitorów Q_i odwiedzonych już przez znacznik
- ❖ pole flagi niepoprawności procesu detekcji *invalid.* Flaga ta przyjmuje ostatecznie wartość *True*, jeżeli którykolwiek proces aplikacyjny otrzymał między kolejnymi cyklami detekcji wiadomość, która narusza warunek konieczny poprawności detekcji: $maxDetectNo_i \geqslant tokenIn.detectNo$.

Jednofazowy algorytm detekcji zakończenia dla modelu atomowego

- Wstęp
- Typy komunikatów
- Deklaracje
- Procedury
- Akcje

- Wstęp
- Typy komunikatów
- Deklaracje
- Procedury
- Akcje

TYPY KOMUNIKATÓW

```
type PACKET  extends FRAME  is record of
        detectNo  : INTEGER
        data      : MESSAGE
      end record

type TOKEN  extends FRAME  is record of
        initId    : PROCESS_ID
        detectNo  : INTEGER
        SRAccu    : INTEGER
        invalid   : BOOLEAN
      end record
```

Jednofazowy algorytm detekcji zakończenia dla modelu atomowego (deklaracje)

DEKLARACJE

msgIn	: MESSAGE
pcktOut	: PACKET
tokenOut	: TOKEN
$SRBalance_i$	: INTEGER := 0
$detectNo_i$	: INTEGER
$maxDetectNo_i$	: INTEGER := 0
$terminationDetected_i$	: BOOLEAN := *False*

Jednofazowy algorytm detekcji zakończenia dla modelu atomowego (akcje)

AKCJE

1. **when** e_start(Q_α, *TerminationDetection*) **do**
2. $detectNo_\alpha := detectNo_\alpha + 1$
3. $tokenOut.initId := Q_\alpha$
4. $tokenOut.detectNo := detectNo_\alpha$
5. $tokenOut.SRAccu := SRBalance_\alpha$
6. *tokenOut.invalid* := *False*
7. **send**(Q_α, $Q_{succ(\alpha)}$, *tokenOut*)
8. **end when**

9. **when** e_send(P_i, P_j, *msgOut* : MESSAGE) **do**
10. $SRBalance_i := SRBalance_i + 1$
11. $pcktOut.detectNo := detectNo_i$
12. *pcktOut.data* := *msgOut*
13. **send**(Q_i, Q_j, *pcktOut*)
14. **end when**

15. **when** e_receive(Q_j, Q_i, *pcktIn* : PACKET) **do**
16. $SRBalance_i := SRBalance_i - 1$
17. $maxDetectNo_i := \max(pcktIn.detectNo, maxDetectNo_i)$
18. *msgIn* := *pcktIn.data*
19. **deliver**(P_j, P_i, *msgIn*)
20. **end when**

21. **when** e_receive($Q_{pred(i)}$, Q_i, *tokenIn* : TOKEN) **do**
22. $detectNo_i := \max(detectNo_i, tokenIn.detectNo)$
23. **if** $tokenIn.initId = Q_i$ **then**
24. **if** $tokenIn.SRAccu = 0 \wedge \neg tokenIn.invalid$ **then**
25. $terminationDetected_i := True$
26. **decide**($terminationDetected_i$)
27. **else**
28. $detectNo_i := detectNo_i + 1$
29. $tokenOut.initId := Q_i$
30. $tokenOut.detectNo := detectNo_i$
31. $tokenOut.SRAccu := SRBalance_i$
32. *tokenOut.invalid* := *False*
33. **send**(Q_i, $Q_{succ(i)}$, *tokenOut*)
34. **end if**
35. **else**
36. $tokenOut.detectNo := detectNo_i$
37. $tokenOut.SRAccu := tokenIn.SRAccu + SRBalance_i$
38. $tokenOut.invalid := tokenIn.invalid \vee$
39. $(maxDetectNo_i \geqslant tokenIn.detectNo)$
40. $tokenOut.initId := tokenIn.initId$
41. **send**(Q_i, $Q_{succ(i)}$, *tokenOut*)
42. **end if**
43. **end when**

Wektorowy algorytm detekcji zakończenia (1)

- W algorytmie wektorowym wykorzystywany jest wektor liczników $vSRNo_i$ będący tablicą [1..n].
- Procesy nie komunikują się same ze sobą
- Monitory tworzą strukturę o topologii pierścienia.
- Cyrkulujący w pierścieniu znacznik typu `TOKEN` przenosi n-elementowy wektor $vSRAccu$ zawierający sumę wektorów $vSRNo_i$ monitorów odwiedzonych już w danym cyklu detekcji przez `TOKEN`.
- Elementy $vSRNo_i[j]$, dla $i \neq j$, każdego procesu P_i określają liczbę wiadomości wysłanych przez proces P_i do P_j od czasu ostatniej wizyty znacznika.
- Wartość bezwzględna elementu $vSRNo_i[i]$ określa liczbę wiadomości odebranych przez P_i od czasu ostatniej wizyty znacznika.
- W dowolnej chwili τ, suma k-tych elementów wszystkich liczników $vSRNo_i$, dla każdego $i \in \{1, 2, ..., n\}$, oraz wartości pozycji k wektora cyrkulującego znacznika $tokenIn.vSRAccu[k]$, jest równa liczbie wiadomości będących w drodze do P_k.

Wektorowy algorytm detekcji zakończenia (2)

- Inicjator Q_α rozpoczyna detekcję wysyłając znacznik typu TOKEN z wektorem $vSRAccu = 0$.
- Każdy monitor Q_i nalicza indywidualnie wiadomości aplikacyjne wysłane przez P_i, zwiększając licznik $vSRNo_i[j]$ na pozycji odpowiadającej procesowi przeznaczenia P_j.
- Z drugiej strony monitor Q_i zmniejsza licznik własny $vSRNo_i[i]$, gdy odebrana zostanie wiadomość aplikacyjna.
- Po otrzymaniu znacznika monitor Q_i sumuje własny wektor wiadomości $vSRNo$, z wektorem $vSRAccu$ zawartym w znaczniku. Warunek $vSRAccu_i[i] > 0$ oznacza, że istnieje wiadomość aplikacyjna uwzględniona już w znaczniku $vSRAccu$ jako wysłana do P_i, a która nie została jeszcze odebrana przez P_i. W tym wypadku znacznik jest *zatrzymywany* w monitorze Q_i (nie jest dalej wysyłany), a jego dalsze przesłanie jest uwarunkowane otrzymaniem odpowiedniej liczby wiadomości aplikacyjnych przez P_i.
- W całym przetwarzaniu, co najwyżej jeden z monitorów (procesów) ma licznik $vSRNo_i[i] > 0$.

Wektorowy algorytm detekcji zakończenia (3)

Ważnymi charakterystykami wektorowego algorytmu detekcji zakończenia są:

- wyeliminowanie konieczności dołączania etykiet do wiadomości aplikacyjnych,
- zatrzymywanie znacznika do momentu spełnienia koniecznych warunków zakończenia,
- wykonywanie algorytmu w sposób ciągły aż do momentu stwierdzenia zakończenia przetwarzania aplikacyjnego.

Wektorowy algorytm detekcji zakończenia dla modelu atomowego

Wektorowy algorytm detekcji zakończenia dla modelu atomowego (typy komunikatów)

➲ Wstęp
➲ Typy komunikatów
➲ Deklaracje
➲ Procedury
➲ Akcje

TYPY KOMUNIKATÓW

```
type PACKET extends FRAME  is record of
        data : MESSAGE
     end record

type TOKEN extends FRAME  is record of
        vSRAccu  : array [1..n] of INTEGER
        data     : MESSAGE
     end record
```


Wektorowy algorytm detekcji zakończenia dla modelu atomowego (deklaracje)

➲ Wstęp
➲ Typy komunikatów
➲ Deklaracje
➲ Procedury
➲ Akcje

DEKLARACJE

msgIn	: MESSAGE
pcktOut	: PACKET
tokenOut	: TOKEN
$vSRNo_i$	: **array** [1..*n*] **of** INTEGER := 0
$firstWave_i$	: BOOLEAN := *True*
$terminationDetected_i$	: BOOLEAN := *False*

Wektorowy algorytm detekcji zakończenia dla modelu atomowego (akcje)

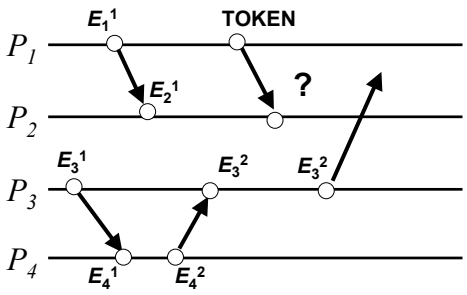
- Wstęp
- Typy komunikatów
- Deklaracje
- Procedury
- Akcje

AKCJE

```
1.  when e_start( Q_o, TerminationDetection ) do
2.    for all k ∈ {1, 2, ..., n } do
3.       tokenOut.vSRAccu[k] := 0
4.    end for
5.    send( Q_o, Q_succ(o), tokenOut)
6.    firstWave_i := False
7.  end when

8.  when e_receive( Q_j, Q_i, tokenIn : TOKEN) do
9.    vSRNo_i := vSRNo_i + tokenIn.vSRAccu
10.   if vSRNo_i[i] ⩽ 0
11.   then
12.     if ∀k :: k∈{1,2,...,n}:: vSRNo_i[k]=0 ∧ ¬firstWave_i
13.     then
14.       terminationDetected_i := True
15.       decide( terminationDetected_i )
16.     else
17.       tokenOut.vSRAccu := vSRNo_i
18.       send( Q_i, Q_succ(i), tokenOut)
19.       for all k ∈{1, 2, ..., n} do
20.                 vSRNo_i[k] := 0
21.       end for
22.       firstWave_i := False
23.     end if
24.   end if
25. end when

26. when e_send( P_i, P_j, msgOut : MESSAGE ) do
27.   vSRNo_i[j] := vSRNo_i[j] + 1
28.   pcktOut.data := msgOut
29.     send( Q_i, Q_j, pcktOut )
30. end when

31. when e_receive( Qj, Qi, pcktIn : PACKET ) do
32.     msgIn := pcktIn.data
33.     deliver( P_j, P_i, msgIn )
34.     vSRNo_i[j] := vSRNo_i[j] - 1
35.     if vSRNo_i[i] = 0 then
36.       if ∀k :: k∈{1,2,...,n}::
            vSRNo_i[k]=0 ∧ ¬firstWave_i then
37.         terminationDetected_i := True
38.         decide( terminationDetected_i )
39.       else
40.         tokenOut.vSRAccu := vSRNo_i
41.         send( Q_i, Q_succ(i), tokenOut)
42.         for all k ∈{1, 2, ..., n} do
43.           vSRNo_i[k] := 0
44.         end for
45.         firstWave_i := False
46.     end if
47.   end if
48. end when
```


Wektorowy algorytm detekcji zakończenia

$firstWave_i$



Algorytm detekcji zakończenia statycznego

- W algorytmie wykorzystano koncepcję ciągu cykli detekcyjnych, gdzie monitory są logicznie połączone w strukturę topologiczną gwiazdy.
- W celu stwierdzenia zakończenia statycznego przetwarzania aplikacyjnego określonego przez predykat

  $Sterm(\mathcal{P}) \equiv \forall P_i :: P_i \in \mathcal{P} :: (passive_i \wedge (\mathcal{IT}_i \neq \varnothing) \wedge \neg\, activate_i(\mathcal{AV}_i))$

  inicjator Q_α rozpoczyna detekcję przez wysłanie do wszystkich monitorów, w tym do siebie, wiadomości kontrolnej typu `QUERY`.
- W odpowiedzi monitory przesyłają wiadomości typu `REPLY` zawierające zmienną logiczną w polu *contPassive*.
- Po otrzymaniu wszystkich odpowiedzi, inicjator wyznacza iloczyn logiczny otrzymanych zmiennych *contPassive* jako wartość zmiennej $sTermDetected_\alpha$.
  - Jeżeli $sTermDetected_\alpha$ przyjmuje wartość *True*, to Q_α stwierdza zakończenie
  - W przeciwnym przypadku inicjator wysyła ponownie zapytanie `QUERY`

Algorytm detekcji zakończenia statycznego

Aby uniknąć pominięcia wiadomości aplikacyjnej, która została wysłana przed otrzymaniem zapytania `QUERY` przez pewien monitor Q_i, a odebrana po otrzymaniu zapytania przez inny monitor Q_k – wprowadzona jest zmienna logiczna $contPassive_i$, która jest inicjowana wartością *True*.

Za każdym razem, gdy P_i jest uaktywniany, $contPassive_i$ przyjmuje wartość *False*. Z kolei wysyłając wiadomość kontrolną `REPLY` monitory nadają zmiennej $contPassive_i$ ponownie wartość *True*. Wartość *True* zmiennej $contPassive_i$ w danej chwili oznacza, że proces P_i był pasywny przez cały czas od chwili wysłania ostatniej wiadomości `REPLY`.

Detekcja zakończenia statycznego

- Wstęp
- Typy komunikatów
- Deklaracje
- Procedury
- Akcje

- Wstęp
- Typy komunikatów
- Deklaracje
- Procedury
- Akcje

TYPY KOMUNIKATÓW

```
type PACKET  extends FRAME  is record of
        data : MESSAGE
      end record

type REPLY extends FRAME  is record of
        contPassive  : BOOLEAN
      end record

type QUERY  extends SIGNAL

type ACK  extends SIGNAL
```

Detekcja zakończenia statycznego (deklaracje)

➲ Wstęp
➲ Typy komunikatów
➲ Deklaracje
➲ Procedury
➲ Akcje

DEKLARACJE

msgIn	: MESSAGE
pcktOut	: PACKET
queryOut	: QUERY
replyOut, replyIn	: REPLY
ackOut	: ACK
$contPassive_i$	: BOOLEAN := *True*
$sTermDetected_i$	: BOOLEAN := *False*
$\mathcal{AV}_i$	: **set of** PROCESS_ID
$notAck_i$	: INTEGER := 0
$terminationDetected_i$	: BOOLEAN := *False*

Detekcja zakończenia statycznego (akcje)

➲ Wstęp
➲ Typy komunikatów
➲ Deklaracje
➲ Procedury
➲ Akcje

AKCJE

1. **when** e_start(Q_α, *TerminationDetection*) **do**
2. $sTermDetected_\alpha$:= False
3. **repeat**
4. **send**(Q_α, $\mathcal{Q}$, *queryOut*)
5. **for all** $Q_j \in \mathcal{Q}$ **do**
6. **receive**(Q_j, Q_α, *replyIn*)
7. $sTermDetected_\alpha$:= $sTermDetected_\alpha \wedge replyIn.contPassive$
8. **end for**
9. **until** $sTermDetected_\alpha = True$
10. $terminationDetected_\alpha := True$
11. **decide**($terminationDetected_\alpha$)
12. **end when**

13. **when** e_receive(Q_α, Q_i, *queryIn* : QUERY) **do**
14. **wait until** $passive_i \wedge notAck_i = 0 \wedge \neg activate_i(\mathcal{AV}_i)$
15. $replyOut.contPassive := contPassive_i$
16. $contPassive_i := True$
17. **send**(Q_i, Q_α, *replyOut*)
18. **end when**

19. **when** e_receive(Q_j, Q_i, :*ackIn* : ACK) **do**
20. $notAck_i := notAck_i - 1$
21. **end when**

22. **when** e_send(P_i, P_j, *msgOut* : MESSAGE) **do**
23. $notAck_i := notAck_i + 1$
24. $pcktOut.data := msgOut$
25. **send**(Q_i, Q_j, *pcktOut*)
26. **end when**

27. **when** e_receive(Q_j, Q_i, *pcktIn* : PACKET) **do**
28. **send**(Q_i, Q_j, *ackOut*)
29. $msgIn := pcktIn.data$
30. **deliver**(P_j, P_i, *msgIn*)
31. **end when**

32. **when** e_activate(P_i) **do**
33. $contPassive_i := False$
34. **end when**

Algorytm detekcji zakończenia statycznego

Niech:

- ❑ τ^k_i – czas globalny wysłania odpowiedzi przez monitor Q_i w k-tym cyklu detekcji.
- ❑ τ_b^k – czas globalny rozpoczęcia k-tego cyklu detekcji
- ❑ τ_e^k – czas globalny zakończenia k-tego cyklu detekcji
- ❑ $\mathcal{X}[\tau_i^k]$ – jest wartością zmiennej (predykatu) w chwili τ^k_i
- ❑ Q_α – inicjator jest dodatkowym procesem, nie związanym z procesami aplikacyjnymi

Zachodzi:

$$\tau_b^k < \tau_i^k < \tau_e^k$$

Jeżeli algorytm detekcji zakończenia statycznego jest wykonywany w chwili, gdy przetwarzanie aplikacyjne osiągnęło stan zakończenia statycznego, to algorytm ten zakończy się w skończonym czasie, stwierdzając wystąpienie zakończenia statycznego przetwarzania aplikacyjnego ($sTerminationDetected_\alpha = True$).

(c) Zakład Systemów Informatycznych Slajd 67

Algorytm detekcji zakończenia statycznego

Dowód

Jeżeli przetwarzanie aplikacyjne osiągnęło stan zakończenia w chwili τ^t, to zgodnie z definicją, od tego momentu wszystkie procesy będą zawsze pasywne ($passive_i = True$), ich warunki uaktywnienia nie będą spełnione ($activate_i(\mathcal{AV}_i) = False$), a ponadto w kanałach nie będzie znajdowała się żadna wiadomość aplikacyjna ($\mathcal{IT}_i = \varnothing$).

Stąd, w skończonym czasie zostaną w systemie odebrane potwierdzenia od monitorów, i liczniki $notAck_i$ przyjmą wartość 0. Oznaczmy tę chwilę przez τ^x. Oczywiście $\tau^x > \tau^t$. Od momentu τ^x bieżący cykl detekcyjny nie będzie z pewnością wstrzymywany przez monitory.

Zauważmy jednak, że pewne monitory w tym cyklu mogły wysłać odpowiedzi `REPLY` przed momentem τ^t. Stąd, dopiero następny cykl detekcyjny z pewnością przypisze już na stałe wszystkim zmiennym $contPassive_i$ wartość *True*.

W konsekwencji, przez cały kolejny cykl detekcyjny k+1, wszystkie zmienne logiczne $contPassive_i$ będą miały wartość *True* dla każdego $i \in \{1,2,...n\}$. Po odebraniu zatem wiadomości `REPLY` i przyjęciu przez $sTerminationDetected_\alpha$ wartości *True*, algorytm zakończy się.

(c) Zakład Systemów Informatycznych Slajd 68

Algorytm detekcji zakończenia statycznego

Jeżeli algorytm stwierdza wystąpienie zakończenia statycznego przetwarzania aplikacyjnego, to przetwarzanie aplikacyjne jest w istocie w stanie zakończenia statycznego.

Dowód

Rozważmy dwa kolejne cykle detekcyjne k oraz k+1. Załóżmy, że algorytm stwierdza zakończenie przetwarzania aplikacyjnego kończąc cykl k+1, a więc w chwili τ_e^{k+1}. Wówczas $sTerminationDetected_\alpha = True$.

Ponieważ cykle są inicjowane sekwencyjnie, istnieje taki moment τ^x, że:

$$\tau_b^k < \tau_e^k \leqslant \tau^x \leqslant \tau_b^{k+1} < \tau_e^{k+1}$$

Udowodnimy, że jeżeli $sTerminationDetected_\alpha[\tau_e^{k+1}] = True$, to w chwili $\tau = \tau^x$ przetwarzanie aplikacyjne jest statycznie zakończone, a więc:

$$Sterm(\mathcal{P})[\tau^x] = True.$$

Innymi słowy wykażemy, że dla wszystkich procesów spełnione będą następujące warunki:

C1. $passive_i[\tau^x] = True$

C2. $\mathcal{IT}_i[\tau^x] = \varnothing$.

C3. $activate(\mathcal{AV}_i)[\tau^x] = False$.

Algorytm detekcji zakończenia statycznego

Dowód warunku C1

Z konstrukcji algorytmu wynika, że zakończy się on w chwili τ_e^{k+1} z $sTerminationDetected[\tau_e^{k+1}] = True$, tylko wówczas, gdy każdy proces P_i był pasywny w czasie między τ_i^k a τ_i^{k+1}. Skoro jednak $\tau_i^k \leqslant \tau^x \leqslant \tau_i^{k+1}$, więc możemy zatem wnosić, że dla każdego $P_i \in \mathcal{P}$, $passive_i[\tau^x] = True$.

Dowód warunku C2

Zgodnie z konstrukcją algorytmu, w każdym cyklu detekcyjnym odpowiedź `REPLY` jest wstrzymywana aż do momentu nadejścia potwierdzeń od monitorów procesów przeznaczenia, dla wszystkich wysłanych wcześniej wiadomości aplikacyjnych.

Wówczas $notAck_i$, przyjmie wartość 0. Z drugiej strony można jednak zauważyć, że od chwili τ_i^k do τ_i^{k+1} proces P_i jest ciągle pasywny, a więc nie wysyła wiadomości. Stąd więc wnosimy, że dla każdego procesu $P_i \in \mathcal{P}$, $\mathcal{IT}_i[\tau^x] = \varnothing$.

Algorytm detekcji zakończenia statycznego

Dowód warunku C3

Z konstrukcji algorytmu wynika, że w każdym cyklu detekcyjnym odpowiedź jest opóźniana do momentu uzyskania przez predykat $\neg activate(\mathcal{AV}_i)$ wartości *True*. Skoro algorytm kończy się w chwili τ_e^{k+1}, to predykat $activate(\mathcal{AV}_i)$ $[\tau_i^{k+1}]$ miał wartość *False*. Ponieważ w przedziale $\langle \tau_i^k, \tau_i^{k+1} \rangle$ proces P_i był przez cały czas pasywny, żadna wiadomość nie została w tym czasie odebrana. Stąd też, w przedziale tym do zbioru $\mathcal{AV}_i$ mogły być co najwyżej dołączone dodatkowe wiadomości. W efekcie:

$$\mathcal{AV}_i[\tau^x] \subseteq \mathcal{AV}_i[\tau_i^{k+1}]$$

Z własności monotoniczności predykatu $activate_i$ możemy przy tym wnioskować, że skoro zachodzi $\neg activate_i(\mathcal{AV}_i)[\tau_i^{k+1}]$ to również zachodzi $\neg activate(\mathcal{AV}_i)[\tau^x]$

Wykazaliśmy zatem, że w chwili τ^x, dla każdego procesu $P_i \in \mathcal{P}$ spełnione są warunki C1, C2 i C3, a więc $Sterm(\mathcal{P})[\tau^x] = True$. Ponieważ predykat ten jest stabilny, więc również dla każdego $\tau > \tau^x$, w tym również dla τ_e^{k+1}, $Sterm(\mathcal{P})[\tau^x]=True$.

Algorytm detekcji zakończenia statycznego

Efektywność, a także złożoność czasowa i komunikacyjna, algorytmu zależy od implementacji cykli detekcyjnych i topologii przetwarzania monitorującego.

W celu wyznaczenia złożoności czasowej zauważmy, że po wystąpieniu zakończenia statycznego, dla jego stwierdzenia potrzebne są w najgorszym przypadku dwa pełne cykle detekcyjne oraz musi być dokończony cykl bieżący.

W każdym cyklu przesyłanych jest n wiadomości typu `QUERY` i n wiadomości typu `REPLY` przenoszących jednobitową zmienną. Tak więc zaniedbując wiadomości potwierdzeń (których liczba jest równa liczbie wiadomości aplikacyjnych), otrzymujemy liczbę wiadomości kontrolnych wystarczających do stwierdzenia zakończenia, równą: $3 \times 2n = 6n$.

Przyjmijmy teraz, że struktura topologiczna przetwarzania detekcyjnego jest grafem w pełni połączonym. Pomijając ponownie wiadomości potwierdzeń, łatwo już zauważyć, że złożoność czasowa algorytmu wynosi 3. Oczywiście przy innych topologiach złożoność ta zmienia się odpowiednio. Przykładowo, w przypadku pierścienia złożoność czasowa wyniesie $3n$.

Algorytm detekcji zakończenia dynamicznego

W celu wyznaczenia wartości predykatów
$Dterm(\mathcal{P}) \equiv \forall P_i :: P_i \in \mathcal{P} :: (passive_i \wedge \neg\, activate_i(\mathcal{AV}_i \cup \mathcal{IT}_i))$
zaproponowano algorytm, który stosuje mechanizm cykli detekcyjnych oraz wiadomości kontrolne typu `QUERY` i `REPLY`. Monitory Q_i przetwarzania detekcyjnego posiadają następujące zmienne lokalne, wykorzystywane w procesie detekcji:

- $contPassive_i$ – zmienna logiczna, której wartość *True* oznacza, że proces P_i pozostawał pasywny od momentu wysłania ostatniej wiadomości typu `REPLY`
- $vSentNo_i$ – tablica [1..*n*] liczników, w której element $vSentNo_i[j]$oznacza liczbę wiadomości wysłanych przez P_i do P_j;
- $vRecvNo_i$ – tablica [1..*n*] liczników, w której element $vRecvNo_i[j]$ oznacza liczbę wiadomości odebranych przez P_i od procesu P_j.

Ponadto, inicjator Q_α przechowuje zmienną $tSentNo_\alpha$ będącą tablicą [1..*n*, 1..*n*]. Wiersz *i* tej tablicy zawiera wektor $vSentNo_i$ przesłany ostatnio do inicjatora Q_α przez monitor Q_i, w wiadomości kontrolnej typu `REPLY`. W efekcie, każdy element tablicy $tSentNo_\alpha$ reprezentuje wiedzę inicjatora, o liczbie wiadomości aplikacyjnych wysłanych z P_i do P_j. Zauważmy, że wiedza ta nie jest precyzyjna, gdyż nadesłane wektory $vSentNo_i$, $i \in \{1,2,...,n\}$, odnoszą się w ogólności do różnych momentów czasu.

Algorytm detekcji zakończenia dynamicznego

- Inicjator detekcji Q_α rozpoczyna cykl detekcyjny wysyłając do wszystkich monitorów Q_i wiadomość kontrolną typu `QUERY`, zawierającą w polu *vSentNo* odpowiednią kolumnę tablicy $tSentNo_\alpha$. Wektor ten informuje odbiorcę Q_i o znanej inicjatorowi Q_α liczbie wiadomości wysłanych do P_i przez inne procesy.
- Po otrzymaniu zapytania typu `QUERY`, monitor Q_i może wyznaczyć zbiór $\mathcal{AIT}_i$ będący aproksymacją zbioru tych procesów, których wiadomości wysłane do P_i są potencjalnie jeszcze w kanałach. (porównywany jest wektor *queryIn.vSentNo* odebrany przez Q_i z $vRecvNo_i$ – jeżeli $queryIn.vSentNo[j] > vRecvNo_i[j]$, to w kanale $C_{j,i}$ znajdą się wiadomości wysłane przez P_j do P_i, a nie odebrane jeszcze przez P_i)
- Każdy z monitorów wyznacza wartość zmiennej *replyOut.contPassive*.(*replyOut.contPassive* = *True* wtedy i tylko wtedy, gdy $contPassive_i$ = *True*, a predykat $activate_i(\mathcal{AV}_i \cup \mathcal{AIT}_i)$ = *False*.
- Po wyznaczeniu wartości *replyOut.contPassive* monitor wysyła odpowiedź typu `REPLY` zawierającą pole *contPassive* oraz aktualny wektor $vSentNo_i$, który będzie użyty przez Q_α do uaktualnienia tablicy $tSentNo_\alpha$.
- Algorytm stwierdza wykrycie zakończenia dynamicznego przetwarzania aplikacyjnego, gdy wszystkie odebrane w danym cyklu przez Q_α zmienne *replyIn.contPassive* mają wartość *True*. W przeciwnym wypadku, Q_α rozpoczyna następny cykl detekcyjny.

Detekcja zakończenia dynamicznego

- Wstęp
- Typy komunikatów
- Deklaracje
- Procedury
- Akcje

- Wstęp
- Typy komunikatów
- Deklaracje
- Procedury
- Akcje

TYPY KOMUNIKATÓW

```
type PACKET extends FRAME is record of
        data        : MESSAGE
     end record

type QUERY extends FRAME is record of
        vSentNo     : array [1..n] of INTEGER
     end record

type REPLY extends FRAME is record of
        contPassive : BOOLEAN
        vSentNo     : array [1..n] of INTEGER
     end record
```

Detekcja zakończenia dynamicznego (deklaracje)

➲ Wstęp
➲ Typy komunikatów
➲ Deklaracje
➲ Procedury
➲ Akcje

DEKLARACJE

msgIn	: MESSAGE
pcktOut	: PACKET
queryOut	: QUERY
replyOut, replyIn	: REPLY
$tSentNo_i$	: array [1..*n*, 1..*n*] of INTEGER := 0
$vSentNo_i$	: array [1..*n*] of INTEGER := 0
$vRecvNo_i$	: array [1..*n*] of INTEGER := 0
$\mathcal{AIT}_i$	: set of PROCESS_ID
$\mathcal{AV}_i$	: set of PROCESS_ID
$contPassive_i$	: BOOLEAN := *True*
$dTermDetected_i$	: BOOLEAN := *False*
$terminationDetected_i$	: BOOLEAN := *False*

Detekcja zakończenia dynamicznego (akcje)

➲ Wstęp
➲ Typy komunikatów
➲ Deklaracje
➲ Procedury
➲ Akcje

AKCJE

1. **when** e_start(Q_α, *TerminationDetection*) **do**
2. $dTermDetected_\alpha$:= False
3. **repeat**
4. **for all** $Q_j \in \mathcal{Q}$ **do**
5. **for all** $k \in \{ 1, 2, ..., n \}$ **do**
6. $queryOut.vSentNo[k] := tSentNo_\alpha[k, j]$
7. **end for**
8. **send**(Q_α, Q_j, *queryOut*)
9. **end for**
10. **for all** $Q_j \in \mathcal{Q}$ **do**
11. **receive**(Q_j, Q_α, *replyIn*)
12. **for all** $k \in \{ 1, 2, ..., n \}$ **do**
13. $tSentNo_\alpha[j,k] := replyIn.vSentNo[k]$
14. **end for**
15. $dTermDetected_\alpha := dTermDetected_\alpha \wedge replyIn.contPassive$
16. **end for**
17. **endrepeat** $dTermDetected_\alpha = True$
18. $terminationDetected_\alpha := True$
19. **decide**($terminationDetected_\alpha$)
20. **end when**

21. **when** e_receive(Q_α, Q_i, *queryIn* : QUERY) **do**
22. $\mathcal{AIT}_i := \{P_j :: queryIn.vSentNo[j] > vRecvNo_i[j]\}$
23. $replyOut.contPassive := contPassive_i \wedge \neg activate_i(\mathcal{AV}_i \cup \mathcal{AIT}_i)$
24. $contPassive_i := passive_i$
25. $replyOut.vSentNo := vSentNo_i$
26. **send**(Q_i, Q_α, *replyOut*)
27. **end when**

28. **when** e_send(P_i, P_j, *msgOut* : MESSAGE) **do**
29. $vSentNo_i[j] := vSentNo_i[j] + 1$
30. $pcktOut.data := msgOut$
31. send(Q_i, Q_j, *pcktOut*)
32. **end when**

33. **when** e_receive(Q_j, Q_i, *pcktIn* : PACKET) **do**
34. $vRecvNo_i[j] := vRecvNo_i[j] + 1$
35. $msgIn := pcktIn.data$
36. deliver(P_j, P_i, *msgIn*)
37. **end when**

38. **when** e_activate(P_i) **do**
39. $contPassive_i := False$
40. **end when**

Algorytm detekcji zakończenia dynamicznego

Jeżeli algorytm jest wykonywany w chwili, gdy przetwarzanie aplikacyjne osiągnęło stan zakończenia dynamicznego, to algorytm ten zakończy się w skończonym czasie, stwierdzając wystąpienie dynamicznego zakończenia przetwarzania aplikacyjnego ($dTerminationDetected_{\alpha} = True$).

Algorytm detekcji zakończenia dynamicznego

Dowód

Cykl detekcyjny nie jest wstrzymywany przez monitory, a więc inicjator otrzyma odpowiedź od każdego Q_i w skończonym czasie.

Zakładamy, że przetwarzanie aplikacyjne osiągnęło stan zakończenia dynamicznego w chwili τ^t ($Dterm(\mathcal{P})[\tau^t] = True$). Od tego momentu zatem, wszystkie procesy pozostają ciągle pasywne:

$passive_i[\tau^x]$ i $activate_i(\mathcal{AV}_i \cup \mathcal{IT}_i)[\tau^x] = False$, dla każdego $\tau^x \geqslant \tau^x$.

Tak więc każdy cykl zainicjowany po τ^t, powiedzmy cykl k, nada wszystkim zmiennym $contPassive_i$ wartość *True*, i odbierze od monitorów Q_i wektor końcowych wartości liczników $vSentNo_i$. Dlatego, w chwili τ_b^{k+1} rozpoczęcia kolejnego cyklu detekcyjnego, wysłane zostaną ostateczne wartości liczników wiadomości wysłanych, a stąd $\mathcal{AIT}_i[\tau_i^{k+1}] = \mathcal{IT}_i[\tau_i^{k+1}]$. W efekcie otrzymujemy, że:

$activate_i(\mathcal{AV}_i \cup \mathcal{AIT}_i)[\tau_i^{k+1}] = activate_i(\mathcal{AV}_i \cup \mathcal{IT}_i)[\tau_i^{k+1}]$

Ponieważ ten ostatni predykat ma wartość *False* od chwili τ^t, a $\tau^t \leqslant \tau_i^k < \tau_i^{k+1}$, więc w cyklu k+1 wszystkie zmienne *replyIn.contPassive* będą miały wartość *True*, i w efekcie algorytm detekcji dynamicznego zakończenia przetwarzania aplikacyjnego, zakończy się. □

Algorytm detekcji zakończenia dynamicznego

Jeżeli algorytm stwierdza wystąpienie dynamicznego zakończenia przetwarzania rozproszonego, to przetwarzanie aplikacyjne jest w istocie w stanie zakończenia dynamicznego.

Dowód

Oznaczymy dwa ostatnie cykle detekcyjne odpowiednio przez k i k+1, a przez τ^x taki moment, że:

$$\tau_b^k < \tau_e^k \leqslant \tau^x \leqslant \tau_b^{k+1} < \tau_e^{k+1}$$

W dalszej części dowodu pokażemy, że jeżeli algorytm zakończy się w chwili τ_e^{k+1} z $dTerminationDetected_\alpha[\tau_e^{k+1}] = True$, to $Dterm(\mathcal{P})[\tau^t] = True$. Oznacza to dalej, że są spełnione następujące dwa warunki dla każdego Q_i, $i \in \{1,2,...n\}$,

C4. $passive_i[\tau^x] = True$
C5. $activate_i(\mathcal{AV}_i \cup \mathcal{IT}_i)[\tau^x] = False$

Algorytm detekcji zakończenia dynamicznego

Dowód warunku C5

Z konstrukcji algorytmu wynika, że algorytm ten zakończy się w chwili τ_e^{k+1} pod warunkiem, że predykat $activate_i(\mathcal{AV}_i \cup \mathcal{AIT}_i)$ miał wartość *False* w chwili τ_e^{k+1} dla każdego $i \in \{1,2,...n\}$.

Ponieważ każdy proces był pasywny w przedziale $\langle \tau_i^k, \tau_i^{k+1} \rangle$ dla każdego zachodzi:

C6. $\mathcal{AV}_i[\tau^x] \subseteq \mathcal{AV}_i[\tau_i^{k+1}]$
oraz $vSentNo_i[j][\tau_i^k] = vSentNo_i[j][\tau^x]$

Wyznaczając zbiór $\mathcal{AIT}_i[\tau_i^{k+1}]$ procesów, od których wiadomości są aktualnie transmitowane do P_i, algorytm uwzględnia w istocie następujące zależności:

$$vSentNo_j[i][\tau_j^k] \leqslant vRecvNo_i[j][\tau_i^{k+1}], \text{ gdyż}$$
$$vSentNo_j[i][\tau_j^k] = gueryIn.vSentNo[j][\tau_i^{k+1}].$$

Zauważmy, że liczniki są monotoniczne:

$$vRecvNo_i[j][\tau_i^k] \leqslant vRecvNo_i[j][\tau^x] \leqslant vRecvNo_i[j][\tau_i^{k+1}]$$

Algorytm detekcji zakończenia dynamicznego

Dowód warunku C5 (c.d.)

Ponieważ dla każdego Q_i, $vSentNo_i[j]$ $[\tau_i^k] = vSentNo_i[j][\tau^x]$ różnica między rzeczywistym zbiorem $\mathcal{IT}_i[\tau^x]$ a jego aproksymacją $\mathcal{AIT}_i[\tau_i^{k+1}]$, może wynikać tylko z dotarcia pewnych wiadomości do procesów docelowych. Wówczas jednak odpowiedni nadawcy zostaną uwzględnieni w zbiorze $\mathcal{AV}_i$. Tak więc, biorąc pod uwagę C6 możemy wnosić, że:

$$(\mathcal{AV}_i \cup \mathcal{IT}_i)[\tau^x] \subseteq (\mathcal{AV}_i \cup \mathcal{AIT}_i)[\tau_i^{k+1}].$$

Dalej, z monotoniczności predykatu $activate_i$ wynika, że dla każdego Q_i:

$$\neg activate_i(\mathcal{AV}_i \cup \mathcal{AIT}_i)[\tau_i^{k+1}] \Rightarrow \neg activate_i(\mathcal{AV}_i \cup \mathcal{IT}_i)[\tau^x]$$

Oznacza to, że warunek C5 jest spełniony.

Algorytm detekcji zakończenia dynamicznego

- Aby wykryć wystąpienie zakończenia dynamicznego, niezbędne są w najgorszym wypadku dwa cykle detekcyjne po zakończeniu cyklu bieżącego.
- Ponieważ nie są przesyłane potwierdzenia, więc złożoność komunikacyjna algorytmu wynosi $4n$.
- Wiadomości kontrolne w tym algorytmie są bardziej złożone, gdyż zawierają wektory liczników.
- Opóźnienie detekcji jest mniejsze, ze względu na brak konieczności oczekiwania na potwierdzenia. Ponadto, wystąpienie zakończenia dynamicznego wyprzedza w czasie wystąpienie zakończenia statycznego. Uzyskane w sumie przyspieszenie momentu wykrycia zakończenia obliczeń rozproszonych może zatem mieć istotne znaczenie praktyczne, zwłaszcza w zastosowaniach czasu rzeczywistego.