ЛЕКЦИЯ 8
РЕГИОНИ И ПОДВИЖНИ КОМПОНЕНТИ

- **Модела Опашка**
- **Транспортни модели**
- **Стратегии**

ВЪВЕДЕНИЕ

Елементите на модел, които дефинират състояние на основните компоненти могат да се разделят на следните 4 групи:

- **Моделни променливи;**
- **Случайни променливи;**
- **Индикатори;**
- **Региони.**

ВЪВЕДЕНИЕ

Регионите са места за съхранение на подвижни компоненти. Подвижна компонента е компонента, която има декларационна част, но няма част на динамичното поведение. Свойствата на подвижна компонента могат да се променят само външно от основна компонента.

Примери за подвижни компоненти:

- Клиенти;
- Обработвани детайли;
- Палети;
- Превозни средства;
- Задачи;
- Съобщения.

ВЪВЕДЕНИЕ

Регионът може да съдържа една или повече подвижни компоненти. Компонентите се подреждат по специфициран критерий (първи влязъл – пръв излязъл, последен влязъл – първи излязъл и т.н.). По този начин регионът може да се използва в модела, за да моделира обикновено обслужване и опашка. Simplex3 прави статистика върху съдържанието на регион, което е достъпно за потребителя.

Пример за региони:

- Буфери;
- Съхраняващи области;
- Обработващи региони;
- Маршрутни участъци за превозни средства;
- Места за сядане в превозни средства;
- Опашки за клиенти;
- Пощенски кутии за съобщения.

ВЪВЕДЕНИЕ

Преместването на подвижните компоненти и изменението на атрибутите им е възможно само в събития, специфицирани в основните компоненти.

Подвижните компоненти могат да се създават, местят и унищожават, и техните свойства могат да се модифицират.

Примери:

- Регионът с име Queue получават нова подвижна компонента от класа Customer.

Queue^ : ADD 1 New Customer; #създаване

- Подвижна компонента от клас Customer се изтрива от региона, именуван Server и се унищожава. В този случай компонентата се поставя на второ място.

Server^ : REMOVE Customer [2]; #унищожаване

ВЪВЕДЕНИЕ

- Първата подвижна компонента, която е в регион Queue се премества в регион Server и се нарежда в опашката.

Server^ : FROM Queue GET Customer [1]; #преместване

- Първата подвижна компонента от регион Queue се изпраща в регион Server

Queue^ : TO Server SEND Customer [1]; # преместване

- Всички компоненти от клас Customer имат атрибут, наречен Tdone. Тук се присвоява крайното време за обработка. Атрибутът Tdone на първата компонента в регион Server получава стойност T +10.

Server : Customer [1].Tdone^ : = T +10; #промяна на атрибут

ВЪВЕДЕНИЕ

- Когато първата подвижна компонента се премести от регион Queue в регион Server, нейните атрибути TDone получават нова стойност.

```
Server^ : FROM Queue GET Customer [1]
                CHANGING
                    TDone^ : = T +Tserver; #промяна на атрибут
END
```

- Всички подвижни компоненти, които са в регион Server се унищожават и се заменят от 3 нови подвижни компоненти.

```
Server^ : PLACE 3 New Customer; #унищожаване и поставяне на нова стойност
```

Модела Опашка

Как се декларират регионите и как се създават подвижните компоненти?

Какви стратегии за подреждане в опашка са популярни?

Модела Опашка

Сървъри и опашки.

Ще създадем модел на M/M/1 опашка, състоящ се от източник, опашка, сървър и контейнер. Разликата се състои във факта, че в текущия модел Queue задачите имат атрибути и затова трябва да се моделират чрез подвижни компоненти. Това не беше необходимо в модела QueueD. Тъй като всички задачи в модела QueueD бяха идентични беше достатъчно просто да се приложи брояч, който съответно беше увеличаван и намаляван.

Модела Опашка

- Източника генерира задачи през експоненциално разпределени интервали време със средна стойност 15 времеви единици. Тези задачи са разпределени в 3 равномерно разпределени приоритетни класа 1, 2 и 3, като приоритет 3 е най-висок.
- Така интегрираните задачи влизат в опашката чрез стратегията PFIFO. Това означава, че те са сортирани в намаляващ ред на приоритет. Задачите с равен приоритет са подредени съобразно тяхното време на пристигане, т.е. чрез стратегията FIFO.
- Сървърът има 1 обслужваща станция, която може да обработва само една задача в един момент. Обслужващото време е експоненциално разпределено със средна стойност 10 времеви единици.

Модела Опашка

BASIC COMPONENT Queue1

MOBILE SUBCOMPONENTS OF CLASS Customer1

DECLARATION OF ELEMENTS

STATE VARIABLES
DISCRETE
  TArrive (REAL)   := 0,
  TWork   (REAL)   := 0,
  Protocol (LOGICAL) := FALSE

Модела Опашка

```
RANDOM VARIABLES
   Arrive (REAL)      : EXPO (Mean := 15),
   Work (REAL)        : EXPO (Mean := 10),
   Prio    (INTEGER)  : IUNIFORM (LowLimit:=1, UpLimit:=3)

LOCATIONS
   WaitP   (Customer1 ORDERED BY DEC Priority) := 0
    Customer1,
   Station (Customer1)                          := 0 Customer1,
   Sink    (Customer1)                         := 0 Customer1

END OF Queue1
```

Модела Опашка

```
DYNAMIC BEHAVIOUR
# Създаване на задача
WHENEVER T >= TArrive
DO
   WaitP^ : ADD 1 NEW Customer1
   CHANGING
     Priority^ := Prio;
   END
   TArrive^ := T + Arrive;
   IF Protocol
   DO DISPLAY ("T= %f New Customer \n",T); END
END
```

Модела Опашка

```
# Начало на обработката
WHENEVER (NUMBER(Station)=0) AND
   (NUMBER(WaitP)>0)
DO
   Station^ : FROM WaitP GET Customer1[1];
   TWork^ := T + Work;
   IF Protocol
   DO DISPLAY ("T= %f Customer enters Station
   \n",T); END
END
```

Модела Опашка

```
# Край на обработката
WHENEVER (T >= TWork) AND
   (NUMBER(Station)=1)
DO
   Station^ : TO Sink SEND Customer1[1];
   IF Protocol
   DO DISPLAY ("T= %f Customer leaves Station
    \n",T); END
END
```

Модела Опашка

```
# Унищожаване на задача
WHENEVER NUMBER(Sink) >= 4
DO
   Sink^ : REMOVE Customer1{ALL};
END

END OF Queue1
```

Модела Опашка

След обслужване задачите напускат обслужващия пункт и влизат в контейнера. Когато контейнерът съдържа 4 задачи те се унищожават.

Моделът Queue съдържа следните преходи между състояния:

Модела Опашка

Преход между състояния	Условие
Създаване на задача	Достигнато е времето на пристигане
Задача се премества от опашката към обслужващия пункт Започва обработката на работата	Станцията е празна и опашката съдържа поне 1 задача
Задача се изтрива от обслужващия пункт	Станцията се заема и времето за обслужване приключва
4 задачи са унищожени	Контейнерът съдържа 4 задачи

Модела Опашка

Подвижната компонента Customer1

MOBILE COMPONENT Customer1

DECLARATION OF ELEMENTS

STATE VARIABLES
DISCRETE
Priority (INTEGER) := 1 # Priority

END OF Customer1

Модела Опашка

Деклариране на региони.

Регионите принадлежат на елементите на модела. Те трябва да се декларират в секцията DECLARATION OF ELEMENTS във всички основни компоненти, в които се появяват.

declarations_of_elements ::= DECLARATION OF ELEMENTS
[list_of_constants]
[list_of_state_variables]
[list_of_dependent_variables]
[list_of_sensor_variables]
[list_of_random_variables]
[list_of_transition_indicators]
[list_of_sensor_indicators]
[list_of_locations]

Модела Опашка

[list_of_sensor_locations]

list_of_locations ::= LOCATION[S]
location {‘,’ location}

location ::= dim_identifier ‘(‘ identifier
[ordering_criteria_list] ‘)’
‘:=’ unsigned_number identifier

ordering_criteria_list ::= ORDERED BY ordering_criterion
{‘,’ ordering_criterion}

ordering_criterion ::= FIFO | LIFO
| INC identifier
|;DEC identifier

Модела Опашка

Всеки регион си има име.

След това се дава името на класа на подвижните компоненти, които региона може да съдържа. Следва списък на възможностите на една или повече стратегии за подреждане.

Възможните стратегии за подреждане са:

FIFO: Първи добавен – първи излиза;

LIFO: Последен добавен – първи излиза;

INC идентификатор: нарастващ атрибут;

DEC идентификатор: намаляващ атрибут.

Ако не е дадена стратегия за подреждане, тогава се приема FIFO.

Модела Опашка

Критериите INC и DEC се отнасят за атрибути на подвижни компоненти, които ще бъдат намерени в региона.

Ако първия критерий не дава уникално място за пристигане на подвижната компонента, тогава се използва следващия критерий в списъка, за да стесни възможностите. Списъкът може да съдържа до пет критерия. Ако дори след прилагането на петия критерий няма уникално решение, тогава се използва FIFO.

Критериите FIFO и LIFO могат да се появят само в края на списъка.

Подобно на променливите на състоянията, регионите трябва да се инициализират. За да се постигне това се използва присвояващия оператор, последван от броя на подвижните компоненти. Този брой може да е нула.

Модела Опашка

Примери:

- Регионът Server може да съхранява подвижни компоненти на клас Customer и се инициализира с един Customer.

LOCATION Server (Customer) : = 1 Customer

- Регионът Queue съхранява подвижните компоненти на клас Customer в намаляващ ред на атрибута си Priority. Подвижните компоненти с по-висок приоритет се поместват по-напред в опашката. Ако двама или повече клиенти имат еднакъв приоритет те се сортират според FIFO критерия.

LOCATION Queue (Customer ORDERED BY DEC Priority) : = 0 Customer

Модела Опашка

- Регионът Queue съдържа подвижните компоненти на клас Customer в намаляващ ред на атрибута Priority и тогава за клиентите с еднакъв приоритет, нарастващ ред на атрибута Age. Клиентите с равен приоритет и възраст са подредени съобразно LIFO.

LOCATION Queue (Customer ORDERED BY DEC Priority, INC Age, LIFO) :=5 Customer

Като допълнение към индивидуалните региони могат да се създават масиви от региони.

- Това определя пет региона.

LOCATIONS ARRAY [5] WaitP (Customer ORDERED BY INC Prio) := 0 Customer

- Регионите позволяват всички типове на индексиране.

WaitP[3]

WaitP{2..4}

WaitP{ALL}

Модела Опашка

Работното пространство се определя автоматично от всеки регион, който е деклариран, който се използва при симулационното изпълнение, за да съхранява статистическа информация. Тези променливи не са достъпни в модела. Техните стойности могат да се записват от наблюдатели. Когато се създаде наблюдател, разширяването на регион дава достъп до различни статистики за този регион.

Тези стойности включват:

Модела Опашка

Име	Значение
N	Текущия брой на подвижните компоненти на региона
NLeave	Броя на подвижните компоненти, които са въведени и напуснали региона
NMax	Максималния брой подвижни компоненти в региона във всеки един момент
NMean	Средния брой подвижни компоненти
DWMean	Средното задържане на подвижните компоненти в региона
DEmpty	Времето, за което региона е празен
DOccupy	Времето, за което региона е зает от поне една подвижна компонента
TLeave	Времето, за което последната подвижна компонента напуска региона

Модела Опашка

Описание на подвижните компоненти.

Ако основна компонента съдържа подвижни компоненти.

Синтаксиса има вида:

basic_component : : = BASIC COMPONENT identifier
[mobile_subclass_declaration]
[unit_definition_part]
[local_definitions]
declaration_of_elements
[dynamic_behaviour]
END OF identifier

mobile_subclass_declaration : : = MOBILE SUBCOMPONENT [S] OF CLASS
identifier { ‘,’ identifier }

Модела Опашка

Регионът може да съхранява само подвижни компоненти от един клас. Името на този клас се дефинира в декларацията на региона.

Mobile_subclass_declaration информира основната компонента относно всички възможни класове на подвижни компоненти.

Подвижната компонента се описва по подобен начин на основната компонента, но без динамичното поведение, докато подвижните компоненти имат свои динамики, и промените на състоянието им стават факт само чрез основните компоненти.

С изключение на динамиките, подвижните компоненти могат да съдържат всякакви елементи на модела.

Модела Опашка

Синтаксисът за описание на компонента има вида:

mobile_component ::= MOBILE COMPONENT identifier
[mobile_subclass_declaration]
[unit_definition_part]
[local_definitions]
declarations_of_elements
END OF identifier

mobile_subclass_declarations ::= MOBILE SUBCOMPONENT[S]
OF CLASS
Identifier { ‘,’ identifier}

Модела Опашка

unit_definition_part ::= USE OF UNITS
[unit_definition { unit_definition}]
[time_ unit_definition]

local_definitions ::= LOCAL DEFINITION[S]
[dimension_constant_declaration_part]
[enumeration_set { enumeration_set]}
[tabular_function { tabular_function }]
[tabular_distributions { tabular_distributions }]

Модела Опашка

declaration_of_elements ::= DECLARATION OF ELEMENT[S]
[list_of_constants]
[list_of_state_variables]
[list_of_dependent_variables]
[list_of_sensor_variables]
[list_of_transitions_indicators]
[list_of_sensor_indicators]
[list_of_locations]
[list_of_sensor_locations]

Модела Опашка

Примерът с подвижната компонента Customer1 показва, че в този случай от многото възможности е необходима само една променлива на състоянията от тип INTEGER.

Може да се види, че регионите са възможни също и при подвижните компоненти. Това означава, че подвижна компонента, която се намира в основна компонента може в себе си да съдържа региони за съхраняване на различен клас от подвижни компоненти. По този начин е възможно да се моделират превозни средства, които се движат от регион в регион и които съдържат подвижни компоненти.

Модела Опашка

С въвеждането на подвижни компоненти имаме следната йерархия:

Подвижни компоненти:

- Без динамики;
- Поставени в региони;

Основни компоненти:

- Описание на динамики;

Компоненти от по-високо равнище:

- Свързване на основни компоненти;
- Структурно описание;

Модела Опашка

Моделните елементи на подвижните компоненти са достъпни с цел да се позволи да бъдат изменяни и да се осигури достъп до текущите им стойности.

Моделните елементи на подвижни компоненти се идентифицират от име и път, което започва с регион в основна компонента. Синтаксис:

selected_element :: = indexed_identifier {‘ : ‘ indexed_identifier ‘.’ indexed_identifier}

Модела Опашка

Регионът е последван от двоеточие и след това от класа идентификатор на подвижната компонента. Това трябва да се съпроводи от индекс, който посочва мястото на текуща подвижна компонента при този регион.

Точката е последвана от идентификатор на моделен елемент на подвижната компонента. Този моделен елемент може да е променлива или регион. Ако е регион избраният път може да се продължи.

Модела Опашка

Пример:

- В момента T = 10 приоритетът на задачата, разположена в отрез 5 на региона WaitP се променя на 1.

ON ^T = 10^

WaitP : Customer [5].Prio^ = 1;

END

- Когато има подвижна компонента в първия отрез в регион WaitP с приоритет по-голям от 1, този приоритет се поставя да е Prio = 1.

WHENEVER WaitP : Customer [1].Prio > 1

DO WaitP : Customer [1].Prio^ : = 1;

END

Модела Опашка

Пример 2 не е верен във всяка ситуация. Възможно е да се направи опит за достъп до подвижна компонента, която не присъства в региона.

В пример 2 непрекъснато се проверява следното условие:

WHENEVER WaitP : Customer [1].Prio > 1

Тестът също се провежда, когато регионът WaitP е празен и не присъстват никакви подвижни компоненти. В такъв случай ще се регистрира грешка при симулационното изпълнение и то ще спре.

За да се избегнат такива грешки се препоръчва пространството на състоянията да се раздели на подпространства. Условието в израза WHENEVER трябва да се проверява само, когато може да се гарантира, че в региона присъства подвижна компонента. Това може да се постигне чрез стандартната функция NUMBER. Тази функция връща броя на подвижните компоненти в региона. Тя е от тип INTEGER.

Модела Опашка

Следващото решение е по-добро:

```
IF NUMBER (WaitP) > = 1
    DO WHENEVER WaitP : Customer [1].Prio >1
DO WaitP : Customer [1].Prio^ : = 1;
    END
END
```

Модела Опашка

Забележка:

ON израза не позволява разделянето на пространството на състоянията както е в пример 3. Ако има възможност да се направи опит за достъп до празен регион, тогава трябва да се използва WHENEVER израз.

Когато се обръщаме за достъп към подвижна компонента в регион има други възможности освен прякото определяне на нейното място.

Името на класа на подвижната компонента може да се последва от индекс или набор от индекси. Това прави възможни всички операции за подвижните компоненти.

Модела Опашка

Примери:

- Всички подвижни компоненти в региона Server получават приоритет Prio = 1.

Server : Customer {ALL}.Prio^ : = 1;

- Всички подвижни компоненти до позиция i < = 5 получават приоритет Prio = 1.

Server : Customer {ALL i | i < = 5}.Prio^ : = 1;

- Подвижните компоненти на позиции 4, 5 и 6 получават приоритет Prio = 1.

Server : Customer {4..6}.Prio^ : = 1;

Модела Опашка

Забележка:

Индексните множества са позволени само за лявата страна на изразите за присвояване.

Индексните множества не са позволени в събитийно-стартиращ механизъм. За изразите ON и WHENEVER се позволява само фиксиран индекс.

Модела Опашка

Описание на динамичното поведение.

В основните компоненти създаването, унищожаването и преместването на подвижни компоненти от регион в регион се прави чрез събития, за които могат да се използват и ON и WHENEVER изрази. Възможни са следните команди, последвани от условие или индикатор:

- PLACE : унищожава и реинициализира;
- ADD: създава;
- REMOVE: унищожава;
- GET: донася;
- SEND: изпраща.

Модела Опашка

Всеки от тези изрази позволява също изменението на елементите на модела, принадлежащи на подвижни компоненти, за които се прилагат изразите. Това се постига със следната конструкция:
CHANGING <change of element> END

Модела Опашка

В каква точка при обработването на събития участва движението на подвижни компоненти?

event_defining_statement ::= triggering_mechanism
[procedural_part]
transitions_part

triggering_mechanism ::= WHENEVER expression
| ON indication {OR indication}

transitions_part ::= ON [TRANSITON[S]]
transtition_statement_sequence
END

Модела Опашка

transtition_statement_sequence ::= transtition_statement
{transtition_statement}
transition_statement ::= state_transition_definition
| signal_statement
| event_region_defining_statement
| display_statement
state_transition_definition::=state_variable_assignment
| transfer_statement

Модела Опашка

transfer_statement::=place_statement transfer_end

| add_statement transfer_end

| remove_statement transfer_end

| get_statement transfer_end

| send_statement transfer_end

place_statement ::= selected_element '^' ':'

PLACE expression NEW identifier

Модела Опашка

add_statement ::= selected_element '^' ':' ADD expression NEW identifier

remove_statement ::= selected_element '^' ':' REMOVE indexed_identifier

get_statement ::= selected_element '^' ':'
FROM selected_element
GET indexed_identifier

Модела Опашка

```
send_statement      ::= selected_element  ‘^’ ':'
                        TO selected_element
                        SEND    indexed_identifier
transfer_end                 : :=   ' ; ‘
                                  | transfer_extension
transfer_extention   : :=   ' ; ‘
                        | CHANGING
                        state_transition_definition
                        { state_transition_definition }
                        END
```

Модела Опашка

Забележка:

Избраният елемент от ляво на двоеточието отбелязва региона, в който нещо се случва. Символът ‘^’ се използва, за да подчертае времевото поведение. Съответната промяна на състоянието на състоянието се извършва в следващия цикъл. В това отношение регионите се държат подобно на променливите на състоянието.

Промяна на състоянието на променлива на състоянието:

Tdone^ : = T +10;

Промяна на състоянието за регион: WaitP^ : ADD 1 New Customer;

Модела Опашка

Ако се декларира масив от региони са позволени всички типове индекси.

Примери:

- Петият регион от масива Server донася първата подвижна компонента от регион Queue.

Server [5]^ : FROM Queue GET Customer [1];

- Всяка обслужваща станция в масив Server донася подвижна компонента от регион Queue. Първият Server получава подвижна компонента от първата позиция и т.н. Потребителят трябва да се увери, че региона Queue съдържа достатъчен брой подвижни компоненти.

Server {ALL i}^ : FROM Queue GET Customer [i];

Модела Опашка

- Регионът Server донася първата подвижна компонента от всеки регион в масива Queue.

Server^ : FROM Queue {ALL} GET Customer[1]

- Регионът Server донася всички подвижни компоненти от всички региони в масива Queue.

Server^ : FROM Queue {ALL} GET Customer {ALL}

- Регионът Server донася всички подвижни компоненти от всички региони на масива Queue, докато там има повече от 3 подвижни компоненти.

Server^:FROM Queue {ALL i | NUMBER (Queue [i])> 3}

GET Customer {ALL}

Модела Опашка

• Всички подвижни компоненти в региони с повече от 3 подвижни компоненти и приоритет по-голям от 2 се преместват в регион Server.

Server^ : FROM Queue {ALL i | NUMBER (Queue [i]) > 3}
    GET Customer {ALL j | Queue [i] : Customer [j].Prio > 2}

В основна компонента не се прави разлика дали се използват GET или SEND команди.

Следните 2 команди са еквивалентни:

Station^ : FROM QueueP GET Customer [1];

QueueP^ : TO Station SEND Customer [1];

Разликите са само в гледната точка към модела. По този начин моделът демонстрира дали се подчертава изпращане или получаване.

Модела Опашка

Стартиране на модела Queue1.

Ще направим експерименти с модела Queue1.

Средно време между интервалите

$E[A] = 1/\lambda$

Модела Опашка

Средно време за обработка E[B] = 1/μ
Вероятност станцията да е заета: p = λ/ μ
Средна дължина на опашката: E[Q] = p^2/(1–p)
Средно време на изчисляване: E[W] = (p/ μ)/(1–p)
Системата M/M/1, която разглеждаме показва:
E[A] = 15 BE
E[B] = 10 BE
E[Q] = 1.33 клиенти
E[W] = 20.0 BE
λ= 0.066
μ = 0.1

Модела Опашка

Хибридни модели.

SIMPLEX MDL е особено силен, когато се дефинират хибридни модели, съдържащи подвижни компоненти, събития и диференциални уравнения.

Хибридният модел QueueK е основан на модела Queue1. В модела Queue1 услугата се моделира просто като определено количество от време. В модела QueueK това се заменя с непрекъснат процес.

Моделът QueueK описва доставката и запълването на барели в пълнеща станция, която се състои от помпа и сензор, които следят процеса на пълнене.

Зареждащата станция може да пълни само един барел в един момент от време. Барелите, които пристигат докато помпата е заета формират опашка в региона Queue.

Модела Опашка

Скоростта на пълнене на барел зависи от количеството течност в барела. Започвайки от начална скорост 1.0 литра/секунда се увеличава пропорционално с коефициент 0.2 към количеството в барела, докато се достигне максималната скорост от 0.6 литра/секунда.
Скоростта на запълване е:
Rate : = Min {0.2 * Amount + 1.0 Litre/s; 6.0 Litre/s}
Така Rate е зависима променлива, чиято стойност е между 1.0 и 6.0 и зависи от количеството в барела.
Запълването на барела в регион Pump се описва от диференциално уравнение:
Amount’ : = Rate

Модела Опашка

Когато барелът е пълен, процесът на пълнене спира. Скоростта на пълнене Rate се поставя да е 0.

Напълнените барели се изпращат в регион Store.

Количеството в барела нараства непрекъснато. Когато скоростта на пълнене достигне максималната си стойност от 6.0, тя става константа, увеличението в количеството нараства линейно.

Използва се събитие, за да се провери дали барела е пълен. Тъй като пълненето е непрекъснат процес е нужно кръстосано търсене.

Когато барелът е пълен се поставя индикатор. По този начин се моделира сензор, който дава сигнал за преместване на барел от мястото за пълнене. Барелите се представят чрез подвижна компонента CustomerK. Тази компонента съдържа променливата Volume (вместимост).

Модела Опашка

BASIC COMPONENT QueueK

MOBILE SUBCOMPONENTS OF CLASS CustomerK

USE OF UNITS
  UNIT[Litre] = BASIS
  TIMEUNIT   = [s]

DECLARATION OF ELEMENTS

STATE VARIABLES
DISCRETE
  TArrive (REAL[s]) := 0 [s], # Интервали време на пристигане
 Protocol (LOGICAL) := FALSE

Модела Опашка

CONTINUOUS

Amount (REAL[Litre]) := 0 [Litre] # Текущо съдържание на барел

DEPENDENT VARIABLES

CONTINUOUS

Rate (REAL[Litre/s]) := 0.0 [Litre/s] # Скорост на пълнене на помпата

RANDOM VARIABLES

Arrive (REAL[s]) : EXPO (Mean:=14[s],LowLimit:=0.45[s],UpLimit:=70[s]),

Size (INTEGER) : IUNIFORM (LowLimit:=2, UpLimit:=6)

Модела Опашка

TRANSITION INDICATORS

BARREL_FULL

LOCATIONS

Queue (CustomerK) := 0 CustomerK, # Източник на празни барели

Pump (CustomerK) := 0 CustomerK, # Място за пълнене

Store (CustomerK) := 0 CustomerK # Съхраняване на пълни барели

Модела Опашка

```
DYNAMIC BEHAVIOUR

# Създаване на барел
WHENEVER (T >= TArrive)
DO
  TArrive ^ := T + Arrive;
  Queue^ : ADD 1 NEW CustomerK
    CHANGING
      Volume^ := Size * 10 [Litre];
    END
  IF Protocol
  DO
    DISPLAY("T = %f : %d litre barrel created\n",T,Size*10);
  END
END
```

Модела Опашка

```
# Преместване на барел от опашката към пълнещата станция
WHENEVER (NUMBER(Pump) = 0) AND (NUMBER(Queue) > 0)
DO
  Pump^ : FROM Queue GET CustomerK[1];
  IF Protocol
  DO
    DISPLAY("T = %f : Barrel moved from \n", T);
    DISPLAY("         queue to filling station\n");
  END
END

DIFFERENTIAL EQUATIONS
  # пълнене на барел
  Amount' := Rate;
END
```

Модела Опашка

```
# Алгебрични уравнения за контролиране на скоростта на пълнене
IF (NUMBER(Pump)> 0) AND (Amount <
   Pump:CustomerK[1].Volume)
DO
  Rate := MIN (0.2 * Amount * 1 [1/s] + 1 [Litre/s], 6 [Litre/s]);
END
ELSE DO
  Rate := 0 [Litre/s];
END
# Завършване на пълненето
IF (NUMBER(Pump) > 0)
DO
  WHENEVER (Amount >= Pump:CustomerK[1].Volume)
  DO
    Amount^ := 0 [Litre];
   SIGNAL BARREL_FULL;
  END      END
```

Модела Опашка

```
# Напускане на пълнещата станция
ON BARREL_FULL
DO
  Pump^ : TO Store SEND CustomerK[1];
  IF Protocol
   DO
     DISPLAY("T = %f : Barrel full!\n",T);
   END
END
END OF QueueK
```

Модела Опашка

В модела QueueK има и непрекъснати и зависещи от събития процеси, както и премествания на подвижни компоненти. По-нататък се използва алгебрично уравнение, за да се продължи промяната на стойността на зависимата променлива Rate.

Модела Опашка

```
MOBILE COMPONENT CustomerK

 USE OF UNITS
   UNIT [Litre] = BASIS

 DECLARATION OF ELEMENTS

 STATE VARIABLES
 DISCRETE
   Volume (REAL[Litre])  := 0 [Litre]      # Volume of the barrel

 END OF CustomerK
```

Подвижната компонента CustomerK

Транспортни модели

Регионите са позволени в декларацията на елементи в подвижна компонента. Това означава, че подвижната компонента, която е разположена в регион в основна компонента може да съдържа региони за разполагане на бъдещ клас от подвижни компоненти. По този начин е възможно да се моделира превозно средство, което се мести от регион в регион, и самото то да транспортира подвижни компоненти.

Подвижни компоненти, които са разположени в региона на транспортната среда могат и те да съдържат региони. По този начин могат да се опишат транспортни процеси от много равнища.

Транспортни модели

Пример:

На входната станция се зарежда нисък товарител с камиони, които возят стоки. Зареждащата станция се моделира от основната компонента с регион за разполагане подвижната компонента за клас нисък товарител. Той има регион, който може да настани камиони като по-нататъшен клас от подвижни компоненти. Подвижните компоненти за представяне на камионите също имат регион, в който се разполагат подвижните компоненти представящи стоките.

Транспортни модели

Получаване на достъп до атрибути и описание на динамика за превозни процеси.

Ако е достъпна променлива на състоянието на подвижна компонента, която представя вещи, доставени от превозни средства, тогава трябва да се специфицира пълното име на пътя.

Пример 1:

Първата позиция на регион Station съдържа камион, който превозва коли. Тези коли са разположени в региона Ramp върху камиона. На първата кола на товарната рампа се дава скорост Speed = 5.

Station: Truck[1] .Ramp : Car[1] .Speed^ : = 5;

Синтаксиса има следната форма:

state_variable_assignment : : = selected_element '^' ': ='
expression ';'

selected_element : : = indexed_identifier { ':' indexed_identifier
'.' indexed_identifier }

Транспортни модели

Пътят може да е произволно дълъг. Това позволява подвижни компоненти да функционират като превозвачи за по-нататъшни подвижни компоненти и т.н.

Позволени са командите:

PLACE, ADD, REMOVE, GET и SEND.

По-нататък моделните елементи на подвижната компонента, към която се обръщаме чрез всеки от горните изрази може да се модифицират. Това се постига чрез командата CHANGING.

Транспортни модели

Примери:

• Регионът Station съдържа подвижната компонента Taxi. Един от елементите на модела на таксито е регион, наречен Seat. Подвижната компонента от регион WaitP на клас Customer се поставя върху Seat на таксито.

Station: Taxi[1] .Seat^ : FROM WaitP GET Customer[1]

• Регионът Station съдържа подвижната компонента Taxi. Всички подвижни компоненти на класа Customer, които са в регион Seat са преместени и унищожени.

Station: Taxi[1] .Seat^ : REMOVE Customer {ALL};

Транспортни модели

Моделът Транспорт.

Моделът Transport описва цикъл, в които такситата могат да качват и превозват пътници както и да ги оставят на мястото закъдето те пътуват.

Транспортни модели

В началото на изпълнението всички таксита са на първата таксиметрова спирка QTaxi1. Те са индивидуално номерирани.

Клиентите пристигат на първата таксиметрова спирка WaitP. Техните интервали на пристигане са експоненциално разпределени със средна стойност 10 времеви единици.

Когато пристигне клиент на първата таксиметрова спирка, такси го откарва на граничната станция Station 1. Тук таксито може да вземе до 4 пътника. Качването отнема точно 3 времеви единици.

Транспортни модели

След качването таксито влиза в Road1. Времето за придвижване е нормално разпределено със следните параметри:

Средно = 36 ВЕ

- Стандартно отклонение = 10 ВЕ
- Долна граница = 0 ВЕ
- Горна граница = 72 ВЕ

След изтичане на времето за придвижване 3 таксита се нареждат на опашката при спирка QTaxi2.

Транспортни модели

Слизането отнема 2 времеви единици. Тогава празното такси кара през Road2 обратно до изчакваща точка QTaxi1. Времето за придвижване по Road2 е нормално разпределено както е и за Road1. За тестови цели се записва цялостното поведение на модела.
Начално се декларират подвижните компоненти на клас Taxi1 и Customer1. Тогава елементите са декларирани. Времето за качване на пътници и това за слизане в региони Station1 и Station2 е константа.

Транспортни модели

BASIC COMPONENT Trans1

MOBILE SUBCOMPONENTS OF CLASS
Taxi1,Customer1

DECLARATION OF ELEMENTS

CONSTANTS

```
Load(REAL)      := 3,     # Време за качване
Unload(REAL)    := 2      # Време за слизане
```

Транспортни модели

```
STATE VARIABLES
   TNext(REAL)        := 0,      # Генериране на пътник
   TLoad(REAL)        := 0,      # Край на качването
   TUnload(REAL)      := 0,      # Край на слизането
   Protocol(LOGICAL)  := FALSE   # Контрол на протокола

RANDOM VARIABLES
   ZArrive(REAL)      : EXPO  (Mean := 10),
   ZTravel1(REAL)     : GAUSS (Mean:=36,Sigma:=10,
                              LowLimit:=0,UpLimit:=72),
   ZTravel2(REAL)     : GAUSS (Mean:=36,Sigma:=10,
                              LowLimit:=0,UpLimit:=72)
```

Транспортни модели

```
LOCATIONS
   QTaxi1   (Taxi1)                        := 10 Taxi1,
   Station1 (Taxi1)                        := 0 Taxi1,
   WaitP    (Customer1)                       := 0 Customer1,
   Road1    (Taxi1 ORDERED BY INC TTravel) := 0
Taxi1,
   QTaxi2   (Taxi1)                        := 0 Taxi1,
   Station2 (Taxi1)                        := 0 Taxi1,
   Road2    (Taxi1 ORDERED BY INC TTravel) := 0
Taxi1
```

Транспортни модели

Нужни са променливи на състоянието за генериране на нов клиент и края на времената за качване и слизане. Променливата Protocol се използва, за да контролира извеждането на протокола. Промяната на началната й стойност от FALSE на TRUE чрез командата Model parameters в контекстното меню на обект Break0 ще активира извеждането.
След това се декларират случайни променливи, които описват времената на пристиганията на клиенти и времената на придвижване по пътищата Road1 и Road2.
Наредбата за всички региони е FIFO.
Следващото такси, което ще напусне Road1 или Road2 съответно е в началото на опашката.

Транспортни модели

```
DYNAMIC BEHAVIOUR

# Събитие 1: Номериране на такситата

ON START
DO
  QTaxi1^ : FROM QTaxi1 GET Taxi1{ALL i}
    CHANGING
      TaxiN^ :=i;
    END
    IF Protocol
    DO
      DISPLAY("T= %f Event 1 \n",T);
      DISPLAY("   Number the Taxis \n\n");
    END
END
```

Транспортни модели

```
# Събитие 2: Генериране на пътник

WHENEVER T >= TNext
DO
  WaitP^ : ADD 1 NEW Customer1;
  TNext^ := T + ZArrive;
  IF Protocol
  DO
    DISPLAY("T= %f Event 2 \n",T);
    DISPLAY("   Customer enters WaitP \n");
    DISPLAY("   Next generation in T= %f \n",T+ZArrive);
    DISPLAY("   Customers in WaitP= %ld
    \n\n",NUMBER(WaitP)+1);
  END
END
```

Транспортни модели

```
# Събитие 3: Такси спира на Спирка 1


WHENEVER (NUMBER(WaitP) > 0) AND (NUMBER(QTaxi1) >
    0) AND
        (NUMBER(Station1) = 0)
DO
  Station1^ : FROM QTaxi1 GET Taxi1[1];
  IF Protocol
  DO
    DISPLAY("T= %f Event 3 \n",T);
    DISPLAY("   Taxi number %ld enters Station 1 \n\n",
                                QTaxi1:Taxi1[1].TaxiN);
  END
END
```

Транспортни модели

```
# Събитие 4: Качване

IF NUMBER(Station1) = 1
DO
  WHENEVER NUMBER(Station1:Taxi1[1].Seat) = 0
  DO
    Station1:Taxi1[1].Seat^ : FROM WaitP GET
            Customer1{ALL i | (i<=NUMBER(WaitP)) AND
(i<=4)};
    TLoad^ := T + Load;
    IF Protocol
    DO
      DISPLAY("T= %f Event 4 \n",T);
```

Транспортни модели

```
IF NUMBER(WaitP) < 4
    DO
     DISPLAY("   Loading Taxi Number %ld, # Cust %ld \n",
              Station1:Taxi1[1].TaxiN, NUMBER(WaitP));
    END
    ELSE DO
     DISPLAY("   Loading Taxi Number %ld, # Cust %ld \n",
              Station1:Taxi1[1].TaxiN, 4);
    END
    DISPLAY("   Loading ends in T=% f \n\n",T+Load);
   END
  END
END
```

Транспортни модели

```
# Събитие 5: Влизане в маршрут 1

ON ^T >= TLoad^
DO
  Road1^ : FROM Station1 GET Taxi1[1]
    CHANGING
      TTravel^ := T + ZTravel1;
    END
  IF Protocol
  DO
    DISPLAY("T= %f Event 5 \n",T);
    DISPLAY("   Taxi Number %ld enters Road1 \n",
                Station1:Taxi1[1].TaxiN);
    DISPLAY("   Journey ends in T= %f \n\n",T+ZTravel1);
  END
END
```

Транспортни модели

```
# Събитие 6: Излизане от маршрут 1
IF NUMBER(Road1) > 0
DO
  WHENEVER T >= Road1:Taxi1[1].TTravel
  DO
    QTaxi2^ : FROM  Road1 GET Taxi1[1];
    IF Protocol
    DO
      DISPLAY("T= %f Event 6 \n",T);
      DISPLAY("   Taxi Number %ld enters QTaxi2 \n\n",
                Road1:Taxi1[1].TaxiN);
    END
  END
END
```

Транспортни модели

```
# Събитие 7: Таксито спира на Спирка 2
WHENEVER (NUMBER(Station2) = 0) AND
      (NUMBER(QTaxi2) >0)
DO
  Station2^ : FROM QTaxi2 GET Taxi1[1];
  IF Protocol
  DO
    DISPLAY("T= %f Event 7 \n",T);
    DISPLAY("   Taxi Number %ld enters Station 2  \n\n",
                QTaxi2:Taxi1[1].TaxiN);
  END
END
```

Транспортни модели

```
# Събитие 8: Слизане
IF NUMBER(Station2) =1
DO
  WHENEVER NUMBER(Station2:Taxi1[1].Seat) > 0
  DO
    Station2:Taxi1[1].Seat^  : REMOVE Customer1{ALL};
    TUnload^ := T + Unload;
```

Транспортни модели

```
IF Protocol
  DO
    DISPLAY("T= %f Event 8 \n",T);
    DISPLAY("   Unloading Taxi number %ld, # Cust. %ld
  \n",
                Station2:Taxi1[1].TaxiN,
  NUMBER(Station2:Taxi1[1].Seat));
    DISPLAY("   Unloading ends in T= %f \n\n",T+Unload);
  END
 END
END
```

Транспортни модели

```
# Събитие 9: Влизане в маршрут 2
ON ^T >= TUnload^
DO
  Road2^ : FROM Station2 GET Taxi1[1]
    CHANGING
      TTravel^ := T + ZTravel2;
    END
  IF Protocol
  DO
    DISPLAY("T= %f Event 9 \n",T);
    DISPLAY("   Taxi Number %ld enters Road2 \n",
                Station2:Taxi1[1].TaxiN);
    DISPLAY("   Journey ends in T= %f \n\n",T+ZTravel2);
  END
END
```

Транспортни модели

```
# Събитие 10: Излизане от маршрут 2

IF NUMBER(Road2) >0
DO
  WHENEVER T >= Road2:Taxi1[1].TTravel
  DO
    QTaxi1^ : FROM Road2 GET Taxi1[1];
    IF Protocol
    DO
      DISPLAY("T= %f Event 10 \n",T);
      DISPLAY("   Taxi Number %ld enters QTaxi1 \n\n",
                Road2:Taxi1[1].TaxiN);
    END
  END
END

END OF Trans1
```

Транспортни модели

Описанието на динамиките се нуждае от 10 събития:

- Събитие 1:

Такситата се преместват от регион QTaxi1, номерират се и се връщат в регион QTaxi1. Индексът i за номериране на таксита е валиден и в следващата команда CHANGING.

- Събитие 2:

Създаване на нови клиенти. Изходът дава текущата дължина на опашката.

- Събитие 3:

Такситата влизат в спирката Station 1.

- Събитие 4:

Позволява се максимум от 4 пътници в таксито. Броя на клиентите се записва.

Транспортни модели

- Събитие 5:

След качването таксито поема по Road1. Времето на пътуване се определя и присвоява на променливата на състоянието TTravel. Изходът съдържа броя на такситата и изчисленото време на пристигане в QTaxi2.

- Събитие 6:

Всички таксита, които са на маршрут Road1 са в региона Road1. Те са сортирани в намаляващ ред за времето на завършено пътуване. Всяко такси носи това време като съставна компонента. Може да се определи кога следващото такси напуска пътя и достига региона QTaxi2.

- Събитие 7:

Ако Station2 не е заета таксито в QTaxi2 може да влезе на спирката.

Транспортни модели

- Събитие 8:

Всички клиенти се преместват и унищожават. По същото време се записва времето на слизане.

- Събитие 9:

Входа за пътя Road2 е аналогичен на входа за пътя Road1.

- Събитие 10:

Изхода от път Road2 и входа в точката на изчакване QTaxi1 са идентични на Събитие 6.

В допълнение към основната компонента е необходимо описанието на класа на подвижната компонента Taxi1. Подвижните компоненти на този клас могат да се намерят в региони QTaxi1, Station1, Road1, QTaxi2, Station2 и Road2.

Транспортни модели

```
MOBILE COMPONENT Taxi1

MOBILE SUBCOMPONENTS OF CLASS Customer1

DECLARATION OF ELEMENTS

STATE VARIABLES
   TTravel(REAL)   := 0,
   TaxiN(INTEGER)  := 0

LOCATION
   Seat(Customer1) := 0 Customer1

END OF Taxi1
```

Транспортни модели

Когато моделните променливи се записват, командата DISPLAY извежда стойността в текущия цикъл. DISPLAY изразът не трябва да съдържа TNext, тъй като TNext няма да се свърже с правилната стойност до следващия цикъл. Вместо това трябва да се напише T + ZArrive.
Събитие 3 е подобно. Ще се отпечата броя на такситата, които се преместват от регион QTaxi1 в Station1. При текущия цикъл таксито все още е в регион QTaxi1. Затова трябва да се даде този регион.
Друга възможност е да се забави командата DISPLAY с един цикъл. Това може да се постигне като се направи извеждащата команда обект на индикатор за преместване.

Транспортни модели

В допълнение към променливата на състояние подвижната компонента QTaxi1 съдържа региона Seat, който може да разположи подвижни компоненти от клас Customer1.

Когато подвижните компоненти не се нуждаят от атрибути, описанието е празно. Въпреки това то се изисква.

Транспортни модели

MOBILE COMPONENT Customer1

DECLARATION OF ELEMENTS

END OF Customer1

Стратегии

Критерия, използван за да подреди задачите в опашка (и затова също редът, в които те ще се обработват) е познат като стратегия.

Стратегията използва приоритети, за да определи позицията на задача в опашка. Приоритета е мярка за важност, която може да определи приоритета на задача сама по себе си или може да се изчисли съобразно определени критерии.

В особено важни случаи може да е нужна опцията да освободим зает сървър, за да направим възможно задача с висок приоритет да се обработи незабавно. Механизмът, който позволява това да се направи се нарича изместване.

Стратегии

Изместването (pre-emtion) винаги изисква определено количество време. По тази причина стратегията на изместването може да реагира бързо на спешните случаи, но на цената на допълнително време за презареждане.

Има разлика между статичните и динамични приоритети. Със статични приоритети, на задача се присвоява приоритет, който не се променя по време на нейния престой в опашката. Обратно с динамичните приоритети може да се промени приоритета на задача, зависеща от текущото състояние на системата.

Стратегии

Статични стратегии.

Тази секция описва няколко важни и основни стратегии, които използват статични приоритети

FIFO (Първи вътре, Първи вън)

FIFO стратегията дава приоритети съобразно времето на пристигане. Задачата, която е била в опашката най-дълго има най-висок приоритет. Той отговаря на концепцията за безпристрастност. Приоритетът се поставя равен на входното време:

Prio : = T_{ARRIVE}

Задачите са подредени в опашката според приоритета си. Задача, която влиза в опашката първа има най-малката стойност за Prio и така стои при началото на опашката. Тя ще бъде първата за обработка.

PFIFO (Preemptive FIFO) означава, че задача влязла в опашката ще измести задачата, която в момента се обработва, ако има по-висок приоритет.

Стратегии

Пример:

По-малко важна задача се премества през производственото съоръжение. Ако пристигне спешна задача, която трябва да се приеме и обработи веднага, тогава оригиналната задача ще се премахне от машината. Тя трябва да напусне машината и да се върне в опашката.

LIFO (Last In, First Out)

Обратно на FIFO, LIFO стратегията дава предимство на задачата, пристигнала в опашката последна. LIFO работи съобразно стек принципа, където пристигащите задачи се поставят на върха на стека и се премахват оттам. Приоритетът е отрицателното време на пристигане.

Prio : $=-T_{ARRIVE}$

Стратегии

SJF (Shortest Job First)
Стратегията SJF присъединява най-високия приоритет към най-краткото време за обработка в сървъра. Тя максимализира производството, т.е. най-големият възможен брой задачи се обработва и най-голям възможен брой задачи могат да бъдат удовлетворени. Тази стратегия се основава на идеята, че е добро хрумване е да се обработват първо малките задачи преди да се премине към по-големите.
Пример: Планирането на задачи решавани с помощта на компютър позволява кратките задачи да се обработват първи. Времеемките изчислителни задачи приемат дълги времена за изчакване, защото имат нужда от по-големи ресурси.
Приоритетът е поставен равен на времето за обработка:
Prio : = $T_{PROCESS}$
SJF позволява презареждане.

Стратегии

Round Robin (Cyclic Strategy)

Round Robin стратегията свързва сървъра с всяка задача за определен период от време. След като този ‘отрез от време’ изтече се проверява дали задачата е завършена и може да напусне сървъра. Ако случаят не е такъв, тогава тя се връща в опашката с нейното оставащо време за обработка. Тази стратегия предполага, че е добра идея да се обработват по малко всички задачи. Това е особено подходящо, ако задачата може да върши други дейности, които не се нуждаят от сървъра след като тя е била обработена за кратко време.

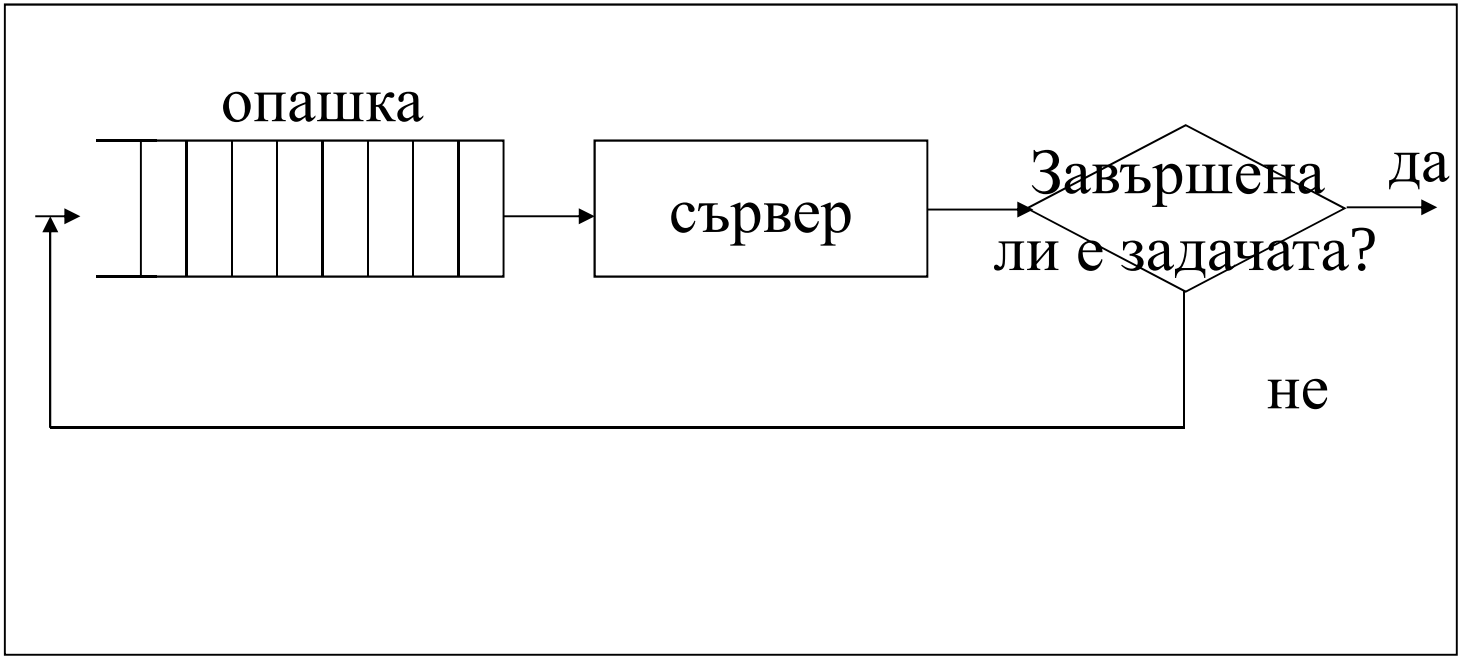
Пример:

Компютърът, работещ в режим на времеделение, свързва всяка задача с отрез от време. Задачата се обработва, но с ефективна скорост, която се намаля с толкова повече, колкото повече задачи влизат в системата.

Round Robin може да се осъществи само, ако е налично презареждане.

Стратегии

Стратегията Round Robin



Стратегии

Round Robin може да се направи по-гъвкава, ако дължината на времевия отрез не е константна, а зависи от важността на задачата. Това ражда стратегията RRP (Round Robin with Priorities).

LFB (Limited Feedback)

Цикличните стратегии изискват допълнителни ресурси. Затова в много случаи е препоръчително да се ограничи броя на задачите в стратегията Round Robin. Има различни методи за определяне на задачи, които могат да участват в Round Robin, докато други задачи се третират по различен начин.

Прост пример е LFB стратегията, където всяка задача получава Nmax времеви отрези по начина на Round Robin. Когато всички Nmax времеви отрези се използват, задачата влиза в различна опашка, която работи с различна стратегия, като FIFO.

Тогава задачите се премахват от тази опашка само, ако няма останали в опашката на Round Robin.

Стратегии

Стратегиите с опашки отразяват организацията на намиращата се отдолу реална система. Много аспекти на организацията могат да се представят чрез наредба при обработване в модела, и по този начин чрез стратегия. Има толкова стратегии, колкото са формите на организация.

Пример:

- Опашката пред областта за съхранение може да се организира според нужното количество за съхранение. Например задачата с най-малко изисквания може да получи най-висок приоритет. Това означава, че задачите, имащи големи изисквания ще трябва да чакат прекалено дълго.
- Може да се използва и противоположен подход, където на големи изисквания се дава висок приоритет.
- Възможно е също да се комбинират двата метода и да се използва стратегия с редуващо се поставяне на задачи с големи и малки изисквания в опашката.

Стратегии

Статичните стратегии свързват приоритети към задачи, когато те пристигнат в опашката. Обратно, динамичните стратегии позволяват променянето на приоритетите на чакащите задачи в зависимост от определени условия.

Условията, които определят динамичното присвояване на приоритети може да се мени. Общо тяхно описание не е възможно. Приоритетите трябва да се представят като функция, която взима всички системни състояния като аргументи.

Стратегии

Следващите секции описват няколко основни динамични стратегии:

UTL (Upper Time Limit)

Всяка задача може да получи горна времева граница, за която тя трябва да приключи. Колкото повече задачата се приближи към тази граница, толкова по-висок става приоритета ѝ. В този случай приоритета може да се определи от проста функция, която взима като аргумент времето, оставащо до границата. Това оставащо време съдържа оставащото време за обработка плюс времето за чакане.

В най-простия случай приоритета се увеличава линейно с оставащото време. Приоритетът се оценява според следната формула:

Prio : = T – Tcomplete = remaining time allowed (позволено оставащо време)

В този случай всички приоритети са отрицателни.

Стратегии

UTLP (Upper Time Limit with Priorities)

Стратегията UTL може да се рафинира и да се подобрява с това динамичните приоритети да зависят от начален приоритет вместо всички да са идентични. UTLP стратегията увеличава приоритета за всички задачи с висок начален приоритет по-бързо от този на задачи с нисък начален приоритет.

Приоритетите могат да получат различни поведения, които зависят от началния им приоритет Prio0.

В момента T = 1 задачата пристига с начален приоритет Prio0 = 0. Според динамичните приоритети тя достига стойност Prio = 13 в момента T = 13. Задачата, която влиза в системата в момента T= 8 с начален приоритет Prio0 = 4 достига максималния приоритет Prio = 20 в момента T = 14, поради привилегированите ѝ стойности. Важният ефект е, че редът на задачите не е запазен. UTLP позволява задачите да се настигат една друга.

Стратегии

Едно изменение на UTL заема натрупаното време на изчакване:

Prio = желано време за изчакване/натрупано време на изчакване

Коефициентът на желаното време за изчакване към натрупаното изчакващо време се намалява с увеличаване на времето за чакане. Затова задачите са подредени в намаляващ приоритет в опашката.

Всички стратегии с динамични приоритети изискващи преоценяване на приоритетите се стартират отделно. Моментът време, в което това се случва зависи от типа задача.

Два основни метода са:

- Приоритетите се изчисляват в набор от интервали;
- Приоритетите се изчисляват всеки път, когато задача трябва да се избере от стратегията.

Стратегии

Честотата, с която приоритетите се свързват определя точността, с която те представят верните условия. Тъй като приоритетите се прави първо, това подобрение има също недостатъци.

Ако времената на обслужване са кратки сравнени с динамичното присвояване на приоритети, тогава по-късно не може да се пренебрегнат.

Приоритетите зависят от броя на задачите, засегнати от динамичните приоритети. При повече засегнати задачи е нужно по-дълго време.

Стратегии

Стратегии на презареждане.

На модела Queue2 са направени следните изменения:

- Всяка десета задача е спешна задача с приоритет 0. Всички други задачи имат приоритет 1.
- Сървърът се презарежда. Всеки път, когато се появи спешна задача сървърът се освобождава и презареденатa задача отива в края на опашката с нейното оставащо време за обработка. Подготвителното време за всяко презареждане е 1 времева единица.

Стратегии

Регионът WaitP се подрежда според стратегията ‘намаляващ приоритет’. Тъй като няма друга информация, задачи с идентични приоритети се нареждат според FIFO стратегията.

Всеки път, когато се появи нова задача в опашката трябва да се провери дали презареждането участва или не. Това ще е случая, когато нова задача е спешна и сървърът все още не съдържа спешна задача.

Ако е необходимо презареждане се поставя индикатора Preempt. Този индикатор стартира събитие, в което текущата задача се изпраща към края на опашката и нейното оставащо време за обработка се отбелязва. По същото време се извършва присвояването TReequip = T + Reequip. Това осигурява, че спешна задача влиза в станцията след като времето за пренастройването за нея е изтекло.

За да се избегне нуждата от различаване на оставащото време за обработка и общото време за обработка, декларацията на подвижната компонента Customer2 също съдържа времето TimeA. В момента на създаване всяка задача се свързва с времето за нейната обработка, което се съхранява в TimeA.

Времето за обработка на задачата е достъпно, когато се оцени крайното време Twork.

Стратегии

```
MOBILE COMPONENT Customer2

DECLARATION OF ELEMENTS

   STATE VARIABLES
   DISCRETE
     Priority (INTEGER) := 1,  # Приоритет
     TimeA   (REAL)    := 0    # Оставащо време

END OF Customer2
```

Стратегии

BASIC COMPONENT Queue2

MOBILE SUBCOMPONENTS OF CLASS Customer2

LOCAL DEFINITIONS

VALUE SET StateType : ('busy', 'reequipping', 'free')

DECLARATION OF ELEMENTS

CONSTANTS

Reequip (REAL) := 1 # Време за презареждане на сървър когато спешна работа измести друга

Стратегии

```
STATE VARIABLES

DISCRETE
   TArrive (REAL)     := 0,          # Следващо време за пристигане
   TWork    (REAL)    := 0,          # Край на обработката
   TReequip (REAL)    := 0,          # Край на презареждането
   Protocol (LOGICAL) := FALSE,      # Контрол на протокола
   Job      (INTEGER) := 1,          # Брояч на задачите
   State  (StateType) := 'free'      # Състояние на сървъра

RANDOM VARIABLES
   Arrive (REAL)      : EXPO (Mean:=15,LowLimit:=0.5,UpLimit:=75),
   Work   (REAL)      : EXPO (Mean:=10,LowLimit:=0.32,UpLimit:=50)
```

Стратегии

TRANSITION INDICATOR

Preempt # Индикатор за стартиране на презареждането

LOCATIONS

WaitP (Customer2 ORDERED BY DEC Priority) := 0 Customer2,

Station (Customer2) := 0 Customer2,

Sink (Customer2) := 0 Customer2

Стратегии

```
DYNAMIC BEHAVIOUR
# Създаване на задачи
WHENEVER T >= TArrive
DO
  IF Job = 10      # Нова спешна задача
  DO
    WaitP^ : ADD 1 NEW Customer2
      CHANGING
        Priority^ := 0;
        TimeA^   := Work;
      END
    Job^ := 1;
    IF (State = 'busy') AND (Station:Customer2[1].Priority = 1)
    DO          SIGNAL Preempt;                END                    END
```

Стратегии

```
 ELSE DO              # Нова стандартна задача
   WaitP^ : ADD 1 NEW Customer2
     CHANGING
       TimeA^ := Work;
     END
   Job^ := Job + 1;
 END

 TArrive^ := T + Arrive;
 IF Protocol
 DO
    DISPLAY ("T= %f New job \n", T);
 END
END
```

Стратегии

```
# Начало на обработката на стандартна задача
 WHENEVER (NUMBER(Station)=0) AND
          (NUMBER(WaitP)>0)  AND
          (State <> 'reequipping')
 DO
  Station^ : FROM WaitP GET Customer2[1];
  TWork^ := T + WaitP:Customer2[1].TimeA;
  State^ := 'busy';
  IF Protocol
  DO
     DISPLAY ("T= %f Standard job enters server\n",T);
  END
END
```

Стратегии

```
# Стандартна задача е изместена от спешна
 ON Preempt
 DO
   IF (State = 'busy')
   DO
     Station:Customer2[1].TimeA^ := TWork - T;  # оставащо
    време
     Station^   :  TO WaitP SEND Customer2[1];
     TReequip^  := T + Reequip;                  # време за
    презареждане
     State^   := 'reequipping';
   END
```

Стратегии

```
IF Protocol
  DO
    DISPLAY ("T= %f Rush job arrives, \n", T);
    DISPLAY ("    Standard job is preempted\n");
  END
END
```

Стратегии

```
# Начало на обработка на спешна задача
 WHENEVER (T >= TReequip) AND (State = 'reequipping')
 DO
   Station^ :  FROM WaitP GET Customer2[1];
   TWork^ := T + WaitP:Customer2[1].TimeA;
   State^ := 'busy';
   IF Protocol
   DO
     DISPLAY ("T= %f Rush job enters server\n", T);
  END
END
```

Стратегии

```
# Край
 WHENEVER (T >= TWork) AND (State = 'busy')
  DO
   Station^ :  TO Sink SEND Customer2[1];
   State^ := 'free';
   IF Protocol
   DO
     DISPLAY ("T= %f Job leaves server\n", T);
  END
END
```

Стратегии

```
# Унищожаване на задачи
 WHENEVER NUMBER(Sink) >= 4
 DO
   Sink^ : REMOVE Customer2{ALL};
   IF Protocol
   DO
     DISPLAY ("T= %f 4 jobs destroyed\n", T);
  END
END

END OF Queue2
```

Стратегии

Стратегии с процедурна част.

Някои сложни стратегии изискват претърсване на опашката преди сървъра. Това се постига в процедурна част.

Моделът Queue3 също се базира на Queue1. В допълнение, при създаването си всяка задача се свързва с максимално време за чакане, което се съхранява в променливата WaitT на подвижната компонента Customer3. Ако това максимално време за чакане се достигне, задачата напуска опашката преждевременно без да се обработи.

Стратегии

MOBILE COMPONENT Customer3

DECLARATION OF ELEMENTS

STATE VARIABLES

DISCRETE

Priority (INTEGER) := 1, # Приоритет

WaitT (REAL) := 0 # Максимално време, за което задача може да остане в опашката

END OF Customer3

Стратегии

```
BASIC COMPONENT Queue3
MOBILE SUBCOMPONENTS OF CLASS Customer3
DECLARATION OF ELEMENTS

STATE VARIABLES
DISCRETE
   TArrive  (REAL)    := 0,       # Следващо време на
пристигане
   TWork    (REAL)    := 0,       # Край на обработката
   Protocol (LOGICAL) := FALSE,   # Контрол на
протокола
   MinTime  (REAL)    := 0        # Следващо време на
напускане
```

Стратегии

RANDOM VARIABLES

Arrive (REAL) : EXPO (Mean:=15,LowLimit:=0.5,UpLimit:=75),
Work (REAL) : EXPO (Mean:=10,LowLimit:=0.32,UpLimit:=50),
MaxW (REAL) : GAUSS (Mean:=25,Sigma:=5,
UpLimit:=40,LowLimit:=10) # Максимално време за изчакване

TRANSITION INDICATOR

MinNew # Индикатор за стартиране на преизчисляване на следващото време за напускане

Стратегии

LOCATIONS

WaitP (Customer3) := 0 Customer3, # Входна опашка

Station (Customer3) := 0 Customer3, # Сървър

Sink (Customer3) := 0 Customer3 # Контейнер

Стратегии

```
DYNAMIC BEHAVIOUR
# Създаване на задачи
 WHENEVER T >= TArrive
 DO
   WaitP^ : ADD 1 NEW Customer3
   CHANGING
     WaitT^ := T + MaxW;
   END
   IF (MinTime - MaxW > T) OR
      (NUMBER(WaitP) = 0)
   DO
     MinTime^ := T + MaxW;
   END
   TArrive^ := T + Arrive;
   IF Protocol
   DO       DISPLAY ("T= %f New customer\n",T);     END     END
```

Стратегии

```
# Начало на обработката
 WHENEVER (NUMBER(Station)=0) AND (NUMBER(WaitP)>0)
 DO
   Station^ : FROM WaitP GET Customer3[1];
   IF (WaitP:Customer3[1].WaitT = MinTime)
   DO
     SIGNAL MinNew;    # Стартиране на преизчисляването на
    MinTime
   END
   TWork^ := T + Work;
   IF Protocol
   DO
     DISPLAY ("T= %f Customer enters Server\n",T);
  END
END
```

Стратегии

```
# Край на обработката
 WHENEVER (T >= TWork) AND
   (NUMBER(Station)=1)
 DO
   Station^ : TO Sink SEND Customer3[1];
   IF Protocol
   DO
     DISPLAY ("T= %f Customer leaves server\n",T);
  END
END
```

Стратегии

Намиране на следващото най-кратко време за изчакване, ако индикатора е поставен

```
ON MinNew
DECLARE
  P (REAL)
DO PROCEDURE                          # Процедурна секция за събитие
  IF (NUMBER(WaitP) > 0)
  DO      P := WaitP:Customer3[1].WaitT;
    FOR i FROM 1 TO NUMBER(WaitP)   # Цикъл за определяне на следващото време за напускане
```

Стратегии

```
REPEAT
      IF (WaitP:Customer3[i].WaitT < P)
      DO
        P := WaitP:Customer3[i].WaitT;
      END
    END_LOOP
  END  END
DO TRANSITIONS                    # Част на преход
  MinTime^ := P;                  # Даване на стойност на
     променлива на състояние
END
```

Стратегии

```
# Чакаща работа ‚губи търпение‘
 ON ^(T >= MinTime) AND (NUMBER(WaitP) > 0)^
 DO
   WaitP^ : REMOVE Customer3{ALL
    i|WaitP:Customer3[i].WaitT <= T};
   SIGNAL MinNew;      # Стартиране на преизчисляване на
    MinTime
   IF Protocol
   DO
     DISPLAY("T= %f Waiting job leaves queue
    prematurely!\n",T);
  END
END
```

Стратегии

```
# Унищожаване на задача
  WHENEVER NUMBER(Sink) >= 4
  DO
    Sink^ : REMOVE Customer3{ALL};
    IF Protocol
    DO
     DISPLAY("T= %f 4 Customers destroyed\n",
T);
  END
END

END OF Queue3
```

Стратегии

Първо, задачите са в региона WaitP. Независимо от това променливата MinTime се свързва с най-ранното време, при което чакащата задача може да напусне опашката преждевременно и да се унищожи.
Когато нова задача влезе в опашката, MinTime трябва да се преоцени. Просто сравнение определя дали MinTime трябва да се преоцени.
Ако симулационният часовник T е достигнал MinTime, всички задачи, чието Max време за чакане е изтекло, трябва да се премахнат от опашката. По същото време стойността на MinTime трябва да се присвои отново.
Подобно MinTime трябва да се преоцени, ако задачата, която текущо определя стойността на MinTime (най-малката стойност на WaitP от всички задачи в опашката) напуска опашката, за да се обработи.

Стратегии

В следващите два случая MinTime трябва да получи най-ниската стойност на WaitT, съдържаща се във всяка задача в опашката. Не може да се определи предварително, на коя позиция в опашката може да се намери тази задача.

По тази причина опашката трябва да се претърси. Това се извършва в събитие, което е стартирано от индикатора MinNew.

Събитието съдържа процедурна част. Декларира се променлива P, която се свързва с най-ранното време за изчакване. След напускането на процедурната част тази стойност се присвоява на MinTime.

Стратегии

Стратегии и MDL функции.

В големи модели може да са нужни идентични функции на различни места. В този случай е добре да се използват MDL функции.

За да илюстрация на употребата на MDL функциите, ще се модифицира модела Queue3, така че търсенето на най-ранното време за изчакване да се извърши във функцията SearchW. Регионът ще бъде предаден като вътрешен параметър. Върнатият параметър е стойността на MinTime.

Стратегии

```
# Determine next earliest waiting time if indicator is set
ON MinNew
DO                                  # Извикване на функция
SearchW
  MinTime^ := SearchW(LOCATION WaitP);  # с регион
WaitP като параметър
END
```

Извадка от модел Queue4

Стратегии

Дефинирането на функцията SearchW е дадено в основната компонента с LOCAL DEFINITIONS:

BASIC COMPONENT Queue4

MOBILE SUBCOMPONENTS OF CLASS Customer4

LOCAL DEFINITIONS

Функция за изчисляване на следващото време за напускане

Входен параметър е регион (WaitP)

Връщания параметър е реален P (новото MinTime)

Стратегии

```
FUNCTION SearchW (LOCATION FOR Customer4 : Job --> REAL)

DECLARE
  P (REAL)
BEGIN
  IF (NUMBER(Job) > 0)
  DO
    P := Job:Customer4[1].WaitT;
    FOR i FROM 1 TO NUMBER(Job)  # Циккъл за определяне на
     следващото време за напускане
    REPEAT
```

Стратегии

```
 IF (Job:Customer4[i].WaitT < P)
      DO
        P := Job:Customer4[i].WaitT;
      END
    END_LOOP
  END
  RETURN (P);     # Команда за връщане
END_FUNC
```