ZESZYTY NAUKOWE POLITECHNIKI ŚLĄSKIEJ 1996
Seria: INFORMATYKA z. 30 Nr kol. 1315

Marek DRAGA
Piotr PECKA
Aleksander REWER

ZORIENTOWANY OBIEKTOWO JĘZYK DO OPISU MODELI KOLEJKOWYCH

Streszczenie. W artykule przedstawione jest narzędzie do opisu sieci modeli kolejkowych. Narzędziem tym jest odpowienia biblioteka klas do języka C++. W połączeniu z odpowiednim oprogramowaniem, stanowiącym jądro danej metody rozwiązywania sieci, tworzy system w pełni obiektowy, pozwalający użytkownikowi w prosty sposób budować i obliczać modele kolejkowe.

AN OBJECT ORIENTED LANGUAGE FOR QUEUEING NETWORK MODELS DESCRIPTION

Summary. The article discusses a language which was conceived to describe queueing models. It is composed of the object oriented language (C++) and a proper class library. It is full object oriented system and allows users to build a in simple way queueing networks models.

LANGUAGE ORIENTE OBJET POUR DESCRIPTION DE FILES D'ATTENTE MODELISANT LES RESEAUX DES ORDINATEURS

Résumé. L'article discute un langage pour décrire des modèles de files d'attente appliqués dans l'analyse et evaluation des performances des réseaux informatiques. Il est composé de l'angage orienté objet (C++) et d'une bibliothèque de classes. Avec ce système, un utilisateur peut facilement définir un modèle du réseau considéré.

Niniejszy tekst został opracowany częściowo w ramach Projektu Badawczego KBN nr 8 T11C 032 08.

1. Wprowadzenie

Język zapisu modelu służy użytkownikowi systemu symulacyjnego do opisu postaci modelu sieci, dla której ma być wykonany eksperyment za pomocą metod symulacyjnych czy analitycznych. Opis taki powinien w jednoznaczny sposób określić elementy, z których składa się dana sieć, oraz powiązania występujące między nimi. Składnia języka powinna być także zrozumiała i łatwa do przyswojenia dla użytkownika. Przy tworzeniu omawianego w dalszej części języka zapisu modelu przyjęto jeszcze jedno założenie - powinien on dać możliwość łatwego rozszerzenia swoich możliwości. Spełnienie tego założenia przy określonej z góry składni języka jest dość trudne, ponieważ aby przekształcić opis sieci w języku opisu modelu do postaci danych, na podstawie których może być przeprowadzona symulacja, musi istnieć procedura dokonująca wstępnej kompilacji takiego opisu. Aby ominąć ten problem, zrezygnowano z tworzenia własnego programu konwersji języka do postaci kodu pośredniego, wykorzystując do tego celu kompilator języka C++. Istnieje wiele zalet takiego rozwiązania: łatwa rozbudowa takiego systemu, alokację danych w pamięci wykonuje kompilator, co znacznie upraszcza operację na takich danych.

Programowanie obiektowe różni się od tradycyjnego wprowadzeniem pojęć klas i obiektów. Klasa zawiera definicję własności obiektów, które do niej należą. Ta definicja polega na określeniu, jakie zmienne będą wchodzić w skład obiektów danej klasy oraz jaki jest dozwolony zestaw procedur (metod), które mogą być wykonywane na zmiennych zawartych w tych obiektach. Sam obiekt jest fizyczną realizacją definicji zawartej w klasie. Do jednej klasy może należeć wiele obiektów. Powyższy opis jest oczywiście uproszczeniem, ale pozwala zrozumieć, jaka jest idea programowania obiektowego. Sieć stanowisk obsługi składa się z połączonych ze sobą obiektów należących do odpowiednich klas. Za obiekty możemy uznać stanowisko obsługi, które ma określone atrybuty (parametry), takie jak: liczba kanałów obsługi, liczba obsługiwanych aktualnie klientów oraz zestaw operacji (procedur), jakie może wykonać . Takie operacje to pobranie klienta z kolejki, obsługa klienta według określonego regulaminu czy przesłanie go do następnego stanowiska. Jako obiekty przedstawia się także źródła generujące klientów wchodzących do sieci i samych klientów. Obiektem jest też cała sieć, na której dokonujemy symulacji. Przy takim podejściu i zastosowaniu do konstrukcji języka opisu modelu jakiegoś języka obiektowego nie trzeba już konstruować analizatora składni języka opisu modelu, bo poprawność zapisu sprawdza kompilator zastosowanego języka. Nie znaczy to wcale, że aby zapisać model sieci należy znać dobrze C++, konieczna jest tylko znajomość sposobu tworzenia dozwolonych obiektów, z których składa się sieć, połączenia ich ze sobą i uruchomienia procedury symulacji, co jest omówione w dalszej części artykułu. Ponieważ traktujemy składowe sieci jako obiekty, które zawierają pewne dane i pewne metody przetwarzające te dane, to dodanie nowego elementu języka (np. nowego rodzaju stanowiska obsługi) polegać będzie na dopisaniu nowej klasy, opisującej właściwości nowego rodzaju obiektów, zgodnie z konwencją przyjętą dla istniejących już klas. To wymaga już znajomości C++, ale nie pociąga za sobą konieczności dokonywania zmian w innych elementach programu.

2. Składnia języka opisu modelu

Aby uruchomić symulację należy stworzyć plik tekstowy (tekst źródłowy programu) o strukturze jak poniżej:

```
#include network.h
main()
{

Network net1( "Net1");// delaracja sieci

// sekcja deklaracji obiektow sieci

// sekcja  laczen elementow sieci

// sekcja ustawiania parametrow generatorow

Simulate(3,1000,net1,5);//Wywolanie symulacji z odpowiednimi parametrami

}
```

2.1. Deklaracja obiektów sieci

W bloku programu jako pierwszy należy zadeklarować obiekt typu "Network", który reprezentuje sieć poddawaną symulacji. W sekcji deklaracji obiektów sieci deklarujemy obiekty składające się na symulowaną sieć. Wyróżniamy następujące typy obiektów:

- `SimpleSource` – jest to źródło generujące klientów danej klasy. Źródła klientów występują tylko w sieciach otwartych. W deklaracji obiektu typu SimpleSource podajemy jego nazwę i opcjonalnie sieć, do której stanowisko należy. Ze źródłem związany jest typ rozkładu czasu obsługi (np. SH_EXP - rozkład wykładniczy, SH_COX - rozkład Coxa). Parametry rozkładów ustalamy w sekcji ustawiania parametrów generatorów. W deklaracji obiektu typu SimpleSource należy podać również klasę klientów, jaką ma generować źródło. Domyślnie przyjmuje się klasę CL_A. Przykład deklaracji obiektu zrodlo1 typu SimpleSource o nazwie z1, należącego do sieci net1, generującego klientów klasy CL_A z wykładniczym czasem obsługi:

  ```
  SimpleSource    zrodlo1("z1" ,  SH_EXP , net1 ,Cl_A);
  ```

- Sink - jest to wyjście z sieci. Deklaracja obiektu typu Sink, podobnie jak obiektu typu SimpleSource, występuje tylko w sieciach otwartych. W deklaracji obiektu typu Sink podajemy jego nazwę i opcjonalnie sieć, do której stanowisko należy. Przykład deklaracji obiektu wyjście1 typu Sink o nazwie, należącego do sieci net1:

  ```
  Sink wyjscie1(wy1, net1);
  ```

- SimpleServer - jest to n-kanałowe stanowisko obsługi. Deklaracja obiektu typu SimpleServer zawiera nazwę stanowiska, typ regulaminu kolejki, typ rozkładu obsługi, sieć, do której stanowisko należy (opcjonalnie), liczbę kanałów stanowiska. Parametry rozkładów ustalamy w sekcji ustawiania parametrów generatorów. Przykład deklaracji obiektu stanowisko1 typu SimpleServer o nazwie st1 i 5 kanałach obsługi; kolejka do stanowiska jest typu LIFO, a obsługa w stanowisku ma rozkład Coxa:

```
SimpleServer stanowisko1 ("st1", Q_LIFO , SH_COX , net1 , 5);
```

- TokenHolder - jest to stanowisko trzymające żetony, tzn. limitujące dostęp zadań do fragmentu sieci zawartego pomiędzy stanowiskami TokenDisposer i TokenCollector. Stanowisko typu TokenHolder może występować tylko ze stanowiskami typu TokenCollector i TokenDisposer, z którymi ściśle współpracuje. W deklaracji obiektu typu TokenHolder podajemy kolejno jego nazwę, sposób przydzielania żetonów (globalnie dla wszystkich klas GLOB lub dla każdej klasy z osobna LOC), sieć do której stanowisko należy, liczbę klas objętych ograniczeniami. Przykład deklaracji obiektu token_holder1 typu TokenHolder o nazwie th1, należącego do sieci net1 i globalnym sposobie przydzielania żetonów (pula żetonów wynosi 12, a klasy objęte żetonowaniem to: CL_A, CL_B, CL_D):

```
TokenHolder token_holder1
( "th1" , GLOB , net1 , 3, 12, CL_A, CL_B, CL_D);
```

  W przypadku lokalnego sposobu przydziału żetonów, w deklaracji obiektu typu TokenHolder po parametrze określającym liczbę klas objętych żetonowaniem występują w kolejności następujące parametry: klasa objęta żetonowaniem, liczba żetonów przydzielonych danej klasie itd. Przykład deklaracji obiektu token_holder1 typu TokenHolder o nazwie th1, należącego do sieci net1 i lokalnym sposobie przydzielania żetonów (liczba żetonów przydzielonych klasom jest następująca: CL_A - 2 żetony, CL_B - 3 żetony, CL_D - 7 żetonów):

```
TokenHolder  token_holder1
("th1", LOC, ni1, 3, CL_A,2, CL_B,3, CL_D,7 );
```

- TokenDisposer - jest to stanowisko rozdzielające żetony. Stanowisko typu TokenDisposer może występować tylko ze stanowiskami typu TokenHolder i TokenCollector. Jeżeli dla zadania przychodzącego do stanowiska nie ma w danym momencie żetonów, to zadanie to ustawia się w kolejce. W deklaracji obiektu typu TokenDisposer podajemy nazwę deklarowanego obiektu i odpowiadający mu obiekt typu TokenHolder, a także opcjonalnie można podać sieć, do której stanowisko należy. Przykład deklaracji obiektu typu TokenDisposer o nazwie td1, który współpracuje z obiektem token_holder i należy do sieci net1:

```
TokenDisposer  token_disposer ("td1" , token_holder);
```

  Dla stanowiska typu TokenDisposer można określić wyjście z sieci, dokąd będą kierowane zadania, dla których w momencie przyjścia nie będzie żetonów. Powyższe zachowanie można wymusić poprzez wywołanie metody SetExitSt na rzecz obiektu typu TokenDisposer. Przykład wywołania:

```
token_disposer.SetExitSt(wyjscie1);
```

Dla powyższego wywołania klienci, którzy w momencie przyjścia do stacji token_disposer nie otrzymają żetonu, bedą przesyłani do stacji wyjście1.

- TokenCollector - jest to stanowisko zbierające żetony. Stanowisko typu TokenCollector może występować tylko ze stanowiskami typu TokenHolder i TokenDisposer. W deklaracji obiektu typu TokenCollector podajemy nazwę deklarowanego obiektu, nazwę sieci, do której stanowisko należy i odpowiadający mu obiekt typu TokenHolder. Przykład deklaracji obiektu typu TokenCollector o nazwie tc1, który współpracuje z obiektem token_holder i należy do sieci net1:

```
TokenCollector  token_collector (tc1" , net1 , token_holder);
```

- MultiServer - jest to n kanałowe stanowisko obsługi. W deklaracji stanowiska typu MultiServer podajemy: jego nazwę, typ regulaminu kolejki do tego stanowiska, sieć, do której stanowisko należy (opcjonalnie) i liczbę kanałów, jakie stanowisko posiada. Parametry rozkładów obsługi i typy rozkładów obsługi ustalamy w sekcji ustawiania parametrów generatorów. Przykład deklaracji stanowiska typu MultiServer o nazwie stanowisko nr 2, regulaminie kolejki LIFO, należącego do sieci net1 i posiadającego 3 kanały obsługi:

```
MultiServer  multi_server ("stanowisko nr 2", Q_LIFO , 3);
```

2.2. Tworzenie połączeń między obiektami sieci i ustalanie parametrów rozkładów obsługi

Po zadeklarowaniu obiektów składających się na sieć i przyłączeniu ich do sieci należy stworzyć połączenia pomiędzy obiektami. Połączenia te nazywamy tranzycjami; tworzenie połączeń pokażemy na następującym przykładzie. Niech obiekt A dla klientów klasy 0 ma być połączony z obiektem B z prawdopodobieństwem 0.3 i z obiektem C z prawdopodobieństwem 0.7, a dla klientów klasy 1 z obiektem D z prawdopodobieństwem 0.5 i z obiektem C z prawdopodobieństwem 0.5. W celu stworzenia tych połączeń należy napisać sekwencję poniższych instrukcji:

```
A.SetTransition ( B , 0.3 , 0 );
A.SetTransition ( C , 0.7 , 0 );
A.SetTransition ( D , 0.5 , 1 );
A.SetTransition ( C , 0.5 , 1 );
```

Metoda SetTransition wywołana jest na rzecz obiektu A. Pierwszym parametrem tej metody jest obiekt, z którym ma zostać stworzone połączenie. Kierunek tego połączenia jest od obiektu, na rzecz którego metoda została wywołana, do obiektu, który jest parametrem tej metody. Drugim parametrem tej metody jest prawdopodobieństwo łączenia (jest to prawdopodobieństwo, z jakim wysyłane są zadania z obiektu, na rzecz którego wywoływana jest metoda SetTransition, do obiektu, który jest parametrem tej metody). Suma prawdopodobieństw łączenia dla wszystkich połączeń utworzonych z jednego obiektu musi być równa 1.0. Trzecim parametrem metody SetTransition jest numer klasy, której dotyczy połączenie.

Po stworzeniu połączeń między obiektami należy określić parametry rozkładów (dotyczy to tylko obiektów typu SimpleSource, SimpleServer i MultiServer). Dla obiektów typu SimpleSource i SimpleServer parametry rozkładów podajemy w następujący sposób (przyjmijmy, że stanowisko obsługi, dla którego ustalamy parametry rozkładu, zostało zadeklarowane jako SO1):

- rozkład wykładniczy o wartości średniej 2.0

  ```
  SO1.SetScheduleParam(1,2.0);
  ```

  Pierwszy parametr powyższej metody jest nieistotny dla symulacji, lecz musi to być liczba całkowita, drugim parametrem jest średnia, z jaką dany generator ma generować wartości.

- rozkład Erlanga o trzech fazach; średnia każdej fazy ma wartość 2.5

  ```
  SO1.SetScheduleParam(3,2.5);
  ```

  Pierwszy parametr określa liczbę faz a drugi to wartość średnia każdej z faz.

- rozkład Coxa o dwóch fazach; pierwsza faza ma średnią 2.1 , druga faza występuje z prawdopodobieństwem 0.5 i ma średnią 3.5

  ```
  SO1.SetScheduleParam(2 , 2.1 , 0.5 , 3.5);
  ```

  Pierwszy parametr określa liczbę faz, drugi parametr to średnia pierwszej fazy, trzeci parametr to prawdopodobieństwo przejścia do drugiej fazy, a czwarty to średnia drugiej fazy. Jeżeli w rozkładzie występuje więcej faz, to parametry należy podawać w następującej kolejności: prawdopodobieństwo wystąpienia danej fazy i wartość średnia tej fazy.

- rozkład hiperwykładniczy drugiego rzędu o prawdopodobieństwie wylosowania liczby zgodnie z pierwszym rozkładem 0.3 i średniej tego rozkładu 3.5, prawdopodobieństwo wylosowania liczby zgodnie z drugim rozkładem 0.7 ; średnia tego rozkładu wynosi 6.5

  ```
  SO1.SetScheduleParam(2 , 0.3 , 3.5 , 0.7 , 6.5);
  ```

- rozkład deklarowany przez użytkownika. Generator ma losować dwie liczby - pierwszą 5.0 z prawdopodobieństwem 0.25 i drugą 2.5 z prawdopodobieństwem 0.75

  ```
  SO1.SetScheduleParam(2 , 0.25 , 5.0 , 0.75 , 2.5);
  ```

  Pierwszy parametr to liczba wartości losowanych przez generator, drugi parametr to prawdopodobieństwo wylosowania wartości będącej następnym parametrem itd. W rozkładzie nie ma ograniczeń dotyczących liczby wartości losowanych przez generator. Rozkłady Erlanga, Coxa nie posiadają ograniczeń dotyczących liczby faz rozkładu, również rząd rozkładu hiperwykładniczego jest nieograniczony.

Ustawienie parametrów rozkładów występujących w stanowiskach typu MultiServer i InfinitServer wykonujemy następująco:

- klasa zerowa klientów jest obsługiwana zgodnie z rozkładem Coxa o parametrach jak w powyższym przykładzie,
- klasa pierwsza jest obsługiwana zgodnie z rozkładem wykładniczym o średniej 2.5,
- klasa druga jest obsługiwana zgodnie z rozkładem hiperwykładniczym o parametrach jak w powyższym przykładzie,
- klasa trzecia jest obsługiwana zgodnie z rozkładem Erlanga o trzech fazach, z których każda ma wartość średnią równą 3.4,

```
SO1.SetScheduleParam(0 , SH_COX , 2 , 2.1 , 0.5 , 3.5);
SO1.SetScheduleParam(1 , SH_ExP , 1 , 2.5);
SO1.SetScheduleParam(2 , SH_HIP , 2 , 0.3 , 3.5 , 0.7 , 6.5);
SO1.SetScheduleParam(3 , SH_ERL ,3 ,  3.4);
```

Jeżeli dla którejś z klas nie został określony typ rozkładu obsługi za pomocą metody SetScheduleParam, to klasa ta domyślnie jest obsługiwana zgodnie z rozkładem zadeklarowanym dla klasy zerowej.

Ponadto w sieciach zamkniętych należy przed uruchomieniem symulacji umieścić klientów w sieci. W tym celu należy na rzecz stanowiska, które ma przyjąć klientów, wywołać metodę `SO1.AddNewClients ( 0 ,10 )`. Oznacza to, że w kolejce stanowiska SO1 ma zostać umieszczonych dziesięć zadań klasy zerowej. Klasy klientów mogą być oznaczane na dwa sposoby:

- jako liczby typu integer np. 0,1,2 ...
- jako wartości typu wyliczeniowego Classes, przyjmujące odpowiednio następujące wartości: CL_A, CL_B, CL_C itd.

2.3. Wywoływanie symulacji

Po zadeklarowaniu wszystkich obiektów sieci i stworzeniu połączeń pomiędzy nimi należy uruchomić symulację poprzez napisanie słowa Simulate z odpowiednimi parametrami. Kolejne parametry wywołania symulacji oznaczają: liczbę przebiegów symulacji, czas trwania pojedynczego przebiegu symulacji, sieć poddaną symulacji, czas rozbiegu symulacji, poziom ufności, z którym mają być obliczane wyniki (parametr ten jest typu wyliczeniowego o następującej postaci: `enum ProbLevel { p_9, p_95, p_975, p_99,p_995, p_999 };`), nazwę pliku, do którego mają być zapisane wyniki.

Przykład wywołania symulacji: `Simulate( 4 , 10000.0 , siec1 , 500.0 , p_95 , wyniki.wyn" );`

Biblioteki klas służące symulacji zostały skompilowane i umieszczone w archiwum netsym.a. W celu uruchomienia badań symulacji należy skompilować program źródłowy z opisem sieci (plik źródłowy powinien mieć rozszerzenie cc) i skonsolidować go z istniejącym archiwum. W powyższym celu piszemy następującą komendę:

```
CC   -w   mm1.cc   netsym.a   -o   mm1.out
```

gdzie:

-w	-opcja kompilacji bez drukowania ostrzeżeń,
mm1.cc	-nazwa pliku źródłowego zawierającego opis sieci,
netsym.a	-nazwa archiwum zawierającego skompilowane biblioteki klas,
-o	-opcja pozwalająca użytkownikowi na nadanie własnej nazwy dla pliku wynikowego,
mm1.out	- nazwa pliku wynikowego.

Po skompilowaniu i skonsolidowaniu należy uruchomić program wynikowy i oczekiwać na wyniki symulacji. Utworzony został również skrypt pozwalający zautomatyzować uruchamianie symulacji. Skrypt nosi nazwę Simu, a jego parametrem jest nazwa pliku źródłowego z opisem sieci. Przykład wywołania:

```
Simu  mm1.cc
```

3. Przykłady

3.1. Stanowisko typu M/Cox/1

Stanowisko tego typu to połączone ze sobą źródło, generujące klientów w odstępach czasu wyznaczanych przez rozkład wykładniczy i stanowisko obsługujące klientów przez czas określony rozkładem Coxa.

```
Network ni1("NI1");
Sink sn1("SINK1", ni1);
SimpleSource sc1("zrodlo", SH_EXP, ni1, CL_A, 0);
sc1.SetSchParam(1,2.0);
SimpleServer ss1("stanowisko", SH_COX, Q_FIFO, ni1, 1);
ss1.SetSchParam(2, 0.5, 0.5, 1.0);
sc1.SetTransition(ss1, 1.0, CL_A) ;
ss1.SetTransition(sn1, 1.0, CL_A) ;
Simulate(50,1000.0,ni1,p_95,"mcox1.sym");
```

3.2. Model transmisji pakietów w ścieżce wirtualnej sieci komputerowej

Sieć zawiera trzy jednokanałowe stanowiska obsługi typu SimpleServer (rys.1), w których występuje ograniczenie liczby klientów. Jeżeli w stanowisku aktualnie przebywa 12 klientów i do stanowiska przybywa nowy klient, zostaje on usunięty z sieci (symulacja przepełnienia bufora). Ponadto sieć zawiera 6 źródeł generujących klientów. Źródła o numerach parzystych generują klientów o wyższym priorytecie, natomiast źródła o numerach nieparzystych generują klientów o niższym priorytecie. Przed stanowiskami ustawiają się kolejki o regulaminie priorytetowym (LIFO_PM). Klienci generowani przez źródła wirt1 i wirt2 przechodzą przez całą sieć zgodnie z drogą zaznaczoną ciągłą linią, natomiast

klienci generowani przez źródła loc1,loc2, loc3, loc4, loc5, loc6 przechodzą tylko przez jedno stanowisko obsługi i następnie wychodzą z sieci.

Zapis sieci w języku zapisu modelu:

```
main()
{
Network ni1("NI1");

Sink        sn1("SINK1", ni1);
Sink        sn2("SINK2", ni1);

SimpleSource wirt1("wirt1", SH_EXP, ni1, CL_A, 0);
     wirt1.SetSchParam(1, 3.0);
SimpleSource wirt2("wirt2", SH_EXP, ni1, CL_B, 1);
     wirt2.SetSchParam(1, 5.0);
SimpleSource loc1("loc1", SH_COX, ni1, CL_C, 0);
     loc1.SetSchParam(2, 2.5,0.5,5.0);
SimpleSource loc2("loc2", SH_COX, ni1, CL_D, 1);
     loc2.SetSchParam(2, 2.5,0.5,5.0);
SimpleSource loc3("loc3", SH_COX, ni1, CL_E, 0);
     loc3.SetSchParam(2, 2.5,0.5,5.0);
SimpleSource loc4("loc4", SH_COX, ni1, CL_F, 1);
     loc4.SetSchParam(2, 2.5,0.5,5.0);
SimpleSource loc5("loc5", SH_COX, ni1, CL_G, 0);
     loc5.SetSchParam(2, 5.0,0.5,10.0);
SimpleSource loc6("loc6", SH_COX, ni1, CL_H, 1);
     loc6.SetSchParam(2, 5.0,0.5,10.0);
SimpleServer ss1("SS1", SH_USR, Q_FIFO_PM , ni1, 1);
               ss1.SetSchParam( 1, 1.0 , 1.0);
SimpleServer ss2("SS2", SH_USR, Q_FIFO_PM , ni1, 1);
     ss2.SetSchParam( 1, 1.0, 1.0);
SimpleServer ss3("SS3", SH_USR, Q_FIFO_PM , ni1, 1);
     ss3.SetSchParam( 1, 1.0, 1.0);

TokenHolder  th1("Th1",LOC,ni1,4,CL_A,4,CL_B,2,CL_C,4,CL_D,2);
TokenHolder  th2("Th2",LOC,ni1,4,CL_A,4,CL_B,2,CL_E,4,CL_F,2);
TokenHolder  th3("Th3",LOC,ni1,4,CL_A,4,CL_B,2,CL_G,4,CL_H,2);

TokenDisposer  td1("Td1",ni1,th1);
td1.SetExitSt(sn2);
TokenDisposer  td2("Td2",ni1,th2);
td2.SetExitSt(sn2);

TokenDisposer  td3("Td3",ni1,th3);
td3.SetExitSt(sn2);
```

```
TokenCollector  tc1("Tc1",ni1,th1);
TokenCollector  tc2("Tc2",ni1,th2);
TokenCollector  tc3("Tc3",ni1,th3);

wirt1.SetTransition(td1, 1.0, CL_A) ;
wirt2.SetTransition(td1, 1.0, CL_B) ;
loc1.SetTransition(td1, 1.0, CL_C) ;
loc2.SetTransition(td1, 1.0, CL_D) ;

td1.SetTransition(ss1, 1.0, CL_A) ;
td1.SetTransition(ss1, 1.0, CL_B) ;
td1.SetTransition(ss1, 1.0, CL_C) ;
td1.SetTransition(ss1, 1.0, CL_D) ;

loc3.SetTransition(td2, 1.0, CL_E) ;
        loc4.SetTransition(td2, 1.0, CL_F) ;

td2.SetTransition(ss2, 1.0, CL_A) ;
td2.SetTransition(ss2, 1.0, CL_B) ;
td2.SetTransition(ss2, 1.0, CL_E) ;
td2.SetTransition(ss2, 1.0, CL_F) ;

loc5.SetTransition(td3, 1.0, CL_G) ;
loc6.SetTransition(td3, 1.0, CL_H) ;

td3.SetTransition(ss3, 1.0, CL_A) ;
td3.SetTransition(ss3, 1.0, CL_B) ;
td3.SetTransition(ss3, 1.0, CL_G) ;
td3.SetTransition(ss3, 1.0, CL_H) ;

ss1.SetTransition(tc1,1.0 ,CL_A) ;
ss1.SetTransition(tc1,1.0 ,CL_B) ;
ss1.SetTransition(tc1,1.0 ,CL_C) ;
ss1.SetTransition(tc1,1.0 ,CL_D) ;

tc1.SetTransition(td2,1.0 ,CL_A) ;
tc1.SetTransition(td2,1.0 ,CL_B) ;
tc1.SetTransition(sn1,1.0 ,CL_C) ;
tc1.SetTransition(sn1,1.0 ,CL_D) ;

ss2.SetTransition(tc2,1.0 ,CL_A) ;
ss2.SetTransition(tc2,1.0 ,CL_B) ;
ss2.SetTransition(tc2,1.0 ,CL_E) ;
ss2.SetTransition(tc2,1.0 ,CL_F) ;
```

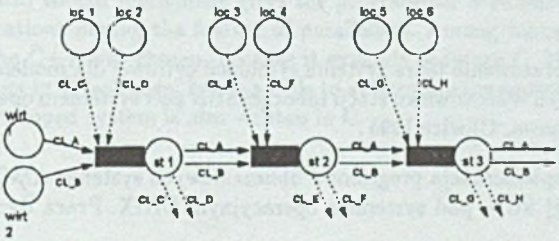
```
tc2.SetTransition(td3,1.0 ,CL_A) ;
tc2.SetTransition(td3,1.0 ,CL_B) ;
tc2.SetTransition(sn1,1.0 ,CL_E) ;
tc2.SetTransition(sn1,1.0 ,CL_F) ;

ss3.SetTransition(tc3,1.0 ,CL_A) ;
ss3.SetTransition(tc3,1.0 ,CL_B) ;
ss3.SetTransition(tc3,1.0 ,CL_G) ;
ss3.SetTransition(tc3,1.0 ,CL_H) ;

tc3.SetTransition(sn1,1.0 ,CL_A) ;
tc3.SetTransition(sn1,1.0 ,CL_B) ;
tc3.SetTransition(sn1,1.0 ,CL_G) ;
tc3.SetTransition(sn1,1.0 ,CL_H) ;

Simulate(50,2000.0,ni1,5000.0,p_95,"siec_o_loc.sym");

}
```



Rys. 1. Model sieci stanowisk o ograniczonej kolejce
Fig. 1. Queueing network with limited queue

4. Uwagi końcowe

Celem artykułu było przedstawienie oprogramowania pośredniczącego do budowy systemów kolejkowych. Dotychczas połączono je z jądrem symulacji [1] oraz z jądrem analitycznej metody MVA [2], uzyskując efektywne narzędzie do rozwiązywania modeli kolejkowych. Prowadzone są badania nad opracowaniem nowego jądra do symulacji, opartego na mechanizmie wątków, które są częścią takich nowoczesnych systemów operacyjnych,

jak: Solaris 2.0, Linux, HP Unix, OS2. Połączenie techniki obiektowej z mechanizmem wątków daje programiście mocne narzędzie do tworzenia aplikacji, dla których równoległość jest naturalna. Symulacja z pewnością zalicza się do tego typu aplikacji. Język C++ nie został wybrany przypadkowo. Jak wiadomo, jest to rozszerzenie języka C o cechy charakterystyczne dla języków, przy zapewnieniu kompatybilności z językiem C. Ponieważ większość systemów operacyjnych została napisana w C, a funkcje systemowe dostarczane są w tym języku, dlatego łatwo wywołuje się je z języka C++.

Dalszy rozwój tego oprogramowania może przebiegać w dwóch kierunkach. Pierwszy z nich to rozbudowa możliwości samej symulacji poprzez tworzenie nowych typów stanowisk i schematów obsługi. Drugi kierunek to budowanie innych metod rozwiązywania sieci stanowisk obsługi zapisanych przy użyciu przedstawionego wcześniej języka zapisu modelu. Przyjęte założenie o obiektowej konstrukcji programu wydaje się być jego podstawową zaletą. Także język zapisu modelu nie jest skomplikowany i można go łatwo opanować. Niestety, obiektowe podejście ma także pewne wady. Pierwszą z nich jest zajmowanie większej, niż w tradycyjnym programowaniu, przestrzeni w pamięci komputera, a drugą wolniejsze działanie. Jednak przy stale rosnących wielkościach pamięci i mocy obliczeniowej komputerów wady te mają mniejsze znaczenie. Tak więc wydaje się, że korzyści płynące z przyjętego przez nas założenia przeważają nad wadami.

LITERATURA

[1] Draga M.: Opracowanie jądra systemu symulacji cyfrowej dla modelowania systemów komputerowych w środowisku stacji roboczej SUN pod systemem operacyjnym UNIX. Praca dyplomowa, Gliwice 1995.

[2] Meres M.: Implementacja programów obliczeniowych systemu AMOK w środowisku stacji roboczej SUN, pod systemem operacyjnym UNIX. Praca dyplomowa, Gliwice 1995.

[3] Rewer A.: Opracowanie oprogramowania pośredniczącego systemu symulacji cyfrowej dla modelowania systemów komputerowych w środowisku stacji roboczej Sun.Praca dyplomowa, Gliwice 1995.

[4] Czachórski T.: Modele kolejkowe systemów komputerowych. Skrypt Politechniki Śląskiej nr 1844, Gliwice 1994.

[5] Dokumentacja systemu SunOS 5.4.

[6] Dokumentacja kompilatora SPARCompiler 4.0.1.

[7] Coad P.: Programowanie obiektowe. Oficyna Wydawnicza READ ME, Warszawa 1993.

Recenzent: Dr inż. Przemysław Szmal

Wpłynęło do Redakcji 4 stycznia 1995

Abstract

The paper describes a part of software aiming to build queuing system models. Composed with kernel for simulation and kernel for numerical methode called MVA, it gives a useful tool for solving queueing networks. The work is in progress and a new kernel is being developed. It is based on multi thread mechanism which is a part of such modern operating systems as Solaris 2.0, Linux, Hp-Unix, OS2. Composition of object oriented technique and multi thread mechanism gives the programmer a powerfull tool to create a group of applications having the feature of parallelism. Among many OOP programmig languages, the C++ was chosen, because it expands ordinary C. Most of operating systems are written in C language, hence a calls to system library procedures are easy to perform if the developed system is also written in C.