ZESZYTY NAUKOWE POLITECHNIKI ŚLĄSKIEJ 1989
Seria: TRANSPORT z. 13 Nr kol. 1034

Roman KONIECZNY

ZASOBY SYMULACYJNE JĘZYKA LOGLAN

Streszczenie. Artykuł niniejszy ma na celu prezentację loglanowskich zasobów symulacyjnych, dla potrzeb realizacji symulacji dyskretnych zdarzeń. Typ SIMULATION zaimplementowany w LOGLANie jest językiem problemowym (dialektem) pozwalającym na pisanie programów symulujących systemy rzeczywiste. Jest on wzorowany na klasie SIMULATION z SIMULI-67 lecz różni się od niej w następujących punktach:
1) Struktura danych użyta do szeregowania zdarzeń zapewnia pesymistyczny koszt operacji rzędu O(log n), gdzie n jest liczbą zaplanowanych zdarzeń.
2) Wprowadzone narzędzia pozwalają osiągnąć niedeterminizm w programie symulacyjnym.

1. Uwagi wstępne

Przedstawiony m.in. w publikacjach [16] i [9] język SIMULA-67, oprócz swych niewątpliwych zalet oraz wkładu w rozwój teorii programowania, ma też pewne wady i niedogodności, jak np.:

- nienowoczesna składnia (niedogodną dla piszących kompilator),
- ograniczenia w zakresie stosowania prefiksowania,
- jednoprocesorowa koncepcja sprzętu,
- przestarzałe języki wbudowane: SIMSET (do działań na zbiorach skończonych reprezentowanych przez listy dwukierunkowe) i SIMULATION - o operacjach bardziej kosztownych, niż jest to konieczne...

Dodatkową - można to określić "wadą" - jest brak dobrych kompilatorów języka SIMULA-67 w Polsce.

Wszystkie wymienione wyżej aspekty stały się powodem podjęcia prac nad LOGLANem w roku 1977, przez zespół pracowników Instytutu Informatyki Uniwersytetu Warszawskiego - kierowany przez prof. Andrzeja Salwickiego. W roku 1977 powstała pierwsza wersja - LOGLAN-77, a następnie wersja LOGLAN-82 - zaimplementowana na komputer MERA-400 pracujący pod kontrolą systemu operacyjnego SOM-3M (IIUW). W chwili obecnej - biorąc pod uwagę szerokie rozpowszechnianie się w kraju komputerów z rodziny IBM PC, dokonano implementacji LOGLAN-u na ten właśnie typ komputera, rozszerzając jego możliwości m.in. o elementy grafiki komputerowej.

Cytując za prof. Salwickim [25] można napisać: "*W odróżnieniu od bardziej rozpowszechnionych w kraju języków programowania, LOGLAN jest*

językiem opisu i implementacji systemów. Oczywistą częścią każdego systemu są algorytmy, dlatego LOGLAN zawiera w sobie doświadczenie takich języków jak PASCAL, ALGOL, FORTRAN, SIMULA, ADA i inne. Język umożliwia tworzenie różnorodnych struktur danych procesów, współprogramów i całych systemów o bogatych strukturach hierarchicznych. Wartość handlowa LOGLANu wynika ze znacznego przyspieszenia prac nad oprogramowaniem i z tego, że zwiększa on pewność prawidłowej pracy zaprojektowanego systemu. Posługując się LOGLANem można znacznie szybciej niż dotąd opisać i zrealizować systemy symulacji różnorodnych zjawisk, pakiety do projektowania i analizy układów scalonych o wielkiej skali integracji (VLSI), oprogramowanie tomografii komputerowej, pakiety grafiki komputerowej, różnorodne oprogramowanie robotów przemysłowych, systemy operacyjne operujące techniką ikonograficzną, systemy zarządzania bazami danych i in.

Istotną rolę odgrywa tu możliwość pracy z obiektami klas i korzystanie z hierarchicznej struktury deklaracji klas rozszerzanych dzięki prefiksowaniu, tj. specyficznej technice składania modułów programu. Prefiksowanie wprowadzone po raz pierwszy w SIMULI zdobywa sobie rosnące uznanie, m.in. w takich językach jak SMALLTALK czy PARAGON."

Aktualnie istnieje już szereg publikacji poświęconych LOGLANowi. Publikacje te można podzielić na: dotyczące teorii programowania oraz dotyczące aplikacji. Zagadnieniom teoretycznym poświęcone są m.in. następujące pozycje: [1,7,10,11,13,15,17,18,21,22,23,25,27,28,29], zagadnieniom aplikacyjnym: [2,3,4,5,6,8,12,14,20, 26].

Artykuł niniejszy ma na celu prezentację loglanowskich zasobów symulacyjnych, dla potrzeb realizacji symulacji dyskretnych zdarzeń. Typ SIMULATION zaimplementowany w LOGLANie jest językiem problemowym (dialektem) pozwalającym na pisanie programów symulujących systemy rzeczywiste. Jest on wzorowany na klasie SIMULATION z SIMULI-67 [16], [9] lecz różni się od niej w następujących punktach [23]:

1) Struktura danych użyta do szeregowania zdarzeń zapewnia pesymistyczny koszt operacji rzędu O(log n), gdzie n jest liczbą zaplanowanych zdarzeń.
2) Wprowadzone narzędzia pozwalają osiągnąć niedeterminizm w programie symulacyjnym.

W typie SIMULATION użyte zostały narzędzia pozwalające na quasirównoległe wykonywanie programu. Procesy są reprezentowane przez współprogramy. Modelowany system jest reprezentowany, jako zbiór współbieżnych i współdziałających procesów. Proces jest ciągiem uporządkowanych w czasie zdarzeń związanych z działaniem symulowanego obiektu. Wartość lokalnych atrybutów procesu określa jego stan. Wystąpienie zdarzenia reprezentowane jest przez zmianę stanu procesu, a zatem zmianę stanu systemu.

2. Ogólny schemat zasobów symulacyjnych LOGLANu

Wszystkie zdarzenia (events) w symulowanym systemie są uporządkowane w czasie. Jest to osiągnięte przez umieszczenie znaczników zdarzeń (event notices) w kolejce priorytetowej reprezentującej oś symulowanego czasu systemowego. Do planowania i porządkowania zdarzeń służą operacje: HOLD, SCHEDULE, CANCEL, PASSIVATE i RUN.

Ogólny schemat loglanowskiego typu SIMULATION jest następujący:

```
unit FIFO: class; ... ;
unit PRIORITYQUEUE: FIFO class ; ... ;
unit SIMULATION: PRIORITYQUEUE class ;
  ......
  var  MAINPR: MAINPROGRAM, ...
  unit SIMPROCESS:  FIFOEL coroutine ; ... ;
  unit MAINPROGRAM: SIMPROCESS class ; ... ;
  unit TIME:        function : REAL ; ... ;
  unit CURRENT:     function : SIMPROCESS ; ... ;
  unit SCHEDULE:    procedure (P: SIMPROCESS, T: REAL) ; ... ;
  unit HOLD:        procedure (T: REAL) ; ... ;
  unit PASSIVATE:   procedure ; ... ;
  unit RUN:         procedure (P: SIMPROCESS) ; ... ;
  unit CANCEL:      procedure (P: SIMPROCESS) ; ... ;
begin
......
end SIMULATION;
```

Program symulacyjny jest blokiem posiadającym prefiks SIMULATION.

2.1. Klasa FIFO - kolejki proste

Klasa FIFO implementuje kolejki proste. Ogólna struktura tej klasy jest następująca:

```
unit FIFO: class; (* kolejka prosta *)
  hidden FRONT, REAR;
  signal FIFOEMPTY;
  var    FRONT, REAR: FIFOEL;

  unit FIFOEL: class;
    var SUCC: FIFOEL;
    unit INTO: procedure (Q: FIFO); ...  end INTO;
    end FIFOEL;

  unit OUTFIRST: procedure; ...          end OUTFIRST;

  unit EMPTY:    function: BOOLEAN; ...  end EMPTY;
```

```
  unit FIRST:    function: FIFOEL; ...    end FIRST;
  unit CARDINAL: function: INTEGER; ...   end CARDINAL;
end FIFO;
```

Typ FIFO definiuje narzędzia umożliwiające tworzenie kolejek prostych. Elementami kolejki są obiekty typu FIFOEL. Użytkownik może tworzyć kolejki poprzez prefiksowanie typu ich elementów typem FIFOEL. Przykładowe użycie zasobów klasy FIFO może być następujące: niech dany będzie blok prefiksowany klasą FIFO oraz deklaracja

```
var Q: FIFO, X: FIFOEL, N: INTEGER, Y: FIFOEL, ... ;
  begin
    Q := new FIFO;   X := new FIFOEL; ... ;
  end;
```

Bezpośrednio po obu powyższych instrukcjach kolejka Q jest pusta; element X pozostaje poza nią. Następnie można wykonać następujące czynności:

- wstawić obiekt (element) X na koniec kolejki Q :

```
call X.INTO(Q);
```

- usunąć pierwszy element z kolejki Q :

```
call Q.OUTFIRST;
```

- wskazać pierwszy element w kolejce Q (a ściślej podać referencję do tego obiektu) :

```
Y := Q.FIRST;  teraz poprzez Y można uzyskać dostęp do atrybutów pierwszego obiektu w kolejce Q ;
```

- sprawdzić, czy kolejka Q jest pusta:

```
Q.EMPTY = TRUE (gdy tak) ;
```

- określić liczbę elementów (obiektów) w kolejce Q :

```
N := Q.CARDINAL ;
```

Podstawowymi atrybutami klasy FIFO są FRONT i REAR. Atrybuty te są typu FIFOEL. FRONT wskazuje na pierwszy obiekt w kolejce, REAR na ostatni. Atrybuty te są niedostępne (hidden) dla programisty. Oprócz tego, w klasie FIFO zadeklarowany jest sygnał FIFOEMPTY, wysyłany przez procedurę OUTFIRST w przypadku gdy kolejka jest pusta. Poniżej podano listingi poszczególnych modułów klasy FIFO.

2.1.1. Klasa FIFOEL

```
unit FIFOEL: class;   (*  FIFOEL - element kolejki  *)
   var SUCC: FIFOEL;
   unit INTO: procedure (Q: FIFO);  (*  wstaw do kolejki  *)
      begin
         if Q.FRONT = none
            then Q.FRONT,      Q.REAR := this FIFOEL
            else Q.REAR.SUCC, Q.REAR := this FIFOEL
         fi
      end INTO;
   end FIFOEL;
```

Atrybutem klasy FIFOEL jest SUCC typu FIFOEL (wskazujący następny element kolejki, zaimplementowanej jako lista jednokierunkowa). Klasa ta posiada lokalną procedurę INTO umożliwiającą wstawienie elementu na koniec kolejki wskazanej przez parametr Q.

2.1.2. Procedura OUTFIRST

```
unit OUTFIRST: procedure; (* usun pierwszy element z kolejki *)
   begin
      if FRONT = NONE then raise FIFOEMPTY else
         if REAR = FRONT then REAR,FRONT := NONE
            else FRONT:=FRONT.SUCC
         fi
      fi
   end OUTFIRST;
```

W przypadku, gdy zmienna FRONT nie wskazuje żadnego obiektu, wysyłany jest sygnał FIFOEMPTY i procedura kończy działanie, w przeciwnym wypadku zmienna FRONT wskazywać będzie następny istniejący w kolejce obiekt jako pierwszy.

2.1.3. Funkcja EMPTY

```
unit EMPTY: function: BOOLEAN;   (* ? kolejka pusta *)
   begin
         RESULT := FRONT = NONE
   end EMPTY;
```

Wartość funkcji jest TRUE, gdy zmienna FRONT nie wskazuje żadnego obiektu (kolejka pusta).

2.1.4. Funkcja FIRST

```
unit FIRST: function: FIFOEL; (* pierwszy element w kolejce *)
   begin
        RESULT := FRONT
   end FIRST;
```

Wartością funkcji jest referencja do obiektu wskazywanego przez zmienną FRONT. (Wartością funkcji będzie NONE jeżeli kolejka jest pusta).

2.1.5. Funkcja CARDINAL

```
unit CARDINAL: function: INTEGER;
   VAR I  : INTEGER,  (* liczba elementów w kolejce *)
        AUX: FIFOEL;
   begin
      AUX := FRONT;
```

```
        while AUX =/= NONE do
            I:=I+1;
            AUX:=AUX.SUCC
        od;
        RESULT := I
    end CARDINAL;
```

Wartością funkcji jest liczba elementów obecnych w kolejce. Funkcja ta wykorzystuje zmienne pomocnicze AUX typu FIFOEL oraz I typu INTEGER do zliczania elementów w kolejce.

2.1.4. Przykład programu

Poniżej podano przykład programu wykorzystującego zasoby klasy FIFO :

```
program TESTFIFO; (* TEST KOLEJEK PROSTYCH *)

(*$L-*)
(* .... Tutaj jest obecny tekst klasy FIFO *)
(*$L+*)

begin
  pref FIFO block:
  var  Q: FIFO, X: FIFOEL, N: INTEGER, Y,Z: FIFOEL;
   begin
    Q:=new FIFO; X:=new FIFOEL, Z:=new FIFOEL;
    writeln("Wstawianie elementu 1 ...");
    call X.INTO(Q); N:=Q.CARDINAL;
    writeln("... w kolejce ma byc 1 element...        JEST: ",N);
    writeln("Wstawianie elementu 2 ...");
    call Z.INTO(Q); N:=Q.CARDINAL;
    writeln("... w kolejce maja byc 2 elementy...     JEST: ",N);
    writeln("Usuwanie elementu 1 ...");
    call Q.OUTFIRST; N:=Q.CARDINAL;
    writeln("... w kolejce ma byc 1 element ...       JEST: ",N);
    writeln("Usuwanie elementu 2 ...");
    call Q.OUTFIRST; N:=Q.CARDINAL;
    writeln("... w kolejce ma byc zero elementow ... JEST: ",N);
    if Q.EMPTY then writeln("KOLEJKA Q JEST PUSTA") fi;
    Y:=Q.FIRST;
    if Y=NONE then writeln("KOLEJKA Q JEST NAPRAWDE PUSTA") fi;
   end;
end.
```

Przebieg wykonania przykładowego programu jest następujący:

```
IIUW LOGLAN-82  Concurrent Executor  Version 4.36
May 21, 1988
(C)Copyright  Institute of Informatics, University of Warsaw

Wstawianie elementu 1 ...
... w kolejce ma byc 1 element...          JEST:     1
Wstawianie elementu 2 ...
... w kolejce maja byc 2 elementy...       JEST:     2
Usuwanie elementu 1 ...
... w kolejce ma byc 1 element ...         JEST:     1
Usuwanie elementu 2 ...
... w kolejce ma byc zero elementow ...    JEST:     0
KOLEJKA Q JEST PUSTA
KOLEJKA Q JEST NAPRAWDE PUSTA
```

```
End of LOGLAN-82 program execution
```

Typ FIFO ze względów technicznych jest użyty jako prefiks typu PRIORITYQUEUE (nie jest tam wykorzystywany). Zadeklarowany w nim typ oraz procedury i funkcje są dzięki temu dostępne w blokach lub typach prefiksowanych typem PRIORITYQUEUE. Dotyczy to w szczególności programu symulacyjnego.

2.2. Klasa PRIORITYQUEUE - kolejki priorytetowe

Klasa PRIORITYQUEUE implementuje kolejki priorytetowe. Ogólna struktura tej klasy jest następująca:

```
unit PRIORITYQUEUE: FIFO class; (* kolejka priorytetowa *)
     hidden NODE;

     unit QUEUEHEAD: class;
          hidden LAST, ROOT;
          var    LAST, ROOT: NODE;

          unit MIN:      function: ELEM; ...          end MIN;

          unit INSERT:   procedure(R: ELEM); ...      end INSERT;

          unit DELETE:   procedure(R: ELEM); ...      end DELETE;

          unit CORRECT:  procedure(R: ELEM, DOWN: BOOLEAN); ...
                                                      end CORRECT;
       end QUEUEHEAD;

     unit NODE: class (EL: ELEM);
          var  LEFT,RIGHT,UP: NODE, NS: INTEGER;
          unit LESS: function(X: NODE): BOOLEAN; ... end LESS;
       end NODE;

     unit ELEM: class(PRIOR: REAL);
        var LAB: NODE;
        unit virtual LESS: function(X: ELEM): BOOLEAN;
                                                ...   end LESS;
      end ELEM;

   end PRIORITYQUEUE;
```

Elementami kolejki priorytetowej są obiekty typu ELEM uporządkowane według atrybutu PRIOR. Użytkownik może tworzyć kolejki priorytetowe z własnymi elementami, które są prefiksowane klasą ELEM. Można ustalać sposób uporządkowania obiektów (inny niż standardowy) przez zadeklarowanie w odpowiadającym im typie nowej (wirtualnej) funkcji LESS.

Typ QUEUEHEAD reprezentuje jedną kolejkę priorytetową. Kolejka priorytetowa jest zaimplementowana za pomocą kopca (heap), co daje pesymistyczny czas operacji rzędu O(log n), gdzie n jest liczbą elementów kolejki. Elementem najmniejszym kolejki jest korzeń kopca.

Zakładając, że dany jest blok prefiksowany klasą QUEUEHEAD (tj. pref QUEUEHEAD block:) oraz deklaracja

```
var  X, R: ELEM,  Q: QUEUEHEAD,  PRIOR1: REAL;
begin
   PRIOR1 := 2.5;  R := new ELEM(PRIOR1); Q := new QUEUEHEAD;
   ...
end;
```

Bezpośrednio po powyższych instrukcjach możliwe są następujące czynności do wykonania:

- wstawienie elementu R do kolejki Q (w odpowiednie miejsce):

```
call Q.INSERT(R);
```

- wskazanie na najmniejszy element kolejki Q :

```
X := Q.MIN;  (poprzez zmienną referencyjną X możliwy jest
              teraz dostęp do atrybutów najmniejszego ele-
              mentu kolejki)
( X = NONE  gdy kolejka Q jest pusta );
```

- usunięcie elementu R z kolejki Q :

```
call Q.DELETE(R);
```

Klasa PRIORITYQUEUE skonstruowana jest jako kopiec (tj. drzewo binarne połączonych ogniw, poczynając od korzenia (ROOT)). Elementami (ogniwami) kopca są obiekty typu NODE (węzły) sprzężone z obiektami typu ELEM, które stanowią prefiks dla elementów użytkownika umieszczanych w kolejce priorytetowej. Obiekty typu NODE są zasłonięte (hidden) i niedostępne z zewnątrz klasy PRIORITYQUEUE. Są one podstawą do tworzenia struktury połączeń w obrębie kopca.

Poniżej podano listingi poszczególnych modułów klasy PRIORITYQUEUE.

2.2.1. Klasa QUEUEHEAD

Klasa ta zapewnia narzędzia umożliwiające dokonywanie operacji na kopcu. Operacjami tymi są: wskazanie elementu najmniejszego, wstawienie nowego elementu, usunięcie elementu oraz korekcja kopca. Zmiennymi globalnymi w tej klasie są ROOT oraz LAST typu NODE. Zmienne te są zasłonięte. Wartością ROOT jest referencja do korzenia kopca (pierwszego elementu), wartością LAST jest referencja do elementu ostatniego.

Nagłówek tej klasy jest następujący:

```
unit QUEUEHEAD: class;
   hidden LAST, ROOT;
   var    LAST, ROOT: NODE;
```

2.2.1.1. Funkcja MIN

```
unit MIN: function: ELEM;  (*   najmniejszy   *)
   begin
      if ROOT=/= NONE then RESULT:=ROOT.EL fi;
   end MIN;
```

Wartością funkcji jest referencja do najmniejszego elementu kopca.
(ROOT = NONE , gdy kopiec jest pusty).

2.2.1.2. Procedura INSERT

```
unit INSERT: procedure(R:ELEM);   (*  wstaw  *)
   var X,Z: NODE;
   begin
      X:=R.LAB;
      if LAST = NONE then
         ROOT:=X; ROOT.LEFT,ROOT.RIGHT,LAST:=ROOT
       else
         if LAST.NS = 0 then
            LAST.NS:=1;
            Z:=LAST.LEFT;  LAST.LEFT:=X;
            X.UP:=LAST; X.LEFT:=Z; Z.RIGHT:=X
         else
            LAST.NS:=2;
            Z:=LAST.RIGHT; LAST.RIGHT:=X; X.RIGHT:=Z;
            X.UP:=LAST;  Z.LEFT:=X; LAST.LEFT.RIGHT:=X;
            X.LEFT:=LAST.LEFT; LAST:=Z;
         fi;
      fi;
      call CORRECT(R,FALSE);
   end INSERT;
```

Procedura ta służy do włączenia nowego elementu typu ELEM w strukturę kopca. Ostatnią czynnością jest wywołanie procedury korekcji kopca CORRECT.

2.2.1.3. Procedura DELETE

```
unit DELETE: procedure(R: ELEM);   (*--- usun  ---*)
   var X,Y,Z: NODE;
   begin
      X:=R.LAB;
      if X=ROOT and ROOT.NS=0 then
         ROOT,LAST:= NONE
      else
         Z:=LAST.LEFT;
         if LAST.NS =0 then
            Y:= Z.UP; Y.RIGHT:= LAST;
            LAST.LEFT:=Y; LAST:=Y;
          else
            Y:= Z.LEFT; Y.RIGHT:= LAST;  LAST.LEFT:= Y;
         fi;
         Z.EL.LAB:=X;  X.EL:= Z.EL;
         LAST.NS:= LAST.NS-1;
         R.LAB:=Z; Z.EL:=R;
         if X.LESS(X.UP) then
            call CORRECT(X.EL,FALSE)
          else
```

```
            call CORRECT(X.EL,TRUE)
         fi;
      fi;
   end DELETE;
```

Procedura ta służy do usuwania elementu typu ELEM z kopca. Ostatnią czynnością jest wywołanie procedury korekcji kopca.

2.2.1.4. Procedura CORRECT

```
unit CORRECT: procedure(R:ELEM,DOWN:BOOLEAN);
  (* korekcja kopca *)
   var X,Z: NODE, T: ELEM, FIN,LOG: BOOLEAN;
   begin
      Z:=R.LAB;
      if DOWN then
         while not FIN do
            if Z.NS =0 then
               FIN:=TRUE
            else
               if Z.NS=1 then
                  X:=Z.LEFT
               else
                  if Z.LEFT.LESS(Z.RIGHT) then
                     X:=Z.LEFT
                  else X:=Z.RIGHT
                  fi
               fi;
               if Z.LESS(X) then
                  FIN:=TRUE
               else
                  T:=X.EL; X.EL:=Z.EL; Z.EL:=T;
                  Z.EL.LAB:=Z; X.EL.LAB:=X
               fi
            fi;
            Z:=X;
         od
      else
         X:=Z.UP;
         if X=NONE then LOG:=TRUE
                   else LOG:=X.LESS(Z); fi;
         while not LOG do
            T:=Z.EL; Z.EL:=X.EL; X.EL:=T;
            X.EL.LAB:=X; Z.EL.LAB:=Z;
            Z:=X; X:=Z.UP;
            if X=NONE then LOG:=TRUE
                      else LOG:=X.LESS(Z) fi;
         od;
      fi;
   end CORRECT;
```

Procedura ta służy do korekcji struktury kopca naruszonej przez wstawienie lub usunięcie elementu. Korekcja polega na ponownym wyważeniu drzewa binarnego, jakim jest kopiec. (UWAGA: W aktualnej wersji modułu PRIORITYQUEUE procedura ta nie jest zasłonięta).

2.2.2. Klasa NODE

```
unit NODE: class (EL: ELEMD;   (* węzeł kopca *)
      var LEFT,RIGHT,UP: NODE,  NS:INTEGER;
      unit LESS: function(X: NODE): BOOLEAN;
          begin
             if X= NONE then RESULT:=FALSE
                        else RESULT:=EL.LESS(X.EL) fi;
          end LESS;
   end NODE;
```

Elementy tej klasy opisują węzły kopca. Atrybutami klasy NODE są: LEFT (lewy), RIGHT (prawy), UP (do góry) typu NODE. Wartości tych zmiennych są referencjami do węzłów sąsiednich. Zmienna pomocnicza NS służy do oznaczania węzłów. Składnikiem klasy NODE jest funkcja porządkująca LESS.

2.2.3. Klasa ELEM

```
unit ELEM: class(PRIOR: REAL);   (* element kopca *)
   var LAB: NODE;
   unit virtual LESS: function(X: ELEMD: BOOLEAN;
       begin
          if X=NONE then RESULT:= FALSE else
                         RESULT:= PRIOR< X.PRIOR fi;
       end LESS;
    begin
       LAB:= new NODE(this ELEMD;
    end ELEM;
```

Obiekty tej klasy stanowią prefiks dla informacji przechowywanej w węźle. Obiekty użytkownika umieszczane w kolejce priorytetowej muszą być prefiksowane klasą ELEM. Atrybutem tej klasy jest zmienna LAB typu NODE, przechowująca wartość referencji do węzła sprzężonego z danym elementem. Składnikiem klasy ELEM jest wirtualna funkcja porządkująca LESS.

2.2.4. Przykład programu

Poniżej podano listing przykładowego programu posługującego się kolejkami priorytetowymi:

```
program TESTPRIOR; (* TEST KOLEJEK PRIORYTETOWYCH *)
(*$L-*)
(* ..... Tutaj jest treść klas FIFO i PRIORITYQUEUE *)
(*$L+*)
begin
 pref PRIORITYQUEUE block;
 var  A,B,C: ELEM,  X,Y,Z: OBIEKT,  Q1,Q2: QUEUEHEAD,
      PRIOR1,PRIOR2: REAL;
 unit OBIEKT: ELEM class(NUMER: INTEGER);
   begin (* Klasa OBIEKT jest prefiksowana klasa ELEM *)
     (* ... Tutaj moga byc instrukcje obiektu klasy OBIEKT *)
   end OBIEKT;

 begin
  PRIOR1:=10.5; PRIOR2:=6.2;
  (* ... Generowanie obiektow ... *)
  A:=new ELEM(PRIOR1);      B:=new ELEM(PRIOR2);
  X:=new OBIEKT(PRIOR1,1); Y:=new OBIEKT(PRIOR2,2);
```

```
  Q1:=new QUEUEHEAD;          Q2:=new QUEUEHEAD;
  (* ... Wstawianie obiektow do kolejek ... *)
  call Q1.INSERT(A); call Q1.INSERT(B);
  call Q2.INSERT(X); call Q2.INSERT(Y);
  C:=Q1.MIN;   Z:=Q2.MIN;
  writeln("KOLEJKA  Q1  EL.MIN. (PRIOR)         ",C.PRIOR);
  writeln("KOLEJKA  Q2  EL.MIN. (PRIOR,NUMER) ",Z.PRIOR,Z.NUMER);
  (* ... Usuwanie obiektow z kolejek ... *)
  call Q1.DELETE(C);  call Q2.DELETE(Z);
  C:=Q1.MIN;   Z:=Q2.MIN;
  writeln("KOLEJKA  Q1  EL.MIN. (PRIOR)         ",C.PRIOR);
  writeln("KOLEJKA  Q2  EL.MIN. (PRIOR,NUMER) ",Z.PRIOR,Z.NUMER);
 end;
end TESTPRIOR.
```

Przebieg wykonania programu jest następujący:

```
IIUW LOGLAN-82  Concurrent Executor  Version 4.35
May 21, 1988
(C)Copyright  Institute of Informatics, University of Warsaw

KOLEJKA  Q1  EL.MIN. (PRIOR)            5.2000
KOLEJKA  Q2  EL.MIN. (PRIOR,NUMER)      5.2000        2
KOLEJKA  Q1  EL.MIN. (PRIOR)           10.5000
KOLEJKA  Q2  EL.MIN  (PRIOR,NUMER)     10.5000        1

End of LOGLAN-82 program execution
```

2.3. Klasa SIMULATION

Ogólny schemat tej klasy podano na początku niniejszego rozdziału. Nagłówek klasy SIMULATION jest następujący:

```
unit SIMULATION: PRIORITYQUEUE class;
   taken QUEUEHEAD, ELEM, FIFOEL;
   hidden PQ, CURR, EVENTNOTICE, MAINPROGRAM, CHOICEPROCESS;
   var    CURR:    SIMPROCESS,   (* proces aktywny *)
          PQ:      QUEUEHEAD,    (* oś czasu       *)
          MAINPR:  MAINPROGRAM;
```

Typ SIMULATION definiuje wszystkie niezbędne narzędzia do prowadzenia symulacji dyskretnych zdarzeń. Przykładowo, niech P będzie zmienną referencyjną wskazującą na proces A (P:= new PROCES_A). Wykorzystując zasoby klasy SIMULATION można wykonać następujące czynności:

- uzyskać informację, czy proces jest zakończony lub zawieszony:

  P.IDLE = TRUE (gdy tak);

- uzyskać informację, czy proces jest zakończony:

  P.TERMINATED = TRUE (gdy tak);

- uzyskać informację na kiedy proces ma zaplanowane zdarzenie:

  CZAS_WZNOWIENIA := P.EVTIME ;

- wstrzymać proces na czas DT :

  call HOLD(DT) ;

- zawiesić proces aktywny:

```
call PASSIVATE ;
```

- przekazać sterowanie z bieżącego procesu aktywnego do innego procesu:

```
call RUN(P2) ;
```

- zawiesić inny proces:

```
call CANCEL(P3) ;
```

- ustawić procesowi nowe zawiadomienie na osi czasu:

```
call SCHEDULE(P,t) ;
```

- uzyskać informację o procesie aktywnym:

```
PA := CURRENT.P ;
```

- odczytać bieżący czas symulacyjny:

```
y := TIME;
```

2.3.1. Współprogram SIMPROCESS

```
unit SIMPROCESS: FIFOEL coroutine; (* proces symulacyjny *)
      var EVENT,  (* najbliższe zdarzenie *)
          EVENTPOM: EVENTNOTICE,
          FINISH:   BOOLEAN;
      signal TERMPROC, IDLEPROC;
      unit IDLE: function: BOOLEAN;
         begin
            RESULT:= EVENT= NONE;
         end;
      unit TERMINATED: function :BOOLEAN;
         begin
            RESULT:= FINISH;
         end;
      unit EVTIME: function: REAL; (* czas aktywacji  *)
         begin
            if IDLE then raise IDLEPROC  fi;
            RESULT:= EVENT.EVENTTIME;
         end;
    handlers
   (* przyjęte handlery dla sygnałów TERMPROC AND IDLEPROC *)
      when TERMPROC: writeln(" SIMPROCESS IS TERMINATED ");
                     attach(MAINPR);
      when IDLEPROC: writeln(" SIMPROCESS IS IDLE ");
                     attach(MAINPR);
    end HANDLERS;
    begin
      return;
      inner;
      FINISH:=TRUE;
      call PASSIVATE;
      raise TERMPROC
    end SIMPROCESS;
```

Obiekty typu SIMPROCESS reprezentują procesy symulacyjne. Są one współprogramami (coroutine), co pozwala na ich działanie w systemie quasi-równoległym. Użytkownik może deklarować własne procesy przez prefiksowanie ich typem SIMPROCESS. Instrukcje odpowiadające takim procesom są wstawiane w miejsce symbolu inner w treści typu SIMPROCESS. Zmienna EVENT wskazuje najbliższe zdarzenie jakie ma zajść w danym procesie. Wbudowane funkcje pomocnicze: IDLE, TERMINATED i EVTIME dostarczają

dodatkowych informacji o stanie procesu. Wykonanie akcji każdego procesu polega na wykonaniu instrukcji napisanych przez programistę, a następnie na wywołaniu procedury PASSIVATE. Ponowne przekazanie sterowania do takiego procesu spowoduje błąd (zgłaszany jest sygnał TERMPROC).

Każdy proces (z punktu widzenia realizacji programu symulacyjnego) znajduje się w danej chwili w jednym z czterech stanów:

AKTYWNYM - gdy wykonywane są jego instrukcje (wtedy P.EVTIME wskazuje bieżącą wartość czasu symulacyjnego);

WSTRZYMANYM - gdy ma zaplanowaną chwilę, w której ma być uaktywniony (posiada zawiadomienie w kolejce priorytetowej PQ - odwzorowującej oś czasu);

ZAWIESZONYM - gdy nie ma określonej chwili, w której będzie wznowiony (nie ma zawiadomienia na osi czasu);

ZAKOŃCZONYM - gdy wykonane zostały już wszystkie jego instrukcje.

UWAGA: W programie symulacyjnym nie należy używać operacji attach i detach w odniesieniu do procesów (tj. obiektów typu SIMPROCESS).

2.3.2. Klasa EVENTNOTICE

```
unit EVENTNOTICE: ELEM class;
   var EVENTTIME: REAL, PROC: SIMPROCESS;
   unit virtual LESS: function(X: EVENTNOTICE):BOOLEAN;
      begin
         if X=NONE then RESULT:= FALSE else
               RESULT:= EVENTTIME< X.EVENTTIME or
               (EVENTTIME=X.EVENTTIME and PRIOR< X.PRIOR);
         fi;
      end LESS;
   end EVENTNOTICE;
```

Z każdym procesem związany jest obiekt typu EVENTNOTICE (niedostępny dla użytkownika). Wartością zmiennej EVENTTIME jest moment, w którym zajdzie zdarzenie w procesie wskazywanym przez zmienną PROC. Dzięki prefiksowi ELEM obiekty typu EVENTNOTICE mogą być elementami kolejki priorytetowej. Są w niej uporządkowane ze względu na atrybuty (EVENTTIME, PRIOR). Takie uporządkowanie umożliwia losową kolejność wykonywania akcji procesów, które mają zdarzenia zaplanowane na tę samą chwilę.

2.3.3. Klasa MAINPROGRAM

```
unit MAINPROGRAM: SIMPROCESS class;
   begin
      do ATTACH(MAIN) od;
   end;
```

Klasa ta implementuje moduł główny programu (master) jako proces.

2.3.4. Funkcja TIME

```
unit TIME: function: REAL; (* biezacy czas symulacyjny *)
   begin
      RESULT:=CURRENT.EVTIME
   end;
```

2.3.5. Funkcja CURRENT

```
unit CURRENT: function: SIMPROCESS;
   begin
      RESULT:=CURR;
   end;
```

Funkcja ta wskazuje pierwszy (bieżący) proces na osi czasu.

2.3.6. Procedura SCHEDULE

```
unit SCHEDULE: procedure(P: SIMPROCESS, T: REAL);
   (* aktywacja procesu P w chwili T *)
   begin
      if T<TIME THEN T:= TIME fi;
      if P=CURRENT then
         call HOLD(T-TIME)
       else
         if P.IDLE and P.EVENTPOM=NONE then
            P.EVENT,P.EVENTPOM:= new EVENTNOTICE(RANDOM);
            P.EVENT.PROC:= P;
          else
            if P.IDLE then
               P.EVENT:= P.EVENTPOM;
               P.EVENT.PRIOR:=RANDOM;
             else
               P.EVENT.PRIOR:=RANDOM;
               call PQ.DELETE(P.EVENT)
            fi
         fi;
         P.EVENT.EVENTTIME:= T;
         call PQ.INSERT(P.EVENT);
      fi;
   end SCHEDULE;
```

Procedura SCHEDULE umożliwia zaplanowanie następnej chwili uaktywnienia procesu P. Chwila ta jest równa max(TIME, T). Jeśli P jest procesem aktywnym (current), to wykonanie procedury SCHEDULE(P,T) jest równoważne wywołaniu HOLD(T-TIME). Jeśli proces posiadał już zawiadomienie to, jest ono kasowane i tworzone jest nowe zawiadomienie na chwilę czasową max(TIME, T). Jeżeli P nie jest procesem aktywnym, to proces aktywny nie ulega zmianie.

2.3.7. Procedura HOLD

```
unit HOLD: procedure(T: REAL);-- wstrzymaj
   begin
      call PQ.DELETE(CURRENT.EVENT);
      CURRENT.EVENT.PRIOR:=RANDOM;
      if T<0 THEN T:=0; fi;
      CURRENT.EVENT.EVENTTIME:=TIME+T;
      call PQ.INSERT(CURRENT.EVENT);
      call CHOICEPROCESS;
   end HOLD;
```

Procedura ta przerywa wykonywanie procesu aktywnego z jednoczesnym zaplanowaniem następnej chwili jego uaktywnienia na chwilę TIME+T (jeśli T<0 to przyjmuje się T=0). Nowy proces aktywny zostaje wybrany losowo spośród procesów zaplanowanych na najmniejszą chwilę względem chwili równej TIME.

2.3.8. Procedura PASSIVATE

```
unit PASSIVATE: PROCEDURE;
   begin
     call PQ.DELETE(CURRENT.EVENT);
     CURRENT.EVENT:=NONE;
     call CHOICEPROCESS
   end PASSIVATE;
```

Procedura ta przerywa wykonywanie procesu aktynego zawieszając go, czyli pozbawiając zawiadomienia. Nowy proces aktywny zostaje wybrany analogicznie, jak przy procedurze HOLD. Gdy kolejka priorytetowa jest pusta, to tworzone jest zawiadomienie dla procesu MAINPR - czyli bloku głównego programu symulacyjnego.

2.3.9. Procedura RUN

```
unit RUN: procedure(P: SIMPROCESS);
   begin
      CURRENT.EVENT.PRIOR:=RANDOM;
      if not P.IDLE then
         P.EVENT.PRIOR:=0;
         P.EVENT.EVENTTIME:=TIME;
         call PQ.CORRECT(P.EVENT,FALSE)
       else
         if P.EVENTPOM=NONE then
            P.EVENT,P.EVENTPOM:=new EVENTNOTICE(0);
            P.EVENT.EVENTTIME:=TIME;
            P.EVENT.PROC:=P;
            call PQ.INSERT(P.EVENT)
          else
            P.EVENT:=P.EVENTPOM;
            P.EVENT.PRIOR:=0;
            P.EVENT.EVENTTIME:=TIME;
            P.EVENT.PROC:=P;
```

```
            call PQ.INSERT(P.EVENT);
          fi
        fi;
        call CHOICEPROCESS;
      end RUN;
```

Procedura ta powoduje przerwanie wykonywania procesu aktywnego (który nie traci swojego zawiadomienia) z jednoczesnym uaktywnieniem procesu P (niezależnie od tego czy proces P posiadał zawiadomienie). Zawiadomienie dla procesu P zajmuje wtedy pozycję najmniejszego zawiadomienia w kolejce priorytetowej.

2.3.10. Procedura CANCEL

```
unit CANCEL: procedure(P: SIMPROCESS);
    begin
      if P= CURRENT then call PASSIVATE else
        call PQ.DELETE(P.EVENT);
        P.EVENT:=NONE;
      fi;
    end CANCEL;
```

Procedura ta powoduje zlikwidowanie zawiadomienia dla procesu P, o ile proces ten posiadał zawiadomienie. Jeśli proces P nie posiadał zawiadomienia, to wywołanie call CANCEL(P) jest instrukcją pustą. Wywołanie call CANCEL(CURRENT) równoważne jest wywołaniu call PASSIVATE.

2.3.11. Procedura CHOICEPROCESS

```
unit CHOICEPROCESS: procedure
  (* wybór pierwszego procesu z kolejki PQ do aktywacji   *)
    var P: SIMPROCESS;
    begin
      P:=CURR;
      if PQ.MIN= NONE then
        writeln(" EMPTY QUEUE");
        MAINPR.EVENT:=MAINPR.EVENTPOM;
        MAINPR.EVENT.PRIOR:=0;
        MAINPR.EVENT.EVENTTIME:=TIME;
        call PQ.INSERT(MAINPR.EVENT);
        CURR:=MAINPR;
        attach(MAINPR)
      else
        CURR:=PQ.MIN qua EVENTNOTICE.PROC;
        attach(CURR)
      fi;
  end CHOICEPROCESS;
```

Procedura ta służy do wyboru pierwszego procesu znajdującego się w kolejce priorytetowej PQ (na osi czasu) celem jego uaktywnienia.

2.3.12. Część główna modułu SIMULATION

```
begin
    PQ:=new QUEUEHEAD;  (* generowanie osi czasu *)
    CURR,MAINPR:=new MAINPROGRAM;
    MAINPR.EVENT,MAINPR.EVENTPOM:=new EVENTNOTICE(0);
    MAINPR.EVENT.EVENTTIME:=0;
    MAINPR.EVENT.PROC:=MAINPR;
    call PQ.INSERT(MAINPR.EVENT);
    inner;
    PQ:=NONE;
end SIMULATION;
```

W bloku głównym klasy SIMULATION realizowane są następujące czynności: generowanie osi czasi PQ (jako kolejki priorytetowej), generowanie procesu głównego MAINPR reprezentującego program najbardziej zewnętrzny blok programu symulacyjnego (proces ten jest w pierwszej kolejności uaktywniany). W trakcie kompilacji programu słowo kluczowe inner zostaje zastąpione instrukcjami modułu głównego programu użytkownika.

2.3.13. Przykład programu symulacyjnego

Przykłady programów symulacyjnych w języku LOGLAN, bazujących na zasobach klasy SIMULATION zawierają m.in. publikacje: [3], [4], [8], [23], [25], [27].

3. Procedury i funkcje pomocnicze

Do grupy procedur i funkcji pomocniczych, niezbędnych do realizacji przebiegów symulacyjnych, zaliczyć można: standardowe procedury wejścia/wyjścia, procedury grafiki komputerowej, procedury obsługi manipulatora kulowego (MOUSE), oraz generatory pseudolosowe. Procedury te zostały szczegółowo omówione w następujących publikacjach: [12], [14], [26], [28] i [29] .

4. Uwagi końcowe

Przedstawione w artykule zasoby symulacyjne języka LOGLAN stanowią bardzo silne narzędzie dla naukowca-programisty zajmującego się techniką symulacji komputerowej.

Moduł SIMULATION, którego autorami są pp. W. Bartol, D. Szczepańska oraz doc. A. Kreczmar z Instytutu Informatyki Uniwersytetu Warszawskiego, może być łatwo rozbudowany o nowe moduły, a także o elementy niezbędne do prowadzenia symulacji wielokomputerowej. Idee i koncepcje zawarte w loglanowskim module SIMULATION mogą być również implementowane na bazie innych języków programowania.

Na zakończenie podkreślić tutaj należy znaczący wkład autorów języka LOGLAN w rozwój obiektowego podejścia do programowania oraz techniki symulacyjnej w Polsce.

LITERATURA

[1] BARTOL W.: Programowanie za pomocą współprogramów - Informatyka nr 6 / 1983 .

[2] KOŁODZIEJSKA H.: Implementacja operacji na tekstach w języku LOGLAN - Sprawozdania Instytutu Informatyki Uniwersytetu Warszawskiego, Warszawa 1984 .

[3] KONIECZNY R. (praca zbiorowa): Zastosowanie języka LOGLAN do modelowania dużych systemów transportowych na przykładzie modelu ruchu pociągów - praca naukowo-badawcza NB-277/RT/87 wykonana w ramach programu RI.P.09 *"Rozwój języków, metod i podstaw formalnych oprogramowania"*. Instytut Transportu Politechniki Śląskiej, Katowice 1987 .

[4] KONIECZNY R.: Język programowania LOGLAN jako narzędzie modelowania systemów transportowych - Zagadnienia Transportu, Wyd. PAN, Warszawa nr 3/4 1986/87 .

[5] KONIECZNY R.: Przegląd zasobów języka LOGLAN dla potrzeb modelowania systemów transportowych - Automatyka Kolejowa, nr 7, 9 / 1988 .

[6] KONIECZNY R.: Charakterystyka zasobów języka LOGLAN dla potrzeb modelowania systemów transportowych - Zeszyt Naukowy Politechniki Śląskiej, seria Transport, nr 9 / 1989 .

[7] KONIECZNY R.: Specyfikacja formalna w projektowaniu programów komputerowych - *(niniejszy zeszyt)*

[8] KONIECZNY R.: Język programowania LOGLAN jako narzędzie symulacji systemu transportowego - *(niniejszy zeszyt)*

[9] KONIECZNY R.: Zasoby symulacyjne języka SIMULA-67 - *(niniejszy zeszyt)*

[10] KRECZMAR A., SALWICKI A.: Język programowania LOGLAN - Informatyka nr 7,8 / 1982, 1 / 1983 .

[11] KRECZMAR A.: Język programowania LOGLAN 82 - Materiały II Konferencji Użytkowników Minikomputera MERA-400, Gdańsk 1984

[12] KRECZMAR A.: Dokumentacja dla minikomputera MERA-400 - Język programowania LOGLAN-82 - Podstawowe konstrukcje i cechy charakterystyczne języka - IIUW, 1984 .

[13] KRECZMAR A.: Języki obiektowe - Informatyka nr 1-2 / 1988

[14] KOSTON H.: Generatory rozkładów pseudolosowych w LOGLANIE - praca magisterska, Instytut Informatyki Uniwersytetu Warszawskiego, 1987 (opiekun naukowy: prof. A. Salwicki)

[15] MULDNER T.: Pewne uwagi o dwóch nowych językach programowania wysokiego poziomu LOGLAN i ADA - Biuletyn Techniczny MERA nr 11-12 1980 .

[16] OKTABA H., RATAJCZAK W.: SIMULA 67 - WNT Warszawa 1980.

[17] OKTABA H.: Klasy w LOGLANIE - Informatyka nr 5 / 1983 .

[18] OKTABA H.: Prefiksowanie klasami w LOGLANIE - Informatyka nr 6 / 1983 .

[19] PERKOWSKI P.: Technika symulacji cyfrowej. WNT, Warszawa 1980 .

[20] SALWICKI A.: LOGLAN - narzędzie produkcji oprogramowania - Materiały II Konferencji Użytkowników Minikomputera MERA- 400, Gdańsk 1984 .

[21] SALWICKI A.: Metodologia programowania w LOGLANIE - Materiały III Konferencji Użytkowników Minikomputera MERA-400 , Gdańsk 1985 .

[22] SZAŁAS A., SZCZEPAŃSKA-WASERSZTRUM D.: Exception handling in parallel computations - Sprawozdania Instytutu Informatyki Uniwersytetu Warszawskiego, Warszawa 1984.

[23] SZCZEPAŃSKA D.: Narzędzia symulacyjne - art. poz. [25] .

[24] WINKOWSKI J.: Programowanie symulacji procesów . WNT, Warszawa 1974.

[25] *Język programowania LOGLAN 82 - Materiały Jesiennej Szkoły PTI - Serock, 1985* .

[26] *LOGLAN - USER'S GUIDE (version JANUARY '88) IIUW, Warsaw. (Suplement dla IBM-PC)*
[27] *Materiały: International Summer School of the Programming Language LOG.4N. ZABORÓW, POLAND September, 5-10.1983, IIUW*
[28] *Report on the programming language LOGLAN. Praca zbiorowa, wyd. Instytut Informatyki UW - 1982 .*
[29] *Report on the LOGLAN 82 programming language. Warszawa - Łódź, 1984, PWN .*

SIMULATION RESOURCES OF LOGLAN LANGUAGE

Summary

The present paper aims at presenting LOGLAN simulation resources for the needs of discrete events simulation. The SIMULATION type implemented in LOGLAN is a problem language (dialect) allowing to write the programs that simulate real systems.

It is modelled after the SIMULATION class from SIMULA-67 language but is different from this language in the following points:

1) Data strukture used for arranging (scheduling) the events provides pessimistic cost of the operation of about O (log n) where n is the number of planned events.
2) Introducted instruments allow to achieve indeterminism in the simulation program.

SIMULATIONSVORRÄTE DER SPRACHE LOGLAN

Zusammenfassung

Vorliegender Aufsatz hat die Darstellung von Simulationsvorräten der Sprache LOGLAN zum ziel. Diese Vorräte dienen der Realisierung von Simulation diskreter Ereignisse. Der Typ SIMULATION, der in LOGLAN implementiert wurde, ist eine problemorientierte Sprache (Dialekt), die das Schreiben von Programmen erlaubt, die reale Systeme simulieren. Dieser Typ hat die Klasse SIMULATION von der Sprache SIMULA-67 zum Muster, aber er unterscheidet sich von dieser in folgenden Punkten:

1) die Datenstruktur, die zur Einreihung von Ereignissen eingesetzt wurde, sichert pesymistische Operationskosten der Große O (log n) wo n die Zahl der planierten Ereignisse ist.
2) Eingeführte Werkzeuge erlauben, Undeterminiertheit im

Simulationsprogramm zu erreichen.

СИМУЛЯЦИОННЫЕ РЕСУРСЫ ЯЗЫКА LOGLAN

Резюме

В статье представлены ресурсы симуляционные языка LOGLAN для нужд симуляции дискретных событий. Тип SIMULATION имплементированый в LOGLANe является проблемным языком (диялектом), дающим возможность писать симуляционные программы для реальных систем. Язык этот подражает класс SIMULATION взятый из SIMULa-67. Разница состоит в следующем:

1) Структура данных бзята для упорядочения событий обеспечивает пессимистическую себестоимость операции порядка O (log n), где n число планируемых событий.
2) Введенные инструменты дают возможность получить недетерминизм в симуляционной программе.