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

Roman KONIECZNY

ZASOBY SYMULACYJNE JĘZYKA SIMULA-67

Streszczenie. W artykule przedstawione zostały podstawowe zasoby języka SIMULA-67, który jest językiem uniwersalnym wysokiego poziomu z wbudowanym językiem do symulacji. SIMULA pozwala na łatwe dostosowanie do różnorodnych wyspecjalizowanych problemów i może być użyta jako baza języków problemowych.

Na zasoby symulacyjne języka SIMULA-67 składają się: klasy systemowe SIMSET i SIMULATION oraz zestaw procedur pomocniczych. Klasa *SIMULATION* definiuje środki pozwalające na pisanie programów symulujących systemy rzeczywiste. Zagadnienia przedstawione w niniejszym artykule stanowią jedynie pewien szkic specyfikujący podstawowe zasoby warstwy symulacyjnej języka SIMULA.

1. Uwagi wstępne

Język SIMULA-67 powstał w Norweskim Ośrodku Obliczeniowym w Oslo w 1967 roku - jako wynik współpracy Ole-Johan Dahla, Bjorna Myhrhauga i Kristena Nygaarda. Język ten jest rozszerzeniem ALGOLU 60. Początkowo lokalny - znany tylko w Skandynawii - język ten zdobył zasłużoną popularność w Europie Zachodniej, USA i ZSRR. Jest zaimplementowany na komputerach: IBM 360, IBM 370, Univac 1100, PDP DEC10, CDC i in.

SIMULA jest językiem uniwersalnym wysokiego poziomu z wbudowanym językiem do symulacji. Uniwersalność SIMULI wynika z listy najróżniejszych programów i systemów zaprogramowanych w tym języku: od baz danych, kompilatorów, systemów operacyjnych poprzez programy administracyjne, transportowe, sieciowe, systemy opisu sprzętu, systemy czasu rzeczywistego, programy i systemy statystyczne i socjologiczne do programów symulujących systemy biologiczne, społeczne, komunikacyjne, telefoniczne, rynkowe i programów numerycznych [4].

SIMULA pozwala na łatwe dostosowanie do różnorodnych wyspecjalizowanych problemów i może być użyta jako baza języków problemowych.

Podstawowym pojęciem w języku SIMULA-67 jest *obiekt*. Pojęcie to wywodzi się z tzw. jednostki dynamicznej tworzonej podczas wykonywania programu napisanego w ALGOLU 60. Jak wiadomo, dynamicznymi jednostkami programu algolowego mogą być jednostki bloków lub procedur. Każda z nich ma lokalną strukturę danych oraz listę instrukcji do wykonania. Podobnie każdemu obiektowi jest przyporządkowana struktura danych oraz lista instrukcji.

Obiekty mają jednak cechy odróżniające je od algolowych jednostek dynamicznych. Jedną z ważniejszych własności obiektu jest to, że może występować on jako wartość zmiennej. Dzięki temu nawet po wykonaniu wszystkich instrukcji obiektu możliwe jest korzystanie z jego lokalnej struktury danych. (W przypadku bloku lub procedury lokalna struktura danych jednostki dynamicznej zanika po przeprowadzeniu wszystkich obliczeń w jej treści.)

Obiekt może być zarówno podmiotem, jak i przemiotem obliczeń. Podmiotem jest wówczas, gdy procesor wykonuje instrukcje wewnątrz obiektu. Rola obiektu jest w tedy taka sama jak rola podbloku lub jednostki procedury. Z drugiej strony, obiekt jako wartość zmiennej może stać się argumentem działań, czyli przedmiotem obliczeń. Wówczas procesor działając z zewnątrz obiektu, korzysta z jego lokalnej struktury danych. Funkcje podmiot-przedmiot mogą być dla obiektu wymienne, tzn. w pewnej fazie programu obiekt może być podmiotem obliczeń, a już w następnej pełni rolę przedmiotu, na którym są dokonywane działania. W języku istnieją ponadto mechanizmy pozwalające na przerywanie obliczeń prowadzonych w treści obiektu oraz ponowne ich wznawianie. Umożliwia to to tworzenie systemów obiektów współpracujących, zwanych współprogramami (ang. *coroutines*).

Formalnie każdy obiekt można opisać określając zbiór jego atrybutów oraz listę instrukcji w nim zawartych. Atrybuty obiektu wyznaczają jego lokalną strukturę danych, natomiast instrukcje przedstawiają sposób realizacji zadań przypisanych obiektowi. Za pomocą obiektów można budować złożone struktury danych jak np. listy, kolejki, stosy, drzewa itp. Obiekty mogą być wartościami atrybutów innych obiektów. Pozwala to na budowanie złożonych struktur obiektowych.

Na zasoby symulacyjne języka SIMULA-67 składają się: klasy systemowe SIMSET i SIMULATION oraz zestaw procedur pomocniczych.

2. Klasa SIMSET

Klasa SIMSET opisuje strukturę danych złożoną z list symetrycznych. W klasie SIMSET zdefiniowane są narzędzia do reprezentacji zbiorów skończonych. Każdy taki zbiór można przedstawić jako listę symetryczną - zwaną również łańcuchem. W łańcuchu jest wyróżnione jedno ogniwo (*head*) utożsamiające zbiór, natomiast pozostałe ogniwa (*link*) przedstawiają elementy tego zbioru. Dla każdego ogniwa określono ogniwo następujące po nim (*SUC*) oraz ogniwo poprzedzające (*PRED*). Dzięki tym powiązaniom możliwe jest wykonanie operacji *dołącz element do zbioru*, *usuń element ze zbioru* i innych.

Ogólny schemat deklaracji klasy *SIMSET* ma następującą postać:

```
class SIMSET
  begin
    class linkage; ... ;
    linkage class head; ... ;
    linkage class link; ... ;
  end ;
```

Klasa *linkage* opisuje własności wszystkich ogniw łańcucha. Klasa *head* precyzuje własności ogniwa wyróżnionego, natomiast klasa *link* charakteryzuje ogniwa przedstawiające elementy zbioru.

2.1. Klasa *linkage*

Deklaracja klasy *linkage* jest następująca:

```
class linkage;
  begin
    ref (linkage) SUC, PRED;
    ref (link) procedure suc;
      suc :- if SUC in link then SUC else none ;
    ref (link) procedure pred;
      pred :- if PRED in link then PRED else none ;
  end linkage ;
```

UWAGA: Symbol :- oznacza w języku SIMULA przypisanie referencyjne; słowo kluczowe none oznacza obiekt pusty; deklaracja typu ref (A) X oznacza, że zmienna X jest zmienną referencyjną typu A.

Zmienne *SUC* i *PRED* mają wskazywać następny i poprzedni element listy. (Ze względu na bezpieczeństwo zmienne te nie są dostępne bezpośrednio dla programisty. Ich wartość może programista jedynie odczytać dzięki procedurom *suc* i *pred*.) Wartością procedur *suc* i *pred* jest obiekt obrazujący odpowiednio następny lub poprzedni element listy, ale tylko wtedy, gdy jest on obiektem podklasy klasy *link*.

2.2. Klasa *head*

Organizacja klasy *head* jest następująca:

```
linkage class head;
  begin
    ref (link) procedure first;  first :- suc ;
    ref (link) procedure last;   last :- pred ;
```

```
    boolean procedure empty; empty := SUC == this linkage ;
    integer procedure cardinal ;
      begin  integer i; ref (linkage) X;
        X :- this linkage ;
        for X :- X.suc while X =/= none  do  i := i+1 ;
        cardinal := i ;
      end cardinal;
    procedure clear;
      begin ref (link) X;
        for X :- first while  X=/= none  do  X.out ;
      end clear;
    SUC :- PRED :- this linkage ;
  end head;
```

Dla każdego zbioru, który ma być reprezentowany w programie, generowany jest jeden obiekt klasy *head*. Obiekt ten nie przedstawia elementu zbioru, tylko utożsamia sam zbiór. Podczas tworzenia obiektu klasy *head* wykonana jest instrukcja

```
SUC :- PRED :- this linkage ;
```

która powoduje, że atrybuty *SUC* i *PRED* wskazują na ten właśnie obiekt. Jeżeli lista przedstawiająca zbiór jest niepusta, to wartością procedury *first* jest pierwszy element zbioru, natomiast wartością procedury *last* jest ostatni element zbioru. Procedura logiczna *empty* ma wartość true tylko wtedy, gdy zbiór jest pusty, tzn. reprezentowany tylko przez obiekt klasy *head*. Procedura *cardinal* podaje liczbę elementów zbioru. Natomiast procedura *clear* usuwa wszystkie elementy zbioru. Procedura *clear* korzysta z procedury *out* zdefiniowanej w klasie *link* .

2.3. Klasa *link*

Organizacja klasy *link* jest następująca:

```
linkage class link ;
  begin
    procedure out;  if  SUC =/= none then
      begin
        SUC.PRED :- PRED;  PRED.SUC :- SUC;
        SUC :- PRED :- none ;
      end out;
    procedure follow(X); ref (linkage) X;
      begin
        out;
        if X =/= none then
```

```
            begin if X.SUC =/= none then
              begin PRED :- X; SUC :- X.SUC;
                    X.SUC :- SUC.PRED :- this linkage ; end
            end
        end follow;
      procedure precede(X); ref (linkage) X;
        begin
          out;
          if X =/= none then
            begin if X.SUC =/= none then
              begin SUC :- X; PRED :- X.PRED;
                    PRED.SUC :- X.PRED :- this linkage ; end
            end
        end precede;
      procedure into(S); ref (head) S;  precede(S) ;
    end link ;
```

Obiekty podklasy klasy *link* mogą reprezentować elementy pewnego zbioru. Każdy taki obiekt będzie zawierał atrybuty będące procedurami o nazwach: *suc*, *pred*, *out*, *follow*, *precede*, *into* (zmienne *SUC* i *PRED* są atrybutami niedostępnymi).

Procedura *out* powoduje usunięcie obiektu klasy *link* ze zbioru, o ile obiekt ten należy do jakiegoś zbioru. Procedura *follow(X)* usuwa element ze zbioru, a następnie umieszcza w zbiorze (być może innym) tuż za elementem X. Jeśli X == none lub X nie należy do żadnego zbioru (*X.SUC* == none), to element zostaje tylko usunięty ze zbioru, do którego należał. Podobne jest działanie procedury *precede(X)*, z tą różnicą, że dany element jest umieszczany w zbiorze tuż przed elementem X. Procedura *into(S)* wstawia element do zbioru wskazanego przez parametr aktualny. Jeśli parametr aktualny ma wartość none, to działanie tej procedury jest takie samo, jak procedury *out*.

3. Klasa SIMULATION

Klasa *SIMULATION* definiuje środki pozwalające na pisanie programów symulujących systemy rzeczywiste. Podstawowym pojęciem przy budowie modelu - jest pojęcie *procesu*. Każdy system rzeczywisty składa się z pewnej liczby procesów zachodzących jednocześnie, przy czym zachodzenie jednego procesu zazwyczaj nie pozostaje bez wpływu na zachowanie innych procesów. Procesy mogą być różnych typów oraz mogą trwać przez różne okresy czasu. Zachowanie procesu może również podlegać wpływom różnych czynników zewnętrznych w stosunku do rozważanego systemu.

Model symulacyjny powinien składać się z obiektów odwzorowujących

procesy rzeczywiste. Instrukcje takich obiektów powinny odzwierciedlać zachowanie rzeczywistych procesów podczas upływającego czasu. W tym celu w klasie *SIMULATION* zadeklarowano klasę o nazwie *process*, której obiekty odpowiadają procesom.

Program symulacyjny jest blokiem prefiksowanym klasą *SIMULATION*. W bloku tym zadeklarowane są różne podklasy klasy *process*. Obiekty każdej z nich mają inne atrybuty, a przez to i inne właściwości. W każdej chwili wykonywania programu można określić stany jego procesów na podstawie bieżących wartości ich atrybutów. Instrukcje procesu zmieniają wartości atrybutów, przez co zmieniają stan procesu. Można uważać, że każda zmiana stanu procesu odpowiada konkretnemu zdarzeniu w procesie rzeczywistym. Kolejność zachodzących zdarzeń w procesie rzeczywistym - określa jednoznacznie w jakim porządku mają następować zmiany stanu procesów w programie symulacyjnym.

Podczas wykonywania programu symulowany czas zmienia się w sposób skokowy. To znaczy, że przedmiotem zainteresowania konstruktora programu są tylko te chwile, w których następują zmiany stanu procesu. Zakłada się przy tym, że sama zmiana stanu procesu nie powoduje zwiększania czasu. Czas zmienia się tylko między kolejnymi zdarzeniami. (Ten rodzaj symulacji nazywany jest *symulacją dyskretną opartą na zdarzeniach* albo *symulacją dyskretnych zdarzeń*).

W klasie *SIMULATION* zdefiniowana została oś czasu (*SQS*), na której zaznaczany jest upływający czas (postęp czasu). Na osi *SQS* umieszczane są informacje o chwilach czasowych, w których nastąpią zdarzenia (zmiany stanu) w procesach. Informacje te nazywane są *zawiadomieniami* o zdarzeniach. Każde zawiadomienie jest obiektem klasy *EVENT NOTICE*, który ma atrybuty: *EVTIME* (typu real) oraz *PROC* (typu *process*). Atrybut *EVTIME* oznacza chwilę, w której ma nastąpić zdarzenie, a *PROC* jest nazwą procesu, w którym zajdzie to zdarzenie. Zawiadomienia, które znajdują się na osi czasu *SQS*, są uporządkowane niemalejąco według wartości ich pierwszych atrybutów, czyli według czasu.

Oś czasu *SQS* można przedstawić w sposób następujący:

(EVTIME1, PROC1) (EVTIME2, PROC2) (EVTIMEn, PROCn)

przy czym:

$$EVTIME1 \leq EVTIME2 \leq \ldots \leq EVTIMEn$$

Każda para *(EVTIMEi, PROCi)* przedstawia jedno zawiadomienie. W danej chwili wykonywania programu każdy proces może mieć tylko jedno zawiadomienie umieszczone na osi czasu. Wartość pierwszego atrybutu tego zawiadomienia jest chwilą, w której wystąpi najbliższe zdarzenie w procesie. W danej chwili wykonywania programu są realizowane instrukcje tylko jednego procesu. Pozostałe procesy czekają na swoją kolejność. Wykonywane są instrukcje zawsze tego procesu, którego zawiadomienie znajduje się na

początku osi czasu. Taki proces nazywany jest procesem *aktywnym*.

(Ten rodzaj pracy komputera nazywany jest często przetwarzaniem *quasi-równoległym*; współbieżne procesy rzeczywiste odwzorowane są w postaci quasi-równoległych procesów programowych).

Moduł główny programu symulacyjnego jest również traktowany jako proces - i w związku z tym może mieć swoje zawiadomienie na osi czasu.

3.1. Ogólny schemat klasy *SIMULATION*

Ogólny schemat klasy *SIMULATION* jest następujący (nazwy pisane dużymi literami - z wyjątkiem nazw *SIMSET* i *SIMULATION* - są niedostępne dla programującego) :

```
SIMSET class SIMULATION
   begin
      link class EVENT NOTICE (EVTIME, PROC);
           real EVTIME; ref (process) PROC;
            begin
             ref (EVENT NOTICE) procedure suc;
              suc :- if SUC is EVENT NOTICE then SUC else none;
             ref (EVENT NOTICE) procedure pred;
              pred :- PRED;
              ...
            end EVENT NOTICE;
       link class process;
            begin
              ref (EVENT NOTICE) EVENT; boolean TERMINATED;
              boolean procedure idle;
                idle := EVENT == none ;
              boolean procedure terminated;
                terminated := TERMINATED;
              real procedure evtime;
                if idle then ERROR else EVTIME := EVENT.EVTIME;
              ref (process) procedure nextev;
                nextev :- if idle then none else
                            if EVENT.suc == none then none
                             else  EVENT.suc.PROC ;
              detach;
              inner;
              TERMINATED := true ;
              passivate;
              ERROR;
            end process;
```

```
        ref (head) SQS;
        ref (EVENT NOTICE) procedure FIRSTEV;
           FIRSTEV :- SQS.first;
        ref (process) procedure current;
           current :- FIRSTEV.PROC;
        real procedure time; time := FIRSTEV.EVTIME;
        procedure hold(T); real T; ... ;
        procedure passivate ... ;
        procedure wait(S); ref (head) S ; ... ;
        procedure cancel(X) ; ref (process) X ; ... ;
        procedure ACTIVATE ... ;
        process class MAIN PROGRAM ; while true do detach ;
        ref (MAIN PROGRAM) main ;
        comment ..... część główna ..... ;
        SQS :- new head ;
        main :- new MAIN PROGRAM ;
        main.EVENT :- new EVENT NOTICE(0, main) ;
        main.EVENT.into(SQS) ;
    end SIMULATION ;
```

Pełną treść klasy *SIMULATION* podaje praca [4].

Klasa *SIMULATION* jest podklasą klasy *SIMSET*, dzięki czemu można w niej korzystać ze struktur zdefiniowanych w klasie *SIMSET*.

Klasa *EVENT NOTICE* jest podklasą klasy *link*, zatem obiekty tej klasy (zawiadomienia) mogą być elementami listy. Są one elementami listy o nazwie *SQS* przedstawiającej oś czasu.

Obiekty klasy *process* mają atrybut *EVENT*, który jest nazwą zawiadomienia odpowiadającego temu procesowi. Zmienna *TERMINATED* wskazuje, czy proces jest już zakończony. Wartość tej zmiennej można uzyskać dzięki procedurze o nazwie *terminated*.

Wartością procedury *idle* jest true, jeśli proces nie ma zawiadomienia na osi czasu. Mówi się, że taki proces jest bezczynny, ponieważ wykonanie jego instrukcji zostało zawieszone.

Dzięki procedurze *evtime* można odczytać chwilę czasową, w której jest zaplanowana realizacja najbliższego zdarzenia w procesie.

Wartością procedury *nextev* jest proces, którego zawiadomienie zajmuje następną pozycję na osi czasu.

Pierwszą instrukcją w każdym obiekcie klasy *process* jest instrukcja *detach*. Jej wykonanie powoduje zawieszenie dalszych instrukcji procesu. Dopiero wykonanie odpowiedniej instrukcji aktywacji może spowodować podjęcie dalszych akcji procesu. (Należy zwrócić uwagę, że klasa *process* jest podklasą klasy *link*. Stąd wynika, że procesy mogą być elementami różnych list tworzonych przez programującego.

Wartością procedury *current* jest proces, którego zawiadomienie jest na

początku osi czasu (proces aktywny). Wartością procedury *time* jest chwila zdarzenia zachodzącego w tym procesie. Procedura *time* podaje aktualną wartość czasu symulacyjnego.

3.2. Procedury planujące kolejność zdarzeń w procesach

W skład grupy procedur planujących kolejność zdarzeń zachodzących w procesach wchodzą procedury o nazwach: *hold, passivate, cancel i wait.*

W wyniku wywołania procedury *hold(T)* przerwane zostaje wykonywanie instrukcji procesu aktywnego (który ją wywołał). Ponadto, jeśli zachodzi $T > 0$, to zmieniona zostaje wartość atrybutu *EVTIME* jego zawiadomienia na wartość równą *EVTIME* + *T*. Zawiadomienie to zostaje wstawione w odpowiednie miejsce na osi czasu, tak, aby niemalejące uporządkowanie zawiadomień za względu na pierwszy atrybut nie zostało zaburzone. Jest to miejsce bezpośrednio za zawiadomieniami o chwilach czasowych nie przekraczających wartości *EVTIME* + *T*. Jeśli $T < 0$ to wywołanie *hold(T)* jest równoważne wywołaniu *hold(0)*.

Procedura *passivate* powoduje likwidaję zawiadomienia dla procesu aktywnego i zawieszenie wykonywania jego instrukcji. Proces taki może zostać ponownie uaktywniony za pomocą jednej z procedur aktywacji. (Analogiczną sytuację do wykonania procedury *passivate* można uzyskać poprzez wykonanie dla danego procesu *hold(∞)* gdzie ∞ może być rozumiane jako duża liczba - większa od zaplanowanego czasu symulacyjnego działania wszystkich obecnych w programie procesów).

Wykonanie procedury *cancel(X)* powoduje likwidację zawiadomienia dla procesu X, o ile proces X je ma. (Wywołanie *cancel(current)* jest równoznaczne wywołaniu *passivate*).

Wywołanie procedury *wait(S)* w procesie powoduje zawieszenie wykonywania jego instrukcji z jednoczesną likwidacją zawiadomienia. Sam proces natomiast zostaje wstawiony na koniec listy o nazwie *S*. Lista *S* może być dowolną listą zadeklarowaną przez programistę. (Wywołanie procedury *wait(S)* jest równoważne wykonaniu dwu instrukcji: *current.into(S); passivate;*).

3.3. Instrukcje aktywacji i reaktywacji

W podanym wp.3.1. schemacie klasy *SIMULATION* umieszczono deklarację procedury *ACTIVATE*. Procedura ta jest niedostępna dla programisty, można ją natomiast uważać za opis działania instrukcji aktywacji i reaktywacji. Instrukcje te mogą mieć następujące formy:

AKTYWACJA	REAKTYWACJA
activate X	reactivate X
activate X at T	reactivate X at T
activate X at T prior	reactivate X at T prior
activate X delay T	reactivate X delay T
activate X delay T prior	reactivate X delay T prior
activate X before Y	reactivate X before Y
activate X after Y	reactivate X after Y

przy czym X, Y oznaczają wyrażenia referencyjne o wartościach będących obiektami klasy *process*, T zaś oznacza wyrażenie arytmetyczne. Każde użycie jednej z powyższych instrukcji powoduje wywołanie procedury *ACTIVATE* z odpowiednimi parametrami. Instrukcje te służą do tworzenia nowych zawiadomień dla procesów oraz umieszczania ich na osi czasu w odpowiednich miejscach.

Instrukcje aktywacji dotyczą procesów, które nie mają zawiadomień na osi czasu oraz nie zostały jeszcze zakończone. Są to procesy, w których nastąpiło zawieszenie realizacji instrukcji bez zaplanowania następnej chwili określającej ponowne ich wznowienie.

Instrukcje reaktywacji dotyczą procesów, które mają zawiadomienia na osi czasu. Ich działanie polega na likwidacji istniejącego zawiadomienia dla procesu, a następnie na wykonaniu odpowiedniej instrukcji aktywacji.

Instrukcja activate X powoduje utworzenie zawiadomienia dla procesu X. Zawiadomienie to ma postać (t,X), w której chwila czasowa t jest równa wartości bieżącej czasu symulacyjnego. Czas ten jest dostępny za pomocą procedury *time*. Zawiadomienie to jest umieszczone na pierwszej pozycji osi czasu, więc po wykonaniu instrukcji activate X proces X staje się aktywny. (Aktywacja, w której wyniku instrukcje procesu zostają natychmiast podejmowane, nazywa się aktywacją bezpośrednią.

Instrukcja activate X at T powoduje utworzenie zawiadomienia postaci (t,X) oraz wstawienie go na osi czasu bezpośrednio za zawiadomieniami o chwilach nie przekraczających wartości t. Wartość t. Wartość t jest określana jako większa spośród T oraz *time* .
(Instrukcje activate X oraz activate X at *time* nie zawsze są równoważne. Pierwsza z nich powoduje, że proces X natychmiast staje się aktywny, natomiast w wyniku wykonania drugiej, zawiadomienie dla procesu X zajmuje ostatnią pozycję wśród zawiadomień o chwili równej *time*).

Instrukcja activate X at T prior powoduje utworzenie zawiadomienia postaci (t,X), przy czym t = max(T, *time*), oraz umieszczenie go na osi czasu przed wszystkimi zawiadomieniami o chwilach nie mniejszych niż t. (Należy zauważyć, że instrukcja activate X at *time* prior jest aktywacją bezpośrednią).

Instrukcje activate X delay T (activate X delay T prior) powodują utworzenie zawiadomienia postaci (t,X), przy czym t = max(time + T, time), oraz wstawienie go w odpowiednie miejsce osi czasu (analogicznie jak w opisie poprzednich dwu instrukcji). Wykonanie instrukcji activate X delay T (activate X delay T prior) jest równoważne wykonaniu activate X at time + T (activate X at time + T prior).

Instrukcja activate X before Y powoduje utworzenie zawiadomienia postaci (t,X), przy czym t otrzymuje wartość taką samą jak chwila zawiadomienia dla procesu Y. Zawiadomienie (t,X) jest umieszczone na osi czasu bezpośrednio przed zawiadomieniem dla procesu Y. (Należy zauważyć, że activate X before current jest aktywacją bezpośrednią).

Instrukcja activate X after Y powoduje, że zawiadomienie tworzone dla procesu X otrzymuje tę samą chwilę czasową co zawiadomienie dla Y oraz jest wstawiane bezpośrednio za nim na osi czasu. W przypadku, gdy Y nie ma zawiadomienia na osi czasu, albo Y == none, powyższe dwie instrukcje nie powodują żadnych zmian.

Wykonanie instrukcji aktywacji dla procesu, który ma zawiadomienie na osi czasu nie powoduje żadnej zmiany. Natomiast instrukcja reaktywacji dla procesu X, który nie ma zawiadomienia, jest równoważna odpowiedniej instrukcji aktywacji. Jeśli X == none , to każda z omawianych instrukcji jest instrukcją pustą.

3.4. Program symulacyjny jako proces

Wykonanie programu symulacyjnego polega na realizowaniu akcji poszczególnych procesów. W każdej chwili wykonywane są instrukcje tego procesu, którego zawiadomienie zajmuje pierwszą pozycję na osi czasu. Blok prefiksowany klasą *SIMULATION* (lub jej podklasą) jest również traktowany jako proces, który może mieć swoje zawiadomienie. Podczas wykonywania instrukcji tego bloku, odpowiadające mu zawiadomienie powinno występować na pierwszym miejscu osi czasu. Możliwość traktowania bloku symulacji jako procesu osiągnięto dzięki deklaracji klasy *MAIN PROGRAM*. Jest ona podklasą klasy *pro- cess* i zawiera jedyną instrukcję postaci: while true do *detach* ;

Realizacja każdego programu symulacyjnego zaczyna się od wykonania instrukcji umieszczonych w klasie *SIMULATION*. Po wykonaniu pierwszej z nich *SQS* :- new *head* ; utworzona zostaje oś czasu, która jeszcze nie zawiera żadnego elementu. Następną instrukcją jest: *main* :- new *MAIN PROGRAM* ; w efekcie zostaje utworzony obiekt klasy *MAIN PROGRAM*, a dzięki instrukcji *detach* następuje powrót do bloku oraz wykonanie dwóch instrukcji:

main.EVENT :- new *EVENT NOTICE* (0, *main*) ;

main.EVENT.into(*SQS*) ;

Na listę *SQS* zostaje wstawione zawiadomienie o chwili równej 0. Odpowiada ono procesowi o nazwie *main*. Bezpośrednio przed realizacją instrukcji napisanych przez programistę, oś czasu przedstawia się następująco: aktualny czas systemowy wynosi 0, a procesem aktywnym jest proces *main*.

4. Procedury pomocnicze

W języku SIMULA-67 wyróżnić można dwie grupy procedur pomocniczych:

- procedury pozwalające na dokonywanie operacji wejścia/wyjścia (zdefiniowane w klasie systemowej *BASICIO*);
- procedury pseudolosowe, generujące wartości liczbowe według różnych rozkładów (niezbędne przy realizacji przebiegów symulacyjnych).

Procedury pomocnicze języka SIMULA zostały szczegółowo omówione w pracy [4].

5. Uwagi końcowe

Zagadnienia przedstawione w niniejszym artykule stanowią jedynie pewien szkic (bazujący głównie na literaturze [4]) specyfikujący podstawowe zasoby warstwy symulacyjnej języka SIMULA. Dokładne omówienie walorów tego języka dla potrzeb symulacji systemów transportowych zawiera praca [6]. Ideę języków obiektowych zaprezentowano m.in. w artykule [3]. Ocenę porównawczą aspektów użytkowych języków symulacyjnych zawiera pozycja [7]. Pozostałe pozycje wymienionej poniżej literatury - dotyczą aspektów ogólnych realizacji oprogramowania symulacyjnego.

Należy ogólnie stwierdzić, że rozwiązania i koncepcje zrealizowane w języku SIMULA-67 - wniosły duży wkład do rozwoju teorii i praktyki programowania (nie tylko symulacyjnego). Przykładem rozwinięcia idei SIMULI-67 może być język LOGLAN.

LITERATURA

[1] DAHL O.J., MYHRHAUG B., NYGAARD K.: Simula-67 Common Base Language. Norwegian Computing Center, 1970 .

[2] DANDA J.: SIMULA and Structured Modelling - Simula News-letters, Vol.6, No.4, November 1978.

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

[4] OKTABA H., RATAJCZAK W.: Simula 67. WNT, Warszawa 1980

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

[6] ROMANIUK J.: Zastosowanie języka SIMULA 67 do modelowania systemów transportowych. Zagadnienia Transportu, Wyd. PAN, Warszawa nr 3/4 1980/81

[7] WILCZEK T.: Aspekty użytkowe języków symulacyjnych - Informatyka nr 9 i 10 / 1983 .

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

SIMULATION RESOURCES OF SIMULA-67 LANGUAGE

Summary

Basic resources of SIMULA-67 language which is general-purpose high-level language with a built-in simulation language have been presented in the paper.

SIMULA allows easy adapting to various specialized problems and can be used as a base of problem-oriented languages.

SIMSET and SIMULATION system classes and set of auxiliary procedures contribute to the SIMULA-67 simulation resources.

The SIMULATION class specifies the means allowing to write the programs that simulate real systems.

The problems presented in the present paper make only a certain sketch specifying basic resources of the SIMULA language simulation layer.

SIMULATIONSVORRÄTE DER SPRACHE SIMULA-67

Zusammenfassung

Im Aufsatz wurden grundsätzliche Vorräte der Sprache SIMULA-67 vorgestellt. SIMULA-67 ist eine universelle Sprache der höhen Niveau mit eingebauter Sprache zur Simulation.

SIMULA erlaubt einfache Anpassung an verschiedene spezifierte Probleme und kann als Basis problemorientierten Sprachen eingesetzt werden.

Die Simulationsvorräte der Sprache SIMULA-67 bestehen aus: Systemklassen SIMSET und SIMULATION sowie einer Liste von Hilfsprozeduren.

Die Klasse SIMULATION definiert Mittel zum Schreiben von Programmen, die reale Systeme simulieren.

Die im vorliegenden Aufsatz vorgestellten Probleme bilden nur eine Skizze, die grundsätzliche Vorräte der Simulationsschicht der Sprache SIMULA präsentiert.

СПОСОБЫ СИМУЛИРОВАНИЯ ЯЗЫКА SIMULA-67

Резюме

В статье представлены основные ресурсы языка SIMULA-67, который является универсальным языком высшего уровня с встроенным языком для симуляции

SIMULA дает возможность легкого приспособления для различных специализированных проблем и может быть употреблен как база проблемных языков.

Ресурс симуляционного языка SIMULA-67 состоит из: системных классов SIMSET и SIMULATION а также из состава вспомагающих процедур.

Класс SIMULATION определяет средства, позволяющие на получение симуляционных программ для реальных систем.

Представленные вопросы в статье являются лишь эскизом основных ресурсов симуляционного слоя языка SIMULA.