SZKOŁA JESIENNA PTI

WSPÓŁCZESNE KIERUNKI ROZWOJU INFORMATYKI

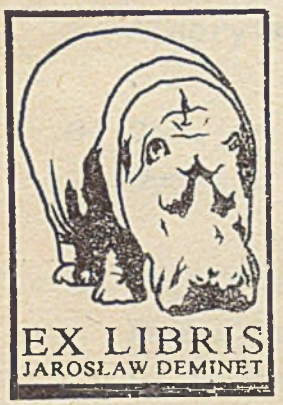
Organizowana przez POLSKIE TOWARZYSTWO INFORMATYCZNE
przy współudziale ZAKŁADÓW ELEKTRONICZNYCH „ELWRO"

RYDZYNA 9—13.X.1984 r.

SZKOŁA JESIENNA PTI

WSPÓŁCZESNE KIERUNKI ROZWOJU INFORMATYKI

Organizowana przez POLSKIE TOWARZYSTWO INFORMATYCZNE
przy współudziale ZAKŁADÓW ELEKTRONICZNYCH „ELWRO"



RYDZYNA 9—13.X.1984 r.

PROGRAM SZKOŁY

1. Prof. dr hab. Jacek Bańkowski
   Modele baz danych jako odwzorowania świata rzeczywistego

2. Prof. dr hab. Andrzej Blikle
   Denotacyjna metoda specyfikacji oprogramowania - VDM

3. Doc. dr hab. Wojciech Cellary
   Systemy wielomikroprocesorowe

4. Dr Jacek Irlik
   Prawne aspekty informatyki

5. Prof. dr hab. Antoni Mazurkiewicz
   Zagadnienia współbieżności w programowaniu

6. Prof. dr hab. Andrzej Salwicki
   Typy danych: od algorytmicznej specyfikacji do implementacji w języku programowania

7. Doc. dr hab. Maciej M. Sysło
   Algorytmy kombinatoryczne i ich efektywność

8. Doc. dr hab. inż. Stanisław Waligórski
   Wybrane języki i metody programowania w zastosowaniach

9. Prof. dr hab. Jan Węglarz
   Ocena działalności systemów komputerowych - przedmiot, metody, kierunki rozwoju

10. Doc. dr hab. Jan Zabrodzki
    Stan obecny i tendencje rozwoju układów LSI i VLSI

SŁOWO WSTĘPNE

Postęp w informatyce uwarunkowany jest nie tylko rozwojem technologii sprzętu i oprogramowania, ale również - i w coraz większym stopniu - postępem w rozwiązywaniu szeregu problemów podstawowych. Należą do nich między innymi problemy związane z podnoszeniem jakości i niezawodności oprogramowania, z organizowaniem wielkich zbiorów informacji, współdziałaniem procesów i systemów. Zdaniem wielu specjalistów, wdrożenie rozwiązań tych problemów do praktyki przemysłowej jest warunkiem koniecznym dalszego postępu w informatyce. Wyrazem tych opinii jest widoczny od kilku lat wzrost zainteresowania problemami podstawowymi w środowisku informatyki przemysłowej i użytkowej w świecie. Wiele firm organizuje kursy dla swoich pracowników poświęcone tematom uznanym niegdyś za akademickie.

Organizacja spotkań naukowych, poświęconych informatyce, jest również jednym z ważnych zadań statutowych Polskiego Towarzystwa Informatycznego. Kierując się tą myślą Zarząd Główny PTI poprzedniej kadencji, już na jednym z pierwszych posiedzeń, jakie odbyły się po Zjeździe Założycielskim w maju 1981 r., podjął inicjatywę zmierzającą do zorganizowania Zimowej Szkoły PTI poświęconej współczesnym kierunkom rozwoju informatyki. Postanowiono, że szkoła odbędzie się w lutym 1982 r. Jesienią 1981 r. gotowy był już program szkoły oraz ustalone zostały wszystkie ważniejsze szczegóły organizacyjne. Niestety, z powodów niezależnych od Towarzystwa szkoły tej nie udało się zorganizować.

Niniejsza Jesienna Szkoła PTI stanowi kontynuację opisanych powyżej zamierzeń. Poświęcona jest tej samej tematyce i realizować ma ten sam, co poprzednio, cel: budowanie kontaktów naukowych pomiędzy różnymi środowiskami informatycznymi w Polsce.

Pierwsza szkoła naukowa PTI jest imprezą o charakterze przeglądowym. Stanowi ona ofertę środowiska akademickiego adresowaną do wszystkich środowisk informatyki zawodowej w Polsce. Jej organizatorzy mają nadzieję, że stanie się ona początkiem

serii szkół, konferencji i sympozjów poświęconych różnym aspektom informatyki.

Na program Szkoły składa się dziesięć wykładów poświęconych wybranym kierunkom rozwoju informatyki na świecie. Przy ich wyborze kierowano się aktualnością i wagą zagadnienia, a także zasadą, aby każdy wykładowca był uznanym specjalistą w reprezentowanej przez siebie dziedzinie, prowadzącym własne badania w danym zakresie i reprezentującym odpowiednio silne krajowe środowisko naukowe.

Teksty wszystkich dziesięciu wykładów, składających się na program Szkoły, opracowane zostały specjalnie na zaproszenie Organizatorów. Poproszono wykładowców, aby były to teksty o charakterze przeglądowym, eksponujące zastosowania i operujące licznymi przykładami. Uczestnicy Szkoły oraz inni czytelnicy niniejszego tomu ocenią czy cel ten został osiągnięty.

Andrzej Blikle

Warszawa, lipiec 1984

Jesienna Szkoła PTI
Rydzyna, październik 1984

MODELE BAZ DANYCH JAKO ODWZOROWANIA ŚWIATA RZECZYWISTEGO

prof. Jacek Bańkowski
Instytut Informacji Naukowej,
Technicznej i Ekonomicznej
ul. Żurawia 3/5
00-926 Warszawa

1. Wstęp

Tendencja do ułatwienia pracy programistom produkującym programy użytkowe datuje się właściwie od początku istnienia informatyki. Już pierwsze języki wysokiego poziomu powstałe w latach pięćdziesiątych pozwalały na stosowanie takich struktur danych jak tablice i pliki złożone z rekordów. Możliwe stało się również wykonywanie operacji na takich strukturach bez odwoływania się do szczegółów związanych z rozmieszczeniem ich elementów w pamięci.

W wielu zastosowaniach koszt zbierania danych - zwłaszcza masowych - jest bardzo duży i naturalna staje się tendencja do wykorzystywania raz zebranych danych w wielu programach. Ponieważ przyszłe zastosowania zebranych danych nie zawsze łatwo przewidzieć, struktury danych użyte przy pierwszych zastosowaniach / zwłaszcza jeśli struktury te były bardzo efektywne i pomysłowe/ mogą być trudne do wykorzystania przy zastosowaniach następnych,

a w każdym razie konieczna jest bardzo szczegółowa analiza dokumentacji.

Na przełomie lat sześćdziesiątych i siedemdziesiątych powstała idea współpracy programów z danymi polegająca na tym, aby dane przechowywać w strukturach które, ogólnie mówiąc, są w pewien zunifikowany sposób /dla każdego proponowanego rozwiązania/ związane ze znaczeniowymi aspektami przechowywanych danych. Dane wypełniające te struktury tworzą bazę danych. System zarządzania bazą danych zapewnia dostęp do poszczególnych danych poprzez używanie - w większym lub mniejszym stopniu - sformułowań bliższych związkom znaczeniowym między danymi. Dzięki temu różne programy mogą korzystać ze wspólnych zasobów danych, nie będąc skrępowane ograniczeniami wynikającymi z różnych sposobów patrzenia na dane.

Możliwości unifikacji sposobów reprezentowania przez dane związków między obiektami świata rzeczywistego jest bardzo wiele. Obecnie najbardziej znanymi takimi sposobami są rozwiązania wynikające z tzw. modelu sieciowego [5], [13], [14], hierarchicznego [8], [9], [14] i relacyjnego [6], [8], [14].

2. Modele danych

Aby móc porównywać różne modele danych trzeba dysponować aparatem, który umożliwia ścisły opis pewnego wycinka świata rzeczywistego. W przeciwnym bowiem razie trudno byłoby odpowiedzieć na pytanie, czy rozważane modele opisują ten sam wycinek świata. Aparat taki istnieje i obrazy wycinka świata rzeczywistego wyrażone przy jego pomocy są nazywane modelem konceptualnym tego wycinka.

Elementami składowymi modelu konceptualnego [4], [14] są obiekty /ang. entities/ i ich zbiory. Pomiędzy zbiorami obiektów mogą zachodzić związki /ang. relationships/, zaś obiekty mogą posiadać cechy /ang. attributes/.

Pojęcie obiektu jest trudne do zdefiniowania. Przyjmuje się, że obiekt jest to coś /o naturze fizycznej lub abstrakcyjnej/

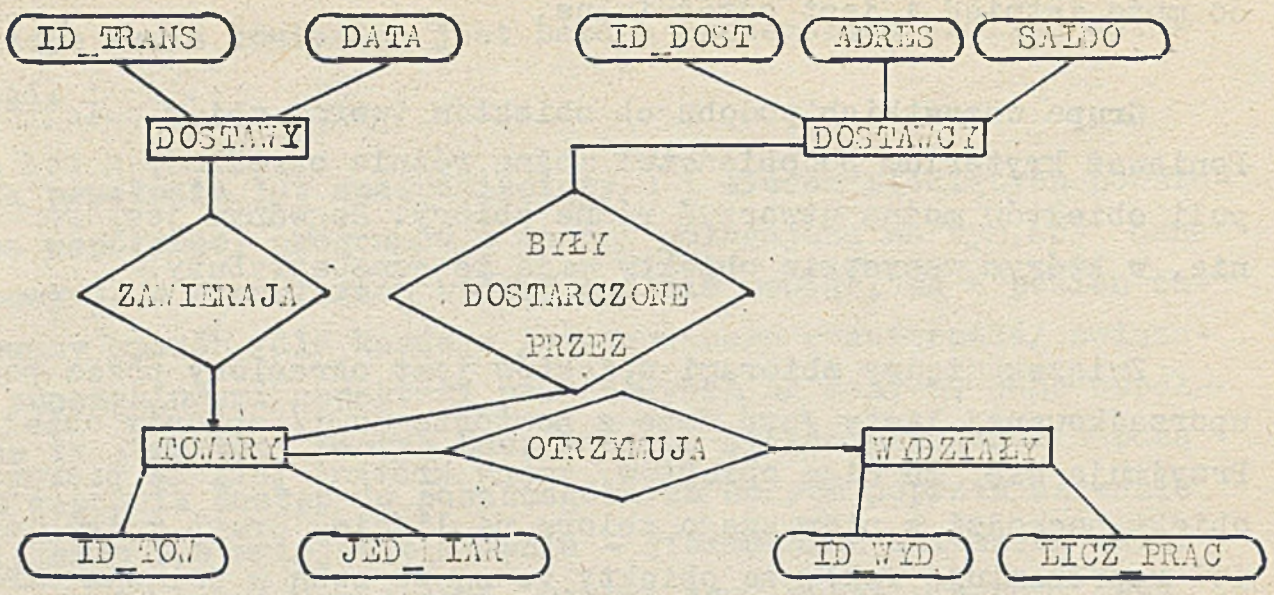
co może istnieć i jest odróżnialne.

Grupa wszystkich podobnych obiektów tworzy zbiór obiektów. Ponieważ kryterium podobieństwa można różnie określać, z tej samej puli obiektów można utworzyć różne zbiory. Dogodnym jest rozwiązanie, w którym wszystkie obiekty mają te same atrybuty.

Związek między zbiorami obiektów jest określony przez podanie uporządkowanej listy /być może z powtórzeniami/ zbiorów obiektów. Przyjmuje się, że ciąg obiektów, zwany krotką, taki że pierwszy obiekt pochodzi z pierwszego zbioru na liście, drugi z drugiego itd reprezentuje fakt, że obiekty te są ze sobą w związku określonym przez tę listę. Największe znaczenie praktyczne mają związki między parami zbiorów /binarne/, aczkolwiek związki między liczniejszymi grupami zbiorów również występują w praktyce. Wśród związków binarnych można wyróżnić pewne szczególne ich rodzaje, których uwzględnienie umożliwia budowanie modeli danych łatwiejszych do implementacji niż w przypadku gdyby pominąć to rozróżnienie. I tak, związek binarny jest typu jeden-do-jednego jeżeli **zawsze** obiektowi w pierwszym zbiorze odpowiada co najwyżej jeden obiekt w drugim zbiorze i odwrotnie, związek jest typu wiele-do-jednego /m:1/ jeżeli **zawsze** obiektowi w pierwszym zbiorze odpowiada co najwyżej jeden obiekt w drugim zbiorze, ale jednemu obiektowi w drugim zbiorze może odpowiadać więcej niż jeden obiekt w zbiorze pierwszym, w pozostałych przypadkach związek jest typu wiele-do-wielu /m:m/.

Atrybuty mogą być przydzielane obiektom w zasadzie dowolnie /oczywiście zgodnie z rzeczywistością/, przy czym z warunku rozróżnialności obiektów w zbiorze wynika, że należy przydzielić co najmniej takie atrybuty, których wartości umożliwiają rozróżnienie każdego obiektu w zbiorze. Ogólnie, wśród atrybutów obiektu może istnieć wiele zestawów atrybutów, które pozwalają na takie odróżnianie i każdy z nich nosi nazwę klucza. Klucz, który jest rzeczywiście używany do tego celu będziemy w dalszym ciągu tej pracy nazywali identyfikatorem zbioru lub krótko identyfikatorem.

Rozważmy następujący przykład: /rysunek na następnej stronie/



W prostokątach podano nazwy zbiorów obiektów, w owalach - nazwy atrybutów zaś w rombach - nazwy związków; brak strzałki na linii między związkiem a zbiorem oznacza, że wiele obiektów tego zbioru może być w związku z jednym obiektem drugiego zbioru zaś strzałka w kierunku jakiegoś zbioru oznacza, że może być co najwyżej jeden taki obiekt. W ten sposób reprezentowane są graficznie wszystkie omówione poprzednio typy związków binarnych.

Przedstawiony schemat należy interpretować następująco: obiekty zbioru DOSTAWY mają atrybuty ID_TRANS /pełni rolę identyfikatora zbioru/ oraz DATA /data dostawy/; zbiór ten jest w związku ZAWIERAJA typu m:1 ze zbiorem TOWARY /w dostawie występuje zawsze jeden towar, ale ten sam towar może występować w wielu dostawach/; obiekty tego ostatniego zbioru mają atrybuty ID_TOW /identyfikator towaru/ i JED_MIAR /jednostka miary towaru/; zbiór ten jest w związku BYŁY_DOSTARCZONE_PRZEZ typu m:m ze zbiorem DOSTAWCY /mogą istnieć towary, które mają wielu dostawców i dostawcy, którzy dostarczają wielu towarów/ oraz w związku OTRZYMUJA typu m:1 ze zbiorem WYDZIAŁY /dany towar trafia na co najwyżej jeden wydział, ale jeden wydział może otrzymywać wiele towarów/; atrybutami obiektów w zbiorze DOSTAWCY są ID_DOST /identyfikator dostawcy/, ADRES /adres dostawcy/ i SALDO /stan rozliczeń dostawcy/, zaś atrybutami obiektów w zbiorze WYDZIAŁY są ID_WYD /identyfikator wydziału/ i LICZ_PRAC /liczba pracowników/.

Na rysunku zaznaczono tylko niektóre związki binarne. Pełny wykaz sensownych typów związków zawiera tabela, przy czym odsy-

łacze typu A, A1 itd wskazują na bliższe objaśnienia.

	DOSTAWY	DOSTAWCY	TOWARY	WYDZIAŁY
DOSTAWY		A/m:1 A1/m:m	B/m:1 B1/m:m	C/m:1 C1/m:m
DOSTAWCY	- A1	/	- D1	E2/m:1 E1/m:m
TOWARY	- B1	D/m:1 D1/m:m		F/m:1 F1/m:m
WYDZIAŁY	- C1	E/m:1 E1	- F1	

A/ dostawa pochodzi od jednego dostawcy, ale jeden dostawca może brać udział w wielu dostawach,

A1/ dostawa może pochodzić od wielu dostawców i na odwrót

B/ dostawa dotyczy jednego towaru, ale może on występować w wielu dostawach

B1/ w jednej dostawie może być wiele towarów i na odwrót

C/ dostawa trafia na jeden wydział, ale wiele takich dostaw może trafiać na jeden wydział

C1/ dostawa może trafiać na wiele wydziałów i na odwrót

D/ każdy towar jest dostarczany przez co najwyżej jednego dostawcę, ale każdy dostawca może dostarczać wielu towarów

D1/ towar może być dostarczany przez wielu dostawców i na odwrót

E/ każdy wydział ma tylko jednego dostawcę, ale jeden dostawca może obsługiwać wiele wydziałów

E1/ wydział ma wielu dostawców i na odwrót

E2/ każdy dostawca może obsługiwać co najwyżej jeden wydział, ale wydział może mieć wielu dostawców

Gdyby przyjąć, że sensowny typ każdego z sześciu podanych związków może być wybrany dowolnie, to mielibyśmy $2^5 \times 3 = 96$ modeli konceptualnych. W rzeczywistości, jak się o tym przekonamy, w niektórych przypadkach typ związku jest wyznaczony przez typy innych związków i powyższy rachunek należy uznać za zawyżony. Z drugiej strony, może istnieć wiele związków binarnych między tymi samymi zbiorami, np. między zbiorami TOWARY i DOSTAWCY może istnieć związek takiego typu, jak zaznaczono na rysunku, interpretowany jako informacja o towarach dostarczonych dotychczas

przez danego dostawcę, oraz związek typu m:n interpretowany jako informacja o tym, jakie towary /niekoniecznie już dostarczone/ są objęte gwarancją przez danego dostawcę. Zbiory par <towar, dostawca> reprezentujące aktualny stan obu związków mogą być zupełnie różne*/. Ponadto, w rachunku nie uwzględniono związków innych niż binarne, a łatwo wykazać, że liczba wszystkich związków rośnie wykładniczo ze wzrostem liczby zbiorów. Tak więc, przeanalizowanie wszystkich możliwych związków już przy zupełnie niebastronomicznej liczbie zbiorów staje się bardzo pracochłonne tym bardziej, że ocena sensowności typu i przydatności związku musi być dokonana w zasadzie przez człowieka.

Utrudnia to praktyczną realizację takiego podejścia do tworzenia modelu konceptualnego, w którym pierwotnymi obiektami byłyby wartości wszystkich atrybutów występujących w danym wycinku świata rzeczywistego. Podejście takie jest teoretycznie możliwe, ponieważ obiekty takie są odróżnialne w ramach danego atrybutu /który pełni wtedy rolę zbioru obiektów/ i między parami, trójkami itd. takich atrybutów można określać związki tak, jak między wcześniej stosowanymi obiektami "intuicyjnymi". Tak określone związki można uważać za swego rodzaju "zbiory obiektów wyższego rzędu" /których identyfikatorem jest, w najgorszym przypadku, zbiór wszystkich atrybutów wchodzących do danego związku/. Zaletą tego podejścia jest możliwość wczesnego wykrycia niebezpieczeństw związanych z niewłaściwym doborem atrybutów dla pomyślanych wcześniej "intuicyjnych" obiektów.

Np. utworzenie zbioru obiektów o nazwie ZAOPATRZENIE_WYDZIAŁÓW z ternarnego związku atrybutów ID_TRANS, ID_DOST, ID_WYD przy istnieniu związków binarnych typu m:1 między ID_TRANS i ID_DOST oraz ID_WYD i ID_DOST prowadzi do takiej właśnie niewygodnej sytuacji, ponieważ z podanych związków wynika, że przy takiej reprezentacji liczba krotek niezbędnych do reprezentowania

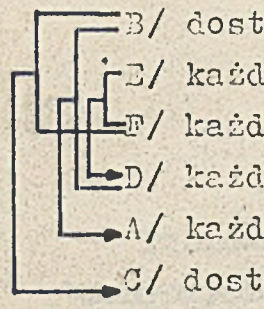
---

*/ w każdym z omawianych dalej modeli danych istnieje możliwość łącznego reprezentowania wielu takich związków za pomocą jednego; por. analiza atrybutu CENA_JEDN. w rozdziałach 6. Atrybuty i 7. Aktualizacja

obiektów o pewnej ustalonej wartości ID_DOST jest równa iloczynowi liczby wartości ID_TRANS z nim związanych przez liczbę wartości ID_WYD z nim związanych

Na zakończenie tego rozdziału określimy dwa warianty modelu konceptualnego wycinka przedsiębiorstwa o zbiorach obiektów i atrybutach takich jak dotychczas opisane, natomiast związki są następujące:

Pierwszy wariant:



B/ dostawa dotyczy jednego towaru
E/ każdy wydział ma jednego dostawcę
F/ każdy towar trafia na jeden tylko wydział
D/ każdy towar ma jednego dostawcę
A/ każda dostawa pochodzi od jednego dostawcy
C/ dostawa trafia w całości na jeden wydział

/linie wskazują, które związki wynikają z innych/

Drugi wariant:

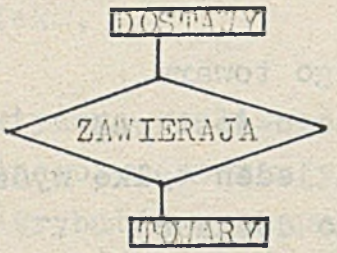
A/ każda dostawa pochodzi od jednego dostawcy
B1/ w jednej dostawie może być wiele towarów
C1/ dostawa może trafiać na wiele wydziałów
D1/ towar może mieć wielu dostawców
E1/ wydział może mieć wielu dostawców
F1/ towar może być przeznaczony dla wielu wydziałów

3. Model sieciowy

Model sieciowy pozwala bezpośrednio odwzorowywać jedynie związki binarne typu n:1, w związku z czym bezpośrednio można odwzorowywać tylko takie modele konceptualne, które są de facto grafami skierowanymi ze zbiorami obiektów w roli węzłów i związkami w roli strzałek /zbędne więc staje się wprowadzanie odrębnych symboli graficznych dla związków w postaci rombów/.

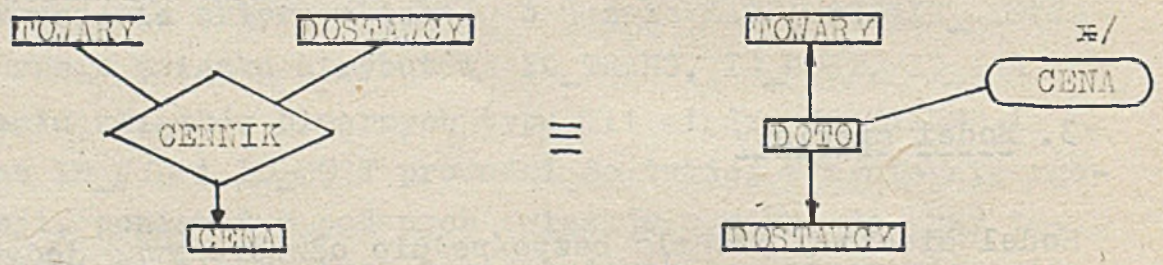
Ograniczenie to można dość łatwo obejść, wprowadzając pewne

sztuczne zbiory obiektów, które poza identyfikatorem /który w dodatku nie musi być fizycznie przechowywan / nie muszą /choć mogą/ mieć żadnych innych atrybutów. Np. gdyby między zbiorami DOSTAWY i TOWARY istniał związek typu m:m, to można by wprowadzić zbiór obiektów TODO, który byłby w związku typu m:1 z każdym z wymienionych wcześniej zbiorów; wtedy, aby wyznaczyć towary występujące w dostawie należy wyznaczyć obiekty z nią związane w zbiorze TODO, a następnie dla każdego z nich wyznaczyć towar. Łatwo sprawdzić, że wycinek modelu konceptualnego



jest, przy opisanej wyżej interpretacji całkowicie równoważny modelowi

Związki ternarne /i bardziej złożone/ można w podobny sposób zamienić na binarne związki typu m:1, np. związek ternarny



*/

*/ wykorzystano fakt, że w modelu z lewej strony obiekt w zbiorze CENA jest jednoznacznie wyznaczony przez parę obiektów towar, dostawca ; wobec tego, wartość ceny może być atrybutem w "sztucznym" zbiorze DOTO, zaś sam zbiór CENA można zlikwidować.

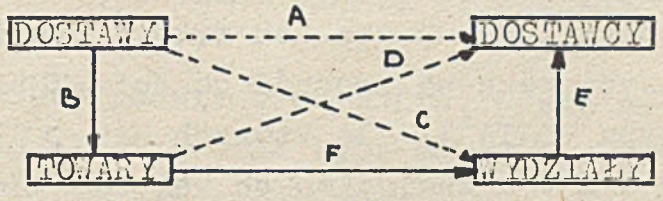
W najbardziej znanej implementacji sieciowego modelu danych, tzw. propozycji CODASYLu [5] zbiory obiektów reprezentowane są przez odpowiadające im pliki, obiekty - przez rekordy, zaś pola w tych rekordach reprezentują wartości atrybutów obiektów. Związki typu m:1 są reprezentowane przez specjalnie wprowadzoną konstrukcję tzw. DBTG SET*/. Każde wystąpienie DBTG SET składa się z części OWNER, zawierającej jeden rekord pliku np. B i części MEMBER, która jest pusta lub zawiera pewną liczbę rekordów pliku np. A. Wtedy mówimy, że związek typu m:1 między zbiorami A i B jest reprezentowany**/ przez DBTG SET XX zdefiniowany /szkicowo/ następująco

```
DBTG SET XX
  OWNER IS B
  MEMBER IS A;
```

Ponieważ język manipulacji danymi umożliwia m in., z jednej strony wyszukanie rekordu o pożądanych wartościach pól w zadanym pliku, zaś z drugiej strony znalezienie części OWNER dla dowolnego rekordu z części MEMBER /który wyznacza wystąpienie pewnego określonego DBTG SET/, a także przejrzenie wszystkich rekordów w części MEMBER wystąpienia DBTG SET wyznaczonego przez rekord z z części OWNER jest jasne, że wraz z naszkicowanymi możliwościami manipulacyjnymi konstrukcja DBTG SET umożliwia reprezentowanie związków typu m:1.

Przejdźmy do przykładu.

Pierwszy wariant:



/linie przerywane oznaczają związki, które niekoniecznie muszą być reprezentowane/

---

*/ trudno o dobry polski odpowiednik, nie budzący mylących skojarzeń.

**/ zakłada się, że rekord, który już się pojawił w części MEMBER pewnego wystąpienia konkretnego DBTG SET nie może pojawić się w

/cd przypisu na następnej stronie/

Związki niezbędne:

```
DBTG SET B
  OWNER IS TOWARY
  MEMBER IS DOSTAWY;
DBTG SET F
  OWNER IS WYDZIAŁY
  MEMBER IS TOWARY;
DBTG SET E
  OWNER IS DOSTAWCY
  MEMBER IS WYDZIAŁY:,
```

Związki opcjonalne:

```
DBTG SET A
  OWNER IS DOSTAWCY
  MEMBER IS DOSTAWY;
DBTG SET C
  OWNER IS WYDZIAŁY
  MEMBER IS DOSTAWY;
DBTG SET D
  OWNER IS DOSTAWCY
  MEMBER IS TOWARY;
```

Pytanie A: w jakich /data i numer dostawy/ tegorocznych dostawach dostarczono nakrętek M5 ?

I Krok: odnszukanie rekordu w pliku TOWARY dla którego ID_TOW = 'nakrętka M5'

II Krok: w określonym przez ten rekord wystąpieniu DBTG SET B wybierane są spośród części MEMBER te rekordy /typu DOSTAWA/ dla których DATA > 831231 i dla każdego z tak znalezionych rekordów drukowane są pola DATA i ID_TRANS.

/cd przypisu z poprzedniej strony/ innym wystąpieniu tego samego DBTG SET /ale może pojawić się w dowolnej części innego DBTG SET/.

Pytanie B: jaki jest adres i stan rozliczeń dostawcy, który dostarczył partii towaru o ID_TRANS = 12300 ?

I Krok: odszukanie rekordu w pliku DOSTAWY, dla którego ID_TRANS = 12300

Jeżeli zadeklarowano DBTG SET A /który jest opcjonalny/ **to**

II Krok: w określonym przez ten rekord /który mieści się w części MEMBER/ wystąpieniu DBTG SET A odnajdywany jest OWNER /który jest typu DOSTAWCY/ i drukowane są jego pola ADRES i SALDO.

W przeciwnym przypadku:

II Krok: w określonym przez ten rekord /który mieści się w części MEMBER/ wystąpieniu DBTG SET B odnajdywany jest OWNER /który jest typu TOWARY/

III Krok: w określonym przez ten ostatni rekord /który mieści się w części MEMBER/ wystąpieniu DBTG SET F odnajdywany jest OWNER /który jest typu WYDZIAŁY/

IV Krok: w określonym przez ten ostatni rekord /który mieści się w części MEMBER/ wystąpieniu DBTG SET E odnajdywany jest OWNER /który jest typu DOSTAWCY/ i drukowane są jego pola ADRES i SALDO.

Drugi wariant:

Należy wprowadzić nowe typy rekordów /i ich pliki/, reprezentujące związki typu m:m

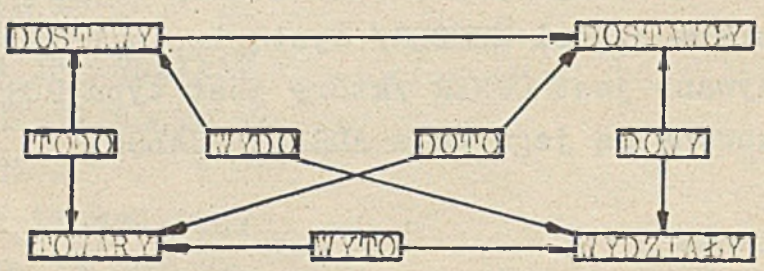
RECORD TODO	związki między towarami i dotawami /B1/
RECORD WYTO	związki między towarami i wydziałami /F1/
RECORD DOWY	związki między wydziałami i dostawcami /E1/

W oróżnieniu od poprzedniego wariantu, gdzie na podstawie związków B, F i E /typu m:1/ można było odtworzyć informację zawartą w związkach C, D i A /również typu m:1/, niezbędne jest wprowadzenie dodatkowych nowych typów rekordów, reprezentujących związki C1 i D1 /typu m:m/ oraz reprezentowanie związku A /typu m:1/. Istotnie, próba ustalenia dostawców pewnego konkretnego towaru /np. nakrętek M5/ na podstawie związków F1 i E1 **może** doprowadzić do następującej sytuacji: na podstawie związku F1 można ustalić listę wydziałów, które otrzymują nakrętki M5, następnie na podstawie związku E1 dla każdego z wydziałów można ustalić listy

ich dostawców i je zsumować; jednakże, ponieważ każdy z wydziałów **może** mieć również dostawców, którzy nie dostarczają nakrętek M5, utworzona w ten sposób lista **może** być błędna, tj zawierać dostawców, którzy w rzeczywistości nie dostarczają nakrętek M5. Podobnie , przy próbie ustalenia dostawcy konkretnej dostawy /z założenia związek typu m:1/ można uzyskać towary dostarczone w tej dostawie, następnie wydziały, dla których te towary są dostarczane oraz dostawców wydziałów, co w ogólnym przypadku **nie musi** się pokrywać z dostawcą występującym w dostawie. Tak więc, niezbędne jest wprowadzenie typów rekordów /i ich plików/ reprezentujących związki:

RECORD DOTO związki między dostawcami a towarami /D1/
RECORD WYDO związki między dostawami a wydziałami /C1/

Model sieciowy w tym wariancie przedstawia się więc następująco:



Według propozycji DBTG niezbędne są więc następujące konstrukcje

```
DBTG SET B11
  OWNER IS DOSTAWY
  MEMBER IS TODO;
DBTG SET B12
  OWNER IS TOWARY
  MEMBER IS TODO;
DBTG SET F11
  OWNER IS TOWARY
  MEMBER IS WYTO;
DBTG SET F12
  OWNER IS WYDZIAŁY
  MEMBER IS WYTO;
DBTG SET E11
```

```
    OWNER IS DOSTAWCA
    MEMBER IS DOWY;
  DBTG SET E12
    OWNER IS WYDZIAŁY
    MEMBER IS DOWY;
  DBTG SET D11
    OWNER IS TOWARY
    MEMBER IS DOTO;
  DBTG SET D12
    OWNER IS DOSTAWCY
    MEMBER IS DOTO;
  DBTG SET C11
    OWNER IS DOSTAWY
    MEMBER IS WYDO;
  DBTG SET C12
    OWNER IS WYDZIAŁY
    MEMBER IS WYDO;
  DBTG SET A
    OWNER IS DOSTAWCY
    MEMBER IS DOSTAWY;
```

Pytanie A:

I: odszukanie rekordu w pliku TOWARY dla którego ID_TOW = 'nakrętka M5'

II: w określonym przez ten rekord wystąpieniu DBTG SET B12 wybierany jest pierwszy rekord spośród części MEMBER /typu TODO/

III: dla znalezionego w ten sposób rekordu typu TODO /określającego wystąpienie DBTG SET B11/ odnajdywany jest OWNER /typu DOSTAWY/ i jeśli spełniony jest warunek DATA > > 831231 to drukowane są pola DATA i ID_TRANS

IV: w określonym w kroku II wystąpieniu DBTG SET B12 wybierany jest następny rekord spośród części MEMBER /jeżeli istnieje; w przeciwnym przypadku następuje zakończenie poszukiwań/ i powraca się do kroku III.

Uwaga: jeżeli ten sam towar występuje wielokrotnie w jednej dostawie, co jest mało sensowne, ale możliwe/ , to dane dotyczące dostaw będą zawierały powtórzenia.

Pytanie B:

I: odszukanie rekordu w pliku DOSTAWY, dla którego ID_TRANS = 12300

II: w określonym przez ten rekord /który mieści się w części MEMBER/ wystąpieniu DBTG SET A odnajdywany jest OWNER /który jest typu DOSTAWCY/ i drukowane są jego pola ADRES i SALDO.

4. Model hierarchiczny

W modelu hierarchicznym graf skierowany, który był matematyczną podstawą modelu sieciowego jest ograniczony do drzewa /drzew/ Jest oczywiste, że te modele konceptualne, które nie implikują drzew bezpośrednio, wymagają specjalnych zabiegów. Prześledzimy je na przykładach.

Pierwszy wariant:

Pytanie A:

I: odszukanie rekordu w pliku TOWARY dla którego ID_TOW = 'nakrętka M5'

II: przeglądnięcie wszystkich "dzieci" znalezionego rekordu /które są typu DOSTAWY/ i wydrukowanie pól DATA i ID_TRANS z wszystkich rekordów dla których DATA > 831231.

Pytanie B:

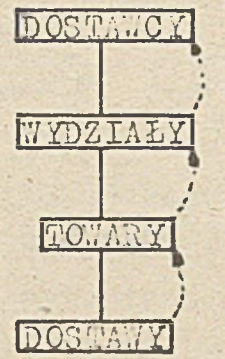
W "czystym" modelu hierarchicznym, który pozwala na wędrówkę po drzewie od rodziców do dzieci znalezienie odpowiedzi na to pytanie jest możliwe /odpowiednia informacja jest więc zawarta w tak zorganizowanej bazie/ ale bardzo pracochłonne:

I: dla pierwszego rekordu w pliku DOSTAWCY należy wyznaczyć wszystkie jego "dzieci" /typu WYDZIAŁY/ a następnie "wnuki" /typu TOWARY/ i "prawnuki" /typu DOSTAWY/

II: jeżeli wśród znalezionych rekordów typu DOSTAWY znajduje się taki, dla którego ID_TRANS = 12300, to z odpowiadającego mu rekordu z pliku DOSTAWCY należy wydrukować pola ADRES i SALDO i proces zakończyć. W przeciwnym przypadku należy powtórzyć wszystkie czynności dla kolejnego rekordu typu DOSTAWCY.

Tak więc, w niekorzystnym przypadku /a zawsze, gdy nie istnieje dostawa o podanym ID_TRANS/ może zajść potrzeba przejrzenia wszystkich plików bazy danych.

Aby tego uniknąć, można wprowadzić odsyłacze od rekordu "dziecka" do rekordu "rodzica". Ułatwienie to nie wykracza poza model hierarchiczny, umożliwia natomiast wędrówkę po drzewie w odwrotnym kierunku. Ponieważ wprowadzenie, a zwłaszcza utrzymywanie systemu odsyłaczy jest kosztowne, należy je stosować wtedy, gdy przewiduje się znaczącą liczbę pytań wymagających korzystania z odsyłaczy. Dla przykładowego pytania, hierarchiczna organizacja bazy danych pokazana poniżej



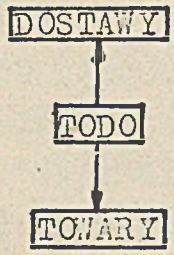
/przerywane linie oznaczają odsyłacze od rekordu "dziecka" do rekordu "rodzica"/ pozwala na zastosowanie następującej metody wyznaczania odpowiedzi:

I: odszukanie rekordu w pliku DOSTAWY, dla którego ID_TRANS równa się 12300

II: dla znalezionego rekordu, na podstawie odsyłacza wyznaczany jest "rodzic" w pliku TOWARY a następnie na podstawie dwóch kolejnych odsyłaczy odnajdywany jest rekord typu DOSTAWCY i drukowana jest zawartość jego pól ADRES i SALDO.

Drugi wariant:

Związki typu m:m mogą być także reprezentowane w modelu hierarchicznym, choć na pierwszy rzut oka wydaje się to niemożliwe, ponieważ w drzewach reprezentowane są bezpośrednio jedynie związki typu m:1. Jednakże, ponieważ baza danych może zawierać więcej niż jedno drzewo, to w przypadku powtarzania się plików w różnych drzewach związki typu m:m są de facto reprezentowane. Np. fragment sieciowej bazy danych



reprezentujący związek typu m:m między dostawami a towarami może być reprezentowany w modelu hierarchicznym w sposób następujący:

DRZEWO_DOSTAW:

DRZEWO_TOWAROW:

Wtedy odpowiedź na pytanie A może być wyznaczona w sposób następujący:

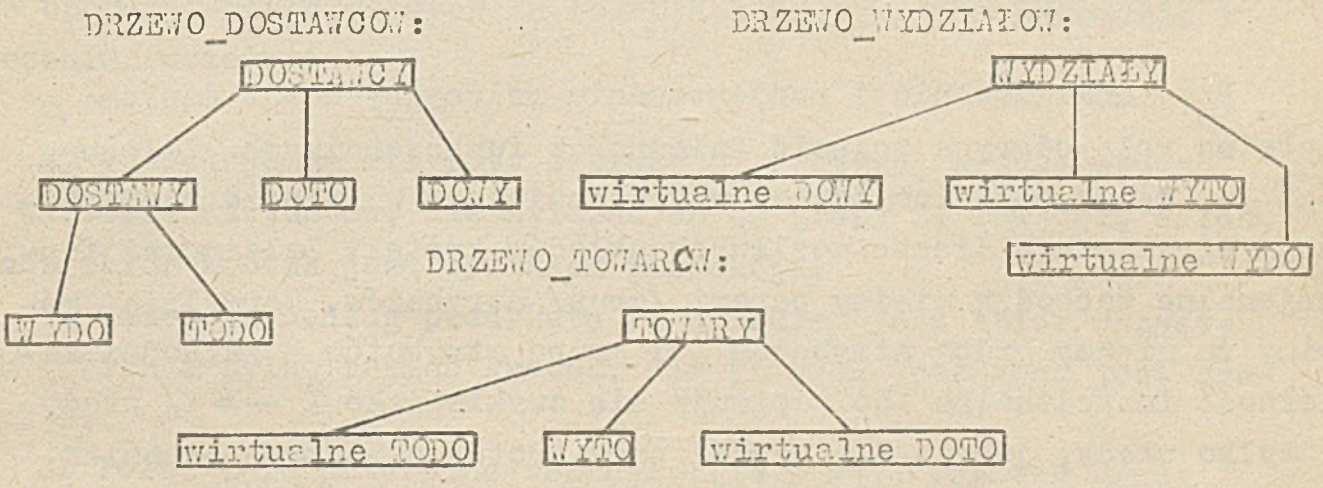
I: w pliku TOWARY, stanowiącym korzeń drzewa DRZEWO_TOWAROW odnajdywany jest rekord dla którego ID_TOW = 'nakrętka M5'

II: dla znalezionego rekordu wyznaczane są wszystkie jego "dzieci" w pliku TODO

III: dla każdego rekordu "dziecka" odnajduje się jego kopię w pliku TODO drzewa DRZEWO_DOSTAW, a następnie na podstawie odsyłaczy odpowiednie rekordy typu DOSTAWY i dla tych rekordów, dla których DATA > 831231 drukowane są pola DATA i ID_TRANS.

Pliki powtarzające się /jak w podanym przykładzie plik TODO/ oprócz ewidentnej straty miejsca stwarzają niebezpieczeń-

stwo powstania niespójności w bazie. Pokazana metoda reprezentowania związków typu m:m wymaga bowiem, aby wszystkie powtarzające się pliki w bazie były dokładnie identyczne, tj w czasie gdy ten warunek nie jest spełniony wyszukiwania powinny być zawieszone. Istnieje możliwość złagodzenia wymienionych trudności poprzez wprowadzenie tzw. rekordów wirtualnych w miejsce rzeczywistych kopii. W istocie, rekord wirtualny zawiera odsyłacz do właściwego rekordu. Dzięki temu, oszczędza się /na ogół/ miejsce, zaś niezbędne modyfikacje i aktualizacje mogą być ograniczone do jednego pliku. Jeden z możliwych wariantów przykładowej bazy danych w postaci hierarchicznej jest pokazany na poniższym rysunku.



Z przykładu wynika od razu główna /poza koniecznością utrzymywania układu odsyłaczy/ wada metody rekordów wirtualnych. Mianowicie, w przypadku pytania o towary dostarczone w pewnej konkretnej dostawie wyszukiwane są rekordy typu TODO /"dzieci" odpowiedniego rekordu typu DOSTAWY/ na podstawie których należy odnaleźć odpowiadające im rekordy wirtualne; wymaga to jednak przejrzenia praktycznie całego pliku wirtualnego. Rozważane poprzednio pytanie o dostawy w których wystąpił określony towar może być przy organizacji bazy takiej, jak to pokazano na rysunku, o wiele łatwiej obsłużone, ponieważ przejście od rekordów wirtualnych do rekordów TODO jest, dzięki odsyłaczom, prawie natychmiastowe.

Trudności te mogą być pokonane przez wprowadzenie dodatkowych odsyłaczy, pozostaje jednak kwestią otwartą, czy taki model jest w dalszym ciągu modelem hierarchicznym.

5. Model relacyjny

Jedynym narzędziem, używanym w modelu relacyjnym do reprezentowania zbiorów obiektów i związków między nimi są relacje. Z matematycznego punktu widzenia relacja jest dowolnym podzbiorem iloczynu kartezjańskiego pewnych zbiorów. W rozumieniu relacyjnych baz danych zbiorami tymi są zbiory możliwych wartości pewnych wielkości, zwanych atrybutami. Praktycznie relacja może być reprezentowana przez dwuwymiarową tablicę, w której kolumnom odpowiadają atrybuty, zaś wiersze reprezentują obiekty lub elementarne związki między obiektami. W terminologii stosowanej w bazach danych wiersz taki nazywany jest krotką.

Przy analizowaniu i projektowaniu relacyjnych baz danych istotną rolę odgrywa pojęcie zależności funkcjonalnych. Zależności te, będąc odbiciem związków występujących w świecie rzeczywistym ograniczają liczbę możliwych układów krotek. Zależności funkcjonalne zachodzą między parami /grup/ atrybutów. Formalnie, mówimy że między grupą atrybutów X i grupą atrybutów Y zachodzi zależność funkcjonalna /co zapisuje się zwykle jako X ---> Y/ wtedy i tylko wtedy, jeżeli dla każdej pary krotek o równych odpowiednio wartościach atrybutów z grupy X wartości atrybutów z grupy Y są również odpowiednio równe. Oznacza to, że w relacji dla której spełnione jest X ---> Y nie może istnieć para krotek o równych wartościach w X i choć jednej parze różnych wartości w Y. Oznacza to również, że zachodzenie zależności funkcjonalnych wprowadza redundancję do reprezentacji danych, ponieważ /dla danego zestawu wartości z X/ znajomość jednego zestawu wartości z Y wystarcza do odtworzenia go we wszystkich krotkach o tym samym zestawie wartości z X.

Klucz K relacji /którym w najgorszym przypadku jest zbiór wszystkich atrybutów/ spełnia zawsze zależność funkcjonalną K ---> a, gdzie a jest dowolnym atrybutem relacji.

W relacji może zachodzić wiele zależności funkcjonalnych, przy czym na ogół nie są one niezależne w tym sensie, że niektóre z nich mogą być implikowane przez inne za pośrednictwem od-

powiednich reguł, np. reguły przechodniości [2], [3].

Przejście z modelu konceptualnego na relacyjny jest bezpośrednie, przy czym poszczególnym zbiorom obiektów odpowiadają relacje o atrybutach odpowiadających atrybutom obiektów, zaś związkom odpowiadają relacje o atrybutach, będących identyfikatorami zbiorów występujących na liście związku. Możliwość takiego przejścia nie oznacza, że uzyskany w taki sposób schemat relacyjny /tj wykaz relacji wraz z ich atrybutami/ będzie dobry z użytkowego punktu widzenia. Zagadnienie to zostanie bliżej omówione na przykładach

Na relacjach można wykonywać trzy podstawowe /jeżeli nie liczyć operacji mnogościowych na relacjach o tym samym schemacie i znaczeniu/ działania:

1. Projekcja na atrybuty z grupy Z; z każdej krotki usuwa się wartości atrybutów, nie należących do grupy Z. Powstały w ten sposób zbiór krotek jest wynikiem projekcji.

2. Selekcja według predykatu P; wynikiem jest zbiór wszystkich takich i tylko takich krotek, dla których spełniony jest predykat P.

3. Połączenie naturalne relacji R(X) i S(Y) /według grupy wspólnych atrybutów V/. Relacja wynikowa ma schemat $X \cup Y$ a jej krotki tworzy się w ten sposób, że każda krotka relacji R jest uwielokrotniana tyle razy, dla ilu krotek z relacji S występuje zgodność wartości dla atrybutów z V, a następnie każdą z nich uzupełnia się wartościami atrybutów z $Y \setminus V$ występujących w znalezionych w opisany sposób krotkach z S.

Powróćmy do standardowego przykładu.
Pierwszy wariant:
Schematy relacji:

DOSTAWY (ID_TRANS, DATA,
TODO (ID_TRANS, ID_TOW,
TOWARY (ID_TOW, JED_MIAR
WYTO (ID_TOW, ID_WYD
WYDZIAŁY (ID_WYD, LICZ_PRAC
DOWY (ID_WYD, ID_DOST
DOSTAWCY (ID_DOST, ADRES, SALDO

WYDO (ID_TRANS, ID_WYD
DOTO (ID_TOW, ID_DOST
DODO (ID_TRANS, ID_DOST

Zależności funkcjonalne:

ID_TRANS → ID_TOW
ID_TOW → ID_WYD
ID_TRANS → ID_WYD /implikowane/
ID_WYD → ID_DOST
ID_TOW → ID_DOST /implikowane/
ID_TRANS → ID_DOST /implikowane/

Ze względu na to, że zachodzi zależność funkcjonalna ID_TRANS → ID_TOW można, nie zmieniając charakteru obiektów w zbiorze DOSTAWY ani ich liczności, związać z każdym obiektem zbioru DOSTAWY wartość atrybutu ID_TOW i zrezygnować z przechowywania relacji TODO, która może zresztą być w każdej chwili odtworzona przez wykonanie projekcji na atrybuty ID_TRANS i ID_TOW zmodyfikowanej relacji NDOSTAWY (ID_TRANS, DATA, ID_TOW, .. . Zwróćmy uwagę, że gdyby nie obowiązywała zależność funkcjonalna ID_TRANS → ID_TOW, to w relacji NDOSTAWY, powstałej w wyniku połączenia naturalnego DOSTAWY * TODO ID_TRANS przestałoby być identyfikatorem. Na tej samej zasadzie można utworzyć relacje:

NTOWARY (ID_TOW, JED_MIAR, ID_WYD

oraz

NWYDZIAŁY (ID_WYD, LICZ_PRAC, ID_DOST

i zrezygnować z przechowywania relacji WYTO i DOWY. Po wprowadzeniu tych trzech relacji można zrezygnować również z przechowywania relacji WYDO, DOTO i DODO, ponieważ np. relację DODO można odtworzyć przez projekcję na atrybuty ID_TRANS i ID_DOST połączenia naturalnego:

NDOSTAWY * NTOWARY * NWYDZIAŁY

Odpowiedź na pytanie: w jakich tegorocznych dostawach i

kiedy dostarczono nakrętek N5? można wyznaczyć stosując projekcję na atrybuty ID_TRANS i DATA podzbioru krotek relacji NDOSTAWY, dla których DATA > 831231 i ID_TOW = nakrętka N5

Odpowiedź na pytanie: jaki jest adres i stan rozliczeń dostawcy, który dostarczył partii towaru o ID_TRANS = 12300 można wyznaczyć stosując projekcję na atrybuty ADRES i SALDO rezultatu następujących działań na relacjach:

I: w relacji NDOSTAWY dokonuje się selekcji krotek, dla których ID_TRANS = 12300 /w tym przypadku powstanie relacja zawierająca co najwyżej jedną krotkę/

II: dla tak uzyskanej relacji tworzy się połączenie naturalne z relacjami NTOWARY, NWYDZIAŁY i DOSTAWCY; powstała relacja ma co najwyżej jedną krotkę i jeżeli jest ona niepusta, to poddaje się ją projekcji opisanej na wstępie /pusta relacja wynikowa oznacza, że baza danych nie zawiera informacji umożliwiających wyznaczenie odpowiedzi na postawione pytanie/.

Fakt, że wszystkie związki w omawianym przykładzie są typu m:1 pozwala na zmodyfikowanie schematu bazy danych w ten sposób, że wyznaczanie odpowiedzi na niektóre pytania ulegnie uproszczeniu, a jednocześnie nie zmienia się charakter i liczność zbioru obiektów. Np. wprowadzenie relacji:

NNDOSTAWY(ID_TRANS,DATA,ID_TOW,ID_WYD,ID_DOST
NNTOWARY (ID_TOW,JED_MIAR,ID_WYD,ID_DOST

i pozostawienie relacji NWYDZIAŁY oraz DOSTAWCY pozwala na zredukowanie II kroku procesu wyznaczania odpowiedzi na drugie pytanie do połączenia naturalnego krotki wyznaczonej w pierwszym z relacji NNDOSTAWY z relacją DOSTAWCY.

Przedstawione rozwiązanie obok oczywistych zalet ma również wady. Po pierwsze, widoczna jest redundancja danych /np związek między ID_TOW oraz ID_WYD przechowywany jest w relacjach NNDOSTAWY i NNTOWARY/ po drugie zaś, schemat relacji może być źle dostosowany do procesu zbierania danych, np. w momencie rejestrowania dostawy wydział, dla którego jest ona przeznaczona

może być nieznany, co może powodować trudności w zarejestrowaniu dostawy. Zwiększona redundancja powoduje zwiększenie nakładów na aktualizację bazy danych, np. zmiana wartości ID_DOST powoduje konieczność zmian w NNDOSTAWY /tyle razy, ile dany dostawca wystąpił w dostawach/, w NNTOWARY /tyle razy, ile różnych towarów dostarcza dany dostawca/ w NWYDZIAŁY /tyle razy, ile różnych wydziałów obsługuje dany dostawcą, nie ma zwiększenia/ oraz w relacji DOSTAWCY /jednokrotnie, bez zmian/.

Dzięki temu, że wymienione dotychczas atrybuty są w zależności funkcjonalnej z identyfikatorami poszczególnych relacji można wszystkie interesujące dane zarejestrować w jednej relacji

NNNDOSTAWY (ID_TRANS,DATA,ID_TOW,JED_MIAR,ID_WYD,LICZ_PRAC,
ID_DOST,ADRES,SALDO

co może ułatwić wyszukiwanie informacji, natomiast redundancja jest duża, ponieważ np. /aktualne/ saldo dostawcy występuje w tylu krotkach, w ilu dostawach uczestniczył dany dostawca.

Rozwiązanie to, aczkolwiek bezpośrednio /=fizycznie/ jest raczej nie zalecane do stosowania, to może być korzystne pojęciowo do organizowania współpracy z niezaawansowanym użytkownikiem /tzw. relacja uniwersalna, [14]/. Z przedstawionych wyżej rozważań wynika, że względnie dobrym, kompromisowym schematem relacyjnej bazy danych może być układ następujący:

XDOSTAWY (ID_TRANS,DATA,ID_TOW,ID_DOST
NNTOWARY (ID_TOW,JED_MIAR,ID_WYD,ID_DOST
NWYDZIAŁY (ID_WYD,LICZ_PRAC,ID_DOST
DOSTAWCY (ID_DOST,ADRES,SALDO

Drugi wariant:

Ponieważ między identyfikatorami zbiorów obiektów tylko jeden związek jest typu m:1 /wiąże się z nim zależność funkcjonalna ID_TRANS ⟶ ID_DOST/ jedynie relacja DODO może być włączona do relacji DOSTAWY, nie powodując zmiany charakteru i liczności zbioru obiektów. Powstanie więc relacja

MDOSTAWY (ID_TRANS,DATA,ID_DOST

zaś pozostałe relacje pozostawimy na razie bez zmian w stosunku do wariantu pierwszego.

Odpowiedź na pytanie o tegoroczne dostawy nakrętek M5 można wyznaczyć w sposób następujący:

I: z relacji TODO wybiera się te krotki, dla których ID_TOW = 'nakrętka M5'
II: wykonuje się połączenie naturalne otrzymanej w ten sposób relacji z relacją MDOSTAWY
III: z otrzymanej relacji wybiera się te krotki, dla których DATA > 831231
IV: dokonuje się projekcji otrzymanej relacji na atrybuty ID_TRANS i DATA.

Odpowiedź na pytanie o adres i saldo dostawcy, który dostarczył partii towaru o ID_TRANS = 12300 można wyznaczyć w sposób następujący:

I: z relacji MDOSTAWY wybiera się te krotki, dla których ID_TRANS = 12300 /w tym przypadku co najwyżej jedną/
II: wykonuje się połączenie naturalne tak otrzymanej relacji z relacją DOSTAWCY
III: dokonuje się projekcji wyniku na atrybuty ADRES i SALDO

Również w opisywanym wariancie możliwe jest włączanie relacji oznaczanych skrótami czteroliterowymi /które reprezentują związki/ do relacji reprezentujących zbiory obiektów. Należy jednak zwrócić uwagę na skutki tego rodzaju modyfikacji modelu:

1.Dołączenie takie zmienia charakter obiektów reprezentowanych przez zmodyfikowaną relację oraz liczność zbioru. Np. jeżeli utworzy się relację MMDOSTAWY = MDOSTAWY * TODO to poszczególne krotki tej nowej relacji reprezentują nie dostawy lecz - używając sformułowania biurokratycznego - poszczególne wiersze faktury. ID_TRANS nie jest już wtedy identyfikatorem obiektów reprezentowanych przez relację MMDOSTAWY. Identyfikatorem takim może

być para wartości atrybutów ID_TRANS i ID_TOW. Można także wprowadzić nowy identyfikator obiektów ID_WF /np. numer kolejny wprowadzanego wiersza faktury/ co byłoby usprawiedliwione wtedy, gdy chcielibyśmy rozszerzyć możliwości rejestrowania danych w bazie o przypadek, w którym ten sam towar występuje wielokrotnie w jednej dostawie /w przedstawionym modelu relacyjnym nie jest to możliwe, ponieważ obiekty reprezentowane przez relację TODO identyfikowane są na podstawie pary wartości atrybutów ID_TRANS,ID_TOW/. Jeżeli przypadki takie w rzeczywistości nie zachodzą, to zachodzi zależność funkcjonalna ID_WF ⟶ ID_TRANS,ID_TOW i wprowadzanie identyfikatora ID_WF jest jałowe.

2. Połączenie naturalne dwóch relacji nie zawsze jest informacyjnie bezstratne, tj nie zawsze można przez odpowiednie projekcje odzyskać relacje składowe [1]. Np. jeżeli w pewnym momencie relacja WYTO zawiera **wszystkie** dane dotyczące wszystkich towarów używanych przez wszystkie wydziały, to utworzenie relacji

MTOWARY = TOWARY * WYTO

w sytuacji, w której nie wszystkie towary są opisane w relacji TOWARY spowoduje utracenie informacji o przyporządkowaniu wydziałom tych towarów, które nie były opisane. Informacja ta musi być uzupełniana w chwili rejestrowania opisu nowego towaru, ponieważ bez znajomości wartości atrybutu ID_WYD dla tego towaru nie można /bez przyjęcia dodatkowych umów dotyczących "pustych" wartości/ utworzyć krotki w relacji MTOWARY. Zbiór reprezentowany przez relację MTOWARY nie jest już zbiorem opisów towarów /jak w przypadku relacji TOWARY/ lecz pewnego rodzaju rozdzielnikiem towarów na wydziały.

3. Opisane w poprzednich punktach relacje MMDOSTAWY i MTOWARY zawierają zbędne powtórzenia wartości atrybutów, np. w relacji MMDOSTAWY wartość atrybutu DATA /i ewentualnie innych niewymienionych/ zostanie powtórzona tyle razy, ile towarów wystąpiło w danej dostawie, podobnie jak w relacji MTOWARY wartość atrybutu JED_MIAR zostanie powtórzona dla każdego wydziału.

Możliwe jest również łączenie - wyjściowych lub zmodyfikowanych - relacji reprezentujących zbiory obiektów. Jednakże, o ile połączenie relacji MDOSTAWY z relacją TOWARY nie spowoduje negatywnych skutków ubocznych /poza powtarzaniem wartości JED_MIAR dla każdego towaru tyle razy, w ilu dostawach wystąpił/ to nie jest to regułą. Np. utworzenie relacji MDOSTAWCY = DOSTAWCY*DOWY /poza powtarzaniem wartości atrybutów ADRES i SALDO/ nie daje jeszcze negatywnych efektów ubocznych, natomiast jej naturalne połączenie z relacją MDOSTAWY i utworzenie relacji DOSTAWY_DOSTAWCY o atrybutach

ID_TRANS,DATA,ID_DOST,ADRES,SALDO,ID_WYD

powoduje powstanie następujących efektów ubocznych:

1. Między atrybutami tak utworzonej relacji pojawił się nowy rodzaj zależności, tzw. zależności wielowartościowe. Istota tych zależności polega na tym, że jeżeli można podzielić zbiór atrybutów relacji na trzy rozłączne podzbiory X,Y i Z takie, że występowanie krotek

x_1yz_2
x_2yz_1

pociąga za sobą występowanie krotki x_1yz_1 /a przez przemianowanie wskaźników również krotki x_2yz_2/ to mówimy, że zachodzi zależność wielowartościowa $Y \twoheadrightarrow X$ /a przez symetrię także $Y \twoheadrightarrow Z$/. Własność ta jest bardzo niekorzystna z punktu widzenia utrzymywania bazy danych, ponieważ zarówno dodanie nowej krotki jak też usunięcie krotki istniejącej może prowadzić do dodania /usunięcia/ wielu krotek. W rozważanym przykładzie występują dwie nietrywialne zależności wielowartościowe ID_DOST $\twoheadrightarrow$ ID_WYD oraz ID_DOST $\twoheadrightarrow$ ID_TRANS. Istotnie, dla konkretnego dostawcy lista obsługiwanych przez niego wydziałów musi być powtórzona /w postaci krotek z jednakowymi wartościami pozostałych atrybutów/ dla każdej występującej kombinacji pozostałych atrybutów; podobnie rzecz ma się z listą dostaw, w których uczestniczył ten dostawca. Nazwa "zależności wielowartościowe" wiąże się z powyższym wywodem, gdzie widać wyraźnie, że w odróżnieniu od zależno-

ści funkcjonalnych, w których wartości pewnej /grupy/ atrybutów determinowały wartości /innej/ grupy atrybutów, w przypadku zależności wielowartościowych wartości pewnych atrybutów determinują pewien zbiór wartości /innych/ atrybutów.

2. Mogą powstać /szczególnie u nieprzygotowanego użytkownika/ wątpliwości co do interpretacji poszczególnych atrybutów. Np. próba określenia wydziałów, na które kierowana jest określona dostawa na podstawie relacji DOSTAWY_DOSTAWCY /przez selekcję krotek z odpowiednią wartością ID_TRANS, a następnie projekcję na atrybut ID_WYD/ może prowadzić do błędu, ponieważ ze sposobu budowania relacji wynika, że ID_WYD odpowiada dostawcom obługującym wydziały, a nie dostawom.

Przykład powyższy świadczy o tym, że sama znajomość atrybutów relacji /bez znajomości zależności funkcjonalnych i innych [3], [7], [10], [11], [12] a także ścisłej interpretacji znaczenia wszystkich atrybutów/ nie może stanowić dostatecznej podstawy do korzystania z tej relacji.

6. Atrybuty

W dotychczasowych rozważaniach obiekty zgrupowane w rozpatrywanych zbiorach były bardzo skromnie wyposażone w atrybuty. Poza identyfikatorami, które są niezbędne do odróżniania obiektów w zbiorze, dodano jedynie nieliczne atrybuty, przydatne w zasadzie jedynie do łatwiejszego zilustrowania procesu wyznaczania odpowiedzi na przykładowe pytania. Wszystkie te atrybuty /jeżeli przyjmiemy, że ilość towaru podaje się zawsze w tych samych jednostkach miary, a dostawcy nie mają filii "widocznych" dla klienta/ są "bezpieczne" w tym sensie, że nie zmieniają charakteru obiektów ani liczności ich zbioru ani identyfikatora obiektu. Każdy "bezpieczny" atrybut może być dodany do opisu obiektu bez żadnych konsekwencji dla pozostałych części modelu /sieciowego, hierarchicznego czy też relacyjnego/. Atrybut jest "bezpieczny" wtedy i tylko wtedy gdy zachodzi zależność funkcjonalna identyfikator_zbioru ⟶ atrybut /tj wartość identyfikato-

determinuje wartość atrybutu/.

Powyższe stwierdzenia są oczywiste podano je jednak aby wyraźniej pokazać różnicę między pojęciami obiektu i atrybutu używanymi w modelach danych, a pojęciami stosowanymi w potocznym rozumieniu.

Np. w potocznym rozumieniu cena jednostkowa jest niewątpliwie cechą towaru. W modelu danych atrybut CENA_JEDN może być dodany do opisu obiektów w zbiorze TOWARY bez żadnych konsekwencji jeżeli jest on "bezpieczny" tj jeżeli identyfikator towaru /np. symbol pewnej klasyfikacji towarowej/ jednoznacznie określa cenę. Jeżeli jednak różni dostawcy oferują ten sam towar po różnych cenach, to proste dołączenie pola /atrybutu/ CENA_JEDN do rekordów nie będzie rozwiązaniem dobrym. Mianowicie, w opisanej sytuacji nie jest sensowne zadawanie pytań o cenę określonego towaru bez podania dostawcy. Przy opisanej strukturze bazy nawet znajomość identyfikatora towaru i dostawcy nie pozwala na wyznaczenie odpowiedzi na to pytanie. Istotnie, znajomość identyfikatora dostawcy pozwala co najwyżej sprawdzić, czy dany dostawca rzeczywiście dostarcza określonego towaru/plik lub relacja DOTO/ zaś w pliku TOWARY można wyznaczyć albo ostatnią zarejestrowaną cenę towaru /jeżeli identyfikująca rola ID_TOW była traktowana poważnie przy aktualizacji/ albo pewien zbiór cen /jeśli nie przestrzegano tego ograniczenia/; w pierwszym przypadku nie wiadomo, czy jest to cena oferowana przez interesującego nas dostawcę, w drugim - która z wyznaczonych cen jest właściwa.

Aby pokonać te trudności należy przebudować model danych. Najprostszym rozwiązaniem wydaje się dołączenie atrybutu ID_DOST do zbioru TOWARY, dzięki czemu wyznaczenie odpowiedzi na omawiane wyżej pytanie staje się bezpośrednio możliwe. Wadą tego rozwiązania jest konieczność powtarzania wartości atrybutu JED_MIAR /i ewentualnie innych "bezpiecznych" atrybutów opisujących towary/ we wszystkich krotkach /rekordach/ dotyczących tego samego towaru, ale pochodzącego od różnych dostawców. Ponadto, wyznaczanie odpowiedzi na pytania dotyczące własności towarów, które są identyfikowane za pośrednictwem relacji /plików/ TODO i/lub WYTO, będzie prowadziło do wyników zawierających powtórzenia, do

wyeliminowania których potrzebna jest dodatkowa operacja /projekcja pomijająca atrybut ID_DOST/. Dzieje się tak dlatego, że ID_TOW przestał być identyfikatorem zbioru TOWARY /dopiero para wartości ID_TOW,ID_DOST identyfikuje obiekt w tym zbiorze/, a pozostał w relacjach /plikach/ TODO i WYTO. Sam zbiór zmienił zresztą przy tej operacji charakter i jest raczej zbiorczym katalogiem wyrobów niż zbiorem opisów towarów.

Inne rozwiązanie może polegać na włączeniu atrybutu CENA_JEDN do relacji DOTO, co eliminuje wszystkie wady poprzedniego rozwiązania. Jednakże, i to rozwiązanie w bezpośredniej postaci ma wadę polegającą na tym, że w przypadku zaniechania przez pewnego dostawcę dostaw pewnego towaru należałoby wycofać odpowiednią krotkę /rekord/ z relacji /pliku/ DOTO po to, aby móc udzielać informacji o aktualnych dostawcach określonego towaru. Z drugiej strony, zniknie również wartość ceny, co uniemożliwi określenie wartości towarów pochodzących z niektórych wcześniejszych dostaw. Można temu zaradzić przez dodanie do DOTO atrybutu o wartościach określających datę zaniechania dostaw /dla dostawców "aktywnych" byłaby to pewna sztuczna, odległa data/. Wtedy wartość ceny byłaby przez cały czas dostępna, zaś przy udzielaniu informacji o aktualnych dostawcach należałoby pomijać krotki z datą zaniechania mniejszą od daty aktualnej. Problemy związane z utrzymywaniem takiej relacji zostaną omówione w następnym rozdziale.

Pewne zastrzeżenia może budzić fakt, że bardzo konkretny atrybut CENA_JEDN został włączony do relacji DOTO, która w sposób abstrakcyjny reprezentuje związki między dostawcami a towarami. Jeżeli jednak /myślowo/ przemianujemy tę relację na CENNIK, to wtedy będziemy w zgodzie również z intuicją, gdyż każda pozycja cennika powinna być opisana przez wprowadzone przez nas atrybuty.

Podobnie, ilość towaru dostarczonego w danej dostawie najdogodniej jest włączyć, w postaci atrybutu np. IL_TOW do relacji TODO.

Z punktu widzenia tworzenia i utrzymywania baz danych najdogodniej jest, gdy wszystkie obiekty w danym zbiorze są jednorod-

ne, tj dla każdego obiektu powinien istnieć zestaw możliwych do określenia wartości ustalonego zbioru atrybutów. W praktyce może się zdarzyć, że pewne atrybuty są stosowalne nie do wszystkich obiektów zbioru, a także wartości niektórych atrybutów mogą być nieznane w chwili rejestrowania /z uwagi na organizację procesów rejestrowania obiektów i aktualizacji bazy/. Np. zbiór TOWARY na ogół będzie bardzo niejednorodny /można co najwyżej wydzielić pewne grupy towarowe o takich samych atrybutach/, zaś wyniki badań jakościowych dla poszczególnych dostaw niektórych towarów, które mogą być atrybutami niektórych obiektów w zbiorze TODO mogą być znane dopiero po pewnym czasie od chwili wpisania do krotki wartości atrybutów ID_TRANS, ID_TOW /stanowiących identyfikator tego zbioru/ i IL_TOW.

Aby uniknąć kłopotów związanych z opisaną sytuacją, które wymagają stosowania "pustych" wartości atrybutów co najmniej dwóch rodzajów /dla odróżnienia obydwu przypadków/ i bezpowrotnych strat miejsca w pierwszym przypadku można dokonać przebudowy modelu danych. Przebudowa ta polega na pozostawieniu w rozpatrywanych zbiorach tylko tych atrybutów, które są stosowalne i znane w chwili rejestracji dla każdego obiektu zbioru /w szczególnym przypadku będą to wyłącznie atrybut/y/ identyfikatora zbioru/. Następnie tworzone są jednorodne /pod względem stosowalności i czasu poznania wartości atrybutów/ podzbiory tego zbioru, w których stosuje się ten sam identyfikator co w zbiorze głównym, a pozostałe atrybuty są dobierane pod kątem spełnienia warunku jednorodności.

Atrybut identyfikatora może być utworzony w sposób sztuczny /np w momencie rejestrowania w systemie obiektowi może być nadawany automatycznie numer kolejny w zbiorze/. W takim przypadku, jeżeli stosuje się podzbiory do przechowywania wartości atrybutów "opóźnionych" to przydzielany automatycznie numer musi zostać związany z obiektem. Rolę identyfikatora mogą również spełniać wartości atrybutów samych obiektów.

7. Aktualizacja

Pojawianie się i znikanie obiektów świata rzeczywistego, związków między nimi a także zmiany wartości atrybutów istniejących powodują konieczność odpowiedniego modyfikowania zawartości bazy danych. Już w chwili tworzenia modelu konceptualnego można podjąć decyzje dotyczące określenia zbiorów obiektów, związków między nimi i przydziału atrybutów, które będą rzutowały na sposób, a nawet wykonalność procesu aktualizacji zawartości bazy danych. Np. jeżeli do zbioru TOWARY zostanie dołączony atrybut CENA_JEDN, to jeśli pojawi się nowy dostawca, oferujący towar już zarejestrowany po innej cenie, to jedynymi możliwymi reakcjami są:

1. zmienić zawartość pola /atrybutu/ CENA_JEDN w odpowiednim dla danego towaru rekordzie /krotce/
2. zignorować aktualizację
3. wprowadzić nowy rekord z tą samą wartością ID_TOW i pozostałych atrybutów z wyjątkiem atrybutu CENE_JEDN, który otrzymuje nową wartość

Dowolna z tych reakcji powoduje, że zawartość bazy danych rozmija się z aktualną wiedzą o świecie rzeczywistym. Można wprawdzie twierdzić, że model konceptualny był po prostu błędny /i tak jest w istocie/, ale błędy takie wcale nie są rzadkie i tak łatwe do wykrycia jak w omówionym przykładzie.

Innego rodzaju trudności powstają jeśli między atrybutami obiektów występują nietrywialne zależności funkcjonalne inne niż zależności typu klucz ⟶ atrybut. Prowadzi to do sytuacji, w której zmiana wartości atrybutu pewnego obiektu rzeczywistego musi być dokonana w wielu rekordach /krotkach/. Np. jeżeli w celu skorygowania błędnego modelu konceptualnego z poprzedniego przykładu do zbioru TOWARY dołączy się atrybut ID_DOST, to ID_TOW przestanie być identyfikatorem /kluczem/ zaś zależność funkcjonalna ID_TOW ⟶ JED_MIAR pozostanie. Wtedy zmiana wartości atrybutu JED_MIAR /np. ze stóp na metry/ musi być dokonana we wszystkich rekordach o takiej samej wartości ID_TOW, choć zmiana w rzeczywistości dotyczy jednej cechy jednego intuicyjnego obiektu.

W zależności od wyboru obiektów i przyporządkowania im atrybutów między atrybutami obiektu /we wszystkich omawianych modelach danych/ mogą zachodzić zależności różne od funkcjonalnych [3], [10], [11], [12]. Ogólnie, zależności można podzielić na generujące i niegenerujące. Przykładem pierwszych są omówione wcześniej zależności wielowartościowe. Zależności generujące są szczególnie szkodliwe przy aktualizacji /dołączenie obiektu, usunięcie obiektu, zmiana wartości atrybutu obiektu istniejącego/, mogą spowodować konieczność dodania lub usunięcia krotek nie związanych bezpośrednio z rozpatrywanym obiektem i to nie zawsze w sposób jednoznaczny. Definicja zależności generującej może być bowiem sprowadzona do takiej postaci, w której ze spełnienia warunku występowania pewnych obiektów wynika konieczność występowania innych obiektów. Przy zależnościach niegenerujących wymaga się, aby wartości atrybutów pewnej skończonej grupy obiektów spełniały określony warunek. Jeżeli nie jest on spełniony, to jeden z obiektów tej grupy nie jest obiektem "legalnym". Wynika stąd, że jedynie próba dołączenia nowego obiektu /zmiana wartości atrybutu jest równoważna logicznie usunięciu pewnego starego i dodaniu nowego obiektu/ może spowodować "konflikt", który musi być rozstrzygnięty w odpowiedni sposób. Zależności funkcjonalne są przykładem zależności niegenerujących. W omawianym wcześniej przykładzie aktualizacji zbioru TOWARY konflikt spowodowała zmiana wartości atrybutu JED_MIAR w jednym z obiektów tego zbioru i został on rozwiązany przez zmianę tej wartości w pozostałych obiektach tego zbioru o tej samej wartości ID_TOW /można byłoby go także rozwiązać przez wycofanie się z pierwszej zmiany/.

Z badań prowadzonych przez autora wynika, że zawsze możliwy jest taki dobór obiektów i atrybutów, że nie występują zależności różne od funkcjonalnych /i to takich, że spełniony jest warunek postaci normalnej Boyce'a i Codda [7]/. Wymaga to jednak przyjęcia założenia, że nie wszystkie kombinacje wartości atrybutów są możliwe. Obecnie trudno ocenić praktyczną przydatność tego stwierdzenia.

Jednym ze źródeł trudności związanych z aktualizacją jest fakt, że w poszczególnych modelach danych udostępniane są narzędzia służące np do aktualizowania pól w rekordach /krotkach/,

całych rekordów, dołączania i odłączania "dzieci" i "rodziców" w drzewie, aktualizacji DBTG SET itd., natomiast fakty zachodzące w świecie rzeczywistym nie zawsze przekładają się na te operacje w oczywisty sposób. Np. utrzymywanie relacji DOTO opisanej w poprzednim rozdziale i przeznaczonej do rejestrowania cen towarów można przedstawić następująco:

Fakt: dostawca X dostarcza towaru Y po cenie Z
Działania:
I: przeszukiwana jest relacja DOTO w celu znalezienia krotek dla których ID_DOST = X oraz ID_TOW = Y; jeżeli nie ma takiej krotki, to
II: tworzy się ją z następującymi wartościami atrybutów: ID_DOST = X, ID_TOW = Y, CENA_JED = Z, DATA_WAŻN = MAX;
III: jeżeli jest dokładnie jedna taka krotka to sprawdza się, czy CENA_JED = Z i jeśli tak, to zmienia się wartość atrybutu DATA_WAŻN na MAX; w przeciwnym przypadku tworzy się nową krotkę jak w kroku II;
IV: jeżeli jest więcej takich krotek, to wybierana jest krotka z największą wartością DATA_WAŻN i dalej postępuje się dokładnie tak jak w kroku III.

Fakt: dostawca XX zaprzestał dostarczania towaru YY od dnia ZZ
Działania:
I: przeszukiwana jest relacja DOTO w celu znalezienia krotek dla których ID_DOST = XX, ID_TOW = YY i DATA_WAŻN $\geq$ ZZ
II: jeżeli nie ma takiej krotki, to może to być jedynie wynikiem błędu;
III: jeżeli jest dokładnie jedna taka krotka, to zmienia się w niej wartość DATA_WAŻN na ZZ - 1.
IV: jeżeli takich krotek jest więcej, to prawdopodobnie jest to sytuacja związana z wystąpieniem błędu.

Jak łatwo zauważyć, nie wszystkie dane potrzebne do skontruowania powyższych procesów można wydedukować z modelu danych.

8. Bibliografia

1. Aho A.V., Beeri C., Ullman J.D., The Theory of Joins in Relational Databases, ACM Trans. on DATABASE SYST. 4:3, 1979

2. Armstrong W. W., Dependency Structures of Data Base Relationships, Proc. 1974 IFIP Congress, 580-583, North Holland

3. Beeri C., Fagin R., Howard J.H., A Complete Axiomatization for Functional and Multivalued Dependencies, Proc. ACM-SIGMOD Conf. Toronto 1977 47-61

4. Chen P.P., The Entity-Relationship Model: Toward a Unified View of Data, ACM Trans. on DATABASE SYST. 1:1, 1976

5. CODASYL Data Base Task Group April 71 Report, ACM New York

6. Codd E.F., A Relational Model for Large Shared Data Banks Comm. ACM 13:6

7. Codd E.F., Further Normalization on the Data Base Relational Model, w: Data Base Systems /wyd. R. Rustin/ Prentice-Hall, Englewood Cliffs, New Jersey 1972

8. Date C.J., An Introduction to Database Systems, Addison-Wesley, Reading Mass.,1981

9. IMS/VS publication series, GH20-1260, General Information

10. Mendelzon A.O., Maier D., Generalized Mutual Dependencies and the Decomposition of Database Relations, Proc. Int. Conf. on VLDB 1978, 75-82

11. Nicolas J.M., Mutual Dependencies and some Results on Undecomposable Relations, ONERA-CERT, Toulouse, France, Feb. 78

12. Paredaens J., Jannsens D., Decomposition of Relations: a Comprehensive Approach, w: Gallaire, Minker, Nicolas - Advances in Database Theory, v.1, Plenum Press, New York 1980

13. Tsichritzis D.C., Lochovsky F.H., Data Base Management Systems, Academic Press, New York 1977

14. Ullman J.D., Principles of Data Base Systems, Computer Science Press, 1982.

Jesienna Szkoła PTI
Rydzyna, październik 1984

DENOTACYJNA METODA SPECYFIKACJI OPROGRAMOWANIA

Andrzej Blikle
Instytut Podstaw Informatyki
Polskiej Akademii Nauk
PKiN skr. poczt. 22
00-901 Warszawa, tel. 203888

1. Wstęp

W ostatnich latach dokonuje się w świecie przemysłu software'owego swoista rewolucja intelektualno-technologiczna. Polega ona na coraz szerszym wprowadzaniu do praktyki formalnych metod projektowania, wytwarzania i pielęgnacji oprogramowania. Metody znane jeszcze kilka lat temu wyłącznie w wąskich środowiskach akademickich zaczynają propagować do przemysłu. Dokonuje się to dwoma kanałami. Po pierwsze, mury wielu uczelni na świecie opuszczają dziś absolwenci przygotowani do stosowania tych metod na skalę przemysłową. Po drugie, wiele firm software'owych inicjuje szeroko zakrojone programy badawczo-szkoleniowe na własny użytek.

Wśród metod zdobywających uznanie praktyków najpowszechniej stosowaną jest tzw. metoda denotacyjna wraz z jej przemysłową wersją VDM (ang. Vienna Development Method)[4]. Celem niniejszego artykułu jest przekazanie czytelnikowi polskiemu kilku sygnalnych wiadomości dotyczących tej metody, a także warunków i przy-

czyn jej powstania, powodów jej sukcesu, zakresu zastosowań, dostępnej w kraju i na świecie literatury. Oczywiście niewielki raport konferencyjny nie może pretendować do roli chociażby wprowadzenia w metodę i jej zastosowania. Wymagałoby to tekstu o objętości sporej książki oraz co najmniej pełnosemestralnego wykładu. Niżej podpisany ma jednak nadzieję, że ta krótka informacja przyczyni się do wzrostu zainteresowania metodą denotacyjną w naszym kraju i spowoduje podjęcie działań zmierzających do jej upowszechnienia.

2. Trochę historii i stan obecny

Inżynieria oprogramowania jest jedyną dziś na świecie dyscypliną techniczną, w której projektowanie dużych, złożonych i spełniających nieraz bardzo odpowiedzialne funkcje systemów przemysłowych prowadzone jest najczęściej metodami rzemieślniczymi. Tam gdzie w elektronice, mechanice, budownictwie, przeprowadza się obliczenia oparte na dobrze ugruntowanej teorii, tam w inżynierii oprogramowania twórcy systemu opierają się (najczęściej) jedynie na własnym doświadczeniu i intuicji. Gwarancję jakości i niezawodności systemu uzyskuje się właściwie wyłącznie na drodze testowania i wstępnej eksploatacji. Mimo że te zabiegi pochłaniają nierzadko połowę (i więcej!) kosztu całego projektu, uzyskiwane wyniki bywają dalekie od zadowalających. Jest powszechnie znanym faktem, że większość poważnych katastrof w programie kosmicznym USA, a także szereg innych spektakularnych wypadków - np. w elektrowniach atomowych - spowodowanych zostało błędami w systemach oprogramowania.

Opisane wyżej problemy są od lat przedmiotem badań wielu naukowców. Badania takie rozpoczęto właściwie jednocześnie z narodzinami informatyki. Już w roku 1949 znakomity matematyk angielski, pionier informatyki teoretycznej, Alan Turing (twórca znanego pojęcia maszyny Turinga) ogłosił pracę [16] poświęconą matematycznej metodzie weryfikacji zgodności programu ze specyfikacją. Niestety, praca Turinga - zapewne zbyt wybiegająca w przyszłość - nie została przez współczesnych zauważona. Dopiero w 18 lat później R.W. Floyd [8] odkrywszy i opisawszy tę metodę ponownie doprowadził do jej wielkiej popularności. Dziś wchodzi ona do klasyki matematycznej teorii programowania.

Poważny rozwój badać matematycznych w teorii programowania datuje się mniej więcej od połowy lat sześćdziesiątych. Jako pierwsze zostały opisane i weszły na stałe do repertuaru narzędzi praktyków metody związane z definiowaniem składni języków programowania (BNF) oraz ich rozbiorem syntaktycznym (parsing). Dziś nie sposób wyobrazić już sobie, aby przy projektowaniu i implementacji jakiegokolwiek języka programowania nie posłużono się tymi metodami.

Opis składni języka jest oczywiście tylko jednym spośród wielu problemów, jakie należy rozwiązać przy projektowaniu języków i systemów. Drugą, bardzo ważną grupę stanowią zagadnienia związane z opisem semantyki oraz badanie poprawności tej semantyki ze względu na różne kryteria. W tej grupie zagadnień natrafiono na szereg trudnych problemów teoretycznych i praktycznych. Okazało się, że do opisu i analizy semantyki stworzyć trzeba nowe narzędzia matematyczne. Mimo że w badania nad tymi problemami zaangażowanych było wiele ośrodków naukowych na świecie, pierwsze wyniki mogące mieć znaczenie praktyczne zaczęto uzyskiwać dopiero w drugiej połowie lat siedemdziesiątych. I wtedy jednak były to - mówiąc językiem przemysłowym - wyniki prototypowe, nie wsparte jeszcze ani dostatecznym doświadczeniem eksperymentalnym, ani tym bardziej żadnymi narzędziami wspomagania komputerowego. Nie ułatwiało to oczywiście ich adaptacji w praktyce. Do trudności tych dochodziły jeszcze poważne opory natury psychologicznej.

Długi okres oczekiwania na praktycznie użyteczne formalne metody projektowania software'u doprowadził w wielu środowiskach do przekonania, że tzw. zdrowy rozsądek, intuicja, doświadczenie i odpowiedni budżet na koszty testowania są jedyną rozsądną metodą zapewniającą powodzenie w praktyce.

Tymczasem ta sama praktyka stwarza nowe fakty. Burzliwy rozwój zastosowań informatyki prowadzi do oczywistej dziś konieczności radykalnego polepszenia niezawodności systemów software'owych. Związane jest to z dwoma czynnikami. Po pierwsze, współczesnym systemom powierza się coraz bardziej odpowiedzialne zadania, od których właściwego wykonania zależy nierzadko gospodarcza, społeczna czy polityczna równowaga świata. Po drugie, rozwój mikroinformatyki prowadzi do powstawania systemów, których kopie

sprzedawane są nie w tysiącach - jak to bywało dotychczas - ale w milionach egzemplarzy. Wszystko to, jak wskazuje McKeeman [12], doprowadziło w ostatnim dziesięcioleciu do dramatycznego wzrostu wysokości strat, jakie spowodować może błąd w oprogramowaniu. Z drugiej strony wiadomo już, że jakości systemów oprogramowania nie da się radykalnie poprawić metodami tradycyjnymi.

Opisana wyżej sytuacja jest oczywiście dobrze znana producentom oprogramowania na całym świecie. Stąd też szereg poważnych firm przemysłowych wykazuje coraz większe zainteresowanie możliwością wprowadzenia metod formalnych do stałego repertuaru narzędzi w swoich fabrykach software'u. Wymaga to oczywiście zorganizowania odpowiednio szeroko zakrojonej akcji szkolenia. Wśród dużych firm jako pierwsza zdecydowała się na taką inicjatywę IBM. Od szeregu już lat organizuje ona regularne kursy dla swoich pracowników w tym zakresie (por. C. Jones [11]). Niezależnie od tego wiedeńskie laboratorium IBM opracowało wspomnianą już wcześniej przemysłową wersję metody denotacyjnej zwaną VDM wraz z metajęzykiem specyfikacji software'u META-IV. Metody VDM nie należy mylić z opracowanym wcześniej w tym samym laboratorium i mniej udanym metajęzykiem VDL (Vienna Definition Language).

VDM wraz z metodami pokrewnymi oraz niezbędną podbudową teoretyczną wchodzą dziś do regularnych programów nauczania w wielu uniwersytetach na świecie. Dla przykładu, wykłada się je w Anglii na jedenastu uniwersytetach, w RFN na dziewięciu, we Francji i we Włoszech na czterech, w Danii i Szwecji na trzech, w Belgii na dwóch (dane pochodzą z roku 1983). Liczba uniwersytetów nauczających podstaw teoretycznych tych i podobnych metod formalnych jest jeszcze większa. Niezależnie od tego wiele firm przemysłowych prowadzi własne szkolenia i badania w tym zakresie. Obok wspomnianego już wcześniej IBM wymienić tu można ICL, Standard Telecommunication Laboratories Limited, AEG Telefunken, Siemens, Olivetti, Philips, VEB Robotron, Xerox, General Electric, Ford Aerospace, Honeywell, Bell Laboratories i wiele innych. Ostatnio rządy państw Wspólnoty Europejskiej postanowiły uruchomić duży projekt badawczo-wdrożeniowy, poświęcony rozwojowi i przemysłowym zastosowaniom formalnych metod projektowania i rozwoju software'u. Jeden z głównych kierunków prowadzonych tam prac dotyczyć będzie metody VDM.

3. Dokładniej o przyczynach, dla których metody formalne stosowane są w inżynierii oprogramowania

Wytworzenie dużego oprogramowania czy to systemowego czy użytkowego jest złożonym procesem przemysłowym konsumującym wiele dziesiątek osobo/lat pracy i prowadzącym do poważnych problemów technologicznych i organizacyjnych. Od systemów oprogramowania wymaga się, aby spełniały szereg wymogów konstrukcyjnych i użytkowych, wśród których do najważniejszych należą:

1) Zgodność z założoną specyfikacją zwana również niekiedy po prostu poprawnością.

2) Niezawodność obejmująca odporność systemu na błędy własne, środowiska, sprzętu i użytkownika.

3) Wydajność w sensie oszczędnego korzystania z zasobów takich, jak: czas, pamięć, urządzenia zewnętrzne itp.

4) Rozszerzalność, tzn. możliwość rozszerzenia systemu w przyszłości bez konieczności dokonywania zbyt wielu zmian w istniejącej części.

5) Przenośność, tzn. możliwość łatwej adaptacji systemu do różnych konfiguracji systemowo-sprzętowych.

6) Łatwość pielęgnacji, tzn. podatność systemu na wprowadzanie niewielkich zmian bez nadmiernego ryzyka wprowadzenia błędów.

7) Odzyskiwalność (ang. reusability), tzn. możliwość ponownego użycia modułów systemu w innych systemach lub w nowych generacjach tego samego systemu.

8) Odporność na włamanie, tzn. zabezpieczenie systemu przed niepowołanym użytkownikiem.

9) Zgodność ze standardami, np. dotyczącymi protokołów komunikacyjnych, reprezentacji informacji, szybkość reakcji itp.

10) Łatwość użycia, tzn. prostota koncepcyjna systemu, umożliwiająca szybkie jego opanowanie oraz pomoc udzielana użytkownikowi przez system w trakcie pracy.

11) Dobra dokumentacja będąca podstawą do dalszej obsługi systemu przez producenta (pielęgnacja, modyfikacja itp.), a także do napisania odpowiednich podręczników dla różnych grup użytkowników.

Wymienione wyżej wymagania dotyczące jakości systemu oprogramowania powinny znaleźć odzwierciedlenie i być realizowane we wszystkich etapach projektowania i implementacji systemu. Wymaga to posługiwania się we wszystkich tych etapach w miarę jednolitym i dostatecznie precyzyjnym językiem tak, aby przechodzenie z poziomu na poziom - od założeń poprzez wszystkie fazy projektowe do kodu wynikowego - gwarantowało zgodność treści pomiędzy poszczególnymi poziomami oraz możliwość formalnej weryfikacji tej zgodności. Konieczne jest również, aby opis każdego modułu systemu na każdym etapie jego projektowania i tworzenia był kompletny i jednoznaczny w ramach danego etapu.

Tradycyjna metoda wytwarzania software'u stosowana do dziś w wielu ośrodkach na świecie przyjmuje w zasadzie trzy poziomy językowe dla prac projektowych nad systemem:

1) całkowicie nieformalny poziom języka potocznego,

2) nieco bardziej formalny poziom różnego rodzaju schematów, np. sieci działań,

3) poziom języka programowania.

Poważną wadą tej metody jest całkowita intuicyjność poziomu pierwszego, niemal całkowita intuicyjność poziomu drugiego oraz brak systematycznych metod przechodzenia z poziomu na poziom oraz sprawdzenia poprawności tych przejść. Aby uprzytomnić sobie rozmiar związanych z tym problemów, zacytujmy za O. Herzogiem [10] dosyć typowy przykład rozwoju dokumentacji projektowej dużego systemu software'owego. System ten realizować miał mechanizm komunikacji pomiędzy niezależnie wykonującymi się procesami.

etap projektu	język opisu
założenia wstępne - 1 system - 61 stron opisu	j. potoczny
wstępna specyfikacja projektowa - 60 modułów - 589 stron opisu	j. potoczny
pełna specyfikacja projektowa - 544 funkcje - 2100 stron opisu	j. potoczny
modułowa specyfikacja logiczna - 4385 modułów - 12000 stron opisu	j. sieci działań
kod wynikowy - 3800000 linii kodu	j. programowania

Posługiwanie się językiem potocznym przy specyfikowaniu i projektowaniu software'u prowadzi z reguły do licznych błędów. Przytoczmy tu ponownie za Herzogiem [10] następującą statystykę typów błędów wykonaną dla dużej grupy projektów przemysłowych, prowadzonych metodą tradycyjną:

- błędy w projekcie 45 %
- błędy kodu 25 %
- błędy wprowadzane przy usuwaniu innych błędów 23 %
- pozostałe błędy 7 %

Zauważmy, że błędy trzeciej grupy zawdzięczają swoje istnienie w dużej mierze błędom grupy pierwszej. Tak więc udział błędów projektowych w ogólnym ich bilansie jest jeszcze większy niż to wynika bezpośrednio z przytoczonej statystyki.

Do podstawowych wad specyfikacji software'u pisanych w języku potocznym zaliczyć należy ich niejednoznaczność, niekompletność oraz niepoddawanie się ani testowaniu ani analizie logicznej, tzn. dowodzeniu poprawności. W tej sytuacji błędy powstające w w fazie projektu są wykrywalne w większości dopiero na etapie tes-

towania kodu. Prowadzi to do wysokich, związanych z tym kosztów, gdyż, jak to wynika z powyższej tabelki, koszt wykrycia i usunięcia błędu rośnie eksponencjalnie wraz z etapem, na którym błąd wykrywamy [10]:

etap	relatywny koszt wykrycia i usunięcia błędu
- projektowanie	1
- testowanie modułów	4
- testowanie funkcji systemu	9
- testowanie całego systemu	15
- eksploatacja	30

Te i liczne inne jeszcze problemy związane z wytwarzaniem i eksploatacją systemów oprogramowania skłaniają coraz więcej środowisk przemysłowych na świecie do stosowania metod formalnych w inżynierii oprogramowania. Oznacza to z jednej strony sięgnięcie do odpowiednich podstaw teoretycznych przy wyborze i opisie mechanizmów projektowanego systemu, a z drugiej strony użycie we wszystkich etapach projektowania języka formalnego w miejsce potocznego. Język taki przypomina zazwyczaj język programowania bardzo wysokiego poziomu - lub notację matematyczną - i ma z zasady precyzyjnie zdefiniowaną semantykę. Posługiwanie się językiem formalnym pozwala na precyzyjne i pełne definiowanie wszystkich elementów systemu na dowolnym etapie projektu. Pozwala również na sprawdzenie poprawności każdego etapu z osobna, a także sprawdzenie zgodności pomiędzy etapami. Oczywiście definicje formalne są tekstowo nie mniejsze od nieformalnych, zatem proces ich tworzenia i weryfikacji powinien być wspomagany komputerowo. Odpowiednie systemy wspomagania znajdują się już w fazie projektowo-prototypowej, nie będą jednak dostępne wcześniej jak za kilka lat. Okazuje się jednak, że nawet "ręczne" stosowanie metod formalnych pozwala na kilkakrotne (do dziesięciokrotnego!) [10] zmniejszenie ilości błędów w systemie. Wynika to stąd, że metody formalne posługują się precyzyjnym językiem opisu oraz narzucają określoną dyscyplinę pojęciową opartą na dobrze zdefiniowanym modelu matematycznym. Precyzyjny język i właściwie dobrany model pozwalają na uniknięcie wielu błędów typu niekompletność lub wewnętrzna sprzeczność opisu oraz

na zauważeniu wielu podobnych i innych błędów na długo przed przystąpieniem do etapu kodowania, testowania i eksploatacji.

4. Idea metody denotacyjnej i jej metajęzyk

Metoda denotacyjna powstała jako matematyczna technika definiowania semantyk języków programowania. Obecnie stosowana jest ona w znacznie szerszym kontekście i służy do definiowania (projektowania) wielu rodzajów oprogramowania systemowego i użytkowego takiego, jak kompilatory, systemy operacyjne, systemy obsługi baz danych, architektura sprzętu itd. W tym i następnym rozdziale omówimy pokrótce tę metodę w kontekście jej pierwotnych (ale nadal aktualnych!) zastosowań, jakimi były definicje języków programowania.

Definicję każdego języka programowania podzielić można na dwie podstawowe części: składnię, zwaną również syntaksą oraz semantykę. Definicja składni jest na ogół generacyjna, tzn. wyjaśnia jak z podstawowych elementów języka takich jak np.: zmienne i symbole operacji, budować elementy złożone takie jak wyrażenia, deklaracje, instrukcje itp. Definicję składni formułuje się najczęściej jako gramatykę bezkontekstową zapisaną w notacji BNF.

Zdefiniowanie semantyki jest problemem znacznie subtelniejszym i trudniejszym. Przede wszystkim należy wyjaśnić, co to jest semantyka języka programowania. W metodzie denotacyjnej przyjmuje się, że semantyka jest funkcją

$$S : L \rightarrow D$$

gdzie L oznacza język programowania traktowany jako zbiór programów, instrukcji, wyrażeń itp., natomiast D oznacza zbiór odpowiadających im znaczeń zwanych denotacjami. Denotacje definiowane są najczęściej jako funkcje operujące na stanach pewnej abstrakcyjnej maszyny. Denotacjami wyrażeń są funkcje przekształcające stany na wartości (np. boolowskie, całkowite itp.). Denotacjami instrukcji i deklaracji są funkcje przekształcające stany na stany. Funkcję semantyki definiuje się przez tzw. indukcję strukturalną. Oznacza to, że denotację każdego złożonego obiektu syntaktycznego opisuje się jako złożenie denotacji jego składowych. Pozwala to na bardzo czytelną strukturalizację definicji

języka i stanowi istotę metody denotacyjnej.

Aby posługiwać się metodą denotacyjną w praktyce dysponować należy odpowiednią do tego celu notacją. Przy jej ustalaniu trzeba wziąć pod uwagę, że typowe definicje denotacyjne liczą od kilkudziesięciu do kilkuset stron tekstu oraz operują tego samego rzędu liczbą pojęć i oznaczeń. Używana notacja powinna więc być możliwie przejrzysta, sugerować jednolitą metodę mnemotechnicznie czytelnych oznaczeń oraz narzucać pewne ogólne zasady dotyczące struktury definicji denotacyjnych. Powinna być również na tyle sformalizowana, aby poddawała się wspomaganiu odpowiednim systemem komputerowym. Mówiąc krótko, powinna przypominać język programowania strukturalnego wysokiego poziomu.

W obecnej chwili istnieje na świecie kilka notacji, lub jak niektórzy mówią metajęzyków, dostosowanych do metody denotacyjnej. Najpowszechniejszym z nich jest wspomniany już język META-IV [1], [2]. Ma on z kolei kilka dialektów. Poniżej przedstawiamy - w wielkim skrócie oczywiście - dialekt nazwany META [5]. Jego zastosowanie zilustrujemy na prostym przykładzie w następnym rozdziale.

Niech A i B oznaczają dowolne zbiory.

$A \mid B$	oznacza sumę teoriomnogościową A i B,
$A \times B$	oznacza iloczyn kartezjański A i B,
A^*	oznacza zb. wszystkich skończonych ciągów nad A wraz z ciągiem pustym,
$A \rightarrow B$	oznacza zb. wszystkich funkcji całkowitych z A do B,
$A \xrightarrow{\sim} B$	oznacza zb. wszystkich funkcji częściowych z A do B.

Przyjmujemy, że $\mid$, $\times$ oraz * wiążą mocniej niż $\rightarrow$ i $\xrightarrow{\sim}$ co pozwoli na opuszczanie wielu nawiasów w wyrażeniach definiujących zbiory.

Zbiory w semantyce denotacyjnej nazywamy tradycyjnie dziedzinami. Ponieważ dziedzin w typowej definicji denotacyjnej jest od kilkudziesięciu do kilkuset wprowadzamy dla nich zawsze nazwy mnemotechniczne oraz definiując dziedzinę wprowadzamy jednocześ-

nie oznaczenie dla jej elementów. To ostatnie jest najczęściej skrótem nazwy dziedziny. Na przykład dziedzinę stanów zdefiniować możemy formułą

$$sta : Stan = Identyfikator \xrightarrow{\sim} Wartość$$

którą odczytamy jak następuje: stany są funkcjami częściowymi ze zbioru identyfikatorów w zbiór wartości. Elementy zbioru Stan, tzn. stany, oznaczamy przez sta być może z indeksami.

Wprowadzimy teraz kilka oznaczeń przydatnych przy definiowaniu funkcji. Przez $f : A \to B$ oraz $f : A \xrightarrow{\sim} B$, zapisujemy fakt, że f jest funkcją całkowitą, lub odpowiednio częściową, z A do B. Jeżeli $f : A \xrightarrow{\sim} B$, $a \in A$ oraz $b \in B$ to

$$P[b/a] : A \xrightarrow{\sim} B$$

jest modyfikacją funkcji f taką, że

$$f[b/a](x) = \underline{\text{if}}\ x = a\ \underline{\text{then}}\ b\ \underline{\text{else}}\ f(x)$$

Tę notację uogólnia się w naturalny sposób na przypadek $f[b_1/a_1,\ldots,b_n/a_n]$, gdzie $a_1,\ldots,a_n$ muszą być wzajemnie różne.

Na użytek definicji przez przypadki wprowadzamy notację

$$b \to c,d$$

na oznaczenie $\underline{\text{if}}$ b $\underline{\text{then}}$ c $\underline{\text{else}}$ d gdzie b reprezentuje oczywiście wartość boolowską. Używamy również złożonych formuł warunkowych postaci

$$b_1 \to c_1,(b_2 \to c_2,\ldots,(b_n \to c_n,d)\ldots)$$

które najczęściej piszemy w kolumnie oddzielając jej wiersze przecinkami:

$$\begin{array}{ll} b_1 & \to c_1, \\ \ldots & \\ b_n & \to c_n, \\ \text{TRUE} & \to d \end{array}$$

Funkcje więcej niż jednego argumentu definiujemy często w tzw. postaci Curry'ego. Np. postacią Curry'ego funkcji $f : A \times B \times C \to B$ jest funkcja

$$f^{*} : A \to (B \to (C \to D)))$$

gdzie $((f^{*}(a))(b))(c) = f(a,b,c)$. Funkcje tej postaci są bardzo wygodne w definicjach denotacyjnych. Aby uniknąć nadmiaru nawiasów przy korzystaniu z takich funkcji, przyjmujemy następującą notację kropkową: $f^{*}.a$ oznacza $f^{*}(a)$ oraz $f^{*}.a.b.c$ oznacza $((f^{*}.a).b).c$. Zwróćmy uwagę, że jest to notacja podobna do przyjmowanej w wielu językach programowania (np. w Pascalu) dla rekordów.

Dla polepszenia przejrzystości definicji dziedzin funkcji w postaci Curry'ego opuszczamy w takich definicjach nawiasy o ile ich zagnieżdżenie rośnie od lewej do prawej. Np. $A \to B \to C \to D$ oznacza $A \to (B \to (C \to D))$. W definicjach takich dziedzin dopuszczamy również używanie operatora $\simeq$ na równi i wraz z operatorem $\to$. Np. możemy zdefiniować dziedzinę $A \to B \simeq C \to D$.

Ostatnia konwencja notacyjna dotyczy tzw. dziedzin syntaktycznych, czyli języków formalnych. Jeżeli A i B są takimi językami, to AB oznaczana ich konkatenację. Jeżeli $a \in A$ to będziemy również pisali aB zamiast {a}B oraz a|B zamiast {a}|B. Ta ostatnia konwencja dotyczy również dziedzin niesyntaktycznych, tzn. dowolnych zbiorów.

5. Prosty przykład definicji denotacyjnej języka programowania

Nasz przykład rozpoczniemy od zdefiniowania trzech dziedzin syntaktycznych uproszczonego języka programowania: ide(ntyfikatory), wyr(ażenia) oraz ins(trukcje).

```
ide : Ide = a|b|...
wyr : Wyr = Ide|not Wyr|(Wyr less Wyr)|
            (Wyr plus Wyr)
ins : Ins = read Ide | print Ide | Ide := Wyr |
            Ins; Ins                          |
            if Wyr then Ins else Ins fi       |
            while Wyr do Ins od
```

Definicje tych dziedzin są oczywiście indukcyjne. Formalnie mamy tu do czynienia z równaniami stałopunktowymi, ale czytelnik nie znający tej techniki może czytać powyższą definicję tak jak się czyta definicje napisane w BNF.

W dalszej kolejności definiujemy dziedziny typów danych dostępnych w naszym języku:

b : Bool = { prawda, fałsz } wartości boolowskie
c : Cał = { i | m ≤ i ≤ M } liczby całkowite

Oczywiście liczby całkowite należą do pewnego skończonego przedziału wyznaczonego przez rozmiar słowa w maszynie.

Dla pełnego zdefiniowania mechanizmu typów danych określić należy poza ich dziedzinami, również operacje, jakie na tych danych możemy wykonywać. Określamy trzy takie operacje:

not : Bool → Bool
less : Cał × Cał → Bool|NADMIAR
plus : Cał × Cał → Cał|NADMIAR

Ich definicje są następujące:

not.prawda = fałsz
not.fałsz = prawda

$$\text{less}.(c_1, c_2) =$$
$$c_1 - c_2 < m \quad \text{lub} \quad c_1 - c_2 > M \rightarrow \text{NADMIAR},$$
$$c_1 - c_2 < 0 \rightarrow \text{prawda},,$$
$$\text{TRUE} \rightarrow \text{fałsz}$$

$$\text{plus}.(c_1, c_2) =$$
$$m \leq (c_1 + c_2) \leq M \rightarrow c_1 + c_2,$$
$$\text{TRUE} \rightarrow \text{NADMIAR}$$

W definicjach tych wystąpił niezdefiniowany wcześniej element NADMIAR. Element ten reprezentuje sygnał błędu jaki powinien pojawić się w momencie, gdy zastosowanie operacji less lub plus prowadzi do powstania nadmiaru. Nazywamy go błędem abstrakcyjnym. Posługiwanie się błędami abstrakcyjnymi na poziomie semantyki denotacyjnej pozwala na opisanie mechanizmu wykrywania i sygnalizacji błędów przez implementację języka.

Po opisaniu mechanizmu typów danych przystępujemy do opisania tzw. typów wykonawczych. Definiujemy tu dziedziny i operacje odnoszące się do pojęcia stanu hipotetycznej maszyny widzianej z poziomu języka. W naszym przypadku wprowadzimy trzy dziedziny wykonawcze:

war : Wartość = Bool|Cał
stan : Stan = Ide|IN|OUT → Wartość|Wartość* | PUSTY
błąd : Błąd = NADMIAR|PUSTY|BŁAD-TYPU|PUSTE-WEJŚCIE

Zdefiniowane powyżej stany są funkcjami opisującymi zawartość pamięci maszyny oraz zawartości urządzeń wejścia i wyjścia. W uzupełnieniu powyższej definicji zakładamy, że IN, OUT ∉ Ide oraz, że każdy stan spełnia następujące trzy warunki, których koniunkcję nazywamy niezmiennikiem stanu:

stan.ide ∈ Wartość|PUSTY
stan.IN ∈ Wartość*
stan.OUT ∈ Wartość*

Intuicyjnie oznacza to, że każdy identyfikator języka jest nazwą komórki pamięci, która albo przechowuje jakąś wartość, albo jest pusta oraz że IN i OUT reprezentują bufory wejścia i wyjścia przechowujące ciągi (pliki) wartości. Zauważmy, że zarówno plik wejściowy jak i wyjściowy mogą być puste.

Mając zdefiniowane dziedziny wykonawcze języka możemy przystąpić do definiowania funkcji semantyki. Zwykle dla każdej kategorii syntaktycznej języka definiujemy odrębną funkcję semantyki. W naszym przypadku zdefiniujemy dwie takie funkcje:

W : Wyr → Stan → Wartość|Błąd
I : Ins → Stan|Błąd ≃→ Stan|Błąd

Funkcje semantyki są funkcjami całkowitymi przypisującymi wyrażeniom i instrukcjom ich denotacje. Denotacje wyrażeń są funkcjami całkowitymi, które biorą jako argument stan i oddają albo wartość albo odpowiedni sygnał błędu. Opisują one mechanizm wyliczania wyrażeń. Denotacje instrukcji są funkcjami częściowymi, które biorą jako argument stan lub sygnał błędu i oddają jako wynik również stan (na ogół zmieniony) lub sygnał błędu. Opisują

one mechanizm wykonywania instrukcji. Przyjmujemy dodatkowo, że ilekroć denotacja instrukcji otrzyma sygnał błędu jako argument, to odda ona ten sam sygnał błędu jako wynik. W ten sposób opisujemy mechanizm propagacji błędu. Fakt, że denotacje instrukcji są funkcjami częściowymi wynika stąd, że nasz język zawiera konstrukcję pętli (instrukcja while), a więc niektóre wykonania instrukcji mogą być nieskończone nie dając tym samym żadnego wyniku końcowego.

Możemy przystąpić teraz do zdefiniowania funkcji W oraz I. Jak mówiliśmy już o tym wcześniej, funkcje semantyk definiuje się metodą indukcji strukturalnej, tzn. przez przypadki odnoszące się do definicji składni języka

(W.1) Identyfikator jako wyrażenie

W[ide].stan = stan.ide

Wartością wyrażenia ide w aktualnym stanie jest wartość przechowywana pod tym identyfikatorem w tym stanie. Zauważmy, że może to być wartość PUSTY.

(W.2) Zanegowane wyrażenie boolowskie

W[not wyr].stan =
, let war = W[wyr].stan in
  war ∈ Błąd → war,
  war ∉ Bool → BŁĄD-TYPU,
  TRUE → not.war

Obliczając wartość wyrażenia not wyr najpierw obliczamy wartość podwyrażenia wyr. Jeżeli wartość ta jest błędem abstrakcyjnym, to staje się ona wartością całego wyrażenia. W przeciwnym przypadku sprawdzamy, czy jest to wartość boolowska. Jeżeli tak nie jest, to generujemy odpowiedni sygnał błędu. W przeciwnym przypadku wartością całego wyrażenia jest zanegowana wartość podwyrażenia wyr.

W powyższej definicji posłużyliśmy się nieopisaną jeszcze konstrukcją notacyjną let ... in. Jest to jedna z typowych konstrukcji w META-IV oraz w META. Pozwala ona na wprowa-

dzenie oznaczenia lokalnego dla danej definicji. Gdyby nasz metajęzyk był interpretowany maszynowo, konstrukcja ta byłaby deklaracją stałej lokalnej.

(W.3) Wyrażenie boolowskie "less"

$$
\begin{aligned}
&W[(wyr_1 \underline{less}\ wyr_2)].stan = \\
&\quad \underline{let}\ war_1 = W[wyr_1].stan\ \underline{in} \\
&\quad war_1 \in Błąd \rightarrow war_1, \\
&\quad war_1 \notin Cał \rightarrow BŁĄD\text{-}TYPU, \\
&\quad \underline{let}\ war_2 = W[wyr_2].stan\ \underline{in} \\
&\quad war_2 \in Błąd \rightarrow war_2, \\
&\quad war_2 \notin Cał \rightarrow BŁĄD\text{-}TYPU, \\
&\quad TRUE \rightarrow less.(war_1, war_2)
\end{aligned}
$$

Interpretacja intuicyjna tej definicji jest podobna do poprzedniej. Zauważmy, że w obu przypadkach odwołujemy się w definicji semantyki wyrażeń do funkcji zdefiniowanych uprzednio w dziedzinach danych. Przyjęliśmy przytem mnemotechniczną odpowiedniość pomiędzy symbolami języka not, less i plus oraz odpowiadającymi im metasymbolami not, less i plus.

(W.4) Wyrażenie arytmetyczne "plus"

$$
\begin{aligned}
&W[(wyr_1\ \underline{plus}\ wyr_2)].stan = \\
&\quad \ldots\ (\text{jak w (W.3)}) \\
&\quad TRUE \rightarrow plus.(war_1, war_2)
\end{aligned}
$$

Przechodzimy teraz do zdefiniowania semantyki instrukcji. Przypominamy, że w myśl przyjętego wcześniej założenia, dla każdej instrukcji $ins \in Ins$ oraz każdego błędu abstrakcyjnego $błąd \in Błąd$ zachodzi

$$I[ins].błąd = błąd$$

tzn., że każda instrukcja przekazuje odziedziczony od innej instrukcji sygnał błędu bez zmiany. Intuicyjnie oznacza to, że mechanizm obsługi błędów ograniczony jest w naszym języku do informowania o każdym błędzie użytkownika. Oczywiście na gruncie metody denotacyjnej opisać można również bardziej zaawansowane mechanizmy tego typu.

Podobnie jak funkcja semantyki dla wyrażeń również funkcja semantyki dla instrukcji zdefiniowana jest metodą indukcji strukturalnej.

(I.1) Instrukcja wejścia

$$
\begin{aligned}
&I[\underline{\text{read}}\ ide].stan = \\
&\quad \underline{\text{let}}\ (war_1,\ldots,war_n) = stan.IN\ \underline{\text{in}} \\
&\quad n > 0 \longrightarrow stan\ war_1/ide,(war_2,\ldots,war_n)/IN \\
&\quad TRUE \longrightarrow PUSTE\text{-}WEJŚCIE
\end{aligned}
$$

Najpierw sprawdzamy, czy plik wejściowy jest niepusty. Jeżeli tak jest, to pierwszy element z pliku wejściowego przypisujemy identyfikatorowi ide, a następnie usuwamy ten element z pliku. W przeciwnym przypadku generujemy odpowiedni sygnał błędu.

(I.2) Instrukcja wyjścia

$$
\begin{aligned}
&I[\underline{\text{print}}\ wyr].stan = \\
&\quad \underline{\text{let}}\ war = W[wyr].stan\ \underline{\text{in}} \\
&\quad war \in Błąd \longrightarrow war, \\
&\quad \underline{\text{let}}(war_1,\ldots,war_n) = stan.OUT\ \underline{\text{in}} \\
&\quad TRUE \longrightarrow stan\ (war_1,\ldots,war_n,war)/OUT
\end{aligned}
$$

Najpierw obliczamy wartość wyrażenia wyr. Jeżeli ta wartość jest sygnałem błędu, to pozwalamy temu sygnałowi propagować dalej. W przeciwnym przypadku umieszczamy tę wartość na końcu pliku wyjściowego.

(I.3) Instrukcja przypisania

$$
\begin{aligned}
&I[ide := wyr].stan = \\
&\quad \underline{\text{let}}\ war = W[war].stan\ \underline{\text{in}} \\
&\quad war \in Błąd \longrightarrow war, \\
&\quad TRUE \longrightarrow stan\ war/ide
\end{aligned}
$$

Wyliczamy wartość wyrażenia wyr i jeżeli jest to sygnał błędu to pozwalamy mu propagować. W przeciwnym przypadku przypisujemy tę wartość identyfikatorowi ide.

(I.4) Sekwencyjne złożenie instrukcji

$$I[ins_1; ins_2].stan = I[ins_2].(I[ins_1].stan)$$

Najpierw wykonujemy ins_1 a potem ins_2.

(I.5) Instrukcja warunkowa

$$\begin{aligned} &I[\underline{if}\ wyr\ \underline{then}\ ins_1\ \underline{else}\ ins_2\ \underline{fi}].stan = \\ &\quad \underline{let}\ war = W[wyr].stan\ \underline{in} \\ &\quad war \in Błąd \rightarrow war, \\ &\quad war \notin Bool \rightarrow BŁĄD\text{-}TYPU, \\ &\quad war = prawda \rightarrow I[ins_1].stan, \\ &\quad TRUE \rightarrow I[ins_2].stan \end{aligned}$$

Najpiew obliczamy wartość wyrażenia wyr i jeżeli prowadzi to do sygnału błędu, to pozwalamy mu propagować dalej. W przeciwnym przypadku sprawdzamy, czy jest to wartość boolowska. Jeżeli tak nie jest, to generujemy odpowiedni sygnał błędu. Jeżeli tak jest, to wykonujemy odpowiednią gałąź naszej instrukcji warunkowej.

(I.6) Instrukcja pętli

$$\begin{aligned} &I[\underline{while}\ wyr\ \underline{do}\ ins\ \underline{od}].stan = \\ &\quad \underline{let}\ war = W[wyr].stan\ \underline{in} \\ &\quad war \in Błąd \rightarrow war, \\ &\quad war \notin Bool \rightarrow BŁĄD\text{-}TYPU, \\ &\quad war = prawda \rightarrow I[\underline{while}\ wyr\ \underline{do}\ ins\ \underline{od}].(I[ins].stan), \\ &\quad TRUE \rightarrow stan \end{aligned}$$

Najpierw obliczamy wartość wyrażenia wyr i jeżeli jest to sygnał błędu, to pozwalamy mu propagować dalej. W przeciwnym przypadku sprawdzamy, czy jest to wartość boolowska i jeżeli tak nie jest, to generujemy sygnał błędu. Jeżeli jest to wartość boolowska, to mamy jeszcze dwie możliwości. Jeżeli wartością tą jest "prawda", to wykonujemy instrukcję ins będącą ciałem pętli, a następnie przesyłamy stan wynikowy ponownie do instrukcji pętli. Jeżeli wartością tą jest "fałsz", to stan wejściowy instrukcji pętli przesyłamy dalej niezmieniony. Zauważmy, że przypadek war = prawda prowadzi pozornie do błędnego koła w definicji. W rzeczywistości

jednak definicja nasza jest poprawna, co wynika z teorii punktu stałego w zbiorach częściowo uporządkowanych. Oczywiście rozmiar niniejszego opracowania nie pozwala na pobieżne chociażby wyjaśnienie podstaw tej teorii. Zainteresowany czytelnik znajdzie odnośniki do odpowiedniej literatury w rozdz. 7.

6. Kilka słów o zastosowaniach metody denotacyjnej

Jak już wspomniano w rozdz. 2 denotacyjna metoda definiowania i projektowania software'u stosowana jest w projektach przemysłowych od kilku już lat, przy czym zarówno zakres jej zastosowań, jak i liczba użytkowników stale rośnie. Dotyczy to szczególnie jej przemysłowej wersji VDM. Wśród ciekawszych zastosowań tej metody (dane z roku 1982) przytoczyć można definicje formalne następujących języków i systemów:

Basic	Prolog
Fortran	Ada
Algol 60	CHILL
Pascal	PL/1
Pascal Plus	System/R
Modula 2	CODASYL/DBTG

Niektóre z tych definicji - np. dla Ady i CHILL - stały się punktem wyjścia do zbudowania już istniejących kompilatorów. Inne - np. dla System/R (IBM-owski system baz danych) - pozwoliły na wykrycie i usunięcie wielu błędów w istniejącej implementacji.

Definicje języków i systemów tworzone metodą VDM reprezentują oczywiście dosyć spore obiekty tekstowe. Dla zorientowania czytelnika o ich rozmiarach podajemy poniższą tabelkę zawierającą kilka typowych przykładów [4]:

język	liczba linii definicji napisanej w META-IV
Algol-60	1000
Pascal	2500
PL/1	4000
Pascal Plus	4500
Ada	14000

Oczywiście przy tworzeniu takich tekstów i posługiwaniu się nimi na szerszą skalę konieczne są komputerowe systemy wspomagające. W kilku ośrodkach na świecie, m.in. na Uniwersytecie w Manchester i na Politechnice w Kopenhadze, prowadzone są już prace zmierzające do zbudowania takich systemów. Mają to być bogate środowiska komputerowe wspomagające pracę projektanta, analityka i programisty systemowego. Przewiduje się, że będą one miały następującą strukturę:

1. Edytor źródłowy
   - 1.1. Sterowany składnią edytor metajęzyka, np. META-IV
   - 1.2. Analizator syntaktycznej pełności i niesprzeczności definicji
   - 1.3. Analizator poprawności definicji ze względu na użyte w niej meta-typy meta-obiektów
2. Meta-interpreter
   - 2.1. Interpreter metajęzyka pozwalający na eksperymentalną eksploatację zdefiniowanego systemu
   - 2.2. Zestaw narzędzi pozwalający na przeprowadzenie dynamicznej analizy systemu, tzn. analizy przepływu danych, testowania itp.
3. Procesor transformacji
   - 3.1. Interaktywny system wspomagający transformowanie definicji źródłowych w metajęzyku na definicje innego poziomu, ale też w metajęzyku, np. transformowanie definicji języka na definicję kompilatora tegoż języka
   - 3.2. Translatory z metajęzyka na poziom języków programowania
4. Weryfikator poprawności
   - 4.1. Systemy wspomagające dowodzenie poprawności projektowanego software'u ze względu na różne kryteria
   - 4.2. Systemy wspomagające dowodzenie zgodności pomiędzy kolejnymi etapami projektu.

Brak, w obecnej chwili, systemów komputerowego wspomagania metody denotacyjnej nie powinien prowadzić do biernego oczekiwania na takie systemy. Aby stosować metodę denotacyjną w przyszłości należy zacząć uczyć się jej już dziś. Na metodę tę składają

się bowiem - poza metajęzykiem - określona teoria matematyczna, metodologia pracy, a nawet filozofia. Opanowanie teorii, przyzwyczajenie się do nowej metodologii, zaakceptowanie nowej filozofii, wymaga czasu i nakładu pracy. Wymaga śledzenia literatury przedmiotu oraz jednoczesnego ręcznego eksperymentowania z metodą. Poświęcony na to czas nie będzie z pewnością stracony. Zauważmy przecież, że wszystkie dotychczasowe zastosowania tej metody - również kompilatory dla języków Ada i CHILL - wykonane zostały całkowicie bez pomocy systemów wspomagania.

7. Uwagi bibliograficzne

Czytelnik, którego niniejszy artykuł zdołał zainteresować metodą denotacyjną zechce zapewne zapytać o dostępną literaturę przedmiotu. Niestety, odpowiedź na to pytanie nie jest zbyt zachęcająca. W literaturze światowej, a tym bardziej krajowej, brak jest dobrego podręcznika zastosowań metody denotacyjnej. Na pocieszenie i zachętę można dodać, że co najmniej dwa takie podręczniki są w trakcie powstawania. Ukażą się one jednak nie wcześniej niż za dwa lata. Na razie więc należy ograniczyć się do tego co jest dostępne.

Zacznijmy od literatury w języku polskim. Podstawową pozycją jest tu książka M. Gordona [9] (tłumaczenie z j. angielskiego) wyjaśniająca zasady stosowania semantyki denotacyjnej. Zaletą tej książki jest jej wielka jasność i przystępność. Wadą - że przedstawia ona tzw. model kontynuacyjny, który ze względu na swoją złożoność nigdy nie był stosowany w praktyce. Warto może jednak wspomnieć, że grupa studentów Uniwersytetu Warszawskiego po wysłuchaniu jednosemestralnego wykładu na podstawie właśnie tej książki napisała z własnej inicjatywy pełną definicję denotacyjną dla Pascala. Do polskiego wydania książki Gordona dołączony jest dodatek napisany przez autora niniejszego artykułu, a poświęcony podstawowej dla semantyki denotacyjnej teorii zbiorów częściowo uporządkowanych i dziedzin Scotta.

Czytelnika, który chciałby uzyskać szersze spojrzenie na metodę denotacyjną w kontekście innych modeli dla matematycznych semantyk języków programowania oraz innych formalnych metod inżynierii oprogramowania odesłać można do książki P. Dembińskiego i

J. Małuszyńskiego [7]. Osoby zainteresowane formalnymi metodami konstruowania i analizy poprawnościowej programów użytkowych polecić można przekład książki C. Jones'a [11], która ukaże się już niebawem nakładem Wydawnictw Naukowo-Technicznych w serii Biblioteka Inżynierii Oprogramowania. Czytelnikom, którzy czują potrzebę podniesienia swoich kwalifikacji ogólno-matematycznych w zakresie przydatnym do lepszego zrozumienia metody denotacyjnej polecić można książkę H. Rasiowej [13].

W literaturze obcojęzycznej (tzn. anglosaskiej) najlepszą pozycją przedstawiającą teorię metody denotacyjnej jest książka J. Stoy'a [14]. Nie jest to jednak, z pewnością, książka dla praktyków. Przedstawia ona, podobnie jak książka M. Gordona, model kontynuacyjny, a ponadto posługuje się bardzo niepraktyczną notacją. Jest to książka napisana wyraźnie dla teoretyków. Niemniej jednak z niej właśnie oraz z jeszcze trudniejszych tekstów korzystali twórcy VDM budując swoją metodę i próbując jej pierwszych zastosowań.

W zakresie VDM najpełniejszym dziś i najaktualniejszym tekstem jest zbiór raportów technicznych [1] wydany i częściowo napisany przez dwóch współtwórców i pionierów tej metody: D. Bjørnera i C. Jones'a. Niestety, jest on w kraju trudno dostępny. Tekstem mniej aktualnym i mniej czytelnym, ale dostępniejszym, jest wydany przez tych samych autorów zbiór [2]. Matematyczne podstawy VDM - będące w zasadzie skrótem matematycznych podstaw dla semantyki denotacyjnej - przedstawione są w artykule J. Stoy'a [15]. Ukazał się on w materiałach konferencji poświęconej podstawom semantyki denotacyjnej, które w całości polecić można jako dobrą uzupełniającą literaturę przedmiotu. Aspekty inżynierskie VDM opisano w pracy [4] D. Bjørnera i S. Prehna, natomiast najobszerniejszy przykład zastosowań - denotacyjna semantyka dla języka ADA - przedstawiona jest w zbiorze [3] wydajnym przez D. Bjørnera i O. Oesta.

Klasyczna semantyka denotacyjna - w tym również i VDM - oparta jest na dosyć trudnym matematycznie modelu tzw. dziedzin zwrotnych stworzonym przez D. Scotta. Ostatnio A. Blikle i A. Tarlecki [6] zaproponowali zastąpienie tego modelu przez prostszy, oparty

na elementarnej teorii mnogości. Wstępna propozycja modyfikacji języka META-IV przystosowana do tego nowego modelu opisana została w pracy [5].

BIBLIOGRAFIA

[1] Bjørner, D., Jones, C.B. (eds.) The Vienna Development Method: The Meta Language, Lecture Notes in CS, vol. 61, Springer-Verlag 1978

[2] Bjørner, D. Jones, C.B. Formal Specification and Software Development, Prentice-Hall International 1982

[3] Bjørner, D., Oest, O.N. (eds). Towards a Formal Description of ADA, LNCS 98, Springer-Verlag 1980

[4] Bjørner, D., Prehn, S. Software Engineering Aspects of VDM; the Vienna Development Method, in: Theory and Practice of Software Technology (D. Ferrari, M. Bolognani, J. Goguen, eds). North-Holland 1983

[5] Blikle, A. A metalanguage for naive denotational semantics, Progetto Finalizzato Informatica, C.N.R. Progetto P_1Cnet 104, Piza 1983

[6] Blikle, A., Tarlecki, A. Naive Denotational Semantics, in: Information Processing 83 (R.E.A. Manson ed.), Proc. IFIP 1983, North-Holland

[7] Dembiński, P., Małuszyński, J., Matematyczne metody definiowania języków programowania, WNT, Warszawa 1981

[8] Floyd, R.W., Assigning Meanings to Programs, in: Mathematical Aspects of Computer Science (J.T. Schwartz, ed.), Proc. of a Symp. in Appl. Math., American Mathematical Society Providence 1967

[9] Gordon, M.J.C., Denotacyjny opis języków programowania, WNT, Warszawa 1983

[10] Herzog, O. Formal Software Development Methods in Industrial Environments, in: Formal Software Development: Combining Specification Methods, proc. workshop, Nyborg 1984

[11] Jones, D. Software Development, Rigorous Approach, Prentice Hall, Englewood Cliffs 1979

[12] McKeeman, W.M., The Role of Software Engineering in the Microcomputer Revolution: An Overview, Proc. 4th Int. Conf. on Soft. Eng., Munich 1979, IEEE Cat. no 79CH1479-5C

[13] Rasiowa, H., Wstęp do matematyki współczesnej, PWN, Warszawa 1973

[14] Stoy, J.E., Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory, The MIT Press, Cambridge Mass. 1977

[15] Stoy, J.E., Foundations of Denotational Semantics, in: Abstract Software Specification, 1979 Copenhagen Winter School Proc. (D. Bjørner, ed.), Lecture Notes in CS, vol. 86, Springer-Verlag 1980

[16] Turing, A.M. On Checking a Large Routine, Report of a Conference on High Speed Automatic Calculating Machines, pp. 67-69, University Mathematical Laboratory, Cambridge 1949

Jesienna Szkoła PTI
Rydzyna, październik 1984

SYSTEMY WIELOMIKROPROCESOROWE

doc. Wojciech Cellary
Instytut Automatyki
Politechnika Poznańska
ul. Piotrowo 3A
60-965 Poznań, tel.782370

1. Wstęp

W rozwoju systemów komputerowych szczególnie wyraźnie uzewnętrznia się naturalne skądinąd dążenie do konstrukcji systemów o coraz większej efektywności działania. Wzrost ten osiągany jest dwoma sposobami. Pierwszym sposobem jest doskonalenie technologii układów składających się na system komputerowy w szczególności na jego procesor. Warto tu uzmysłowić sobie odbytą w ciągu czterdziestu lat drogę od lamp elektronowych do układów ULSI.

Drugim sposobem jest doskonalenie architektury systemów komputerowych, które polega w swej istocie na zwiększaniu stopnia równoległości operacji realizowanych przez system. Można mówić o kilku poziomach tak rozumianego doskonalenia architektury systemu. Pierwszym z nich jest poziom architektury wewnętrznej procesora. Przykładami doskonalenia architektury systemu na tym poziomie jest wprowadzenie zakładkowania pozwalającego na zrównoleglenie fazy wykonywania jednej mikroinstrukcji z fazą pobierania następnej oraz wprowadzenie arytmometru z przewidywaniem przeniesienia, pozwalającego na równoległe wykonywa-

nie operacji arytmetycznych na wszystkich bitach słowa.

Drugim poziomem doskonalenia architektury systemu jest poziom klasycznego komputera. Przykładem z tego poziomu jest wprowadzenie autonomicznych kanałów wejścia/wyjścia umożliwiających równoleglenie działania procesora centralnego z operacjami wejścia/wyjścia. Szczególną rolą w doskonaleniu architektury systemu na tym poziomie odegrało wprowadzenie przerwań jako środka informowania procesora centralnego o zajściu zdarzeń asynchronicznych w stosunku do jego działania, w szczególności o zakończeniu wykonywania równoległych operacji wejścia/wyjścia.

Wreszcie trzecim, najwyższym poziomem jest poziom systemu. Doskonalenie architektury systemu na tym poziomie odbywa się przez wprowadzanie równolegle działających i komunikujących się między sobą procesorów centralnych. Nie zakładamy przy tym, że procesory te muszą być identyczne.

Architektura wieloprocesorowa systemów komputerowych ma oprócz wspomnianej możliwości uzyskania wzrostu efektywności systemu jeszcze dwie dalsze potencjalne zalety. Pierwszą z nich jest możliwość podwyższenia niezawodności systemu. Uzyskuje się ją na przykład na drodze równoległej realizacji tych samych funkcji systemu przez kilka procesorów z okresowym porównywaniem wyników, na drodze testowania i diagnostyki jednych procesorów (ściślej biorąc całych modułów) przez drugie, na drodze przejmowania funkcji uszkodzonych procesorów przez inne sprawne itp.

Drugą potencjalną zaletą jest możliwość modularyzacji systemu zgodnie z wymaganiem danego zastosowania i uzyskanie dzięki temu dużej elastyczności systemu, co ma szczególne znaczenie w przypadku jego modyfikacji i rozwoju.

Zalety wieloprocesorowej architektury systemów komputerowych dostrzeżone zostały już na dość wczesnym etapie rozwoju tych systemów. Począwszy od początku lat siedemdziesiątych rozpoczęto budowę eksperymentalnych systemów o takiej architekturze. Pierwszymi były systemy C.mmp (computer. multi-mini-processor), którego budowę rozpoczęto w 1971 roku oraz Pluribus, którego budowę rozpoczęto w 1972 roku. W systemach tych wykorzystywano technologię, a nawet gotowe moduły minikomputerowe.

Jednakże dopiero rozwój mikroprocesorów, a ściśle rzecz

biorąc rozwój mikroprocesorów trzeciej generacji - szesnastobitowych stworzył techniczne i ekonomiczne warunki do szerokiego stosowania systemów o architekturze wieloprocesorowej. Mówimy w tym przypadku o systemach wielomikroprocesorowych. Dla uściślenia pojęć dodajmy, że mówiąc w tym referacie o systemach wielomikroprocesorowych mamy na myśli systemy wielomikroprocesorowe silnie powiązane (ang. tightly (closely) coupled microcomputer systems), to jest takie, w których komunikacja między mikroprocesorami odbywa się przez wspólną pamięć operacyjną. W przeciwieństwie do systemów silnie powiązanych istnieją jeszcze systemy luźno powiązane (ang. loosely coupled microprocessor systems), w których komunikacja między mikroprocesorami odbywa się poprzez porty wejścia/wyjścia.

Celem tego referatu jest wprowadzenie do problematyki systemów wielomikroprocesorowych. Z tego względu w rozdziale 2 omówimy pokrótce architekturę i zasadę działania systemów jednomikroprocesorowych, tak aby na ich tle ukazać w następnych rozdziałach problematykę systemów wielomikroprocesorowych.

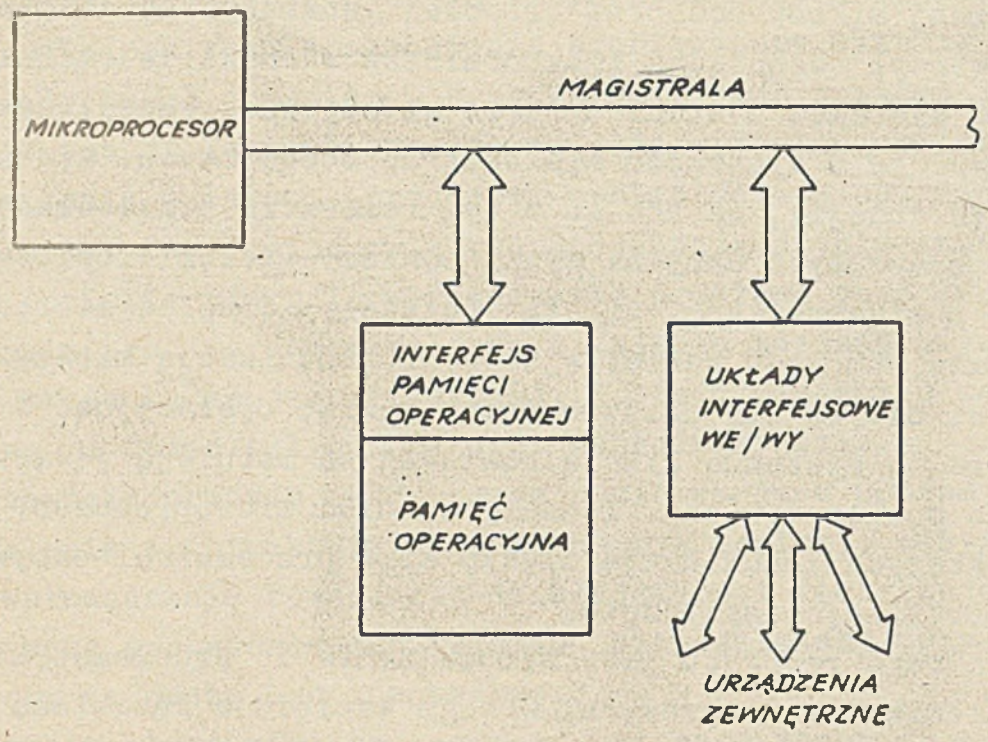
W rozdziale 3 przedstawimy architekturę wewnętrzną mikroprocesora trzeciej generacji Intel 8086, ze szczególnym uwzględnieniem tych jego cech, które przystosowują go do pracy w systemie wielomikroprocesorowym.

W rozdziałach 4 i 5 omówimy architekturę systemów wielomikroprocesorowych z dzieloną magistralą lokalną oraz systemową.

W rozdziale 6 przedstawimy pewne uwagi końcowe.

2. Architektura systemów jednomikroprocesorowych

Najbardziej schematycznie architektura systemów jednomikroprocesorowych przedstawiona została na rys.2.1. W systemie takim można wyróżnić trzy główne części składowe: mikroprocesor, pamięć operacyjną oraz układy interfejsowe we/wy. Części te komunikują się między sobą poprzez magistralę, będącą zbiorem odpowiednich linii sygnałowych. Z funkcjonalnego punktu widzenia magistrala systemu mikrokomputerowego składa się z magistrali sterującej, magistrali adresowej i magistrali danych. Zasadą działania systemu wielomikrokomputerowego jest synchroniczna



Rys.2.1. Architektura systemów jednomikroprocesorowych

praca na magistrali. Oznacza to, że sygnały na magistrali mogą pojawić się wyłącznie w ściśle określonych momentach czasu wyznaczonych przez tak zwany cykl zegarowy. Przykładowo pobranie zawartości komórki pamięci operacyjnej przez mikroprocesor przebiega następująco: mikroprocesor wysyła adres żądanej komórki na magistralę adresową, odczekuje odpowiednią liczbę cykli zegarowych i czyta "w ciemno" z magistrali danych. Zadaniem pamięci operacyjnej jest wysłanie, po zdekodowaniu adresu, zawartości zaadresowanej komórki na magistralę danych w odpowiednim momencie i na odpowiednio długi odcinek czasu. Jeśli pamięć operacyjna jest zbyt wolna w stosunku do szybkości pracy mikroprocesora i nie może wysłać danej w odpowiednio krótkim czasie, to informuje o tym fakcie mikroprocesor wysyłając odpowiedni sygnał (sygnał WAIT) przez magistralę sterującą. Sygnał ten powoduje, że mikroprocesor wydłuża okres oczekiwania na daną o dodatkowy cykl zegarowy i w tak określonym momencie czyta ją z magistrali danych. Jak widać, wymiana informacji między częściami składowymi systemu mikrokomputerowego odbywa się w ściśle określonych momentach czasu i bez wymiany potwierdzeń.

Opisany powyżej cykl pracy mikroprocesora nazywany jest cyklem maszynowym. Cykl ten związany jest ze zmianą adresu przesyłanego przez magistralę adresową. Cykle maszynowe nie są w ogólności jednakowej długości, choć zawsze są wielokrotnością cyklu zegarowego.

Oczywiście, cykl maszynowy zapisu do pamięci operacyjnej jest analogiczny do opisanego powyżej cyklu maszynowego odczytu z pamięci. Mikroprocesor wysyła adres komórki na magistralę adresową oraz następnie daną na magistralę danych. Po upływie określonego czasu kończy cykl maszynowy zdejmując sygnały, a więc zakładając, że dana została przyjęta przez pamięć operacyjną. Podobnie jak poprzednio pamięć może żądać wydłużenia cyklu maszynowego. Jedyna różnica pomiędzy cyklem maszynowym odczytu i zapisu polega na odpowiednim sygnale kierunku transmisji przesyłanym magistralą sterującą.

Również komunikacja między mikroprocesorem a układami interfejsowymi we/wy odbywa się w postaci analogicznych cykli maszynowych. Jeśli adres takiego układu zawarty jest w przestrzeni adresowej pamięci to przebieg cykli jest identyczny, gdyż jest on traktowany tak jak komórki pamięci operacyjnej. Jeśli adres układu interfejsowego jest zawarty w przestrzeni adresowej we/wy, to różnica polega na odpowiednim sygnale na magistrali sterującej, informującym o fakcie adresowania w tej przestrzeni.

Także komunikacja między układami interfejsowymi we/wy a pamięcią operacyjną przebiega w postaci podobnych cykli maszynowych. Komunikacja taka ma miejsce w przypadku transmisji przez kanał DMA - kanał bezpośredniego dostępu do pamięci operacyjnej. Na czas transmisji kanał DMA przejmuje sterowanie magistralą, wymuszając przejście mikroprocesora w stan wysokiej impedancji. Przejście to odbywa się po zakończeniu aktualnego cyklu maszynowego mikroprocesora.

Na zakończenie omawiania cykli pracy magistrali systemu mikrokomputerowego wspomnijmy o tak zwanym cyklu rozkazowym obejmującym wykonanie pełnego rozkazu mikroprocesora. Cykl rozkazowy składa się z pewnej liczby cykli maszynowych, zależnej od formatu rozkazu.

Przejdziemy obecnie do krótkiego scharakteryzowania poszczególnych części składowych systemu mikroprocesorowego. Na

wstępie należy zauważyć, że architektura systemu przedstawiona na rysunku 2.1 jest architekturą logiczną. Architektura fizyczna rozumiana jako podział funkcji systemu pomiędzy poszczególne układy mikroprocesorowe, jest zazwyczaj inna. Można tu mówić o dwóch przypadkach. Pierwszym z nich jest połączenie kilku części składowych architektury logicznej systemu w jednym układzie scalonym. Ekstremalnym przykładem w tym zakresie są mikrokomputery monolityczne, np. serii Intel 8048, zawierające wszystkie części składowe w jednym układzie scalonym. Innym przykładem są układy rodziny mikroprocesora Intel 8085, np. 8155 lub 8755 zawierające pamięć operacyjną i wybrane porty we/wy.

Drugim przypadkiem jest realizacja funkcji jednej części składowej architektury logicznej systemu przez wiele układów scalonych. Omówimy to w odniesieniu do poszczególnych części.

Mikroprocesor rzadko występuje w systemie mikrokomputerowym bez pewnych układów towarzyszących. Takimi układami towarzyszącymi są dla mikroprocesora Intel 8080 układ zegarowy 8224 i sterownik magistrali sterującej 8228, dla mikroprocesora Intel 8085 rejestr zatrzaskowy np. 8212, dla mikroprocesorów Intel 8088 i Intel 8086 układ zegarowy 8284 oraz sterowniki magistral: adresowej (rejestry zatrzaskowe) 8282 lub 8283, danych 8286 lub 8287 i sterującej 8288. Istnienie tych układów towarzyszących wynika z faktu, że z technologicznego punktu widzenia jednym z najbardziej ograniczonych, a przez to najcenniejszych, zasobów mikroprocesora jako układu scalonego są jego wyprowadzenia. Ponieważ sygnałów jest więcej niż nóżek układu scalonego więc do jednej nóżki należy przypisać kilka sygnałów. Istnieje kilka rozwiązań w tym względzie. Pierwsze polega na przetwarzaniu kilku sygnałów z magistrali sterującej i wypracowywaniu przez układ towarzyszący jednego sygnału wejściowego dla mikroprocesora. Rolę taką spełniają układy zegarowe. Drugie rozwiązanie polega na wysyłaniu niektórych sygnałów sterujących w postaci zakodowanej i rozkodowywaniu ich w układzie towarzyszącym. Rolę taką spełniają sterowniki magistrali sterującej. Trzecie rozwiązanie polega na multipleksacji sygnałów, czyli wykorzystywaniu jednej nóżki mikroprocesora do przesyłania różnych sygnałów rozdzielonych w czasie. Najczęściej stosowana jest multipleksacja adresów i danych. Rozwiązanie to wymaga stosowania rejestrów zatrzaskowych

dla przechowania i udostępnienia na magistrali na odpowiednio dłuższy odcinek czasu sygnału pojawiającego się w pierwszej kolejności. Wreszcie czwarte rozwiązanie polega na sprzętowym programowaniu (przez podanie stałego sygnału na określoną nóżkę mikroprocesora) zestawu sygnałów mikroprocesora. Rozwiązanie takie jest stosowane na przykład w mikroprocesorach Intel 8088 i Intel 8086 do określania trybu pracy mikroprocesora.

Pamięć operacyjna systemów mikroprocesorowych dzieli się na pamięć wyłącznie odczytywalną - typu EPROM oraz pamięć zapisywalną - typu RAM. Istnienie pamięci EPROM w systemie jest niezbędne ze względu na to, że pamięć RAM wykonana w obecnej technologii traci swą zawartość po wyłączeniu zasilania. Pamięć EPROM jest więc konieczna dla poprawnego restartu systemu.

Interfejs pamięci operacyjnej stanowi dekoder adresów oraz rejestry buforowe. W niektórych przypadkach stosuje się również rejestry segmentowe pozwalające na rozszerzenie przestrzeni adresowej mikroprocesora. Warto wspomnieć, że w przypadku stosowania pamięci dynamicznej RAM układami towarzyszącymi tej pamięci są układy odświeżające.

Do układów interfejsowych we/wy zaliczamy wszystkie te układy mikroprocesorowe, które pośredniczą w komunikacji mikroprocesora z urządzeniami zewnętrznymi. Należą do nich: port we/wy szeregowego 8251, port we/wy równoległego 8255, układy programowalnych liczników/zegarów 8253 i 8254, kanał DMA 8257, kontroler przerwań 8259, sterowniki pamięci na dyskach elastycznych 8271 i 8272, sterownik monitora ekranowego 8275 i inne.

System mikroprocesorowy - pierwszy 1971 EMP

3. Architektura wewnętrzna mikroprocesora Intel 8086

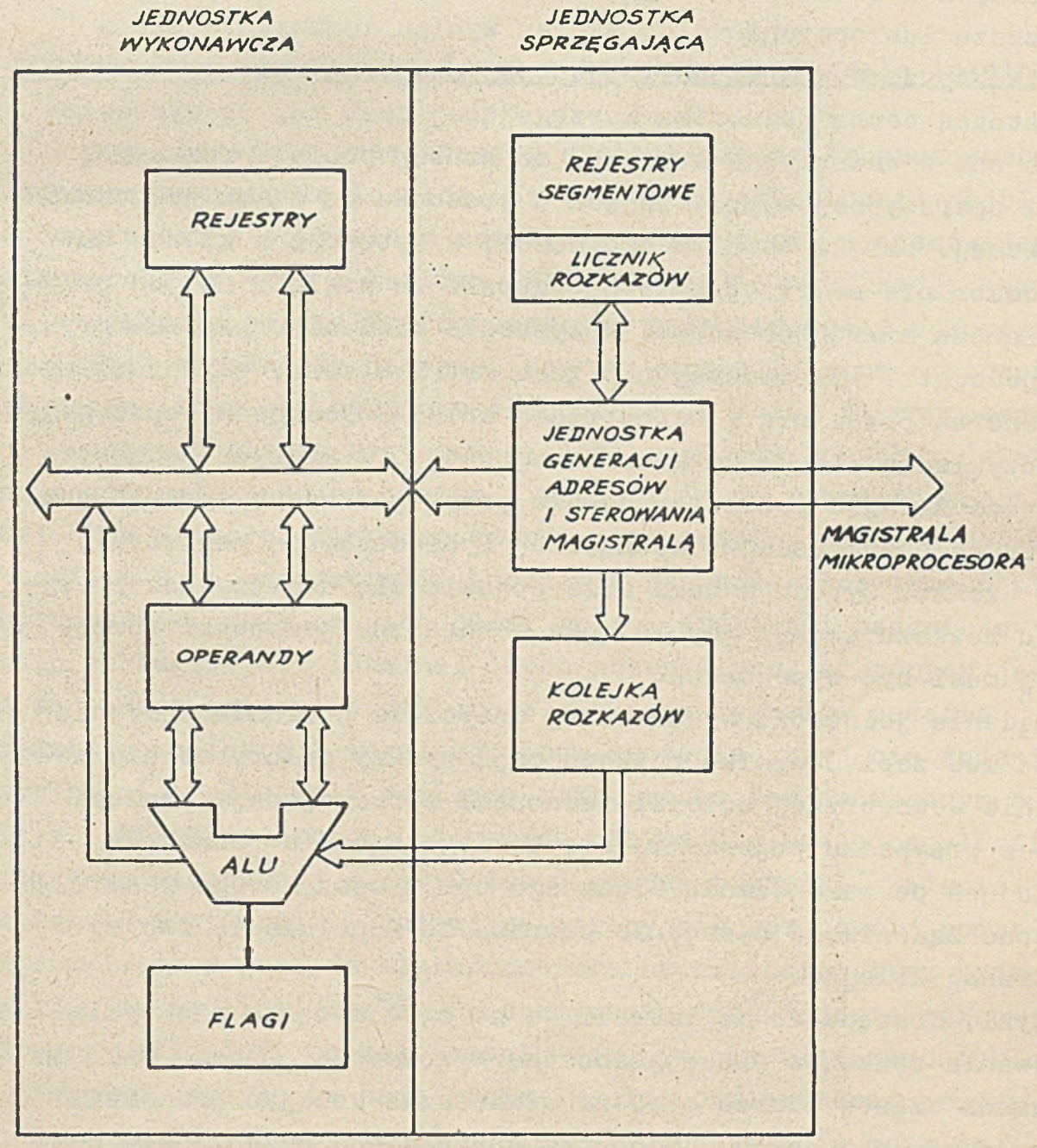
Problemy architektury systemów wielomikroprocesorowych muszą być rozważane na dwóch poziomach: na poziomie architektury wewnętrznej mikroprocesorów oraz na poziomie architektury systemu. W tym rozdziale przedstawimy wewnętrzną architekturę mikroprocesora zaprojektowanego z myślą o pracy w systemach wielomikroprocesorowych na przykładzie mikroprocesora szesnastobitowego Intel 8086. Wybór tego mikroprocesora nie jest przypadkowy, gdyż ze wszystkich mikroprocesorów szesnastobitowych ma on naj-

większe szanse szerszego rozpowszechnienia w Polsce w najbliższej przyszłości.

W celu przedstawienia problemów architektury wewnętrznej mikroprocesorów przystosowanych do pracy w systemach wielomikroprocesorowych powróćmy najpierw do przypadku systemów jednomikroprocesorowych z mikroprocesorem drugiej generacji, np. Intel 8080, wyposażonych w kanał DMA, wspomnianego w rozdziale 2. Z teoretycznego punktu widzenia system taki może być uznany za ograniczony system wielomikroprocesorowy. Kanał DMA można bowiem uznać za specjalizowany mikroprocesor, z tego względu, że może on samodzielnie ubiegać się o dostęp do magistrali. Rozważmy jednak problem równoległości działania mikroprocesora i kanału DMA w takim systemie. Jak wspomnieliśmy w rozdziale 2 uaktywnienie kanału DMA powoduje zawieszenie działania mikroprocesora; praca jednego układu wyklucza więc pracę drugiego - równoległość ich działania jest zerowa. Wniosek taki jest sprzeczny z podstawowym celem budowy systemów wielomikroprocesorowych omówionym w rozdziale 1, jakim jest zwiększenie efektywności systemu przez równoległą realizację jego funkcji. Może powstać w tym miejscu pytanie po co wobec powyższego stosuje się kanały DMA w systemach jednomikroprocesorowych? Aby nie pozostawić tego pytania bez odpowiedzi wyjaśnijmy, że transmisja do/z pamięci operacyjnej za pomocą kanału DMA jest wielokrotnie szybsza niż analogiczna transmisja za pomocą mikroprocesora. Wynika to z faktu, że w każdym cyklu maszynowym kanału DMA przesyłane są do/z pamięci operacyjnej transmitowane dane. Natomiast w przypadku transmisji za pomocą mikroprocesora na jeden cykl maszynowy przeznaczony na przesłanie transmitowanych danych przypada wiele cykli związanych z wykonywaniem programu. Ponadto, jeśli transmisja przez kanał DMA nie jest na tyle szybka aby wykorzystywać wszystkie kolejne cykle maszynowe magistrali, to kanał DMA zwalnia magistralę, umożliwiając wznowienie pracy mikroprocesorowi na przykład na jeden cykl. W horyzoncie całej transmisji możliwa jest zatem pewna równoległość pracy mikroprocesora i kanału DMA.

Powyższe wyjaśnienia nie zmieniają faktu, że gdyby na podobnej zasadzie połączyć w systemie dwa mikroprocesory drugiej generacji, to równoległość ich pracy byłaby minimalna. Z tego

względu, dla mikroprocesora trzeciej generacji jakim jest Intel 8086 opracowano inną architekturę wewnętrzną. Przedstawiono ją na rysunku 3.1.



Rys.3.1. Architektura wewnętrzna mikroprocesora Intel 8086

Mikroprocesor Intel 8086 zbudowany jest z dwóch jednostek, które pracują asynchronicznie względem siebie. Pierwszą jednostką jest jednostka wykonawcza (ang. Execution Unit), drugą -

jednostka sprzęgająca z magistralą (ang. Bus Interface Unit). Jednostka sprzęgająca realizuje wszystkie dostępy mikroprocesora do magistrali. Jednostka wykonawcza realizuje rozkazy mikroprocesora, a w razie potrzeby dostępu do magistrali w celu pobrania rozkazu lub operandu albo zapisu wyniku komunikuje się z jednostką sprzęgającą sygnalizując jej swoje żądanie.

Istotną cechą jednostki sprzęgającej jest to, że wyposażona jest ona w wewnętrzną kolejkę, do której pobiera rozkazy z pamięci operacyjnej wyprzedzająco w stosunku do żądań jednostki wykonawczej. Dzięki temu, w ogólności, w sytuacji w której mikroprocesor nie ma dostępu do magistrali ze względu na jej przydział innemu mikroprocesorowi w systemie wielomikroprocesorowym, praca jednostki wykonawczej nie jest zatrzymywana, gdyż rozkazy pobierane są przez nią z wewnętrznej kolejki jednostki sprzęgającej. Oczywiście nie są wykluczone sytuacje, w których jednostka wykonawcza żądając pobrania rozkazu przez jednostkę sprzęgającą natrafia na pustą kolejkę wewnętrzną i niedostępną magistralę i musi czekać. Sytuacja taka może pojawić się na przykład po wykonaniu rozkazu skoku, gdy kolejka wewnętrzna jednostki sprzęgającej musi być wyzerowana.

Żądania jednostki wykonawczej dotyczące pobrania operandu rozkazu lub zapisu wyniku rozkazu mają wyższy priorytet niż napełnianie wewnętrznej kolejki jednostki sprzęgającej. Oznacza to, że w przypadku pojawienia się takiego żądania jednostka sprzęgająca po zakończeniu bieżącego cyklu maszynowego realizuje zgłoszone żądanie, a następnie powraca do napełniania swojej wewnętrznej kolejki.

Cykle dostępu do magistrali są na ogół krótsze niż cykle wykonywania rozkazów przez jednostkę wykonawczą. Z tego względu stosunkowo często zdarza się, że wewnętrzna kolejka jednostki sprzęgającej jest pełna. Jednostka sprzęgająca odłącza się wówczas od magistrali zwalniając ją dla innych mikroprocesorów. Ponowne dołączenie się przez nią do magistrali może nastąpić w dowolnym cyklu zegarowym, bez potrzeby dodatkowej synchronizacji.

Reasumując należy stwierdzić, że wewnętrzny podział mikroprocesora na dwie asynchroniczne jednostki: wykonawczą i sprzęgającą, z możliwością buforowania pobieranych wyprzedzająco

rozkazów bardzo istotnie redukuje wpływ oczekiwania na dostęp do magistrali na efektywność pracy mikroprocesora, umożliwiając tym samym budowę efektywnych systemów wielomikroprocesorowych.

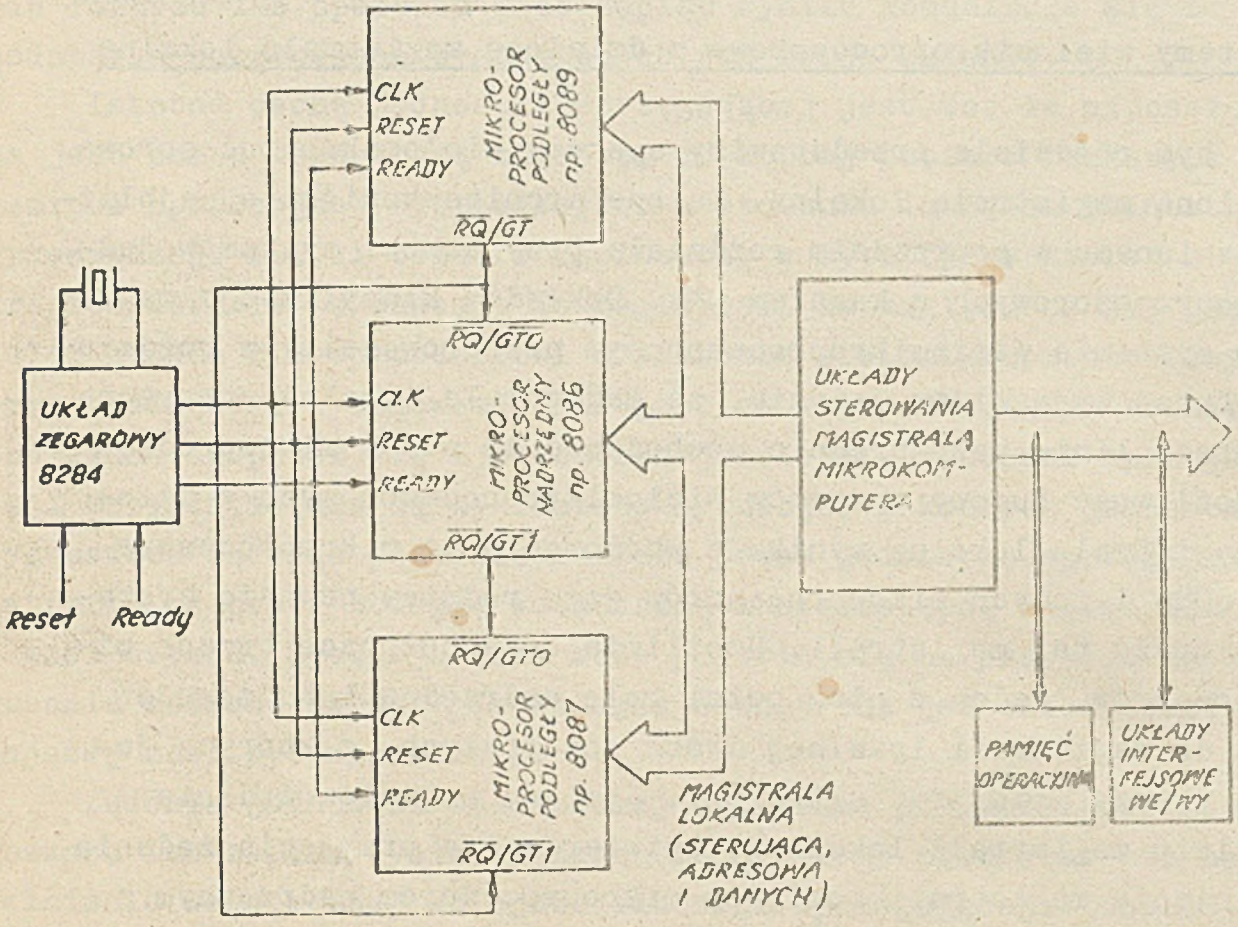
4. Systemy wielomikroprocesorowe z dzieloną magistralą lokalną

W tym rozdziale przedstawimy systemy wielomikroprocesorowe z dzieloną magistralą lokalną. Są one architektonicznie najbliższe omawianemu w poprzednim rozdziale przypadkowi systemów jednomikroprocesorowych z kanałem DMA. Dokładną klasyfikację magistral w systemie wielomikroprocesorowym przeprowadzimy w rozdziale 5. Tutaj wyjaśnijmy jedynie, że magistrala lokalna utworzona jest przez linie bezpośrednio dochodzące do nóżek mikroprocesora.

Możliwość budowy systemów wielomikroprocesorowych z dzieloną magistralą lokalną wynika z wbudowania do mikroprocesora Intel 8086 i innych mikroprocesorów jego rodziny pewnego systemu arbitrażu tej magistrali. Umożliwia on współpracę trzech mikroprocesorów. Jeden z nich pełni rolę nadrzędną i nadzoruje dostęp do magistrali lokalnej dwóch pozostałych, z których jeden ma wyższy priorytet niż drugi. Wymiana sygnałów żądania przydziału magistrali lokalnej, potwierdzenia przyjęcia żądania i zwolnieniu magistrali pomiędzy mikroprocesorem nadrzędnym a jednym z mikroprocesorów podległych odbywa się na jednej linii sygnałowej oznaczonej $\overline{RQ}/GT$ 0 lub 1 (request/grant). Żądanie przydziału magistrali lokalnej ze strony mikroprocesora podległego zgłaszane jest do mikroprocesora nadrzędnego przez podanie impulsu na linii $\overline{RQ}/GT$. Po otrzymaniu tego żądania mikroprocesor nadrzędny, po zakończeniu wykonywania bieżącego cyklu maszynowego, wysyła tą samą linią impuls potwierdzenia przyjęcia żądania i odłącza się od magistrali. Oczywiście może on kontynuować swoje działanie pobierając rozkazy ze swojej wewnętrznej kolejki rozkazów. Mikroprocesor podległy po zrealizowaniu dostępu do magistrali odłącza się od niej i informuje o tym fakcie mikroprocesor nadrzędny przesyłając impuls linią $\overline{RQ}/GT$.

Architektura systemu wielomikroprocesorowego z dzieloną magistralą lokalną przedstawiona jest na rysunku 4.1. Warto

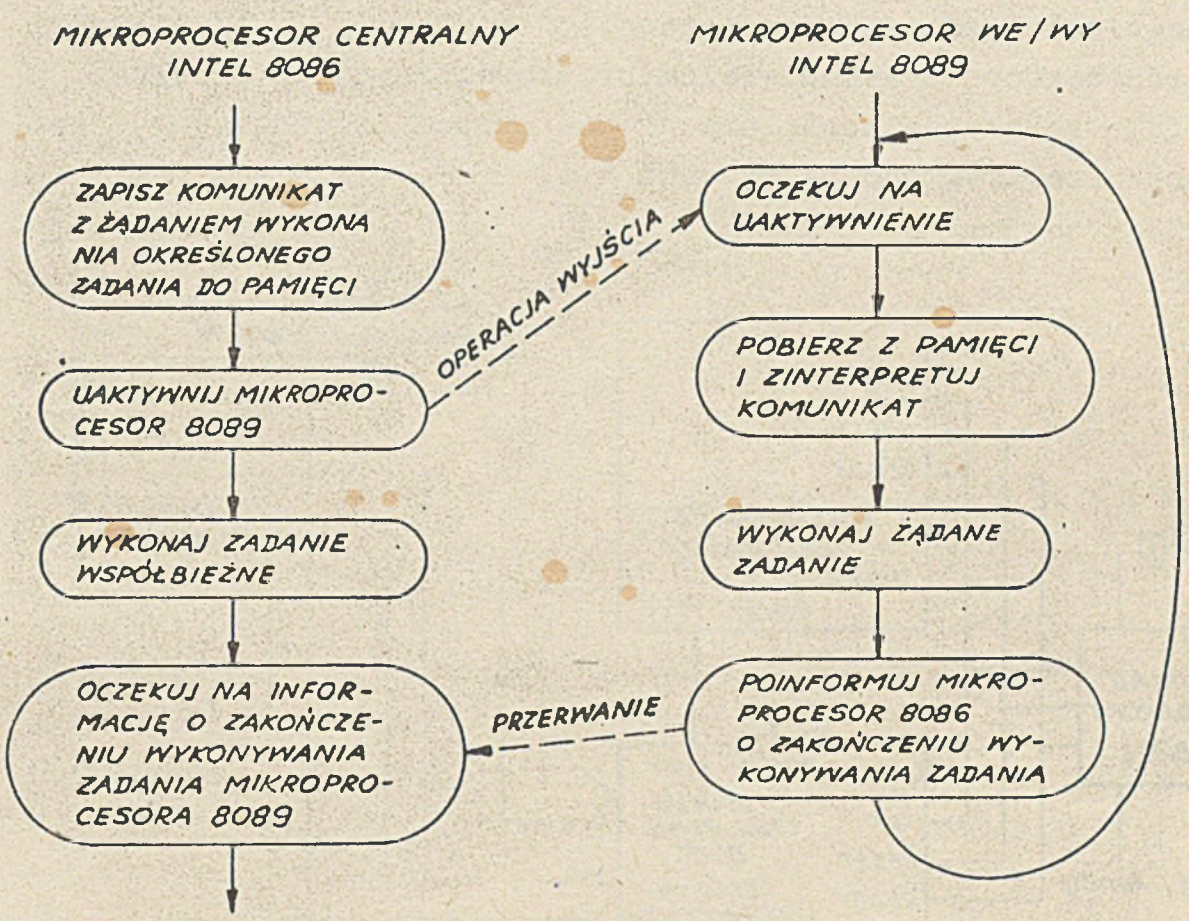
zauważyć, że ten typ organizacji systemu wielomikroprocesorowego wymaga synchronizacji wszystkich mikroprocesorów. Stąd w systemie znajduje się jeden wspólny układ zegarowy 8284.



Rys.4.1. Architektura systemu wielomikroprocesorowego z dzieloną magistralą lokalną.

Istotną zaletą systemów wielomikroprocesorowych z dzieloną magistralą lokalną jest to, że nie wymagają one stosowania specjalnych dodatkowych układów towarzyszących. Odpowiednie mechanizmy wbudowane są w same mikroprocesory, choć należy zwrócić uwagę, że mechanizmy te nie są wystarczające do połączenia w systemie kilku mikroprocesorów centralnych Intel 8086. Głównym przeznaczeniem omawianej architektury jest łączenie mikroprocesorów specjalizowanych. Przykładową konfiguracją tego typu jest połączenie mikroprocesora centralnego Intel 8086 (lub Intel 8088) ze specjalizowanym mikroprocesorem we/wy Intel 8089. Ten ostatni łączy w sobie cechy mikroprocesorów centralnych, a więc zdolność

do wykonywania programów, z cechami kanałów DMA, a więc ze zdolnością do szybkiej, ciągłej transmisji danych. Synchronizacja na poziomie zadań wykonywanych przez dwa mikroprocesory przedstawiona jest na rysunku 4.2.



Rys.4.2. Synchronizacja mikroprocesorów: centralnego i we/wy na poziomie zadań.

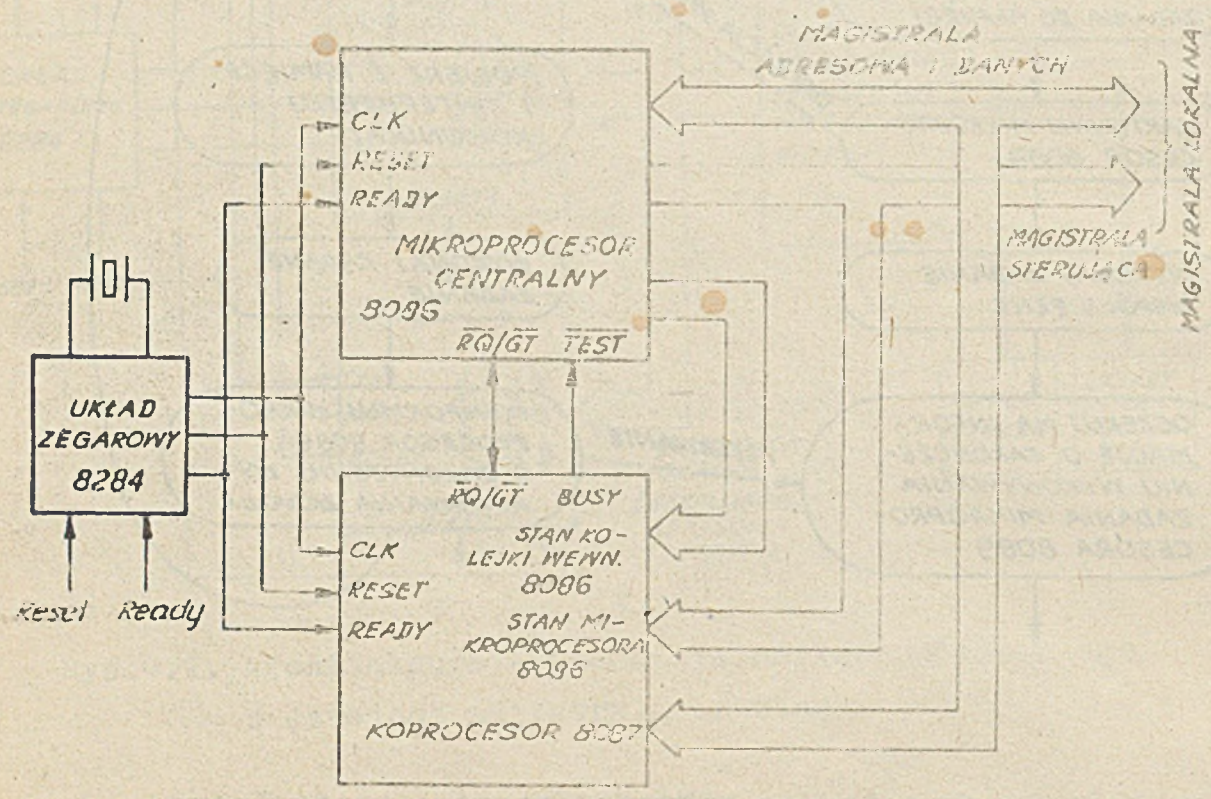
Jak widać, synchronizacja ta odbywa się z wykorzystaniem operacji we/wy, wspólnej pamięci operacyjnej i przerwań.

Inną typową konfiguracją systemu wielomikroprocesorowego z dzieloną magistralą lokalną jest połączenie mikroprocesora centralnego Intel 8086 (lub Intel 8088) z koprocesorem, np. koprocesorem arytmetycznym Intel 8087.

Koprocesor jest nowym typem mikroprocesora, który nie jest mikroprocesorem centralnym. Nie wykonuje on własnego programu i

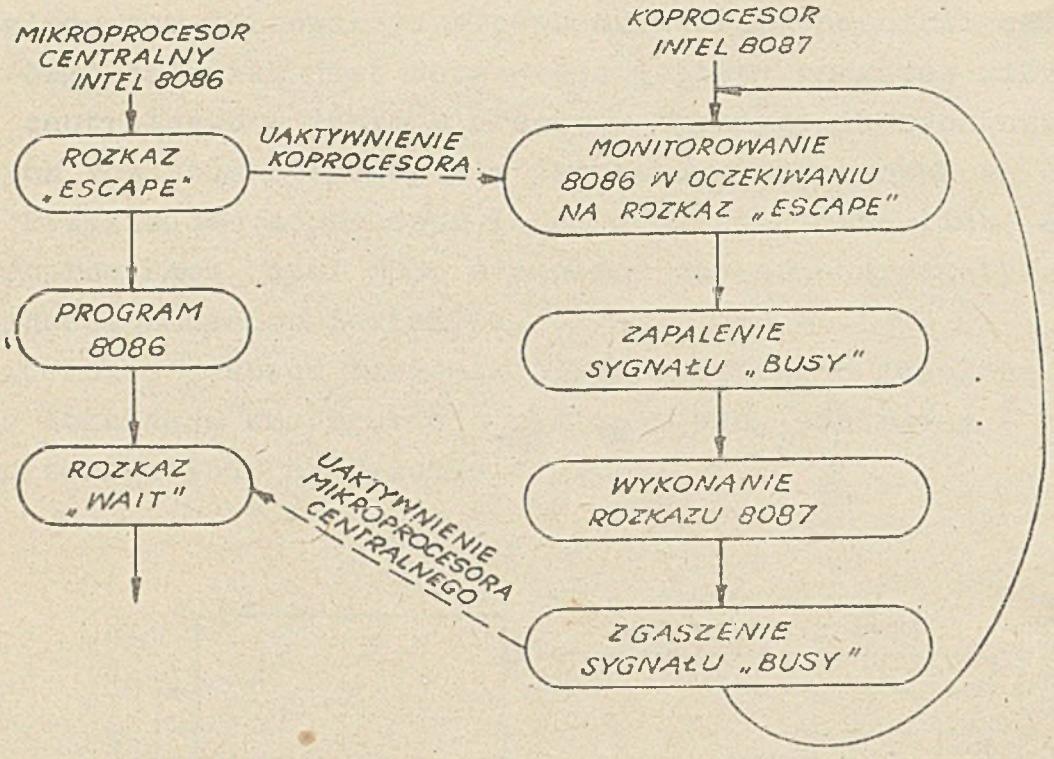
z tego względu zawsze towarzyszy pewnemu mikroprocesorowi centralnemu. Jego zadaniem jest rozszerzenie listy rozkazów mikroprocesora centralnego w danym kierunku i wykonywanie tych uzupełniających rozkazów równolegle z programem mikroprocesora centralnego. Przykładowo koprocesor Intel 8087 rozszerza listę rozkazów mikroprocesora Intel 8086 o złożone operacje arytmetyczne, w szczególności zmiennoprzecinkowe.

Architekturę systemu wielomikroprocesorowego z dzieloną magistralą lokalną w konfiguracji mikroprocesór centralny - koprocesor przedstawiono na rysunku 4.3.



Rys.4.3. System wielomikroprocesorowy z mikroprocesorem centralnym i koprocesorem.

Współpraca koprocesora z mikroprocesorem centralnym polega na monitorowaniu rozkazów pobieranych przez mikroprocesor centralny Intel 8086 w oczekiwaniu na rozkazy przeznaczone dla niego. Rozkazy te to 64 rozkazy ESCAPE. Koprocesor rozpoznaje rozkazy ESCAPE (w szczególności odróżnia je od danych) na podstawie badania stanu mikroprocesora centralnego. Ponadto koprocesor bada



Rys.4.4. Synchronizacja mikroprocesora centralnego i koprocesora na poziomie rozkazów.

Synchronizacja w kierunku od mikroprocesora centralnego do koprocesora została powyżej omówiona. Pozostaje więc do omówienia synchronizacja w drugą stronę. Jej potrzeba zachodzi wówczas gdy w momencie, w którym mikroprocesor centralny musi skorzystać z wyników równolegle wykonywanej przez koprocesor operacji nie ma pewności, czy operacja ta już się zakończyła.

Do synchronizacji tej służy rozkaz WAIT wykonywany przez mikroprocesor centralny. Działanie tego rozkazu jest uzależnione od stanu wejścia $\overline{\text{TEST}}$ mikroprocesora, na które pódaje się normalnie sygnał wyjściowy BUSY koprocesora. Sygnał ten ma wartość logiczną 1 od momentu zdekodowania rozkazu ESCAPE przez koprocesor do momentu zakończenia wykonywania rozkazu. Jeśli sygnał $\overline{\text{TEST}}$ przyjmuje wartość 0 to działanie rozkazu WAIT jest równoważne działaniu rozkazu "nic nie rób". Natomiast jeśli sygnał $\overline{\text{TEST}}$ przyjmuje wartość 1 to rozkaz WAIT powoduje zawieszenie działania mikroprocesora centralnego aż do zmiany stanu tego sygnału.

Powracając do problemów architektury systemów wielomikroprocesorowych z dzieloną magistralą lokalną należy stwierdzić,

że korzystną i zalecaną konfiguracją jest współdzielenie magistrali lokalnej przez trzy mikroprocesory: centralny 8086, we/wy 8089 i koprocesor 8087. (por. rysunek 4.1). Wówczas, ze względu na to, że mikroprocesor we/wy prowadzi szybkie transmisje danych uwarunkowane czasowo, nadaje mu się wyższy priorytet dołączając do linii $\overline{RQ}/\overline{GT}0$ mikroprocesora centralnego. W przypadku jednoczesnego zgłoszenia żądania przydziału magistrali lokalnej przez mikroprocesor we/wy i koprocesor spełnione zostanie w pierwszej kolejności żądanie mikroprocesora we/wy. Jeśli jednak magistrala lokalna jest przydzielona do koprocesora w momencie zgłoszenia żądania przez mikroprocesor we/wy, to musi on czekać na zwolnienie jej. Aby minimalizować to oczekiwanie linię $\overline{RQ}/\overline{GT}$ mikroprocesora we/wy łączy się dodatkowo z linią $\overline{RQ}/\overline{GT}1$ koprocesora (por. rysunek 4.1). Wówczas, na mocy opisanego uprzednio mechanizmu działania sygnałów $\overline{RQ}/\overline{GT}$ w mikroprocesorze centralnym, koprocesor zwalnia magistralę po otrzymaniu żądania od mikroprocesora we/wy natychmiast po zakończeniu bieżącego cyklu maszynowego, nawet jeśli potrzebuje dalszych cykli do dokończenia wykonywania swego rozkazu.

Inną zalecaną konfiguracją mikroprocesorów w omawianej architekturze jest połączenie dwóch mikroprocesorów we/wy w celu zwiększenia liczby kanałów transmisyjnych. W tym przypadku jeden z mikroprocesorów programuje się jako nadrzędny, a drugi podległy. Pozostałe mechanizmy ich wzajemnej współpracy pozostają bez zmian.

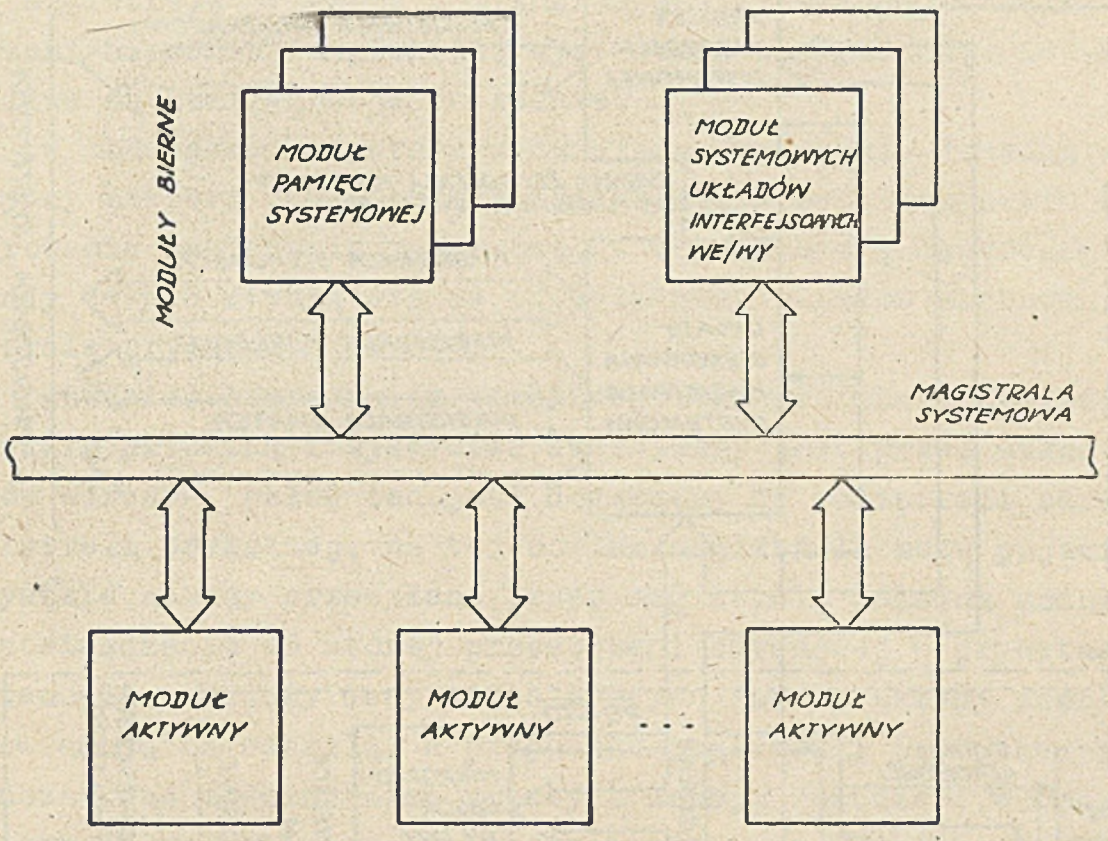
5. Systemy wielomikroprocesorowe z dzieloną magistralą systemową

Podstawową zaletą przedstawionej w poprzednim rozdziale architektury systemów wielomikroprocesorowych z dzieloną magistralą lokalną była możliwość budowy takich systemów bez specjalnych układów towarzyszących, przy wykorzystaniu wbudowanych w mikroprocesory mechanizmów. Architekturę tę cechują jednak pewne istotne ograniczenia. Przede wszystkim jest ona przewidziana do łączenia mikroprocesorów specjalizowanych. Przy wykorzystaniu wyłączenie mechanizmów wbudowanych w mikroprocesory

rodziny Intel 8086 nie można połączyć w takim systemie kilku mikroprocesorów centralnych. Oczywiście, dobudowując odpowiednią logikę na zewnątrz mikroprocesorów można by skonstruować system wielomikroprocesorowy z dzieloną magistralą lokalną, w którym połączonych byłoby kilka mikroprocesorów centralnych. System taki byłby jednak mało efektywny z tego względu, że każdy dostęp do sprzętowych zasobów systemu - pamięci operacyjnej i układów interfejsowych we/wy (por. rysunek 2.1), które są wspólne byłby potencjalnie konfliktowy. Drogą do zminimalizowania liczby konfliktów i tym samym do podniesienia efektywności systemu jest rozdział zasobów systemu na prywatne (dedykowane) i systemowe (współdzielone). Zasoby prywatne to takie, do których dostęp ma wyłącznie jeden mikroprocesor centralny systemu. Natomiast zasoby systemowe to takie, do których dostęp mają wszystkie mikroprocesory. Oczywiście rozdział zasobów systemu na prywatne i systemowe wymaga rozdziału magistrali lokalnej odpowiednio na magistralę prywatną i systemową. Rozdział ten jest dokonany w przestrzeni adresowej mikroprocesorów. Po rozdziale zasobów na prywatne i systemowe jedynie dostęp do zasobów systemowych przez magistralę systemową jest potencjalnie konfliktowy.

Zauważmy, że podział zasobów na prywatne i systemowe jest naturalny dla systemów wielomikroprocesorowych. Poszczególne mikroprocesory wykonują bowiem zazwyczaj autonomiczne, równoległe zadania wykorzystując swoje zasoby prywatne i stosunkowo rzadko komunikują się między sobą, co wymaga wykorzystania zasobów systemowych.

Systemy wielomikroprocesorowe o rozdzielonej w przestrzeni adresowej magistrali lokalnej w sposób opisany powyżej nazywamy systemami z dzieloną (w czasie) magistralą systemową. Ogólną architekturę takiego systemu przedstawiono na rysunku 5.1. W systemie takim wyróżnia się moduły aktywne oraz bierne dołączone do magistrali systemowej. Moduły aktywne w przeciwieństwie do modułów biernych, są to takie moduły, które mogą samodzielnie ubiegać się o dostęp do magistrali systemowej. Należy podkreślić, że wyróżnikiem modułów aktywnych jest ich zdolność do ubiegania się o dostęp do magistrali systemowej, a nie fakt wyposażenia w mikroprocesor, który skądinąd jest konieczny do

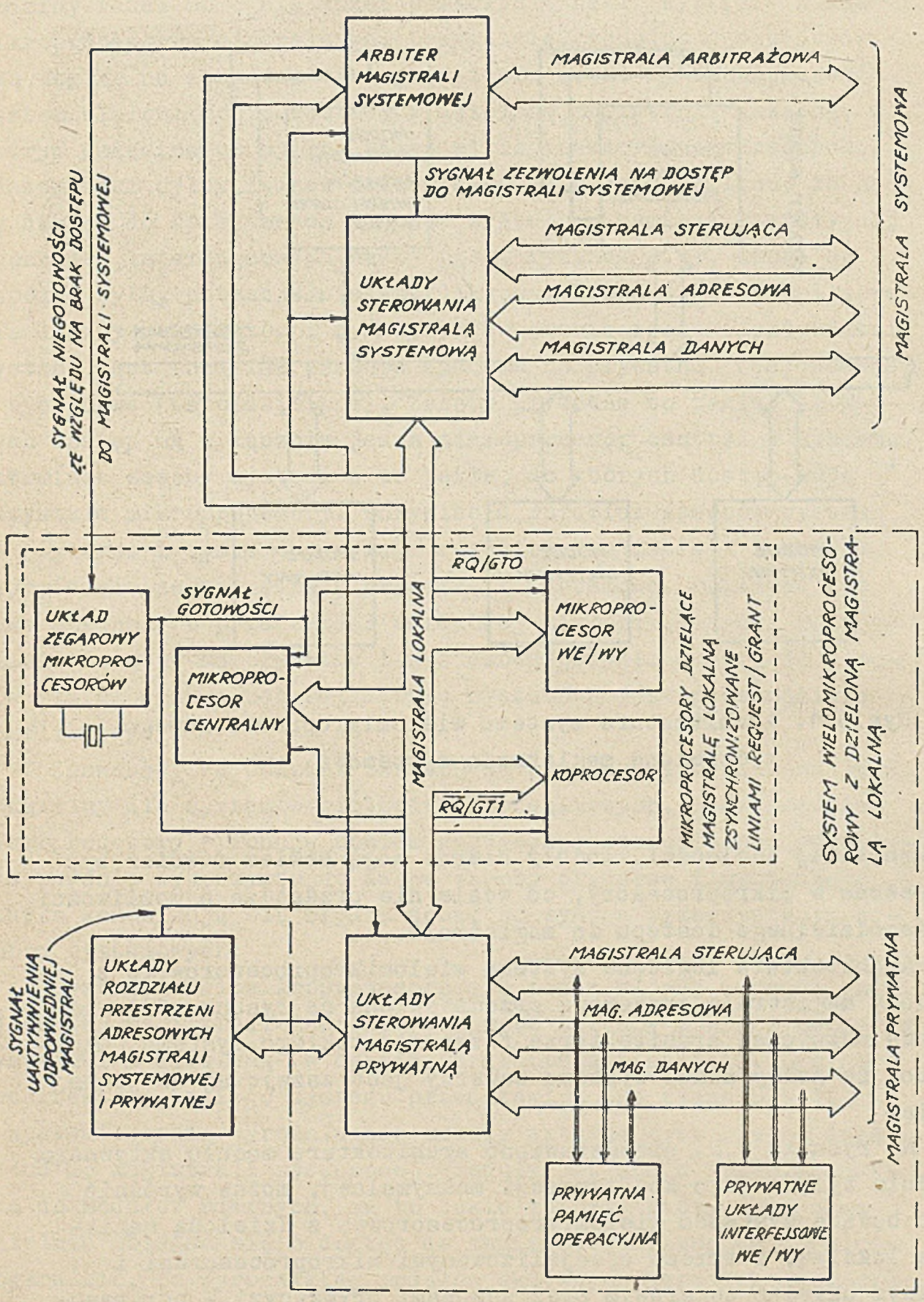


Rys.5.1. Architektura systemu wielomikroprocesorowego z dzieloną magistralą systemową

uzyskania tej zdolności. Moduły bierne mogą bowiem również być wyposażone w mikroprocesory, co wcale nie przesądza o możliwości ich samodzielnego dostępu do magistrali.

Architektura logiczna systemu wielomikroprocesorowego z dzieloną magistralą systemową przedstawiona na rysunku 5.1. odpowiada nazwyczaj architekturze fizycznej takiego systemu. Oznacza to, że każdy moduł systemu zawarty jest zazwyczaj na jednej płycie.

Na rysunku 5.2. przedstawiono architekturę modułu aktywnego. W module tym, w jego konfiguracji maksymalnej, można wyróżnić część będącą systemem wielomikroprocesorowym z dzieloną magistralą lokalną, z trzema specjalizowanymi mikroprocesorami i wspólnym układem zegarowym oraz zasobami prywatnymi - por.rysunek 4.1. W konfiguracji minimalnej modułu część ta redukuje się do pojedynczego mikroprocesora centralnego lub we/wy, bez do-



Rys.5.2. Architektura wewnętrzna modułu aktywnego

datkowych mikroprocesorów, zasobów prywatnych, układów sterowania magistralą prywatną i w konsekwencji układów rozdziału przestrzeni adresowej magistrali systemowej i prywatnej. Oczywiście możliwe są konfiguracje pośrednie.

W stosunku do systemu z dzieloną magistralą lokalną moduł aktywny zawiera pewne układy dodatkowe. Przede wszystkim układy sterowania magistralą systemową. Z dokładnością do szczegółów układy te nie różnią się od odpowiednich układów sterowania magistralą prywatną.

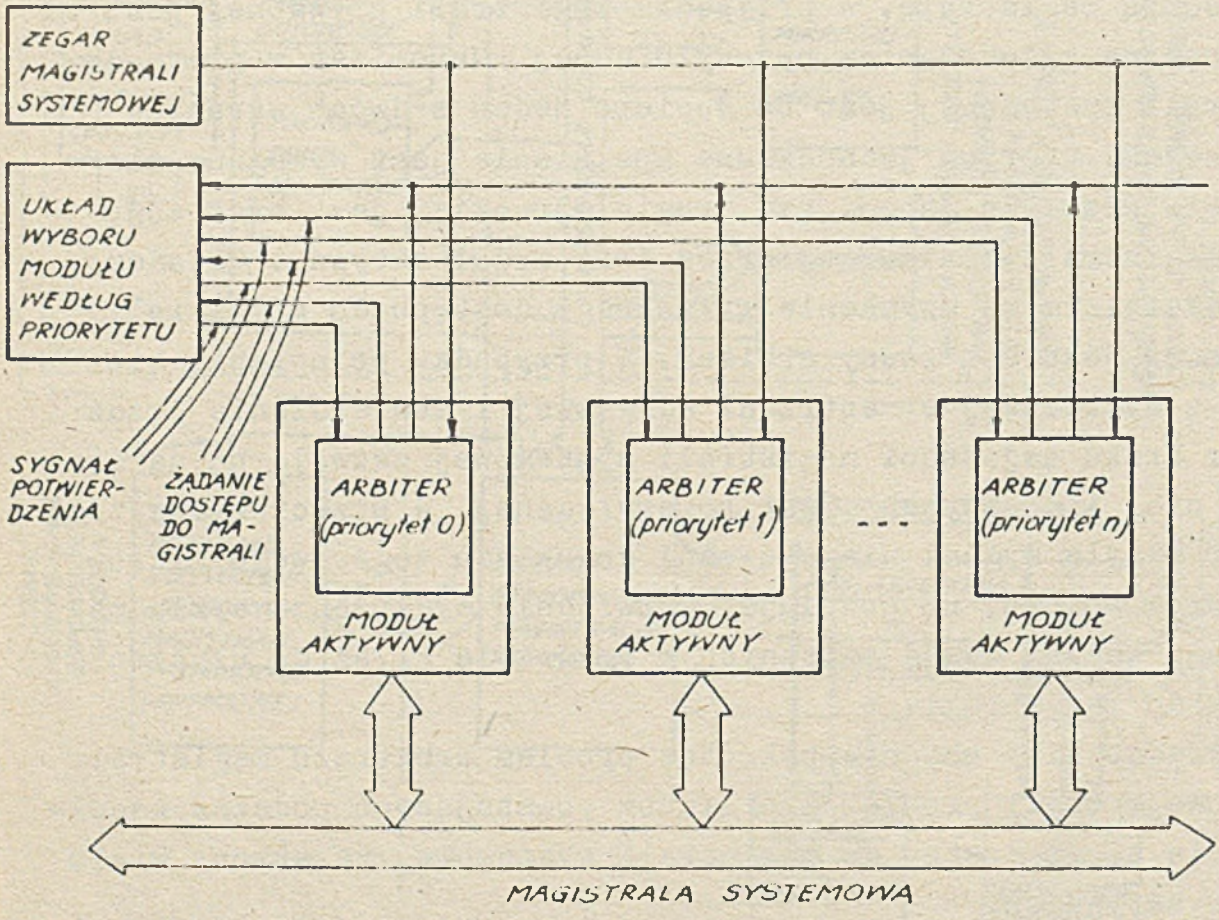
Rozdział przestrzeni adresowej mikroprocesora pomiędzy magostralę prywatną i systemową dokonywany jest przez układ dekoderów adresów. Układ ten jest dołączony do magistrali adresowej magistrali prywatnej, na tej bowiem magistrali mogą pojawić się wszystkie adresy przesyłane przez magistralę lokalną modułu. Po stwierdzeniu do której przestrzeni adresowej - prywatnej czy systemowej - należy dany adres uaktywniane są układy sterowania odpowiednią magistralą. W przypadku magistrali prywatnej jest to równoznaczne z dokonaniem przesyłu danej. Natomiast w przypadku magistrali systemowej jest to dopiero jeden z dwóch warunków koniecznych, których jednoczesne spełnienie jest wymagane dla dokonania przesyłu danej. Tym drugim warunkiem jest brak zajętości magistrali systemowej przez inny moduł aktywny. Układem odpowiedzialnym za uzyskanie wyłącznego dostępu do magistrali systemowej jest tak zwany arbiter. W przypadku pojawienia się adresu z systemowej przestrzeni adresowej i stwierdzenia przez arbiter braku zajętości magistrali systemowej zezwala on na dostęp do niej i dokonywany jest przesył danej. W przeciwnym razie, arbiter wysyła sygnał niegotowości do układu zegarowego mikroprocesorów modułu, co powoduje odpowiednie wydłużenie cyklu maszynowego według zasad opisanych w rozdziale 2, aż do uzyskania dostępu do magistrali.

Przedstawimy obecnie pokrótce problem arbitrażu magistrali systemowej, czyli problem konfliktów powstających podczas współubiegania się o dostęp do magistrali systemowej ze strony wielu modułów aktywnych.

Najprostszym rozwiązaniem tego problemu byłoby wprowadzenie wspólnego zegara dla wszystkich mikroprocesorów systemu, tak aby zapewnić ich synchroniczną pracę i realizować dostęp do

magistrali systemowej cyklicznie. Oczywiście rozwiązanie takie spowodowałoby bardzo nieefektywną pracę systemu, ze względu na to, że często magistrala byłaby przydzielona do mikroprocesora nie żądającego dostępu do niej, podczas gdy inne oczekiwałyby na dostęp. Z tego względu w stosowanym rozwiązaniu mikroprocesory różnych modułów aktywnych nie są zsynchronizowane między sobą i mogą wysyłać żądania dostępu w dowolnych chwilach czasu. Żądania te są następnie synchronizowane przez arbitery z zegarem magistrali systemowej. Po zsynchronizowaniu możliwe jest określenie, który z ubiegających się o dostęp do magistrali systemowej modułów aktywnych uzyska go. Wybór modułu odbywa się według algorytmu priorytetowego.

Miejsce arbiterów w architekturze systemu wielomikroprocesorowego z dzieloną magistralą systemową przedstawiono na rysunku 5.3.



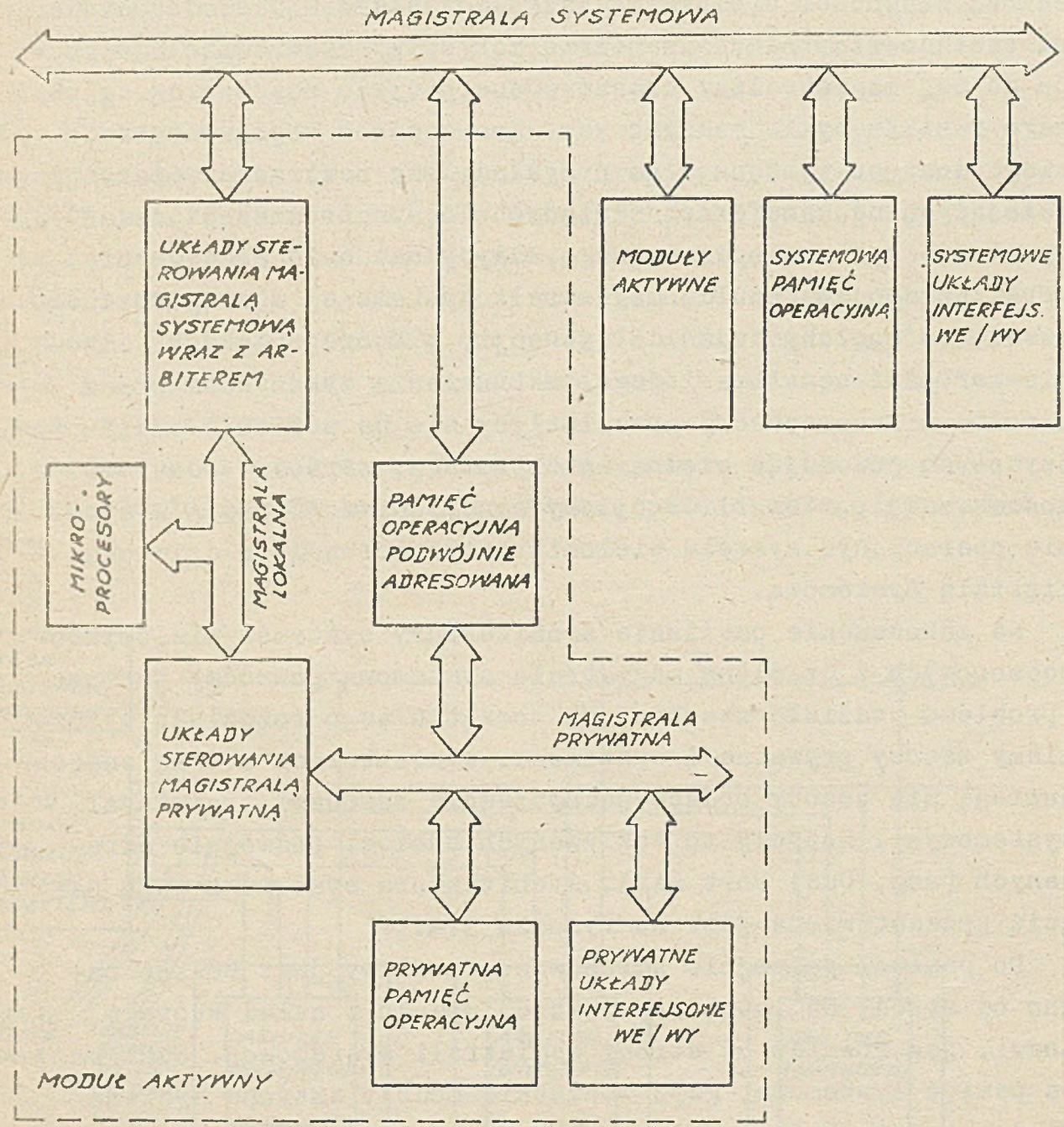
Rys.5.3. Układ arbitrażu magistrali systemowej.

Mówiąc o problemie arbitrażu należy wspomnieć o pewnej koniecznej własności arbitra magistrali systemowej, a mianowicie zdolności do zapewnienia w niektórych przypadkach ciągłego dostępu do tej magistrali w czasie pełnego cyklu rozkazowego złożonego z kilku cykli maszynowych. Konieczność zagwarantowania takiego dostępu występuje na przykład przy realizacji operacji działających na semaforach służących do synchronizacji zadań na poziomie systemu operacyjnego. Gdyby nie było możliwości nieprzerwanego dostępu do magistrali systemowej w ciągu pełnego rozkazu, to mogłoby dojść do sytuacji, w której rozkazy testowania wartości semafora i jego uaktualniania wykonywane przez dwa różne mikroprocesory przeplotłyby się na poziomie cykli maszynowych powodując błędną interpretację wartości semafora. W konsekwencji, niemożliwa byłaby synchronizacja zadań w systemie operacyjnym systemu wielomikroprocesorowego z dzieloną magistralą systemową.

Na zakończenie omawiania architektury systemów wielomikroprocesorowych z dzieloną magistralą systemową powróćmy jeszcze do problemu podziału zasobów. Na początku tego rozdziału wyróżniliśmy zasoby prywatne i systemowe. W niektórych rozwiązaniach pojawiają się zasoby będące jednocześnie zasobami prawatnymi i systemowymi. Dotyczy to tak zwanych pamięci podwójnie adresowanych (ang. Dual Port RAM). Architektura systemu z taką pamięcią przedstawiona jest na rysunku 5.4.

Do pamięci podwójnie adresowanej możliwy jest dostęp zarówno od strony magistrali prywatnej modułu w skład którego wchodzi, jak również od strony magistrali systemowej. Jest to więc pamięć systemowa, gdyż wszystkie moduły aktywne systemu mają do niej dostęp, a jednocześnie jest to pamięć prawatna danego modułu o możliwości szybkiego dostępu z pominięciem arbitrażu magistrali systemowej.

Sterowanie pamięci podwójnie adresowanych redukuje liczbę konfliktów w systemie i tym samym wpływa na wzrost jego efektywności. Stosowanie tych pamięci jest szczególnie korzystne przy wymianie dużych bloków danych między modułami aktywnymi systemu.



Rys.5.4. System wielomikroprocesorowy z pamięcią operacyjną podwójnie adresowaną.

6. Uwagi końcowe

W referacie tym naszkicowalismy główne problemy architektury systemów wielomikroprocesorowych trzeciej generacji. Szerokie upowszechnienie zastosowań tych systemów, również w Pols-

ce, jest naszym zdaniem kwestią przewidywalnej przyszłości. Upowszechnienie to zależeć będzie w dużej mierze od rozwoju systemów operacyjnych systemów wielomikroprocesorowych. Jest to zagadnienie bardzo aktualne i ciekawe, wykraczające jednak niestety poza ramy niniejszego referatu.

Jesienna Szkoła PTI
Rydzyna, październik 1984

PRAWNE ASPEKTY INFORMATYKI

Dr Jacek Irlik
Centrum Techniki Obliczeniowej
Uniwersytet Śląski
ul. Uniwersytecka 4
40-007 Katowice

1. Uwagi wstępne.

Na samym wstępie rozważań na temat prawnych aspektów informatyki warto słów kilka poświęcić bardziej precyzyjnemu zakreśleniu ich przedmiotu. Przede wszystkim należy stwierdzić, że używając określenia "prawne aspekty informatyki" mamy na myśli oceny prawne sytuacji i zjawisk społecznych powstających w związku z informatyką, a więc w związku ze stosowaniem sprzętu komputerowego w procesie zbierania, przechowywania, przetwarzania przesyłania i wydawania informacji w różnych dziedzinach życia. Krąg tych sytuacji czy zjawisk jest bardzo rozległy i powiększa się stale w tempie odpowiadającym rozwojowi zarówno samej techniki komputerowej jak i obszarów jej zastosowań. Ustaliwszy więc wstępnie, że interesować nas będą oceny prawne sytuacji i zjawisk społecznych powstających w związku z informatyką powinniśmy więc w następnej kolejności krąg ten zakreślić dokonując ich wyboru według jakiegoś przyjętego kryterium. W rozważaniach niniejszych ograniczymy się do sytuacji takich, w których ocena prawna jest szczególnie istotna ze względu na

występujące lub mogące w nich wystąpić konflikty interesów różnych podmiotów. W wyniku dokonania powstających w takich sytuacjach stosunków społecznych ustalone zostają, określine wolą prawodawcy, uprawnienia i obowiązki uczestników tych stosunków.

Dokonanie takiej oceny nie jest niestety zadaniem łatwym, a ponadto nie zawsze ma jednoznaczne rozstrzygnięcie, zwłaszcza w sferze stosunków społecznych związanych z nową i ponadto dynamicznie rozwijającą się dziedziną społeczno-gospodarczej aktywności, jaką jest informatyka. Wynika to z kilku przyczyn. Po pierwsze, ze względu na hermetyczny aparat pojęciowy informatyki prawnik może napotkać poważne trudności w zrozumieniu istoty zjawiska, które zamierza ocenić w świetle prawa. Po drugie, nawet zrozumienie tej istoty nie wyeliminuje kolejnej trudności, którą jest zakwalifikowanie zjawiska w oparciu o system instytucji (pojęć) prawnych wykształconych już często w czasach rzymskich. Po trzecie, w związku z nowym rodzajem sytuacji czy zjawisk występujących w informatyce można będzie stosunkowo często spotkać się ze zjawiskiem określanym jako "luka w prawie", a więc z brakiem przepisów, które pozwalałyby na dokonanie wspomnianej oceny bezpośrednio (opinie na ten temat wyrażone zostały np. w raporcie opracowanym [1] przez zespół działający na zlecenie PTI).
Należy ponadto wziąć pod uwagę, że zastosowanie określonych przepisów prawa wymaga zawsze uprzedniego dokonania ich wykładni, tj. wymaga ustalenia ich rzeczywistego znaczenia i zakresu zastosowania. Jak wiadomo z teorii [2]. wynik dokonywanej wykładni przepisu nie zawsze jest taki sam i zależy od przyjętej metody postępowania, tj. od wyboru określonego systemu dyrektyw interpretacyjnych. Ocena dokonanego wyboru tych dyrektyw może jedynie być relatywna w stosunku do określonej hierarchii wartości lub celów a różnych hierarchii uznawanych przez różne osoby porównywać w sposób obiektywny się nie da [3]. Przykład: osoba A uznaje pewność obrotu prawnego za wartość nadrzędną i skłania się do dokonywania tzw. wykładni literalnej określonego przepisu. Osoba B natomiast, twierdząc, że prawo nie nadąża za życiem, powoła się na względy słuszności czy sprawiedliwości i skłonna jest traktować ten sam przepis bardziej elastycznie, a więc godzić się na tzw. rozszerzającą czy tzw. zwężającą wykładnię. W każdym z takich przypadków zakres zastosowania przepisu

może być inny, inna może być więc ocena prawna sytuacji, która raz danym przepisem będzie objęta, a innym razem nie. Przyjęte metody dokonywania wykładni przepisów prawa i jej wynik mają ponadto istotny wpływ na rezultaty działań zmierzających do wypełnienia każdorazowo stwierdzonej luki w prawie, działań polegających na zastosowaniu określonych reguł wnioskowania prawniczego. Rozbieżności pomiędzy różnymi, niezależnie dokonanymi ocenami prawnymi danej sytuacji są oczywiście na ogół tym mniejsze, w im większym stopniu instytucje prawne rozumiane zgodnie z literalną interpretacją przepisów dostosowane są do jej istoty. Tak będzie jeśli kwalifikować będziemy sytuacje od dawna spotykane w życiu, wielokrotnie powtarzające się, dla regulacji których można było dawno ustanowić odpowiednie przepisy. Jeśli określona dziedzina aktywności społeczno-gospodarczej rozwija się na tyle burzliwie, że występują w niej wciąż nowe sytuacje i zjawiska to trudno oczekiwać, aby przepisy interpretowane literalnie nadawały się wprost do zastosowania w celu dokonania jednoznacznej oceny. Nie należy więc dziwić się, że w ocenach prawnych sytuacji spotykanych w informatyce mogą wystąpić nawet poważne rozbieżności.

Należy w tym miejscu wyraźnie zastrzec, że tworzenie szczegółowych przepisów, literalnie odpowiadającym sytuacjom spotykanym w informatyce byłoby iluzorycznym rozwiązaniem wspomnianych powyżej trudności. Przepisy te musiałyby praktycznie zmieniać się w tempie odpowiadającym pojawianiu nowych zjawisk w informatyce. Ich niestabilność stworzyłaby stan prawny wysoce niepewny. Niepewność ta byłaby prawdopodobnie większa niż niepewność wynikająca z możliwości rozbieżnego interpretowania przepisów ustabilizowanych. Brak stabilności w stanie normatywnym uniemożliwiałby ponadto jakąkolwiek głębszą ocenę skutków funkcjonowania takich czy innych rozwiązań legislacyjnych, w związku z czym utrudniałby proces legislacji racjonalnej. Szczególna regulacja prawna dotycząca informatyki powinna ograniczyć się zatem do tych kategorii zjawisk, które zdołały już się w miarę trwały sposób ukształtować i w przypadku których można już ocenić skutki przepisów funkcjonujących aktualnie.
W ocenie zjawisk mniej ukształtowanych społecznie istotne jest raczej uświadomienie wspomnianego związku pomiędzy wyborem war-

tości czy celów uznanych za nadrzędne, a kierunkiem interpretacji przepisów obowiązujących czyli w rezultacie kwalifikacją prawną tych zjawisk. Uświadomienie to może być istotne dla wypracowania w przyszłości postulatów de lege ferenda.
Należy wreszcie zauważyć, że pomimo wszystko wiele sytuacji powstających w związku z informatyką może być opisanych w kategoriach pojęć dobrze ugruntowanych już w systemie prawa (sprzedaż sprzętu, serwis urządzeń technicznych, obrót rozwiązaniami technicznymi w zakresie hardware'u i tp.), a ich ocena prawna nie stanowi zagadnienia szczególnego w stosunku do oceny prawnej sytuacji od dawna spotykanych w życiu.

Po dokonaniu tych uwag można przystąpić do bliższego omówienia wspomnianego w akapicie pierwszym kręgu sytuacji i zjawisk społecznych powstających w związku z informatyką, w których ocena prawna jest szczególnie istotna ze względu na występujące lub mogące wystąpić w nich konflikty interesów różnych podmiotów. Sytuacje te podzielimy wpierw na dwie grupy. W grupie pierwszej rozważymy sytuacje powstające w trakcie tworzenia i rozpowszechniania narzędzi informatycznych, a w grupie drugiej - powstające w trakcie stosowania tych narzędzi w różnych dziedzinach życia. Tworzenie i rozpowszechnianie narzędzi informatycznych wymaga współpracy wielu podmiotów. Od dawna powszechnie uznano [4], że najlepszą metodą dla regulacji prawnej stosunków pomiędzy współuczestnikami szeroko rozumianej działalności gospodarczej jest metoda cywilnoprawna. Metoda ta polega na tym, że poszczególnym podmiotom przyznane zostają formalnie równe kompetencje (zasada równości stron) do względnie swobodnego ukształtowania wzajemnych uprawnień i obowiązków (zasada swobody umów). Ocena prawna sytuacji powstających w trakcie tworzenia i rozpowszechniania narzędzi informatycznych, a więc określenie prawnie sankcjonowanych uprawnień i obowiązków współuczestników tego procesu dokonywana będzie więc w oparciu o system prawa cywilnego. Zagadnieniom tym poświęcony zostanie rozdział 2. Stosowanie narzędzi informatycznych w różnych dziedzinach życia stwarza wiele sytuacji konfliktowych. Konflikty takie występują najczęściej pomiędzy podmiotem, który stosuje narzędzia informatyczne (nazwiemy go przetwarzającym), a podmiotem, którego sytuacja

faktyczna lub prawna zależy do faktu i sposobu wykorzystania takich narzędzi (nazwiemy go zainteresowanym przetwarzaniem). Ocena prawna powstających sytuacji dotyczyć będzie głównie ochrony zainteresowanego przetwarzaniem oraz granic odpowiedzialności przetwarzającego za naruszenie prawnie uzasadnionego interesu pierwszego z tych podmiotów. Oceny takiej poszukiwać można w oparciu o system prawa cywilnego ale okazuje się, że prawo to nie zawsze może dać skuteczną ochronę zainteresowanym. Należałoby więc de lege ferenda postulować wzmocnienie tej ochrony poprzez dodatkową regulację. Rozważania dotyczące tej grupy sytuacji będą przedmiotem niniejszego opracowania w rozdziale 3.

2. Tworzenie i rozpowszechnianie narzędzi informatycznych.

Niniejszy rozdział poświęcony jest zagadnieniom oceny prawnej sytuacji powstających w trakcie tworzenia i rozpowszechniania narzędzi informatycznych. Zainteresowanie nasze koncentrować się będzie głównie na zagadnieniach prawnych związanych z obrotem oprogramowaniem systemów informatycznych. Obrót hardware'm oraz rozwiązaniami technicznymi dotyczącymi sprzętu nie jest bowiem z prawnego punktu widzenia odmienny od obrotu urządzeniami lub rozwiązaniami technicznymi poza informatyką. Podstawową formą prawną obrotu, a wszczególności obrotu oprogramowaniem są umowy. Ocena prawna zawierany h umów dokonywana jest w oparciu o system prawa cywilnego i polega na ustaleniu uprawnień i obowiązków każdej ze stron (a więc treści umowy), wynikających z ich woli oraz "z ustawy, zasad współżycia społecznego i ustalonych zwyczajów" (art.56 k.c.). W ocenie tej należy ponadto brać pod uwagę "zgodny zamiar stron i cel umowy" (art.65 k.c.). Oceny prawnej umów w obrocie oprogramowaniem nie można dokonać bez ustalenia praw bezwzględnych, które określonym osobom mogą przysługiwać w związku z faktem wytworzenia oprogramowania, a więc praw mających za przedmiot oprogramowanie jako dobro niematerialne (zagadnieniom tym poświęcone były np. prace J.Waluszewskiego [5]). W związku z powyższym w niniejszym rozdziale omówione zostaną dwie grupy zagadnień. Pierwsza z nich to zagadnienia kwalifikacji oprogramowania jako niematerialnego przedmiotu praw. Druga

z tych grup obejmuje zagadnienia typologii umów zawieranych w obrocie oprogramowaniem oraz odpowiedzialności za ich nienależyte wykonanie. Należy zwrócić uwagę, że obu tym grupom zagadnień poświęcona była interesująca monografia B.Czachórskiej [6].

Przystępując do rozstrzygnięcia zagadnienia kwalifikacji oprogramowania jako niematerialnego przedmiotu praw ustalimy na wstępie, że przez oprogramowanie rozumieć będziemy łącznie

- p r o g r a m y zakodowane na nośniku i będące postacią, w której ustalone zostały algorytmy,
- d o k u m e n t a c j ę ź r ó d ł o w ą programów, którą stanowią wszelkie materiały (teksty, rysunki, schematy itp.), stanowiące razem jednoznaczny opis algorytmów ustalonych w postaci programów,
- d o k u m e n t a c j ę f u n k c j o n a l n ą programów, którą stanowią wszelkie materiały informujące o realizowanych przy użyciu tych programów funkcjach oraz o treści i strukturze danych wejściowych i wyjściowych,
- d o k u m e n t a c j ę e k s p l o a t a c y j n ą programów, którą stanowią wszelkie materiały umożliwiające i ułatwiające wykorzystanie tych programów.

Ostatnie trzy z wymienionych pozycji będziemy nazywać m a t e r i a ł a m i p o m o c n i c z y m i . Zgodnie z terminologią ustaloną przez WIPO (World Intellectual Property Organisation) materiałami pomocniczymi nazwane są tylko dwie ostatnie z nich, druga z wymienionych pozycji określana jest w tej terminologii osobno jako o p i s y p r o g r a m ó w .

Najwięcej kontrowersji wzbudza kwestia prawnej kwalifikacji programów. Podmiotowe majątkowe prawa bezwzględne na dobrach niematerialnych są w polskim systemie prawnym (podobnie jak w wielu innych krajach) regulowane jako prawa autorskie lub wynalazcze. Jeśli chodzi o prawo wynalazcze, to bowiązująca w Polsce ustawa z dnia 19 października 1972 r. o wynalazczości (Dz.U. nr.43 poz.272 z pózn.zmianami) wyklucza programy dla maszyn cyfrowych jako jego przedmiot (art.2 p.5 ust. o wyn.). W czysto teoretycznych rozważaniach czy programy dla maszyn cyfrowych mogą posiadać cechy uznane jako cechy konstytutywne wynalazku zwraca się uwagę

na brak takich jego cech jak "techniczny charakter rozwiązania" lub "nie wynikanie rozwiązania w sposób oczywisty ze stanu techniki". Tak więc nawet uchylenie jednoznacznego zapisu w art.2 ustawy o wynalazczości nie prowadziłoby do możliwości kwalifikowania programów jako wynalazków.

Wysoce kontrowersyjna jest kwestia uznania programów za przedmiot prawa autorskiego . W świetle ustawy z dnia 10 lipca 1952 r o prawie autorskim (Dz.U. nr.35, poz.234) kwalifikacja taka jest niedopuszczalna jeśli przyjąć literalną wykładnię terminu "utwór literacki, naukowy bądź artystyczny" występującego w art.1 tej ustawy. W doktrynie panuje dość powszechnie pogląd o możli- zaakceptowania wykładni rozszerzającej tego terminu, przy czym rozumie się utwór jako przedmiot scharakteryzowany określonym zespołem cech uznanych za istotne. Różnice w poglądach co do możliwości uznania programu za utwór polegają z jednej strony na uznawaniu rozmaitych cech tak pojmowanych utworów za najbardziej istotne, a z drugiej strony na podnoszeniu rozmaitych cech programów jako cech najbardziej charakterystycznych. Autorzy dopuszczający możliwość kwalifikowania programów jako utworów podnoszą takie cechy programów jak "możliwość ustalenia pismem" czy "odzwierciedlenie piętna osobistego twórcy". Autorzy odmawiający takiej kwalifikacji podnoszą argumenty, że program jest odwzorowaniem obiektywnie istniejącej rzeczywistości, swoistą formułą, zapisem ustalenia naukowego oraz, że odzwierciedlenie piętna osobistego twórcy jest w przypadku programu bez znaczenia. Piszący te słowa podziela poglądy autorów należącej do drugiej z tych grup. Nie jest istotnie przedmiotem prawa autorskiego ani formuła $E = mc^2$, ani zapis równań Maxwella, ani opis nowoustalonej metody pomiaru jakiejś mniej lub bardziej uniwersalnej wielkości przyrodniczej - niezależnie od tego jak wiele pracy włożyli w takie ustalenia ich odkrywcy. Programy dla maszyn cyfrowych nie powinny być więc uznane za utwór w rozumieniu prawa autorskiego.

Należy zwrócić jednak uwagę, że programy - podobnie jak ustalenia naukowe lub inne przejawy działalności twórczej - są przedmiotem ochrony udzielanej przez prawo cywilne jako dobra osobiste twórcy (odkrywcy), na mocy art.23 i 24 k.c.. Osoba taka może domagać się, aby powoływano autorstwo ustalenia i by go sobie nie przywłaszczano.

Nawet jednak przy założeniu, że programy dla maszyn cyfrowych nie są kwalifikowane jako przedmiot majątkowych praw podmiotowych o charakterze bezwzględnym, nie można twierdzić, że prawo odmawia producentom programów jakiejkolwiek ochrony ich interesów. Obowiązująca do chwili obecnej w Polsce ustawa z 1926 roku o zwalczaniu nieuczciwej konkurencji zapewnia prawną ochronę tajemnicom produkcyjnym i handlowym. Utrzymywanie więc w tajemnicy rozwiązań zawartych w programie jest prawnie chronionym stanem faktycznym, a każda osoba, która w sposób sprzeczny z zasadami współżycia społecznego naruszy stan tajemnicy odpowiedzialna jest za wyrządzoną w ten sposób szkodę. Programy dla maszyn cyfrowych oraz całe oprogramowanie można więc traktować jako dobro niematerialne podobne do know-how [7].

Nie budzi zasadniczych wątpliwości kwestia kwalifikacji prawnej samych materiałów pomocniczych. Są to utwory ustalone w postaci pisma, rysunków, diagramów i tp. a dotyczą ustaleń naukowych. Mogą odzwierciedlać piętno osobiste ich autorów. Mogą być więc bez wątpienia uznane za rodzaj utworów naukowych i korzystać z ochrony prawa autorskiego. Sposób korzystania z egzemplarzy takich materiałów i rozporządzanie nimi nie może zatem naruszać praw ich autorów lub osób, na które prawa te zostały przeniesione. Nie wolno więc bez ich zgody takich utworów tłumaczyć, zwielokrotniać czy rozpowszechniać.

Autorka powoływanej wcześniej pracy, B.Czachórska, dzieli umowy zawierane w obrocie oprogramowaniem na dwie grupy. Do pierwszej z nich zalicza umowy o wytworzenie oprogramowania, do drugiej - umowy dotyczące oprogramowania gotowego.
Umowę o wytworzenie oprogramowania Czachórska próbuje zakwalifikować zasadniczo jako przypadek umowy o dzieło (art.621 i nast. k.c.), przez którą "przyjmujący zamówienie zobowiązuje się do wykonania oznaczonego dzieła, a zamawiający do zapłaty wynagrodzenia". Wątpliwości powstają w związku z tym, że zgodnie z powszechnym poglądem doktryny zamawiane dzieło powinno być materialnym lub niematerialnym rezultatem samoistnym, oznaczonym, obiektywnie możliwym i subiektywnie pewnym do uzyskania. Nie zawsze bowiem można uznać, że zamawiane oprogramowanie jest dos-

tatecznie oznaczone oraz, że jego uzyskanie jest obiektywnie możliwe czy subiektywnie pewne (zwłaszcza kiedy zamawiający sam nie bardzo wie jakie oprogramowanie jest mu potrzebne). Brak tych cech może spowodować zasadnicze trudności w wykonaniu zobowiązania przez przyjmującego zamówienie, który za osiągnięcie rezultatu odpowiada przed wierzycielem-zamawiającym. B.Czachórska wspomina o praktyce poprzedzania zawarcia takiej umowy przeprowadzeniem odpowiednich prac w celu dokładnego oznaczenia przedmiotu zamówienia i w celu ustalenia stopnia jego wykonalności, a praktykę taką uważa za godną polecenia. Prace poprzedzające zawarcie takiej umowy mogłyby być traktowane jako usługa i oceniane prawnie w oparciu o przepisy dotyczące umowy zlecenia (art.734 i nast. k.c.). W odniesieniu do umowy o dzieło należy pamiętać, że przewidziany w art.643 kc. obowiązek wydania dzieła zamawiającemu łącznie z obowiązkiem zapłaty przez zamawiającego wynagrodzenia mają ten skutek, że wszelkie prawa do oprogramowania, niezależnie od jego kwalifikacji prawnej, od chwili jego wydania przysługują zamawiającemu. Nie jest tak jedynie jeśli strony umowy postanowiły inaczej. Jeśli zamówienie na wykonanie oprogramowania przyjmuje jednostka naukowa wówczas - w niektórych przypadkach - można zawieraną umowę kształtować jako umowę o prace naukowo-badawcze z wynikającymi stąd skutkami zarówno co do uprawnień i obowiązków stron jak i co do odpowiedzialności wykonawcy.

Obrót oprogramowaniem gotowym budzi wiele kontrowersji. Przede wszystkim zasadnicze wątpliwości budzi często używane określenie "sprzedaż oprogramowania". Zgodnie z art.535§1 k.c. "przez umowę sprzedaży sprzedawca zobowiązuje się przenieść na kupującego własność rzeczy i wydać mu rzecz, a kupujący zobowiązuje się rzecz odebrać i zapłacić sprzedawcy cenę". Nieco inaczej jest w obrocie uspołecznionym (art.535§2 k.c.) ale sprzedaż nadal dotyczy rzeczy. Sprzedaż więc w obrocie oprogramowaniem może dotyczyć jedynie jego materialnych substratów (nośników z programami, egzemplarzy dokumentacji). Wówczas jednak, przy założeniu, że oprogramowanie jako dobro niematerialne nie jest przedmiotem żadnych majątkowych praw bezwzględnych, substraty te mogłyby służyć nabywcy bez żadnych ograniczeń zapewniających och-

ronę interesu wytwórcy. Nabywca mógłby je bowiem dowolnie wykorzystywać (powielać, udostępniać osobom trzecim itd.) bowiem zgodnie z art.140 k.c. "właściciel może z rzeczy korzystać i rzeczą rozporządzać dowolnie w granicach ustaw i zgodnie ze społeczno-gospodarczym przeznaczeniem prawa własności". Sposób korzystania z materialnych nośników dóbr niematerialnych chronionych prawem autorskim (np. kaseta z nagraniem cudzego utworu) czy patentem (np. urządzenie, w którym zastosowano opatentowane rozwiązanie) ograniczony jest natomiast przepisami prawa autorskiego czy wynalazczego. Jedyne więc ograniczenie, któremu podlegałby nabywca materialnych substratów oprogramowania odnosiłoby się do egzemplarzy materiałów pomocniczych. Nie takiego jednak ograniczenia oczekują przede wszystkim wytwórcy oprogramowania. Gdyby uznać oprogramowanie w całości jako przedmiot prawa autorskiego, powstałyby wówczas znaczne trudności w kwalifikowaniu stanów faktycznych w obrocie tym dobrem w ramach systemu pojęć ukształtowanych w prawie autorskim. Dokonując jednak takiej kwalifikacji należałoby sugerować umowę licencyjną jako formę prawną udostępniania oprogramowania gotowego.

Niepewność co do prawnej kwalifikacji oprogramowania jako dobra niematerialnego skłania do postulowania regulacji stosunków w obrocie oprogramowaniem w drodze szczegółowych postanowień umownych. Postanowienia umów dotyczących udostępnienia oprogramowania gotowego powinny więc określać

- zakres i sposób wykorzystania przez odbiorcę udostępnionego oprogramowania,
- uprawnienia odbiorcy w zakresie rozporządzania udostępnionym oprogramowaniem,
- uprawnienia odbiorcy w zakresie dokonywania zmian w oprogramowaniu,
- uprawnienia udostępniającego odnośnie do udostępnionego oprogramowania po zawarciu umowy,
- obowiązki obu stron co do zachowania rozwiązań zawartych w oprogramowaniu w tajemnicy.

Postanowienia te są charakterystyczne dla tzw. umowy know-how [7] podobieństwo do której wykazuje więc umowa u udostępnienie oprogramowania. Niezależnie od postanowień jak wymienione powyżej

należy uznać, że umowa taka zawiera w sobie najczęściej tak czy inaczej elementy sprzedaży w odniesieniu do materialnych substratów oprogramowania, sprzedaży rzeczy, nośników programów, posiadających ściśle określone fizyczne właściwości i użyteczność.

Kodeks cywilny określa odpowiedzialność dłużnika z tytułu nienależytego wykonania zobowiązania (art.471 i nast. k.c.). Dodatkowo, przepisy dotyczące niektórych umów przewidują reżim szczególny takiej odpowiedzialności, uniezależniając ją od winy dłużnika. Są to m.in. przepisy dotyczące rękojmi za wady. Art.556§1 k.c. stanowi, że sprzedawca jest odpowiedzialny względem kupującego, jeżeli rzecz sprzedana ma wady fizyczne t.j.

- ma wadę zmniejszającą jej wartość lub użyteczność ze względu na cel w umowie oznaczony albo wynikający z okoliczności lub z przeznaczenia rzeczy,
- nie ma właściwości, o których sprzedawca zapewnił kupującego,
- została kupującemu wydana w stanie niezupełnym

oraz, zgodnie z art.556§2 k.c., jeśli rzecz sprzedana ma wadę prawną, t.j.

- stanowi własność osoby trzeciej albo jest obciążona prawem osoby trzeciej.

W razie stwierdzenia wady kupującemu przysługują określone uprawnienia jak prawo odstąpienia od umowy, możność żądania obniżenia ceny, żądania wymiany rzeczy na rzecz wolną od wad lub naprawy rzeczy (art.560 ÷ 562 k.c.).
Podobnie, na mocy art.637 i 638 k.c. ukształtowana jest instytucja rękojmi za wady dzieła. Strony mogą odpowiedzialność z tytułu rękojmi ukształtować odmiennie w drodze umowy, ograniczenie jej jest jednak niedopuszczalne w obrocie uspołecznionym i mieszanym chyba, że przewiduje to przepis szczególny. Takiego przepisu w odniesieniu do obrotu oprogramowaniem jednak nie ma. Instytucja rękojmi może mieć zastosowanie w obrocie oprogramowaniem. Nie ulega to wątpliwości w przypadku umowy o wytworzenie oprogramowania (umowy o dzieło). W umowie o udostępnienie oprogramowania traktujemy świadczenie polegające na wydaniu jego materialnych substratów jako element sprzedaży, co usprawiedliwia stosowanie przepisów o rękojmi. Wszelkie wady oprogramowania na-

leży uznać za wady fizyczne, odzwierciedlone zostają bowiem w jego materialnych substratach. Rzeczy te wówczas nie mają mianowicie właściwości, o których kupujący był zapewniany (choćby poprzez treść dokumentacji funkcjonalnej), nie spowodują bowiem określonego działania maszyny cyfrowej. Rzeczy te mają w konsekwencji zmniejszoną użyteczność lub nie posiadają jej wcale.

Szczególna odpowiedzialność za wady oprogramowania może również wynikać z odpowiednich postanowień umowy (klauzul gwarancyjnych). W postanowieniach takich zawarte zostają wówczas zapewnienia strony świadczącej co do określonych cech jakościowych przedmiotu świadczenia (oprogramowania) wraz z określeniem szczególnych jej obowiązków w przypadku, w którym przedmiot świadczenia cech tych nie wykazuje. Uzupełnienie treści umowy, której przedmiotem jest oprogramowanie, postanowieniami tego rodzaju może być szczególnie dogodne w związku z wysoce złożoną naturą przedmiotu świadczenia. Odpowiedzialność z tytułu rękojmi nie w każdym bowiem konkretnym stosunku umownym musi chronić właściwie uzasadniony interes każdej ze stron, zwłaszcza jeśli wziąć pod uwagę ogromną w przypadku oprogramowania rozmaitość możliwych stanów faktycznych.

W literaturze na temat środków prawnych przysługujących odbiorcom przedmiotu świadczenia w razie ujawnienia jego wad wskazuje się również (p. np. [8])na inne możliwości. Środkiem takim jest w szczególności możliwość odstąpienia od umowy z powołaniem się na błąd w oświadczeniu woli składanym w chwili jej zawierania. Prawnie doniosły błąd polega wówczas na tym, że odbiorca składa oświadczenie będąc przekonanym, że programy realizują funkcje określone w dokumentacji, podczas gdy w rzeczywistości jest inaczej. Błąd ten jest wywołany przez adresata oświadczenia woli (wydającego przedmiot świadczenia), który przedstawił odbiorcy dokumentację funkcjonalną, nie odzwierciedlającą faktycznego działania programów. W przypadkach takich, niezależnie od winy udostępniającego oprogramowanie odbiorca może skutecznie od umowy odstąpić (art.84§1 k.c.). Możliwość ta wzmacnia pozycję odbiorcy.

3. Ocena prawna skutków stosowania narzędzi informatycznych.

Niniejszy rozdział poświęcony jest prawnej ocenie sytuacji powstających w trakcie stosowania narzędzi informatycznych w różnych dziedzinach życia. W szczególności zainteresowanie nasze budzi ochrona, jakiej prawo może udzielić podmiotom zainteresowanym przetwarzaniem (por. rozdz. 1) w związku z faktycznymi lub prawnymi skutkami stosowania narzędzi informatycznych przez przetwarzającego. Sytuacje będące przedmiotem uwagi w niniejszym rozdziale możemy kolejno podzielić na dwie grupy. Zasadniczo odmienne są sytuacje, w których stosowanie narzędzi informatycznych odbywa się w ramach zawartej uprzednio umowy pomiędzy przetwarzającym i zainteresowanym (niezależnie od tego czy zbieranie danych, ich przetwarzanie, przesyłanie itp. stanowi główny cel umowy, czy ma tylko charakter pomocniczy), od sytuacji, w których skutki stosowania tych narzędzi dotyczą zainteresowanych, którzy nie mieli i nie mają żadnego wpływu ani na fakt ani na sposób tego stosowania. W sytuacjach należących do pierwszej z określonych powyżej grup zarówno fakt jak i sposób stosowania narzędzi informatycznych jest elementem treści stosunku umownego, na którą ma wpływ zainteresowany jako równorzędny partner umowy. Umowy o stosowanie narzędzi informatycznych (eksploatację systemów) omówione są w powoływanej wcześniej monografii Czachórskiej jako "umowy o usługi z zakresu przetwarzania danych" oraz "inne umowy o usługi informatyczne". W tej grupie sytuacji mamy na myśli ponadto inne umowy, w których zastosowanie narzędzi informatycznych stanowi pomocniczy element świadczenia którejś ze stron (np. umowa o prace projektowe, w ramach której wykonuje się obliczenia na maszynie cyfrowej). Ochrona prawna zainteresowanego przetwarzaniem wynika wówczas z ogólnych przepisów prawa cywilnego dotyczących odpowiedzialności dłużnika z tytułu nienależytego wykonania zobowiązania (art.471 i nast. k.c.). Należy podkreślić, że zainteresowany przetwarzaniem może dodatkowo chronić swój interes, negocjując odpowiednie postanowienia w umowie, określające szczególne obowiązki i odpowiedzialność

przetwarzającego. Szczególnym rodzajem takich postanowień powinny być postanowienia dotyczące obowiązków przetwarzającego w zakresie ochrony danych zarówno przed uszkodzeniem czy utratą jak i przed ujawnieniem osobom niepowołanym. Ten rodzaj odpowiedzialności przetwarzającego jest o tyle dogodny dla zainteresowanego przetwarzaniem, że poniósłszy szkodę w wyniku nienależytego wykonania zobowiązania może domagać się jej naprawienia i nie do niego należy dowód winy przetwarzającego, ale przetwarzający musi dowieść, że dołożył należytej staranności i winy nie ponosi.

Zasadniczo odmienna jest sytuacja prawna podmiotów zainteresowanych przetwarzaniem, którzy nie mieli wpływu ani na fakt ani na sposób zastosowania narzędzi informatycznych, z którego wyniknęły niekorzystne dla nich faktyczne bądź prawne skutki. Przykładami takich sytuacji mogą być

- sytuacja osoby, która poniosła szkodę na skutek wydania przez organ administracji państwowej niewłaściwej decyzji, podjętej w oparciu o dane uzyskane w wyniku nierzetelnego stosowania systemu informatycznego lub stosowania systemu nierzetelnie opracowanego,
- sytuacja osoby, której poufne dane osobiste ujawnione zostały w wyniku nierzetelnego stosowania systemu informatycznego przez jednostkę uprawnioną do gromadzenia takich danych (np. jednostkę służby zdrowia)

oraz wiele innych podobnych. Zgodnie z art.415 kodeksu cywilnego "kto z winy swej wyrządził drugiemu szkodę, obowiązany jest do jej naprawienia". Zapis ten jest podstawowym wyrazem ochrony jakiej prawo cywilne udziela wszelkim osobom, które poniosły szkodę z cudzej winy. Prawo to chroni również dobra osobiste poprzez zapis w art.23 i 24 k.c. udzielając osobom, których dobra zostały naruszone - prawa żądania zaniechania działań naruszających oraz prawa żądania czynności potrzebnych do usunięcia jego skutków. Jeśli z naruszenia dobra osobistego wynikła szkoda, można dochodzić jej naprawienia na zasadach ogólnych t.j. na zasadach określonych przez art.415 i następne k.c. (art.24§2 k.c.). Dochodzenie naprawienia szkody na podstawie art.415, a więc wskazanie na tzw. deliktową odpowiedzialność przetwarzającego jest znacznie mniej dogodne dla zainteresowanego przetwarzaniem

niż dochodzenie tego na podstawie art.471 k.c., a więc niż powołanie się na tzw. kontraktową odpowiedzialność przetwarzającego. Zasadnicze trudności jakie zainteresowany może w tym przypadku napotkać polegają na tym, że to jego obciąża dowód winy przetwarzającego. Proces stosowania narzędzi informatycznych jest procesem złożonym, w którym uczestniczy wiele osób i funkcjonuje wiele elementów. Ocena prawidłowości i rzetelności działania tych osób, a także ocena jakości użytych narzędzi (dobrze działających urządzeń, poprawnych programów) leży więc na ogół poza możliwościami przeciętnej osoby. Można stwierdzić więc, że w przypadkach, w których zainteresowany przetwarzaniem nie ma wpływu na fakt i sposób stosowania narzędzi informatycznych, ochrona udzielana mu przez prawo cywilne może dla niego być de lege lata niewystarczająca. Można rozważyć możliwość ukształtowania odpowiedzialności przetwarzających za szkody wynikłe ze stosowania narzędzi informatycznych na tzw. zasadzie ryzyka. Przykładem ustanowienia takiej odpowiedzialności jest art.435§1 k.c., na mocy którego "prowadzący na własny rachunek przedsiębiorstwo lub zakład wprawiany w ruch za pomocą sił przyrody (pary, gazu, elektryczności, paliw płynnych itp.) ponosi odpowiedzialność za szkodę na osobie lub mieniu wyrządzoną komukolwiek przez ruch przedsiębiorstwa lub zakładu chyba, że szkoda nastąpiła wskutek siły wyższej albo wyłącznie z winy poszkodowanego lub osoby trzeciej". Poszkodowany nie musi w tym przypadku dowodzić niczyjej winy co w istotny sposób wzmacnia jego pozycję. Ustanowienie przepisu określającego na zasadzie ryzyka odpowiedzialność przetwarzających za szkody wynikłe z nierzetelnego stosowania stosowania narzędzi informatycznych wzmocniłoby w zasadniczym stopniu podmiotów zainteresowanych przetwarzaniem w sytuacjach, w których nie mają oni wpływu na fakt i sposób tego stosowania. Odpowiedzialność ta mogłaby być wyłączona przez wykazanie po stronie przetwarzającego, że dopełnił on określonych szczególnych obowiązków np. w zakresie sprawdzenia niezawodności sprzętu, weryfikacji użytego oprogramowania, prawidłowo prowadzonej eksploatacji itp..

Należy mieć na uwadze, że w stosunkach pomiędzy przetwarzającym - organem administracji państwowej, a zainteresowanym - obywatelem sytuację prawną tego ostatniego reguluje ponadto ko-

deks postępowania administracyjnego (Dz.U. 1980 r, nr.9, poz.26). Stronie postępowania administracyjnego służy roszczenie o odszkodowanie (art.153§1 i 160 k.p.a.) w przypadku poniesienia szkody na skutek wydania niewłaściwej decyzji przez organ administracji państwowej lub na skutek uchylenia takiej decyzji. Przepisy te mogłyby mieć zastosowanie w przypadkach, w których decyzja taka wydana została w oparciu o dane uzyskane w wyniku nierzetelnego stosowania systemu informatycznego lub stosowania systemu nierzetelnie opracowanego. Po pierwsze jednak, przy literalnym stosowaniu k.p.a. mogą zaistnieć wątpliwości co do ustalenia praw strony w zakresie wglądu w proces stosowania systemu informatycznego, a po drugie - roszczenie o odszkodowanie następuje w oparciu o przepisy kodeksu cywilnego, wymagającego od zainteresowanego dowodu winy przetwarzającego.

Podobna jest sytuacja zainteresowanego przetwarzaniem, które go co prawda łączy z przetwarzającym stosunek umowny, lecz z uwagi na faktyczny brak równorzędności stron umowy sfera możliwości wpływu zainteresowanego na kształt jej postanowień jest praktycznie żadna. W sytuacji takiej będą np. kontrahenci dużych przedsiębiorstw użyteczności publicznej oferujących zawarcie tzw. umów adhezyjnych (PKP, PPTiT, PZU itp.)

Jak pisze J.Starościak [9] w przypadkach, w których jakieś zjawisko "przekracza ramy sytuacji jednostkowej" powstają zwykle przepisy prawa administracyjnego, określające obowiązki niektórych podmiotów uczestniczących w takim zjawisku w sposób bezwzględny i umożliwiające organom państwowym kontrolę ich dopełnienia (np. ustawa o normalizacji, prawo o ruchu drogowym, prawo budowlane, ustawa o zawodzie lekarza itp.) niezależnie od tego, że prawo cywilne określa odpowiedzialność każdej osoby za szkody wyrządzone z jej winy. Przepisy takie dotyczą na ogół zagadnień związanych z

- wymaganiami dotyczącymi narzędzi czy urządzeń dopuszczonych do obrotu publicznego oraz ze sposobem ich kontroli,
- kwalifikacjami osób prowadzących określoną działalność i z trybem ich stwierdzania,
- ewidencją przebiegu niektórych procesów w celu ew. później-

szego ustalenia przyczyn powstania nieprawidłowości w działaniu,
- inne szczególne obowiązki osób prowadzących określoną działalność

i t.p..

W przypadkach, w których przetwarzającym jest organ administracji państwowej lub duża jednostka użyteczności publicznej, możemy mówić o sytuacji, w której zjawisko przekracza wspomniane ramy sytuacji jednostkowej. Dla przypadków takich pożądana wydaje się regulacja administracyjnoprawna przynajmniej następujących kwestii
- przesłanek i zasad ogólnych stosowania narzędzi informatycznych,
- szczególnych obowiązków przetwarzającego w zakresie wykorzystania właściwego sprzętu i oprogramowania, zapewnienia prawidłowego przebiegu procesu zbierania danych, ich przesyłania, przetwarzania itd., ochrony danych w toku tego procesu, zapewnienia właściwej jego obsługi,
- szczególnych uprawnień zainteresowanego w zakresie dostępu do danych jego dotyczących, do ewidencji wyżej wspomnianego procesu oraz do informacji na temat użytego sprzętu i oprogramowania w zakresie, który jego dotyczy.

Przynajmniej dla przypadków, w których przetwarzającym jest organ administracji państwowej, a zainteresowanymi - obywatele, uregulowanie takie powinno mieć rangę ustawy. Rozwiązanie takie byłoby zgodne z panującym w doktrynie przekonaniem, że zgodnie z zasadą praworządności sfera praw i obowiązków obywateli stanowi przedmiot tzw. materii ustawowej. W szczególności podstawę ustawową powinny mieć wszelkie systemy informatyczne stosowane przez organy administracji państwowej w celu ewidencji ludności. Zagadnieniom tym poświęcona była obszerna praca I.Lipowicz [10].

BIBLIOGRAFIA

1. J.Irlik, T.Mazurkiewicz, R.Pessel, J.Waluszewski, Prawno-organizacyjna sytuacja informatyki w Polsce. Przegląd i omówienie zagadnień. Raport PTI, Warszawa 1983,
2. Z.Ziembiński, Teoria prawa, PWN W-wa 1978,
3. S.Ossowski, O osobliwościach nauk społecznych, PWN W-wa 1983,
4. A.Stelmachowski, Wstęp do teorii prawa cywilnego, PWN W-wa, 1984,
5. J.Waluszewski, Programy dla maszyn cyfrowych jako przedmiot prawa, Zeszyty Naukowe UJ, 1975, PWOWI z.5; Sytuacja prawna twórców programów dla maszyn cyfrowych, Państwo i Prawo, 1978 r, nr.6,
6. B.Czachórska, Umowy w zakresie informatyki i ochrona programów komputerowych, Ossolineum W-wa 1980,
7. B.Gawlik, Umowa know-how. Zagadnienia konstrukcyjne, Zeszyty Naukowe UJ, 1974, PWOWI z.3,
8. Cz.Żuławska, Ewolucja odpowiedzialności za jakość świadczenia (w:) Tendencje rozwojowe prawa cywilnego, Ossolineum W-wa 1983,
9. J.Starościak, Prawo administracyjne, PWN W-wa 1977,
10. I.Lipowicz, Sytuacja prawna systemu informatycznego w świetle prawa administracyjnego, Praca doktorska, Uniwersytet Śląski, Katowice 1981

Jesienna Szkoła PTI
Rydzyna, październik 1984

ZAGADNIENIA WSPOŁBIEŻNOSCI W PROGRAMOWANIU

Antoni Mazurkiewicz
Instytut Podstaw Informatyki
Polskiej Akademii Nauk
PKiN skr. poczt. 22
00-901 Warszawa

1. Wstęp.

Zasadniczą trudnością w programowaniu i analizie systemów współbieżnych jest konieczność uwzględniania niezależności pewnych akcji systemu. Niezależność ta pozbawia nas w praktyce możliwości korzystania z tak ważnego i wygodnego pojęcia, jakim jest pojęcie stanu globalnego systemu, to jest wartości wszystkich zmiennych i stanu sterowania w rozważanym momencie działania systemu. Podczas gdy w programach sekwencyjnych po wykonaniu jakiejś instrukcji stan programu jest jednoznacznie określony, w programie współbieżnym musimy się liczyć w takim przypadku z możliwością działania innych, niezależnych instrukcji, które mogą stan taki zmienić. Próba rozważenia wszystkich możliwych wariantów wykonania lub nie wykonania wzajemnie niezależnych instrukcji jest w praktyce skazana na niepowodzenie; liczba wariantów w realnie istniejących systemach jest astronomiczna.

Inną, również istotną cechą programowania współbieżnego jest praktyczna niewykonalność testowania programów. W sprawdzaniu pro-

gramów sekwencyjnych czas wykonania instrukcji nie ma znaczenia; w przypadku programów współbieżnych różnice w czasach wykonania pewnych instrukcji (nie do uniknięcia w praktyce) mogą spowodować różniące się wzajemnie wyniki testowania tego samego, nie zmienionego programu. Nie jesteśmy w stanie stworzyć w czasie testowania takich samych warunków działania programu, jakie będzie on miał przy dalszej eksploatacji, właśnie ze względu na odmienność i różnorodność warunków czasowych. Wynika stąd ważny wniosek: jedyną metodą stwierdzenia poprawności programów współbieżnych jest ich logiczna analiza. Tej właśnie analizie poświęcone są następne rozdziały.

W świetle powyższych uwag wydawać by się mogło, że należy unikać programowania współbieżnego i ograniczyć się do tworzenia jedynie programów sekwencyjnych. Istnieją jednak dwa powody, dla których jesteśmy skazani na programowanie współbieżne: komputeryzacja dużych systemów działających we współbieżnym środowisku (rezerwacja miejsc lotniczych, bazy danych, systemy telekomunikacji) oraz, w mikroskali, jedyna możliwość przyspieszenia działania systemów cyfrowych (wobec fizycznych granic miniaturyzacji).

2. Przykład programu współbieżnego.

Weźmy pod uwagę system utworzony z czytnika, kodera i drukarki, którego zadaniem jest czytanie kolejnych jednostek informacji, kodowanie ich, a następnie kolejne drukowanie. Koder współpracuje z drukarką za pośrednictwem bufora o pojemności N; ma to na celu wyrównanie możliwych niezgodności czasowych między przekazywaniem jednostek przez koder a odbieraniem ich przez drukarkę. Aby zmniejszyć czas działania systemu wymaga się, aby czynności czytania, kodowania i drukowania były od siebie niezależne, tzn. mogły wykonywać się współbieżnie.

Do opisu działania tego systemu i napisania programu potrzebne nam będzie pojęcie list i pewnych działań na nich. Przez listę rozumieć będziemy skończony lub nieskończony ciąg:

$$l = (l_1, l_2, l_3, \ldots)$$

którego elementy nazywamy elementami listy. Niech l będzie listą jak powyżej; przez p(l) oznaczamy początek listy l, a przez r(l) jej resztę:

$$p(l) = (l_1),$$
$$r(l) = (l_2, l_3, \ldots).$$

Początek listy pustej () (oznaczanej przez nil) jest nieokreślony, reszta listy pustej jest listą pustą, podobnie jak reszta listy jednoelementowej. Jeśli l jest listą skończoną, to liczbę jej elementów nazywamy długością l i oznaczamy przez d(l). Jeśli m jest listą skończoną, $m = (m_1, \ldots, m_k)$, to przez złożenie list m, l rozumiemy listę

$$(m_1, \ldots, m_k, l_1, l_2, \ldots)$$

oznaczaną przez m.l. Jeśli m, n są listami skończonymi, to ma miejsce łączność złożenia:

$$(n.m).l = n.(m.l).$$

Dla wszelkich list l, m, n zachodzą następujące fakty:

$l = p(l).r(l)$, jeśli $l \neq nil$,
$d(l) = 0 \Leftrightarrow l = nil$,
$r(nil) = nil$,
$m = n \Rightarrow m.l = n.l$,
$d(p(l)) = 1$,
$d(m.l) = d(m) + d(l)$, jeśli m i l są skończone.

Fakt, że l jest listą nieskończoną wyrażać będziemy symbolem Inf(l). Zachodzi następujące wynikanie:

$$Inf(l) \Rightarrow Inf(r(l)) \wedge l \neq nil.$$

Jeśli f jest funkcją określoną na elementach listy l, to przez $f^{\#}(l)$ rozumieć będziemy listę

$$(f(l_1), f(l_2), \ldots).$$

Program opisujący działanie rozważanego systemu będzie skonstruowany przy użyciu następujących zmiennych (rejestrów):

C, Xc, We (zmienne czytnika),
K, Xk (zmienne kodera),
B, Y (zmienne bufora),
D, Xd, Wy (zmienne drukarki).

W dowolnym momencie działania systemu wartościami powyższych zmiennych są:

We: lista (skończona lub nieskończona) znajdująca się w danym momencie na wejściu czytnika;
C: jedna z liczb 1, 2;
Xc: ostatnio przeczytany element listy wejściowej;
K: jedna z liczb 1, 2, 3, 4;
Xk: element listy wejściowej pobrany z rejestru Xc, już przekodowany lub jeszcze nie;
B: jedna z liczb 1, 2, 3;
Y: lista skończona o długości conajwyżej N, N jest stałą większą od zera;
D: jedna z liczb 1, 2, 3;
Xd: element pobrany z bufora i przeznaczony do wydrukowania;
Wy: lista skończona elementów zakodowanych i wydrukowanych.

Znaczenie liczb będących możliwymi wartościami zmiennych C, K, B i D jest następujące:

C = 1: czytnik jest gotów do przeczytania kolejnego elementu z listy wejściowej;
C = 2: kolejny przeczytany element jest w rejestrze czytnika gotowy do przekazania koderowi;
K = 1: koder gotów na przyjęcie nowego elementu do kodowania;
K = 2: kolejny element pobrany do kodowania;
K = 3: kolejny element zakodowany i gotowy do zapisania w buforze;
K = 4: koder ma dostęp do bufora;
B = 1: bufor gotów do współpracy z koderem lub z drukarką;
B = 2: bufor do dyspozycji kodera

B = 3: bufor do dyspozycji drukarki
D = 1: drukarka gotowa do pobrania kolejnego elementu z bufora do drukowania;
D = 2: drukarka ma dostęp do bufora;
D = 3: kolejny element z bufora pobrany do drukowania.

Bufor zorganizowany jest na zasadzie FIFO ("first in, first out"), tzn. elementy pobierane są z bufora przez drukarkę w tej samej kolejności, w jakiej były wpisywane do bufora przez koder. Ponadto, współpraca z buforem ma być zorganizowana w taki sposób, aby nie mogło nastąpić przepełnienie bufora (próba zapisania informacji w pełnym buforze) i niemożliwa była próba odczytu z pustego bufora. Funkcję kodującą oznaczymy przez f.

Program działania rozważanego systemu przedstawimy jako zbiór (a nie ciąg) instrukcji przypominających nieco instrukcje dozorowane Dijkstry, a także instrukcje warunkowe stosowane przez autora w tzw. zamkniętych systemach programowania. Oto program:

C = 1 ∧ We ≠ nil: (C,Xc,We) := (2,p(We),r(We)) (1)
C = 2 ∧ K = 1: (C,K,Xk) := (1,2,Xc) (2)
K = 2: (K,Xk) := (3,f(Xk)) (3)
K = 3 ∧ B = 1 ∧ d(Y) < N: (K,B) := (4,2) (4)
K = 4 ∧ B = 2: (K,B,Y) := (1,1,Y.Xk) (5)
D = 1 ∧ B = 1 ∧ d(Y) > 0: (D,B) := (2,3) (6)
D = 2 ∧ B = 3: (D,B,Xd,Y) := (3,1,p(Y),r(Y)) (7)
D = 3: (D,Wy) := (1,Wy.Xd) (8)

Znaczenie podanych wyżej instrukcji jest następujące:

(1) Jeśli czytnik jest gotów i lista wejściowa jest niepusta, to początek tej listy prześlij do rejestru Xc, za nową listę wejściową przyjmij resztę dotychczasowej listy i przygotuj czytnik do przekazania zawartości Xc koderowi;

(2)Jeśli czytnik jest gotów do przekazania informacji koderowi i koder jest gotów na przyjęcie nowego elementu do kodowania, prześlij zawartość Xc do Xk, przygotuj czytnik do czytania nowego elementu i przygotuj koder do kodowania;

(3) Jeśli koder jest gotów do kodowania, to zastąp zawartość rejestru kodera przez kod tej zawartości i zgłoś gotowość kodera do zapisania zakodowanej informacji w buforze;

(4) Jeśli bufor jest gotów do współpracy, koder gotów do przekazania informacji i bufor nie jest całkowicie wypełniony, to oddaj bufor do dyspozycji kodera;

(5) Jeśli bufor jest do dyspozycji kodera, to dopisz do zawartości bufora zawartość rejestru kodera, zgłoś gotowość kodera do przyjęcia nowego elementu do kodowania i gotowość bufora do współpracy z drukarką lub koderem;

(6) Jeśli bufor jest gotów do współpracy, drukarka gotowa do przyjęcia kolejnego elementu do drukowania i bufor nie jest całkowicie opróżniony, to oddaj bufor do dyspozycji drukarki;

(7) Jeśli bufor jest do dyspozycji drukarki, to pierwszy element listy zawartej w buforze wpisz do rejestru drukarki, usuń pierwszy element tej listy pozostawiając w buforze jej resztę, zgłoś gotowość bufora do dalszej współpracy i gotowość drukarki do drukowania;

(8) Jeśli drukarka gotowa do drukowania, wydrukuj zawartość jej rejestru i zgłoś gotowość drukarki do pobrania kolejnego elementu z bufora.

Zauważmy, że jest możliwa sytuacja, w której odbywa się jednocześnie czytanie, kodowanie i drukowanie; możliwość ta powstaje wówczas, gdy równocześnie spełnione są warunki $C = 1 \wedge We \neq nil$, $K = 2$, $D = 3$. Zauważmy również, że jest możliwa sytuacja, w której $B = 1$, $D = 1$, $K = 3$, $0 < d(Y) < N$, tzn. gdy bufor nie jest ani całkowicie wypełniony, ani opróżniony, lecz gotów do współpracy, i akces do tej współpracy zgłosił równocześnie koder i drukarka. W tym przypadku z instrukcji (4) i (6), możliwych do wykonania, zostanie (niedeterministycznie) wybrana i wykonana tylko jedna (instrukcje te bowiem nie są niezależne, dotyczą mianowicie pewnych wspólnych rejestrów).

Ogólnie rzecz biorąc, w przyjętym powyżej języku programowania programy są skończonymi zbiorami instrukcji postaci

$$p: (X_1, \ldots, X_k) := (e_1, \ldots, e_k),$$

w których p jest warunkiem dotyczącym wartości pewnych zmiennych programu, $X_1, \ldots, X_k$ są pewnymi zmiennymi programu, zaś $e_1, \ldots, e_n$ wyrażeniami (termami), zbudowanymi ze zmiennych i działań oraz funkcji określonych w języku. Zbiór zmiennych występujących bądź to w warunku p, bądź w wyrażeniach $e_1, \ldots, e_k$, bądź wreszcie wśród $X_1, \ldots, X_k$, nazywamy zakresem instrukcji.

Wykonanie instrukcji powoduje zmianę stanu programu, to jest wartości pewnych zmiennych programu. Wykonanie instrukcji o podanej wyżej postaci jest możliwe jedynie wówczas, gdy wartości zmiennych występujących w warunku p spełniają ten warunek; w wyniku wykonania takiej instrukcji zmiennym $X_1, \ldots, X_k$ zostają przypisane wartości wyrażeń $e_1, \ldots, e_k$, odpowiednio, obliczone w stanie bezpośrednio poprzedzającym wykonanie tej instrukcji. Instrukcje o rozłącznych zakresach nie interferują wzajemnie i mogą być wykonywane niezależnie. Instrukcje o przecinających się zakresach wykonywane są w kolejności wynikającej ze spełnienia ich warunków, bądź w kolejności wybranej niedeterministycznie, jeśli ich warunki nie wykluczają się wzajemnie. Dwie instrukcje, jedna o warunku p, druga o warunku q, nazywamy

- współbieżnymi, jeśli ich zakresy są rozłączne;
- zgodnymi, jeśli ich zakresy nie są rozłączne, lecz warunki p i q wykluczają się wzajemnie;
- konfliktowymi, jeśli ich zakresy nie są rozłączne, a istnieją stany, w których spełnione jest zarówno p jak i q.

Tak więc kolejność wykonania instrukcji współbieżnych nie jest określona, instrukcji zgodnych jest określona przez stan zmiennych, zaś instrukcji konfliktowych jest określona albo przez stan zmiennych albo niedeterministycznie. Wykonanie programu polega na wykonywaniu jego instrukcji; program się zatrzymuje, gdy zostanie osiągnięty stan, w którym nie jest możliwe wykonanie żadnej instrukcji; w przypadku tym mówimy także, że nastąpiła blokada pro-

gramu, a osiągnięty stan nazywamy martwym. Jeśli taki stan nie może być osiągnięty, to program nazywamy żywotnym. Jeśli zostanie osiągnięty stan, wykluczający w dalszym działaniu programu wykonanie pewnego zbioru instrukcji, mówimy o częściowej blokadzie programu; program, w którym taka blokada nie może wystąpić, nazywamy silnie żywotnym. Jeśli jest możliwe takie działanie programu, w którym nieskończenie wiele razy wystąpi stan umożliwiający wykonanie pewnej instrukcji (lub grupy instrukcji), natomiast instrukcja ta (grupa instrukcji) nie zostanie nigdy wykonana, to mówimy o nieuczciwości programu względem tej instrukcji (grupy instrukcji).

Te i inne własności programu zależą od warunków nałożonych na stan początkowy programu. Ustalając te warunki można postawić następujące pytania odnośnie zachowania się programu dla dowolnego warunku q:

Czy istnieje takie wykonanie programu, w którym występuje stan, spełniający q?

Czy w każdym wykonaniu programu występuje stan, spełniający q?

Czy istnieje takie wykonanie programu, w którym każdy stan spełnia q?

Czy w każdym wykonaniu programu każdy stan spełnia q?

Liczba tych pytań może być zredukowana do dwóch, gdyż pierwsze z nich jest negacją czwartego dla warunku nie-q i podobnie drugie jest negacją trzeciego dla tegoż warunku. Znając metody pozwalające udzielać odpowiedzi na te podstawowe pytania można odpowiadać na pytania złożone, np.

Czy w każdym wykonaniu programu wystąpi stan, począwszy od którego istnieje takie wykonanie programu, w którym każdy stan spełnia q?

Jak zauważyliśmy wyżej, bylibyśmy bezradni wobec tych pytań bez

sięgnięcia do aparatu formalnego, opisanego w następnym rozdziale. Aparat ten zastosujemy do analizy przytoczonego programu; interesować nas będzie uzasadnienie następujących stwierdzeń:

(T1) Dla warunku początkowego $C = K = B = 1 \wedge Y = Wy = nil \wedge We = We_0 \wedge d(We_0) = m \geqslant 0$, w ktorym We_0 jest dowolną listą skończoną, m jest liczbą całkowitą, każde (pełne) wykonanie programu kończy się w stanie spełniającym warunek $Wy = f^{\#}(We_0)$ (tzn. lista wyjściowa jest zakodowaną listą wejściową).

(T2) Dla warunku początkowego $K = B = D = 1 \wedge Inf(We)$, każde (pełne) wykonanie programu jest nieskończone.

3. Aparat formalny.

Niech P będzie dowolnym programem, ustalonym na przeciąg tego rozdziału. Instrukcje tego programu utworzone są ze zmiennych, symboli funkcyjnych i relacyjnych oraz ze znaków pomocniczych, takich jak nawiasy, dwukropek, symbole operacji logicznych, itp. Niech V będzie zbiorem zmiennych programu P. Każdej zmiennej v z V przyporządkowany jest zbiór jej możliwych wartości oznaczany przez T(v) i zwany typem zmiennej v. Jeśli każdej zmiennej z V przyporządkowana jest pewna wartość z jej typu, to powiemy, że określony jest pewien stan programu. Tak więc stanem jest każda funkcja s określona na V i taka, że $s(v) \in T(v)$ dla każdej zmiennej $v \in V$. Jeśli dany jest stan s programu, to mając określone znaczenia symboli funkcyjnych i relacyjnych, określone są także wartości wyrażeń i warunków zawierających zmienne programu (znaczenie symboli pomocniczych jest ustalone). Jeśli e jest takim wyrażeniem, a p warunkiem, to ich wartość w stanie s będzie oznaczana przez e(s), p(s), odpowiednio. Jeśli wartość p(s) = true (true i false będą oznaczać, jak zwykle, wartości logiczne prawda i fałsz), to mówimy, że p jest spełnione w stanie s. Jeśli warunek p jest spełniony w każdym stanie programu, to będziemy pisać !p.

Z programem P związane są trzy warunki: S, Z, i M; S(s) oznacza, że s jest stanem P, Z(s) - że conajmniej jedna instrukcja może być wykonana w stanie s, M(s) - że żadna instrukcja nie może być wykonana w stanie s; inaczej mówiąc, Z jest spełniona w stanach żywych i tylko w nich, M jest spełniona w stanach martwych i tylko w nich. Formalnie,

$$S(s) \Leftrightarrow \forall\, v \in V\colon s(v) \in T(v),$$
$$Z(s) \Leftrightarrow p_1(s) \vee p_2(s) \vee \ldots \vee p_K(s),$$

gdzie $p_1, p_2, \ldots, p_K$ są warunkami wszystkich instrukcji programu P,

$$M(s) \Leftrightarrow S(s) \wedge \neg Z(s).$$

Dla rozważanego przez nas programu mamy:

$$S \Leftrightarrow \mathbb{L}(We) \wedge \mathbb{N}(C) \wedge \mathbb{E}(Xc) \wedge \mathbb{N}(K) \wedge \mathbb{E}(Xk) \wedge \mathbb{N}(B) \wedge \mathbb{L}(Y)$$
$$\mathbb{N}(D) \wedge \mathbb{E}(Xd) \wedge \mathbb{L}(Wy) \wedge d(Y) \leqslant N \wedge C \leqslant 2 \wedge K \leqslant 4 \wedge$$
$$B \leqslant 3 \wedge D \leqslant 3,$$

gdzie $\mathbb{L}(x)$, $\mathbb{N}(x)$, $\mathbb{E}(x)$ oznaczają odpowiednio: x jest listą, x jest liczbą naturalną, x jest elementem listy. Warunek S jest użyteczny w rozumowaniach typu "skoro nie jest C = 1, musi być C = 2" i podobnych.

$$Z \Leftrightarrow C = 1 \wedge We \neq nil \vee C = 2 \wedge K = 1 \vee K = 2 \vee K = 3 \wedge$$
$$B = 1 \wedge d(Y) < N \vee K = 4 \wedge B = 2 \vee D = 1 \wedge B = 1 \wedge$$
$$d(Y) > 0 \vee D = 2 \wedge B = 3 \vee D = 3. \qquad (Z)$$

Stąd i z formuły S dostajemy po formalnych przekształceniach

$$M \Leftrightarrow C = 1 \wedge We = nil \wedge K = 1 \wedge (B = 1 \wedge (d(Y) = 0 \wedge D = 1 \vee$$
$$D = 2) \vee B = 2 \wedge D \neq 3 \vee B = 3 \wedge D = 1) \vee (C = 2 \vee$$
$$C = 1 \wedge We = nil) \wedge K = 3 \wedge (B = 1 \wedge d(Y) = N \wedge D = 2 \vee$$
$$B = 2 \wedge D \neq 3 \vee B = 3 \wedge D = 1). \qquad (M)$$

Niech s będzie stanem, $v_1, v_2, \ldots, v_k$ pewnymi zmiennymi z V, $u_1, u_2, \ldots, u_k$ pewnymi wartościami z $T(v_1), T(v_2), \ldots, T(v_k)$, odpowiednio. Przez

$$s(u_1/v_1,\ u_2/v_2,\ \ldots,\ u_k/v_k)$$

oznaczamy taki stan s_1, że $s_1(v_i) = u_i$ dla $i = 1, 2, \ldots, k$, i $s_1(v) = s(v)$ dla $v \in V - \{v_1, v_2, \ldots, v_k\}$. Inaczej mówiąc, s_1 powstaje ze stanu s przez zastąpienie w nim wartości zmiennych $v_1, \ldots, v_k$ przez wartości $u_1, \ldots, u_k$. Niech r,

$$r = (p:\ (v_1, \ldots, v_k) := (e_1, \ldots, e_k))$$

będzie dowolną instrukcją programu; wówczas przez $\hat{r}(s_1,s_2)$ będziemy rozumieć warunek

$$p(s_1) \wedge s_2 = s_1(e_1(s_1)/v_1, \ldots, e_k(s_1)/v_k).$$

Tak więc $\hat{r}(s_1,s_2)$ wyraża fakt, że instrukcja r przeprowadza stan s_1 w stan s_2. Ciąg stanów

$$(s_0, s_1, \ldots, s_n, \ldots), \quad n \geqslant 0,$$

skończony lub nieskończony, nazywać będziemy wykonaniem programu, jeśli dla każdego i, $i \geqslant 1$, istnieje instrukcja r programu taka, że $\hat{r}(s_{i-1},s_i)$. Wykonaniem pełnym programu nazywamy albo wykonanie nieskończone, albo takie wykonanie skończone

$$(s_0, \ldots, s_n),$$

że $M(s_n)$ (s_n jest stanem martwym). Jest jasne, że każdy stan s_i należący do wykonania nieskończonego jest żywy, tj. $L(s_i)$ dla każdego $i \geqslant 0$. Wykonanie programu jest przedłużalne, jeśli jego oststni element jest żywy; tak więc wykonania pełne są dokładnie wykonaniami nie przedłużalnymi. Wykonanie, którego początkowy element spełnia pewien warunek Q nazywamy wykonaniem od Q.

Niech Q będzie pewnym warunkiem. Powiemy, że program jest:

- nieśmiertelny dla Q, jeśli każde jego pełne wykonanie od Q jest nieskończone, tj. gdy każde jego wykonanie skończone od Q jest przedłużalne;

- rozbieżny dla Q, jeśli istnieje jego wykonanie nieskończone od Q;

- śmiertelny dla Q, gdy istnieje jego pełne skończone wykonanie od Q;

- zbieżny dla Q, gdy każde jego wykonanie pełne od Q jest skończone.

Niech Q_0, Q będą pewnymi warunkami. Powiemy, że Q jest:

- niezmiennikiem dla Q_0, jeśli każdy element dowolnego wykonania od Q_0 spełnia Q;

- osiągalny z Q_0, jeśli istnieje wykonanie od Q_0 zawierające element spełniający Q;

- nieuchronny dla Q_0, jeśli w każdym pełnym wykonaniu od Q_0 istnieje element spełniający Q;

- trwały dla Q_0, jeśli istnieje pełne wykonanie od Q_0, którego każdy element spełnia Q.

Niezmienniczość, osiągalność, nieuchronność i trwałość warunków są podstawowymi własnościami programu. Nieśmiertelność, śmiertelność, zbieżność i rozbieżność mogą być wysłowione zapomocą tych pierwszych: nieśmiertelność dla Q jest równoważna niezmienniczości warunku Z dla Q, rozbieżność dla Q jest równoważna trwałości Z dla Q, śmiertelność - osiągalności M z Q, zbieżność - nieuchronności M dla Q. Mówiąc potocznie, niezmienniczość służy do stwierdzenia, że coś nie może nigdy zdarzyć się podczas wykonywania programu, osiągalność - że coś może się zdarzyć, nieuchronność - że coś napewno się zdarzy i trwałość - że coś może nigdy się nie zdarzyć.

Dowodzenie osiągalności i trwałości polega naogół na skonstruowaniu wykonania spełniającego żądane własności; zagadnieniami tymi nie będziemy się tutaj zajmować, a ograniczymy się do niezmienniczości i nieuchronności.

Niech Q będzie warunkiem i r instrukcją. Symbolem Qr oznaczymy warunek określony następująco:

$$(Qr)(s) = \exists s_0: Q(s_0) \wedge \hat{r}(s_0, s),$$

dla każdego stanu s. Jeśli $(r_1, \ldots, r_n)$, $n \geq 0$, jest ciągiem instrukcji, to kładziemy

$$(Qr_1 \ldots r_n)(s) \Leftrightarrow (\ldots(Qr_1)\ldots r_n)(s),$$

przyczym $Q() \Leftrightarrow Q$. Dla dowolnego warunku Q, wyrażeń $e_1, \ldots, e_k$, i zmiennych $v_1, \ldots, v_k$, $k > 0$, niech

$$Q(e_1/v_1, \ldots, e_k/v_k)$$

oznacza warunek, w którym za zmienne $v_1, \ldots, v_k$ podstawiono wyrażenia $e_1, \ldots, e_k$, odpowiednio. Niech $r = (p: (v_1, \ldots, v_k) := (e_1, \ldots, e_k))$ będzie dowolną instrukcją; od tej chwili zakładać będziemy, że dla każdego stanu s spełniającego p mamy $e_i(s) \in T(v_i)$, tj. że zachodzi zgodność typów wyrażeń e_i i zmiennych v_i, $i = 1, \ldots, k$. Następujący lemmat pozwala wyznaczać warunek Qr dla dowolnego warunku Q.

LEMMAT 1. Dla każdych warunków Q_1, Q_2 następujące implikacje są równoważne:

(a) $!(Q_1 \wedge p \Rightarrow Q_2(e_1/v_1, \ldots, e_k/v_k))$,
(b) $!(Q_1 r \Rightarrow Q_2)$.

Dowód. Bez zmniejszania ogólności rozważań możemy przyjąć $k = 1$, to jest ograniczyć się do instrukcji $r = (p: (v) := (e))$. Mamy w tym przypadku

$$\begin{aligned}
&\forall s: (Q_1 \wedge p \Rightarrow Q_2(e/v))(s) \\
&\Leftrightarrow \forall s: (Q_1(s) \wedge p(s) \Rightarrow Q_2(s(e(s)/v))) \\
&\Leftrightarrow \forall s, s_1: (Q_1(s) \wedge p(s) \wedge s_1 = s(e(s)/v) \Rightarrow Q_2(s_1)) \\
&\Leftrightarrow \forall s_1: (\exists s: (Q_1(s) \wedge p(s) \wedge s_1 = s(e(s)/v)) \Rightarrow Q_2(s_1) \\
&\Leftrightarrow \forall s_1: (\exists s: (Q_1(s) \wedge \hat{r}(s, s_1)) \Rightarrow Q_2(s_1) \\
&\Leftrightarrow \forall s_1: (Q_1 r \Rightarrow Q_2)(s_1).
\end{aligned}$$

Koniec dowodu.

Następujące twierdzenie pozwala dowodzić niezmienniczości pewnych warunków. Niech Q_0, Q_1, Q będą dowolnymi warunkami.

TWIERDZENIE 1. Jeśli zachodzą następujące implikacje:

(a) $!(Q_0 \Rightarrow Q)$,
(b) $!(Qr \Rightarrow Q)$, dla każdej instrukcji r programu,
(c) $!(Q \Rightarrow Q_1)$,

to Q_1 jest niezmiennikiem dla Q_0.

Dowód. Załóżmy (a), (b) i (c). Niech $(s_0, \ldots, s_n, \ldots)$ będzie dowolnym wykonaniem programu. Istnieje zatem ciąg instrukcji $(r_1, \ldots, r_n, \ldots)$ programu taki, że $f_i(s_{i-1}, s_i)$ dla każdego $i \geqslant 1$. Niech $Q_0(s_0)$; wówczas na mocy warunku (a) dostajemy $Q(s_0)$. Z uwagi na (b) mamy $Q(s_{i-1}) \wedge f_i(s_{i-1}, s_i) \Rightarrow Q(s_i)$ dla każdego i 1; stąd przez indukcję wnosimy, że $Q(s_n)$ dla każdego $n \geqslant 0$. Z (c) mamy $Q(s_n) \Rightarrow Q_1(s_n)$, a zatem Q_1 jest niezmiennikiem dla Q_0. Koniec dowodu.

Zapomocą twierdzenia 1 wykażemy niezmienniczość warunku $(K = 4 \Leftrightarrow B = 2) \wedge (D = 2 \Leftrightarrow B = 3)$ dla warunku $C = K = B = D = 1$ w rozważanym przez nas programie. Mamy oczywiście

$$(C = K = B = D = 1) \Rightarrow ((K = 4 \Leftrightarrow B = 2) \wedge (D = 2 \Leftrightarrow B = 3))$$

dla każdego stanu programu. Mamy ponadto

$$((K = 4 \Leftrightarrow B = 2) \wedge (D = 2 \Leftrightarrow B = 3))r = ((K = 4 \Leftrightarrow B = 2) \wedge (D = 2 \Leftrightarrow B = 3))$$

dla każdej instrukcji r rozważanego programu. Możemy się o tym przekonać korzystając z lemmatu 1: dla instrukcji (1) - (4), (6) i (8) poprzednik implikacji (a) lemmatu jest fałszywy, a zatem cała implikacja jest prawdziwa; dla instrukcji (5) implikacja (a) przybiera postać

$$(K = 4 \Leftrightarrow B = 2) \wedge D \neq 2 \Rightarrow (1 = 4 \Leftrightarrow 1 = 2) (D = 2 \Leftrightarrow 1 = 3),$$

a więc jest implikacją prawdziwą w każdym stanie; podobnie dla instrukcji (7) mamy

$$K\neq 4 \wedge (D=2 \Leftrightarrow B=3) \Rightarrow (K=4 \Leftrightarrow 1=2) \wedge (3=2 \Leftrightarrow 1=3),$$

a więc też implikację spełnioną w każdym stanie. Na mocy twierdzenia 1 wnosimy, że $(K=4 \Leftrightarrow B=2) \wedge (D=2 \Leftrightarrow B=3)$ jest niezmiennikiem dla $^{A}C=K=B=D=1$.

Powyższa metoda dowodzenia własności niezmienniczych programów, identyczna z używaną w programach sekwencyjnych, wymaga wzbogacenia w przypadku programów współbieżnych, gdyż próba jej bezpośredniego zastosowania do udowodnienia bardziej złożonych niezmienników byłyby praktycznie uniemożliwiona przez wielką liczbę przypadków koniecznych do rozpatrzenia. Następujące twierdzenie, będąc uogólnieniem twierdzenia 1, daje takie wzbogacenie. Niech T będzie dowolnym zbiorem i dla każdego $t \in T$ niech $Q(t)$ będzie warunkiem. Niech ponadto Q_0, Q_1 będą warunkami.

TWIERDZENIE 2. Jeśli zachodzą następujące implikacje:

(a) Istnieje takie $t_0 \in T$, że $!(Q_0 \Rightarrow Q(t_0))$;
(b) Dla każdej instrukcji r programu i każdego $t \in T$ istnieje takie $t_1 \in T$, że $!(Q(t)r \Rightarrow Q(t_1))$;
(c) Dla każdego $t \in T$: $!(Q(t) \Rightarrow Q_1)$,

wówczas Q_1 jest niezmiennikiem dla Q_0.

Dowód. Wykażemy najpierw, że warunek $\exists\, t \in T: Q(t)$ spełnia założenia twierdzenia 1 jako warunek Q. Istotnie, z (a) dostajemy $!(Q_0 \Rightarrow \exists\, t \in T: Q(t))$; z (c) mamy $!((\exists\, t \in T: Q(t)) \Rightarrow Q_1)$; warunek (b) jest zaś równoważny warunkowi

$$!((\exists\, t \in T: Q(t)r) \Rightarrow (\exists\, t \in T: Q(t)));$$

Z definicji warunku Qr, przez przestawienie kwantyfikatorów, dostajemy

$$!((\exists\, t \in T: Q(t))r \Rightarrow (\exists\, t \in T: Q(t))).$$

Tak więc, podstawiając w twierdzeniu 1 $\exists t \in T: Q(t)$ za Q, otrzymujemy tezę twierdzenia 2. Koniec dowodu.

Ponieważ T jest dowolnym zbiorem, możemy przyjąć $T = T_1 \times T_2 \times \ldots \times T_m$ dla dowolnych zbiorów $T_1, T_2, \ldots, T_m$, $m \geqslant 1$, i zamiast $Q(t)$ rozważać $Q(t_t, t_2, \ldots, t_m)$; twierdzenie oczywiście pozostaje w mocy. Zastosujemy je do wykazania, że w rozważanym programie, jeśli w pewnym stanie spełniony jest warunek

$$C=K=B=D=1 \wedge We = We_0 \wedge Wy=Y= nil, \qquad (A)$$

to w każdym stanie następnym spełniającym

$$C=K=B=D=1 \wedge We=Y= nil \qquad (B)$$

spełniony jest warunek $Wy = f^{\#}(We_0)$. W tym celu wykażemy, że następujący warunek (C) spełnia dla każdego stanu implikację $(C)r \Rightarrow (C)$. A oto warunek (C):

$$\begin{aligned}
&(C=1 \wedge t_1.f^{\#}(We)=f^{\#}(We_0) \vee \\
&\ C=2 \wedge t_1.f(Xc).f^{\#}(We)=f^{\#}(We_0)) \wedge \\
&(K=1 \wedge t_2=t_1 \vee \\
&\ K=2 \wedge t_2.f(Xk)=t_1 \vee \\
&\ K=3 \wedge t_2.Xk=t_1 \vee \\
&\ K=4 \wedge t_2.Xk=t_1) \wedge \\
&(B=1 \wedge 0 \leqslant d(Y) \leqslant N \wedge t_2=t_3.Y \vee \\
&\ B=2 \wedge 0 \leqslant d(Y) < N \wedge t_2=t_3.Y \vee \\
&\ B=3 \wedge 0 < d(Y) \leqslant N \wedge t_2=t_3.Y) \wedge \\
&(D=1 \wedge t_3=Wy \vee \\
&\ D=2 \wedge t_3=Wy \vee \\
&\ D=3 \wedge t_3=Wy.Xd).
\end{aligned} \qquad (C)$$

Nim przystąpimy do wykazania prawdziwości interesującej nas implikacji zauważmy, że warunek (C) jest zależny od trzech parametrów t_1, t_2, t_3 przebiegających listy. Zauważmy też, że parametry te posiadają przejrzystą interpretację: na dowolnym etapie działania programu, w którym C=K=D=1, t_1 jest listą dotychczas przeczytaną i zakodowaną, t_2 jest tą samą listą, składa się ona z listy już wydrukowanej t_3 i zawartości bufora Y.

Warunek (C) jet koniunkcją czterech warunków

$$Q_1(C,t_1,We,Xc) \wedge Q_2(K,t_1,t_2,Xk) \wedge Q_3(B,Y,t_2,t_3) \wedge Q_4(D,t_3,Wy,Xd).$$

Niech r_1 oznacza instrukcję (1) rozważanego programu. Wykażemy, że $(C)r_1 \Rightarrow (C)$ dla każdego stanu programu i każdego t_1,t_2,t_3. Ponieważ zmienne występujące w r_1 nie wchodzą do Q_2 ani Q_3, ani Q_4, zatem

$$(Q_1 \wedge Q_2 \wedge Q_3 \wedge Q_4)r_1 \Leftrightarrow (Q_1 r_1 \wedge Q_2 \wedge Q_3 \wedge Q_4)$$

Wystarczy więc pokazać, że

$Q_1 r_1 \Rightarrow Q_1$ dla każdego stanu programu i każdego t_1.

Korzystając z lemmatu 1 wystarczy sprawdzić, czy

$$Q_1(C,t_1,We,Xc) \wedge C=1 \wedge We \neq nil \Rightarrow Q_1(2/C,p(We)/Xc,r(We)/We)$$

to znaczy, czy

$$Q_1(C,t_1,We,Xc) \wedge C=1 \wedge We \neq nil \Rightarrow t_1.f(p(We)).f^{*}(r(We))=f^{*}(We_0).$$

Następnik tej implikacji jest równoważny warunkowi

$$t_1.f^{*}(We) = f^{*}(We_0),$$

który oczywiście wynika z Q_1.

Rozważmy teraz instrukcję (2) programu i oznaczmy ją przez r_2. W instrukcji tej występują zmienne C, K, Xc, Xk, zatem

$$(Q_1 \wedge Q_2 \wedge Q_3 \wedge Q_4)r_2 \Leftrightarrow (Q_1 \wedge Q_2)r_2 \wedge (Q_3 \wedge Q_4).$$

Wykażemy, że

$$(Q_1 \wedge Q_2)r_2 \Rightarrow (Q_1 \wedge Q_2)(t_1.f(Xc)/t_1)$$

dla każdego t_1, t_2. Korzystając, jak poprzednio z lemmatu 1 sprawdzamy prawdziwość implikacji

$$Q_1 \wedge Q_2 \wedge C=2 \wedge K=1 \Rightarrow (Q_1 \wedge Q_2)(1/C, 2/K, Xc/Xk, t_1.f(Xc)/t_1).$$

Poprzednik tej implikacji jest równoważny warunkowi

$$C=2 \wedge K=1 \wedge t_1.f(Xc).f^{*}(We)=f^{*}(We_0) \wedge t_2=t_1,$$

zaś jej następnik warunkowi

$$1=1 \wedge 2=2 \wedge t_1.f(Xc).f^{*}(We)=f^{*}(We_0) \wedge t_2.f(Xc) = t_1.f(Xc)$$

czyli

$$t_1.f(Xc).f^{*}(We)=f^{*}(We_0) \wedge t_2=t_1,$$

który to warunek wynika z poprzednika w oczywisty sposób.

Implikację $(C)r \Rightarrow (C)$ sprawdzamy w podobny sposób dla pozostałych instrukcji. Mamy oczywiście

$$(A) \Rightarrow (C)(nil/t_1, nil/t_2, nil/t_3).$$

Podstawiając w twierdzeniu 2 (A) za Q_0, (C) za Q, (C) za Q_1 stwierdzamy, że warunek (C) zachodzi dla wszystkich stanów dowolnych wykonań programu przy odpowiednio dobranych t_1, t_2 i t_3. Jeśli więc pewien stan następujący po (A) spełnia warunek (B), to musi być (B) $\wedge$ (C), czyli

$$C=1 \wedge t_1.nil = f^{*}(We_0) \wedge K=1 \wedge t_2=t_1 \wedge B=1 \wedge t_2=t_3.nil \wedge D=1 \wedge t_3=Wy$$

$$\Leftrightarrow C=K=B=D=1 \wedge f^{*}(We_0)=t_1=t_2=t_3=Wy,$$

skąd wynika $Wy = f^{*}(We_0)$.

Wykazaliśmy więc, że dla każdego wykonania programu od warunku $C=K=B=D=1 \wedge Wy=Y=nil \wedge We = We_0$ każdy stan tego wykonania spełniający warunek $C=K=B=D=1 \wedge We =Y=nil$ spełnia też warunek $Wy = f^{*}(We_0)$.

Nic jednak, jak dotychczas, nie gwarantuje nam, że taki stan będzie osiągnięty. Naszym celem jest obecnie wykazanie, że tak jest istotnie, tzn. że warunek $C=K=B=D=1 \wedge We=Y=nil$ jest nieuchronny dla $C=K=B=D=1 \wedge We=We_0 \wedge d(We_0) = m \wedge Y=nil$, gdzie m jest liczbą całkowitą nieujemną. Skorzystamy w tym celu z następującego twierdzenia.

TWIERDZENIE 3. Jeśli $Q(t)$, $t = 0,1,2, \ldots$, są warunkami takimi, że:

(a) $!(Q_0 \Rightarrow Q(0))$,

(b) dla każdego $t \geq 0$ i każdej instrukcji r programu
$!(Q(t)r \Rightarrow Q(t+1))$,

(c) Istnieje liczba K taka, że dla każdego $t \geq 0$
$!(Q(t) \Rightarrow t < K)$,

to program jest zbieżny dla Q_0.

Dowód. Przypuśćmy, że istnieje wykonanie nieskończone od Q_0: $(s_0, s_1, \ldots, s_n, \ldots)$; istnieje więc ciąg instrukcji programu $(r_1, r_2, \ldots, r_K)$ taki, że $(Q_0 r_1 r_2 \ldots r_K)(s_K)$. Z uwagi na (a) mamy $(Q(0) r_1 r_2 \ldots r_K)(s_K)$. Z warunku (b) przez prostą indukcję wykazujemy, że $Q(0) r_1 r_2 \ldots r_K \Rightarrow Q(K)$, a z warunku (c) wynika, że $Q(K) \Rightarrow K < K$, tj. że $Q(K) \Rightarrow$ false; oznacza to, że $Q(K)$ nie może być spełnione przez żaden stan, co jest w sprzeczności z $(Q(0) r_1 r_2 \ldots r_K)(s_K)$. Tak więc każde wykonanie od Q_0 jest skończone. Koniec dowodu.

Dla wykazania żądanej nieuchronności konstruujemy warunki, o których mowa w twierdzeniu 3, w postaci

$$\exists\, t_1, \ldots, t_8 : t_1 \geq 0 \wedge \ldots\ t_8 \geq 0 \wedge Q(t_1, \ldots, t_8) \wedge t = t_1 + \ldots + t_8,$$

a własność (b) wykazujemy podobnie jak w poprzednim przypadku. Poniżej podajemy warunek Q; warto zauważyć, że parametr t_i, $i = 1, \ldots, 8$, odpowiada wykonaniu i-tej instrukcji rozważanego programu. Warunek Q wyraża ściśle intuicyjnie oczywiste związki między liczbami wykonań poszczególnych instrukcji programu. Korzystając z twierdzenia 3 nie będziemy podawać liczby K w postaci jawnej, lecz ograniczymy się tylko do wykazania jej istnienia.

Oznaczmy wyżej wymieniony warunek przez R(t); warunek Q ma postać:

$$
\begin{aligned}
&(C=1 \wedge t_1+d(We)=m \wedge t_1=t_2 \vee\\
&C=2 \wedge t_1+d(We)=m \wedge t_1=t_2+1)\wedge\\
&(K=1 \wedge t_2=t_3=t_4=t_5 \vee\\
&K=2 \wedge t_2=t_3+1=t_4+1=t_5+1 \vee\\
&K=3 \wedge t_2=t_3=t_4+1=t_5+1 \vee\\
&K=4 \wedge t_2=t_3=t_4=t_5+1)\wedge\\
&(B=1 \wedge t_4+t_6=t_5+t_7 \wedge t_5=t_7+d(Y) \wedge 0 \leqslant d(Y) \leqslant N \vee\\
&B=2 \wedge t_4+t_6=t_5+t_7+1 \wedge t_5=t_7+d(Y) \wedge 0 \leqslant d(Y) < N\\
&B=3 \wedge t_4+t_6=t_5+t_7+1 \wedge t_5=t_7+d(Y) \wedge 0 < d(Y) \leqslant N)\wedge\\
&(D=1 \wedge t_6=t_7=t_8 \vee\\
&D=2 \wedge t_6=t_7+1=t_8+1 \vee\\
&D=3 \wedge t_6=t_7=t_8+1).
\end{aligned}
$$

Podobnie jak w rozważanym wyżej przypadku dowodzimy, że dla i-tej instrukcji programu, oznaczmy ją przez r_i, mamy dla wszelkich stanów:

$$Qr_i \Rightarrow Q(t_i+1/t_i),$$

tak więc dla warunku R(t) dostajemy

$$
\begin{aligned}
Rr_i \Rightarrow\ & \exists\, t_1,..,t_8:\ t_1 \geqslant 0 \ \ldots\ t_8 \geqslant 0 \wedge Q(t_1,..,t_i+1,..,t_8)\\
& t+1 = t_i+\ldots+t_i+1+\ldots+t_8\\
& = R(t+1/t).
\end{aligned}
$$

Jest jasne, że $C=K=B=D=1 \wedge We=We_0 \wedge d(We_0)=m \wedge Y=nil$ implikuje $R(0/t)$. Pozostaje wykazać istnienie ograniczenia na t, lub co na jedno wychodzi, ograniczenia na każdą z liczb $t_1,..,t_8$.
Z postaci warunku Q wynika, że t_1 jest ograniczone przez m. Wynika stąd, że t_2 jest ograniczone, a więc t_3, t_4, t_5 są też ograniczone. Skoro t_5 jest ograniczone i d(Y) jest ograniczone, t_7 jest ograniczone. Ograniczone są więc t_4, t_5 i t_7; wobec tego t_6 jest ograniczone, a zatem ograniczone są t_7 i t_8. Tak więc wszystkie t_i są ograniczone dla i = 1,..,8; ograniczone jest zatem i t. Z twierdzenia 3 wnosimy, że rozważany program jest zbieżny dla warunku $C=K=B=D=1 \wedge We=We_0 \wedge d(We_0)=m \wedge Y=nil$. Z definicji zbieżności wynika, że ostatni stan każdego wykonania programu jest martwy; spełnia

zatem warunek M. Ponieważ uprzednio udowodniliśmy niezmienniczość warunku $(B=2 \Leftrightarrow K=4) \wedge (B=3 \Leftrightarrow D=2)$ dla warunku $C=K=B=D=1$, zatem każde wykonanie od $C=K=B=D=1 \wedge We=We_0 \wedge d(We_0)=m \wedge Y=Wy=nil$ kończy się stanem spełniającym warunek

$$M \wedge (B=2 \Leftrightarrow K=4) \wedge (B=3 \Leftrightarrow D=2)$$

równoważnym warunkowi

$$C=D=K=B=1 \wedge We=nil=Y.$$

Jak wykazaliśmy wyżej, jeśli ten warunek jest spełniony, to również jest spełniony warunek

$$Wy = f^{*}(We_0).$$

Tak więc rozważany program rozpoczynając działanie od dowolnego stanu spełniającego warunek $C=K=B=D=1 \wedge We=We_0 \wedge Wy=Y=nil$ kończy swe działanie jeśli lista We_0 jest skończona, i wówczas stan końcowy spełnia warunek $Wy = f^{*}(We_0)$. Udowodniliśmy zatem stwierdzenie T1. Aby dowieść T2 zauważmy, że warunek Q:

$$Inf(We) \wedge We \neq nil \wedge (B=2 \Leftrightarrow K=4) \wedge (B=3 \Leftrightarrow D=2)$$

jest niezmienniczy dla $C=K=B=D=1 \wedge Inf(We)$, a ponieważ ponadto

$$Q \Rightarrow Z,$$

zatem każde wykonanie od warunku $C=K=B=D=1 \wedge Inf(We)$ zawiera wyłącznie żywe elementy; każde pełne wykonanie więc od tego warunku jest nieskończone. Wynika stąd, że rozważany przez nas program nigdy się nie blokuje przy nieskończonym strumieniu danych wejściowych.

4. Podsumowanie.

Podaliśmy powyżej pewien język programowania współbieżnego i rozważaliśmy pewien przykład programu napisanego w tym języku.

Na tym przykładzie zilustrowaliśmy zastosowanie trzech twierdzeń o zachowaniu się programów, użytecznych w dowodzeniu niezmienniczości i nieuchronności zajścia pewnych warunków. Rozmiary tej pracy nie pozwalają na bardziej wnikliwą analizę poruszanych zagadnień; należy jedynie stwierdzić, że przyjęta tutaj sekwencyjna metoda reprezentacji wykonania programów współbieżnych, jakkolwiek najprostsza, nie obejmuje pewnych istotnych cech współbieżności, np. podana tutaj definicja nieuchronności jest zbyt wąska w wielu przypadkach procesów współbieżnych. Tak więc niniejsza praca ma jedynie na celu wstępne zaznajomienie czytelnika z problematyką programów współbieżnych i zachęcenie go do samodzielnych badań w tym kierunku.

BIBLIOGRAFIA

Mazurkiewicz, A.:Invariants of Concurrent Programs, IFIP-Infopol Conference 1975, North Holland, 1976

Owicki,S. and Gries,D.: Verifying properties of parallel programs: an axiomatic approach, CACM 19, 1976.

Jesienna Szkoła PTI
Rydzyna, październik 1984

TYPY DANYCH: OD ALGORYTMICZNEJ SPECYFIKACJI DO IMPLEMENTACJI W JĘZYKU PROGRAMOWANIA

Andrzej Salwicki
Instytut Informatyki
Uniwersytet Warszawski
PKiN skr. poczt. 22
00-901 Warszawa

STRESZCZENIE

Główne tezy tej pracy można sformułować w kilku punktach:

A/ Zastosowanie własności algorytmicznych umożliwia specyfikację struktur danych. Inne metody specyfikacji są bardziej skomplikowane.

B/ Algorytmiczne aksjomaty struktur danych umożliwiają prostszą analizę programów przeznaczonych do wykonania w tych strukturach.

C/ Istnieje naturalny sposób uzasadniania modułów programowych tzw. klas, implementujących struktury danych.

1. Wstęp

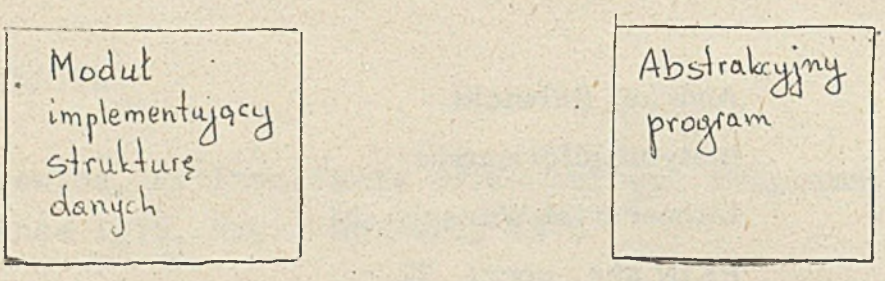
Jest rzeczą ogólnie uznaną, że struktury danych mają zasadnicze znaczenie dla programowania. Niemal każdy programista jest świadom, że jego pra-

ca nad oprogramowaniem może być w naturalny sposób podzielona na dwa etapy:

/i/ specyfikacja i implementacja struktury danych

/ii/ projektowanie, analiza i uruchamianie programu w którym korzysta się z operacji tej struktury danych.

Ta metodologiczna zasada pochodzi od C.A.R.Hoare'a [7]. Zgodnie z nią podczas tworzenia programu możemy i powinniśmy abstrahować od szczegółów implementacji, natomiast powinniśmy wykorzystywać tylko te własności struktury danych, które zostały wymienione w jej specyfikacji lub mogą być z niej wyprowadzone. Na końcowy produkt programistyczny składają się dwa moduły.



Zasada Hoare'a umożliwia wykonywanie abstrakcyjnego programu w towarzystwie różnych modułów implementujących. Jeżeli przestrzegaliśmy tej zasady to zmiana modułu implementującego nie wymaga żadnych dodatkowych zmian w abstrakcyjnym programie, jego poprawność zostanie zachowana. Natomiast możemy zyskać /bądź stracić/ na koszcie wykonania zadania jeżeli lepiej /lub gorzej/ dobierzemy implementację operacji struktury danych jakie wykonuje algorytm.

Podział pracy według tej zasady ma i drugą zaletę, jeden moduł implementacyjny może być wykorzystany przez wiele programów. Moduł taki jest więc implementacją pewnego języka problemowo-zorientowanego.

Odłóżmy na chwilę pytanie czy istnieją języki programowania umożliwiające programowanie zgodne z zasadą Hoare'a.

Zajmiemy się najpierw zagadnieniami bardziej "teoretycznymi" /czy naprawdę tylko teoretycznymi?/. Kiedy opis struktury danych /inaczej, specyfikacja/ możemy uznać za opisujący zamierzoną strukturę? W jaki sposób akceptować /bądź odrzucać/ moduły implementujące? tzn. kiedy możemy uznać, że

implementacja jest poprawna? Jakie metody pracy należy zaakceptować podczas pracy nad programem, tak by uzyskać pewność, że zrealizuje on podstawowe zadanie w towarzystwie każdego modułu implementującego? Tak więc pojawiają się trzy ważne problemy

- specyfikacji /opisu struktury danych/,
- implementacji /realizacji struktury danych/,
- weryfikacji /analizy programu abstrakcyjnego/.

Problemom tym poświęcono wiele prac, uzyskano wiele interesujących wyników. Kwestia specyfikacji wydaje się podstawową i początkową. Zacząć wypada od pytania: co to jest typ danych /inaczej struktura danych/? W tej mierze niemal wszyscy są zgodni. Struktura danych to system algebraiczny składający się ze zbioru elementów /inaczej uniwersum/ operacji na elementach oraz pewnych relacji.

PRZYKŁAD

Bardzo ważną strukturą danych są liczby naturalne

$$\langle \text{Nat};\ 0,\ +1,\ = \rangle$$

Nat oznacza zbiór liczb naturalnych ,
0 stałą /zeroargumentową operację/ zero
+ 1 1-argumentową operację następnika
= dwuargumentową relację równości.

Czasami w uniwersum wyróżniamy rozłączne podzbiory zwane sortami a operacje są określone właśnie na tych podzbiorach np. w przypadku struktury stosów uniwersum rozpada się na dwa podzbiory E - elementów i S - stosów operacje określone są na dziedzinach następujących

$$\text{push} : E \times S \to S \qquad \text{empty?} : S \to \{\text{prawda, fałsz}\}$$
$$\text{top} : S \to E$$
$$\text{pop} : S \to S$$

Struktura stosów ma więc następującą sygnaturę

$$< E, S;\ push,\ pop,\ top,\ empty?\ ,\ = >$$

Kwestie sygnatury nie budzą kontrowersji, jest to łatwiejsza, syntaktyczna część opisu struktury danych. Trudniej poradzić sobie z semantyką, jakie własności mają być spełnione przez operacje i relacje byśmy uznali, że struktura $< U, const, f(.) , => $ /gdzie U jest zbiorem, const stałą, f jednoargumentową operacją, $f : U \to U$/ jest strukturą liczb naturalnych? Jakie własności ma spełniać struktura $U_1, U_2; f_1(.,.) , f_2(.) , f_3(.) , r_1 => $ byśmy ją uznali za strukturę stosów?

W literaturze można wyróżnić trzy podejścia do tego zagadnienia:

1. Specyfikacja algebraiczna. Najbardziej popularne jest podejście algebraiczne. W tym przypadku do opisanej wyżej sygnatury stosów dodaje się algebraiczne aksjomaty równości i równości warunkowe

$$top(push(e, s)) = e$$
$$pop(push(e, s)) = s$$
$$\neg\ empty(s) \Rightarrow\ s = push(top(s), pop(s))$$
$$empty(push(e, s)) = fałsz$$

Łatwo zauważyć, że nie są to informacje wystarczające do zidentyfikowania tych struktur, które chcemy uznać za stosy bo na przykład nieskończone ciągi elementów ze zbioru E z odpowiednio określonymi operacjami też spełniają powyższe postulaty. Aksjomaty te nie przynoszą też dostatecznej informacji dla wnioskowania o zachowaniu się programów. Jak np. z wymienionych wyżej aksjomatów wyprowadzić fakt, że nie zapętli się obliczenie poniższego programu

```
begin   s1 := s;
        while ¬ empty (s2) do s2 := pop (s2) od;
        while ¬ empty (s1) do
          if e ≠ top (s1) then push(top (s1) , s2) fi;
          s1 := pop (s1)
        od
end
```

W pewnych pracach postępuje się więc tak: rozważamy kategorię K wszystkich modeli aksjomatów równościowych. Umawiamy się, że idzie nam o wyróżniony obiekt w tej kategorii np. obiekt początkowy 0.

Ta metoda pozwala w dwu etapach zidentyfikować system algebraiczny. Szkopuł w tym, że prawda o obiekcie 0 jest znacznie bogatsza od naszych początkowych czterech równości ale powiedzenie: "rozważmy obiekt początkowy" nie wnosi nowych aksjomatów.

2. Identyfikacja dziedziny. Podobne w duchu jest podejście zainicjowane przez D. Scotta. W tym przypadku by zdefiniować co to jest drzewo binarne powiadamy: jest to najmniejsze rozwiązanie równania

$$D = A + D \times D$$

gdzie A jest dziedziną atomów a D - poszukiwaną dziedziną drzew. D. Scott przedstawił wyrafinowaną teorię matematyczną dotyczącą zagadnień rozwiązalności takich równań. Praktyk ma jednak niewiele pożytku z tej teorii. Nie pozwala ona jak dotąd na opis własności operacji na drzewach ani na badanie zagadnień weryfikacji bądź implementacji. Przyznał to sam autor w w swoim odczycie po otrzymaniu nagrody im. Turinga.

3. Konstrukcja dziedzin. Istnieje wreszcie trzecie podejście, w którym za strukturę danych uznaje się to co się da skonstruować ze struktur przyjętych jako pierwotne. Podejście to zwane konstruktywnym łatwo jest skryty-

kować za stosowanie operacji niekonstruktywnych np. klasa wszystkich funkcji ze zbioru A w zbiór B jest uznawana przez zwolenników tego kierunku za dopuszczalną operację na strukturach. Drugą słabością tego podejścia jest podobnie jak w poprzednich przypadkach brak rozwiniętego aparatu dedukcyjnego dla wnioskowania o własnościach programów i samych struktur.

W dalszym ciągu będziemy używać pojęcia: algorytmiczna teoria. Składają się na nią trzy elementy:

a/ język algorytmiczny,

b/ system dedukcyjny /aksjomaty logiki i reguły wnioskowania/,

c/ zbiór aksjomatów specyficznych /poza logicznych/.

Nasze rozważania dotyczące teorii struktur danych będą oparte o logikę algorytmiczną programów iteracyjnych, deterministycznych. W tej pracy rozważamy trzy operacje programotwórcze.

begin ... end	złożenie
if ... then ... else ... fi	rozgałęzienie
white ... do ... od	iterację.

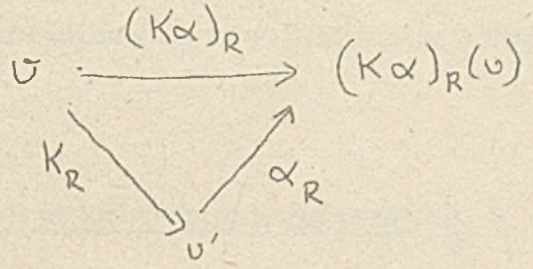
Wszystkie teorie mają języki zdefiniowane w podobny sposób. Klasa języków algorytmicznych, którą tu się będziemy posługiwać ma jeden ogólny wzorzec. Różnice pomiędzy językami wynikają z różnic w zestawach symboli funkcyjnych i relacyjnych. W każdym z języków zbiór wyrażeń dobrze zbudowanych składa się z trzech podzbiorów: termów, formuł i programów. Struktura zbioru termów jest dobrze znana. Programy są zbudowane z programów atomowych /tzn. instrukcji przypisania/ przy pomocy operacji programotwórczych. Zbiór formuł zawiera formuły bezkwantyfikatorowe 1-go rzędu i jest zamknięty ze względu na zwykłe reguły tworzenia formuł a ponadto dla każdego programu K i formuły α wyrażenie postaci

$$K\alpha$$

jest znowu formułą. Semantyczne znaczenie formuły $K\alpha$ jest następujące dla ustalonej interpretacji R symboli funkcyjnych i relacyjnych i dla ustalo-

nego stanu v znaczenie $K\alpha$ dla R i v jest prawdą wtedy i tylko wtedy gdy obliczenie programu K rozpoczynające się od danego stanu v jest skończone a jego stan końcowy v' spełnia formułę α , w pozostałych przypadkach wartość formuły α jest fałszem.

x



Przykład

Formuły postaci

$$\alpha \Rightarrow K\beta$$

wyrażają poprawność /całkowitą/ programu K ze względu na warunek początkowy α i warunek końcowy β .

Niech α , β , K będą wyrażeniami:

α : $\big(f(a)\cdot f(b) < 0\big) \wedge \big(b - a > \varepsilon > 0\big)$

β : $\big(f(a)\cdot f(b) < 0\big) \wedge \big(0 < (b - a) < \varepsilon\big)$

K: while $(b - a) > \varepsilon$ do $x := (a + b)/2$;
if $f(x)\cdot f(a) < 0$ then $b := x$ else $a := x$ fi od

Przy tych oznaczeniach formuła $\alpha \Rightarrow K\beta$ jest prawdziwa wtedy i tylko wtedy gdy algorytm bisekcji zatrzymuje się znajdując przybliżenie zera funkcji f lub nie spełniony jest warunek α . Formuła ta może być formalnie wyprowadzona z następującego aksjomatu Archimedesa

$$x > y > 0 \Rightarrow \big(z := y;\ \text{while } z < x \text{ do } z := z + x \text{ od}\big)\ \text{true}$$

A więc algorytm bisekcji będzie działał poprawnie w każdej strukturze danych w której prawdziwy jest aksjomat Archimedesa, tzn. w każdym ciele Archimedesowskim.

Logika algorytmiczna pozwala wyspecyfikować semantyczne własności programu i struktur danych. Problem aksjomatycznej definicji semantyki operatorów programotwórczych został rozwiązany [9,10]. Operacja konsekwencji logicznej jest zdefiniowana przez podanie zbioru schematów aksjomatów logicznych i reguł wnioskowania.

Przykłady

Niech K oznacza program, α, β, γ - formuły

$$K(\alpha \vee \beta) \Leftrightarrow (K\alpha \vee K\beta)$$

$$\underline{\text{while}}\ \gamma\ \underline{\text{do}}\ K\ \underline{\text{od}}\ \alpha \Leftrightarrow (\neg\gamma \wedge \alpha) \vee (\gamma \wedge K\ \underline{\text{while}}\ \gamma\ \underline{\text{do}}\ K\ \underline{\text{od}}\ \alpha) \quad \square$$

Spośród reguł wnioskowania wymieńmy

$$\frac{\alpha,\ \alpha \Rightarrow \beta}{\beta} \qquad \frac{\alpha \Rightarrow \beta}{K\alpha \Rightarrow K\beta}$$

$$\frac{\{(\underline{\text{if}}\ \gamma\ \underline{\text{then}}\ K\ \underline{\text{fi}})^i(\neg\gamma \wedge \alpha) \Rightarrow \beta\}_{i \in \mathbb{N}}}{\underline{\text{while}}\ \gamma\ \underline{\text{do}}\ K\ \underline{\text{od}}\ \alpha \Rightarrow \beta}$$

Wiele struktur danych nie może być opisanych w logice klasycznej, natomiast własności algorytmiczne w pełni je charakteryzują np.

"y jest liczbą naturalną"

$$x := 0\ ;\ \underline{\text{while}}\ x \neq y\ \underline{\text{do}}\ x := x+1\ \underline{\text{od}}\ \underline{\text{true}}$$

"s jest stosem"

$$\underline{\text{while}}\ \neg\ \text{empty}(s)\ \underline{\text{do}}\ s := \text{pop}(s)\ \underline{\text{od}}\ \underline{\text{true}}$$

aksjomat ciała charakterystyki zero

$$\neg\ (x := 1\ ;\ \text{while}\ x \neq 0\ \text{do}\ x := x+1\ \text{od true})$$

W dalszym ciągu nasze poglądy i techniki postępowania zostaną zilustrowane na przykładzie algorytmicznej teorii kolejek priorytetowych. Przedyskutu-

jemy problemy specyfikacji i implementacji kolejek priorytetowych w drzewach binarnych poszukiwań. STąd niemal natychmiast zmieniając tylko ortografię otrzymamy programistyczny moduł implementujący kolejki priorytetowe zapisany jako klasa w LOGANie. /LOGAN jest uniwersalnym językiem programowania zaprojektowanym i zrealizowanym w Instytucie Informatyki UW/.

2. Algorytmiczna teoria kolejek priorytetowych

Kolejka priorytetowa jest strukturą danych dla wykonywania operacji: insert, delete, min, member na skończonych zbiorach. Kolejki priorytetowe są ważne jako jedna z najczęściej występujących struktur danych. Niech U /uniwersum/ będzie pewnym zbiorem uporządkowanym przez relację . Używane są gdy mamy zamiar:

- dowiedzieć się czy pewien element uniwersum znajduje się w danym /skończonym/ zbiorze? /member/,
- powiększyć zbiór wstawiając /insert/ element,
- usunąć /delete/ element ze zbioru,
- znaleźć element najmniejszy w danym zbiorze /min/,
- dowiedzieć się czy zbiór jest pusty /empty/.

Kolejki priorytetowe tworzą abstrakcyjny typ danych, ponieważ mogą być realizowane na wiele różnych sposobów [1 AHU,] [2,8]. A więc będziemy myśleć o klasie kolejek priorytetowych, podobnie jak myślimy o klasach grup, pierścieni, algebr Booleowskich i in.

Definicja 1.

System algebraiczny nazywamy kolejką priorytetową gdy jego uniwersum składa się z dwu rozłącznych zbiorów

E i S

nazywanych rodzajami /ang. sorts/ i gdy dopuszcza on następujące operacje i

predykaty /funkcje charakterystyczne relacji/

insert : $E \times S \to S$

delete : $E \times S \to S$

min : $S \to E$

member: $E \times S \to B_o$; B_o jest dwuelementową algebrą Boole'a wartości logicznych {prawda, fałsz}

empty : $S \to B_o$

$\leq$: $E \times E \to B_o$

Wymienione operacje i relacje powinny spełniać następujące postulaty:
/Zmienne e i s przyjmują wartości ze zbiorów E i S/

P1. $\neg$ empty (s) $\Rightarrow$ member (min (s), s)

P2. empty (s) $\Leftrightarrow (\forall e) \neg$ member (e, s)

P3. dla każdego s instrukcja s := delete (min (s), s) może być powtórzona tylko skończoną ilość razy dopóki zbiór s nie będzie pusty tzn. następujący program zawsze kończy swe obliczenia

while $\neg$ empty (s) do s := delete (min (s), s) od

P4. dla każdego e $\in$ E, dla każdego s $\in$ S

member (e, insert (e, s))

$\neg$ member (e, delete (e, s))

P5. dla dowolnych e, e', s

$e' \neq e \Rightarrow \{$ member (e', s) $\Leftrightarrow$ member (e', insert (e, s)) $\}$

$e \neq e \Rightarrow \{$ member (e', s) $\Leftrightarrow$ member (e', delete (e, s)) $\}$

P6. Zbiór E jest uporządkowany liniowo przez relację $\leq$.

P7. $\neg$ empty (s) $\Rightarrow (\forall e)$ member (e, s) $\Rightarrow$ min (s) $\leq$ e

Nie jest w tym momencie jasne czy postulaty te nie przeczą sobie wzajemnie i czy rzeczywiście opisują tę klasę struktur o jaką nam chodzi.

Odpowiedzi na te pytania zaczniemy od przedstawienia bardziej precyzyjnej aksjomatyzacji kolejki priorytetowej. Zostanie ona zapisana w języku logiki algorytmicznej.

A1. Zbiór E jest liniowo uporządkowany przez relację $\leq$. /Odpowiednie formalne aksjomaty znaleźć można np. w Rasiowa

A2. while $\neg$ empty (s) do s := delete (min (s), s) od true

A3. $\neg$ empty (s) $\Rightarrow$ ($\forall$e) (member (e, s) $\Rightarrow$ min (s) $\leq$ e)

A4. $[s' := \text{insert}(e, s)]\{\text{member}(e, s) \wedge e' \neq e \Rightarrow \{\text{member}(e', s) \Leftrightarrow \text{member}(e', s')\}\}$

A5. $[s := \text{delete}(e, s)]\{\neg\ \text{member}(e, s) \wedge e' \neq e \Rightarrow (\text{member}(e', s) \Leftrightarrow \text{member}(e', s'))\}$

A6. member (e, s) $\Leftrightarrow$ begin s1 := s ; bool := false ;
while $\neg$ empty (s1) $\wedge$ bool do
e1 := min (s1) ;
bool := (e1 = e) ;
s1 := delete (e1, s1)
od
end bool

Zbiór aksjomatów A1 - A6 jest niesprzeczny, istnieje mianowicie prosta interpretacja terminów E, S, insert, delete, min, member, empty, w której zdania A1 - A6 są prawdziwe.

Niech E będzie dowolnym zbiorem liniowo uporządkowanym przez relację $\leq$. Jako S weźmy rodzinę skończonych podzbiorów zbioru E . Operacje member, insert, delete, empty niech będą teoriomnogościowymi operacjami

member - wynikiem tej operacji jest odpowiedź na pytanie czy element e należy do zbioru s

insert - wynikiem operacji jest teoriomnogościowa unia zbiorów $\{e\}$ i s

delete - wynikiem operacji jest zbiór $s \setminus \{e\}$

empty (s) zbiór s jest pusty

min (s) jest najmniejszym elementem zawartym w niepustym zbiorze s.

Łatwo teraz sprawdzić, że każda z formuł A1 - A6 przyjmuje w tej interpretacji stale wartość prawda.

Czy mamy jakiś pożytek z tej aksjomatyzacji?
Okaże się, że tak. Rozpatrzmy program posługujący się operacjami kolejek priorytetowych. Najprościej będzie wykazać, że program występujący w aksjomacie A6 nie zapętla się. Zauważmy, że jeżeli pominąć zmienne e1 i bool, to programy

s1 := delete (min (s1), s1) oraz

begin
e1 := min (s1) :
bool := (e1 = e) :
s1 := delete (e1, s1)
end

są równoważne. O programie

while ¬ empty (s1) do s1 := delete (min (s1) , s1) od

wiemy, że się nie zapętla, (aksjomat A2.) stąd łatwo wyprowadzić jako wniosek

while ¬ empty (s1) do e1 := min (s1) : bool := (e1 = e) :
s1 := delete (e1, s1) od true

Następnie możemy zastosować bardzo pożyteczną regułę wnioskowania

$$\frac{\alpha \Rightarrow \beta}{\text{while } \beta \text{ do } K \text{ od } \text{true} \Rightarrow \text{while } \alpha \text{ do } K \text{ od } \text{true}}$$

otrzymując

while ¬ empty (s1) ∧ bool do
e1 := min (s1) ; bool := (e1 = e) :
s1 := delete (e1, s1)
od true

Następnie stosując regułę wnioskowania

$$\frac{\alpha \ , \ K \ \underline{\text{true}}}{K \alpha}$$

możemy tę formułę poprzedzić instrukcjami przypisania

s1 := s ; bool := false ;

a następnie stosując aksjomat logiczny

begin K ; M end $\alpha \Leftrightarrow$ K M α

otrzymujemy pożądany wynik

```
begin s1 := s ; bool := false ;
    while ¬ empty(s1) ∧ bool do
        e1 := min (s1) ; bool := (e1 = e) ;
        s1 := delete (e1, s1)
    od
end true
```

Można podać wiele innych przykładów dowodzenia poprawności programu w oparciu o aksjomaty struktury danych. Jeszcze raz zwróćmy uwagę na to, że dowód ten nie ulega zmianie wraz ze zmianą interpretacji symboli min, delete etc. jeżeli tylko nowa interpretacja spełnia aksjomaty A1 - A6.

Nie łatwo odpowiedzieć na pytanie czy podana wyżej specyfikacja zawiera dość aksjomatów. Chcieliśmy opisać własności klasy struktur, w których wykonuje się operacje na skończonych zbiorach. W klasie tej znajdują się bardzo różne struktury różniące się rodzajem elementów, sposobem organizacji zbioru i algorytmami wykonywania operacji insert, min, delete. W naszej specyfikacji nie zajmujemy się pytaniami jak wykonać tę czy inną operację lecz jakie są własności operacji występujących w kolejce priorytetowej. Następujące twierdzenie da pozytywną odpowiedź na pytanie czy udało się nam znaleźć wszystkie własności klasy kolejek priorytetowych.

Twierdzenie 1 /o reprezentacji/

Każda struktura danych $\mathfrak{M}$ spełniająca aksjomaty A1 - A6 kolejek priorytetowych jest izomorficzna ze standardowym modelem.

Dowód tego twierdzenia pomijamy z braku miejsca [13].

Twierdzenie to ma wiele konsekwencji. Przede wszystkim orzeka ono, że mamy do wyboru dwie możliwości gdy chcemy dowieść, że pewna własność kolejek priorytetowych jest twierdzeniem Algorytmicznej teorii kolejek priorytetowych: ATPQ to

albo

a/ możemy skonstruować dowód w oparciu o aksjomaty A1- A6

bądź

b/ możemy wykazać, że badana własność jest prawdziwa w każdym modelu standardowym tzn. niezależnie od wyboru zbioru .

Inne sformułowanie twierdzenia o reprezentacji

Formuła α jest prawdziwa w standardowym modelu ATPQ wtedy i tylko wtedy gdy jest prawdziwa w każdym modelu tej teorii.

Nie jest zaskakujący fakt, że ATPQ nie jest teorią zupełną, ma ona przecież wiele różnych modeli. Niech $\underset{S}{=}$ oznacza relację równości kolejek priorytetowych, zauważmy, że istnieje procedura stwierdzająca równość $s_1 \underset{S}{=} s_2$ zdefiniowana w terminach insert, delete, min, empty niezależnie od tego jak te operacje są wykonywane. Zdanie

$$(\exists s_o)(\forall e)\ (s_o \underset{S}{=} \text{insert } e, s_o)$$

jest niezależne od aksjomatów ATPQ, ponieważ jest prawdziwe w tych tylko przypadkach gdy E jest zbiorem skończonym.

3. Algorytmiczna teoria drzew binarnych poszukiwań - ATBST

Niech E będzie zbiorem liniowo uporządkowanym przez relację $\leq$. Drzewem binarnych poszukiwań jest etykietowane drzewo binarne, w którym każdy wierzchołek w jest etykietowany przez element $e(w) \in E$ i takie, że

a/ dla każdego wierzchołka q w lewym poddrzewie w: $e(q) < e(w)$

b/ dla każdego wierzchołka q w prawym poddrzewie w: $e(w) < e(q)$

Drzewa binarnych poszukiwań są zwykle implementowane przy pomocy następującej deklaracji typu

```
unit N : class (v : E);
    variable l, r : N
end N .
```

Z deklaracją tą możemy związać system algebraiczny o następującej sygnaturze

$$\langle E, N;\ v, l, r,\ \underline{new}\ N, ul, ur, isnone,\ \overset{=}{}_E\ ,\ \overset{\leq}{}_E \rangle,$$

gdzie $\underline{new}\ N : E \to N$

$v : N \to E$, $l : N \to N$, $r : N \to N$

$ul : N \times N \to N$, $^{u}\ : N \times N \to N$

$isnone : N \to B_o$

$\overset{=}{}_E$ i $\overset{\leq}{}_E$ są relacjami identyczności i liniowego porządku w E

W językach programowania, które bierzemy pod uwagę ta deklaracja typu N może być interpretowana jako opis klasy obiektów o następującej strukturze

n :

v	e
l	n_1
r	n_2

gdzie $e \in E$, $n_1, n_2 \in N$. Klasa N zawiera także pusty obiekt oznaczony none.

Wymienione powyżej operacje mają oczywiste znaczenie
$v(n)$ - odczytaj wartość atrybutu v w obiekcie n,
$l(n)$, $r(n)$ - wskazują obiekty stowarzyszone z n jako korzenie jego lewego i prawego poddrzewa odpowiednio, operacje ul i ur oznaczają zmianę /update/ wartości l i r. W języku programowania instrukcje $n := ul(n', n)$ oraz $n := ur(n', n)$ są zapisywane $n \cdot l := n'$ i $n \cdot r := n'$ i będziemy się trzymać tej konwencji w dalszym ciągu. Będziemy też pisać $n.v$ zamiast $v(n)$ oraz $n \cdot l$ i $n \cdot r$ zamiast $l(n)$ i $r(n)$.

System algebraiczny o opisanej wcześniej sygnaturze będzie nazywany systemem drzew binarnych poszukiwań jeśli spełnia aksjomaty B1 - B9

B1/ new N(e). v = e
{ wartość atrybutu v w nowo kreowanym obiekcie jest e }
B2/ isnone(new N(e). l)
B3/ isnone(new N(e). r)
{ w nowo kreowanym obiekcie atrybuty l i r mają wartość none - taki obiekt należy interpretować jako liść }

Następująca definicja będzie używana w aksjomatach B4 i B5

$$mb(e, n) \overset{df}{\equiv}$$

```
begin n1 := n ; bool := false ;
  while ¬ isnone(n1) ∧ ¬ bool do
  if n1.v = e then bool := true else
  if e < n1.v then n1 := n1.l else
                   n1 := n1.r fi fi
  od
end bool
```

{ relacja mb zdefiniowana powyżej jest relacją należenia }

B4/ $mb(e, n.l) \Rightarrow e < n.v$

B5/ $mb(e, n.r) \Rightarrow n.v < e$

{ dla każdego niepustego drzewa o korzeniu n, każdy element jego lewego poddrzewa jest mniejszy niż wartość przypisana wierzchołkowi n i każdy element jego prawego poddrzewa jest większy niż wartość przypisana korzeniowi n }

B6/ isnone (n) $\vee$

(begin $n' := n$; while $\neg$ isnone (n') do

if isnone $(n'.l)$ then $n1 := n'.r$ else $n1 := new\ N(n'.l.v)$;

$n1.l := n'.l.l$;

$n2 := new\ N(n'.v)$;

$n2.l := n'.l.r$;

$n2.r := n'.r$;

$n1.r := n2$ fi

$n' := n1$

od

end) true

{ dla każdego elementu n, n jest korzeniem skończonego drzewa binarnego }

B7/ $(n.r = n'' \wedge n.v = e \wedge$

[begin $n2 := n'$; while $\neg$ isnone $(n2.r)$ do $n2 := n2.r$ od ;

if $n2.v < n.v$ then bool := true else bool := false fi end bool

$\vee$ isnone (n')]) $\Rightarrow (n.l := n')\{ n.r = n'' \wedge n.v = e \wedge n.l = n' \}$

{ jeżeli największy element w drzewie n' jest mniejszy niż n.v lub n' jest drzewem pustym - isnone (n') to przypisanie określające n' jako lewego syna n jest dobrze określone a pozostałe atrybuty n pozostają bez zmian }

B8/ $(n.l = n'' \wedge n.v = e \wedge$

begin $n2 := n'$; while $\neg$ isnone $(n2.l)$ do $n2 := n2.l$ od ;

if $n2.v > n.v$ then bool := true else bool := false fi

end bool $\vee$ isnone (n')])

$\Rightarrow (n.r := n')\{ n.l = n'' \wedge n.v = e \wedge n.r = n' \}$

{ jeżeli najmniejszy element w drzewie n' jest większy niż n.v lub n' jest drzewem pustym - isnone (n') to przypisanie określające n' jako prawego syna n jest dobrze określone a pozostałe atrybuty n pozostają bez zmian }

B9/ Zbiór E jest liniowo uporządkowany przez relację $\leq$.

Można zapytać czy zbiór B1 - B9 nie jest przypadkiem sprzeczny. Rozstrzygnięcie tego pytania przynosi

Twierdzenie
Algorytmiczna teoria drzew binarnych poszukiwań ATBST ma model, jest więc niesprzeczna.

Dowód. Rozpatrzmy zbiór S wyrażeń nad zbiorem E , który zawiera wyrażenie $()$ reprezentujące none i taki, że dla każdego $e \in E$

1° wyrażenie $(() \; e \; ())$ należy do S,

2° jeżeli dwa wyrażenie ν oraz τ należą do S oraz jeśli

dla każdego elementu f występującego w ν $f < e$,

i dla każdego elementu f występującego w τ $f > e$

to wyrażenie

$$(\nu \; e \; \tau) \quad \text{jest w S}$$

3° S jest najmniejszym zbiorem wyrażeń zamkniętym ze względu na 1° i 2°.

Interpretacja funktorów jest następująca

isnone $(v) \equiv \nu = ()$

new $N(e) = (() \; e \; ())$

dla każdego $\nu' \neq ()$ wyrażenie ν' ma więc postać $(\upsilon \; e \; \tau)$

kładziemy

$$\upsilon(\nu') = e \quad . \quad l(\nu') = \nu, \quad r(\nu') = \tau$$

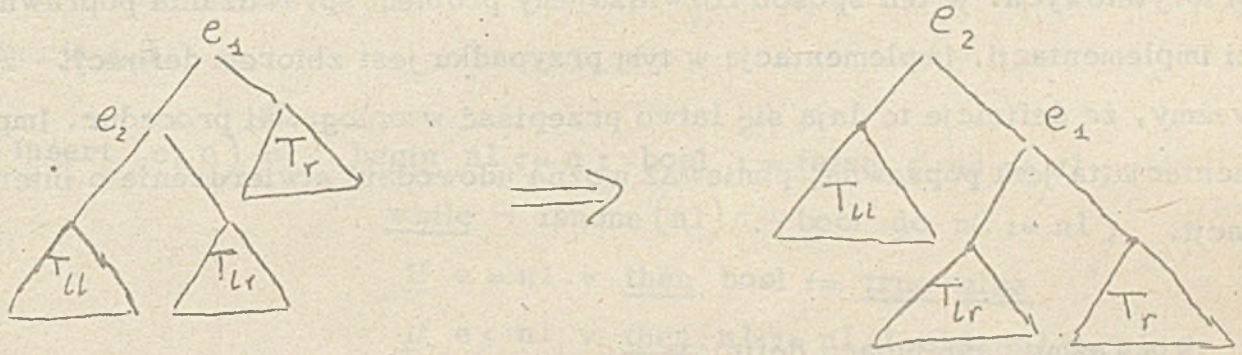
Operacje ul oraz ur są częściowymi operacjami określonymi w następujący sposób: Niech n oznacza wyrażenie $(\nu \; e \; \tau)$ i niech n' będzie innym wyrażeniem. Operacja $ul(n, n')$ jest określona wtedy i tylko wtedy gdy wszystkie elementy występujące w n są mniejsze niż e i jej wynikiem jest wyrażenie $(n' \; e \; \tau)$. Definicja operacji nr jest podobna. Łatwo sprawdzić, że wszystkie aksjomaty B1 - B9 są prawdziwe w tej strukturze . □

Bez dowodu przytoczymy następujące

Twierdzenie /o reprezentacji/

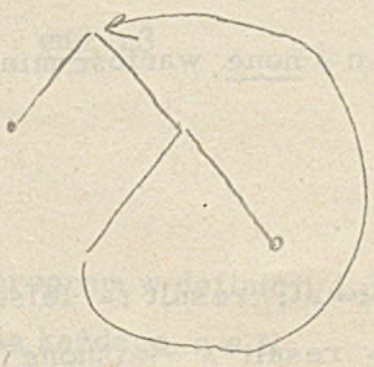
Każdy model aksjomatów B1 - B9 jest izomorficzny ze standardowym modelem opisanym powyżej.

W istocie aksjomat B6 zapewnia, że każde drzewo będzie przeanalizowane w skończonym czasie. Zauważmy, że istota algorytmu występującego w tym aksjomacie sprowadza się do powtarzania "rotacji" tzn. transformacji



która zachowuje własność być drzewem binarnych poszukiwań, ale skraca wysokość lewego poddrzewa.

Aksjomat B6 nie akceptuje żadnego cyklu np. dla obiektu



program wymiany w B6 nie zatrzyma się. Podana aksjomatyzacja nie zaakceptuje też grafów acyklicznych, ponieważ aksjomaty B4 i B5 zapewniają, że każde dwa poddrzewa danego drzewa są różne.

4. Interpretacja teorii kolejek priorytetowych

Zamierzamy wykazać, że istnieje interpretacja teorii kolejek priorytetowych w teorii drzew binarnych poszukiwań. Interpretacja ta zachowuje strukturę drzew binarnych poszukiwań i rozszerza zbiór operacji. Definicje operacji member, insert, delete, min są algorytmiczne. W teorii jaką uzyskujemy dodając do aksjomatów B1 - B9 teorii drzew BST nowe aksjomaty - definicje nowo wprowadzanych operacji potrafimy udowodnić aksjomaty teorii kolejek priorytetowych. W ten sposób rozwiązujemy problem sprawdzania poprawności implementacji. Implementacja w tym przypadku jest zbiorem definicji - zauważmy, że definicje te dają się łatwo przepisać w ortografii procedur. Implementacja ta jest poprawna, ponieważ można udowodnić stwierdzenie o interpretacji.

Rozważmy następującą definicję

Definicja 4.1

$$\min(n) \overset{df}{=} \Big(\underline{\text{if}}\ \text{isnone}(n)\ \underline{\text{then}}\ \text{ALARM}\ \underline{\text{else}}\ n1 := n\ \underline{\text{fi}}\,;\ \underline{\text{while}}\ \neg\ \text{isnone}(n1.l)\ \underline{\text{do}}\ n1 := n1.l\ \underline{\text{od}}\Big)\, n1.v$$

Jest niemal oczywiste, że dla $n \neq \underline{\text{none}}$ wartość $\min(n)$ jest określona.

Definicja 4.2

$$\text{member}(e, n) \overset{df}{=} \underline{\text{begin}}\ n1 := n\,;\ \text{result} := \text{false}\,;\ \underline{\text{while}}\ \neg\ \text{result} \wedge \neg\ \text{isnone}(n1)\ \underline{\text{do}}\ \underline{\text{if}}\ e = n1.v\ \underline{\text{then}}\ \text{re}\neg\ \text{isnone}(n1)\ \underline{\text{do}}\ \underline{\text{if}}\ e < n1.v\ \underline{\text{then}}\ n1 := n1.l\ \underline{\text{else}}\ n1 := n1.r\ \underline{\text{fi}}\ \underline{\text{fi}}\ \underline{\text{od}}\ \underline{\text{end}}\ \text{result}$$

Również i ten program zawsze się zatrzymuje /ponieważ zawsze zawsze zatrzymuje swoje obliczenia w standardowym modelu drzew BST/.

Lemat 4.1

Jeżeli wartość $\min(n)$ jest określona to

$$(\forall e)\ \text{member}(e, n) \Rightarrow \min(n) \leq e$$

Zamiast długiego formalnego dowodu z aksjomatów teorii ATBST wystarczy zbadać na mocy twierdzenia o reprezentacji prawdziwość tego lematu w standardowym modelu. □

Definicja 4.3

$$\text{insert}(e, n) \overset{df}{=}$$

```
begin n1 := n ; bool := false ; n3 := n1 ;
  while ¬ isnone(n1) ∧ ¬ bool do n2 := n1 ;
   if e = n1.v then bool := true else
   if e < n1.v then n1 := n1.l else n1 := n1.r
    fi fi od ;
   if ¬ bool then if isnone(n) then n3↑ := new N e
   else
   if e < n2.v then n2.l := new N(e) else
                    n2.r := new N(e) fi fi fi
end n3
```

Lemat 4.2

Niech M oznacza program w definicji 4.3.

Dla każdego $e \in E$, dla każdego $n \in N$

/i/ M member $(e, n3)$

/ii/ for every $e \neq e'$

$$\text{member}(e', n) \Leftrightarrow \text{M member}(e', n3)$$

Definicja 4.4

Aby zaoszczędzić miejsca podajemy nieformalny zapis procedury delete. Czytelnik znajdzie pełny jej tekst /LOGLANowski/ w następnym paragrafie.

delete $(e, n) \overset{df}{=}$ begin
{ szukaj n - cf. member }
{ załóżmy, że e jest odnalezione w wierzchołku n1
i że ojciec n1 = n2 }
{ jeżeli n1 jest liściem usuń n1 }
{ jeżeli n1 ma dokładnie jednego syna - uczyń n2
ojca n1 ojcem tego syna }
{ jeżeli n1 ma dwu synów - znajdź najmniejszy
element min $(n1, r)$ w prawym poddrzewie n1.
Usuń min $(n1, r)$ z drzewa o korzeniu n1. r.
Połóż ten element w korzeniu poddrzewa n1 }
end n3 (:= drzewo skonstruowane powyżej)

Lemat 4.3

Niech K oznacza program naszkicowany powyżej. Dla każdego e ∈ E, dla każdego n ∈ N

/i/ K ⊣ member $(e, n3)$

/ii/ dla każdego $e' \neq e$

member $(e', n) \iff$ K member $(e', n3)$

Składając obserwacje lematów 4.1 - 4.3 możemy sformułować

Twierdzenie 4.1 /o interpretacji teorii kolejek priorytetowych/

Wszystkie aksjomaty kolejek priorytetowych mogą być udowodnione z aksjomatów drzew binarnych poszukiwań oraz z aksjomatów dodatkowych definiujących operacje insert, delete, member, min, empty.

Oznacza to, że dla dowolnie danego modelu teorii drzew BST możemy zdefiniować model teorii kolejek priorytetowych. Co więcej - ponieważ wszystkie definicje są algorytmiczne - możemy efektywnie zdefiniować taki model.

5. Implementacja kolejek priorytetowych

Twierdzenie o interpretacji teorii kolejek priorytetowych w teorii drzew binarnych poszukiwań stwierdza ni mniej ni więcej poprawność deklaracji typu danych kolejka priorytetowa implementującego

```
unit BST: class (type E ; function less(e, e': E): Boolean);
  unit node : class (v : E);
    variable l,r : node ;
  end node :
unit min : function(n : node): E ;
  begin
    while n.l ≠ none do n := n.l od ;
    result := n.v
  end min ;
unit member : function(e : E, n : node ): Boolean ;
  variable n1 : node, bool : Boolean ;
  begin n1 := n ; bool := false ;
    while none ≠ n1 ∧ ¬ bool do
    if n1 . v = e then bool := true else
    if e< n1.v then n1 := n1.l else n1 := n1.r fi fi od ;
    result := bool
end member ;
unit empty : function(n : node): Boolean ;
begin
    if n = none then result := true else result := false fi
end empty ;
```

```
 Unit  insarte : function  /e: E, n: node / : node:
 veriable n1, n2, n3: node , bool: Boolean:
begin n1 := n3 :=n : bool := false :
while ¬ none=n1 ∧  ¬ bool do n2 :=n1;
if e = n1.v then bool := true else
if e n1.v then n1 :=n1.1 else n1 :=n1.r fi fi od ;
if ¬ bool then if none =n3 then n3 := new N/e/ else
if e< n2.v then n2.1 :=new N/e/ else n2.r :=new N/e/ fi fi fi
result := n3
end insert :
unit delete : function /e : E :, n : node/: node ;
variable n1, n2, n3, n4, n5: node, bool, leftson : Boolean;
begin n1 := n: n3 := n; bool :=false;
while  ¬ none = n1 ∧ ¬ bool do :=n1
    if e = n1.v then bool := true else
    if  e<n1.v then n1 :=n1.1 else n1 := n1.r fi fi od;
    if bool then {znaleziono w n1 a n2 jest ojcem n1}
    if e<n2.v then leftson := true else leftson := false fi;
    {n1 jest lewym synem n2  = leftson}
    if n1.1 = none ∧ n1.r = none then if leftson then n2.1 := none
    {n1 jest liściem}
                                        else n2.r := none fi
else if n1.1 = none then if n1 = n then n3 := n1.r else
if leftson then n2.1 := n1.r else n2.r := n1.r fi fi
else if n1.r = none then if n1=N then n3 :=n1.1 else if leftson then
n2.1 := n1.1 else n2.r := n1.r fi fi
else {n1 ma dwóch synów}  n4 := n1.r ;
   while n4.1 ≠  none do n5 := n4 : n4 := n4.1 od
                                        n5.1 := n4.r
   n1.v := n4.v fi fi fi fi
   result := n3
end delate ;
end BST
```

Istnieje też możliwość, w której unikamy formalnego parametru E . W tym przypadku stosujemy składanie deklaracji typów i procedury virtualne.

```
unit BST' : class
  unit E : class ; end E ;
  unit less : virtual function (e, e' : E) : Boolean ; end less ;
  unit node : class( v : E) ; var l, r : node ; end node ;
  unit min ...
  unit member ...
  unit insert ...
  unit empty ...
  unit delete ...
end BST'
```

Jednostki BST i BST są dwoma różnymi implementacjami języka problemowo zorientowanego. BST i BST' są stosowane w odmiennych otoczeniach. Język LOGLAN dopuszcza parametryzowane deklaracje typu np. BST. Aby zastosować BST należy przekazać nazwę typu - aktualnego parametru - opisującego zbiór elementów /kluczy/ i równocześnie nazwę funkcji boolowskiej porównującej dwa elementy /sprawdzającej czy są one w relacji $\leq_E$ /.

Zauważmy, że klasa BST' może być pojmowana jako generyczny opis całej rodziny struktur danych. Reprezentuje ona wzorzec, który ma być uzupełniony przez użytkownika.
Mianowicie, deklaracja

```
unit moje_BST : BST' class
  unit Elem : E class ... end Elem ;
  unit less : function (e, e : Elem) : Boolean ... end less ;
end moje_BST
```

reprezentuje pewne rozszerzenie BST przez określenie czyjegoś konkretnego zbioru Elem i odpowiedniej relacji less.

By zastosować taki język problemowo-zorientowany piszemy np.

```
pref moje_BST block
  {deklaracje np.  n, n : node , e, e  : Elem }
  begin
  {instrukcje np.  n := delete (e, n)}
end
```

UWAGI KONCOWE

Okazało się, że algorytmiczne teorie umożliwiają specyfikację struktur danych. Wprowadzenie własności algorytmicznych umożliwiło zdefiniowanie zarówno tych struktur, które chcemy określić z dokładnością do izomorfizmu jak i tych przypadków gdy zamierzamy określić klasę struktur np. kolejki priorytetowe.

Algorytmiczne aksjomaty ułatwiają też zadanie dowodzenia własności programów. Stosowanie reguł wnioskowania podsuwanych przez logikę algorytmiczną i oparcie się o algorytmiczne aksjomaty struktur danych czynią zbędnym dowodzenie przez indukcję ze względu na postać elementów struktury.

Zagadnienie implementacji struktur danych znajduje swój formalny odpowiednik w pojęciu interpretacji jednej teorii algorytmicznej w innej. Rozważania dotyczące interpretacji prowadzą do konstrukcji poprawnych modułów implementacyjnych struktur danych w LOGLANie. I w ten sposób zamyka się koło, od problemów softwareowych do teorii programów i stąd do konstrukcji oprogramowania. Warto podkreślić komplementarność i wyjątkową wprost zbieżność metod logiki algorytmicznej i metodologii programowania podsuwanej przez LOGLAN.

LOGLAN zainspirował wiele owocnych badań i nadal będzie źródłem ciekawych i ważnych problemów. Tu warto wspomnieć o algorytmicznej teorii refe-

rencji [Oktaba], która pozwala wyjaśnić wiele zjawisk nie objętych teoriami abstrakcyjnych typów danych. W teoriach tych nie sposób wyjaśnić aliasingu i zjawisk związanych z dealokacją obiektów.

LOGLAN dzięki swoim atrakcyjnym narzędziom programowania umożliwia szybką implementację języków programowania. Zaprogramowanie i uruchomienie sporego języka dla zadań symulacji - równoważnika systemowej klasy SIMULATION SIMULI - zajęło mniej niż dwa tygodnie. Przeprowadzono też eksperymenty z wyrażeniem w LOGLANie systemu operacyjnego czasu rzeczywistego i systemu plików. Wyniki są pozytywne.

Składanie deklaracji typów /prefiksowanie/ umożliwia nie tylko tworzenie hierarchii typów oraz języków problemowo-zorientowanych ale także dyscyplinowanie pracy programisty. Mamy tu na myśli fakt, że w LOGLANie daje się zdefiniować wiele narzędzi synchronizacji procesów współbieżnych. W szczególności zdefiniowano pojęcie monitora i okazało się, że składanie deklaracji typów umożliwia też narzucanie protokołów współpracy procesów współbieżnych. Ogólniej, można stwierdzić, że narzędzia LOGLANu nie zostały jeszcze w pełni poznane i że kryją jeszcze niejedną przyjemną niespodziankę.

BIBLIOGRAFIA

1. Aho, A., Hopcroft, J., Ullman, J., Projektowanie i analiza algorytmów komputerowych, PWN, 1983
2. Banachowski, L., On proving program correctness by means of stepwise refinement method w: Proc. Logics of Programs and their Applications, Poznań 1980
   Lecture Notes in Computer Science v. 148, Springer, Berlin, 1983
3. Materiały szkoły letniej LOGLANu, Zaborów 1983
   Wyd. UW 1983
4. Engeler, E., Algorithmic properties of structures, Math. System Theory 1 /1967/, 183-195

5. Goguen, J.A., Thatcher, J.W., Wagner, E.G., An initial algebra approach to the specification, correctness and implementation of abstract data types, IBM Rep RC 6487 /1977/
6. Guttag, J., Abstract data types ... CACM 20 /1977/, 396-404
7. Hoare, C.A.R., Proof of correctness of data representation, Acta Informatica 1 /1972/, 271-281
8. Knuth, D., The art of computer programming, vol. 3, Addison-Wesley 1973
9. Mirkowska, G., Algorithmic logic and its applications in the theory of programs, Fundamenta Informaticae 1 /1977/, 1-17, 147-167
10. Mirkowska, G., Salwicki, A., Algorithmic logic, ukaże się w PWN
11. Oktaba, H., Algorytmiczna teoria referencji, rozpr. dokt. Uniw. Warsz. Wydz. Matematyki i Informatyki, 1981
12. Salwicki, A., On algorithmic theory of stacks w: Proc MFCS' 78 Lecture Notes on Computer Science 64, 1978, także w Fundamentach
13. Salwicki, A., On algorithmic theory of dictionaries, w: Proc. Logic of Programs /E.Engeler ed./ LNCS vol. 125, 145-168
14. Scott, D., Data types as lattices, SIAM J.Comp. 5 /1976/, 522-587

Jesienna Szkoła PTI
Rydzyna, październik 1984

ALGORYTMY KOMBINATORYCZNE I ICH EFEKTYWNOŚĆ

Maciej M. Sysło
Instytut Informatyki
Uniwersytet Wrocławski
ul. Przesmyckiego 20
51-151 Wrocław

1. Wstęp

Praca ta poświęcona jest omówieniu wybranych metod rozwiązywania trzech przykładowych problemów kombinatorycznych: problemu najkrótszego drzewa rozpinającego w sieci (w skrócie, SST), problemu plecakowego i problemu kolorowania grafu. Najogólniej ujmując, *problem kombinatoryczny* polega na znalezieniu szczególnego elementu (lub tylko na stwierdzeniu czy taki element istnieje) w zbiorze, który jest dyskretny, najczęściej skończony i zwykle tworzy pewną strukturę taką jak graf, sieć lub jest rodziną permutacji czy też partycji zbioru. Konkretne wartości parametrów problemu określają *zadanie* tego problemu.

W zadaniu SST, dany jest graf i funkcja wagowa opisana na jego krawędziach - znaleźć należy drzewo rozpinające grafu o sumarycznie najmniejszej wadze wśród wszystkich drzew rozpinających, których graf n wierzchołkowy może mieć n^{n-2}. Zbiór możliwych rozwiązań zadania plecakowego tworzą wektory o współrzędnych 0 lub 1 - określić należy wektor spełniający jedno ograniczenie liniowe i maksymalizujący zysk wśród wszystkich wektorów dopuszczalnych. Kolorowanie wierzchołków grafu jest równoważne z określeniem funkcji, która sąsiednim wierzchołkom grafu przyporządkowuje różne liczby naturalne. Problem kolorowania grafu polega na znalezieniu pokolorowania najmniejszą liczbą kolorów. W tym

celu, wystarczy rozpatrzyć co najwyżej n^n funkcji kolorujących, gdzie n jest liczbą wierzchołków grafu. (W rzeczywistości, możemy ograniczyć uwagę do $n!$ takich przyporządkowań, por. [Jer], str. 110-12.)

Przyjmijmy, że n jest parametrem określającym wielkość problemu (zadania). We wszystkich trzech przypadkach, chociaż przestrzeń możliwych rozwiązań jest skończona, jej przegląd, gdy n rośnie, szybko przekracza możliwości jakiejkolwiek maszyny cyfrowej, nawet tej, którą człowiek będzie w stanie zbudować w najbliższym dziesięcioleciu. Cała nasza nadzieja leży zatem w efektywnych metodach rozwiązywania, które przeglądają jedynie niewielką część zbioru wszystkich rozwiązań. W sensie mocy istniejących metod rozwiązywania, problemy, które omawiamy, należą do trzech różnych grup. Ilustrujemy tę sytuację w następnych paragrafach podając odpowiednie algorytmy. I tak, problem SST może być rozwiązywany algorytmem *wielomianowym*, czyli takim, który wykonuje liczbę kroków proporcjonalną do stałej potęgi n. Nie są znane natomiast żadne algorytmy wielomianowe rozwiązywania pozostałych dwóch problemów. Problem plecakowy jest jednak w pewnym sensie łatwiejszy od problemu kolorowania, gdyż może być rozwiązywany algorytmem wielomianowym względem liczby zmiennych n oraz maksymalnej wartości parametrów. Problem kolorowania należy do klasy najtrudniejszych problemów kombinatorycznych, dla których nie znaleziono dotychczas (i jest mało prawdopodobne, że istnieją) żadnych efektywnych algorytmów, które dostatecznie dobrze przybliżałyby rozwiązanie optymalne.

Chociaż wybrane przez nas problemy mogą być omawiane niezależnie jeden od drugiego, naszym nadrzędnym celem jest zilustrowanie *zachłannego* podejścia do rozwiązywania zagadnień kombinatorycznych. Nie definiujemy dokładnego znaczenia pojęcia „zachłanności", gdyż jest to możliwe jedynie w wąskiej klasie zagadnień, por. § 5. Na użytek tego opracowania przyjmujemy, że zachłanna metoda rozwiązywania problemu konstruuje rozwiązanie element po elemencie w taki sposób, że kolejno dołączane elementy przyjmują najlepsze z możliwych wartości oraz każde rozwiązanie częściowe jest dopuszczalne, tzn. może być uzupełnione do kompletnego rozwiązania. (Nie omawiamy tutaj zachłanności w sensie lokalnej optymalizacji, która polega na przechodzeniu od jednego rozwiązania do innego rozwiązania, najlepszego w pewnym otoczeniu tego pierwszego.) We wszystkich trzech omawianych przez nas przypadkach, idea zachłanności dostarcza bardzo efektywnych algorytmów rozwiązywania, których dokładność maleje jednak z problemu na problem. Pokażemy najpierw, że algorytm zachłanny znajduje najkrótsze drzewo rozpinające w sieci. Następnie, zastosujemy podejście zachłanne do generowania dostatecznie bliskich rozwiązań dla zadań plecakowych. Na końcu wreszcie, zilustrujemy bardzo złe

zachowanie się pokolorowań grafu otrzymywanych metodą, która w każdym kroku bardzo oszczędnie gospodaruje kolorami.

Od Czytelnika niniejszego opracowania oczekujemy jedynie znajomości podstawowych faktów z zakresu Wstępu do Informatyki. Dla pogłębienia wiedzy w dziedzinach, które omawiamy, polecamy literaturę uzupełniającą w postaci książek: [Lip] i [Wil] - poświęcone kombinatoryce i teorii grafów oraz ich algorytmom, [GW] - omawiającą programowanie (tj. optymalizację) dyskretne oraz [AHU] i [BK] - poświęcone analizie algorytmów i ich złożoności obliczeniowej. Polecamy także pracę [Sys] omawiającą w sposób elementarny bogatszą rodzinę problemów kombinatorycznych oraz efektywne algorytmy ich rozwiązywania wraz z elementami złożoności obliczeniowej.

Teoretyczny opis własności algorytmów zawiera najczęściej jedynie informacje o zachowaniu się algorytmów w najgorszym przypadku danych. Z praktycznego punktu widzenia, niemniej ważne są wyniki eksperymentów maszynowych przeprowadzonych z konkretnymi realizacjami algorytmów. Czytelnika zainteresowanego implementacjami algorytmów kombinatorycznych oraz wynikami obliczeń testowych odsyłamy do zbioru algorytmów w języku ALGOL 60 [KS] i do książki [SDK] zawierającej programy w Pascalu.

2. Problem najkrótszego drzewa rozpinającego

Wyznaczanie najkrótszego drzewa rozpinającego w sieci jest jednym z najpopularniejszych problemów kombinatorycznych pojawiających się zarówno w rozważaniach teoretycznych jak i w zastosowaniach praktycznych.

Niech $G_c = (V,E;c)$ będzie *siecią*, gdzie $G = (V,E)$ jest spójnym grafem symetrycznym, a c jest dowolną funkcją rzeczywistą określoną na zbiorze krawędzi grafu G. Nie tracąc nic na ogólności, możemy przyjąć, że G jest grafem pełnym, czyli że funkcja c określona jest na wszystkich nieuporządkowanych parach wierzchołków $\{u,v\}$. Jeśli bowiem $\{u,v\} \notin E$, to za $c(u,v)$ przyjmujemy dostatecznie dużą liczbę, która gwarantować będzie, iż krawędź $\{u,v\}$ nie znajdzie się w żadnym rozwiązaniu. Wystarczy na przykład by wartość $c(u,v)$ była n razy większa od największej wagi krawędzi. Funkcja c może być rozszerzona na dowolny podzbiór $F \subseteq E$ krawędzi grafu. Dla zbioru pustego przyjmujemy $c(\emptyset) = 0$, a w pozostałych przypadkach $c(F)$ jest sumaryczną wagą krawędzi należących do F, czyli

$c(F) = \sum_{f \in F} c(f)$. Przypomnijmy, że *drzewem rozpinającym* w grafie jest maksymalny podgraf acykliczny.

Problem najkrótszego drzewa rozpinającego (SST).

Dane: Sieć symetryczna $G_c = (V,E;c)$.
Cel: Znaleźć najkrótsze drzewo rozpinające T w sieci G_c.□

Sieć G_c zdefiniowana na grafie pełnym G zawiera n^{n-2} drzew rozpinających (tw. Cayley'a). Zatem na przykład, przestrzeń rozwiązań zadania SST dla 10-cio wierzchołkowej sieci pełnej składa się z 10^8 elementów. Dzięki jednak szczególnej strukturze rodziny drzew rozpinających grafu i ich poddrzew, problem SST może być rozwiązywany w bardzo naturalny sposób. W ostatnim paragrafie opiszemy ogólne własności problemów, które mogą być rozwiązywane w podobnie prosty sposób.

Najpopularniejsza i jednocześnie jedna z najefektywniejszych metod rozwiązywania problemu SST jest realizacją bardzo prostej idei, która poleca budować rozwiązanie krawędź po krawędzi, wybierając kolejno najkrótszą z krawędzi, których dołączenie do rozwiązania częściowego nie powoduje powstania w nim cyklu. Innymi słowy, metoda ta dokonuje w każdym kroku najlepszego wyboru dbając jednocześnie o to by każde rozwiązanie częściowe było dopuszczalne, tzn., dało się uzupełnić do rozwiązania kompletnego. Postępujemy więc w sposób jak najbardziej zachłanny. Uściślimy najpierw opis tej metody, a następnie opiszemy dokładnie jej wariant dający się bardzo łatwo zaprogramować.

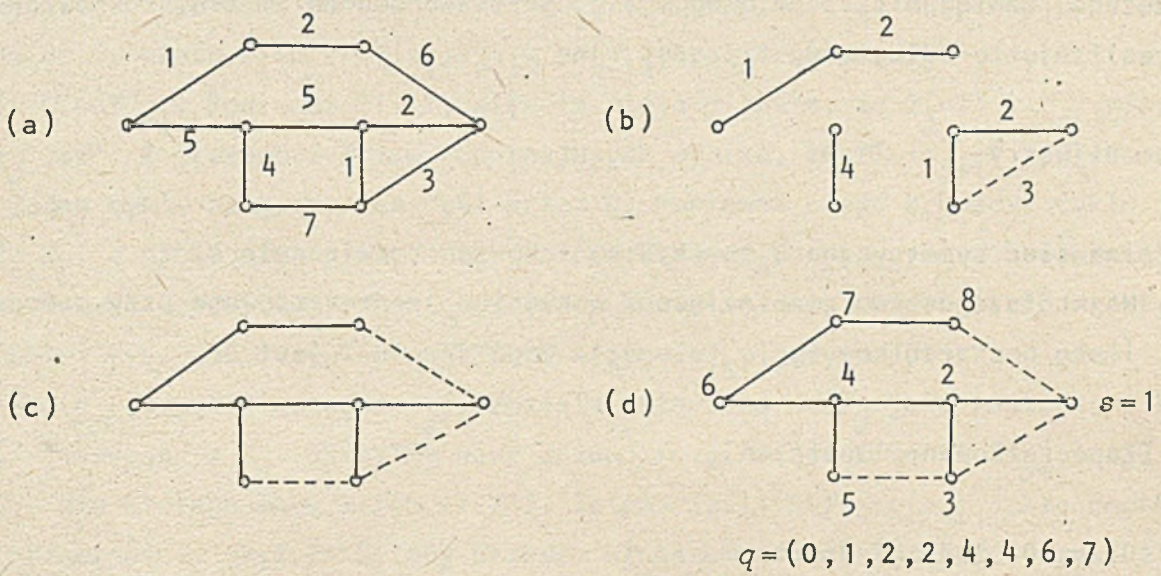
Algorytm zachłanny dla problemu najkrószego drzewa rozpinającego.

Dane: Sieć symetryczna $G_c = (V,E;c)$.
Wynik: Najkrótsze drzewo rozpinające T w sieci G_c.

```
begin
  T←∅; F←E;
  while |T|<(n-1) and F≠∅ do
  begin
    e ← najkrótsza krawędź w F;
    F←F-{e};
    if Tu{e} nie zawiera cyklu then T←Tu{e}
  end
end
```

□

Działanie powyższego algorytmu ilustrujemy na przykładzie sieci z Rys. 1(a). Rysunek 1(b) przedstawia rozwiązanie częściowe po 6-ciu krokach algorytmu, a rozwiązanie kompletne pokazane jest na Rys. 1(c). (Krawędzie należące do rozwiązania oznaczone są linią ciągłą a pozostałe - linią przerywaną.) Zauważmy, że rozwiązanie częściowe może składać się z więcej niż jednej części spójnej.



Rysunek 1.

Algorytm zachłanny może być zmodyfikowany i ulepszany na wiele różnych sposobów. Bardzo prostym usprawnieniem, które uwzględniliśmy już w powyższym opisie, a które często prowadzi do znacznej redukcji liczby iteracji jest sprawdzanie w kroku iteracyjnym mocy zbioru T. Jeśli $|T|$ osiąga n-1, gdzie n jest liczbą wierzchołków grafu, to obliczenia mogą być przerwane, gdyż T jest już drzewem rozpinającym w sieci G_c, a więc żadna krawędź spoza T nie może być dołączona do T. Przy użyciu wyspecjalizowanych struktur danych oraz algorytmów posługiwania się nimi, algorytm zachłanny może być zrealizowany w czasie $O(m \log n)$, gdzie m jest liczbą krawędzi grafu. Zainteresowanego Czytelnika odsyłamy do książek [AHU], [BK] i [SDK].

Opiszemy teraz wersję algorytmu zachłannego podaną przez E.W. Dijkstrę (i niezależnie przez wielu innych autorów), która różni się od algorytmu zachłannego tym, że każde rozwiązanie częściowe jest spójne, a krawędzie są tylko częściowo porządkowane w trakcie działania algorytmu. Algorytm Dijsktry konstruuje najkrótsze drzewo rozpinające T od dowolnie wybranego wierzchołka s grafu G (s nazywamy *korzeniem* drzewa T), i w każdym z n-1 kroków dołącza do rozwiązania częściowego najkrótszą krawędź spośród krawędzi o jednym końcu w rozwią-

zaniu, a drugim spoza. W przedstawionym niżej opisie algorytmu, rozwiązanie T pamiętane jest w postaci *listy poprzedników*, tj. dla każdego wierzchołka v (z wyjątkiem s) określony jest numer wierzchołka bezpośrednio poprzedzającego wierzchołek v na drodze z s do v. Oznaczmy przez p_i oraz q_i pomocnicze cechy wierzchołka i, gdzie p_i jest odległością (w sensie funkcji c) wierzchołka i od bieżącego rozwiązania częściowego, a q_i jest wierzchołkiem tego rozwiązania, który realizuje tę odległość, tj. $c(q_i,i) = p_i$.

Algorytm Dijkstry

Dane: Pełna sieć symetryczna $G_c = (V,E;c)$ i korzeń rozwiązania s.
Wyniki: Najkrótsze drzewo rozpinające T w sieci G_c reprezentowane przy pomocy listy poprzedników $(q_1,q_2,\ldots,q_n)$. Wagą drzewa T jest cT.

Krok 1 [Zapoczątkowanie obliczeń]

$p_s \leftarrow 0$; $q_s \leftarrow 0$; $T \leftarrow \emptyset$; $cT \leftarrow 0$; $r \leftarrow s$;
for każdego $v \in W = V - \{s\}$ do
begin $p_v \leftarrow c_{sv}$; $q_v \leftarrow s$ end;

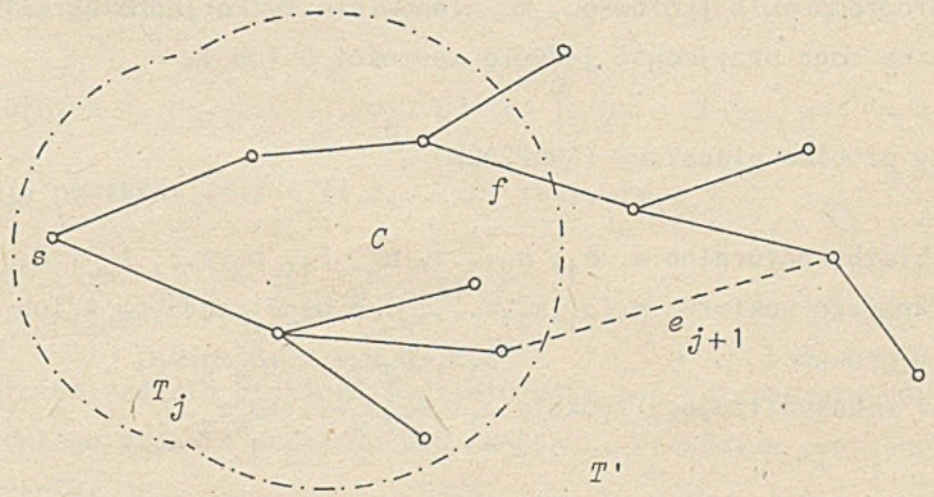
Krok 2 [Iteracja]

for $i = 1,2,\ldots,n-1$ do
begin
  wyznaczyć $p_w = \min\{p_v : v \in W\}$;
  $T \leftarrow T \cup \{q_w, w\}$;
  $cT \leftarrow cT + p_w$;
  $W \leftarrow W - w$;
  for każdego $u \in W$ do
  if $p_u > c_{wu}$ then
  begin $p_u \leftarrow c_{wu}$; $q_u \leftarrow w$ end
end

□

Działanie algorytmu Dijkstry ilustrujemy na przykładzie sieci z Rys. 1(a). Najkrótsze drzewo rozpinające wygenerowane przez ten algorytm przedstawione jest na Rys. 1(d). (Numery wierzchołków oznaczają kolejność, w jakiej wierzchołki dołączane są do rozwiązania.)

Naszkicujemy teraz dowód poprawności algorytmu Dijsktry. Przypuśćmy, że drzewo T wyznaczone przez ten algorytm dla sieci G_c nie jest najkrótszym drzewem rozpinającym, a więc istnieje w G_c inne drzewo T' spełniające $c(T') < c(T)$. Nie tracąc nic na ogólności można przyjąć, że oba te drzewa są zakorzenione w wierzchołku, z którego algorytm rozpoczął tworzyć drzewo T. Przyjmijmy, że krawędzie drzewa T są ponumerowane zgodnie z kolejnością, w jakiej były dołączane do T, niech więc $T = \{e_1, e_2, \ldots, e_{n-1}\}$ oraz oznaczmy $T_i = \{e_1, e_2, \ldots, e_i\}$, czyli $T = T_{n-1}$. Ponieważ $T' \neq T$, więc istnieje j takie, że $T_j \subset T'$ i $T_{j+1} \not\subset T'$, czyli $e_{j+1} \notin T'$. Z własności drzew rozpinających wynika, że $T' \cup \{e_{j+1}\}$ zawiera dokładnie jeden cykl, oznaczmy ten cykl przez C. Ponieważ część krawędzi cyklu C należy do T_j, a część nie należy, więc oprócz krawędzi e_{j+1}, cykl C zawiera jeszcze jedną krawędź, powiedzmy f, której jeden koniec należy, a drugi nie należy do T_j (patrz Rys. 2). Oczywiście $f \in T'$. Z własności algorytmu Dijkstry mamy $c(f) \geq c(e_{j+1})$, gdyż krawędź e_{j+1} została dołączona do T_j jako najkrótsza krawędź wychodząca z T_j. Określmy nowe drzewo rozpinające $T'' = T' \cup \{e_{j+1}\} - \{f\}$ w sieci G_c, dla którego mamy $c(T'') \leq c(T')$. Zatem, jeśli $c(f) > c(e_{j+1})$, to dochodzimy do sprzeczności, gdyż T'' byłoby drzewem krótszym od najkrótszego drzewa T'. Jeśli natomiast $c(f) = c(e_{j+1})$, to $c(T'') = c(T')$ i $T_{j+1} \subset T''$. W tym drugim przypadku powtarzając powyższe rozumowanie, dochodzimy albo do drzewa krótszego od T', albo do drzewa $\tilde{T}$, które spełnia $\tilde{T} = T_{n-1} = T$, $c(\tilde{T}) = c(T')$, czyli także sprzeczność z założeniem dowodu nie wprost, że T nie jest najkrótszym drzewem.



Rysunek 2.

Łatwo jest policzyć, że ilość elementarnych operacji wykonywanych przez algorytm Dijkstry jest $O(n^2)$. W tym celu zauważmy, że Krok 1 wykonuje $O(n)$ operacji, a w i-tej iteracji Kroku 2 wykonujemy $O(n-i)$ operacji związanych z wyzna-

czaniem minimum oraz $O(n-i)$ operacji uaktualniających cechy p_i, q_i.

Porównajmy efektywność obu algorytmów rozwiązywania problemu SST. Dla sieci rzadkich (takimi są na przykład sieci płaskie), dla których m jest $O(n)$, algorytm zachłanny ma złożoność $O(n \log n)$, podczas gdy algorytm Dijkstry działa w czasie $O(n^2)$. Z kolei dla sieci pełnych, porównanie efektywności wypada na korzyść algorytmu Dijkstry. Te porównania, oparte jedynie na asymptotycznym zachowaniu się algorytmów, potwierdzają obliczenia testowe wykonane dla losowych sieci, por. [SDK].

Niemal identyczne rozumowanie jak powyżej można przeprowadzić dla problemu wyznaczania *najdłuższego drzewa rozpinającego* w sieci (w skrócie, LST). Istnieje szereg innych problemów kombinatorycznych, które mogą być rozwiązywane metodą zachłanną - piszemy o tym w Podsumowaniu (§ 5).

3. Problemy plecakowy

3.1. Różne wersje problemu

W paragrafie tym omówimy *problem plecakowy* (ang. *knapsack problem*), zwany także problemem załadunku. Problem ten jest najprostszą wersją całkowitoliczbowego programowania liniowego, ma mianowicie tylko jedno ograniczenie liniowe, a zmienne mogą przyjmować jedynie wartości 0 lub 1.

Binarny problem plecakowy (KNAPSACK)

Dane: Liczby naturalne n, a_1, a_2,..., a_n, b_1, b_2,..., b_n i B.
Cel: Znaleźć wektor $x = (x_1, x_2, \ldots, x_n)$ o współrzędnych 0 lub 1 spełniający nierówność
$$b_1x_1+b_2x_2+\ldots+b_nx_n \leq B,$$
i maksymalizujący funkcję $a_1x_1+a_2x_2+\ldots+a_nx_n$. □

Mimo swej prostoty, problem plecakowy jest w pewnym sensie reprezentantem wszystkich problemów programowania liniowego w liczbach całkowitych. Każdemu zadaniu programowania całkowitoliczbowego można bowiem przyporządkować równoważne mu zadanie plecakowe. Nie będziemy dokładniej omawiać tej zależności, gdyż nie ma ona zbyt wielkiego znaczenia praktycznego, ponieważ zadania plecakowe, które powstają w ten sposób, mają niewspółmiernie dużą liczbę zmiennych. Przed-

stawimy natomiast dwie inne wersje problemu plecakowego by zilustrować na ich przykładzie kilka typowych faktów. Ponieważ klasyczne, dokładne metody rozwiązywania problemu plecakowego są opisane w łatwo dostępnych opracowaniach (por. [GN], a także [SDK]), swoją uwagę skupimy przede wszystkim na przybliżonych metodach rozwiązywania. Uzasadnione to jest także tym, że problem plecakowy należy do trudnych problemów kombinatorycznych, mało jest więc prawdopodobne, by którakolwiek z dokładnych metod rozwiązywania była efektywna (tj. wielomianowa). Decyzyjny problem plecakowy ilustruje sposób formułowania problemów decyzyjnych dla problemów optymalizacyjnych.

Decyzyjny binarny problem plecakowy (DEC-KNAPSACK)

Dane: Liczby naturalne n, a_1, a_2,..., a_n, b_1, b_2,..., b_n, B i K.
Pytanie: Czy istnieje wektor $x = (x_1, x_2, \ldots, x_n)$ o współrzędnych 0 lub 1 spełniający następujące nierówności

$$b_1x_1+b_2x_2+\ldots+b_nx_n \leq B,$$
$$a_1x_1+a_2x_2+\ldots+a_nx_n \geq K.\square$$

Jedną z wersji problemu DEC-KNAPSACK jest następujący problem podziału zbioru liczb na dwa podzbiory o tej samej sumie wartości ich elementów.

Problem podziału (PARTITION)

Dane: Liczby naturalne n, a_1, a_2,..., a_n, gdzie $\sum_{i=1}^{n} a_i = 2b$.

Pytanie: Czy istnieje podzbiór $I' \subset I = \{1,2,\ldots,n\}$ taki, że

$$\sum_{i \in I'} a_i = \sum_{i \in I-I'} a_i = b? \square$$

Rozwiązanie każdego zadania problemu podziału może być sprowadzone do rozwiązania odpowiedniego zadania problemu DEC-KNAPSACK, prawdziwe jest bowiem następujące stwierdzenie:

Zadanie problemu PARTITION o danych n, a_1, a_2,..., a_n, *gdzie* $\sum_{i=1}^{n} a_i = 2b$ *ma rozwiązanie TAK wtedy i tylko wtedy, gdy zadanie problemu DEC-KNAPSACK o danych* n, a_1, a_2,..., a_n, a_1, a_2,..., a_n, b, b *ma rozwiązanie TAK.*

Zatem każde zadanie problemu PARTITION może być łatwo przetransformowane w zadanie problemu DEC-KNAPSACK, czyli każdy algorytm rozwiązywania tego drugiego problemu może być zastosowany do rozwiązywania pierwszego problemu. Ta zależność między problemami PARTITION i DEC-KNAPSACK ilustruje często spotykane relacje między problemami kombinatorycznymi - w szczególności, i matematycznymi - w ogólności, gdy rozwiązanie jednego problemu może być sprowadzone do rozwiązania innego problemu. Transformacje takie odgrywają podstawową rolę w złożoności obliczeniowej.

Przestrzeń możliwych rozwiązać powyższych problemów składa się z wektorów o współrzędnych 0 lub 1. Zatem zadanie plecakowe z n zmiennymi ma co najwyżej 2^n możliwych rozwiązań. Dla $n = 10$, mamy $2^{10} = 1024$ możliwych rozwiązań, a więc znacznie mniej (aż o pięć rzędów), niż graf o 10 wierzchołkach ma drzew rozpinających. Niestety, w dziedzinie metod rozwiązywania obu problemów, sytuacja jest do pewnego stopnia odwrotna. Problem SST może być rozwiązywany algorytmem o złożoności $O(n^2)$, podczas gdy nie jest znany żaden algorytm rozwiązywania problemu plecakowego, którego pracochłonność byłaby ograniczona przez wielomian zmiennej n i mało jest prawdopodobne, że taka metoda istnieje.

Opis przybliżonej metody rozwiązywania problemu plecakowego poprzedzimy podaniem dokładnego algorytmu dla problemu podziału, który bazuje na schemacie programowania dynamicznego. Algorytm ten, mimo iż z teoretycznego punktu widzenia jest metodą nieefektywną, z powodzeniem może być stosowany do rozwiązywania praktycznych zadań, w których parametry należą do dość wąskiego przedziału zmienności.

3.2. Algorytm dokładny

Algorytm programowania dynamicznego, który rozwiązuje problem podziału, wyznacza wartości funkcji boolowskiej (tj. funkcji o wartościach TAK lub NIE) $T(i,j)$ dla $1 \le i \le n$, $1 \le j \le b$. Wartością $T(i,j)$ jest TAK, jeśli wśród liczb a_1, $a_2,\ldots, a_i$ istnieje podzbiór, którego suma jest równa j, natomiast NIE, w przeciwnym przypadku. Innymi słowy, $T(i,j)$ jest TAK wtedy, gdy $\sum_{k=1}^{i} a_k x_k = j$ dla x_1, $x_2,\ldots, x_i$ przyjmujących wartości 0 lub 1, a NIE - w przeciwnym razie. Zatem zadanie problemu podziału o danych n, a_1, $a_2,\ldots, a_n$ ma rozwiązanie TAK wtedy i tylko wtedy, gdy $T(n,b)$ ma wartość TAK. Wartość $T(n,b)$ liczymy z następującej zależności rekurencyjnej $T(n,b) = T(n-1,b) \cup T(n-1,b-a_n)$, w której pierwszy człon

po prawej stronie odpowiada rozwiązaniu zadania, w którym $x_n = 0$, a drugi - gdy $x_n = 1$. Ogólnie, prawdziwa jest następująca zależność

$$T(i,j) = T(i-1,j) \cup T(i-1,j-a_i)$$

dla $j = 1, 2, \ldots, b$, $i = 2, 3, \ldots, n$. W przypadku $i = 1$, przyjmujemy, że $T(1,a_1)$ jest TAK i $T(1,j)$ jest NIE dla $j \neq a_1$. Ponadto określamy, że $T(i,j)$ jest NIE dla $j \leq 0$. Powyższą zależność rekurencyjną stosujemy dla $i = 1$, $j = 1, 2, \ldots, b$, $i = 2$, $j = 1, 2, \ldots, b, \ldots, i = n$, $j = 1, 2, \ldots, b$. Następująca tablica ilustruje zastosowanie tej zależności do rozwiązania zadania podziału o parametrach $n = 4$, $a_1 = 2$, $a_2 = 3$, $a_3 = 5$, $a_4 = 6$. Element $T(4,8)$ jest T = TAK, zatem zadanie to ma rozwiązanie, którym jest np. a_1+a_4 lub a_2+a_3.

i \ j	1	2	3	4	5	6	7	$b = 8$
1	N	T	N	N	N	N	N	N
2	N	T	T	N	T	N	N	N
3	N	T	T	N	T	N	T	T
$n = 4$	N	T	T	N	T	N	T	T

Rozwiązanie problemu podziału powyższą metodą wymaga wypełnienia $n \times b$ elementów tablicy o n wierszach i b kolumnach, zatem czas potrzebny do wykonania wszystkich operacji z tym związanych jest ograniczony przez $O(n \cdot b)$. Wyrażenie $n \cdot b$ jest wielomianem zmiennych n i b, nie jest jednak wielomianem rozmiaru problemu, gdzie za rozmiar problemu przyjmujemy liczbę bitów potrzebnych do zapamiętania danych. Policzmy więc, jaki jest rozmiar problemu podziału, a więc ile potrzeba bitów do zapamiętania danych n, a_1, $a_2, \ldots, a_n$. Każda z liczb a_i wymaga co najwyżej $\lceil \log_2 a_i \rceil + 1$ bitów, zatem cały układ danych może być zapamiętany na $\sum_{i=1}^{n} \lceil \log_2 a_i \rceil + n$ bitach. Ponieważ $a_i \leq b$ dla każdego i, więc rozmiar problemu podziału jest co najwyżej $c \cdot n \cdot \log_2 b$, gdzie c jest pewną stałą. Wielomian $n \cdot b$ nie jest zaś ograniczony przez żaden wielomian zmiennej $n \cdot \log_2 b$, gdyż b nie jest ograniczone przez $\log_2^k b$ dla żadnej stałej naturalnej k, bowiem $b / \log_2^k b \to \infty$, gdy $b \to \infty$, dla każdej stałej k.

Powyższy schemat wyznaczania $T(n,b)$ nie jest więc algorytmem wielomianowym

względem rozmiaru problemu, ale ta niewielomianowość jest wynikiem dopuszczenia do obliczeń liczb o nieograniczonych wartościach. Algorytm ten jest jednak wielomianowy względem liczby danych n i maksymalnej liczby występującej w obliczeniach. Takie algorytmy nazywamy *pseudowielomianowymi*. Problem kolorowania grafów, omówiony w następnym paragrafie, nie ma niestety nawet algorytmu pseudowielomianowego. W tym właśnie sensie problem plecakowy jest łatwiejszy do rozwiązywania niż problem kolorowania.

3.3. Algorytm przybliżony

Zanim podamy przybliżony algorytm rozwiązywania problemu plecakowego, wykażemy najpierw, że wyznaczanie rozwiązań absolutnie przybliżonych jest tak samo trudne, jak wyznaczanie rozwiązań optymalnych. Mówimy, że algorytm rozwiązywania problemu KNAPSACK wyznacza *rozwiązania absolutnie ε-przybliżone*, jeśli dla dowolnego zadania tego problemu algorytm ten generuje rozwiązanie o wartości K' różniącej się od wartości optymalnej K_{opt} o nie więcej niż ε, czyli

$$K_{opt}-K' \leq \varepsilon.$$

W tym celu, wraz z zadaniem o danych n, $a_1, a_2, \ldots, a_n, b_1, b_2, \ldots, b_n, B$, rozpatrzmy zadanie o danych n, $(k+1)a_1, (k+1)a_2, \ldots, (k+1)a_n, b_1, b_2, \ldots, b_n, B$. Rozwiązanie optymalne drugiego z tych zadań ma wartość $(k+1)$ razy większą od wartości pierwszego zadania. Oznaczmy przez K_{opt} i K^{k+1}_{opt} wartości rozwiązań optymalnych tych zadań, a przez K i K^{k+1} wartości rozwiązań absolutnie ε-przybliżonych, dla $\varepsilon = k$. Mamy więc

$$K^{k+1}_{opt}-K^{k+1} \leq k,$$

a stąd $(k+1)(K_{opt}-K) \leq k$, czyli

$$K_{opt}-K \leq \frac{k}{k+1} < 1.$$

Ponieważ wszystkie liczby występujące w obu zadaniach są całkowite, z ostatniej nierówności wynika, że $K_{opt} = K$. Wnioskujemy stąd, że ewentualny algorytm wyznaczający rozwiązania absolutnie k-przybliżone dla zadań zmodyfikowanych wyznaczałby jednocześnie rozwiązania optymalne dla wszystkich zadań plecakowych. Istnienie wielomianowego algorytmu generującego rozwiązania absolutnie przybliżone jest mało prawdopodobne.

Przedstawimy teraz ogólny schemat metody, która dla dowolnego zadania problemu KNAPSACK i ustalonego $\varepsilon > 0$ wyznacza rozwiązanie ε-*przybliżone* o wartości K_ε, której *względny* błąd nie jest większy od ε, czyli

$$\frac{K_{opt}-K_\varepsilon}{K_{opt}} \leq \varepsilon.$$

Metoda, którą przedstawimy, jest rodziną n algorytmów, gdzie n jest liczbą zmiennych, z której dla każdego $\varepsilon > 0$ można zawsze wybrać algorytm generujący rozwiązania ε-przybliżone. Rozpatrzmy najpierw zadanie programowania liniowego stowarzyszone z zadaniem problemu KNAPSACK, a powstałe przez zastąpienie ograniczenia $x_i = 0$ lub 1 ograniczeniem $0 \leq x_i \leq 1$ ($i = 1, 2, \ldots, n$). By rozwiązać zadanie stowarzyszone, najpierw ustawiamy zmienne w nierosnącym porządku ilorazów a_i/b_i, a następnie rozpatrując je w takiej kolejności przydzielamy im możliwie największe wartości, aż do napięcia ograniczenia $\sum_{i=1}^{n} b_i x_i \leq B$. Idea ta przeniesiona na problem plecakowy sugeruje, że dla osiągnięcia najcenniejszej zawartości powinniśmy wypełniać plecak towarami w kolejności nierosnących ilorazów a_i/b_i, które można uważać za względne zyski. Następujący przykład pokazuje, że taka strategia nie zawsze jednak generuje rozwiązanie optymalne:

zmaksymalizować $60x_1+35x_2+30x_3$
przy ograniczeniach $3x_1+2x_2+2x_3 \leq 4$
$x_1, x_2, x_3 = 0$ lub 1.

Biorąc pod uwagę względne zyski, towary powinny być rozpatrywane w kolejności x_1, x_2, x_3, co prowadzi do rozwiązania (1,0,0). Takie samo rozwiązanie otrzymujemy rozpatrując zmienne w nierosnącej kolejności bezwzględnych zysków, tj. współczynników funkcji celu. Rozwiązaniem optymalnym powyższego zadania jest zaś wektor (0,1,1). Chociaż zachłanne sposoby wypełniania plecaka nie zawsze zapewniają optymalną zawartość, podejście to z powodzeniem stosowane jest w algorytmie wyznaczającym rozwiązania względnie przybliżone.

Załóżmy dla wygody, że zmienne w zadaniu plecakowym są już uporządkowane tak, by spełnione były nierówności $a_1/b_1 \geq a_2/b_2 \geq \ldots \geq a_n/b_n$. Niech $I \subset N = \{1, 2, \ldots, n\}$ będzie podzbiorem towarów mieszczących się w plecaku, tj. $\sum_{i \in I} b_i \leq B$. Uzupełnienie zbioru I do możliwie najcennjejszego ładunku polega, zgodnie z po-

wyższą ideą, na dokładaniu do plecaka towarów w porządku wzrastania ich numerów. Oznaczmy przez $L(I)$ wartość dodatkowej zawartości otrzymanej w ten sposób dla początkowego ładunku I. Wartość $L(I)$ liczymy w następujący sposób:

```
begin
  L←0; C←B- Σ_{i∈I} b_i;
  for i←1 to n do
    if i∉I and b_i≤C then
    begin L←L+a_i; C←C-b_i end
end
```

Niech k będzie liczbą naturalną z przedziału $(1,n)$. Algorytm Knapsack(k) wyznacza rozwiązanie binarnego zadania plecakowego, najlepsze jakie może być otrzymane przez wybór dowolnych k towarów mieszczących się w plecaku, a następnie uzupełnienie ich metodą zachłanną do najcenniejszego załadunku. Oznaczmy przez $Zysk(k)$ wartość rozwiązania otrzymanego algorytmem Knapsack(k).

Algorytm Knapsack(k)

Dane: Parametry zadania plecakowego n, a_1, a_2,..., a_n, b_1, b_2,..., b_n, B.
Wynik: $Zysk$ - wartość najlepszego rozwiązania otrzymanego metodą zachłanną z podzbioru co najwyżej k towarów mieszczących się w plecaku.

```
begin
  Zysk←0;
  for każdego podzbioru I⊂{1,2,...,n} takiego, że |I|≤k
    and Σ_{i∈I} b_i ≤B do
    Zysk←max{Zysk, Σ_{i∈I} a_i+L(I)}
end
```

□.

S. Sahni, który jest autorem prezentowanej metody, udowodnił, że wartość rozwiązania $Zysk(k)$ spełnia nierówność

$$\frac{K_{opt}-Zysk(k)}{K_{opt}} \leq \frac{1}{k+1}.$$

Zatem, by otrzymać rozwiązanie ε-przybliżone należy wybrać k spełniające

$1/(k+1) \leq \varepsilon$, czyli $k \geq 1/\varepsilon - 1$. Łatwo sprawdzić, że algorytm Knapsack(k) może być zrealizowany w czasie $O(kn^{k+1})$, jest to więc algorytm wielomianowy zmiennej n (pamiętajmy, że k jest stałą).

Zastosujmy powyższą klasę algorytmów przybliżonych do rozwiązania następującego zadania plecakowego:

zmaksymalizować $200x_1+155x_2+115x_3+90x_4$
przy warunkach $50x_1 + 40x_2 + 30x_3+25x_4 \leq 95$
i $x_1, x_2, x_3, x_4 = 0$ lub 1.

Następująca tablica ilustruje działanie algorytmów Knapsack(k) dla kolejnych $k = 1, 2, 3$. Rozwiązanie optymalne, którym jest (0,1,1,1), generowane jest przez Knapsack(2).

$k = 1$	I	{1}	{2}	{3}	{4}		
	Zysk	355	355	315	290		
	Rozwiązanie	(1100)	(1100)	(1010)	(1001)		
$k = 2$	I	{1,2}	{1,3}	{1,4}	{2,3}	{2,4}	{3,4}
	Zysk	355	315	290	360	360	360
	Rozwiązanie	(1100)	(1010)	(1001)	(0111)	(0111)	(0111)
$k = 3$	I	{2,3,4}	Pozostałe grupy 3 towarów są				
	Zysk	360	niedopuszczalnymi ładunkami.				
	Rozwiązanie	(0111)					

4. Kolorowanie wierzchołków grafu

4.1. Problem i jego proste własności

Paragraf ten poświęcimy kolorowaniu wierzchołków grafu w taki sposób, by wierzchołki sąsiednie pomalowane były różnymi kolorami. Formalnie, *k-kolorowaniem* grafu $G = (V,E)$ nazywamy funkcję $f: V \xrightarrow{na} \{1,2,\ldots,k\}$, spełniającą $f(i) \neq f(j)$ jeśli $\{i,j\} \in E$. *Liczbą chromatyczną* $\chi(G)$ grafu G nazywamy najmniejsze k, dla którego G ma k-pokolorowanie. Każdy graf można pomalować $n = |V|$ kolorami. Wynika stąd, że dla wyznaczenia liczby $\chi(G)$ wystarczy rozpatrzyć n^n funkcji ko-

lorujących wierzchołki grafu. To oszacowanie liczby możliwych pokolorowań może być zmniejszone do $n!$ przez wyeliminowanie tych, które powstają z innych przez prostą zmianę nazwy kolorów (por. [Jer], str. 110-12).

Problem k-pokolorowania grafu

Dane: Graf $G = (V,E)$ i liczba naturalna k $(1 \leq k \leq |V|)$.
Cel: Stwierdzić czy G może być pomalowany k kolorami. Jeśli tak to wyznaczyć takie jedno k-pokolorowanie.□

Zauważmy, że potrafiąc efektywnie rozwiązywać powyższy problem dla dowolnego k, będziemy w stanie znaleźć liczbę chromatyczną grafu oraz pomalować graf najmniejszą liczbą kolorów. W tym celu wystarczy rozpatrzyć k-pokolorowanie dla kolejnych $k = n, n-1, n-2,...$ aż do otrzymania negatywnej odpowiedzi. Ostatnie k, dla którego G może być pomalowany k kolorami jest liczbą chromatyczną grafu G.

Wśród najpopularniejszych zastosowań problemu kolorowania grafu, wymienić można układanie rozkładów zajęć oraz harmonogramowanie prac, por. [SDK] i [Coff]. Jednak swoją największą popularność, kolorowanie grafu zawdzięcza problemowi czterech kolorów, który polega na określeniu najmniejszej liczby barw potrzebnych do pomalowania administracyjnej mapy państw tak, aby żadne dwa państwa sąsiadujące ze sobą granicą dłuższą od punktu nie były tych samych kolorów. Mniej więcej od połowy zeszłego wieku, aż do 1976 roku znane było przypuszczenie, że każda mapa może być pomalowana czterema kolorami i dopiero w połowie 1976 roku, K. Appel, W. Haken i J. Koch, przy istotnym wykorzystaniu maszyny cyfrowej, wykazali prawdziwość tej hipotezy. Malowanie mapy jest w gruncie rzeczy kolorowaniem specjalnych grafów. Otóż, każdej mapie państw można przyporządkować graf, w którym wierzchołki odpowiadają państwom (są na przykład ich stolicami) i dwa wierzchołki są sąsiednie jeśli odpowiadające im państwa graniczą ze sobą. Malowanie mapy sprowadza się w ten sposób do malowania grafów sąsiedztwa państw (por. [Ste] i [SDK]), są to tzw. grafy płaskie.

Kolorowanie grafów należy do najtrudniejszych problemów kombinatorycznych. Ponieważ algorytmy dokładne są mało efektywne, wiele uwagi poświęca się efektywnym algorytmom heurystycznym. Jednym z niewielu rezultatów dokładnych jest twierdzenie Königa charakteryzujące grafy, które mogą być malowane dwoma kolorami:

Graf G jest 2-chromatyczny wtedy i tylko wtedy, gdy zawiera co najmniej jedną krawędź i nie zawiera cykli nieparzystej długości.

Twierdzenie to można udowodnić podając algorytm, który maluje dwoma kolorami każdy graf nie zawierający nieparzystych cykli. (Zauważmy, że cykl nieparzystej długości nie może być pomalowany dwoma kolorami.) Zatem, charakteryzacja 2-chromatycznych grafów jest efektywna. Nie jest znana natomiast żadna efektywna charakteryzacja grafów k-chromatycznych dla $k \geq 3$ nawet w klasie grafów płaskich, tzn. odpowiadających mapom.

Podamy teraz dwa proste oszacowania liczby chromatycznej, a w następnym punkcie podamy dalsze oszacowania, wynikające z pewnych heurystycznych metod kolorowania. Zauważmy, że jeśli graf G zawiera *klikę* (tj. podgraf pełny) złożoną z k wierzchołków, to $\chi(G) \geq k$, gdyż każdy wierzchołek tej kliki musi być pomalowany innym kolorem. Mamy więc

(1) $\omega(G) \leq \chi(G)$,

gdzie $\omega(G)$ jest *liczbą klikową* grafu, tj. mocą największej kliki grafu G. Z drugiej zaś strony,

(2) $\chi(G) \leq \Delta(G)+1$,

gdzie $\Delta(G)$ jest maksymalnym stopniem wierzchołka w grafie G. Nierówność ta wynika ze spostrzeżenia, że spośród $k = \Delta(G)+1$ dostępnych kolorów w każdym wierzchołku v, zawsze jeden kolor jest wolny, gdyż v ma mniej niż k sąsiadów. Wspólną cechą obu oszacowań (1) i (2) jest istnienie wielu nieskończonych klas grafów, dla których stają się one dowolnie złe. J. Mycielski i W.T. Tutte skonstruowali nieskończony ciąg grafów F_n, które nie zawierają trójkątów, czyli $\omega(F_n) = 2$, i $\chi(F_n) = n$. (Graf F_4 pokazany jest na Rys. 3.) Z drugiej strony, gwiazda S_n z n promieniami jest grafem 2-chromatycznym i $\Delta(S_n) = n$. Z praktycznego punktu widzenia, oszacowania (1) i (2) należą jednak do różnych klas. Wyznaczanie liczby $\omega(G)$ jest bowiem tak samo trudne jako obliczanie $\chi(G)$, natomiast wartość $\Delta(G)$ może być znaleziona w czasie $O(m)$.

4.2. Sekwencyjne metody kolorowania

Omówimy teraz sekwencyjne metody kolorowania grafów, a wśród nich klasę

efektywnych algorytmów przybliżonych oraz metodę bazującą na przeglądzie z powracaniem, która znajduje rozwiązania optymalne. *Algorytm sekwencyjny* rozpatruje wierzchołki grafu w ustalonym porządku i maluje kolejny wierzchołek możliwie najmniejszym kolorem. Zatem, kolory przydzielane są w możliwie najoszczędniejszy sposób. Zauważmy, że tak określona ogólna idea postępowania nie wyklucza możliwości otrzymywania optymalnych pokolorowań. Wystarczy w tym celu przyjąć, że wierzchołki grafu uporządkowane są zgodnie z kolorami: najpierw wierzchołki koloru 1, potem koloru 2 itd.

Algorytm sekwencyjny

Dane: Graf G oraz uporządkowanie jego wierzchołków $v_1, v_2, \ldots, v_n$.
Wynik: Przyporządkowanie f określające pokolorowanie grafu G otrzymane metodą sekwencyjną.

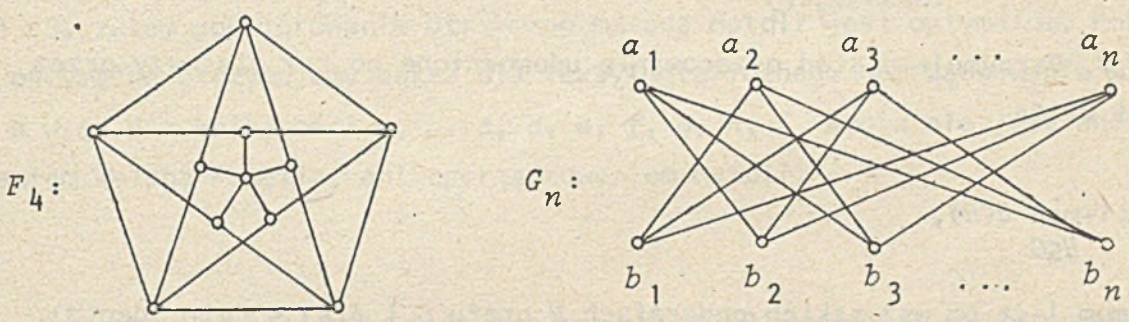
```
begin
  f(v_1)←1;
  for i←2,3,...,n do
    f(v_i)←{min k: k≥1 oraz f(v_j) ≠ k dla każdego v_j (1 ≤ j < i) sąsiedniego
            z v_i}
end                                                              □
```

Oszacujemy liczbę kolorów użytych przez ten algorytm. Zauważmy najpierw, że dla każdego i mamy $f(v_i) \leq i$, gdyż i-ty w kolejności wierzchołek może być zawsze pomalowany i-tym kolorem. Z drugiej zaś strony, kolorami zabronionymi dla $f(v_i)$ są kolory sąsiadów wierzchołka v_i, wśród wierzchołków $v_1, v_2, \ldots, v_{i-1}$, których jest co najwyżej $\deg(v_i)$, gdzie $\deg(v)$ jest liczbą wierzchołków sąsiednich z v. Zatem $f(v_i) \leq \deg(v_i)+1$. Otrzymujemy więc $f(v_i) \leq i$ oraz $f(v_i) \leq \deg(v_i)+1$, czyli $f(v_i) \leq \min\{i, \deg(v_i)+1\}$, dla $i = 1, 2, \ldots, n$. Oznaczmy przez $h(G)$ liczbę kolorów użytych przez algorytm sekwencyjny zastosowany do uporządkowania $v_1, v_2, \ldots, v_n$. Mamy więc

$$\chi(G) \leq h(G) \leq \max_{1 \leq i \leq n} \min\{i, \deg(v_i)+1\} = h_s(G).$$

Wielkość $h_s(G)$ jest jedynie oszacowaniem liczby $h(G)$. W praktyce, co wykazały wielokrotnie wykonane eksperymenty maszynowe, liczba użytych kolorów $h(G)$ jest znacznie mniejsza niż $h_s(G)$. Oszacowanie $h_s(G)$ dla uporządkowania wierzchołków spełniającego $\deg(v_1) \leq \deg(v_2) \leq \ldots \leq \deg(v_n)$ staje się znanym już

nam oszacowaniem $h_s(G) = \Delta(G)+1$. Postać wyrażenia określającego $h_s(G)$ sugeruje natomiast, że $h_s(G)$ jest najmniejsze dla odwrotnego uporządkowania wierzchołków. Metodę sekwencyjną z uporządkowaniem wierzchołków spełniającym nierówności $\deg(v_1) \geq \deg(v_2) \geq \ldots \geq \deg(v_n)$ zaproponowali jako pierwsi D.J.A. Welsh i M.B. Powell. Oznaczmy przez $h_{WP}(G)$ wartość $h_s(G)$ w tym przypadku. Istnieją niestety klasy grafów, dla których wartość $h_{WP}(G)$ może dowolnie daleko odbiegać od $\chi(G)$. Dla przykładu, zastosujmy algorytm sekwencyjny do pomalowania wierzchołków grafu G_n z Rys. 3 w kolejności $a_1, b_1, a_2, b_2, \ldots, a_n, b_n$, będącej uporządkowaniem Welsha-Powella. Każdy graf G_n jest 2-chromatyczny, gdyż $f(a_j) = 1$ i $f(b_j) = 2$ jest pokolorowaniem właściwym. Natomiast algorytm sekwencyjny używa aż n kolorów, a także $h_{WP}(G_n) = n$.



Rysunek 3.

Wyprowadzając oszacowanie $h_s(G)$ wykazaliśmy, że $f(v_i) \leq \deg(v_i)+1$. W rzeczywistości, prawdziwa jest nierówność $f(v_i) \leq \deg_i(v_i)+1$, gdzie $\deg_i(v_i)$ jest stopniem wierzchołka v_i w podgrafie generowanym przez wierzchołki $\{v_1, v_2, \ldots, v_i\}$. Otrzymujemy stąd jeszcze jedno oszacowanie liczby kolorów użytych przez algorytm sekwencyjny

$$\chi(G) \leq h(G) \leq h'_s(G) = \max_{1 \leq i \leq n} \deg_i(v_i)+1.$$

D. Matula udowodnił, że kolejność wierzchołków minimalizująca wielkość $h'_s(G)$ po wszystkich $n!$ możliwych uporządkowaniach może być wyznaczona w następujący prosty sposób:

(i) Za v_n przyjmij wierzchołek o minimalnym stopniu w G.
(ii) Dla $i = n-1, n-2, \ldots, 1$
wybierz wierzchołek v_i o minimalnym stopniu w podgrafie indukowanym przez $V_i = V-\{v_n, v_{n-1}, \ldots, v_{n+1}\}$.

Algorytm sekwencyjny zastosowany do powyższego uporządkowania wierzchołków nazywamy algorytmem Matuli. Podobnie jak algorytm Welsha-Powella, także i w tym przypadku, liczba użytych kolorów jest znacznie mniejsza niż wynosi wartość oszacowania $h_s'(G)$, a także istnieją klasy grafów o ograniczonej liczbie chromatycznej, które malowane są tą metodą bardzo źle. Zauważmy, że dla uporządkowania Matuli

$$\deg_i(v_i) = \min_{1\le j\le i} \deg_i(v_j).$$

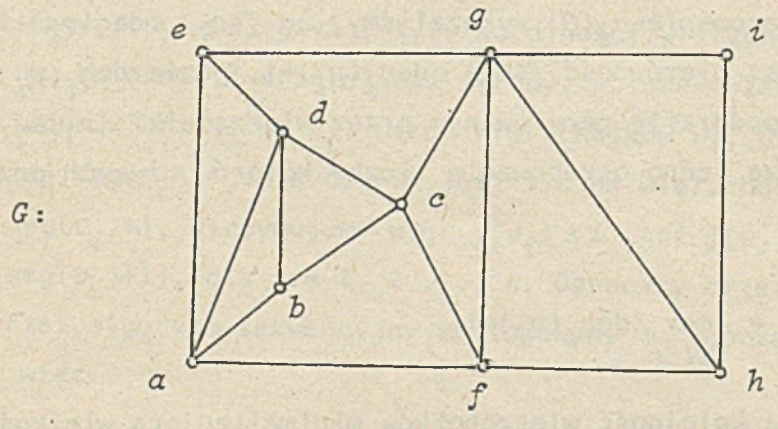
Zatem

$$\chi(G) \le h_s'(G) = 1 + \max_{1\le i\le n} \min_{v_j\in V_i} \deg_i(v_j),$$

gdzie $V_n = V$. Otrzymujemy stąd oszacowanie udowodnione po raz pierwszy przez Szekeresa i Wilfa.

$$\chi(G) \le 1+\max_{H\subseteq G} \delta(H),$$

gdzie maximum jest po wszystkich podgrafach H grafu G i $\delta(H) = \min_{v\in V(H)} \deg(v)$. Ostatnie oszacowanie jest bardzo pożyteczne w przypadku grafów, w których każdy podgraf ma wierzchołek o niewielkim stopniu. Np. dla grafu płaskiego G mamy $\delta(G) \le 5$.



Rysunek 4.

Zastosujmy teraz algorytmy Welsha-Powella i Matuli do pomalowania grafu G z Rys. 4. W rezultacie otrzymujemy:

			liczba użytych kolorów	oszacowania
Algorytm Welsha-Powella	kolejność wierzchołków	$g\ a\ c\ d\ f\ b\ e\ h\ i$	4	$h_{WP}(G) = 5$
	kolory	$1\ 1\ 2\ 3\ 3\ 4\ 2\ 2\ 3$		
Algorytm Matuli	kolejność wierzchołków	$a\ b\ d\ c\ e\ f\ g\ h\ i$	3	$h_{M}(G) = 4$
	kolory	$1\ 2\ 3\ 1\ 2\ 2\ 3\ 1\ 2$		

Zauważmy, że w tym przypadku $\chi(G) = 3$, gdyż G zawiera trójkąt, a więc $\omega(G) \geq 3$; zatem pokolorowanie otrzymane metodą Matuli jest optymalne. Pokolorowanie optymalne otrzymujemy także dla leksykograficznego uporządkowania wierzchołka, a więc dla kolejności $a, b, c, d, e, f, g, h, i$, która nie jest ani uporządkowaniem Welsha-Powella, ani uporządkowaniem Matuli.

Powyższy przykład potwierdza to, co powiedzieliśmy wyżej o obu wersjach metody sekwencyjnej, a mianowicie, że wyznaczane pokolorowania są lepsze od odpowiednich oszacowań wynikających z działania tych algorytmów. Czytelnik nie powinien mieć większego kłopotu z napisaniem efektywnych realizacji obu algorytmów sekwencyjnych. Oba algorytmy mogą być wykonane w czasie $O(m)$, por. [SDK], a także [K].

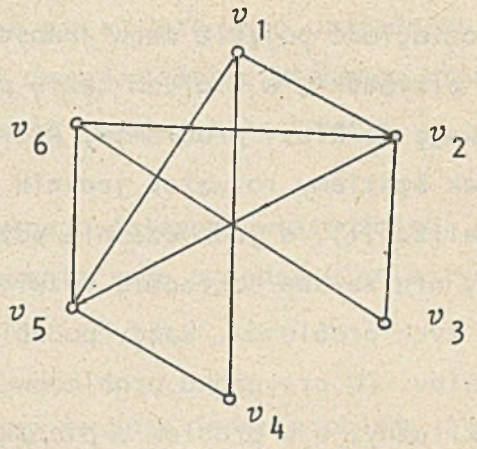
Algorytm sekwencyjny może być usprawniony przez wprowadzenie możliwości wymiany dwóch kolorów w pewnych podgrafach, dzięki której można uwolnić jeden z kolorów w sytuacji, gdy oryginalny algorytm zmuszany jest do użycia nowego koloru. Ta modyfikacja może być zrealizowana także w czasie wielomianowym, nie prowadzi jednak do polepszenia oszacowania liczby chromatycznej w ogólnym przypadku.

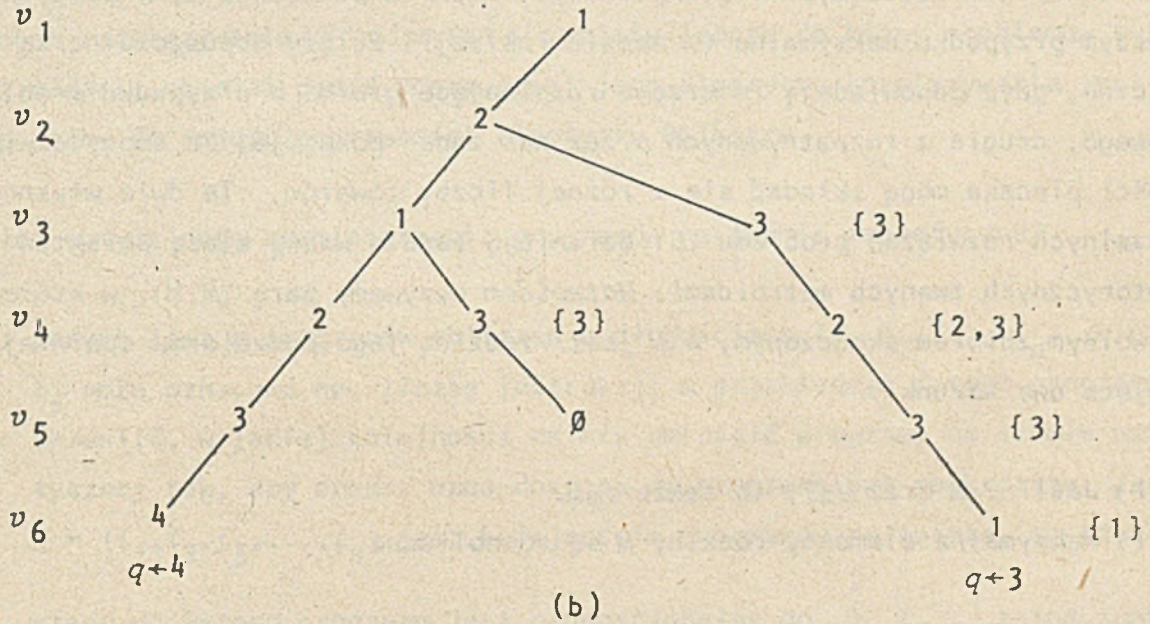
O słabości przybliżonych metod kolorowania grafów świadczyć może fakt udowodniony przez M.R. Garey'a i D.S. Johnsona. Wykazali oni, że jeżeli znana będzie efektywna metoda kolorowania grafów G przy pomocy mniej niż $2 \cdot \chi(G)$ kolorów, to będzie można efektywnie wyznaczać chromatyczne pokolorowania grafów. Ponieważ, w świetle istniejącej teorii, ta druga ewentualność jest mało pradowpodobna, przyjąć musimy i tę pierwszą za prawie niemożliwą.

Algorytm sekwencyjny może być łatwo rozszerzony do metody, która znajduje optymalne pokolorowania. Naszkicujemy teraz jeden z takich algorytmów i zilustrujemy jego działanie na niewielkim przykładzie. W trakcie działania algorytmu, wierzchołki grafu rozpatrywane są w tym samym porządku $v_1, v_2, \ldots, v_n$, który jednak możemy wybrać dowolnie. Jako początkowe rozwiązanie przyjmujemy pokolorowanie otrzymane algorytmem sekwencyjnym. Niech q będzie liczbą użytych kolorów. (Dalej, q oznacza liczbę kolorów użytych w najlepszym pokolorowaniu.) Następne kroki algorytmu są realizacją przeglądania ciągu wierzchołków z powracaniem (and. *backtracking*), którego celem jest znalezienie możliwie najlepszego pokolorowania. Na początku, by otrzymać pokolorowanie lepsze od sekwencyjnego, musimy zmienić kolory wszystkim tym wierzchołkom, które otrzymały kolor q. Niech v_k będzie pierwszym wierzchołkiem koloru q. Ponieważ q jest najmniejszym dopuszczalnym kolorem dla v_k, cofnąć się musimy dalej i zmienić kolor wierzchołka v_{k-1}. Jeśli v_{k-1} nie może być pomalowany nowym kolorem mniejszym od q, to cofamy się dalej aż do wierzchołka v_j, któremu można przydzielić nowy kolor nie większy niż $q-1$. Następnie malujemy wierzchołki $v_{j+1}, v_{j+2}, \ldots$ pamiętając, iż aby znaleźć lepsze pokolorowanie i nie natrafić na pokolorowanie zbędne, dopuszczalny kolor wierzchołka v_i: (i) jest mniejszy od q, (ii) nie jest większy niż $\deg(v_i)+1$, oraz (iii) nie jest większy niż $l_{i-1}+1$, gdzie l_{i-1} jest największym kolorem użytym do pomalowania wierzchołków $v_1, v_2, \ldots, v_{i-1}$. Dla ilustracji, prześledźmy działanie tego algorytmu na przykładzie grafu z Rys. 5(a). Drzewo przeglądania przedstawione jest na Rys. 5(b). Poziomy drzewa odpowiadają wierzchołkom, a cyfry w węzłach drzewa są numerami kolorów przydzielanych wierzchołkom grafu. Zbiory wyszczególnione przy węzłach drzewa zawierają numery kolorów dopuszczalnych dla poszczególnych wierzchołków grafu w chwili, gdy algorytm znajduje się w danym węźle drzewa.

Dokładny opis metody dokładnej oraz wielu jej modyfikacji znaleźć można w [SDK]. Zamieszczone tam wyniki eksperymentów maszynowych przeprowadzone na maszynie Amdahl 470 są raczej pesymistyczne. Nawet daleko usprawniony algorytm dokładny nie był w stanie w rozsądnym czasie pomalować optymalnie wszystkich grafów losowych o 45 wierzchołkach. Ten sam algorytm, zastosowany do grafu F_4 z Rys. 3, znalazł rozwiązanie optymalne w ciągu 1 msek, a następnie spędził 80 msek na sprawdzenie, że jest to rzeczywiście rozwiązanie optymalne.

(a)





(b)

Rysunek 5.

5. Podsumowanie

W poprzednich trzech paragrafach staraliśmy się zilustrować jaki jest zakres stosowania zachłannego sposobu rozwiązywania trzech wybranych problemów kombinatorycznych. Przedstawiliśmy: wielomianowy algorytm zachłanny wyznaczający najkrótsze drzewa rozpinające w sieciach; klasę efektywnych algorytmów zachłannych, które są w stanie generować dowolnie bliskie optymalnym rozwiązania zadań plecakowych; i algorytm sekwencyjny kolorujący grafy w możliwie najoszczędniejszy sposób, który nie gwarantuje jednak jak dobre są rozwiązania. Zachowanie się tych algorytmów, czyli jakość generowanych rozwiązań, odpowiada w pewnym sensie teoretycznej złożoności obliczeniowej rozpatrywanych problemów,

Nieprecyzyjność i rozległość pojęcia zachłanności nie pozwala na formalizację obejmującą wszystkie przypadki, w których takie podejście jest stosowane. Ograniczymy więc naszą uwagę do klasy problemów, która zawiera problem LST (a także SST, dalej jednak będziemy rozważać jedynie problem LST, gdyż mowa będzie o problemach maksymalizacji), a jednocześnie pozwoli wyjaśnić dlaczego prosty algorytm zachłanny nie zawsze dokładnie rozwiązuje problem plecakowy. Zauważmy, że w przypadku tych problemów, każdy podzbiór rozwiązania dopuszczalnego jest także dopuszczalny. (W przypadku problemów drzew, dopuszczalny podzbiór krawędzi jest acykliczny, a w problemie plecakowym - dopuszczalny jest podzbiór towarów mieszczących się w plecaku.) Różnica polega na tym, że tylko w pierwszym przypadku maksymalne (w sensie inkluzji) zbiory dopuszczalne są równoliczne, gdyż odpowiadają im drzewa rozpinające grafu. W przypadku problemu plecakowego, drugie z rozpatrywanych przez nas zadań pokazuje, że dopuszczalne zawartości plecaka mogą składać się z różnej liczby towarów. Te dwie własności dopuszczalnych rozwiązań problemu LST definiują bardzo ważną klasę obiektów kombinatorycznych zwanych matroidami. *Matroidem* nazywamy parę $(M,\mathcal{M})$, w której M jest dowolnym zbiorem skończonym, a $\mathcal{M}$ jest rodziną jego podzbiorów spełniającą następujące dwa warunki:

(i) Jeśli $I \in \mathcal{M}$ oraz $J \subseteq I$, to także $J \in \mathcal{M}$.

(ii) Maksymalne elementy rodziny $\mathcal{M}$ są równoliczne.

Jeśli c jest wagową funkcją rzeczywistą określoną na M, to wagę problemu $I \subseteq M$ określamy $c(I) = \sum_{e \in I} c(e)$. Rozpatrzmy teraz problem wyznaczenia najcięższego podzbioru w dowolnej rodzinie podzbiorów $\mathcal{M}$, poszukujemy zatem I^* takiego, że $c(I^*) = \max\{c(I): I \in \mathcal{M}\}$. Zauważmy, że szczególnymi przypadkami tak ogólnie sformułowanego problemu są dwa pierwsze nasze problemy, dla których rodzina $\mathcal{M}$ składa się z podzbiorów dopuszczalnych, spełnia zatem warunek (i) z definicji matroidu. Następujące twierdzenie charakteryzuje jednocześnie matroidy i określa zakres stosowania algorytmu zachłannego, który w tym przypadku przedkłada elementy cięższe nad lżejszymi i dba o to, by rozwiązania częściowe należały do $\mathcal{M}$:

Niech para $(M,\mathcal{M})$ określa skończony zbiór M i pewną rodzinę $\mathcal{M}$ jego podzbiorów, o której zakładamy jedynie, że spełnia postulat (i) z definicji matroidów. Algorytm zachłanny, dla dowolnej funkcji wagowej c, wyznacza element rodziny $\mathcal{M}$ o największej wadze wtedy i tylko wtedy, gdy $\mathcal{M}$ spełnia dodatkowo postulat (ii), a więc gdy $(M,\mathcal{M})$ jest matroidem.

Twierdzenie to dostarcza nam dodatkowego uzasadnienia poprawności algorytmu zachłannego dla problemu LST oraz wyjaśnia, dlaczego prosty algorytm zachłanny nie zawsze dokładnie rozwiązuje problem plecakowy.

Czytelnika, zainteresowanego dalszymi własnościami matroidów i algorytmu zachłannego, odsyłamy do elementarnego przedstawienia tych zagadnień w książkach [Lip] i [Wil].

Na zakończenie rozpatrzmy jeszcze jeden problem, dla którego podejmowanie decyzji zachłannych jest nie tylko najbardziej naturalnym, ale także optymalnym sposobem rozwiązywania. Problem ten nie należy jednak do klasy problemów związanych z matroidami, chociaż dowód poprawności jego algorytmu rozwiązywania jest bardzo podobny do dowodu poprawności algorytmu Dijkstry.

Problem optymalnego rozmieszczania programów na taśmie magnetycznej

Dane: Danych jest n programów obliczeń o długościach $l_1, l_2, \ldots, l_n$. (Wielkość l_i może oznaczać np. liczbę instrukcji w przekładzie i-tego programu.)

Cel: Określić, w jakiej kolejności należy umieścić programy na taśmie magnetycznej tak, aby *średni czas* dostępu do programu był najkrótszy. Jeśli $I = (i_1, i_2, i_3, \ldots, i_n)$ jest kolejnością programów na taśmie, to czas dostępu do k-tego programu jest proporcjonalny do $\sum_{j=1}^{k} l_{i_j}$, zatem funkcja celu ma postać $L(I) = \frac{1}{n} \sum_{k=1}^{n} \sum_{j=1}^{k} l_{i_j}$ i należy ją zminimalizować po wszystkich możliwych kolejnościach programów.□

Wykażemy, że optymalna strategia umieszczania programów na taśmie polega na uszeregowaniu ich od najkrótszego do najdłuższego. W tym celu zauważmy najpierw, że średni czas dostępu do programów zapamiętanych w kolejności $(i_1, i_2, \ldots, i_n)$ wynosi $L(I) = \frac{1}{n} \sum_{k=1}^{n} (n-k+1) l_{i_k}$. Wykorzystując tę postać $L(I)$, łatwo można przeliczyć, że jeśli istniałyby dwa programy p i q takie, że $p < q$ i $l_{i_p} > l_{i_q}$, to dla $I' = (i_1, i_2, \ldots, i_{p-1}, i_q, i_{p+1}, \ldots, i_{q-1}, i_p, i_{q+1}, \ldots, i_n)$ mielibyśmy $L(I) > L(I')$. Zatem funkcja $L(I)$ jest minimalizowana przez uporządkowanie I spełniające $l_{i_1} \le l_{i_2} \le \ldots \le l_{i_n}$. Tłumacząc ten wniosek na język strategii postępowania mającego na celu minimalizację średniego czasu dostępu, programy na taśmie należy umieszczać w niemalejącej kolejności ich długości.

BIBLIOGRAFIA

[AHU] Aho, A.V., Hopcroft, J.E., Ullman, J.D.: *Konstrukcja i Analiza Algorytmów*, PWN, Warszawa 1984.

[BK] Banachowski, L., Kreczmar, A.: *Elementy Analizy Algorytmów*, WNT, Warszawa 1981.

[Coff] Coffman, E.G. (red.): *Teoria Szeregowania Zadań*, WNT, Warszawa 1980.

[GN] Garfinkel, R.S., Nemhauser, G.L.: *Programowanie Całkowitoliczbowe*, PWN, Warszawa 1978.

[Jer] Jerszow, A.P.: *Wprowadzenie do Teorii Programowania*, WNT, Warszawa 1981.

[Kub] Kubale, M.: Analiza efektywności algorytmów kolorowania grafów, *Matematyka Stosowana* 19 (1982), 23-41.

[KS] Kucharczyk, J., Sysło, M.M.: *Algorytmy Optymalizacji w Języku ALGOL 60*, PWN, Warszawa 1975.

[Lip] Lipski, W.: *Kombinatoryka dla Programistów*, WNT, Warszawa 1982.

[Ste] Steen, L.A. (red.): *Matematyka Dzisiaj*, WNT, Warszawa 1984.

[Sys] Sysło, M.M.: *Optymalizacja Kombinatoryczna*, Instytut Informatyki, Uniwersytet Wrocławski, Wrocław 1981. Skrócona wersja ukaże się jako Rozdział III w książce T. Kasprzak (red.), *Ekonomiczne Zastosowania Optymalizacji Dyskretnej*, PWE, Warszawa 1984.

[SDK] Sysło, M.M., Deo, N., Kowalik, J.S.: *Discrete Optimization Algorithms with Pascal Programs*, Prentice-Hall, Englewood Cliffs 1983.

[Wil] Wilson, R.J.: *Wprowadzenie do Teorii Grafów*, PWN, Warszawa 1985.

Jesienna Szkoła PTI
Rydzyna, październik 1984

WYBRANE JĘZYKI I METODY PROGRAMOWANIA W ZASTOSOWANIACH

doc. dr hab. Stanisław Waligórski
Instytut Informatyki
Uniwersytetu Warszawskiego
00-901 Warszawa, PKiN p.850
tel. 268258.

Gdyby ktoś chciał zbadać, jakie języki programowania są w Polsce najczęściej używane w zastosowaniach informatyki, poza środowiskiem akademickim, to najprawdopodobniej pierwsze miejsca zajęłyby Fortran, Cobol, PL/I. Poza tym baruzo wielu programistów zastosowań programuje w różnych asemblerach. W związku z pojawianiem się mikrokomputerów renesans popularności zaczyna przeżywać Basic. Język Pascal, który jest najbardziej rozpowszechnionym w świecie językiem programowania strukturalnego, dopiero z trudem toruje sobie drogę.

Przyczyny tego stanu rzeczy są proste:

1. Dostępność języków w oprogramowaniu firmowym różnych zainstalowanych w kraju komputerów: są to przede wszystkim języki z podanej wyżej listy: Fortran, Cobol, PL/I, Basic, Asembler.

2. Powszechność oprogramowania: większa część oprogramowania zastosowań, używanego w kraju, jest właśnie w tych językach.

3. Przygotowanie kadr: istniejąca już od ponad 20 lat tradycja przygotowywania kadr dla zastosowań nakazuje zaczynać naukę programowania od któregoś z tych właśnie języków - często zresztą

na nim się kończy.

Zauważmy, że przyczyny te, każda z osobna, są poważne, ale trwałość ich wynika głównie z tego, że skutki ich działania uzupełniają się i wzmacniają nawzajem. Powodem 1 jest 3, brak programistów z odpowiednim przygotowaniem, a ponadto dobrym uzasadnieniem 1 jest 2. Powodem 2 jest 1 i 3. Uzasadnieniem 3 jest 1 i 2.

Nie jesteśmy pod tym względem w Polsce wyjątkiem. Podobna sytuacja jest także w wielu innych krajach, przynajmniej tych na zbliżonym szczeblu rozwoju i rozpowszechnienia metod informatyki. Można powiedzieć, że jest typowa.

Z punktu widzenia tego wykładu interesują mnie główne cechy tych języków: słabe przystosowanie do wymagań, narzucanych przez współczesne metody programowania i zbyteczne absorbowanie programującego w tych językach drobiazgowymi szczegółami technicznymi i różnymi zbędnymi ograniczeniami, a w związku z tym i wskutek tego wyrabianie w programistach złych nawyków programowania; w szczególności przyzwyczajanie do programowania niestrukturalnego. Ostatecznie programowanie strukturalne, którego odkrycie narobiło kiedyś tyle szumu, było nowością przede wszystkim dla programistów Fortranu i Cobolu - stanowili oni zresztą, co trzeba podkreślić, przygniatającą większość programistów zastosowań w tamtych czasach. Ci, którzy znali i stosowali Algol lub Lisp, mieli drogę do nowoczesnego programowania znacznie krótszą i wiele niezbędnych metod i nawyków mieli opanowane od samego początku. Podobnie jest z innymi uznanymi obecnie metodami i koncepcjami współczesnej informatyki.

Mówiąc to wszystko, nie chcę oczywiście sugerować, że współcześni programiści zastosowań trzymają się wyłącznie tradycyjnych metod i nowe idee nie mają do nich dostępu, gdyż mijałoby się to z prawdą. Ale temat tego wykładu dobrałem na zasadzie kontrastu z najbardziej rozpowszechnioną odmianą konserwatywnego podejścia do programowania zastosowań.

Ponieważ mikrokomputery są modne, zacznijmy nasze rozważania

od nich. Każdy, kto kupuje mikrokomputer, nabywa razem z nim translator Basica. Będzie więc zapewne programować w tym języku on sam, a jeśli jest to komputer domowy, to zapewne także jego dzieci. Nauczą się przy tym złego stylu programowania i nabiorą złych nawyków programistycznych, ale o to mniejsza, bo może jeszcze będą miały szanse nauczyć się czegoś lepszego. Dla dorosłych Basic też nie jest dobry, bo zamiast ułatwiać stosowanie właściwych metod rozwiązywania problemów, absorbuje ich uwagę nieistotnymi szczegółami, które z tymi metodami związku nie mają, jak numeracja wierszy, struktura goto i gosub, albo też ciągle takie same komponowanie ze zbyt elementarnych operacji języka ciągle takich samych fragmentów programów, zapewniających wykonywanie standardowych czynności wyboru, powtarzania, badania złożonych warunków logicznych itp. W rezultacie zostaje znacznie mniej czasu i możliwości na dobre obmyślenie metod rozwiązywania problemów, doboru struktur programów, kontroli ich poprawności, sprawnego uruchamiania i testowania.

Na pytanie, jaka jest lepsza możliwość, można by zapewne usłyszeć wiele odpowiedzi, a wśród nich: Smalltalk, Prolog lub Logo. Przyjrzyjmy się na chwilę ostatniemu z tych języków.

Został on stworzony przede wszystkim dla dzieci i tak opracowany, by ich uwaga mogła się cały czas koncentrować właśnie na rozwiązywaniu problemów, na tym, co ma zrobić komputer, a nie na technicznych problemach programowania. Aby wynik czynności wykonywanych przy komputerze przez dziecko był widoczny możliwie natychmiast, a w każdym razie jak najszybciej. Aby wykonywanie raz zaprogramowanych złożonych akcji komputera było równie łatwe, jak wykonanie akcji najbardziej elementarnych. Żeby można było z już opracowanych procedur tworzyć inne w każdym momencie, kiedy się zechce to zrobić, łatwo i nie będąc absorbowanym technicznymi szczegółami tego procesu. Żeby tak stworzone nowe procedury nadawały się do użycia praktycznie natychmiast, niezależnie od tego, jak bardzo są skomplikowane.

Czy to w ogóle można osiągnąć i jak?

Logo kojarzy się wielu z tak zwaną "żółwią grafiką" (turtle

graphics). Być może dzieje się tak dlatego, że zwykle każdy elementarny wykład Logo właśnie od niej się zaczyna i rzeczywiście przy jej pomocy można łatwo pokazać różne charakterystyczne cechy i możliwości języka, mimo że nie ograniczają się one tylko do grafiki. Dlatego ja również od niej zacznę. Jest to opracowana specjalnie dla potrzeb uczenia dzieci metoda tworzenia rysunków możliwie najprostszymi środkami. Stworzona dla dzieci, okazała się jednak dobra także dla dorosłych i stąd jej stale rosnąca popularność. Istotą jej jest użycie wskaźnika, zwanego tu żółwiem (metoda była stworzona dla dzieci!), którym można sterować z klawiatury przy pomocy prostych komend i w ten sposób rysować obrazy na ekranie.

Żółw ma w każdym momencie określone dwie współrzędne położenia na ekranie oraz kąt, określający kierunek, w jakim jest ustawiony i może się poruszać. Zwykle przedstawia się go na ekranie w formie małego trójkąta, tak że zarówno jego położenie, jak kierunek, są dobrze widoczne. Odpowiednikiem żółwia może być pisak plotera, przyłączonego do tego samego mikrokomputera, albo sprzężony z mikrokomputerem żółw-"robot"; małe urządzenie na kółkach, przypominające nieco kształtem żółwia, wyposażone w pisak, który może się podnosić lub opuszczać. Można go postawić na dużym arkuszu papieru i wtedy będzie powtarzać ruchy i czynności żółwia-wskaźnika z ekranu, przenosząc na papier w powiększonej skali rysowany obraz.

Początek układu współrzędnych jest w środku ekranu. Współrzędne poziome x rosną od lewej strony do prawej, współrzędne pionowe y z dołu do góry. Kąt określający kierunek, w jakim żółw jest ustawiony, jest odmierzany w stopniach, licząc od kierunku pionowo w górę, w prawo, to znaczy zgodnie z kierunkiem obrotu wskazówek zegara. Tak więc żółw skierowany pionowo w górę ma kąt 0^{o}, skierowany poziomo w prawo 90^{o} itd.

Przy pomocy komend języka Logo można powodować przesuwanie żółwia w przód lub w tył o zadaną odległość, albo obracać go w miejscu w prawo lub w lewo o zadaną liczbę stopni. Można przesuwać żółwia do dowolnego miejsca ekranu o zadanych współrzędnych, nie zmieniając jego kierunku, albo skierować go pod dowolnym zadanym kątem.

Żółw trzyma pióro, które może podnieść i wtedy przesuwanie się żółwia nie pozostawia śladu, albo opuścić je i wtedy w czasie ruchu żółwia pióro wlecze się po ekranie (lub papierze), pozostawiając ślad. Jeśli jest to ekran kolorowy, można określić kolor rysowanej kreski, do czego służy specjalna komenda. Zmieniając kolor pióra w czasie rysowania można dostać obraz o dowolnie dobranych barwach. Szczególne zastosowanie ma rysowanie kresek o kolorze identycznym z kolorem tła: w ten sposób można zacierać kreski już narysowane, gdyż nowa kreska, rysowana na starej, pokryje ją i w ten sposób usunie z ekranu. Jeśli na ekranie powstanie zamknięty kontur, to można wprowadzić żółwia do środka i zażądać wypełnienia tego obszaru, na którym żółw stoi, dowolnie dobranym kolorem.

Obraz żółwia można w razie potrzeby ukryć, usuwając z ekranu, albo uczynić go znowu widocznym. Operacje te nie wpływają na zmianę położenia ani kierunku żółwia. Skutki komend powodujących przesuwanie lub obracanie żółwia są w obu przypadkach takie same, niezależnie od tego, czy wizerunek żółwia jest widoczny na ekranie, czy ukryty.

Ta metoda tworzenia elementarnych rysunków okazała się tak dobra, że obecnie wiele języków, stosowanych na mikrokomputerach, ma już wbudowaną żółwią grafikę jako element języka.

Metody grafiki komputerowej są tematem dla odrębnego wykładu; tutaj będziemy się zajmować nią tylko w takim stopniu, w jakim to będzie potrzebne do zilustrowania interesujących nas charakterystycznych cech języka Logo. Chodzi tu w szczególności o pokazanie przy pomocy nieco uproszczonego zbioru komend Logo idei posługiwania się nim i sposobów korzystania z jego możliwości. Przy okazji zobaczymy praktyczne stosowanie zasad posługiwania się żółwiem przy tworzeniu prostych rysunków.

Przykłady są skrajnie elementarne, ale oczywiście możliwości języka nie są ograniczone tylko do takich błahostek.

Zacznijmy od komend umożliwiających rysowanie - oto najważniejsze z nich, które się tu mogą przydać:

DRAW - oczyść ekran, ustaw żółwia w środku ekranu, skieruj go pionowo do góry i uczyń widocznym.

HOME - przesuń żółwia do położenia w środku ekranu.

CLEARSCREEN - usuń rysunek z ekranu, nie zmieniając położenia ani kierunku żółwia.

HIDETURTLE - uczyń żółwia niewidocznym, nie zmieniając jego położenia ani kierunku.

SHOWTURTLE - pokaż żółwia, czyniąc go odtąd widocznym na ekranie.

PENDOWN - opuść pióro, umożliwiając rysowanie.

PENUP - podnieś pióro, umożliwiając ruch żółwia bez rysowania.

PENCOLOR - ustal podany kolor jako kolor kreski, rysowanej w położeniu pendown.

FORWARD - przesun żółwia w przód o podaną odległość.

BACK - przesuń żółwia w tył o podaną odległość, nie zmieniając jego kierunku.

RIGHT - obróć żółwia w miejscu w prawo o podaną ilość stopni.

LEFT - obróć żółwia w miejscu w lewo o podaną ilość stopni.

SETHEADING - obróć żółwia w miejscu tak, aby ustawił się pod podanym kątem.

SETXY - ustaw żółwia w punkcie o podanych współrzędnych, nie zmieniając jego kierunku.

SETX - zmień współrzędną x żółwia na podaną, przesuwając go poziomo do tego miejsca i nie zmieniając jego współrzędnej y ani kierunku.

SETY - zmień współrzędną y żółwia na podaną, przesuwając go pionowo do tego miejsca i nie zmieniając jego współrzędnej x ani kierunku.

XCOR - podaj współrzędną poziomą x żółwia.

YCOR - podaj pionową współrzędną y żółwia.

HEADING - podaj kąt określający kierunek żółwia.

TOWARDS - podaj kąt, pod jakim byłby ustawiony żółw, gdyby był skierowany na punkt o podanych współrzędnych.

Załóżmy, że żółw jest w środku ekranu, skierowany pionowo w górę, widoczny, pióro jest opuszczone. Jeśli jest to ekran kolorowy, to kolor rysowanej kreski i kolor tła są różne i oba są określone.

Pisząc

```
RIGHT 90
```

powodujemy obrócenie żółwia w prawo bez zmiany jego położenia. Komenda

```
FORWARD 100
```

spowoduje narysowanie poziomego odcinka o podanej długości. Pisząc dalej

```
LEFT 90
FORWARD 50
LEFT 90
FORWARD 100
LEFT 90
FORWARD 50
LEFT 90
```

poprowadzimy żółwia tak, aby narysował prostokąt i wrócił do położenia wyjściowego. Ostatni obrót w lewo ustawił go tak, że znów jest skierowany poziomo w prawo.

Teraz możemy podnieść pióro i przesunąć żółwia nieco w przód,

```
PENUP
FORWARD 70
PENDOWN
```

po czym można znów narysować prostokąt o nieco innych wymiarach. Zauważmy jednak, że w sekwencji, rysującej poprzedni prostokąt, ostatnie cztery komendy były powtórzeniem poprzednich. Aby zapisać to krócej, użyjemy komendy REPEAT, powodującej powtórzenie podaną ilość razy sekwencji komend, ujętej w nawiasy kwadratowe.

```
REPEAT 2 [FORWARD 15 LEFT 90 FORWARD 35 LEFT 90]
```

Po wykonaniu tej komendy na ekranie jest nowy prostokąt, a żółw jest w jego dolnym lewym rogu, skierowany poziomo w prawo.

Jeśliby takie rysowanie prostokątów miało być powtarzane wielokrotnie, to można stworzyć odpowiednią procedurę, dodając na początku wykonującej to sekwencji komend zwrot TO z nazwą, jaką chcemy nadać tej procedurze, a na końcu pisząc END. Jeśli ponadto ten prostokąt ma mieć wymiary dobierane stosownie do potrzeb, to można wprowadzić zamiast konkretnych liczb odpowiednie parametry. Dla odróżnienia parametru od nazwy procedury daje się przed nim dwukropek.

```
TO PROSTOKAT :SZEROKOSC :WYSOKOSC
REPEAT 2 [FORWARD :SZEROKOSC LEFT 90 FORWARD :WYSOKOSC LEFT 90]
END
```

Po wprowadzeniu END nazwa procedury i jej znaczenie zostają zapamiętane i odtąd można się tą nazwą posługiwać tak samo, jak komendami języka. Można więc użyć jej jako komendy z odpowiednimi parametrami w celu wykonania określonej przez nią akcji, albo wykorzystać w definicji nowej procedury.

Aby ułatwić sobie lokowanie prostokątów w różnych miejscach ekranu, stwórzmy jeszcze dodatkową procedurę

```
TO PRZESUN :POZIOMO :PIONOWO
PENUP
SETXY XCOR + :POZIOMO  YCOR + :PIONOWO
PENDOWN
END
```

Teraz można przesunąć żółwia i narysować nowy prostokąt:

```
PRZESUN -70 50
PROSTOKAT 100 45
```

A oto przykład użycia nazwy procedury w definicji nowej procedury.

```
TO OKNO
REPEAT 3 [PROSTOKAT 10 20 FORWARD 10]
END
```

Umieszczamy to okno w właściwym miejscu i doprowadzamy żółwia do prawego dolnego rogu budynku:

```
PRZESUN 15 -35
OKNO
PRZESUN 55 -15
```

Płot możemy potraktować jako ciąg określonej liczby sztachet.

```
TO SZTACHETA
FORWARD 2
PROSTOKAT 3 15
FORWARD 3
END
```

Rysowanie płotu wykorzystamy jako okazję do wprowadzenia przykładu procedury rekurencyjnej.

```
TO PLOT :ILESZTACHET
```

```
IF :ILESZTACHET = 0 THEN STOP
SZTACHETA
PLOT :ILESZTACHET - 1
END
```

Komenda STOP powoduje zakonczenie wykonywania procedury. Jeśli więc liczba sztachet jest dodatnia, to procedura rysuje jedną sztachetę i obok niej płot krótszy o tę sztachetę. Gdy ilość sztachet dojdzie do zera, działanie procedury się kończy.

W czasie wpisywania tych definicji żółw pozostawał na miejscu. Teraz możemy dorysować koło domu kawałek płotu:

```
PLOT 20
```

I wreszcie można zbudować blok o zadanej liczbie pięter oraz okien na każdym piętrze. Jeśli ilość pięter jest równa 0, to będzie to budynek parterowy, ale liczba okien musi być zawsze dodatnia. Wrysowywanie okien w fasadzie można by też zaprogramować rekurencyjnie, ale tym razem użyjemy REPEAT. Za to procedura będzie okazją do użycia komendy MAKE, nadającej wartości zmiennym oznaczającym wysokość i szerokość rysowanego budynku, oraz komendy warunkowej o konstrukcji IF ... THEN ... ELSE ... , która w tym przypadku uzależnia kształt dachu od wysokości domu. Komenda MAKE nadaje zmiennej, która jest jej pierwszym parametrem, wartość równą wartości jej drugiego parametru. Dotychczas mieliśmy do czynienia z parametrami poprzedzonymi dwukropkiem, teraz spotykamy drugi sposób oznaczania parametrów. Dwukropek oznacza, że w czasie wykonywania procedury należy brać wartość parametru. Cudzysłów poprzedzający nazwę parametru oznacza, że w tym przypadku chodzi o nazwę parametru, a nie wartość. Rzeczywiście, jeżeli zmiennej nadaje się nową wartość, to stara wartość jej jest niepotrzebna, trzeba natomiast wiedzieć, jak się ta zmienna nazywa. Drugi parametr MAKE określa wartość nadawaną i tam dwukropek może wystąpić.

```
TO DOM :ILEPIETER :ILEOKIEN
MAKE "WYSOKOSC (:ILEPIETER + 1) * 45 + 25
MAKE "SZEROKOSC :ILEOKIEN * 50 + 20
PROSTOKAT :SZEROKOSC :WYSOKOSC
REPEAT (:ILEPIETER + 1) [PIETRO :ILEOKIEN :SZEROKOSC]
```

```
IF :WYSOKOSC<150 THEN DACH :SZEROKOSC ELSE PRZESUN :SZEROKOSC 0
PRZESUN 0 -:WYSOKOSC
END
TO PIETRO :ILEOKIEN :SZEROKOSC
PRZESUN 20 25
OKNO
REPEAT (:ILEOKIEN - 1) [PRZESUN 20 0 OKNO]
PRZESUN (40 - :SZEROKOSC) 45
END
TO DACH :SZEROKOSC
LEFT 60
FORWARD 70
RIGHT 60
FORWARD :SZEROKOSC - 70
RIGHT 60
FORWARD 70
LEFT 60
END
```

Możemy jeszcze wprowadzić procedurę rysującą drzewo. Jej treść pozostawiamy inwencji Czytelnika. Trzeba pamiętać, aby po zakończeniu procedury żółw był tak ustawiony, aby mógł zaraz zacząć rysować następny element obrazu, bez straty czasu na specjalne ustawianie go w właściwej pozycji.

```
TO DRZEWO
...
END
```

W trakcie wprowadzania tych wszystkich definicji rysunek się nie zmienił, a żółw pozostawał pod płotem, ostatnio narysowanym elementem obrazu, po jego prawej stronie, skierowany poziomo w prawo. Jego współrzędne: 200, 0 . Jest on gotowy do rozpoczęcia rysowania następnego elementu rysunku. Może to być na przykład dom, drzewo lub płot, ale trzeba uważać, żeby nie przekroczyć krawędzi ekranu.

Reakcję na dojście żółwia do brzegu ekranu określają trzy komendy wyboru trybu pracy:

WINDOW - żółw może wyjść poza brzeg ekranu i poruszać się da-

lej dowolnie poza nim, lecz jego ślad staje się niewidoczny dopóty, dopóki nie powróci znów w obręb ekranu i będzie w stanie pendown.

FENCE - każda próba wyjścia poza brzeg ekranu jest nieskuteczna i sygnalizowana jako błąd odpowiednim komunikatem, pojawiającym się na ekranie.

WRAP - przejście żółwia poza brzeg ekranu powoduje pojawienie się go po przeciwnej stronie, tak jakby przeciwległe krawędzie ekranu - lewa i prawa oraz górna i dolna - były sklejone ze sobą.

Rozmiary ekranów, mierzone w jednostkach, używanych w komendzie FORWARD, mogą być różne, zależnie od implementacji. Mogą być ustalone lub dobierane przy pomocy odpowiedniej komendy. W naszych rozważaniach możemy założyć, że prawy i lewy brzeg ekranu mają odpowiednio współrzędne x równe 250 i -250; szerokość ekranu jest więc równa 500 jednostek. Wybierzmy tryb pracy, w którym żółw przechodzi cyklicznie z prawego brzegu ekranu na lewy i rysujmy dalej.

```
WRAP
DRZEWO
PLOT 12
DOM 1 2
PRZESUN -120 0
PLOT 24
DRZEWO
SETX 0
```

Ten nieskomplikowany obrazek z domami, drzewami, płotami i ulicą jest prostym przykładem tego, co można uzyskać przy pomocy środków oferowanych przez Logo, pokazanych zresztą tylko częściowo i w dużym uproszczeniu - stosunkowo niewielki rozmiar tego wykładu nie pozwala zagłębiać się w szczegółach. Szczególnie zwraca uwagę użycie żółwiej grafiki i możliwość swobodnego definiowania procedur w dowolnym momencie rozmowy z komputerem. Przykład ten pokazuje także inne środki, jak możliwość dowolnego składania procedur i swobodnego użycia rekurencji. Wszystko to jest wprowadzone w sposób prosty i zrozumiały dla użytkownika, można powiedzieć, naturalny, ale na to, żeby to wszystko mogło w ten sposób działać, trzeba było użyć dość skomplikowanego aparatu, który jest raczej ukryty przed okiem przeciętnego programisty, ale tak zbudowany, by uwolnić go od troski o techniczne

szczegóły tego procesu. W skład tego aparatu wchodzą między innymi: gospodarka pamięcią dynamiczną bez stałych rezerwacji miejsc, tworzenie, przechowywanie i likwidowanie struktur danych bez potrzeby angażowania w to programisty, środki analizy i interpretacji programów, obsługa wykonania procedur, obliczania i przekazywania wartości parametrów dla złożeń procedur i wywołań rekurencyjnych, kontrola poprawności obliczeń w warunkach pracy konwersacyjnej itp. Część tych możliwości, jak na przykład obróbka tekstów lub działania na strukturach listowych, jest dostępna dla programisty Logo dzięki istnieniu odpowiednich komend języka.

W literaturze, poświęconej Logo, główny akcent kładzie się na jego walory dydaktyczne i wobec tego mniej się pisze o rzeczywistej mocy tego języka. Logo powstał na bazie Lispu, języka stworzonego na początku lat 60-tych, który jest nadal najbardziej znanym i najczęściej stosowanym językiem w takich działach informatyki, jak robotyka, analiza i synteza obrazów, komputerowo wspomagane systemy dowodzenia twierdzeń, badania języków naturalnych i naturalnej komunikacji człowieka z maszyną. Logo przejął wiele dobrych cech i wypróbowanych możliwości Lispu, ale ma bardziej rozbudowane możliwości operowania na tekstach i znacznie dogodniejszą postać zewnętrzną. Niektórzy autorzy traktują Logo jako dialekt Lispu, ze względu na zbieżność możliwości obu języków w zaawansowanych zastosowaniach, ale specyficzne cechy Logo i podejście jego twórców do ogólnych zagadnień programowania są na pewno godne uwagi.

Rozważmy teraz programowanie zadań zupełnie innego typu. Na prostokątnym placu składowiska kontenerów każde miejsce ma swoje współrzędne. Ponieważ kontenery mogą być składowane jeden na drugim do pewnej wysokości, położenie każdego kontenera jest określone przez trzy współrzędne x, y, z. Każdy kontener ma numer, który go jednoznacznie identyfikuje. Po placu jeździ dźwig - dla uproszczenia złóżmy, że tylko jeden - który wykonuje wszystkie operacje przemieszczania kontenerów. Chodzi o całkowite zaprogramowanie jego pracy, tak, aby wszelkie czynności, związane z rozmieszczaniem kontenerów na placu, wyszukiwaniem ich i wysyłką były sterowane komputerowo.

Ewidencja wszystkich kontenerów i ich położenia jest prowadzona na bieżąco. Głównym jej elementem jest rejestr wszystkich kontenerów, znajdujących się na placu. Można go przedstawić jako tablicę, w której każdy wiersz jest przeznaczony na dane o jednym kontenerze. Zawiera ona w kolejnych kolumnach numer kontenera, jego trzy współrzędne oraz inne informacje, które są niezbędne zarządowi placu i w szczególności mogą mieć wpływ na sposób składowania. Do takich informacji mogą na przykład należeć: miejsce przeznaczenia i przewidywany termin wysyłki, rodzaj i zawartość kontenera, specjalne wymagania w związku z przechowywaniem lub przewozem kontenera. Oprócz tego dla przyspieszenia przetwarzania danych można dla każdego kontenera umieścić tam informacje tego typu, jak numer kontenera znajdującego się bezpośrednio na nim, jeśli taki istnieje. Ewidencja jest stale aktualizowana: możemy przyjąć, że zmiany opisu sytuacji są zawsze wprowadzane równocześnie z wykonaniem operacji, która jest powodem tych zmian.

Wydobycie partii kontenerów ze składowiska jest proste, jeśli na żadnym z nich nie stoi kontener, który do niej nie należy. W przeciwnym przypadku najpierw trzeba te blokujące kontenery zdjąć i odstawić na inne miejsca, robiąc to tak, aby nie utrudnić wydobycia tej partii, a w miarę możności także następnych. Błędne rozwiązywanie tego zagadnienia mogłoby w końcu doprowadzić do tego, że przy dużym zapełnieniu placu kontenerami praca na składowisku zaczęłaby przypominać pasjans, tyle byłoby przestawień do wykonania, zanim by się dotarło do kontenerów właściwych i mogło je ze składowiska zabrać.

Jest to zadanie innego typu, niż omówiony poprzednio przykład z rysowaniem obrazów i większość zadań, z jakimi mają do czynienia i rozwiązują programiści zastosowań. Cechą charakterystyczną tamtych było to, że algorytm postępowania był określony ze wszystkimi szczegółami od początku do końca. Przy tworzeniu rysunków z pomocą Logo każdy krok żółwia był jawnie zaprogramowany. Ułatwienie polegało głównie na tym, że po połączeniu prostszych akcji w bardziej skomplikowane i ogólne programista mógł mieć tylko z nimi do czynienia i nie musiał więcej zajmować się strukturą algorytmów na najbardziej prymitywnym poziomie. Tym nie mniej algorytm tak, czy

inaczej musiał być przygotowany ze wszystkimi szczegółami.

W zadaniu z kontenerami trudno byłoby określić z góry algorytmy dla wszystkich możliwych sytuacji, bo jest ich zbyt wiele i są zbyt różne, aby wszystko dało się ująć przy pomocy rozsądnie małej liczby reguł postępowania, a przygotowywanie programów na wszystkie ewentualności byłoby nieopłacalne i mało sensowne. Trzeba do tego podejść inaczej.

W życiu codziennym polecenie wykonania jakiejś czynności nie zawiera na ogół pełnego opisu tego, co należy zrobić, ale raczej określa rezultat, jaki ma być osiągnięty. Wykonawca, do którego polecenie jest kierowane, ma na ogół dostateczne informacje o sytuacji, w jakiej będzie działać, oraz o czynnościach, jakie może lub powinien przedsięwziąć, aby móc zapewnić wykonanie tego polecenia. Gdyby jakichś wiadomości mu brakło, może postarać się uzupełnić je przed przstąpieniem do pracy. Niektóre informacje może otrzymać nawet w trakcie wykonywania zadania.

Nie ma podstaw sądzić, że nie można by w podobny sposób podejść do formułowania zadań dla komputerów. Ich wadą jest to, że można znacznie mniej pozostawić domyślności lub doświadczeniu, niż w przypadku innych wykonawców. Również swoboda wyboru sformułowań poleceń będzie bardziej ograniczona. Dla zapewnienia prawidłowej interpretacji poleceń oraz prawidłowego planowania i kontroli ich wykonania musi istnieć sformalizowany aparat opisywania sytuacji, badania warunków, niezbędnych dla oceny sytuacji, oraz opisywania ciągów operacji, jakie mają być wykonywane. W szczególności można założyć, że każda czynność, prowadząca do jakiejkolwiek zmiany sytuacji, jest wynikiem wykonania pewnej sekwencji operacji elementarnych. Dla uproszczenia rozważań nie zakładamy, że operacje elementarne mogą być wykonywane równolegle. Możemy również założyć, że cele do osiągnięcia są zawsze formułowane w postaci koniunkcji warunków. Na przykład polecenie "zabierz ze składowiska wszystkie kontenery z partii nr 1" określa cel, który można opisać w postaci warunku: "na składowisku nie ma kontenerów z partii nr 1".

Sposób wykonania polecenia zależy od sytuacji, w jakiej zaczy-

na się działać. Wygodnie więc będzie założyć, że przez zadanie rozumie się komplet dwu informacji: o sytuacji początkowej i o warunkach do spełnienia. Planem wykonania zadania nazwiemy opis ciągu operacji elementarnych, które z sytuacji początkowej doprowadzają do sytuacji, w której jest spełniona koniunkcja warunków zadania. Nietrudno zauważyć, że takich sytuacji może być wiele i również wiele może być planów, a wobec tego trzeba będzie stosować pewne zasady selekcji planów. Optymalnym będzie plan, który będzie możliwie najkrótszy, chociaż można stosować również inne kryteria doboru. W każdym razie chodzi o to, aby takie plany można było przygotowywać przy pomocy komputera, mając tylko wiadomości o możliwych sytuacjach, o repertuarze operacji elementarnych i o zadaniach. W ten sposób ciężar drobiazgowego doboru algorytmów zostanie przerzucony na oprogramowanie - trzeba tylko wiedzieć, jak je wykonać. Spróbujmy przynajmniej przyjrzeć się, od czego należałoby zacząć.

Załóżmy, że interesują nas kontenery z trzech partii, oznaczonych numerami 1, 2 i 3. Wszystkie te kontenery są umieszczone na czterech miejscach o tej samej współrzędnej y, tak że do scharakteryzowania ich położenia wystarczy podać współrzędną x, przyjmującą dla tych miejsc wartości od 1 do 4. Przyjmijmy, że kontenery tego typu można ustawiać na sobie w co najwyżej trzech warstwach. Wszystkie inne miejsca w bliskim sąsiedztwie są całkowicie zastawione trzema warstwami kontenerów i nie można ich wykorzystać do przestawiania kontenerów, które nas aktualnie interesują.

Z rejestru wszystkich kontenerów możemy wydobyć informacje o tych właśnie, ograniczając je tylko do danych, które będą potrzebne w dalszych rozważaniach. Jeśli ten rejestr jest przechowywany w bazie danych ze standardowym oprogramowaniem i standardowymi metodami dostępu, to taka selekcja danych może być opisana przy pomocy jednego zdania w używanym w tej bazie języku. Gdyby to na przykład był Sequel, wyglądałoby to tak:

```
SELECT KONTENER,NRPARTII,MIEJSCE,WARSTWA,NAD,POD
FROM KONTENERY
WHERE NRPARTII IN (1,2,3)
ORDER BY NRPARTII,KONTENER
```

Otrzymamy następujące dane:

KONTENER	NRPARTII	MIEJSCE	WARSTWA	NAD	POD
A	1	1	3	B	-
C	1	1	1	-	B
E	1	2	2	F	D
J	1	3	1	-	H
K	1	4	3	L	-
B	2	1	2	C	A
F	2	2	1	-	E
H	2	3	2	J	G
L	2	4	2	M	K
D	3	2	3	E	-
G	3	3	3	H	-
M	3	4	1	-	L

Dla tego fragmentu składu możemy wydobywać wszystkie interesujące nas informacje szczegółowe. Na przykład dane o kontenerach partii 1, leżących na wierzchu:

```
SELECT KONTENER,MIEJSCE FROM KONTENERY
WHERE NRPARTII=1 AND POD='-'
```

Otrzymamy:

KONTENER	MIEJSCE
A	1
K	4

Kontenery partii 1 leżące pod innymi:

```
SELECT KONTENER,MIEJSCE,WARSTWA,POD FROM KONTENERY
WHERE NRPARTII=1 AND NOT POD='-'
```

KONTENER	MIEJSCE	WARSTWA	POD
C	1	1	B
E	2	2	D
J	3	1	H

Kontenery, na których można ustawić inne:

```
SELECT KONTENER,MIEJSCE,WARSTWA FROM KONTENERY
WHERE POD='-' AND WARSTWA<3 AND NRPARTII IN (1,2,3)
```

Wynikiem jest tablica pusta, bo takich kontenerów nie ma.

Dla ułatwienia rozważań wprowadzimy oznaczenia dla niektórych warunków.

Fakt, że na kontenerze k można ustawić inny, czyli że dla k jest spełniony warunek klauzuli WHERE ostatniego zdania SELECT, będziemy oznaczać wolny(k). Podobnie dla miejsca m napiszemy wolny(m). W ten sposób jednolicie potraktujemy oba rodzaje obiektów, co nie powinno prowadzić do nieporozumień, bo w każdym konkretnym przypadku będzie można odróżnić po oznaczeniach, czy chodzi o kontener, czy miejsce, Fakt, że kontener k jest na wierzchu, to znaczy nie stoi na nim żaden inny i dźwig może go wziąć, zapiszemy nawierzchu(k). Ten warunek nie może być spełniony dla miejsc. To, że kontener k stoi na kontenerze lub miejscu l, zapiszemy na(k,l).

Podstawiając zamiast zmiennych konkretne oznaczenia kontenerów lub miejsc możemy utworzyć koniunkcję warunków, opisującą całkowicie stan naszego zespołu kontenerów.

Mamy do dyspozycji tylko dwie operacje elementarne: zabierz kontener k, stojący na l, poza składowisko oraz przenieś kontener k z l na m. Czytelnik zapewne zauważy, że jeśli mówimy o k, to nie trzeba określać w definicji operacji l, bo przecież zawsze wiadomo, na czym k stoi. Jest to słuszne, ale w tym przypadku chodzi o wyraźne wymienienie wszystkich obiektów, na które dana operacja ma wpływ; warunki dotyczące innych obiektów, nie wymienionych wśród parametrów operacji, pozostają niezmienione.

Każda z tych operacji elementarnych może być wykonana tylko wtedy, gdy są spełnione pewne warunki wstępne. Każda powoduje usunięcie lub wpisanie pewnych danych z zapisów, dotyczących jej parametrów. Oto pełny opis tych operacji elementarnych:

zabierz(k,l) - zabierz kontener k, stojący na obiekcie l, ze składowiska.
warunki wstępne: nawierzchu(k), na(k,l)
fakty usuwane: nawierzchu(k), na(k,l), oraz wolny(k), o ile ten warunek był spełniony.
fakty dodawane: wolny(l), nawierzchu(l).

przenieś(k,l,m) - przenieś kontener, stojący na obiekcie l, na obiekt m.
warunki wstępne: nawierzchu(k), na(k,l), wolny(m)
fakty usuwane: na(k,l), wolny(m), nawierzchu(m)

fakty dodawane: wolny(l), nawierzchu(l), na(k,m), oraz, jeśli m jest miejscem lub kontenerem stojącym na miejscu, tzn w warstwie 1, to także wolny(k). Gdyby natomiast ten ostatni warunek nie był spełniony, a przedtem było wolny(k), to należałoby ten fakt usunąć.

W opisanym wyżej stanie początkowym można wykonać operacje zabierz(A,B) i zabierz(K,L), bo są spełnione ich warunki wstępne. Po wykonaniu ich możemy otrzymać następujące informacje o kontenerach na wierzchu:

KONTENER	NRPARTII	MIEJSCE	NAD
B	2	1	C
D	3	2	E
G	3	3	H
L	2	4	M

i o wolnych kontenerach (wolnych miejsc nie ma):

KONTENER	MIEJSCE	WARSTWA
B	1	2
L	4	2

Zestawiając ze sobą te dwie tablice widzimy, że istnieje 6=4*2-2 możliwości wykonania operacji przenieś. Ale patrząc na dane o pozostałych kontenerach partii 1 widzimy, że jeden z nich jest pod B, a więc nie byłoby rozsądne stawianie czegokolwiek na kontenerze B. Można natomiast przenieść coś na kontener L. Daje to 3 różne możliwości rozpoczęcia planu; dwie z nich prowadzą do planów o najmniejszej możliwej długości. Ale nawet te dwie wersje nie są równoważne, jeśli weźmiemy pod uwagę dalsze skutki. Tylko jeden ruch, przeniesienie B na L, umożliwia osiągnięcie żądanego celu bez blokowania partii 2 kontenerami partii 3.

Tak szkicowo przedstawiony sposób postępowania polega na badaniu możliwości wykonywania kolejnych kroków planu, poczynając od sytuacji wyjściowej i kończąc na sytuacji, w której są spełnione warunki zadania. Bardziej skomplikowana, ale także bardziej skuteczna metoda polega na budowaniu planu nie tylko od początku, ale jednocześnie także od końca ciągu operacji, badając nie tylko jakie operacje można wykonać w różnych sytuacjach, ale także jakie operacje można wykonać, aby uzyskać spełnienie zadanych warunków.

Po wyznaczeniu planu można go wykonywać, ale w fazie wykonywania również nie mamy klasycznej sytuacji realizowania raz na zawsze ustalonego algorytmu. Różne sytuacje awaryjne, błędy obsługi, odstępstwa od normy, albo po prostu napływ nowych informacji, mogą spowodować konieczność odstąpienia od aktualnego planu i wprowadzenia doń zmian, albo nawet wyznaczenie nowego. Taką nową informacją może być na przykład wiadomość o zmianie terminów i kolejności wysyłki partii kontenerów; plan, który był w naszym przykładzie optymalny dla kolejności partii 1, 2, 3, nie jest takim dla kolejności 1, 3, 2. Wobec tego w trakcie wykonywania planów jest konieczne ciągłe kontrolowanie sytuacji, aby można było aktualizować plan w każdym momencie, kiedy to staje się niezbędne.

Elementy i metody takiego programowania pragmatycznego, polegającego na określaniu przez programistę celów i repertuaru dostępnych środków, a nie algorytmów działania, są stosowane już w wielu dziedzinach zastosowań i w miarę rozwoju metod informatyki stają się coraz bardziej popularne. Najszersze zastosowanie znalazły w przetwarzaniu danych. Zauważmy, że przytoczone wyżej zdania selekcji danych precyzują cel, to znaczy warunki, jakie mają być spełnione, a nie sposób wykonania tej selekcji. Jest zupełnie obojętne, jak ona będzie wykonana, bo jest sprawą systemu zarządzania bazą danych, a nie programisty, określić, z jakich pamięci te dane będą wydobyte i w jaki sposób. Podobnie jego sprawą jest kontrola poprawności danych, pełne zabezpieczenie procesu przetwarzania i przechowywania danych, właściwa ochrona przed czynnikami, które by ten proces mogły zakłócić, właściwe stosowanie procedur odnowy po awariach lub błędach. Żaden programista nie może się tym zajmować i nie powinien, gdyż ingerencja jego mogłaby zakłócić lub uniemożliwić normalne działanie tego aparatu obsługi. Zajmowanie się drobiazgami na poziomie wykonawczym byłoby nie tylko zbędne, ale wręcz szkodliwe.

Opisane wyżej przygotowywanie planów można programować w różnych językach - bardzo często był do tego używany Lisp. Ale językiem programowania, w którym zasady programowania pragmatycznego są stosowane z całą konsekwencją, jest Prolog. Styl rozwiązywania problemów przy pomocy Prologu można pokazać na następującym, z konieczności bardzo uproszczonym przykładzie:

```
*****************************************************************
* TAK WYGLADA KOMENTARZ - ZACZYNA SIE OD % I KONCZY KROPKA      *
*****************************************************************
* PROGRAM W PROLOGU JEST BUDOWANY Z KLAUZUL;                    *
* KAZDA KLAUZULA JEST CIAGIEM PREDYKATOW,                       *
*   POPRZEDZONYCH ZNAKIEM + ALBO -,                             *
*   KTORY JEST ZAKONCZONY KROPKA LUB WYKRZYKNIKIEM;             *
* ZNAK + MOZE BYC TYLKO PRZED PIERWSZYM PREDYKATEM KLAUZULI,    *
*   WTEDY KLAUZULA KONCZY SIE KROPKA I JEST DEFINICJA;          *
* PIERWSZY PREDYKAT DEFINICJI, POPRZEDZONY +,                   *
*   JEST JEJ NAGLOWKIEM,                                        *
*   POZOSTALE, POPRZEDZONE -, SA WYWOLANIAMI (JESLI SA);        *
* KLAUZULA, ZACZYNAJACA SIE OD PREDYKATU POPRZEDZONEGO -,       *
*   KONCZY SIE WYKRZYKNIKIEM I JEST DYREKTYWA, POWODUJACA       *
*   NATYCHMIASTOWE WYKONANIE JEJ CIAGU WYWOLAN;                 *
* WYKONANIE WYWOLANIA, OKRESLONEGO PREDYKATEM, POPRZEDZONYM -,  *
*   POLEGA NA                                                   *
*   1: ZNALEZIENIU DEFINICJI Z NAGLOWKIEM O TAKIEJ SAMEJ        *
*      NAZWIE PREDYKATU,                                        *
*   2: UZGODNIENIU PARAMETROW NAGLOWKA I WYWOLANIA,             *
*   3: JESLI UZGODNIENIE JEST MOZLIWE, TO                       *
*        WYKONANIU CIAGU WYWOLAN, ZAPISANEGO W DEFINICJI,       *
*        Z UZGODNIONYMI PARAMETRAMI,                            *
*      A JESLI UZGODNIENIE NIE JEST MOZLIWE, TO                 *
*        POWTORZENIU POSTEPOWANIA OD 1: DLA KOLEJNEJ            *
*        DEFINICJI)                                             *
* WYKONANIE WYWOLANIA ZAWODZI, GDY NIE DA SIE ZNALEZC          *
*   WLASCIWEJ DEFINICJI I UZGODNIC PARAMETROW, ALBO GDY         *
*   DLA KAZDEJ TAKIEJ DEFINICJI UZGODNIENIE JEST MOZLIWE,       *
*   ALE ZAWODZI WYKONANIE KTOREGOS Z JEJ WYWOLAN;               *
* JESLI WYKONANIE WYWOLANIA ZAWIEDZIE, POWODUJE TO NAWROT       *
*    (SZCZEGOLY W PODRECZNIKU);                                 *
* NAZWY WSZYSTKICH ZMIENNYCH SA POPRZEDZONE ZNAKIEM % ;         *
*****************************************************************
*  KLAUZULE POSTACI                                             *
*      +ALFA.                                                   *
*   MOZNA ODCZYTYWAC: JESLI SPELNIONE ALFA,                     *
*      +ALFA -BETA -GAMMA -DELTA.                               *
*   MOZNA ODCZYTAC: NA TO, ABY BYLO SPELNIONE ALFA,             *
*   KONIECZNE JEST SPELNIENIE BETA I GAMMA I DELTA;             *
*  DYREKTYWE POSTACI                                            *
*      -OMEGA!                                                  *
*   MOZNA ODCZYTAC: MA BYC SPELNIONE OMEGA)                     *
*****************************************************************
+JEST(NAWIERZCHU(B),NAPOCZATKU).
+JEST(NAWIERZCHU(K),NAPOCZATKU).
+JEST(NA(B,C),NAPOCZATKU).
+JEST(NA(K,L),NAPOCZATKU).
+JEST(NA(*X,*Z),PO(*S,PRZENIES(*X,*Y,*Z)))
 -JEST(WOLNY(*Z),*S) -JEST(NA(*X,*Y),*S) -JEST(NAWIERZCHU(*X),*S).
+JEST(WOLNY(*Y),PO(*S,ZABIERZ(*X,*Y)))
 -JEST(NA(*X,*Y),*S) -JEST(NAWIERZCHU(*X),*S).
+JEST(*WAR,PO(*S,ZABIERZ(*X,*Y)))
 -JEST(*WAR,*S) -ROZNY(*WAR,NAWIERZCHU(*X)); -ROZNY(*WAR,NA(*X,*Y)).
+ROZNY(*X,*X) -/ -NIEMYJDZIE.
+ROZNY(*X,*Y).
-JEST(NA(B,L),*NAKONCU)!
```

Z dwóch dostępnych wersji Prologu wybrałem tę, która jest opisana w ogólnie dostępnym podręczniku. Druga, w której nie używa się prefiksów + i - przed predykatami, ani gwiazdek przed nazwami zmiennych, jest na razie mniej znana. Przypomina ona bardziej inne języki programowania, jeżeli chodzi o postać zenętrzną zapisów, ale tym samym bardziej oddala się od pierwowzoru Prolo u, czyli rachunku predykatów pierwszego rzędu, niż ten pierwszy wariant.

Poza predykatami, określonymi w tekście, przykład zawiera jeszcze dwa predykaty:

jest(warunek,stan) stwierdzający, że podany warunek jest w tym stanie spełniony,

różne(warunek1,warunek2) stwierdzający, że zapisy obu podanych warunków są różne i nie dadzą się uzgodnić,

oraz funkcję:

po(stan,operacja) której wartością jest stan, do którego z podanego stanu przeprowadza podana operacja.

Wykonanie wywołania -JEST(NA(B,L),*NAKONCU) wymaga znalezienia definicji, która miałaby w nagłówku taki sam predykat z podobnym termem +JEST(NA ...). Pierwsze dwa takie predykaty mają parametry, których nie da się uzgodnić, bo w pierwszym jest na drugim miejscu C, a w drugim jest na pierwszym miejscu K. Dopiero następny predykat, +JEST(NA(*X,*Z),PO(*S,PRZENIES(*X,*Y,*Z))) umożliwia takie uzgodnienie; predykaty staną się identyczne, jeżeli zamiast *X wstawi się B, zamiast *Z L i zamiast *NAKONCU będzie PO(*S,PRZENIES(B,*Y,L)).
Po uzgodnieniu parametrów przechodzimy do wykonania ciągu wywołań tej definicji. Pierwszym z tych wywołań jest -JEST(WOLNY(L),*S).
Może ono pasować tylko do nagłówka +JEST(WOLNY(*Y),PO(*S,ZABIERZ(*X,*Y))).
Uzgodnienie parametrów polega znowu na doprowadzeniu obu predykatów do tej samej postaci. Uzyskujemy to przez podstawienie L zamiast *Y oraz PO(*S,ZABIERZ(*X,L)) zamiast *S z poprzedniej klauzuli. Nie możemy dać się zmylić jednakową postacią zmiennych z różnych klauzul - w rzeczywistości są one niezależne i wszystkie uzgodnienia przeprowadza się lokalnie, tylko dla wybranych par klauzul. Nie przypisuje się zmiennym żadnych ustalonych wartości, a wykorzystuje się je tylko do zamian przeprowadzanych w opisany sposób.

Z kolei wykonanie pierwszego wywołania dla tej definicji, -JEST(NA(*X, L),*S) i uzgodnienie jego parametrów z jednym z warunków początkowych prowadzi do zastąpienia *X przez K oraz *S przez NAPOCZATKU. Następnym wywołaniem tej samej definicji jest -JEST(NAWIERZCHU(K),NAPOCZATKU), które jest zgodne z innym warunkiem początkowym. W ten sposób wszystkie wywołania dla tej definicji zakończyły się sukcesem, można więc przejść do kolejnego wywołania w ramach poprzedniej definicji. Ostatecznie po zakończeniu wszystkich działań programu otrzymujemy jako końcowy wynik wszystkich uzgodnień następującą postać dyrektywy:

-JEST(NA(B,L),PO(PO(NAPOCZATKU,ZABIERZ(K,L)),PRZENIES(B,C,L))!

BIBLIOGRAFIA.

1. Abelson H.: A beginner's guide to Logo. Byte Aug. 1982, 88-112.
2. Date C.J.: Wprowadzenie do baz danych. WNT 1981.
3. Harvey B.: Why Logo? Byte Aug. 1982, 163-193.
4. Kluźniak F., Szpakowicz S.: Prolog. WNT 1983.
5. Kluźniak F., Szpakowicz S.: Toyprolog user's manual. Tekst nie publikowany. IIUW 1984.
6. Lawler R.W.: Designing computer-based microworlds. Byte Aug. 1982, 138-160.
7. Nilsson N.J.: Artificial intelligence. Information Processing 74, Proc. of the IFIP Congress 74, North Holland 1975, 778-801.
8. Nilsson N.J.: Principles of artificial intelligence. Tioga Publ. Co. 1980.
9. Siklossy L.: Let's talk Lisp. Prentice Hall 1976.
10. Solomon C.: Introducing Logo to children. Byte Aug. 1982, 196-208.
11. Teitelman W.: Interlisp Reference Manual. Xerox Palo Alto Research Center 1973.
12. Tomaszewska M.: Oprogramowanie wyjścia graficznego dla Pascala Mera 400. Praca magisterska IIUW 1984.
13. Warren D.H.D: Warplan - a system for generating plans. DCL Memo 76, University of Edinburgh 1974.
14. Williams G.: Logo for the Apple II, the TI-99/4A, and the TRS-80 Color Computer. Byte Aug. 1982, 230-290.

Jesienna Szkoła PTI
Rydzyna, październik 1984

OCENA DZIAŁANIA SYSTEMÓW KOMPUTEROWYCH - PRZEDMIOT, METODY, KIERUNKI ROZWOJU

prof. Jan Węglarz
Instytut Automatyki
Politechniki Poznańskiej
ul. Piotrowo 3a
60-965 Poznań, tel.782375

1. Wstęp

Z punktu widzenia teorii systemów każdy system jest, ogólnie mówiąc, całością złożoną z określonych części (podsystemów), powiązanych w określony sposób dla realizacji określonych celów w określonym otoczeniu. Zagwarantowanie, przynajmniej potencjalne realizowania przez system jego celów stanowi zatem naczelne zadanie jego projektanta, stopień realizacji celów - podstawę do decyzji nabywcy, a maksymalne wykorzystanie potencjalnych możliwości w danych warunkach - główną troskę właściciela. W każdej z powyższych sytuacji, czy ogólniej - na każdym z powyższych etapów, mamy do czynienia z oceną systemu (lub jego modelu) z punktu widzenia efektywności jego działania. Oczywiście pytanie o efektywność musi zostać poprzedzone pytaniem o warunki konieczne, a więc o poprawność działania systemu, np. o stabilność systemu automatycznej regulacji, czy o równowagę statystyczną systemu komputerowego jako systemu masowej obsługi. Dochodzimy zatem do jednoznacznego wniosku, że ocena działania systemu ma podstawowe znaczenie w procesie jego projektowania,

eksploatacji i rozwoju. Choć wniosek ten istotnie jest bezsporny, to jednak jego praktyczna realizacja wygląda bardzo różnie dla różnych typów systemów. Fakt ten jest obiektywnie uzasadniony różną możliwością budowy modeli matematycznych różnych systemów w oparciu o prawa ilosciowe. Stąd bardzo zaawansowany rozwój teorii i praktyki badań ewaluacyjnych w zakresie systemów mechanicznych, elektrodynamicznych, termodynamicznych, czy ogólniej - systemów opisywanych określonymi typami równań operatorowych. Wszak niczym innym, jak działem badań ewaluacyjnych jest np. projektowanie systemów automatycznej regulacji (liniowych, nieliniowych, ciągłych, dyskretnych, stacjonarnych, niestacjonarnych, o parametrach skupionych, o parametrach rozłożonych), czy jego bardziej zaawansowane stadium - teoria sterowania optymalnego. W przypadku systemów komputerowych sytuacja wyjściowa dla badań ewaluacyjnych jest oczywiście znacznie gorsza. Mamy w nich bowiem do czynienia z wieloma całkowicie różnymi aspektami, z których tylko niektóre i to przy licznych założeniach dadzą się opisać ilościowo. Jest to prawda, która jednak, zamiast stanowić usprawiedliwienie dla nieuwzględniania badań ewaluacyjnych w nauczaniu informatyków czy w działalności projektantów systemów komputerowych, winna - wprost przeciwnie - inspirować do wysiłku. Tak właśnie dzieje się w światowej informatyce już co najmniej od lat kilkunastu [1], tak jednak nie dzieje się u nas w Polsce, gdzie, przynajmniej dotychczas, na próżno by szukać odpowiedniego przedmiotu w programie studiów na kierunku "informatyka", czy choćby jednej książki (nawet tłumaczenia) z tego zakresu. Ze względu na brak miejsca nie rozwijam tu dyskusji na temat: "Kształcenie informatyków, a rzeczywistość informatyczna w Polsce"; oponentom kształcenia w miarę możności zgodnego z współczesnymi światowymi trendami rozwojowymi w informatyce

---

1) Por. liczne konferencje i szkoły, działalność Grupy Roboczej 7.3 IFIP-u "Computer System Modelling", czasopisma: "Performance Evaluation","Systems Performance", działy w wielu czołowych czasopismach informatycznych, czy monografie, z których kilka wymieniamy w Bibliografii.

poddaję pod zastanowienie tylko dwie myśli, dotyczące interesującego nas przedmiotu (grupy przedmiotów). Po pierwsze, właściwie prowadzone zajęcia (wykład, ćwiczenia audytoryjne i laboratoryjne) z oceny działania systemów komputerowych mogłyby (i winny) stanowić jedną z głównych osi logicznych, porządkujących i integrujących cały proces kształcenia informatyków. W końcu po to głównie się ich kształci, zwłaszcza na studiach technicznych, by potrafili zaprojektować, dobrać czy dostroić system komputerowy tak, by jak najlepiej spełniał wymagania danego zastosowania (obciążenia). Po drugie, w przypadku projektowania systemów mikroprocesorowych rola tej grupy przedmiotów staje się kluczowa z uwagi na możliwość elastycznego podziału funkcji systemu między sprzęt i oprogramowanie i związaną z nią koniecznością oceny różnych rozwiązań.

Na zakończenie tych uwag wstępnych powiem jeszcze, co pominąłem, a co być może winno się we wstępie znaleźć. Otóż pominąłem dyskusję nad przetłumaczeniem terminu "performance evaluation" jako "ocena działania" oraz dyskusję koncepcji tego referatu. Zdaję sobie sprawę z niedoskonałości obu propozycji. Konsekwencje merytoryczne pierwszej są żadne, druga jest próbą realizacji zamierzenia Organizatorów Szkoły, by referaty miały charakter wprowadzający i były dostępne dla szerokiego grona odbiorców. Pełna realizacja tego zamierzenia, zwłaszcza w odniesieniu do tak obszernej i wielowątkowej problematyki jest zadaniem bardzo trudnym - wybór zagadnień, o których zdoła się choćby wspomnieć, musi być dyskusyjny.

2. Przedmiot badań ewaluacyjnych

Ogólnie można powiedzieć, że przedmiotem badań ewaluacyjnych systemów komputerowych są metody uzyskiwania informacji o działaniu i metody wpływania na działanie tych systemów w celu zapewnienia ich poprawnej i efektywnej - w określonym sensie - pracy w danych warunkach. Warunki te są określone przede wszystkim przez obciążenie systemu, a więc to wszystko, co pojawia się z otoczenia systemu na jego wejściu. Już z tego ogólnego określenia przedmiotu badań ewaluacyjnych wynika, że istotną w nich

rolę odgrywają kryteria oceny działania, jak również modelowanie obciążenia systemu. Zagadnieniom tym poświęcimy dwa odrębne rozdziały. W tym rozdziale chcielibyśmy krótko przedstawić klasyfikację problemów ewaluacyjnych i wskazać obiekty i ich charakterystyki, będące głównymi przedmiotami badań ewaluacyjnych. Przedtem jednak poczynimy kilka niezbędnych uściśleń zakresu naszych rozważań. Otóż, po pierwsze, będziemy rozpatrywali systemy scentralizowane, z wyjątkiem rozdziału 6, w którym wspomnimy o systemach rozproszonych. Oznacza to, że nie podejmiemy problemów ewaluacyjnych specyficznych dla tych ostatnich systemów, choć praktycznie wszystkie pozostałe rozważania odnoszą się również do nich. Po drugie, nie będziemy poruszać problematyki oceny działania samych programów, których efektywność, mierzona choćby czasem wykonywania i zapotrzebowaniem na pamięć, ma oczywiście wpływ na ocenę działania systemu jako całości. Dekompozycja wyodrębniająca optymalizację samych programów jest jednak dość naturalna, zwłaszcza w systemach ogólnego przeznaczenia. Musimy bowiem zwrócić w tym miejscu uwagę na oczywisty fakt, że rozwiązanie, a nawet ścisłe sformułowanie ogólnego problemu zapewnienia najwłaściwszego funkcjonowania systemu komputerowego jako całości w określonych warunkach jest praktycznie niemożliwe. Byłby to bowiem niezwykle złożony problem znalezienia najwłaściwszego (jak go zdefiniować ?) kompromisu między wieloma przeciwstawnymi sobie kryteriami, przy niezdeterminowanym opisie przynajmniej niektorych fragmentów systemu i powiązań między nimi. W większości przypadków tak ogólne postawienie problemu nie jest na szczęście potrzebne - sytuacja praktyczna narzuca zwykle jedno główne kryterium oceny (inne, wśród nich zawsze koszt, są uwzględniane w ograniczeniach) i jeden lub kilka fragmentów czy zasobów systemu mających na nie największy wpływ. Nie oznacza to negacji potrzeby podejść "globalnych", uwzględniających w rozsądnych i uzasadnionych granicach różne aspekty działania systemu, tym nie mniej rozsądna dekompozycja pozostanie zawsze cechą badań ewaluacyjnych. Inna sprawa, że dekompozycja ta bywa często dokonywana nieumiejętnie, bez dostatecznego uwzględnienia wpływu innych podsystemów z tego samego lub z innych poziomów hierarchii, czy innych aspektów funkcjonowania systemu (np. nieuwzględnianie narzutu czasowego systemu operacyjnego przy

konstruowaniu algorytmów zarządzania zasobami).

Przechodząc do klasyfikacji problemów ewaluacyjnych przypomnijmy, że istotą punktu widzenia badań ewaluacyjnych jest pytanie, jak dobrze system lub jego fragment spełnia swe funkcje. Jest rzeczą oczywistą, że pytanie to trzeba stawiać już na etapie projektowania systemu. Niestety, oczywistość ta często nie dociera do projektentów systemów komputerowych, którzy liczą na to, że żądany poziom działania systemu zawsze można będzie uzyskać na drodze jego późniejszych modyfikacji. Te ostatnie jednak przy kiepskim zaprojektowaniu systemu wyjściowego muszą prowadzić do modyfikacji sprzętowych, co, nawet pomijając koszt, implikuje zwykle odpowiednie zmiany w oprogramowaniu. Trzeba ponadto podkreślić, że na ogół na drodze wyłącznie sprzętowej nie da się uzyskać zadanych wartości kryteriów oceny działania systemu. Natomiast dobrze zaprojektowany system można dostroić do obciążenia, nie uciekając się wcale do zmian sprzętowych. Osiągi systemu i jego podatność na modyfikacje stanowią oczywiście obok ceny, główne kryteria, którymi kieruje się nabywca, dokonując wyboru spośród systemów dostępnych na rynku. Dochodzimy zatem do wyróżnienia następujących trzech klas problemów ewaluacyjnych:

- projektowania
- wyboru
- ulepszania.

Kolejność, w jakiej wymieniliśmy te problemy ma ukazywać logiczne następstwo ich rozwiązywania, choć rozwiązującymi są w ogólności różne osoby. Dodajmy, że problemy ulepszania rozumiane są szeroko i obejmują, obok strojenia, wszelkie zmiany sprzętowe. Zauważmy także, że zaliczenie problemu do jednej z powyższych klas zależy od kontekstu, w którym jest on atakowany. Przykładowo, problem określenia rozmiaru pamięci operacyjnej może być rozpatrywany jako problem projektowania, gdy należy określić minimalny rozmiar pamięci zapewniający określony poziom działania systemu dla danego zastosowania, jako problem wyboru, gdy porównuje się rozmiar pamięci oferowanych systemów z punktu widzenia potrzeb nabywcy, czy jako problem ulepszania, gdy rozmiar pamięci jest czynnikiem limitującym efektywność działania systemu dla danego obciążenia. Wynika stąd podobieństwo, a na-

wet zbieżność metod stosowanych do rozwiązywania poszczególnych problemów.

Podana wyżej klasyfikacja, zaproponowana przez Ferrariego [10], nie jest oczywiście jedyną klasyfikacją problemów ewaluacyjnych. Reprezentowany przez nią punkt widzenia jest jednak na tyle istotny, że jest ona szeroko stosowana. Wspomnijmy jeszcze o drugim, uniwersalnym punkcie widzenia, wyróżniającym problemy analizy i syntezy. Stwierdzmy tylko, że w przypadku systemów komputerowych mało jest metod syntezy - projektowanie systemu spełniającego określone wymagania efektywnościowe dokonuje się na ogół na drodze rozwiązywania problemów analizy modeli tego systemu dla różnych wartości parametrów tych modeli.

Podamy teraz przykładowe obiekty i ich charakterystyki, będące najczęściej przedmiotem badań ewaluacyjnych, zarówno w zakresie problemów projektowania, jak i ulepszania. Pominiemy problemy wyboru, jako mniej podatne na analizę ilościową; nie wymienimy również poszczególnych składników sprzętu, z których każdy może być oczywiście przedmiotem badań ewaluacyjnych z punktu widzenia liczby jednostek i parametrów techniczno-eksploatacyjnych.

Tablica 1. Przykładowe obiekty i ich charakterystyki, będące przedmiotem badań ewaluacyjnych.

Obiekty	Charakterystyki
Parametry systemu operacyjnego	Stopień wieloprogramowości Rozmiary buforów Kwant czasu Rozmiar strony
Algorytmy zarządzania zasobami	Algorytmy szeregowania Algorytmy zarządzania pamięcią Metody dostępu do pamięci zewnętrznych Algorytmy zarządzania urządzeniami zewn.
Połączenia	Połączenia: kanały - urządzenia zewnętrzne

Rozmieszczenie informacji	Rozmieszczenie zbiorów na różnych poziomach pamięci Rezydujące moduły systemu operacyjnego
Polityka cen	Opłaty za programy nieefektywne Opłaty za wykorzystanie najbardziej obciążonych zasobów Zachęty do wykorzystywania okresów małego obciążenia
Polityka przyjmowania zadań	Wykluczanie niektórych typów zadań lub użytkowników Ograniczanie obciążenia

Na zakończenie tego rozdziału wspomnijmy, że z problematyką oceny działania systemów komputerowych, jak zresztą każdej innej klasy systemów, wiąże się problematyka ich modelowania. Jest to związek tak ścisły, że słowo "modelowanie" często pomija się w nazwie tej dyscypliny informatyki, a czasem przeciwnie - stanowi ono hasło obejmujące badania ewaluacyjne. Trzeba bowiem pamiętać, że nawet w przypadku metod pomiarowych, gdy informacja pochodzi bezpośrednio z systemu, znajduje się on na ogół pod obciążeniem będącym modelem obciążenia rzeczywistego.

3. Kryteria oceny działania systemów komputerowych

Kryteria (wskaźniki), według których oceniamy jakość działania systemów komputerowych mogą być bardzo różnorodne. Znaczną ich liczbę stanowią kryteria trudne lub niemożliwe do ilościowego wymierzenia, np. moc zbioru (listy) rozkazów, łatwość obsługi systemu, dostępność serwisu, możliwość rozwoju itd. Takie kryteria są trudne do naukowej obróbki, choć oczywiście są brane pod uwagę w problemach ewaluacyjnych, zwłaszcza w wielokryterialnych problemach wyboru. Zasadniczo jednak w badaniach ewaluacyjnych rozważa się kryteria mierzalne ilościowo. Za Svobodovą [19] podzielimy te kryteria na zewnętrzne i wewnętrzne, zwane też odpowiednio kryteriami skuteczności (effec-

tiveness) systemu i jego sprawności (efficiency). Kryteria zewnętrzne są widziane przez użytkownika systemu i, ogólnie mówiąc, charakteryzują zdolność przetworzenia danego obciążenia i zdolność spełnienia wymagań czasowych użytkownika. Natomiast kryteria wewnętrzne charakteryzują obiektywną sprawność systemu, pomagając zidentyfikować przyczyny jego zbyt niskiej skuteczności. W poniższych tablicach podamy przykłady kryteriów z obu wymienionych klas.

Tablica 2. Przykłady kryteriów zewnętrznych

Kryterium	Definicja
Przepustowość	Ilość informacji [1] przetworzona przez system w jednostce czasu dla danego obciążenia
Przepustowość względna	Stosunek czasu niezbędnego do przetworzenia określonej ilości informacji z określonego obciążenia w systemie S_1 do czasu niezbędnego do przetworzenia tej samej ilości informacji z tego samego obciążenia w systemie S_2.
Przepustowość maksymalna	Maksymalna ilość informacji przetwarzana przez system w jednostce czasu dla określonego obciążenia
Czas odpowiedzi	Czas upływający między podaniem zadania [2] na wejście systemu, a uzyskaniem odpowiedzi
Dostępność	Procent czasu, w którym system jest dostępny dla użytkownika

[1] Używane są różne jednostki ilości informacji przetwarzanej przez system: praca, zadanie, program, proces, transakcja, interakcja, instrukcja. Żadna z nich nie jest niezależna od systemu (organizacja, język wewnętrzny) i od obciążenia (skład, język programowania).

[2] W systemie interakcyjnym każde zlecenie z końcówki użytkownika jest traktowane jako nowe zadanie.

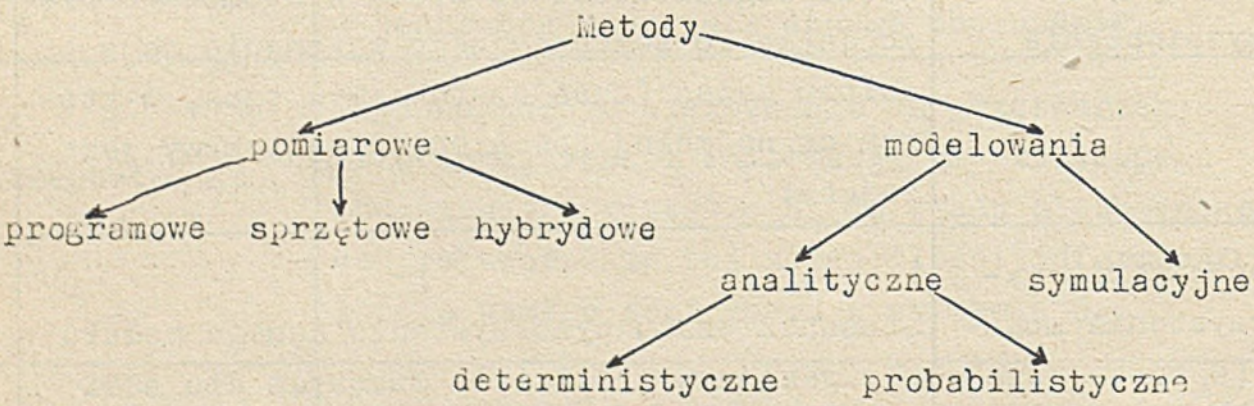
Tablica 3. Przykłady kryteriów wewnętrznych

Kryterium	Definicja
Współczynnik opóźnienia	Stosunek czasu odpowiedzi dla zadania do czasu jego przetwarzania
Czasowy współczynnik wieloprogramowości	Stosunek czasu odpowiedzi dla zadania w warunkach wieloprogramowości do czasu odpowiedzi dla tego zadania, gdy jest ono jedynym zadaniem w systemie
Współczynnik przyspieszenia	Stosunek czasu systemowego niezbędnego do wykonania danego zbioru zadań w warunkach wieloprogramowości do czasu systemowego niezbędnego do sekwencyjnego wykonania tego zbioru zadań (czas systemowy = czas, w którym co najmniej jeden moduł systemowy jest zajęty)
Współczynnik wykorzystania modułu (sprzętu, systemu operacyjnego, kompilatora etc.)	Stosunek czasu wykorzystania danego modułu w określonym przedziale czasu do długości tego przedziału
Współczynnik wykorzystania systemu	Suma ważona współczynników wykorzystania poszczególnych zasobów systemu
Narzut czasowy	Procent czasu procesora centralnego zużywany przez system operacyjny
Czas reakcji	Czas upływający między zakończeniem wprowadzania danych do systemu, a otrzymaniem pierwszego kwantu czasu procesora
Częstotliwość błędów strony	Liczba błędów strony w jednostce czasu

Współczynnik jednoczesności pracy	Procent czasu, w którym dwa (lub więcej) składniki systemu pracują równocześnie

4. Metody oceny działania systemów komputerowych

Przedstawimy teraz kilka uwag dotyczących metod, które pozwalają uzyskiwać informację o wartościach, w szczególności optymalnych, kryteriów oceny działania systemów komputerowych dla danego modelu systemu i jego obciążenia. Zacznijmy od podstawowej klasyfikacji tych metod, którą przedstawia rys.1.



Rys.1. Klasyfikacja metod oceny działania systemów komputerowych

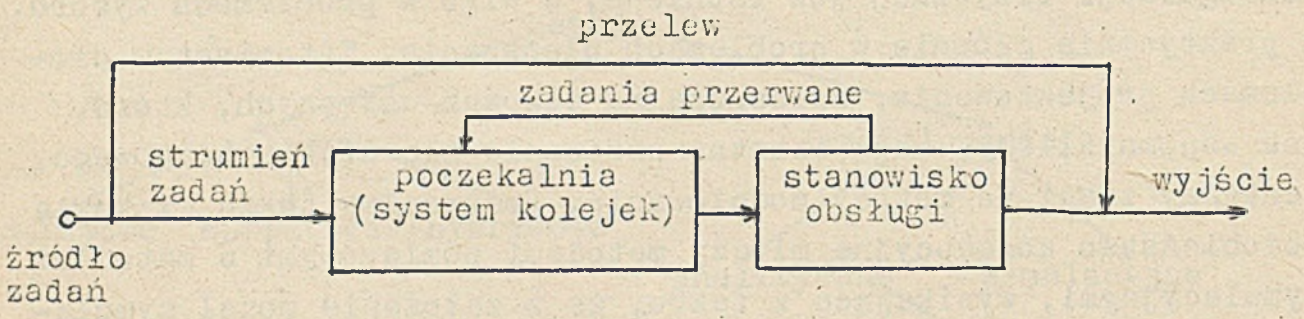
W metodach pomiarowych informacja, o której pisaliśmy na początku tego rozdziału, jest uzyskiwana bezpośrednio z systemu komputerowego, natomiast w metodach modelowania - z jego modelu. Ten ostatni może być analityczny lub symulacyjny, przy czym zastosowanie metod numerycznych do uzyskiwania rozwiązań nie przesądza o zaliczeniu metody do klasy metod symulacyjnych. O tych ostatnich mówimy tylko wtedy, gdy posługujemy się modelem symulującym zachowanie się rzeczywistego systemu w czasie jego działania. Nie jesteśmy rzecz jasna w stanie omawiać tu wymienionych trzech klas metod, z których każda posiada bogatą literaturę, nawet w rozbiciu na podklasy, choćby podane na rys.1.

Ograniczymy się zatem tylko do paru uwag ogólnych, a następnie powiemy trochę więcej o metodach analitycznych, które w pewnym sensie stanowią, a przynajmniej winny stanowić podstawę lub punkt wyjścia dla pozostałych metod. Po pierwsze bowiem, z uwagi na ponoszony koszt nie ma sensu rozwiązywać na drodze pomiarowej lub symulacyjnej tych problemów, których rozwiązanie można uzyskać metodami analitycznymi. Po drugie, trudno sobie wyobrazić właściwe zaplanowanie eksperymentu pomiarowego, a w szczególności symulacyjnego, bez znajomości zasad analitycznego opisu działania systemu. Niestety, często obserwuje się pseudopraktyczne, dyletanckie podejścia, ignorujące całkowicie metody analityczne i prowadzące do pożałowania godnych rezultatów.

Zanim przejdziemy do metod analitycznych zauważmy, że metody pomiarowe można stosować tylko wtedy, gdy system w pewnej konfiguracji fizycznej już istnieje, a więc w problemach wyboru, a praktycznie głównie w problemach ulepszania. Natomiast w problemach projektowania, zwłaszcza na etapach wstępnych, które, jak wspomnieliśmy, mają istotne znaczenie dla efektu końcowego, jesteśmy zdani na metody modelowania. Podkreślmy także głębokie podobieństwo koncepcyjne między metodami pomiarowymi a metodami symulacyjnymi, wynikające z faktu, że z założenia model symulacyjny ma działać w sposób zbliżony do rzeczywistego systemu.

Na rys.1 wyróżniliśmy wśród metod analitycznych metody deterministyczne i probabilistyczne, co wymaga krótkiego komentarza. Otóż, po pierwsze, do deterministycznych zaliczymy metody, w których nie jest wykorzystywany żaden parametr (rozumiany ogólnie - jak w teorii złożoności obliczeniowej problemów decyzyjnych) o opisie niedeterministycznym. W szczególności możemy nie dysponować żadną aprioryczną wiedzą o danym parametrze; wiedza ta nie jest także uzyskiwana w sposób adaptacyjny. Na przykład nie dysponujemy a priori żadną wiedzą o momentach przybycia zadań - momenty te stają się znane w sposób dynamiczny, wraz z przybyciem zadań do systemu. Po drugie, nie wymieniliśmy metod, w których niedeterminizm jest opisany inaczej niż w sposób probabilistyczny (np. na gruncie teorii zbiorów rozmytych), gdyż ich wykorzystanie do oceny działania systemów komputerowych jest, jak dotychczas, znikome. Wśród metod probabilistycznych najlepiej opracowane matematycznie i najszerzej wykorzystywane w praktyce są metody, w których system komputerowy jest

rozpatrywany jako system masowej obsługi. Metody te mają także najbogatszą bibliografię, że wymienimy tu tylko klasyczną monografię Kleinrocka [12] oraz pozycje książkowe [11,13,17], poświęcone ich szerszemu wykorzystaniu w badaniach ewaluacyjnych. W celu porównania podstawowych założeń przyjmowanych w metodach probabilistycznych (ściślej - masowo-obsługowych) i deterministycznych, przypomnijmy krótko metodykę postępowania w przypadku tych pierwszych, posługując się najprostszym modelem, w którym zadania ubiegają się tylko o jeden zasób dostępny w liczbie jednej jednostki. Jednostka ta (najczęściej procesor centralny) jest reprezentowana jako stanowisko obsługi w jednostanowiskowym systemie obsługi, którego ogólny schemat przedstawia rys.2.



Rys.2. Jednostanowiskowy system obsługi

Dla systemu takiego zakłada się w ogólności, że znane są:
- wymiarowość źródła zadań
- rozkład odstępu czasowego U między przybyciem kolejnych zadań
- pojemność poczekalni (rozmiar odpowiedniego bufora)
- algorytm szeregowania (regulamin obsługi)
- rozkład czasu obsługi V.

Największe znaczenie praktyczne z punktu widzenia oceny działania systemu ma analiza w stanie równowagi statystycznej, w którym zostały osiągnięte granice:

$$\lim_{t\to\infty} P[N(t)=k] = p_k,\ k=1,2,\ldots;\quad \lim_{t\to\infty} N(t) = N$$

gdzie N(t) jest liczbą zadań w systemie (w poczekalni i na sta-

nowisku obsługi) w chwili t.
Podkreślmy, że sam warunek dostateczny osiągania stanu równowagi statystycznej (w rozpatrywanym systemie ma on postać $\varrho = \lambda/\mu < 1$ gdzie $\lambda = 1/\overline{U}$, $\mu = 1/\overline{V}$) ma bardzo istotne znaczenie i w bardziej złożonych przypadkach (ogólna sieć kolejkowa) jest głównym przedmiotem badań.
Przedmiotem analizy systemu w stanie równowagi statystycznej są w ogólności rozkłady zmiennych losowych: N, W (czas oczekiwania na obsługę) i T=W+V (czas odpowiedzi). W szczególności interesują nas wartości średnie $\overline{T} = \overline{W} + 1/\mu$, $\overline{N} = \lambda \overline{T}$. Dla algorytmów szeregowania zapewniających preferencje zadaniom w funkcji ich długości (czasu trwania obsługi) bardzo istotna jest ponadto znajomość rozkładów warunkowych zmiennych losowych W i T, opisanych dystrybuantami

$$F_W(x|t) = P(W < x \mid V=t)$$

$$F_W(x|t) = P(T < x \mid V=t) \quad ,$$

a w szczególności ich wartości średnich $\overline{W}(t)$ i $\overline{T}(t)$, gdzie $\overline{T}(t) = \overline{W}(t) + t$.

Ustosunkujmy się teraz do dość rozpowszechnionego, oczywiście poza gronem specjalistów, nieporozumienia, które sprowadza się do stwierdzenia, że metody deterministyczne przyjmują silniejsze założenia niż metody masowoobsługowe i wobec tego mają mniejsze znaczenie praktyczne. Wyjaśnienie tego nieporozumienia jest tym ważniejsze, że jego źródło leży w niezrozumieniu istoty badań ewaluacyjnych w ogóle. Otóż badania te nie mają na celu wyeliminowania projektanta i zastąpienia go zautomatyzowanymi procedurami projektowania optymalnych systemów. W przypadku systemów komputerowych jest to aktualnie praktycznie niemożliwe, ze względu na hierarchiczną złożoność tego systemu, powiązań między jego podsystemami i wielorakość aspektów, które należy uwzględnić. Intuicji projektanta (zespołu projektantów) nic zatem ostatecznie nie zastąpi, można ją tylko, i to w niekiedy w znacznym stopniu, podbudować i ukierunkować. Temu właśnie celowi służą badania ewaluacyjne. Mają one to do siebie, że odpowiadają na pytania o działanie systemu lub jego fragmentów z punktu widzenia różnych kryteriów i przy różnych założeniach co do modelu systemu i jego obciążenia. W nietrywialnych

sytuacjach dopiero suma tych odpowiedzi pomaga projektantowi w dokonaniu wyboru najwłaściwszego rozwiązania. Natomiast poszczególne założenia są zwykle spełnione tylko w przybliżeniu, w odniesieniu do fragmentów systemu i jego obciążenia i lokalnie w sensie czasowym. Dotyczy to także założeń czynionych w metodach probabilistycznych, tylko że świadomość tego faktu jest zwykle mniejsza (ze szkodą dla interpretacji wyników) niż w odniesieniu do metod deterministycznych. Może wynika to z faktu, że w tych ostatnich częściej formułuje się problemy ewaluacyjne przy założeniach rzadko spełnionych w rzeczywistości, w celu np. analizy najgorszego przypadku czy zidentyfikowania mechanizmu działania algorytmów przybliżonych w sytuacjach zbliżonych do modelowych. W tym ostatnim, bardzo ważnym przypadku problem ewaluacyjny może mieć sens nawet wtedy, gdy przyjęte w nim założenia być może nigdy nie będą spełnione w rzeczywistości. Spójrzmy w świetle powyższych uwag np. na najbardziej "bulwersujące" założenie metod deterministycznych, którym jest założenie znajomości czasów wykonywania zadań. Otóż istotnie bardzo mało jest sytuacji praktycznych (myślimy o systemach komputerowych!), w których założenie to, dosłownie rozumiane, jest spełnione. Zauważmy jednak, że w deterministycznym problemie ewaluacyjnym w charakterze tych czasów mogą wystąpić górne granice rzeczywistych czasów wykonywania zadań, czy wartości średnie tych czasów, traktowanych jako zmienne losowe. Pierwsza sytuacja jest charakterystyczna dla problemów szeregowania zadań w środowisku sztywnego czasu rzeczywistego (hard real time), gdy zadania muszą zostać wykonane przed upływem narzuconych terminów (linii krytycznych). Chcąc odpowiedzieć np. na pytanie o minimalną liczbę procesorów zapewniającą wykonanie danego zbioru zadań przy takich warunkach, musimy rozwiązać odpowiedni deterministyczny problem szeregowania, w którym wystąpią górne granice czasów wykonywania zadań. Te ostatnie można łatwo wyliczyć, znając funkcje złożoności obliczeniowej czasowej odpowiednich algorytmów (zwykle są to algorytmy sterowania wykonywane dla cyklicznie pobieranych danych ze sterowanego procesu). W drugiej sytuacji, rozwiązując deterministyczny problem szeregowania w celu minimalizacji długości uszeregowania otrzymujemy optymistyczne oszacowanie wartości średniej czasu wykonania danego zbioru za-

dań [9]. Wreszcie, czasy wykonywania zadań mogą być znane a posteriori, po ich wykonaniu; rozwiązanie odpowiedniego problemu szeregowania jest wówczas pomocne dla oceny procedury szeregującej. Zauważmy także, że często zakładamy znajomość czasów wykonywania zadań, a nawet ich równość (czasy jednostkowe) po to, by konstruując dokładny (czyli znajdujący dla każdych danych uszeregowanie optymalne, jeśli ono istnieje) algorytm szeregowania dla przyjętego kryterium, rozpoznać mechanizm szeregowania, który będzie dawał dobre wyniki przybliżone z punktu widzenia tegoż kryterium, gdy czasy wykonywania zadań nie są znane, nawet w sensie probabilistycznym. W ten sposób skonstruowano wiele efektywnych algorytmów szeregowania, z których szereg przebadano z punktu widzenia zachowania się zarówno średniego, jak i w najgorszym przypadku. Ta metodyka konstrukcji algorytmów przybliżonych przez identyfikację mechanizmu działania algorytmów dokładnych dotyczy oczywiście nie tylko szeregowania zadań, ale jest powszechnie stosowana w badaniach ewaluacyjnych. Jej użyteczność jest szczególnie widoczna w przypadku złożonych modeli, obejmujących przywłaszczalne i nieprzywłaszczalne zasoby systemu, gdy zastosowanie metod masowoobsługowych jest istotnie ograniczone [3]. Metody deterministyczne stanowią podstawę dla tej metodyki, gdyż pozwalają stawiać i rozwiązywać problemy, w których poszukuje się algorytmów dokładnych, przy bardzo różnorodnych założeniach dotyczących charakterystyk czasowych zadań, żądań zasobowych, ograniczeń kolejnościowych, kryteriów oceny działania systemu etc. [2,8]. Podsumowanie tych kilku uwag prowadzi do stwierdzenia, że "porównanie" metod deterministycznych i probabilistycznych, z którym polemizowaliśmy jest istotnie nieporozumieniem. Obie klasy metod dotyczą różnych klas problemów ewaluacyjnych i zwykle dopiero ich łączne umiejętne wykorzystanie, w połączeniu z intuicją projektanta może dać właściwy efekt.

5. Modelowanie obciążenia

Właściwe zamodelowanie obciążenia systemu ma podstawowe znaczenie w badaniach ewaluacyjnych. Wynika to z faktu, że w badaniach tych prawie wyłącznie posługujemy się modelami rzeczywistego obciążenia. Tak jest zawsze w przypadku metod modelowania (analitycznych i symulacyjnych), a na ogół także w przypadku metod pomiarowych, gdy wykorzystujemy próbki rzeczywistego obciążenia, które, zaznaczmy to wyraźnie, także stanowią jego model. W tym rozdziale scharakteryzujemy krótko główne typy modeli obciążenia, przedtem jednak podamy kilka uwag ogólnych. Otóż przed przystąpieniem do konstrukcji modelu obciążenia należy sprecyzować poglądy w dwóch sprawach. Pierwsza dotyczy granic systemu, od których oczywiście zależy skład obciążenia. Zwykle zadania (procesy) użytkowników są traktowane jako zewnętrzne w stosunku do systemu, a zatem wchodzą w skład obciążenia, podczas gdy moduły systemu operacyjnego zarządzające zasobami są zaliczane do systemu. Natomiast te programy systemowe, które są wykonywane na żądanie indywidualnego użytkownika (kompilatory, asemblery, edytory etc.) tworzą "strefę graniczną", a ich zaliczenie do systemu lub obciążenia zależy od rozpatrywanego problemu ewaluacyjnego. Druga sprawa to typ i rodzaj obciążenia, które będzie modelowane (np. naukowo-techniczne, komercyjne, normalne, szczytowe) i wymagany horyzont czasowy (obciążenie godzinne, dobowe, tygodniowe, roczne). Trzeba pamiętać, że wraz ze zmniejszaniem się horyzontu czasowego pojawia się problem uwzględnienia wpływu warunków granicznych.

Ważnym zagadnieniem jest także ocena i wybór modelu obciążenia, jako że na ogół nie jest on jednoznacznie narzucony przez specyfikę problemu ewaluacyjnego. Oceny tej można dokonać z wielu punktów widzenia, z których najważniejsze są następujące:

- reprezentatywność, czyli adekwatność, w sensie wartości badanych kryteriów oceny działania, do modelowanego obciążenia rzeczywistego
- elastyczność, czyli podatność na modyfikacje odwzorowujące zmiany w modelowanym obciążeniu rzeczywistym
- koszt konstrukcji, czyli koszt zbierania informacji niezbędnej do budowy modelu

- koszt wykorzystania
- zwartość (w odniesieniu do określonej reprezentatywności i kosztu)
- niezależność od systemu, czyli możliwość przeniesienia do innego (także w sensie wprowadzonych modyfikacji) systemu bez utraty reprezentatywności
- powtarzalność
- kompatybilność z systemem lub jego modelem.

Waga powyższych kryteriów jest oczywiście zależna od typu problemu ewaluacyjnego i metody jego rozwiązywania, np. niezależność od systemu jest szczególnie ważna w problemach wyboru, a powtarzalność odgrywa istotną rolę w przypadku metod pomiarowych.

Przejdziemy teraz do krótkiego przedstawienia głównych typów modeli obciążenia.

M o d e l e n a t u r a l n e

Są to próbki rzeczywistego obciążenia, pod którym system aktualnie się znajduje. Nie zaliczamy zatem do modeli naturalnych próbek obciążenia zapamiętanych i poźniej podanych na wejście systemu. Konstrukcja tych modeli sprowadza się zatem do określenia momentów rozpoczęcia i zakończenia eksperymentów pomiarowych. Należy przy tym zwrócić baczną uwagę na reprezentatywność modelu, a więc na stacjonarność istotnych parametrów obciążenia i na zgodność uzyskiwanych z pomiarów estymatorów badanych kryteriów oceny działania systemu, czyli ich stochastyczną zbieżność do wartości tych kryteriów przy rzeczywistym obciążeniu. Przed przystąpieniem do wykorzystania modeli naturalnych należy sobie zatem przypomnieć podstawowe wiadomości z teorii estymacji. Po uwzględnieniu powyższego można uzyskać wysoką reprezentatywność modelu. Ocena modeli naturalnych z pozostałych punktów widzenia jest następująca: elastyczność - bardzo mała, koszt konstrukcji - niski, koszt wykorzystania - dość wysoki (długie sesje pomiarowe), zwartość - zwykle mała (z uwagi na wymaganą reprezentatywność) niezależność od systemu - mała (nawet niewielkie modyfikacje systemu mogą spowodować niezgodność estymatorów), powtarzalność - bardzo mała, kompatybilność - pełna. Główną zaletą tych modeli jest fakt,

że nie wymagają one przerwania normalnej pracy systemu.

W y k o n y w a l n e m o d e l e s z t u c z n e

Są to modele złożone z programów wykonywalnych w danym systemie komputerowym, a nie będące modelami naturalnymi. Wykorzystuje się je głównie w metodach pomiarowych (z wyjątkiem symulatorów sterowanych programami wykonalnymi w danym systemie) w sytuacjach, gdy modele naturalne są nieprzydatne lub uciążliwe w zastosowaniu. Z pierwszym przypadkiem mamy do czynienia np. wtedy, gdy chcemy porównać kilka systemów (problemy wyboru) lub zbadać działanie systemu pod obciążeniem przewidywanym w przyszłości. Drugi przypadek ma miejsce np. wtedy, gdy chcemy zbadać wpływ nie w pełni sprawdzonych modyfikacji systemu operacyjnego na działanie systemu. Zastosowanie modelu naturalnego narażałoby wówczas użytkowników systemu na trudne do zinterpretowania błędy, a ponadto miałoby niewielki zakres z uwagi na małą powtarzalność tego modelu.
Do najdawniej stosowanych wykonywalnych modeli sztucznych należą tak zwane mieszanki. W szczególności mieszanka rozkazowa jest zbiorem względnych częstości wykonywania głównych klas rozkazów, wspólnych dla szerokiej klasy procesorów. Oczywiście mieszanka jako taka nie jest wykonywalna, ale każdy program charakteryzujący się takimi samymi względnymi częstościami wykonywania odpowiednich rozkazów, który można łatwo napisać, jest już modelem wykonywalnym. Mieszanka rozkazowa uzyskana z pomiarów w systemie o typowej architekturze i dla typowego obciążenia (np. naukowo-technicznego) może być wykorzystywana także do badań ewaluacyjnych w podobnych systemach pod zbliżonym obciążeniem. Do takich standardowych mieszanek rozkazowych należą: mieszanka Gibsona, zmierzona w systemie IBM 7090 oraz mieszanka Flynna - w systemie IBM 360, które podajemy za [10] w Tablicy 4.

Tablica 4. Mieszanki Gibsona i Flynna

Klasa rozkazów	f [%]	
	Gibson	Flynn
ładuj do pamięci	31,2	45,1
indeksuj	18	
skok warunkowy	16,6	27,5
porównaj	3,8	10,8
arytmetyka stałoprzec.	6,9	7,6
arytmetyka zmiennoprzec.	12,2	3,2
przesunięcie logiczne	6	4,5
inne	5,3	1,3
	100,0	100,0

Mieszanki mogą odpowiadać różnym typom i rodzajom obciążenia, mogą również dotyczyć oddzielnie kompilacji i wykonywania programów. Mogą one być tworzone również na poziomie wyższym niż poziom języka wewnętrznego; na poziomie języków typu Fortran czy Algol mamy tzw. mieszanki instrukcyjne, na poziomie języków zleceń operatorskich - mieszanki zleceniowe itd. Ogólną wadą mieszanek jest to, że z samej swej natury nie zawierają one żadnej informacji o współzależnościach w modelowanym obciążeniu. Ich skuteczne zastosowanie ograniczone jest zatem do tych problemów ewaluacyjnych i architektur systemów, dla których uzasadnione jest założenie, że wpływ na zachowanie się systemu mają pojedyncze rozkazy, a nie ich ciągi. W szczególności założenia takiego nie można przyjąć w systemach z procesorami, w których wykorzystuje się zakładkowanie. Po uwzględnieniu powyższej uwagi ocena mieszanek jest następująca: reprezentatywność - średnia, elastyczność - duża, koszt konstrukcji - średni, koszt wykorzystania - niski, zwartość - duża, niezależność od systemu - dość duża, powtarzalność - pełna, kompatybilność - pełna.

Innym typem wykonywalnych modeli sztucznych są wykonywalne ślady. Zależą one oczywiście od rodzaju badanego systemu: dla systemu z przetwarzaniem wsadowym ślad składa się z chronologicznie uporządkowanych kart sterujących, dla systemu interakcyjnego zawiera zlecenia użytkowników i czasy zastanawia-

nia się użytkowników (tj. czasy od momentu zakończenia przetwarzania poprzedniego zlecenia do momentu zakończenia wprowadzania danych następnego zlecenia). Wykonywalne ślady nie mają głównej wady mieszanek, gdyż odwzorowują współzależność w modelowanym obciążeniu. Także ich reprezentatywność może być wysoka, przy odpowiednim doborze długości śladu i uwzględnieniu wpływu procedury zdejmowania śladu na pamiętane czasy zdarzeń. Ten ostatni wpływ oraz konieczność starannego inicjowania systemu dla zapewnienia powtarzalności eksperymentów (np. wszystkie pliki związane z zapamiętanymi zadaniami muszą być zapamiętane i ponownie zapisane w pamięci w miejscach, które zajmowały w czasie zdejmowania śladu) stanowią główne wady wykonywalnych śladów. W porównaniu z mieszankami cechuje je również mniejsza elastyczność, zwartość i niezależność od systemu, a wyższy koszt wykorzystania.

N i e w y k o n y w a l n e m o d e l e s z t u c z n e

Trudno o ogólną charakterystykę tych modeli, typowych dla metod modelowania, gdyż jest ich bardzo wiele i są bardzo różnorodne, począwszy od jednej lub dwóch zdeterminowanych wartości (czas procesora, rozmiar rekordu) aż do bardzo szczegółowych śladów dla symulatorów sterowanych śladem.

6. Niektóre kierunki rozwoju

Jest rzeczą oczywistą, że na kilku stronach nie sposób choćby krotko scharakteryzować licznych kierunków rozwoju badań ewaluacyjnych. Samo w miarę ścisłe określenie wielu z nich wymagałoby wprowadzenia zaawansowanych definicji, znacznie wykraczających poza ramy tego referatu. Także zacytowanie głównych prac związanych z poszczególnymi kierunkami nie jest w tych warunkach realizowalne. Jednak dla uniknięcia jakichkolwiek nieporozumień przyjmijmy, że dobór treści tego rozdziału stanowi wyłącznie wynik subiektywnych decyzji jego autora. A treść ta będzie obejmowała tak zwane podejścia globalne do poprawy działania systemów scentralizowanych oraz niektóre problemy oceny działania systemów rozproszonych. Te ostatnie wykraczają poza

ramy poprzednich rozdziałów, w których mówiliśmy zasadniczo o systemach scentralizowanych, jednąk w tym rozdziale winny być zasygnalizowane, jako że już dziś stanowią główny kierunek natarcia badań ewaluacyjnych.

Zacznijmy zatem od podejść globalnych do systemów scentralizowanych. Wyjaśnijmy, że do globalnych zaliczymy te podejścia, w których rozpatruje się więcej niż jeden rodzaj zasobów systemu komputerowego (do tego samego rodzaju zaliczamy wszystkie jednostki zasobu spełniające identyczne funkcje) [5,21]. Wyjaśnijmy też, że zwrócimy uwagę tylko na aspekty analityczne tych podejść, choć na ogół jest to dopiero podstawa do konstrukcji algorytmów, których wpływ na ocenę działania systemu jest określany na drodze odpowiednio ukierunkowanego eksperymentu symulacyjnego.

Dla metod probabilistycznych najbardziej rozpowszechnione podejscie globalne to podejście, w którym model systemu stanowi sieć kolejkowa, złożona z n wierzchołków, reprezentujących rodzaje zasobów, a więc zawierających określone liczby stanowisk obsługi (jednostek zasobu danego rodzaju). Po uzyskaniu obsługi w wierzchołku i zadanie przechodzi do wierzchołka j z prawdopodobieństwem p_{ij} lub opuszcza siec z prawdopodobieństwem $1-\sum_{j=1}^{n} p_{ij}$. Jesli w sieci znajduje się dokładnie k zadań (zewnętrzne dopływy i odpływy zadań są zabronione; $\sum_{j=1}^{n} p_{ij}=1$ dla każdego i), to mówimy o sieci zamkniętej, w przeciwieństwie do otwartej. Często bada się zamkniętą sieć kolejkową ze sprzężeniem zwrotnym do tzw. wierzchołka terminali. Zastosowanie sieci kolejkowych (zwłaszcza zamkniętych) w badaniach ewaluacyjnych dało dobre rezultaty, zwłaszcza w zakresie wykrywania wąskich gardeł, a więc wierzchołków (tak zwanych nasyconych), przy których dla $k \to \infty$ tworzą się nieskończone kolejki zadań (por. [12] tom II). Wiąże się to oczywiście z doborem liczb jednostek zasobów poszczególnych rodzajów, zapewniających osiąganie w całym systemie stanu równowagi statystycznej. Jednak szersze wykorzystanie tego podejścia jest ograniczone ze względu na trudności z modelowaniem dowolnych żądań i uwolnień zasobów, co ma istotne znaczenie m.in. dla maksymalizacji stopnia wykorzystania zasobów, zwłaszcza z uwzględnieniem rozwiązania problemów

martwego punktu (deadlock) i stałego zablokowania (permanent blocking, starvation). Także założenie znajomości odpowiednich prawdopodobieństw i rozkładów prawdopodobieństwa, wymagane w tym podejściu, może się okazać mało realistyczne w wielu zastosowaniach. Stąd dążenie do konstrukcji podejść deterministycznych, w sensie zdefiniowanym w rozdziale 4. Wśród tych podejść można wyróżnić dwa nurty. W pierwszym przyjmuje się założenia identyczne jak w klasycznej teorii szeregowania zadań na procesorach przy czym rozpatruje się także inne zasoby systemu (por. [1,20]. W przypadku zasobów nieprzywłaszczalnych przejmuje się przy tym, że wszystkie żądania zasobowe zadań są znane a priori i realizowane w całości od momentu rozpoczęcia do momentu zakończenia wykonywania zadań. W ten sposób zapobiegamy oczywiście powstaniu martwego punktu, jednak kosztem niskiego stopnia wykorzystania zasobów. Tak mocne założenia pozwalają jednakże na subtelną analizę złożoności obliczeniowej odpowiednich problemów, z określeniem granicy, do której istnieją algorytmy dokładne wielomianowe, a także na rozpoznanie mechanizmu przydziału zasobów, który można wykorzystać do konstrukcji algorytmów przybliżonych przy słabszych założeniach. Te ostatnie stanowią właśnie przedmiot zainteresowania drugiego nurtu podejść, z których wspomnimy o jednym, a raczej o całej klasie ze wspólną ideą, w którym przyjmowane założenia są najsłabsze [3]. Zakłada się w nim mianowicie znajomość (a priori) jedynie zamówień zasobowych zadań, to znaczy maksymalnych liczb jednostek zasobów poszczególnych rodzajów jednocześnie wykorzystywanych przez każde zadanie, nie zakłada się natomiast znajomości liczb żądanych i uwalnianych jednostek zasobów oraz wystąpień żądań przydziałów i uwolnień zasobów. Także momenty przybywania zadań nie są a priori znane. Ogólna idea podejścia jest następujące. Oryginalne metody: unikania martwego punktu oraz wykrywania i likwidacji stałego zablokowania, opracowane przez autora podejścia [2,3], wyznaczają w każdej chwili zbiory zadań, których żądania zasobowe mogą być spełnione. Żądania zadań w tych zbiorach są szeregowane zgodnie z algorytmami priorytetowymi, identycznymi dla zasobów przywłaszczalnych i nieprzywłaszczalnych, przy czym priorytety są nadawane w zależności od kryterium oceny działania systemu, z wykorzystaniem dostępnej wiedzy o charakterystykach zadań.

Całe podejście ma strukturę zdarzeniową, wyróżniającą algorytmy obsługi następujących zdarzeń: żądanie przydziału zasobu, uwolnienie zasobu, rozpoczęcie wykorzystywania zasobu, zakończenie wykorzystywania zasobu, redukcja zamówienia zasobowego i wykrycie zadania stale zablokowanego.

Przejdźmy obecnie do kierunków rozwoju badań ewaluacyjnych, związanych z systemami rozproszonymi. Wyróżnimy dwa, szczególnie ważne, ogólne obiekty tych badań: sieci komputerowe (SK) i rozproszone bazy danych (RBD). Dokonując takiego wyróżnienia zakładamy, że na poziomie SK nie interesuje nas aspekt samego zastosowania sieci (w tym przypadku RBD), a na poziomie RBD przyjmujemy, że istnieje SK, spełniająca wszystkie swe funkcje. W ogólności jest to oczywiście dekompozycja ogólnego problemu oceny działania systemu. Dekompozycja ta musi zresztą iść głębiej, wyróżniając w przypadku SK strukturę logiczną (wynikającą ze strukturalizacji funkcji komunikacyjnych), strukturę konfiguracyjną (wynikającą z odwzorowania struktury logicznej w składniki sieci, tj. z implementacji funkcji komunikacyjnych w tych składnikach) i strukturę topologiczną (lokalizacja węzłów, koncentratorów, terminali, zasobów informacyjnych, wybór struktury połączeń). Wybór i ocena tych struktur oraz ich wzajemne związki stanowią prawdziwą kopalnię problemów ewaluacyjnych na etapie projektowania SK. Niektóre z tych problemów zostały naświetlone w [16]. Jeśli chodzi o problemy ulepszania działania SK, to aktualnie prowadzone prace dotyczą głównie podsieci komunikacyjnej i poszczególnych protokołów. Ostatnio, wraz z rozwojem bardzo szybkich (od M bajt/sek) sieci lokalnych, dużego znaczenia nabierają problemy sterowania w czasie rzeczywistym protokołami sieci, w celu adaptacyjnej maksymalizacji wydajności i jakości usług . Ocenę metod pomiarowych w SK pod kątem rozwiązywania tych problemów zawiera praca [6], a próbę zdefiniowania odpowiednich miar oceny działania protokołów - praca [7]. W [18] przedstawiono propozycję metody oceny działania protokołów dla celów sterowania nimi w czasie rzeczywistym, wykorzystującą monitorowanie transmisji danych dla zbierania danych pomiarowych i modele formalne protokołów dla ich interpretacji.

Jeśli chodzi o rozproszone bazy danych, to szereg problemów ewaluacyjnych dotyczy wyboru struktury RBD, rozproszenia

danych i mocy obliczeniowej systemu, doboru liczby kopii danych fizycznych i optymalizacji bibliotek systemowych. Przegląd tych problemów można znaleźć w [15]. Wiele nowych problemów dotyczy zagadnień zarządzania RBD i koncentruje się w dwóch kierunkach: poprawy efektywności mechanizmów synchronizacji transakcji (por. [5,14]) oraz optymalizacji wykonywania transakcji (por.[3,4]).

Na zakończenie podkreślmy rolę, jaką w formułowaniu i rozwiązywaniu problemów ewaluacyjnych spełniają badania operacyjne. Można śmiało powiedzieć, że stanowią one podstawę metod analitycznych badań ewaluacyjnych, a w znacznej mierze także metod symulacyjnych. Dlatego dalszy rozwój tych metod wiąże się ściśle z rozwojem badań operacyjnych i z rozwojem przepływu postulatów i idei pomiędzy tymi dziedzinami. Wynika stąd również rola badań operacyjnych w kształceniu informatyków. Próbę pomocy w jej realizacji stanowi praca [2].

BIBLIOGRAFIA

1. J.Błażewicz: Problemy optymalizacji kombinatorycznej - złożoność obliczeniowa, algorytmy optymalizacyjne. PWN (w druku).
2. J.Błażewicz, W.Cellary, R.Słowiński, J.Węglarz: Badania operacyjne dla informatyków. WNT, 1983.
3. W.Cellary: Rozdział zasobów w systemach komputerowych - próby podejścia globalnego. Wyd.Uczelniane Polit.Poznańskiej, Rozprawy 127, 1981.
4. W.Cellary, Z.Królikowski, T.Morzy: Analytical model for performance evaluation of transactions. Przedstawiono do druku.
5. W.Cellary, T.Morzy: Bi-ordering approach to concurrency control in distributed database systems. Przedstawiono do druku.
6. W.Cellary, M.Stroiński: Analysis of methods of computer networks performance measurement. W: W.Bux, H.Rudin (red.): Performance of Computer Communication Systems. North-Holland, 1984.
7. W.Cellary, M.Stroiński: Performance parameters of computer network protocols refered to their formal model. Proc. IV Workshop on Protocol Specification, Testing and Verification, 1984.

8. E.G.Coffman, jr (red.): Teoria szeregowania zadań. (tłum. z angielskiego), WNT, 1980.
9. E.G.Coffman, Jr., P.J.Denning: Operating Systems Theory. Prentice-Hall, 1973.
10. D.Ferrari: Computer Systems Performance Evaluation. Prentice-Hall, 1978.
11. E.Gelenbe, I.Mitrani: Analysis and Synthesis of Computer Systems. Academic Press, 1980.
12. L.Kleinrock: Queuing Systems, Vol.I: Theory, Wiley, 1975, Vol.II: Computer Applications, Wiley, 1976.
13. H.Kobayashi: Modelling and Analysis: An Introduction to System Performance Evaluation Methodology. Addison-Wesley, 1978.
14. T.Morzy: Ordered-transaction approach to performance evaluation of concurrency control algorithms for distributed database systems. W: J.Akoka (red.): Management of Distributed Data Processing. North-Holland, 1982.
15. B.W.Lampson, M.Paul, H.J.Siegert (red.): Distributed Systems-Architecture and Implementation: An Advanced Course. Lecture Notes in Computer Sci. No.105, Springer Verlag, 1981.
16. M.Reiser: Performance Evaluation of Data Communication Systems. Proc. of the IEEE, Vol.70, No.2, Feb.1982.
17. C.H.Sauer, K.Mani Chandy: Computer Systems Performance Modelling. Prentice-Hall, 1981.
18. M.Stroiński: External measurement method for computer network protocols performance evaluation. CMG Transactions, June 1983.
19. L.Svobodova: Computer Performance Measurement and Evaluation Methods: Analysis and Applications. Elsevier, 1976.
20. J.Węglarz: Multiprocessor scheduling with memory allocation-a deterministic approach. IEEE Trans. Comput., C-29, No.8, 1980.
21. J.Węglarz: Towards global approaches to resource allocation in computer systems. Proc. INFORMATICA'82, 1982.

Jesienna Szkoła PTI
Rydzyna, październik 1984

STAN OBECNY I PERSPEKTYWY ROZWOJU UKŁADÓW LSI I VLSI

doc.dr hab.inż.Jan Zabrodzki
Instytut Informatyki
Politechnika Warszawska
ul.Nowowiejska 15/19
00-665 Warszawa, tel.25-31-79

1. Wstęp.

Począwszy od końca lat pięćdziesiątych, kiedy to pojawiły się pierwsze monolityczne układy scalone, ma miejsce stały wzrost liczby elementów scalanych w jednej strukturze. Stopień scalenia układu charakteryzuje się często określając liczbę bramek sieci logicznej równoważnej pod względem funkcjonalnym strukturze scalonego układu. Pierwsze układy scalone o złożoności kilku czy kilkunastu bramek określane były jako układy małej skali integracji /SSI/. Układy scalone o złożoności do 100 bramek przyjęto zaliczać do grupy układów średniej skali integracji /MSI/.

Układy dużej skali integracji /LSI/ o złożoności powyżej stu bramek pojawiły się na początku lat siedemdziesiątych.

Na przełomie lat siedemdziesiątych i osiemdziesiątych, w ślad za pojawieniem się układów cyfrowych o złożoności kilku tysięcy bramek, pojawiło się określenie - układy bardzo

wielkiej skali integracji /VLSI/. W odniesieniu do układów analogowych, dla których stopień scalenia rośnie wolniej niż to ma miejsce dla układów cyfrowych, nazwy układy VLSI używa się dla określenia układów o złożoności funkcjonalnej porównywalnej ze złożonością 20 wzmacniaczy operacyjnych.

Zaliczenie układu do jednej z grup SSI, MSI, LSI czy VLSI ma oczywiście charakter całkowicie umowny i orientacyjny. Odzwierciedla to jednak w pewnym stopniu złożoność procesów technologicznych wykorzystywanych do wytworzenia układu. Dla przykładu,układy VLSI zawierające kilkadziesiąt i więcej tysięcy tranzystorów są wykonywane w materiale półprzewodnika o powierzchni kilkudziesięciu mm^2 a najmniejsze wymiary poszczególnych elementów struktury są rzędu 1 μm. Osiągnięcie takiej precyzji wykonywania struktur scalonych wymagało rozwiązania wielu problemów natury czysto technologicznej. Równocześnie jednak, uzyskanie stopnia scalenia cechującego układy VLSI wymaga rozwiązania wielu innych problemów związanych z projektowaniem takich układów oraz ich testowaniem.

Na początku niniejszej pracy przedstawiono krótki przegląd obecnych osiągnięć w zakresie wytwarzania układów LSI i VLSI. W przeglądzie tym uwzględniono zarówno układy specjalizowane dostępne handlowo, jak też układy wytwarzane na zamówienie. Z kolei scharakteryzowane są wybrane problemy związane z rozwojem technologii wytwarzania układów VLSI. Dalsza część pracy poświęcona jest zagadnieniom projektowania i testowania układów LSI i VLSI, przy czym zwrócono szczególną uwagę na udział komputerów w procesie projektowania, tak w fazie syntezy jak i w fazie analizy układu. W pracy pominięta jest problematyka związana ze skutkami rozwoju układów scalonych. Jedynie w zakończeniu zwrócono uwagę na niektóre problemy wiążące się z wykorzystaniem układów LSI i VLSI w konstrukcjach systemów komputerowych.

2. Przegląd układów LSI i VLSI.

Obecnie produkowanych jest tak wiele różnych typów układów LSI i VLSI, że jakakolwiek próba ich wymieniania czy klasyfikacji mijałaby się z sensem. Dla scharakteryzowania możliwości jakie oferuje współczesna technologia półprzewodnikowa można ograniczyć się do wymienienia kilku wybranych rozwiązań. Najbardziej spektakularne osiągnięcia wiążą się z rozwojem układów mikroprocesorowych oraz pamięci półprzewodnikowych.

W zakresie mikroprocesorów, największy stopień scalenia osiągnięto przy produkcji układów mikroprocesorowych 32-bitowych. Opracowany w 1983 r. mikroprocesor firmy Hewlett-Packard, wykorzystywany w komputerze HP-9000, zawiera około 450 tysięcy tranzystorów na powierzchni 30,2 mm^2 i pozwala na pracę z częstotliwością 18 MHz. Inny 32-bitowy mikroprocesor Bellmac-32A firmy Bell zawiera 146 tysięcy tranzystorów na powierzchni 100 mm^2. Dla porównania warto dodać, że 16-bitowy mikroprocesor 8086 z 1978 r. zawierał 29 tys. tranzystorów na powierzchni 30 mm^2, a pierwszy mikroprocesor 4004 z 1971 r. zawierał około 2300 tranzystorów /układ 8080 zawiera 4500 tranzystorów na powierzchni 15 mm^2/.

Jeśli chodzi o pamięci półprzewodnikowe, to laboratoryjnie są wytwarzane układy dynamicznej pamięci RAM o pojemności 1 Mb /na powierzchni 70 mm^2/, natomiast handlowo są dostępne pamięci 256 kb o czasie dostępu 90 ns. W zakresie pamięci nieulotnych dostępne są pamięci ROM o pojemności 1 Mb i czasie dostępu 350 ns, pamięci PROM 64 kb /40 ns/, pamięci EPROM 256 kb /170ns/, pamięci E^2PROM 64 kb /200 ns/. Produkowane są również statyczne pamięci RAM o pojemności 64 kb i czasie dostępu 40 ns, 16 kb o czasie dostępu 15 ns, 4 kb o czasie dostępu 4 ns oraz 1 kb o czasie dostępu 2 ns /jest to pamięć wykonana w technologii bazującej na arsenku galu/.

Spośród innych typów układów LSI i VLSI warto wymienić specjalizowane układy mikroprocesorowe takie jak układ mikro-

procesora analogowego 2920 firmy Intel umożliwiający konwersję na postać cyfrową sygnałów o częstotliwości do 6,5 kHz /układ zawiera 9-bitowe przetworniki AD, DA, układy multipleksera analogowego, układ próbkująco-pamiętający, jednostkę ALU pozwalającą wykonywać operacje arytmetyczne na słowach 25-bitowych, pamięci RAM i EPROM/, układ TMS-320 firmy Texas Instruments przeznaczony do przetwarzania sygnałów cyfrowych, układ PF-474 firmy Proximity Technology do przeglądania baz danych, układy do realizacji złożonych systemów mikroprogramowanych /AMD 29116/, układ rozpoznawania i syntezy mowy /SP 1000, General Instruments/ itd.

Jakościowo nowe podejście zaprezentowała firma Nippon Telegraph, która na początku 1984 r. /Electronics, January 26/ opublikowała dane o pamięci o pojemności użytecznej 1.5 Mb, zorganizowanej w 256 k słów 6-bitowych i przeznaczonej dla pamiętania informacji o matrycy punktów 512 x 512, z których każdy niesie informację o jednym z 64 kolorów. Pamięć ta wykonana została na całej powierzchni płytki krzemu o średnicy 102 mm, takiej, na której w tradycyjnej technologii wykonuje się kilkadziesiąt lub kilkaset identycznych struktur, które po podzieleniu i testowaniu stanowią niezależne, identyczne układy VLSI. Odpowiednio duża obudowa układu ma 36 wyprowadzeń. Dla zapewnienia rozsądnego poziomu uzysku /p.pkt.3/ faktycznie wykonywana jest pamięć o pojemności 3,7 Mbit, co oznacza ponad 100% redundancję. Po pierwszym włączeniu pamięci wyszukiwane i eliminowane są uszkodzone fragmenty pamięci. Ponadto wbudowane są mechanizmy kontroli parzystości i inne układy zapezpieczające niezawodną pracę pamięci.

Wymienione wyżej przykładowe układy LSI i VLSI reprezentują grupę układów scalonych dostępnych handlowo, których strukturę i własności funkcjonalne określa producent, który decyduje się na poniesienie wysokich kosztów projektu wtedy, gdy przewidywane zapotrzebowanie rzędu setek tysięcy sztuk układów gwarantuje odpowiedni zysk. Obok takich układów, technologia półprzewodnikowa umożliwia w coraz większym stopniu wytwarza-

nie układów na zamówienie - są to układy produkowane w oparciu o dokumentację dostarczaną, w tej lub innej postaci przez wyłącznego odbiorcę tych układów. Uproszczenie niektórych faz projektu, kosztem niewykorzystania wszystkich możliwości oferowanych przez technologię, pozwala na istotne obniżenie kosztów projektu nowego układu.

Dla uproszczenia i skrócenia fazy projektowania wykorzystuje się dwa rodzaje rozwiązań. W jednym z nich dopuszcza się możliwość wykorzystywania w procesie projektowania jedynie standardowych, opracowanych wcześniej bramek czy zespołów funkcjonalnych. Rozwiązanie takie skraca cykl projektowy i zmniejsza granicę opłacalności podjęcia produkcji z około 100 000 sztuk do 10 000 sztuk układów. Dalsze skrócenie czasu i zmniejszenie kosztów oferują sieci tranzystorowe. Przy tym podejściu adaptacja projektu użytkownika wymaga jedynie wykonania projektu masek dla końcowego połączenia gotowych struktur tranzystorów. Opłacalność tego typu rozwiązania jest na poziomie 1 000 sztuk układów. Czas realizacji zamówień na tego typu układy dużej skali integracji jest rzędu pojedyńczych tygodni. Dla przykładu układy firmy Fujitsu umożliwiają realizację sieci logicznych o złożoności równoważnej 8000 bramek NAND o czasach propagacji 2,5 ns.

3. Wybrane problemy wytwarzania układów LSI i VLSI.

Układy LSI i VLSI są układami monolitycznymi, w których elementy czynne i bierne są wykonywane w materiale półprzewodnika. Materiałem podstawowym dla współczesnej technologii półprzewodnikowej jest krzem i przewiduje się, że będzie on dominował do końca lat osiemdziesiątych. Za ewentualnego następcę krzemu uważa się obecnie arsenek galu - materiał, który teoretycznie umożliwia uzyskanie większych szybkości działania układów /ruchliwość elektronów jest pięciokrotnie większa w arsenku galu niż w krzemie/.

Wśród monolitycznych układów scalonych wyróżnia się układy

bipolarne i układy unipolarne zależnie od tego, czy podstawowymi elementami czynnymi wykorzystywanymi do budowy tych układów są tranzystory bipolarne czy też tranzystory unipolarne typu MOS. Nie wnikając w szczegóły związane z budową i zasadą działania tych elementów można powiedzieć, że układy bipolarne charakteryzują się większą szybkością działania niż układy unipolarne, natomiast te ostatnie mają zdecydowaną przewagę, jeśli chodzi o uzyskanie dużej gęstości upakowania. W ramach technologii bipolarnej dominują obecnie układy TTL i ECL, podczas gdy w technologii unipolarnej dominują układy nMOS. Przewiduje się jednak, że coraz większego znaczenia będą nabierały układy CMOS i z nimi wiąże się nadzieje na dalszy wzrost stopnia scalenia. Jest to związane z bardzo małym poborem mocy przez układy CMOS.

Zagadnienie odprowadzania mocy wydzielanej przez strukturę VLSI jest jednym z istotniejszych zagadnień limitujących wzrost stopnia scalenia. Dla zasygnalizowania istniejących tu trudności zauważmy, że jeśli typowa obudowa struktury scalonej jest w stanie odprowadzić 1 W mocy a struktura scalona zawiera kilkaset tysięcy elementów aktywnych, to dopuszczalna moc wydzielana przez jeden element jest rzędu pojedyńczych μW. Dla porównania przypomnijmy, że moc wydzielana przez bramkę ALS TTL jest większa od 1 mW.

Dla charakteryzowania możliwości jakie stwarza technologia, jeżeli chodzi o dokładność wykonywania struktur scalonych, podaje się często albo wartość minimalnej szerokości ścieżki w strukturze bądź też minimalnego odstępu między dwoma sąsiednimi ścieżkami albo też wartość wielkości oznaczonej przez λ, która określa tolerancję z jaką mogą być wykonywane poszczególne elementy struktury. Przy określaniu wartości λ dla danej technologii uwzględnia się zarówno tolerancje wykonania elementów dla jednej maski, jak też tolerancje związane z centrowaniem kolejnych masek względem siebie. W praktyce minimalne szerokość ścieżki jest równa około 4λ. Obecnie, przemysłowo stosowane technologie pozwalają uzyskiwać szerokość ścieżek

poniżej 2 μm, natomiast laboratoryjnie osiąga się wartość poniżej 1 μm.

Przy wartościach rozdzielczości, które stają się porównywalne z długością fali świetlnej /0,75 μm dla czerwieni/, zaczynają zawodzić stosowane dotychczas powszechnie metody fotolitograficzne. Uzyskanie procesów technologicznych o większych rozdzielczościach powinny zapewnić badane obecnie metody wykorzystujące wiązki elektronów albo promienie X. Zakładając, że metody te zostaną opanowane na tyle, że będą mogły być stosowane w praktyce produkcyjnej i że zostanie pokonana kolejna bariera technologiczna, można postawić pytanie czy i kiedy pojawią się bariery, których już nie da się pokonać i osiągnie się kres możliwości zwiększania gęstości upakowania. Odpowiedź na to pytanie dają prace, które na podstawie analizy zjawisk fizycznych związanych z budową i działaniem układów cyfrowych określają bariery teoretyczne, przynajmniej dla obecnej technologii półprzewodnikowej.

Dla zmiany stanu dowolnego elementu przełączającego potrzebna jest pewna minimalna energia. Można próbować ją zmniejszać drogą minimalizacji wymiarów elementów i poprzez zmniejszanie wartości napięć zasilających. Okazuje się jednak, że w obu przypadkach występują różnego rodzaju ograniczenia, które powodują na przykład, że długość kanału tranzystora MOS nie może być mniejsza od 0,4 μm a napięcie zasilania nie powinno być mniejsze od 0,7 V. Ograniczenia te są związane z różnego rodzaju zjawiskami fizycznymi, które zaczynają odgrywać dominującą rolę przy zbyt małych wartościach wymienionych parametrów /wpływ elementów pasożytniczych, efekt tunelowania, efekt elektromigracji atomów itd./ i w rezultacie okazuje się, że po przekroczeniu pewnej granicy, zmniejszaniu wartości elementów zaczyna towarzyszyć zwiększanie wartości energii potrzebnej dla przełączania układów.

Porównując podane wyżej przykładowe wartości graniczne z wartościami, które charakteryzują obecny stan technologii

/długość kanału rzędu 1,5 μm, typowe napięcie zasilania 5 V/ widać, że różnice między nimi są jeszcze bardzo duże i można się spodziewać, że istniejąca od ponad 20 lat tendencja do dwukrotnego zwiększania skali scalenia co około dwa lata będzie się nadal utrzymywała.

Inną drogę dla zwiększania skali scalenia można by widzieć w zwiększaniu powierzchni, na której wykonywana jest struktura scalona. Jednak i w tym wypadku występują bardzo silne ograniczenia związane z aspektami technologicznymi. Podstawowym warunkiem uzyskania poprawnie działającej struktury scalonej jest czystość i jednorodność wyjściowego materiału półprzewodnikowego. Każde zanieczyszczenie powierzchni krzemu, w którym wykonuje się układ scalony oznacza praktycznie jego niesprawność. Ponieważ uzyskanie materiału o stuprocentowej czystości nie jest możliwe, zawsze część wytworzonych struktur scalonych jest niesprawna. Jeżeli prawdopodobieństwo wytworzenia struktury na powierzchni wolnej od zanieczyszczeń jest odwrotnie proporcjonalne do powierzchni zajmowanej przez układ , to chcąc otrzymać uzysk /stosunek liczby dobrych struktur do wszystkich wytworzonych struktur/ różny od zera, konieczne jest ograniczenie powierzchni zajmowanej przez pojedyńczą strukturę. Być może pewną drogę obejścia problemu niejednorodności materiałów wyjściowych będzie rozwinięcie metody zastosowanej przy omówionej wyżej pamięci firmy Nippon Telegraph.

4. Projektowanie układów VLSI.

Technologia półprzewodnikowa daje możliwość scalania coraz bardziej złożonych sieci logicznych. Równocześnie jednak, wzrost złożoności sieci stwarza wiele nowych problemów projektowych. Obecnie wyraża się opinię, że rozwój metod projektowania jest wyraźnie opóźniony w stosunku do rozwoju technologii. Koszty i czas opracowania nowego układu VLSI są limitowane przez fazę projektowania układu a nie przez fazę jego wytwarzania. Stąd też prowadzone są intensywne prace nad nowymi metodami projekto-

wania i narzędziami wspomagającymi proces projektowania. Przy projektowaniu układów VLSI powszechnie wykorzystuje się komputery we wszystkich fazach projektu. Bez udziału komputerów, zarówno czas wykonania projektu, jak i jego jakość wyrażona w postaci liczby błędów projektowych, byłyby nie do przyjęcia.

Złożoność problemu projektowania układów VLSI niejako wymusza wyróżnienie w procesie projektowania etapów realizowanych przez różnych specjalistów. W szczególności, występuje wyraźna tendencja do oddzielania fazy projektu na poziomie struktury funkcjonalnej od fazy wykonania układu scalonego. Uważa się, że podanie kilku podstawowych reguł, czy ograniczeń wynikających z możliwości technologii i ich przestrzeganie w trakcie projektu funkcjonalnego powinno wystarczać dla zapewnienia możliwości wytworzenia układu. Reguły te dotyczą najczęściej ograniczeń co do gęstości upakowania układów i określają minimalne wymiary elementów struktury oraz odległości między nimi, dopuszczalną powierzchnię struktury oraz liczbę poziomów masek. /Przy rosnącej złożoności struktur logicznych limitującym czynnikiem stają się połączenia; zajmowana przez nie powierzchnia może przekraczać powierzchnię zajmowaną przez łączone elementy. Jedną z dróg obejścia problemu jest wykonywanie połączeń na kilku warstwach, co prowadzi do zwiększania liczby poziomów masek./

Taka organizacja procesu projektowania układu VLSI, w której faza projektu struktury logicznej jest związana z technologią wytwarzania układu jedynie poprzez kilka reguł, ma istotną zaletę przy zmianie technologii bądź przy zmianie producenta wytwarzającego układ. Poprzez wprowadzenie poprawek do projektu wynikających ze zmiany reguł technologicznych można dokonywać stosunkowo szybko adaptacji gotowego projektu struktury logicznej, zwłaszcza jeżeli może to być zrealizowane automatycznie.

Pomijając w dalszym ciągu problemy związane z technologią wykonania układów, zwróćmy uwagę na skalę trudności jakie wiążą się z opracowaniem projektu układu VLSI przygotowanego do wy-

twarzania produkcyjnego. Scaleniu podlegają sieci zawierające setki tysięcy tranzystorów, równoważne pod względem złożoności funkcjonalnej wcześniejszym, dużym komputerom czy minikomputerom. Przypomnijmy na przykład, że scalone zostały jednostki centralne maszyn PDP 11, Eclipse czy też IBM 370. Jeśli chodzi o nakład pracy i czas trwania projektu, to opracowanie końcowej wersji mikroprocesora Bellmac-32A trwało 14 miesięcy i wymagało 100 "osobolat" pracy projektantów przy ogromnym udziale środków wspomagania komputerowego. Dla porównania dodajmy, że układ 4004 był opracowany przez jednego projektanta w ciągu 9 miesięcy.

W procesie projektowania układu VLSI wyróżnić można trzy etapy: etap opracowania założeń funkcjonalnych i koncepcji układu, zajmujący około 30% czasu projektu; etap projektu logicznego zajmujący około 25% czasu oraz ostatni etap zajmujący około 45% czasu poświęcony określeniu rozplanowania geometrii układu, a więc rozmieszczenia tranzystorów z uwzględnieniem elementów składających się na ich konstrukcję, sieci połączeń między tranzystorami, pól pod łączenie wyprowadzeń itd., z rozbiciem na poszczególne warstwy odpowiadające w późniejszym procesie kolejnym maskom.

Podane przykłady scalania struktur komputerów w pewnym sensie charakteryzują stan istniejący w zakresie architektury układów VLSI. Dotychczas nie pojawiły się jakościowo nowe pomysły, które pozwoliłyby odejść od podstawowych koncepcji von Neumanna. Projektanci, wzorując się na wcześniejszych rozwiązaniach, wykorzystują znane koncepcje, z których być może nie wszystkie były wcześniej stosowane praktycznie. Istotnym elementem, który ułatwia realizację niektórych koncepcji jest zmniejszenie różnicy czasów wykonywania instrukcji i czasów dostępu do pamięci dzięki scaleniu pamięci i układów przetwarzania w jednej strukturze. W układach VLSI występuje tendencja do wprowadzania równoległości działania, zarówno w sensie równoczesnego wykonywania różnych czynności przez różne, niezależnie działające bloki, jak też w sensie równoczesnego wy-

konywania kilku instrukcji przez kolejne, szeregowo połączone bloki. Stosowane są układowe rozwiązania systemów zarządzania pamięcią dla realizacji pamięci wirtualnych i protekcji pamięci systemów wieloprocesorych. Występuje tendencja do przejmowania przez "sprzęt" funkcji dawniej realizowanych programowo - ma to na celu zwiększenie szybkości wykonywania programów i wprowadzania ułatwień dla tworzenia oprogramowania i programów użytkowych oraz zabezpieczania przed błędami programowymi. Rozwijane są koncepcje scalania systemów operacyjnych /całych, jak np. w układzie 80150 zawierającym kompletny system CP/M-86, bądź zasadniczych fragmentów, które nie ulegają zmianie przy ewentualnych modyfikacjach czy rozbudowie systemu/. Stosowane są również takie architektury, które są przystosowane do realizacji określonych algorytmów /jak np. FFT, mnożenie macierzy, przeszukiwanie/.

Po ustaleniu koncepcji funkcjonalnej układu i jego organizacji następuje faza określenia struktury logicznej układu. Również i na tym poziomie, podobnie jak i poprzednio trudno jest mówić o jednolitej metodyce postępowania. Można w zasadzie wymienić jedynie kilka podstawowych tendencji. Przede wszystkim należy zwrócić uwagę na inne kryteria projektowe niż to miało miejsce w klasycznej teorii układów logicznych. W przypadku układów VLSI istotne są takie elementy jak minimalizacja powierzchni zajmowanej przez całą sieć łącznie z połączeniami, minimalizacja mocy wydzielanej przez układy wchodzące w skład sieci oraz szybkość pracy układu. Wychodząc z tych przesłanek przyjmuje się tezę o przewadze rozwiązań, które zapewniają regularność sieci logicznej. Stąd też m.inn. preferowane są rozwiązania wykorzystujące sieci PLA. Dążąc do skracania czasu wykonywania projektów wykorzystuje się metody standardowych komórek oraz sieci tranzystorów.

Jak wspomniano wcześniej, najbardziej pracochłonny jest etap wyznaczania geometrycznego rozmieszczenia struktury logicznej układu. Etap ten, przy obecnej skali scalania nie może być wykonywany ręcznie ze względu na: czas wykonania projektu, zbyt

wielką ilość danych, których człowiek nie jest w stanie zapamiętać i kontrolować oraz wiążący się z tym problem błędów, których przy ręcznym projekcie nie można uniknąć.

Wśród metod komputerowego wspomagania procesu projektowania układów VLSI wyróżnić można dwie grupy: metody analizy układu i metody syntezy. Metody analizy związane są głównie z problemami symulacji i weryfikacji projektu na wszystkich jego etapach. Metody syntezy obejmują zarówno te metody, które wspomagają projektanta pracującego interaktywnie z systemem komputerowym, jak i te, które zastępują projektanta przy realizacji określonych zadań. Ostatnio szczególnie dużo uwagi poświęca się tej ostatniej grupie metod. Powszechnie są stosowane programy automatycznego rozmieszczania elementów, prowadzenia połączeń czy też ścieśniania. Prowadzone są prace nad metodami automatycznej syntezy sieci PLA, pozwalającymi na przejście od równań boolowskich do planu geometrycznego układu. Realizowane są koncepcje automatyzacji procesu generacji planu geometrycznego na podstawie szkicu topologii układu przedstawianej w postaci symbolicznej. Prowadzone są próby z tzw. "kompilatorami krzemu", które na podstawie opisu własności funkcjonalnych w języku wysokiego poziomu, wytwarzają plan geometryczny układu. Rozważane są również możliwości wykorzystania koncepcji sztucznej inteligencji, rozwijanych ostatnio w ramach systemów ekspertowych wykorzystujących bazy wiedzy.

5. Testowanie układów VLSI.

Jedna z najpoważniejszych trudności pojawiających się wraz ze wzrostem skali scalenia wiąże się z zagadnieniem testowania wytworzonych struktur i układów. Przy scaleniu kilkudziesięciu tysięcy elementów w pojedyńczej strukturze i dostępie jedynie do kilkudziesięciu końcówek wejściowych i wyjściowych, tradycyjne metody testowania stają się nieefektywne. Różne próby szacowania czasu pełnego testowania układu VLSI prowadzą do wyników na poziomie setek czy milionów lat. W związku z tym po-

szukuje się nowych rozwiązań dla problemu testowania.

Najprostsze metody sprowadzają się do ograniczania zakresu testowania, na przykład tylko do sprawdzenia podstawowych własności funkcjonalnych, ewentualnie w połączeniu z badaniem własności pomocniczych wbudowanych struktur próbnych. Jednak w większości przypadków rozwiązanie takie nie jest zadowalające.

Obecnie uważa się, że problem testowania można rozwiązać jedynie poprzez włączenie go do zadań realizowanych w czasie projektu struktury funkcjonalnej i logicznej. W czasie projektu można przyjąć takie rozwiązania, które ułatwią problem testowania. Można to realizować rozmaicie. Na przykład, przyjmując sterowanie synchroniczne można uniknąć konieczności sprawdzania czy w sieci nie występują hazardy. Wydzielając w sieci logicznej mniejsze bloki czy podzespoły, niezależne z punktu widzenia testowania, można zadanie testowania rozbić na wiele prostszych podzadań.

W wielu wypadkach dopuszcza się rozbudowę sieci logicznej o pomocnicze układy przeznaczone wyłącznie do celów testowania. Stopień rozbudowy sięga nawet 50% pierwotnej sieci. Jednak najczęściej przyjmuje się, że rozsądny kompromis między zyskami płynącymi z uproszczenia testowania a stratami wynikającymi z pogorszenia uzysku na skutek zwiększenia powierzchni krzemu jest na poziomie 10 - 20%.

Wprowadzenie pomocniczych układów umożliwia realizację różnych metod testowania. Wykonanie dodatkowej pamięci ROM umożliwia przechowywanie programów testowania wewnętrznych podzespołów. Dodatkowe układy umożliwiają stosowanie metod analizy sygnatur w stosunku do wewnętrznych podzespołów. W niektórych metodach dąży się do zredukowania problemu do testowania sieci kombinacyjnych. W tym celu tak organizuje się strukturę logiczną by można było w czasie testowania zapisać stan wszystkich elementów pamiętających struktury do specjalnego rejestru szeregowego, a następnie zawartość tego rejestru wyprowadzić na zewnątrz.

Znana jest koncepcja, która zakłada, że wszystkie elementy pamiętające struktury logiczne są dostępne dla celów testowania, tak jak komórki pamięci RAM.

Rozwinięcie koncepcji stosowania dodatkowych układów dla celów testowania prowadzi do realizacji idei samotestowania. Z zagadnieniem tym wiążą się z kolei problemy związane z możliwościami restrukturalizacji układów, bądź to bezpośrednio po wytworzeniu układu, bądź też w trakcie jego eksploatacji po wystąpieniu niesprawności pewnych elementów struktury.

6. Zakończenie.

Wychodząc z założenia, że scharakteryzowanie stanu obecnego i perspektyw rozwoju układów VLSI jest możliwe jedynie wtedy, gdy obok danych ilościowych czy jakościowych o produkowanych układach znajdą się informacje o problemach związanych z wytwarzaniem tych układów, w pracy przedstawiono krótki przegląd zagadnień dotyczących projektowania, wytwarzania i testowania układów LSI i VLSI. Większość poruszonych problemów została jedynie zasygnalizowana - jednak powinno to dać pewien obraz złożoności zagadnień wiążących się z produkcją współczesnych układów scalonych a także ułatwić określenie kierunków prac, które są i będą rozwijane w ślad za ciągłym postępem technologii półprzewodnikowej.

Ograniczając w pracy do minimum zagadnienia technologiczne, zwrócono uwagę na te problemy, które mogą interesować informatyków, bądź jako użytkowników współczesnego sprzętu budowanego z wykorzystaniem układów LSI i VLSI, bądź też jako współuczestników procesu projektowania takich układów. Rozwój procesów wytwarzania, metod projektowania i testowania nie może odbywać się bez udziału komputerów. O ile aspekty związane z technologią wytwarzania układów VLSI leżą poza obszarem działalności informatyków, to pozostałe fazy procesu wytwarzania takich układów nie mogą rozwijać się bez udziału specjalistów z zakre-

su organizacji systemów cyfrowych, projektowania sieci logicznych, testowania, jak i programistów specjalizujących się w systemach operacyjnych, językach programowania, konstrukcji narzędzi wspomagających projektanta itp. Zapotrzebowanie na specjalistów z tych dziedzin szybko rośnie co jest wynikiem m.inn. tego, iż postęp technologii wyprzedza rozwój teorii i metod projektowania układów VLSI - metod, które pozwoliłyby pokonać barierę kosztów i czasu opracowywania nowych projektów. Przełamanie tej bariery może oznaczać przejście z obecnego etapu rozwoju określanego jako rewolucja mikroprocesorowa, do etapu rewolucji na poziomie systemów komputerowych.

Analizując problemy związane z rozwojem układów VLSI nie można pominąć zagadnień związanych z wykorzystaniem tych układów. Ostatecznie, układy te są wytwarzane po to,by stosować je w różnych urządzeniach. Modyfikacje układów muszą prowadzić do zmian w zakresie konstrukcji urządzeń, a równocześnie problemy wynikające w trakcie projektowania i eksploatacji urządzeń stanowią istotny bodziec dla dalszego rozwoju układów.

Stały wzrost stopnia scalania podstawowych układów stwarza z jednej strony możliwość realizacji systemów o coraz większej złożoności i coraz lepszych parametrach a z drugiej strony konstrukcję tanich, przenośnych urządzeń zarówno uniwersalnych jak np. komputery osobiste,jak i specjalizowanych wykorzystywanych np. w samochodach. Jest to wynikiem zmniejszania ceny pojedyńczego układu zastępującego wiele układów o mniejszym stopniu scalenia przy równoczesnym zmniejszeniu wymiarów i poborze mocy, zwiększeniu szybkości i niezawodności.

Możliwość korzystania z układów LSI i VLSI stawia konstruktorów sprzętu cyfrowego przed koniecznością zmiany stylu projektowania: projektowanie na poziomie logiki zostaje coraz częściej zastępowane projektowaniem na poziomie systemowym - problem łączenia bramek zamienia się na problem łączenia na przykład mikroprocesorów.

BIBLIOGRAFIA

1. Ahdoot K.,Alvarodiaz.,Crawley L.: IBM FSD VLSI chip design methodology. 20th Design Automation Conference Proceedings, 1983, June 27-29, s. 39-45.
2. Avenier J.P.: Digitizing, layout, rule checking - the everyday tasks of chip designers, Proc. of the IEEE, 1983, 1, s. 49-56.
3. Beyers J.W.,Zeller E.R.,Seccombe S.D.: VLSI technology packs 32-bit computer system into a small package. Hewlett-Packard Journal, 1983, 8, s. 3-6.
4. Burkhart K.P.,Forsyth M.A.,Hammer M.E.,Tankealvala D.F.: An 18-MHz , 32-bit VLSI microprocessor, Hewlett-Packard Journal, 1983, 8, s. 7-11.
5. Bursky D.: Innovative chip design leads to dense, superfast RAM's. Electronic Design, 1983, 17, s. 97-112.
6. Cushman R.H.: Digital signal processing advances slowly but steadily, EDN, 1983, 16, s. 60-72.
7. Donze R.L.,Sporzyński G.: Masterimage approach to VLSI design, Computer, 1983, 12, s. 18-25.
8. Fischetti M.A.: Solid state, IEEE Spectrum, 1984, 1, s. 58-63.
9. Gupta A.,Toong H.D.: Microprocessors - the first twelve years, Proc. of the IEEE, 1983, 11, s. 1236-1256.
10. Guterl F.: Chip architecture: a revolution brewing, IEEE Spectrum, 1983, 7, s. 30-37.
11. Mead C.,Convay L.: Introduction to VLSI systems, Addison-Wesley, 1980.
12. Murphy B.T.: Microcomputers: trends, technologies and design strategies, IEEE Journal of Solid-State Circuits, 1983, 5, s. 236-244.
13. Noyce R.N.,Hoff M.E.: A history of microprocessor development at Intel, IEEE Micro, 1981, 1, s. 8-20.
14. Reisman A.: Device, circuit, and technology scaling to micron and submicron dimensions, Proc. of the IEEE, 1983, 5, s. 550-565.
15. Schwettmann F.N.,Moll J.L.: IC process technology: VLSI and beyond, Hewlett-Packard Journal,1982, 8, s. 3-4.

16. Slamp R.,Person J.: Software that resides in silicon. VLSI Design, 1983, March/April, s. 24-28.
17. Southard J.R.: MacPitts: an approach to silicon compilation. Computer, 1983, 12, s. 74-82.
18. Tarr M.,Boudreau D.,Murphy R.: Defect analysis system speeds test and repair of redundant memories, Electronics, 1984, 1, s. 175-179.
19. Toong H.D.,Gupta A.: An architectural comparison of contemporary 16-bit microprocessors. IEEE Micro, 1981, 2, s. 26-37.
20. Williams T.W.,Parker K.P.: Design for testability: a survay. Proc. of the IEEE, 1983, 1, s. 98-112.
21. Yanilos P.N.: A dedicated comparator matches symbol strings fast and intelligently. Electronics, 1983, December 1, s. 113-117.

Poligraficzny Zakład Usługowy "POLIGRAFUS" Warszawa
Zam. 179-84r. nakład 150 T-80