SZÓSTA JESIENNA SZKOŁA PTI

WSPÓŁCZESNE KIERUNKI ROZWOJU INFORMATYKI

Organizowana przez POLSKIE TOWARZYSTWO INFORMATYCZNE

Do użytku
wewnętrznego

Mrągowo, 6 — 10 listopada 1989 r.

SZÓSTA JESIENNA SZKOŁA PTI

WSPÓŁCZESNE KIERUNKI ROZWOJU INFORMATYKI

Organizowana przez POLSKIE TOWARZYSTWO INFORMATYCZNE

Do użytku
wewnętrznego

Mrągowo, 6 — 10 listopada 1989 r.

Druk Wydawnictwa "ALFA" Nakł. 300 Zam. 1748/89

S P I S T R E Ś C I

Str.

1. Doc. Marian DĄBROWSKI (Politechnika Warszawska) 1
   PROBLEMY INFORMATYCZNE W SIECIACH TELEKOMUNIKACYJNYCH

2. Dr Jacek IRLIK (Instytut Systemów Sterowania) 41
   JAKOŚĆ OPROGRAMOWANIA

3. Prof. Jan MAŁUSZYŃSKI (Linköping University) 71
   PROGRAMOWANIE W LOGICE

4. Mgr Marek RYĆKO (Polska Akademia Nauk) 89
   SYSTEM T_EX W RĘKU AUTORA, WYDAWCY I DRUKARZA

5. Doc. Maciej SYSŁO (Uniwersytet Wrocławski) 99
   JAKIE KORZYŚCI Z KOMBINATORYKI MOŻE MIEĆ PROGRAMISTA

6. Dr Andrzej SZAŁAS (Uniwersytet Warszawski) 123
   LOGIKA I INFORMATYKA

7. Dyskusja Panelowa: Wspólny obszar informatyki

Jesienna Szkoła PTI
Mrągowo, listopad 1989

PROBLEMY INFORMATYCZNE W SIECIACH TELEKOMUNIKACYJNYCH

Marian Dąbrowski
Instytut Telekomunikacji
Politechniki Warszawskiej
00-665 Warszawa
Nowowiejska 15/19, t.252160

1. WSTĘP

Fundamentalną funkcją klasycznej telekomunikacji jest transfer informacji tzn. możliwie szybkie i wierne przekazanie informacji na drodze elektrycznej. We współczesnych systemach informacyjnych, wzrasta rola również przetwarzania informacji. Różnorodność usług niefonicznych, wynikająca z zapotrzebowania na szybką komunikację człowiek - maszyna i maszyna - maszyna, wymaga ścisłego współdziałania na poziomie procedur między źródłami generacji informacji, punktami odbioru informacji i systemami transportu informacji. Powoduje to zwiększenie zakresu współczesnej telekomunikacji i jej ściślejszego powiązania z systemami komputerowymi - głównymi elementami źródeł i ujść informacji. Dynamikę rozwoju wybranych usług we współczesnych sieciach telekomunikacyjnych prowadzi do następujących wniosków:

- na świecie zainstalowano prawie 700mln zakończeń telefonicznych i przewiduje się dalszy wzrost o ok. 4% rocznie,
- mimo wieloletniego, szybkiego rozwoju usług telematycznych, łączna liczba terminali jest o kilka rzędów wielkości niższa od zakończeń telefonicznych ale tempo wzrostu wynosi ok. 20% rocznie.

Ten stan wywołuje symbiozę informatyki i telekomunikacji, która odbywa się w trzech płaszczyznach:

1. Komputeryzacja i cyfryzacja sieci telekomunikacyjnych jako nowoczesnej infrastruktury transportu i przetwarzania informacji.
2. Oferowanie nowych usług informatycznych możliwe przy współpracy obu dziedzin.
3. Wspólne podstawy teoretyczne w zakresie teorii informacji, techniki cyfrowej, teorii systemów i oprogramowania a także, wspólna baza technologiczna.

Informatyka i telekomunikacja, istniejące kilkadziesiąt lat obok siebie, poprzez unowocześnianie się we własnym zakresie stawały się sobie potrzebne wzajemnie [1][2]. Telekomunikacja zmierza do jednolitej sieci cyfrowej z integracją usług ISDN♦), informatyka rozwija metody rozproszonego przetwarzania, obie dziedziny zmierzają do oprogramowania opartego o języki postproceduralne i systemy ekspertowe. W ostatnich latach [3], są rozważane koncepcje telekomunikacyjnych sieci inteligentnych drugiej generacji IN/2♦♦) budowane z wykorzystaniem możliwości technicznych sieci ISDN. Zainteresowane środowiska są skłonne uważać, że ewolucja systemów informacyjnych doprowadzi do systemów C&C♦♦♦)

Tablica 1. Tempo rozwoju bazy technologicznej

Technologia	Podstawowy parametr	Tempo rozwoju
Technika optyczna	Szybkość transmisji x odcinek regeneratorowy Mbit/s x km	1000 razy na 10 lat (2 razy rocznie)
Technologia krzemowa	Gęstość elementów na strukturze (pamięci)	100 razy na 10 lat (1.6 razy rocznie)
Technologia GaAs	Gęstość elementów na strukturze (pamięci)	1000 razy na 10 lat (2 razy rocznie)
Szybkość przetwarzania informacji	MIPS (miliony instrukcji na sekundę	100 razy na 10 lat (1.6 razy rocznie)

Rozwój współczesnych systemów telekomunikacyjnych jest stymulowany przez stan dostępnej technologii oraz przez wyniki współpracy z informatyką. W tablicy 1 zilustrowano tempo rozwoju bazy technologicznej [4]. Utrzymanie takiego tempa oznacza, że w latach 90-tych będą dostępne łącza o przepływności kilku Gbit/s dla odcinków między regeneratorami 100km (lub łącza 10 Gbit/s o długości 10km). Dla uzupełnienia tych możliwości transmisyjnych, technologia półprzewodników powinna umożliwić szybkości około 200Mbit/s dla CMOS, 1Gbit/s dla ECL i 3 do 5 Gbit/s dla GaAs. Postęp w technologii jest motywacją dla nowego podejścia systemowego. Pierwsze cyfrowe systemy transmisyjne PCM w technologii lampowej i tranzystorowej nie wpłynęły stymulująco na systemy komutacyjne; dopiero rozwój systemów PCM w nowych technologiach półprzewodnikowych rozpoczął etap integracji techniki i rozwój

---

♦) Integrated Services Digital Network

♦♦) IN - Intelligent Network

♦♦♦) C&C - Computer and Communications, Compunications

cyfrowych pól komutacyjnych. Wynalazek tranzystora i elektronika dyskretnych układów tranzystorowych nie spowodowała szoku w telekomutacji. Tranzystor i dioda nie stały się zamiennikami przekaźnika na zasadzie "jeden za jeden". Dopiero ustabilizowana technologia układów scalonych dużej integracji, niskiej mocy i rosnącej niezawodności jest stymulatorem nowej generacji central elektronicznych. Podobny był stosunek do rozwoju komputerów i informatyki. Powolne, zawodne komputery o prymitywnym oprogramowaniu z lat 40, 50, i 60-tych nie zostały zaakceptowane w telekomutacji. Nowoczesne metody informatyki: szybkie komputery wielozadaniowe i wieloprocesorowe o rozproszonej strukturze, nowoczesne systemy operacyjne i języki oprogramowania pozwalające na implementacje niezależnej kompilacji i współbieżności procesów zostały adaptowane i rozwinięte w systemach komutacyjnych.

2. Tendencje rozwoju struktury sieci

Zastosowanie cyfrowej techniki transmisyjnej i komutacyjnej powoduje zasadnicze zmiany w strukturze hierarchicznej sieci i w metodach jej eksploatacji. Centrale o elektromechanicznych polach komutacyjnych przestrzennych i sterowaniu w technologii przekaźnikowej na ogół nie przekraczały pojemności 10...20 tys. NN co zmuszało do budowy sieci telefonicznych o 4 lub 5 stopniach hierarchii. Współczesne centrale elektroniczne o cyfrowych polach komutacyjnych mogą mieć pojemności powyżej 100000 NN zarówno jako centrale końcowe jak i tranzytowe co pozwala na zmniejszenie hierarchiczności sieci do 3 stopni oraz zmniejszenie liczby central przy ustalonej pojemności sieci.

Rozwój wielokrotnych, cyfrowych systemów teletransmisyjnych polega na zwiększeniu krotności systemów pracujących na jednym torze co powoduje względne zmniejszenie kosztów sieci transmisyjnej w przeliczeniu na jeden kanał. Szacuje się, że *czterokrotne zwiększenie krotności systemu obniża trzykrotnie cenę transmisji jednego bitu.* Uzasadnione jest zatem zastosowanie wiązek międzycentralowych o dużej krotności. Większa pojemność central i większy wymiar wiązek międzycentralowych polepszają parametry ruchowe sieci. Sterowanie komputerowe węzłów ułatwia elastyczne kierowanie ruchem w sieciach.

Rozwój sieci cyfrowych był, do czasu zastosowania światłowodów, ograniczony ze względów technicznych, tzn. trakty cyfrowe były ograniczone do obszaru odpowiadającemu w krajowej sieci telefonicznej okręgowi telefonicznemu lub strefie numeracyjnej. Przy zastosowaniu światłowodów jednomodowych, już obecnie odcinki regeneratorowe mogą wynosić 40km dla przepływności sygnałów cyfrowych w granicach od 2Mb/s do 565Mb/s (a więc - w zakresie stosowanych obecnie zwielokrotnień systemów

PCM), a zasięg systemu transmisyjnego wydaje się praktycznie nieograniczony.

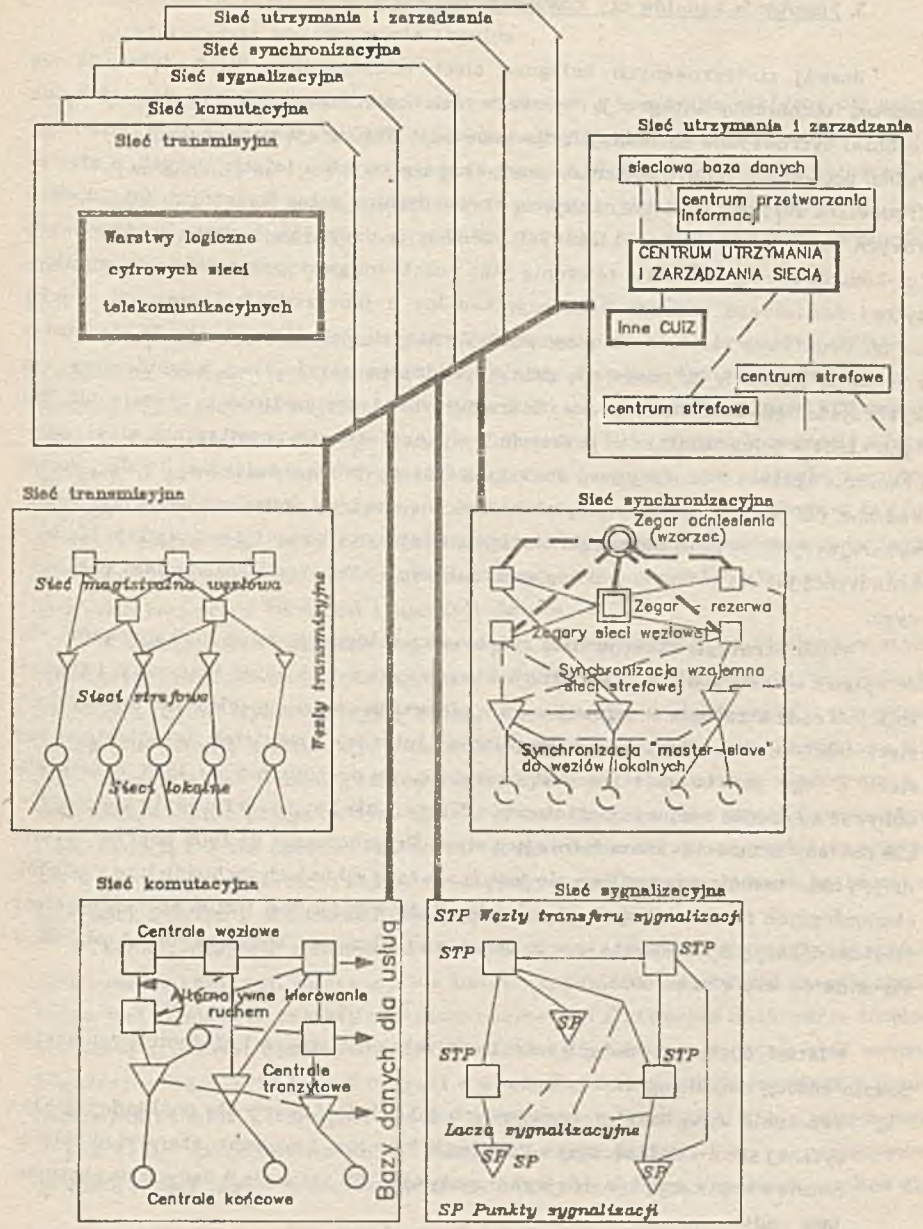
W ślad za rozwojem transmisyjnej sieci cyfrowej, następuje rozwój elektronicznych węzłów komutacyjnych. W wyniku zastosowania:

- cyfrowej magistralnej sieci transmisyjnej,
- węzłów komutacyjnych z cyfrowymi polami komutacyjnymi, sterowanymi programowo,
- zmian strukturalnych w sieci telekomunikacyjnej

ewolucji podlega logiczna struktura sieci.

W nowoczesnej sieci cyfrowej można wyróżnić następujące warstwy logiczne sieci (rys.1):

1. Międzycentralowa sieć transmisyjna odpowiedzialna za tworzenie dróg dla transportu informacji.
2. Węzły komutacji kanałów cyfrowych tworzących sieć komutacyjną.
3. Autonomiczna sieć sygnalizacyjna. Dla sieci cyfrowych przewidziany jest scentralizowany system sygnalizacji. Logicznie, wymiana sygnalizacji pomiędzy węzłami jest wymianą informacji pomiędzy procesorami sterującymi węzłów. Procesory sygnalizacyjne w węzłach połączone między sobą tworzą sieć sygnalizacyjną o cechach komputerowej sieci z komutacją pakietów.
4. Autonomiczna sieć synchronizacji węzłów. Sieć z węzłami o cyfrowych polach komutacyjnych wymaga jednolitej synchronizacji pól komutacyjnych. Wymaga się dużej dokładności zegarów w węzłach i utrzymywania długoterminowej ich stabilności; dla realizacji tych wymagań zegary węzłów sieci są wzajemnie powiązane w sieć synchronizacyjną.
5. Sieć utrzymania i zarządzania. Rozproszona struktura central końcowych (z zastosowaniem wydzielonych geograficznie koncentratorów i modułów wyniesionych) oraz wprowadzenie metod alternatywnego sterowania ruchem w sieci magistralnej uzasadnia centralizację zarządzania i utrzymania sieci; tworzone są centra eksploatacji technicznej (CET) obejmujące conajmniej kilka central. Centra CET są ze sobą powiązane w sieć zarządzania i utrzymania. Komunikacja pomiędzy CET jest tworzona jako logicznie wydzielona sieć.



Rys.1 Struktura logiczna cyfrowych sieci telekomunikacyjnych

3. Komutacja kanałów czy komutacja pakietów

Rozwój zintegrowanych usługowo sieci telekomunikacyjnych dokonuje się poprzez techniczną integrację systemów telekomunikacyjnych z wykorzystaniem techniki cyfrowej dla transmisji i dla komutacji. Konkurują ze sobą dwie tendencje. Jedna, pochodząca od projektantów sieci komputerowych w latach 70-tych, a więc ze środowiska informatycznego, optuje za wprowadzeniem usług fonicznych jako dodatkowych usług w sieciach pakietowych. Tendencja druga, pochodząca ze środowisk telekomunikacyjnych, uznaje telefonię jako podstawową usługę w sieci telekomunikacyjnej realizowaną poprzez komutację kanałów a inne rodzaje transmisji - jako usługi dodatkowe realizowane przy pomocy różnych metod komutacji. Za sieciami z komutacją pakietów przemawiają działające implementacje sieci komputerowych z wykorzystaniem techniki cyfrowej i komputerowej wprowadzone w okresie lat 70-tych kiedy w telekomunikacji powszechnie wykorzystywano przestarzałe, elektromechaniczne systemy komutacyjne i analogowe systemy teletransmisyjne. Z kolei, wprowadzane od połowy lat 70-tych elektroniczne systemy komutacyjne w telefonii, wykorzystujące tę samą bazę technologiczną co systemy komputerowe, wydają się być konkurencyjne dla integracji usług w porównaniu z komputerowymi sieciami pakietowymi.

Wybór strategii rozwoju sieci cyfrowych z integracją usług ISDN sprowadza się do wyboru właściwych metod komutacji i związanych z nimi metod transmisji i sygnalizacji. Każda strategia integracji musi w pierwszym rzędzie uwzględniać obecny stan sieci telekomunikacyjnej, a w szczególności interesy właścicieli (eksploatatorów) sieci. Z tego punktu widzenia jest oczywiste, że dochodzenie do ISDN będzie się odbywać na drodze ewolucji, dostatecznie długotrwałej by zamortyzowały się olbrzymie nakłady zainwestowane w istniejące sieci. Przechodzenie do ISDN poprzez "rewolucję", choć technicznie możliwe nie jest rozważane w krajach zachodnich ze względów ekonomicznych i komercyjnych. Ocenia się [5], że w roku 1994 liczba abonentów sieci telekomunikacyjnej korzystająca z usług niefonicznych będzie wynosić ok. 5% i wzrośnie do 40% w roku 2000.

W latach 60-tych i 70-tych integracja usług fonicznych i niefonicznych następowała według dwóch reguł:

a) Tworzenie usług teleinformatycznych jako dodatkowych dla podkładowej klasycznej sieci telefonicznej z komutacją kanałów. Klasyczna, stara sieć telefoniczna z komutacją kanałów wykorzystywana dla transmisji danych ma następujące wady:

- konieczność użycia kosztownych modemów i odpowiedniej adjustacji łączy,
- stosunkowo mała szybkość transmisji,

- często duża, nieakceptowalna stopa błędów,
- niski wskaźnik wykorzystania kanałów.

b) Tworzenie wydzielonych sieci komputerowych z komutacją pakietów dla usług teleinformatycznych a następnie - ze względów ekonomicznych - dokonywanie prób dodania do tych sieci usług fonicznych. Największą siecią doświadczalną testującą różne koncepcje komutacji pakietów jest sieć ARPANET (USA). Inne duże sieci z komutacją pakietów to DATAPAC (Kanada), TRANSPAC (Francja), TELENET, TYMENET (USA).

Klasyczna sieć telefoniczna jest niedopasowana do ruchu transmisji danych ze względu na odmienne statystyki ruchu. Typowy terminal foniczny generuje ruch poniżej 0.1E w godzinie największego ruchu (GNR) w postaci kilku rozmów kilkuminutowych. Terminal interakcyjny transmisji danych z kolei generuje znacznie więcej zgłoszeń o krótszym czasie trwania. Taki ruch powoduje przeciążenie zespołów sterujących w centralach klasycznych oraz napotyka częściej na blokadę w telefonicznych polach komutacyjnych. Ponadto systemy sygnalizacji stosowane dotychczas w ruchu telefonicznym są relatywnie powolne dla krótkich transakcji przy interakcyjnej pracy terminali transmisji danych.

W wydzielonych sieciach komputerowych, projektowanych na własnych węzłach komutacyjnych stosowano, komutację informacji (komutacja wiadomości lub komutacja pakietów), lepiej wykorzystującą kanały transmisyjne w porównaniu z klasyczną siecią telefoniczną*). Koncepcja komutacji wiadomości jest conajmniej tak stara jak sieć telegraficzna. Istotna różnica między systemami z komutacją wiadomości i z komutacją kanałów polega na tym, że te pierwsze nie pracują w czasie rzeczywistym i dopuszczają znaczne opóźnienia w dostarczaniu wiadomości. W przypadku przeciążenia w systemach z komutacją kanałów występują straty nowych zgłoszeń, natomiast w systemach komutacji wiadomości powiększają się opóźnienia.

Konkurencyjne w stosunku do klasycznej komutacji kanałów okazały się systemy z odmianą komutacji informacji - z komutacją pakietów. Wiadomość w węźle źródłowym jest dzielona na pakiety opatrzone nagłówkami służbowymi (m.in. adres docelowy, adres źródłowy, numer pakietu itd.). Pakiety przesyłane są przez sieć metodą komutacji wiadomości różnymi drogami w zależności od aktualnych możliwości przepustowych kanałów a następnie w węźle docelowym są składane w kopię wiadomości. W poprawnie zaprojektowanym systemie komutacji pakietów są wykorzystywane znacznie lepiej kanały transmisyjne sieci; kanały są zbiorem wspólnych zasobów dla

*) Kryterium wykorzystania kanałów transmisyjnych było podstawowe przy projektowaniu sieci komputerowych w latach 70-tych.

wszystkich użytkowników w znacznie krótszych odcinkach czasu. Jednakże te pozorne zalety mają swoje ograniczenia:

1) Kanały transmisyjne zajmowane są przez dużą liczbę informacji służbowych (nagłówki, potwierdzenia itp.),
2) System sterowania w każdym węźle jest bardzo obciążony. W komutacji kanałów, węzeł wykonuje "jednokrotne" zadanie sterowania dla całej transakcji. W komutacji pakietów, w każdym węźle następuje obsługa przez sterowanie każdego pakietu oddzielnie, polegająca na odbiorze, przeredagowaniu i wygenerowaniu nowego nagłówka, nadaniu potwierdzenia do węzła poprzedniego i uzyskaniu potwierdzenia od węzła następnego.

Podstawowe zalety komutacji pakietów w stosunku do komutacji wiadomości to:

1) znacznie krótsze czasy opóźnienia, ponieważ węzły nie muszą buforować całych wiadomości a jedynie pakiety,
2) jednakowa długość pakietów niezależnie od wiadomości upraszcza i usprawnia zarządzanie pamięciami buforowymi w węzłach,
3) możliwe są różne strategie obsługi dla różnych długości wiadomości (priorytety). Na przykład wiadomości krótkie, jednopakietowe przesyłane są z najwyższym priorytetem i z najmniejszym opóźnieniem, natomiast wiadomości długie są bardziej opóźniane [6],
4) przy dzieleniu wiadomości na pakiety zwiększa się prawdopodobieństwo bezbłędnej transmisji pakietu na drodze między węzłami bez konieczności retransmisji.

Dedykowane sieci z komutacją pakietów dla transmisji danych mają następujące zalety w stosunku do klasycznej sieci telefonicznej:

1) Podział zasobów. Podział zasobów oznacza, że różne typy zadań informatycznych są kierowane do komputerów specjalizowanych dla tych zadań. Istnieje jednak wiele zadań, które są typowe dla każdego lub dla większości komputerów. Dzięki podziałowi ruchu, mogą one być kierowane według aktualnej dostępności sieci. Zgłoszenie użytkownika i deklaracja typu zadania powoduje, że "system operacyjny" sieci powoduje połączenie użytkownika do odpowiedniego komputera po drodze optymalizowanej według kryteriów pracy sieci pakietowej. Podział ruchu oznacza możliwość elastycznej rozbudowy sieci: konieczność rozszerzenia możliwości obliczeniowych może oznaczać dodanie nowego identycznego komputera i zadeklarowanie go w systemie operacyjnym sieci bez zmian w wyposażeniu użytkowników.

2) **Standaryzacja styków.** Producenci sprzętu komputerowego mają komercyjne tendencje do usztywniania zasad współpracy swoich terminali ze swoimi komputerami. Choć funkcje wielu terminali i oprogramowania są takie same to nie są one kompatybilne wśród różnych producentów. Sieci teleinformatyczne wymuszają standaryzację styków i oprogramowania różnych producentów. Klasycznym standardem jest CCITT V.24 dla dostępu do łączy cyfrowych sieci telekomunikacyjnej tworzonych przy użyciu modemów. Dla sieci z komutacją pakietów standardem jest CCITT X.25 – jest to standard wyższego rzędu od V.24 definiujący nie tylko warstwę fizyczną jak V.24 ale również kontrolę błędów, formatowanie wiadomości, sterowanie przepływem informacji. W rzeczywistości, X.25 specyfikuje kilka typów styku logicznego, między innymi dla połączeń wirtualnych ustalanych na cały czas zgłoszenia, a więc podobnie jak w systemie komutacji kanałów. W praktyce terminale i komputery różnych firm, mimo postępów standaryzacji, nie są kompatybilne. Stąd istotną rolą sieci jest konwersja styków logicznych najczęściej drogą emulacji na rozwiązanie typowe, stosowane przez uznane firmy.

3) **Konwersja szybkości.** Systemy z komutacją pakietów są szczególnie wygodne dla współpracy terminali i komputerów o różnych szybkościach komunikacji. Każde z tych urządzeń współpracuje z węzłem granicznym sieci i tylko w tym węźle odczuwana jest szybkość transmisji urządzenia końcowego. Po podziale wiadomości na pakiety, transmisje międzywęzłowe odbywają się według wewnętrznej, sieciowej szybkości.

4) **Konwersja kodów.** Podobnie jak konwersja szybkości dokonywana jest konwersja kodów. Najpopularniejszym, choć nie jedynym jest kod ISO-7 (ASCII), ale i tu interpretacja poszczególnych znaków sterujących nie jest jednolita.

5) **Kontrola błędów transmisji.** Kontrola błędów w sieci z komutacją pakietów odbywa się pomiędzy każdymi dwoma węzłami tranzytującymi na zasadzie sprzężenia zwrotnego decyzji. Wykrywane przez zastosowany kod cykliczny bloki (pakiety) błędne są powtarzane na żądanie kanału powrotnego. W szczególności protokół X.25 posiada mechanizmy retransmisji błędnych bloków. Również gałęzie terminal – węzeł źródłowy lub terminal – węzeł docelowy objęte są kontrolą błędów.

6) **Alternatywne metody kierowania ruchem.** Systemy z komutacją pakietów tworzą alternatywne drogi dla połączeń w sposób naturalny wynikający z zasad działania.

7) **Podsystemy wirtualne.** Z punktu widzenia użytkownika, najczęściej interesujący jest dedykowany system wielodostępny. Na przykład obsługa kart kredytowych, rezerwacja biletów są systemami aplikacyjnymi tworzącymi zamknięte grupy użytkow-

ników. Z punktu widzenia ekonomii rozwiązania sieci teleinformatycznej, sieć powinna być technicznie jednolita ze wspólnym podziałem zasobów sieciowych co jest szczególnie wygodne dla dużej liczby niewielkich podsystemów; dla każdego podsystemu budowa sieci dedykowanej byłaby nieekonomiczna.

3.1 Włączenie usług fonicznych do sieci pakietowych

Motywacją dla integracji usług fonicznych i niefonicznych w jednej sieci pakietowej są korzyści ekonomiczne ze wspólnego wykorzystania zasobów sieci oraz kosztów jej utrzymania. Rozważane są metody pakietowego przesyłania mowy wykorzystujące porowatość sygnałów fonicznych. Zainteresowanie tymi metodami pochodzi z systemów militarnych. Przy pakietyzacji mowy wykorzystuje się również przejścia z nadmiarowego systemu na kanałach dupleksowych na system naprzemienny, podobnie jak w analogowym systemie transoceanicznym TASI oraz w systemie Bell Co. pod nazwą DSI. Integracja usług fonicznych i niefonicznych, przy założeniu komutacji kanałów jednokierunkowych (simpleksowych), oznacza de facto stosowanie metody komutacji kanałów wirtualnych. Wynika to ze szczególnych wymagań na opóźnienia dla sygnałów fonicznych. Całkowite opóźnienie sygnału fonicznego w sieci (niezależnie od liczby tranzytów) musi być ściśle ograniczone, co przy przypadkowym i zmiennym w czasie (ze względu na charakter sygnału mowy) dostarczaniu pakietów do węzła docelowego i konieczności ich sekwencyjnego szeregowania w czasie rzeczywistym wymaga dużych buforów i szybkiego przetwarzania. Drugą podstawową trudnością w pakietowym przesyłaniu mowy jest duża czułość sieci na przeciążenia przy dużym współczynniku wykorzystania kanałów. Niewielkie, chwilowe przekroczenia wskaźnika wykorzystania kanałów powodują nieproporcjonalnie większe opóźnienia pakietów. Ułatwieniem pakietowych transmisji fonicznych w stosunku do pakietowych transmisji niefonicznych są:

1) zbędność stosowania kodów cyklicznych i sprzężenia zwrotnego dla poprawy stopy błędów dla typowych kanałów,
2) rezygnacja z potwierdzania przyjętych pakietów fonicznych jeżeli prawdopodobieństwo straty pakietu w sieci (średnie) nie przekracza 1÷3%.

Jak widać, możliwa jest technicznie integracja usług fonicznych i niefonicznych w sieci pakietowej jednak obie usługi stawiają odmienne wymagania i muszą być obsługiwane różnymi protokołami. Dlatego też, nie należy traktować komutacji pakietów jako metody integracji usług fonicznych i niefonicznych ponieważ:

1) Niektóre typy usług niefonicznych jak przesyłanie długich wiadomości, np. telefax o dużej rozdzielczości są efektywniej obsługiwane przez komutację kanałów.

2) Sterowanie przepływem i duża dyspersja opóźnienia dla fonii wymagają komutacji kanałów wirtualnych.
3) Liczba węzłów tranzytowych oraz wskaźnik wykorzystania kanałów jest ograniczony z powodu dopuszczalnego całkowitego opóźnienia pakietów.
4) Protokoły wymiany pakietów fonicznych i niefonicznych muszą być różne.
5) Wyczuwanie początku nadawania segmentu rozmowy i rekonstrukcja czasowa fonii nie jest zadaniem ruchowym w sieci lecz wymaga dodatkowego wyposażenia w węzłach końcowych.
6) Podstawowe zalety pakietowego przesyłania fonii (m.in. dobre wykorzystanie kanałów transmisyjnych) mogą być spełnione w nowoczesnych sieciach z komutacją kanałów.

4. Sieci zintegrowane ISDN

Zgodnie z CCITT, ISDN jest to sieć rozwinięta z cyfrowej sieci telefonicznej zapewniająca cyfrowe połączenia od terminala do terminala (end-to-end) dla szerokiego zakresu usług, z usługami fonicznymi i niefonicznymi włącznie, do których użytkownicy mają dostęp poprzez ograniczony zbiór standardowych wieloserwisowych styków abonenckich.

Pod pojęciem ISDN rozumie się powszechną sieć cyfrową, zdolną zintegrować szeroki zakres usług fonicznych i niefonicznych (fonia, dane, tekst, obraz). Obecna, powszechna komutowana sieć telefoniczna PSTN[♦)] przekształcana jest w sieć cyfrową z automatycznie komutowanymi kanałami transmisyjnymi PCM 64kb/s. Cyfryzacja sieci prowadzi do cyfrowej sieci zintegrowanej IDN[♦♦)], integrującej technikę transmisyjną i komutacyjną. Sieć IDN, tworzona obecnie w wielu krajach, jest etapem pośrednim między PSTN a przyszłą siecią ISDN.

Prace w CCITT w zakresie ISDN koncentrują się na następujących problemach [7]:

- koncepcja i zasady ISDN (ewolucja obecnej sieci);
- możliwości usługowe;
- aspekty sieciowe i funkcjonalne, problemy ekonomiczne oraz współpraca sieci ISDN z innymi sieciami i systemami w tym z systemami wydzielonymi;
- styk użytkownik - sieć;

Dla jednolitej analizy koncepcji i protokołów ISDN przyjęto model odniesienia OSI RM (Reference Model) [8] oparty na koncepcji i zasadach modelu odniesienia ISO/CCITT OSI. Podstawową cechą modelu RM-ISDN jest rekursywne zastosowanie siedmiowarstwo-

---

♦) Public Switched Telephone Network

♦♦) Integrated Digital Network

wej struktury architektury systemów otwartych OSI w modelowaniu dwóch rodzajów przepływu informacji:

- przepływ informacji użytkownika U (user);
- przepływ informacji sterujących C (control).

Pojęcie "użytkownik" jest szerokie i obejmuje zarówno stosunkowo prosty terminal abonencki, inne sieci (sieci wydzielone i prywatne, centrale PBX, sieci LAN), jak również wyposażenia utrzymaniowe i zarządzające sieci ISDN.

Srodowisko ISDN obejmuje następujące zagadnienia:

- Komutacja kanałów. Podstawową szybkością w kanale cyfrowym będzie 64kb/s, chociaż są przewidziane inne szybkości. Komutacja kanałów jest uznana za efektywną metodę komutacji dla usług w czasie rzeczywistym (fonia) oraz dla przekazywania dużych porcji informacji. W ISDN sterowanie komutacją kanałów odbywać się będzie przy użyciu sygnalizacji scentralizowanej.

- Komutacja pakietów. Zakłada się szybkości transmisji do 64kb/s z możliwością szybkości wyższych. Komutację pakietów stosować się będzie głównie do pracy interakcyjnej - jest to usługa komplementarna do komutacji kanałów.

- Sygnalizacja scentralizowana. Będzie używana do ustanawiania, nadzorowania i rozłączania połączeń w systemie komutacji kanałów dla usług fonicznych i niefonicznych.

- Zarządzanie bazami danych. Sieciowe bazy danych będą niezbędne dla efektywnego i zunifikowanego zarządzania usługami fonicznymi i niefonicznymi oraz sprzętem dla tych usług.

- Przetwarzanie i buforowanie informacji. Integralną częścią sieci ISDN będą wyposażenia dla przetwarzania i buforowania informacji, np. wyposażenia dla takich usług telematycznych, jak: teletex, videotex, poczta elektroniczna, systemy dystrybucji wiadomości, telefax. Istnieją różne podejścia do włączania systemów przetwarzania i buforowania informacji jako integralnych części środowiska ISDN. - - Sterowanie dostępem do sieci szerokopasmowych. Powszechnie uznaje się, że sieci szerokopasmowe, o pasmach rzędu 6 MHz lub przepływnościach powyżej kilkudziesięciu Mb/s stosowane dla videokomunikacji ruchomych obrazów i ewentualnie dla szybkiej komunikacji międzykomputerowej, pozostaną przez bardzo długi czas jako wydzielone z ISDN drogi transportu informacji użytkowych, jednakże procesy sterujące (dostęp użytkowników do tych sieci) będą implementowane w jednolity sposób w sieci ISDN.

4.1 Dostęp użytkownika do sieci

W koncepcji ISDN zasady dostępu użytkownika do sieci są określone poprzez zdefiniowanie fizycznych i logicznych styków dostępu. Dla minimalizacji liczby typów

i struktury kanałów dostępu w CCITT zdefiniowano dwa typy kanałów:

a. Kanał informacyjny B o przepływności 64 kb/s;

b. Kanał sygnalizacyjny D, którego priorytetowym przeznaczeniem jest przenoszenie informacji sygnalizacyjnych dla komutacji kanałów (typ-s). Poza tym kanał ten może służyć do przenoszenia informacji w systemie komutacji pakietów (typ-p) oraz danych telemetrycznych (typ-t) o przepływności 16 kb/s lub 64 kb/s;

Obok typów kanałów, zdefiniowano struktury dostępu użytkownika odpowiednie dla kanałów fizycznych "użytkownik-sieć", z których najważniejszą jest struktura podstawowa dostępu (Basic Access) składająca się z kanałów 2B+D gdzie B=64 kb/s, D=16 kb/s.

Struktura wielokrotna, pierwotna (Primary Rate Access) tworzona z kanałów 23B+D (przepływność 1,544Mb/s, USA, Kanada, Japonia) lub 30 B+D (przepływność 2,048Mb/s, Europa). Poza tym w rozwiązaniach krajowych przewidziane są inne struktury np. struktura pośrednia n·B+D, n=6,9, B=D=64 kb/s.

Standaryzacja szybkości w kanale B 64 kb/s jest sprzeczna z postępami w zakresie cyfrowego przetwarzania mowy (systemy LPC, DM, DPCM) oraz z szybkościami transmisji danych n·2400 b/s, n=1,2,4. Zakłada się, że szybkości transmisji danych będą sprowadzane do szybkości podwielokrotnej 64 kb/s (8 lub 16 kb/s) natomiast szybkości będące podwielokrotnością 64 kb/s będą adaptowane metodą wielokrotnego wstawiania bitów w ramki kanału B.

Koncepcja dostępu fizycznego terminali do sieci ISDN polega na określeniu:

- punktów standaryzacji, to znaczy styków separujących grupy funkcji realizowanych przez każdą stronę styku;
- funkcji realizowanych przez urządzenia stykowe.

4.2 Sygnalizacja scentralizowana

Istotnym elementem sieci ISDN jest system sygnalizacji scentralizowanej zarówno w sposobie dostępu użytkownika do sieci, jak i pomiędzy węzłami komutacyjnymi. Sygnalizacja scentralizowana umożliwia sterowanie grupą połączeń w systemie komutacji kanałów poprzez jeden fizyczny kanał sygnalizacyjny. Przy łączu informacyjnym o dostępie 2B+D, sygnalizacja abonencka np. typu LAP-D jest przesyłana kanałem D do węzła komutacyjnego źródłowego, gdzie jest transformowana na sygnalizację SS7, stosowaną między węzłami. Taka separacja kanałów sygnalizacyjnych od kanałów informacyjnych (rozmównych) umożliwia projektowanie uniwersalnych protokołów sygnalizacyjnych obsługujących różne usługi. W sieci ISDN, w systemie komutacji kanałów, scentralizowany system sygnalizacji jest całkowicie odmienny od metod sygnalizacji stosowanych w systemach komutacji pakietów wykorzystujących protokoły X.21 i X.25. Model odniesienia ISO/CCITT OSI był pierwotnie opracowany dla komu-

tacji pakietów i zakładał procedury "wewnątrzszczelinowe" dla przesyłania sygnałów informacyjnych i sygnalizacyjnych przez tę samą logiczną i fizyczną drogę. W ISDN zakłada się wiele trybów komunikacji pomiędzy użytkownikami i dlatego model OSI jest adaptowany z uwzględnieniem nowych potrzeb, nie stosowanych w systemach pakietowych, a mianowicie:

- wielokanałowy styk dostępu do sieci,
- sygnalizacja scentralizowana na poziomie "użytkownik-sieć" oraz na poziomach "węzeł-węzeł" oraz "sieć-sieć".

Takie podejście do sygnalizacji umożliwia:

- komutację kanałów pod kontrolą sygnalizacji scentralizowanej;
- komutację pakietów przez kanały B oraz przez kanały D;
- sygnalizację między użytkownikiem a wyposażeniem sieci, np. komunikację z sieciowymi bazami danych (np. videotex);
- sygnalizację między terminalami użytkowników ("end-to-end") dla zmiany trybu transmisji w istniejącym połączeniu;
- kombinację wyżej wymienionych możliwości na przykład dla jednoczesnej komunikacji w kilku trybach pod kontrolą sygnalizacji scentralizowanej.

System sygnalizacji CCITT SS7 jest protokołem transmisji danych przeznaczonym dla transferu wiadomości sygnalizacyjnych i innych pomiędzy procesorami sieci telekomunikacyjnej. Organizacja systemu sygnalizacji scentralizowanej opiera się o podział systemu na dwie części:

1. Część sterowania wymianą wiadomości (sygnalizacyjnych),
2. Część "użytkową" precyzującą procedury "użytkownika", którym jest w tym przypadku dowolny podsystem funkcjonalny systemu sterowania węzłów lub sieci.

Przy projektowaniu SS7 przyjęto strukturę warstwową protokołu zgodnie z architekturą OSI. Dla SS7 przyjęto pierwsze cztery warstwy nazywane tu poziomami i w pełni je wyspecyfikowano. Poszczególne poziomy określają bloki logiczne niezależne w sposób hierarchiczny.

4.3 Integracja usług

W dotychczasowych sieciach stosowane są już niektóre usługi przewidziane w sieci zintegrowanej. Usługi te implementowane są na następujących zasadach:

- Urządzenia i kanały używane dla komunikacji i przetwarzania informacji różnych kategorii (fonia, dane, tekst, obraz) są jednofunkcyjne. Brak kompatybilności i integracji pomiędzy usługami wymusza konwersję informacji na drodze pozaelektronicznej a więc w sposób kosztowny i czasochłonny.

- Kanały komunikacyjne są specjalnie wyposażane dla różnych typów informacji.
- Korzyści z wprowadzenia określonej usługi są widoczne tylko w wąskim, specjalizowanym zakresie.

Istnieje wiele rodzajów sieci telematycznych między którymi współpraca jest uciążliwa i kosztowna; każda z tych sieci tworzy de facto autonomiczny system. Nawet taka klasyczna już usługa jak transmisja danych realizowana jest w trzech typach systemów teleinformatycznych: w sieci telefonicznej analogowej, w dedykowanych sieciach z komutacją łączy i z komutacją pakietów. Nie ma możliwości bezpośredniej współpracy różnych terminali ze względu na różne szybkości i różne procedury wymiany informacji.

Implementacja usług w sieci zintegrowanej wykazuje zalety zarówno dla eksploatacji sieci jak i dla użytkowników:

W zakresie środków technicznych

- cyfrowa technika transmisji i komutacji od terminala do terminala,
- dwa kanały po 64kb/s dla fonii i danych oraz szybki kanał sygnalizacyjny 16kb/s,
- te same drogi transmisyjne dla wszystkich kategorii informacji,

W zakresie komunikacji

- możliwość jednoczesnego zestawiania połączeń z terminala dla różnych usług do różnych użytkowników,
- możliwość przesyłania danych, tekstów lub obrazu jednocześnie z transmisją foniczną,
- jeden numer sieciowy abonenta dla wszystkich form komunikacji.

W zakresie interfejsów

- jednolity, standardowy interfejs użytkownika niezależny od producenta (jednolita norma dla producentów),
- otwartość systemów dla przyszłych usług,
- jednolite *łącze informacyjne* do abonenta zamiast różnych łączy stosowanych obecnie,
- kompatybilność systemów abonenckich z siecią publiczną.

W zakresie terminali

- możliwość produkcji terminali wielofunkcyjnych nowego typu w dużych seriach po niższych kosztach,
- standaryzacja różnorodnych obecnie interfejsów użytkowych w terminalach.

4.4 Usługi w sieciach zintegrowanych

Dotychczasowy rozwój sieci telekomunikacyjnej jest zdominowany przez telefonię. Podstawowe przepustowości kanałów transmisyjnych (3 kHz - analogowe, 64 kbit/s

- cyfrowe) wynikają z założonej jakości transmisji fonicznej. Inne usługi, poprzez zastosowanie adapterów, dopasowywały się do tych kanałów podstawowych lub wykorzystywały sporadycznie kanały o większej przepustowości.
Wynikało to z zakresu potrzebnych szybkości od telemetrii do telefonii cyfrowej, a więc, w zakresie 3 rzędów przepływności binarnej. Stąd wynika główne założenie dla koncepcji CCITT dotyczącej sieci ISDN [9], w której *za podstawę przyjęto komutacje synchronicznych, dwukierunkowych kanałów cyfrowych o przepływności 64 kbit/s.*

Tablica 2. Cechy usług telekomunikacyjnych

USŁUGI	CHARAKTERYSTYKA		
RODZAJ TYP	TYP POŁĄ-CZENIA	KANAŁ CYFROWY kb/s	ZAINTE-RESOWA-NIE
FONICZNE			
TELEFONIA	D/P-P	64	G/O
POCZTA FONICZNA	S/P-S	64	R/O
TELEKONFERENCJA	D/M-M	64	G/Z
TEKSTOWE			
TELEX	S/P-P	0.05	G/O
TELETEX	S/P-P	2.4	G/O
POCZTA ELEKTR.	S/P-S	0.3-64	G/Z
OBROBKA TEKSTOW	S/P-P	0.3-64	R/Z
DANYCH			
TELEMETRIA	S/P-S	0.01	L/Z
DOSTĘP DO BAZ	D/P-S	0.3-64	R/Z
DOSTĘP DO KOMP.	D/P-S	0.3-2000	R/Z
TRANSFER ZBIOROW	S/P-P	64-60000	R/Z
OBRAZOWE			
VIDEOTEX	D/P-S	0.3-64	R/O
TELEFAX	S/P-P	0.3-64	G/O
VIDEOFON	D/P-P	64-70000	G/O
VIDEOKONFERENCJA	D/M-M	64-70000	O/Z
DYSTRYBUCJA TV	S/S-M	>100Mb/s	L/O
GRAFIKA KOMPUT.	D/P-S	>0.5Gb/s	R/Z

Typ połączenia:
S - komunikacja jednokierunkowa,
D - komunikacja dwukierunkowa,
P-P - od punktu do punktu,
M-M - komunikacja wielopunktowa,
P-S - od punktu do podsystemu,

Zainteresowanie:
G - globalne,
R - regionalne,
L - lokalne,
O - ogólnie dostępne,
Z - zamknięte grupy.

Zintegrowany usługowo terminal powinien mieć możliwość standardowej komunikacji za pomocą fonii, danych, tekstów i obrazów przez sieć ISDN z innymi terminalami oraz określonymi podsystemami w bazach danych i systemach przetwarzania i buforowania informacji. Rozwój cywilizacyjny stymuluje nowe żądania dotyczące zakresu usług i ich jakości. Obok zwykłej telefonii rośnie zapotrzebowanie na transmisję telefoniczną wyższej jakości, odpowiadającej pasmu analogowemu 7Hkz [10] oraz na transmisję foniczną wysokiej jakości np. o jakości porównywalnej z zapisem

na płytach kompaktowych (wymagana przepustowość rzędu 1 Mbit/s bez kompresji sygnałów). Obok telewizji kablowej nadawanej według dotychczasowych standardów, przesyłanej w postaci cyfrowej (wymagana przepustowość rzędu 100 Mbit/s) są rozwijane systemy telewizyjne wysokiej jakości (HDTV) wymagające kanałów o przepustowości setek Mbit/s. Rozproszone systemy komputerowe są stosowane do zadań wykorzystujących grafikę komputerową o wysokiej rozdzielczości. Typowy staje się kolorowy obraz monitora o rozdzielczości 1000x1000 pikseli z atrybutami zapisanymi na 8-10 bitach. Komfort pracy wymaga przesłania takiego obrazu w czasie 1 sekundy, a więc wymaga kanału o przepływności 10^6 (przy zastosowaniu kompresji) lub 10^7 bit/s. Charakterystykę wybranych usług telekomunikacyjnych podano w tablicy 2.

Te nowe usługi telekomunikacyjne, z punktu widzenia planowania sieci, mają następujące cechy:

1. Przyszły "abonent zintegrowany" będzie się posługiwał terminalami pracującymi z bardzo różnymi szybkościami, różniącymi się o 7 rzędów wielkości
2. Należy się liczyć, ze względów komercyjnych, z różnymi standardami (formatami) sygnałów w poszczególnych usługach, utrudniającymi współpracę pomiędzy sieciami wydzielonymi lub regionalnymi.
3. W obecnej sieci telekomunikacyjnej przeważają połączenia "punkt-punkt" pomiędzy dwoma abonentami charakteryzujące się wymianą informacji w obu kierunkach o zbliżonej objętości (np. w telefonii). Nowe usługi mają odmienne cechy ruchowe. Objętość informacji wymieniana w obu kierunkach połączenia jest bardzo różna (np. telefax, tekstowe żądanie obrazu z komputera i graficzna odpowiedź). Coraz więcej będzie połączeń typu "punkt-wiele punktów" oraz "wiele punktów-punkt" (różnego rodzaju połączenia konferencyjne, komunikacja z bazami danych, dystrybucja wiadomości). Uzasadnia to tworzenie połączeń o asymetrycznej przepustowości kanałów w różnych kierunkach oraz szybkie tworzenie połączeń jednokierunkowych.

W poszczególnych typach usług przewidziane są (niekiedy stosowane już obecnie) usługi wspomagające (udogodnienia) z których przykładowo wymienimy użyteczne również w telefonii:

- przekazywanie numeru abonenta wywołującego - wywoływanemu,
- wybieranie skrócone,
- czasowe wyłączenie "nie zakłócać",
- połączenie z abonentem zajętym rozmową po jej zakończeniu,
- połączenie zabronione,
- zamawianie połączeń na określony czas,

- budzenie automatyczne,
- połączenia konferencyjne,
- połączenie z dobraniem "trzeciego",
- grupy zamknięte użytkowników,
- wywołania zbiorowe,
- transfer wywołań na wskazany numer,
- identyfikacja wywołań złośliwych,
- połączenia bez wybierania numeru "gorąca linia",
- przekazywanie komunikatów przez pocztę foniczną.

4.5 Charakterystyka użytkowników

Tablica 3. Przykładowe parametry ruchowe wybranych usług

Usługa	Ruch E/łącze	Zgłoszenia GNR	Czas obsł.(s)	Liczba trans. w czasieobsł.
Telef.(dom)	0.1	3.0	120	1
Telef.(biuro)	0.2	6.7	108	1
Poczta fonicz.	0.01	0.3	120	1
Teletex	0.01	5.0	50	1.5
Poczta elektr.	0.007	0.2	120	1
Dane interakc.	0.3	27	30	10
Videotex(dom)	0.03	0.45	200	17.5
Videotex(biuro)	0.2	3.6	200	17.4
Telefax	0.01	0.3	120	1
Przetw. tekstu	0.25	3.75	240	12
Przetw. danych	0.25	3.75	240	12
Telemetria	$2.7 \cdot 10^{-6}$	1.0	0.01	1
Videokonfer.	0.5	1	1800	1
Telenadzór	$2.7 \cdot 10^{-5}$	10	0.01	1
Telemedycyna	0.4	6	240	4

W tablicy 3 podano przykładowo parametry ruchowe dla usług zintegrowanych. Tak przedstawione dane ruchowe, pozornie uwzględniające wszystkie podstawowe parametry (ruch w erlangach, liczbę zgłoszeń, czas obsługi, liczbę transakcji w czasie obsługi) mogą wskazywać na dominującą rolę w sieci ruchu niefonicznego. Pominiemy usługi videokonferencyjne, wymagające kanałów o dużej przepustowości i realizowane w wydzielonych sieciach w dającej się przewidzieć przyszłości. Pozostałe usługi niefoniczne cechują się bądź małą intensywnością zgłoszeń bądź kanałami o małej przepustowości przy podanych czasach obsługi. Istotna jest objętość ruchu generowanego przez poszczególne usługi dla kanału o unormowanej przepustowości (tablica 4).

Tablica 4. Długość (w bitach) typowych wiadomości

Typ wiadomości	Długość (bit)
Typowa rozmowa telefoniczna (kanał PCM)	8·10+6
Rozmowa przez wokoder	10+5
Strona w grafice faksymilowej	2·10+5
Strona tekstu w kodzie komputerowym (1200zn.)	10+5
Typowe pismo biurowe (600 liter)	3·10+3
Telegram (15 wyrazów)	400
Fotografia gazetowa	10+5
Fotografia barwna	2·10+6
Ramka TV-color	10+6
Kadr videotelefonu	10+5

Nawet drastyczny wzrost ilościowy usług niefonicznych nie jest w stanie zachwiać dominującej roli ruchu telefonicznego. Przedstawione proporcje ruchowe stają się intuicyjnie oczywiste gdy uświadomimy sobie, że ilość informacji generowanej przez terminal telefoniczny (6 połączeń 120 sekundowych w ciągu godziny w kanale 64 kb/s) odpowiada informacji generowanej przez ponad 300 interakcyjnych terminali teledacyjnych (27 transakcji na godzinę, na każdą składają się 3 wiadomości po 1500 bitów). W [11] zamieszczono obszerniejszą dyskusję na ten temat. Wnioski często cytowane, przyjmujące arbitralnie od 20 do 50% udziału interakcyjnego ruchu teleinformatycznego, prowadzą do nieporozumień. Z technicznych powodów udział ruchu telefonicznego będzie dominujący w ISDN, podobnie jak to jest w dotychczasowej sieci telekomunikacyjnej. Postępy związane z kodowaniem sygnałów mowy metodą DPCM i ewentualne przejście sieci na kanały 32kb/s nie zmienią proporcji w sposób istotny.

Ewolucja tak potężnej infrastruktury w państwie, jaką jest sieć telekomunikacyjna, stwarza wiele rozwiązań dyskusyjnych zarówno z powodów obiektywnych, jak i subiektywnych. Rozwiązanie problemu zależy od użytkownika, administracji sieci i producentów sprzętu, przy czym każda z tych stron kieruje się swoimi interesami. Obok zrozumienia obiektywnych korzyści wynikających z docelowej sieci ISDN, na poszczególnych etapach górują interesy subiektywne każdej ze stron. Stąd, obok zachwytów koncepcją sieci ISDN, można spotkać również skrajne opinie, które dadzą się streścić w odczytywaniu skrótu ISDN jako "Integration Subscriber Don't Need".

Obecnie administracje rozwiniętych sieci telekomunikacyjnych, producenci i użytkownicy są zgodni co do konieczności rozwoju sieci i systemów cyfrowych niezależnie od typu usługi telekomunikacyjnej. Zintegrowana sieć cyfrowa jest już bardzo zaawansowana w wielu krajach. Ewolucyjne zmiany w kierunku ISDN są prognozowane powszechnie, jednak tempo i strategia tej ewolucji jest zróżnicowana.

Można zaryzykować następującą, prowokacyjną opinię: przy obecnym technicz-

nym zaawansowaniu telefonicznych systemów transmisyjnych i komutacyjnych w krajach rozwiniętych, nie powstałyby teraz sieci komputerowe z lat 70-tych (głównie z komutacją pakietów). Zintegrowane technicznie sieci cyfrowe (IDN) są już znacznie zaawansowane w wielu krajach. Jednym z elementów tej integracji są centrale elektroniczne (z komutacją kanałów). Dzięki ich możliwościom - obsługa dużego ruchu, komutacja kanałów cyfrowych, szybka sygnalizacja - zmienia się zainteresowanie usługami telematycznymi. Na przykład, nie są rozbudowywane sieci telegraficzne, natomiast rozwijane są usługi typu teletex i telefax w jednolitej sieci telefonicznej zapewniając znacznie lepsze parametry użytkowe. Budowa wyspecjalizowanych sieci (telefonicznych, telegraficznych i teleinformatycznych) powiększa koszt produkcji sprzętu i koszt eksploatacji, a także stanowi utrudnienie dla użytkowników. Na przykład sieć telegraficzna wykorzystuje bardzo mało środków technicznych wspólnie z siecią telefoniczną, natomiast musi posiadać inne, na ogół dłuższe doprowadzenia abonenckie i świadczy usługi o bardzo niskiej - jak na dzisiejsze potrzeby - jakości.

Do podstawowych problemów informatycznych wymagających konsekwentnego rozwijania należą:

- Systemy oprogramowania węzłów komutacyjnych i oprogramowania sieciowego. Oprogramowanie węzłów typu ISDN będzie zawierało około miliona instrukcji w językach wysokiego rzędu (np.CHILL) i będzie tworzone przez długie lata, stąd niezbędność metodycznych prac nad systemami rozwojowymi oprogramowania.
- Standaryzacja protokołów transferu informacji w cyfrowych łączach abonenckich oraz pomiędzy sieciami dedykowanymi.
- Rozwój sygnalizacji scentralizowanej CCITT CCSS-7 i sygnalizacji abonenckiej oraz ich implementacje w sieciach telekomunikacyjnych; sieci sygnalizacyjne tworzą autonomiczne logiczne warstwy sieci z komutacją pakietów.

5 Tendencje rozwoju systemów komutacyjnych

Centrala komutacyjna realizuje trzy podstawowe funkcje. Pierwszą funkcją jest tworzenie dróg dla przepływu informacji pomiędzy zainteresowanymi użytkownikami sieci. Użytkownikami sieci na poziomie logicznym są informacje przesyłane w kanałach, a mianowicie: 1) informacje wymieniane pomiędzy użytkownikami dla których tworzone są drogi przepływu przez węzły i gałęzie sieci; 2) informacje sygnalizacyjne między użytkownikiem i węzłem oraz między węzłami.

Drugą funkcją centrali jest obsługa sygnalizacji. Sygnalizacja jest językiem porozumiewania się pomiędzy użytkownikiem (terminalem) i węzłem oraz między węz-

łami dla wypełniania funkcji komutacyjnych, a także dla wspomagania funkcji utrzymaniowych i administracyjnych w sieci. Na podstawie przetworzonych wiadomości sygnalizacyjnych odbywa się sterowanie procesami komutacyjnymi.

Trzecią grupą funkcji sterowania są procesy utrzymaniowe i administracyjne.

Nowoczesne centrale komutacyjne budowane są z elementów elektronicznych podobnych do używanych w profesjonalnym sprzęcie elektronicznym - w aparaturze pomiarowej, w komputerach. Niezawodność central na poziomie poszczególnych układów (pakietów) jest z tym sprzętem porównywalna. Wymaganie, aby centrala komutacyjna pracowała nieprzerwanie, całą dobę i obsługiwała ruch w dowolnym momencie powoduje szczególne ostre wymagania na niezawodność obsługi, znacznie większą od niezawodności poszczególnych zespołów. Dla uzyskania odpowiedniego wskaźnika niezawodności obsługi stosuje się specyficzne metody projektowania układów, bloków funkcjonalnych i projektowanie całego systemu, ogólnie polegające na nadmiarowości stosowanych zespołów funkcjonalnych, automatycznej, szybkiej diagnostyce, automatycznym zastępowaniu w pracy układów uszkodzonych i natychmiastowym alarmowaniu o niesprawności tak bloków funkcjonalnych, jak i całej centrali. Dalej omówione zostaną niektóre cechy współczesnych systemów komutacyjnych.

Funkcje zespołów sterujących

Funkcje zespołów sterujących można podzielić na następujące główne grupy:

- obsługa zgłoszeń, wymuszającą wykonywanie dużej liczby zadań w czasie rzeczywistym;
- autodiagnostyka (kontrola wykonywania zadań); zespoły sterujące, składające się z wielu procesorów oraz układów komunikacji ze swoimi peryferiami są urządzeniami złożonymi, działającymi pod wielkim i skomplikowanym oprogramowaniem; muszą więc być wyposażone w mechanizmy testujące poprawność pracy, wykrywające przemijające błędy wykonania, uszkodzenia i eliminujące z pracy urządzenia uszkodzone oraz posiadać mechanizmy kontrolujące poprawność wykonywania programów;
- zadania utrzymaniowe związane również z elementami zewnętrznymi w stosunku do zespołu sterującego; np. liniami abonenckimi, łączami centralowymi oraz innymi zespołami central.
- realizacja zadań administracyjnych i technicznych, wykonywanych permanentnie, a więc aktualizację danych o abonentach i o łączach, modyfikacje dróg połączeń, pomiary ruchu i inne.

Poszczególne zadania stawiają odmienne wymagania dla zespołów sterujących [12];[13]. Obsługa zgłoszeń jest zawarta w około 40% oprogramowania, ale absorbuje moc przetwarzania w około 90%. Największą część oprogramowania, związaną z zada-

niami utrzymaniowymi i administracyjnymi (prawie 60%) w niewielkim stopniu zajmuje zespoły sterujące w czasie rzeczywistym. Dane dotyczące obciążenia w czasie rzeczywistym zespołów sterujących są różne dla różnych systemów, ale proporcje między grupami zadań są podobne [14].

Wybór architektury urządzeń sterujących jest wynikiem kompromisu między kosztem sprzętu i oprogramowania a wymaganiami wynikającymi z pracy w czasie rzeczywistym. Trzeba rozróżnić pojęcia struktur fizycznych (zbiór procesorów, pamięci, system połączeń) i struktur logicznych (regulujących wymianę między elementami zespołu na poziomie zadania, sygnału, procesu, przydzielanie zasobów, wykorzystanie systemu).

Podział funkcjonalny precyzuje miejsce urządzenia w strukturach fizycznych. Struktury fizyczne mogą być jedno- lub wielo- procesorowe. Z punktu widzenia obsługi procesów komutacyjnych klasyfikacji dokonuje się ze względu na sposób organizacji logicznej:

- architektura scentralizowana;
- architektura zdecentralizowana;
- architektura rozproszona.

O cechach sterowania nie decyduje fizyczne rozproszenie procesorów lub komputerów lecz rozproszenie funkcji logicznych, tzn.

1) rozproszenie obsługiwanego ruchu

lub 2) rozproszenie funkcji obsługi zgłoszeń.

W strukturze wieloprocesorowej z podziałem ruchu wyróżnić trzeba dwa rodzaje rozproszenia:

1) podział źródeł ruchu

oraz 2) podział obciążenia zadaniami obsługi ruchu.

W praktycznych rozwiązaniach nie ma czystych form strukturalnych, ponieważ konieczność uwzględnienia parametrów niezawodnościowych czyni architekturę bardziej złożoną.

5.1 Niezawodność sterowania

Avizienis [15] wprowadził pojęcie systemu tolerującego błędy jako takiego, który ma wbudowane mechanizmy (sprzętowe i/lub programowe) pozwalające - bez pomocy z zewnątrz - na ciągłość bezbłędnego działania programów i operacji wejścia/wyjścia w obecności pewnego zbioru błędów. Wyróżnia się błędy trwałe (np. uszkodzenia elementów) oraz przemijające (np. chwilowe zakłócenie). Łagodniejszym warunkiem jest częściowe tolerowanie błędów: jest to system, który dla pewnego zbioru błędów redukuje swoją zdolność przetwarzania poprzez zmniejszenie szybkości lub ograniczenie predefiniowanego zbioru wykonywanych funkcji. Wskaźniki nie-

zawodnościowe systemu uzyskuje się przez nadmiar sprzętu i/lub mechanizmy utrzymaniowe. Mechanizmy utrzymaniowe są oparte na dodaniu dodatkowych układów, specjalnych programów i/lub na wykorzystaniu kryterium czasu (time-out, powtarzanie operacji).

W teorii niezawodności systemów wielokomputerowych klasyfikuje się cztery rodzaje redundancji:

a) systemy z głosowaniem;

b) systemy z zimną rezerwą;

c) systemy z gorącą rezerwą (hybrydowe);

d) systemy ze stopniową degradacją.

W systemach typu a) wszystkie zespoły wykonują te same operacje; przez porównanie wyników i głosowanie eliminuje się zespoły uszkodzone. W grupie b) rezerwowy zespół zastępuje zespół główny po detekcji błędu z pewną stratą obsługi w okresie przejściowym. W układach hybrydowych c) zakłada się brak straty w okresie przełączania. Systemy typu d) są wielomaszynowymi zespołami z podziałem obciążenia; każdy z nich może wykonywać te same funkcje. Wykrycie błędu eliminuje uszkodzony zespół, degradując zdolność przetwarzania systemu.

5.2 Komunikacja międzykomputerowa

Ze względu na potrzebną moc obliczeniową oraz ze względów niezawodnościowych, zespoły sterujące central są układami wieloprocesorowymi i wielomaszynowymi. Wybór metody komunikacji pomiędzy procesorami, wchodzącymi w skład zespołu sterującego, stanowi jedną z kluczowych decyzji projektowych. Szczególnych cech nabiera komunikacja wewnętrzna, jeżeli za sterowanie centralą (obsługę zgłoszeń) odpowiedzialne jest wiele organów decyzyjnych, co jest charakterystyczną własnością współcześnie projektowanych systemów komutacyjnych. Wśród rozmaitych układów pracy wielokomputerowej w sterowaniu central można wyróżnić dwie grupy: systemy wieloprocesorowe i systemy wielomaszynowe. Przydział do odpowiedniej grupy zależy od stopnia sprzężenia pomiędzy procesorami wchodzącymi w skład systemu. W systemach wieloprocesorowych sprzężenie to jest na poziomie wymienianych sygnałów sterujących. W systemach wielomaszynowych wymiana następuje na poziomie wiadomości. W systemach tych, spotykanych w centralach SPC, można wydzielić trzy metody wzajemnego dostępu procesorów:

a) kanały bezpośrednie;

b) komutator międzyprocesorowy;

c) kanał z podziałem czasu.

W metodzie a) zwykle występuje podział procesorów na grupy według pełnionych funkcji lub według hierarchii pełnionych zadań. Podział według funkcji i związane z nim drogi wymiany wiadomości między procesorami ma miejsce w centralach E-10 [16]. Do komunikacji między procesorami różnych grup zorganizowane są tu tzw. telestrady, grupujące kanały czasowe z przydziałem na zasadzie rezerwacji dostępu.

W systemach typu b) występuje układ komutacyjny, czyli wydzielone urządzenie, którego zadaniem jest pośrednictwo w wymianie danych pomiędzy procesorami. Można tu wyróżnić systemy z komutacją wiadomości oraz systemy z komutacją kanałów. W centralach 5ESS [29] procesory sterujące autonomicznymi modułami centrali komunikują się poprzez węzeł komutacji pakietów sygnalizacyjnych, a transmisja odbywa się z wykorzystaniem łączy światłowodowych. W centralach ITT 1240 o rozproszonym sterowaniu, kanały dla komunikacji między procesorami są komutowane przez pole komutacyjne [17].

Systemy typu c) wykorzystują zasady pracy sieci lokalnych LAN. W centrali E10B jest wykorzystywana sieć o strukturze magistrali natomiast w centrali DX2000 [18] wykorzystuje się strukturę pętlową.

5.3 Przeciążenia

Zespoły central elektronicznych projektowane są na średni ruch w godzinie największego ruchu GNR. Nawet tak pozornie proste kryterium jak ruch średni jest trudne do sprecyzowania. Z przypadkowej natury zgłoszeń wynika, że przeciążenia w systemie są możliwe, a czasami nieuniknione. Obok parametrów poprzednio omawianych, bardzo ważnym parametrem systemu sterowania jest jego zachowanie w warunkach przeciążenia. W systemach nie mających mechanizmów ochrony przed przeciążeniem, obserwacja ruchu obsłużonego pokazuje, że zaczyna on maleć przy wzroście ruchu oferowanego powyżej pewnej wartości, jak to pokazano na rys.9. Możliwe są nawet zachowania katastroficzne - skokowe zmiany stanu lub histereza [19];[20]. Po ustąpieniu przyczyn przeciążenia ruch oferowany musi zmaleć znacznie poniżej poziomu nominalnego, aby system wrócił do stanu równowagi.

Główną przyczyną katastroficznych zachowań centrali w okresach natłoku jest angażowanie sterowania na rzecz obsługi wywołań, które (w ostatecznym rozrachunku) nie zakończą się połączeniem. W zakresie większych obciążeń istotną rolę zaczynają odgrywać straty wynikające z niemożliwości przewidzenia przyczyn leżących poza centralą (zajętość abonenta wywoływanego, straty w centralach tranzytowych, rezygnacja abonenta wywołującego i in.). Zabiegi, mające na celu minimalizację strat w zakresie obciążeń normalnych (np. alternatywne kierowanie ruchem w sieci, dopuszczenie oczekiwania abonenta na zwolnienie potrzebnego zasobu) obniżają odporność systemu na przeciążenie [21]. Do panowania nad tymi

zjawiskami systemy sterowania central muszą mieć dwa mechanizmy:

- wykrywanie groźby natłoku;
- przeciwdziałanie przeciążeniom.

Metody detekcji polegają na analizie stanu zasobów systemu sterowania, a przede wszystkim na analizie stopnia wykorzystania mocy obliczeniowej procesorów oraz na analizie długości kolejek zadań. W praktyce wykorzystuje się zwykle jednocześnie wiele metod detekcji przeciążenia, możliwych do implementacji w danej strukturze sterowania.

Zabezpieczenie przed natłokiem wymaga określenia celów akcji prewencyjnych. Ponieważ natłok jest skutkiem zaangażowania zasobów centrali w obsługę wywołań nie prowadzących do stanu rozmowy, więc podstawą wszystkich strategii zabezpieczenia jest wprowadzanie preferencji dla wywołań efektywnych. Pojawia się tu jednak trudność określenia, które z wywołań spełnią tak określone kryterium. Nawet w warunkach normalnego ruchu tylko około 50% do 70% wywołań dochodzi do fazy połączenia (rozmowy), pozostałe są tracone z przyczyn niezależnych od centrali.

5.4 Zarządzania pamięcią operacyjną w centralach

Współczesne systemy komutacyjne o sterowaniu programowym (SPC - Stored Program Control) stanowią w istocie szczególny rodzaj systemów komputerowych. Oprogramowanie systemów komutacyjnych ma jednak wiele cech różniących je od typowych systemów wieloprogramowych i systemów czasu rzeczywistego (np. systemów sterowania procesami technologicznymi).
W systemach takich, ze względu na wysoki stopień współbieżności, ważną rolę odgrywa zarządzanie pamięcią procesora (procesorów).

Oprogramowanie central SPC ma pewne specyficzne właściwości, które muszą być wzięte pod uwagę przy wyborze odpowiedniej metody zarządzania pamięcią [22]:

1. Wymagany bardzo krótki czas reakcji systemu. Jeżeli przekroczona zostanie pewna wartość czasu reakcji, to w rezultacie wystąpi strata (w rozumieniu obsługi wywołań) lub/i błąd. Pojawiają się problemy, które nie mogą być rozwiązane przy użyciu jedynie mechanizmów sprzętowych, gdyż:

a) istnieje duża liczba równocześnie aktywnych zewnętrznych źródeł informacji;
b) systemy SPC pracują z reguły w reżimie, w którym priorytety wiążą się z typami sygnałów a nie ich źródłami (jak to ma miejsce w typowych systemach czasu rzeczywistego);
c) wykorzystanie pamięci masowych ograniczone jest do zadań drugoplanowych (utrzymaniowych, administracyjnych itp.) ze względów efektywnościowych.

2. Eksploatowane oprogramowanie jest stosunkowo stabilne, a więc z góry znane są takie charakterystyki programów jak wielkość, czas wykonania itd.
3. Funkcje systemu operacyjnego są przejmowane przez system wykonawczy zaimplementowanego języka. Niektóre specjalizowane języki programowania - w szczególności CHILL [23][24] - nadają się do opracowywania całości oprogramowania systemu komutacyjnego. Wymaga to jednak opracowania wyrafinowanego systemu wykonawczego do zarządzania pamięcią, obsługi wejścia/wyjścia, współbieżności itd. Konieczne jest więc połączenie systemu operacyjnego i systemu wykonawczego języka w jedną całość, co w komputerach ogólnego zastosowania nie jest stosowane.

Efektywność systemu operacyjnego (OS) jest jednym z kluczowych czynników, wpływających na jakość obsługi ruchu przez centralę SPC [25][26]. Gospodarka zasobami (w szczególności pamięcią operacyjną) dzielonymi pomiędzy procesy jest jednym z głównych zadań systemu operacyjnego.

Przy wyborze mętody gospodarki pamięcią należy wziąć pod uwagę właściwości oprogramowania komutacyjnego (pomijamy tu procesy drugoplanowe), a w szczególności :

- niewielka liczba typów procesów;
- duża liczba (setki, tysiące) instancji (wcieleń) procesów;
- przydział pamięci dotyczy raczej obszarów danych niż obszarów kodu programu;
- typowy proces składa się z wielu (do kilkudziesięciu) krótkich zadań, aktywowanych przez zdarzenia zewnętrzne lub sygnały międzyprocesowe;
- programy są pisane w języku o orientacji proceduralnej (CHILL).

Tak więc w pamięci operacyjnej znajduje się duża liczba dynamicznych obiektów (procesy, procedury, bloki) o niewielkich rozmiarach i krótkim czasie życia.

Z telekomunikacyjnego punktu widzenia najistotniejsze są te własności OS, które wpływają na sprawność ruchową systemu (w szczególności - w odniesieniu do godziny największego ruchu). W dotychczasowych pracach dotyczących strategii zarządzania pamięcią koncentrowano się raczej na przypadkach deterministycznych jak również pomijano wzajemne powiązania obiektów umieszczanych w pamięci. Wiąże się to prawdopodobnie - oprócz badania pod kątem raczej pozatelekomunikacyjnych zastosowań - z napotykanymi znacznymi trudnościami, zmuszającymi do rezygnacji z metod analitycznych na rzecz metod symulacyjnych.

6 Zagadnienia oprogramowania systemów telekomunikacyjnych

Komputeryzacja i informatyzacja systemów telekomunikacyjnych stworzyła możliwość zastąpienia większości funkcji realizowanych dotychczas sprzętowo przez oprogramowanie. Systemy telekomunikacyjne są ze swej natury systemami rozproszonymi, natomiast informatyka jako dziedzina zajmowała się przez długi czas głównie systemami o sterowaniu scentralizowanym.

Dobór odpowiedniego języka programowania oraz odpowiedniej metodologii projektowania i programowania do zastosowań telekomunikacyjnych mają bardzo duże znaczenie. Zaawansowane metody projektowania i implementacji wymagają wsparcia przez odpowiednie środki nazywane środowiskami.

Termin *oprogramowanie telekomunikacyjne* oznacza programy sterujące pracą systemów telekomunikacyjnych a ściślej mówiąc pracą *węzłów sieci telekomunikacyjnej*. Zazwyczaj oprogramowanie telekomunikacyjne jest tworzone na jednym a eksploatowane na innym rodzaju komputerów. Wynika to z odmiennych wymagań jakie stawia się komputerom służącym do tworzenia (development), oprogramowania oraz komputerom stosowanym do sterowania systemami telekomunikacyjnymi. Pierwsze z nich określane są jako komputery goszczące (host computers), a drugie jako komputery docelowe (target computers).

Podstawowym środkiem służącym do tworzenia oprogramowania jest język programowania. Obecnie stosowane są głównie języki wysokiego poziomu choć wiele istniejących systemów zawiera w dalszym ciągu znaczne fragmenty oprogramowania napisane w języku asemblera. Programy w języku wysokiego poziomu tłumaczone są na język niskiego poziomu przy pomocy kompilatora, który oprócz funkcji translacyjnych bada również poprawność syntaktyczną i semantyczną programu oraz optymalizuje wynikowy kod programu. W praktyce jakość kodu generowanego przez kompilator może mieć znaczny wpływ na efektywność całego oprogramowania. Kompilacja programów odbywa się na komputerze goszczącym.

W przypadku tworzenia dużych systemów niezbędnym uzupełnieniem języka jest jego środowisko (environment), które wspomaga proces tworzenia oprogramowania na komputerze goszczącym. Zasadniczymi elementami środowiska są kompilator, edytor tekstów, linker i debugger. Środowisko musi zapewniać także środki do utrzymania i pielęgnacji oprogramowania oraz mechanizmy wspomagające pracę wieloosobowych zespołów programistów. W praktyce oznacza to potrzebę utrzymywania baz danych zawierających fragmenty i wersje tworzonych programów.

W wielu przypadkach środowisko korzysta bezpośrednio z uniwersalnych narzędzi dostarczanych przez system operacyjny maszyny goszczącej. Przykładem takiego środowiska jest system UNIX, który sam w sobie stanowi środowisko. Współczesne duże języki programowania, jak Ada lub CHILL [27], wymagają jednak własnych, niekiedy bardzo rozbudowanych, środowisk. Dzieje się to głównie z powodu występowania w nich mechanizmów modularyzacji i niezależnej kompilacji. Kompilatory tych języków muszą być wspomagane przez dodatkowe narzędzia wyszukujące odpowiednie fragmenty (moduły) programu w bibliotekach i analizujące zależności pomiędzy nimi. W przypadku dużych języków programowania granica pomiędzy mechanizmami środowiska a machanizmami samego języka jest bardzo płynna.

Wykonanie programów na komputerze docelowym kontrolowane jest zazwyczaj przez *system operacyjny*, który realizuje podstawowe funkcje, jak przydział czasu procesora, gospodarka pamięcią, obsługa dostępu do urządzeń zewnętrznych itp. Dla wykonania swoich zadań programy korzystają więc z elementarnych mechanizmów dostarczanych przez system operacyjny. Ze względu na systemowy charakter samego oprogramowania telekomunikacyjnego system operacyjny nazywa się często jądrem systemu (kernel).

W przypadku języków programowania o znacznym stopniu złożoności, *system wykonawczy* tych języków może stanowić jednocześnie system operacyjnym komputera docelowego. Przykładem takiego języka może być CHILL posiadający oprócz "normalnych" konstrukcji językowych mechanizmy współbieżności, komunikacji czy obsługi wyjątków. Mechanizmy te pozwalają na tworzenie w języku CHILL kompleksowych systemów oprogramowania telekomunikacyjnego bez konieczności wspomagania przez mechanizmy konkretnego systemu operacyjnego. Ceną jaką się za to płaci jest znaczny koszt implementacji, tzn. budowy kompilatora oraz systemu wykonawczego takiego języka.

Specyficzne cechy systemów telekomunikacyjnych stawiają szereg wymagań, do których powinny być dostosowane narzędzia oraz metodologia projektowania i tworzenia oprogramowania. Poniżej wymienione zostały podstawowe cechy oprogramowania systemów telekomunikacyjnych:

1. Niezawodność
2. Czas rzeczywisty
3. Wielodostęp
4. Rozproszenie
5. Duże rozmiary
6. Długowieczność i przenoszalność.

7 Komutacja szerokopasmowa

Tendencje w nowych usługach szerokopasmowych są źródłem krytyki koncepcji sieci ISDN. Trzy podstawowe cechy komutacji łączy w ISDN: zunifikowany kanał o przepływności 64 kbit/s, komutacja kanałów synchronicznych z zegarem pola komutacyjnego oraz symetryczna przepustowość w obu kierunkach stanowią utrudnienie dla integracji wszystkich usług bądź prowadzą do rozwiązań dalekich od optymalnych. Stąd, pojawiają się koncepcje, dotyczące sieci BISDN*). Mimo braku standaryzacji w tym zakresie, można zauważyć podstawowe trendy przy analizach sieci BISDN:

1. Wykorzystanie technologii fotonicznej w tym światłowodowej techniki transmisyjnej i optyki zintegrowanej w komutacji.
2. Nowe podejście do szybkiej komutacji pakietów. Podstawowym kryterium analizy i projektowania systemów z komutacją pakietów w latach 70-tych była optymalizacja wykorzystania drogich, wąskopasmowych kanałów transmisyjnych przy założeniu dopuszczalnego czasu dostarczenia pakietu. Wyniki prac podstawowych w systemach pakietowych oraz rozwój szerokopasmowych systemów transmisyjnych zmieniły kryteria dotyczące optymalizacji zasobów sieciowych. Z punktu widzenia systemowego, komutacja pakietów ma pewną przewagę nad komutacją kanałów.

Z istoty sterowania przepływem pakietów w sieci wynika:

- asynchroniczność napływu i transferu informacji,
- automatyczne zajmowanie zasobów transmisyjnych stosownie do liczby pakietów, a więc spełnienie warunku asymetrii przepustowości kanałów w obu kierunkach połączenia.

Wydaje się, że dotychczasowy podział na komutację kanałów i komutację pakietów będzie zanikał. Przy nowych możliwościach transmisyjnych i komutacyjnych, kryterium dopuszczalnego opóźnienia wiadomości przez sieć będzie łatwiejsze do spełnienia; komutacja kanałów i komutacja pakietów będą ewoluować do komutacji jednokierunkowych łączy wirtualnych.

Pojawiło się kilka ważnych stymulatorów dla podjęcia prac nad generacją fotonicznych systemów komutacyjnych [28]. *Jeżeli sygnały telekomunikacyjne będą transmitowane przy pomocy fotonów, czy nie byłoby prościej i taniej dokonywać operacji komutacji, marszrutyzacji w sieci i przetwarzania sygnałów również przy pomocy fotonów ?*

---

*) BISDN - Broadband Integrated Services Digital Network

7.1 Cechy układów fotonicznych

Z systemowego punktu widzenia, interesujące są następujące cechy technologii fotonicznej:

Szybkość przełączania. Dla przełączania układu z jednego stanu w drugi potrzebna jest energia przełączenia. Moc wymagana dla przełączenia jest stosunkiem energii do czasu. Wymaganie dużej szybkości przełączania (krótkie czasy) wymusza stosowanie dużych mocy. Fizyczne ograniczenia układów fotonicznych wynikają z mechaniki kwantowej i zjawisk termicznych w materiałach.

Przetwarzanie równoległe Szybkość działania systemu można zwiększyć metodą przetwarzania równoległego zamiast szeregowego. Układy optyczne są dobrze dostosowane do takiego przetwarzania. Typowy system obrazowy posiada rozdzielczość rzędu 1 μm w polu 1 mm co pozwala na jednoczesny dostęp do $100 \times 100 = 10^4$ pikseli. Maksymalna liczba pikseli przetwarzana przez układ optyczny jest stopniem swobody systemu (SBWP - space-bandwidth product). Systemy satelitarne osiągają SBWP=10^8 pikseli, natomiast systemy holograficzne powyżej 10^{10}. Dalsze zrównoleglenie przetwarzania można uzyskać metodą hybrydową: elektroniczno-optyczną, drogą modulacji elektronicznej poszczególnych sygnałów optycznych.

Komunikacja między elementami W systemowych rozwiązaniach komutacji fotonicznej lub w komputerach, trzeba zastosować duże ilości układów przełączających. Dla sprawnej pracy systemu potrzebna jest szybka komunikacja pomiędzy układami. Możliwości komunikacji sygnałów przy użyciu układów fotonicznych sięgają 100 Gb/s i są znacznie większe od technologii półprzewodnikowej.

Z wymienionych trzech cech fotonicznych układów przełączających wynika, że przy podobnych szybkościach przełączania jak w układach elektronicznych, należy dążyć do efektywnego wykorzystania dwóch pozostałych cech: równoległości przetwarzania i szybkości komunikacji. W rezultacie, wynikowa szybkość pracy systemów fotonicznych może być wyższa od półprzewodnikowych.

Znaczna różnica pomiędzy szybkością przenoszenia sygnałów przez matryce optyczne i szybkością ich przełączania, powoduje pewne ograniczenia w ich zastosowaniu jako stopni przestrzennych typu S dla zwielokrotnionych czasowo strumieni optycznych. Dla wykorzystania możliwości transmisyjnych, potrzebna jest szybka komutacja na poziomie kanału lub pakietu w strumieniach wielokrotnych. Dlatego też, dużo uwagi poświęca się komutacji czasowej na zwielokrotnionych strumieniach

cyfrowych w trybie TDM (time division multiplexing), ATM (Asynchronous Transfer Mode) [29][30] lub w trybie pakietowym. Wymagane tu są bardzo krótkie czasy przełączania porównywalne z czasem trwania jednego lub niewielu bitów. Niezbędny jest ciągły dostęp układów sterujących do strumienia cyfrowego. Istnieją dwie drogi rozwiązywania tego problemu.

1. Sterowanie elektroniczne komutatorem. Zanika w tym przypadku pełna przezroczystość przepływu wiadomości; dane muszą być przesyłane w ustalonych blokach, których nagłówki niosą wiadomości wykorzystywane przez sterowanie elektroniczne.
2. Sterowanie optyczne wykorzystujące technologię i metody komputerów fotonicznych. Napływające bloki danych traktowane są jako logiczne jednostki do obróbki w logice komputerowej i kierowane do następnych komutatorów.

Oba powyższe rozwiązania naśladują (zastępują) logiczną strukturę obecnych systemów komutacyjnych z wyraźnym rozdzieleniem zespołów dla transmisji wiadomości (pola komutacyjne) i dla obróbki sygnalizacji (lub nagłówków) i sterowaniem przepływem przez pola.

Inne podejście polega na zmianie architektury systemu komutacyjnego na hierarchiczną strukturę pętlową, składająca się z wielu sieci typu LAN. Pozwala to na wykorzystanie szybkości światłowodów w procesie transmisji. Układy między pętlami mogą być elektroniczne dzięki dużym pamięciom buforowym lub - w miarę postępów technologii - mogą być zastosowane pamięci optyczne. Ze względu na procedury rywalizacji o dostęp do pętli, rozważana jest w takich sieciach tzw. szybka komutacja pakietów [31].

7.2 Optyczne układy logiczne

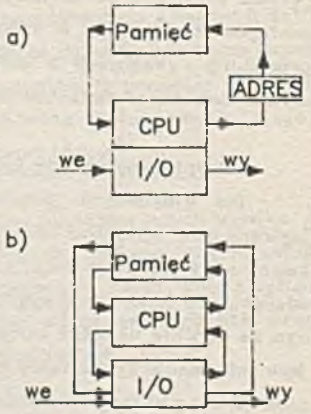
Dla sterowania procesami komutacyjnymi niezbędne są optyczne układy logiczne oraz optyczna komunikacja między nimi. Prowadzone są prace nad układami kombinacyjnymi i sekwencyjnymi. W systemach komutacji łączy niezbędne są układy odzyskujące informacje sygnalizacyjne zawarte w wielokrotnym strumieniu podawanym na wejście układów optycznych a następnie po szybkim przetworzeniu, wplatające informacje sygnalizacyjne do strumieni wyjściowych. W systemach pakietowych, niezbędne są układy odzyskiwania, przetwarzania i wtrącania nagłówków pakietów.

Oprócz elementarnych układów logicznych optyki zintegrowanej, budowa procesorów fotonicznych wymaga rozwiązania problemu komunikacji między układami i systemami. Podobnie jak w technologii elektronicznej, szybkość działania systemów komputerowych zależy od szybkości komunikacji na trzech poziomach sprzętowych:

- połączenia wewnątrz struktur scalonych układów optycznych,
- połączenia między strukturami w zespołach funkcjonalnych,
- połączenia między zespołami w systemie.

Zwiększenie skali integracji zmniejsza liczbę połączeń w systemie na wyższych poziomach i przyczynia się do zwiększenia szybkości działania. Dla zwiększenia szybkości działania układów półprzewodnikowych wielkiej skali integracji, wykorzystuje się komunikację fotoniczną pomiędzy częściami jednego układu scalonego [32]. Jednym z istotnych dylematów komunikacji jest wybór trybu pracy: szeregowej lub równoległej. Sprzężenie równoległe jest znacznie szybsze ale często geometryczne możliwości struktur nie pozwalają na jego zastosowanie. Przejście na sprzężenie szeregowe wymaga zastosowania multipleksacji i demultipleksacji, które obniżają graniczne szybkości pracy. Połączenia pomiędzy optycznymi układami logicznymi dokonywane mogą być za pomocą takich elementów optycznych jak soczewki, pryzmaty, lustra itp. lub przy wykorzystaniu holografii. Przy pomocy holografii można modyfikować i kierować fale świetlne; układ holograficzny może być uważany za bardzo szybkie łącze optyczne pomiędzy optycznymi układami logicznymi.

Nowe możliwości komunikacji między zespołami funkcjonalnymi mogą prowadzić do zmian w architekturze systemów. Na rys.2a) pokazano klasyczną strukturę komputera von Neumanna. Przetwarzanie informacji odbywa się w CPU, natomiast pamięć (program i dane) stanowi oddzielny zespół. Taka struktura wynika z możliwości komunikacji między zespołami wykonanymi w technologii elektronicznej. Rozwój układów elektronicznych cechuje się tym, że maleje koszt układów i rośnie relatywnie koszt komunikacji między układami. Komunikacja pomiędzy CPU i pamięcią jest wąskim gardłem struktury z rys.2a). Stąd, w nowoczesnych, szybkich systemach komputerowych istnieje tendencja do stosowania struktur macierzowych i wykorzystania mechanizmów pipe-linning i cashe-memory a więc tendencja do minimalizacji kosztów komunikacji za cenę liczby układów. Jako dobry przykład takiego rozwoju procesorów półprzewodnikowych może służyć mikroprocesor Intel i860 (N10) [33].

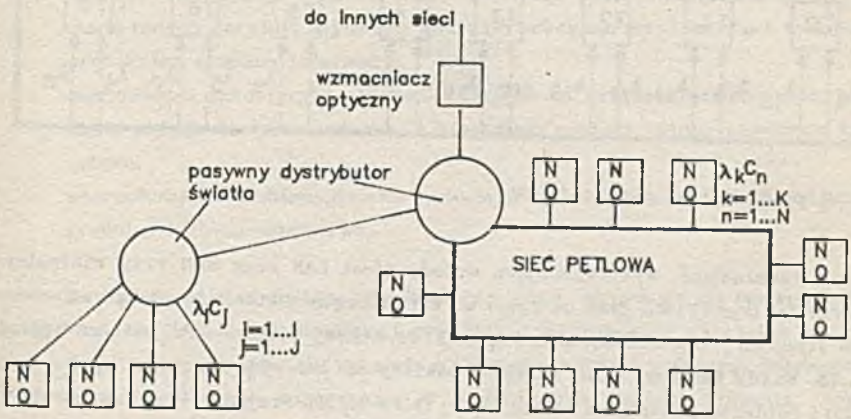


Rys.2. Dwie struktury komputerów: a) klasyczna b) wykorzystująca komunikację równoległą

Wykorzystanie mechanizmów komunikacji oferowanych przez optykę zintegrowaną

może prowadzić do architektury nie von Neumanna, jak na rys.2b). Zasoby wszystkich trzech podstawowych zespołów komputera mogą być dostępne dla siebie wzajemnie w sposób równoległy (na poziomie najmniejszej jednostki informacyjnej). Z kolei, taka zmiana architektury powodująca zwiększenie szybkości działania ma istotny wpływ na metody oprogramowania. W szczególności, rozwijające się systemy ekspertowe wymagają bardzo dużych mocy przetwarzania.

7.3 Ewolucja telekomunikacyjnych sieci lokalnych

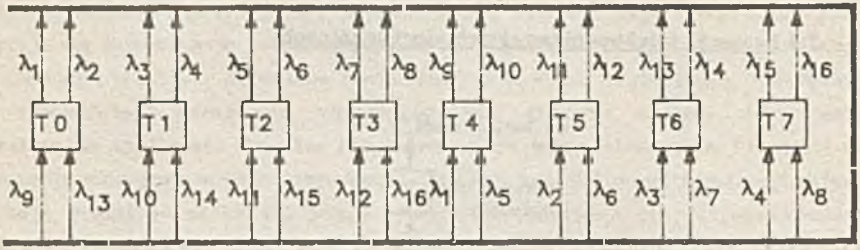


Rys.3 Sieć lokalna wykorzystująca zwielokrotnienie falowe i przestrajalne źródła światła

Na rys.3 pokazano hipotetyczne rozwiązanie sieci lokalnej w technice fotonicznej. Srodowisko transmisyjne w tej sieci można traktować podobnie do środowiska fal radiowych. Sieci takie nazywa się pasywnymi, ponieważ - oprócz światłowodów - zawierają pasywne dystrybutory światła. Dla wyróżnienia kanałów można wykorzystać metodę zwielokrotnienia falowego WDM w połączeniu ze zwielokrotnieniem kodowym CDMA (code division multi access). Każdy odbiornik terminala jest identyfikowany dwuwymiarowo przez (λ_i, C_j), gdzie λ_i jest długością fali świetlnej a C_j - adresem kodowym. Do nawiązania połączenia, nadajnik terminala powinien dostroić się do fali λ_i i wysłać adres C_j. Szerokopasmowe trakty transmisyjne i dystrybutory światła doprowadzą wszystkie sygnały do każdego z odbiorników; sygnał będzie odebrany przez odbiornik wybrany. Jak widać, koncepcja ta może dotyczyć zarówno połączeń w trybie komutacji kanałów jak i pakietów. Problemy związane z unikaniem kolizji są podobne do spotykanych w

7.3 Sieci sortujące typu SN[♦)]

Koncepcja pasywnych sieci lokalnych z wykorzystaniem metod zwielokrotnienia falowego i kodowego (WDM+CDMA) jest perspektywiczna, możliwa do szerszego zastosowania przy łatwym dostępie przestrajalnych - w szerokim zakresie - półprzewodnikowych źródeł światła. Opiszemy przykładowe rozwiązanie komunikacji



Rys.4 Przykład sieci typu SN

między terminalami, wykorzystujące metody sieci LAN oraz WDM przy minimalnym zbiorze fal (rys.4) [36]. Sieć na rys.4 ma strukturę magistrali (mogą być stosowane inne struktury np. drzewa, pierścienia), do której dołączonych jest 8 terminali T1...T8. Każdy terminal może nadawać pakiety na jednej z dwóch fal λ_i, λ_j (i,j = 1...16) i odbierać pakiety na falach λ_k, λ_l (k,l = 1...16). Przypisanie fal nadawczych i odbiorczych do terminali jest trwałe dzięki czemu nie trzeba wyposażać terminali w drogie, przestrajalne elementy optyczne.

Terminal posiada mapę marszrutyzacji sieci. Nadajnik terminala wysyła pakiet z zakodowanym w nagłówku adresem odbiornika na jednej z dwóch fal. Wybór fali wynika z mapy marszrutyzacji. Relacje między zbiorem wszystkich fal i podzbiorem fal charakterystycznych dla terminala nadawczego i odbiorczego powodują, że transmisja pakietu może odbyć się bezpośrednio między zainteresowanymi terminalami lub poprzez tranzytowanie (przesiadki) przez inne terminale. Odbiornik, po odebraniu nagłówka pakietu (na podstawie adresu docelowego, zawartego w nagłówku) podejmuje jedną z dwóch decyzji:

- pakiet przeznaczony jest do własnego terminala,
- pakiet należy tranzytować.

♦) SN - Shuffle Net

sieciach LAN. Dla połączeń wychodzących poza sieć lokalną można wykorzystać zbiór fal λ_r i wzmacniacze optyczne. Do potencjalnych zalet takiej sieci należy zaliczyć:

- pasmo światłowodów jednomodowych pozwala na utworzenie zbioru kilkunastu tysięcy kanałów o przepustowości rzędu 10Mb/s na różnych falach, wykorzystanie ortogonalnego zbioru ciągów kodowych pozwala zwiększyć liczbę kanałów conajmniej o rząd wielkości,
- liczba terminali może powiększać się sukcesywnie bez istotnego wzrostu kosztów urządzeń sieciowych,
- szybkość transmisji w kanale dla określonego połączenia jest niezależna od stanu innych kanałów; szybkość jest uwarunkowana możliwościami terminali i protokołem wymiany informacji,
- oszczędności inwestycyjne i eksploatacyjne na urządzeniach komutacyjnych; znika pojęcie centrali końcowej w klasycznej postaci, istnieją centrale tranzytowe,
- uwarunkowania tłumiennościowe pozwalają na pokrycie taką siecią obszaru o średnicy kilkudziesięciu km.

Ze względu na stan dostępnych elementów, sieci takie mogą się rozpowszechnić za kilkanaście lat. Dla ekonomicznej realizacji, potrzebne są nadajniki laserowe, łatwo przestrajalne w szerokim zakresie o stosunkowo dużej mocy oraz dopracowane, wysokoczułe detektory koherentne. Z systemowego punktu widzenia, analizy wymagają protokoły wymiany informacji, metody unikania kolizji (jednoczesny dostęp do jednego odbiornika) oraz metody kontroli błędów. Wydaje się, że mimo postępów technologii, szybkość przestrajania laserów będzie znacznie niższa od szybkości transmisji, co preferuje stosowanie komutacji kanałów w stosunku do szybkiej komutacji pakietów.

7.4 Wpływ metod sieci LAN

Sieci LAN były pierwotnie przeznaczone do wymiany informacji pomiędzy różnymi terminalami rozmieszczonymi na ograniczonym obszarze o średnicy do kilku km. Ograniczenia odległości wynikały zarówno z potrzeb (np. systemy dedykowane jednego przedsiębiorstwa) jak i z zastosowanych torów przewodowych. Na poziomie logicznym, nie istnieje granica, rozdzielająca sieci LAN od innych sieci. Typową cechą LAN jest wykorzystywanie wspólnego dla wielu komunikujących się terminali ośrodka transmisyjnego - nośnika (medium). Stosowane są dwie podstawowe topologie: magistrala (bus) oraz pierścień (ring).

Istnieje pewna odrębność algorytmów, wykorzystywanych w LAN. Protokoły wymiany informacji stosowane w LAN wywodzą się z protokołów opracowanych już wcześniej dla potrzeb teleinformatyki. Dobór protokołu dostępu do toru ma związek z wykorzystywaną topologią sieci. W protokołach tych stosuje się przesyłanie znacznika (token) uprawniającego do transmisji. Metoda ta, wywodząca się z sieci pierścieniowych, jest wykorzystywana również w przypadku magistrali. W sieciach o strukturze magistrali wykorzystywanych jest wiele odmian rywalizacji o wspólny kanał. Rywalizacja o kanał stwarza możliwość występowania kolizji, powodujących konieczność powtórnej transmisji wiadomości (pakietu). Najstarsze metody tej klasy wywodzą się ze znanej sieci ALOHA. Udoskonaleniem zasady rywalizacji jest protokół CSMA/CD (Carrier Sense Multiple Access / Collision Detection), w którym częstość kolizji jest zminimalizowana dzięki badaniu stanu toru zarówno przed rozpoczęciem jak i podczas transmisji (m.in. w sieci ETHERNET).

Inna grupa protokołów polega na rezerwowaniu dostępu do magistrali. W sieciach pierścieniowych stosuje się często metody polegające na wprowadzeniu do pierścienia pewnej liczby szczelin czasowych obiegających pierścień; dane są przesyłane poprzez wprowadzenie ich do niezajętej szczeliny ("slotted ring"). Obok wymienionych, głównych klas metod dostępu stosowane są liczne ich odmiany; pojawiają się też metody nowe. Podejmowane są próby standaryzacji: norma IEEE 802.x (x określa konkretny protokół) określa podział protokołów LAN na warstwy oraz definiuje protokoły dla każdej z warstw (w warstwie dostępu do medium zdefiniowano dotychczas 4 różne protokoły) [34].

W ostatnich latach coraz częściej wykorzystuje się światłowody jako nośnik, ze względu na ich odporność na zakłócenia elektromagnetyczne i szerokie pasmo przenoszenia. Z kolei, parametry transmisyjne światłowodów (przede wszystkim - odległość międzyregeneratorowa) pozwoliły na zwiększenie średnicy sieci LAN do około 100km. Rozwijane są koncepcje sieci MAN (metropolitan area network).

Rozszerzenie zakresu stosowalności metod transmisyjnych i algorytmicznych sieci LAN przy uwzględnieniu elementów optyki zintegrowanej zmusza do nowego spojrzenia na metody komutacji. Weryfikowane są metody dostępu do nośnika. Zwiększenie szybkości transmisji powyżej 100Mb/s ujawnia wady dostępu CSMA/CD; zmniejszenie stosunku czasu transmisji pakietu do czasu propagacji w torze zwiększa prawdopodobieństwo kolizji. Powstają nowe standardy (FDDI - Fiber Distributed Data Interface [35]) dla dostępu do sieci pierścieniowych z wykorzystaniem znacznika.

Na przykład, transmisja pakietu od T0 do T4 może być przeprowadzona bezpośrednio na fali λ_1. Natomiast, transmisja pakietu od T0 do T1 odbywa się z przesiadką; pakiet jest transmitowany na fali λ_1 do T4, który po analizie nagłówka podejmuje decyzje wysłania go na fali λ_{10} do T1. W stanie dynamicznym, przy przypadkowym ruchu między terminalami sieci, istnieje możliwość kolizji - jednoczesnej transmisji wielu pakietów na tej samej fali. Dla uniknięcia skutków kolizji stosowane jest buforowanie [37].
Rys.4 jest szczególnym przykładem ogólnego typu sieci SN. Sieć SN zawiera $N=kp^k$ (k,p=1,2,3...) węzłów (terminali) i zorganizowana jest jako k kolumn po p^k węzłów w kolumnie (sieć na rys.4 posiada parametry: p=2, k=2, N=8). Sieci tego typu mają następujące cechy:

- struktura grafu jest symetryczna, homogeniczna - sieć wygląda tak samo z punktu widzenia każdego terminala,
- sieć ma dobrą strukturę niezawodnościową; np. przy awarii T4, pakiet od T0 do T1 może być tranzytowany przez T5, T3 i T6,
- liczba dróg alternatywnych rośnie z parametrem p,
- łatwa rozbudowywalność sieci; przy zaprojektowaniu docelowej pojemności $N=kp^k$, sieć można rozbudowywać po k węzłów bez zmiany poprzednio przydzielonych fal,
- możliwość zastosowania adresowania o cechach automatycznego wyboru drogi.

Sieci typu SN mogą być budowane również przy wykorzystaniu matryc przestrzennych wykorzystujących modulatory światła [7].

Stan obecny układów optyki zintegrowanej jest okresem intensywnych badań i eksperymentów, ale jest to początkowe stadium z systemowego punktu widzenia. Trudno przewidzieć, które rozwiązania staną się dominujące. Obok różnorodności w poszczególnych typach elementów optycznych, trudność polega na niezbędnej kompatybilności pomiędzy poszczególnymi typami w zakresie pracy w odpowiednim oknie optycznym, pracy z określoną szerokością pasma, łatwością sprzężenia między elementami, ujednoliceniem napięć zasilających itp. Tym niemniej, pola komutacyjne fotoniczne staną się rzeczywistością. Równolegle, obserwowany jest postęp w zakresie wykorzystania optyki zintegrowanej do zespołów sterujących central (komputery).

Te nowe możliwości technologiczne powodują ewolucję poglądów na przyszłą sieć zintegrowaną. Obok "wąskopasmowej" sieci ISDN z podstawowym kanałem cyfrowym o przepływności 64kb/s, rozważane są "szerokopasmowe" sieci BISDN z podstawowym kanałem 140Mb/s lub 565Mb/s. Kanał taki pozwala na transmisję

cyfrowego sygnału telewizyjnego o wysokiej jakości oraz grafiki komputerowej z dużą rozdzielczością. Szybkość działania pól optycznych przy wzrastającej szybkości sterowania i pojemności pamięci buforowych powoduje nowe podejście do dyskusji między komutacją kanałów i komutacją pakietów. Przy odpowiedniej szybkości systemów komutacyjnych można założyć bardzo małe opóźnienia w systemie komutacji pakietów, nieuciążliwe dla dowolnych usług telekomunikacyjnych. Przy takich założeniach, szybka komutacja pakietów ma pewną systemową przewagę nad komutacją kanałów.

BIBLIOGRAFIA

1 Bem D., Dąbrowski M., Zientalski M.: "Kierunki rozwoju i badań naukowych w telekomunikacji", Wyd. Pol. Wrocławskiej, seria Monografie, 1986

2 World Communications, Wyd. UIT, Genewa, 1985

3 J.S.Doyle, C.S.McMahon, "The Intelligent Network Concept", IEEE Trans. on Communications, No12, 1988.

4 Guest Editorial: Switching for Broadband Communications, IEEE JSAC, v.SAC-5, Nr8, 1987

5 Caballero P.A., Rios F.J., Casali F.: Digital Network Planning. Electrical Communication, Vol 59, No 1/2,1985

6 Sobczak W.: Statystyczna teoria systemów przesyłania informacji. WKŁ,1984

7 Zalecenia CCITT, Komisji Studiów XVIII, XI i VII

8 CCITT SG VII: Recommendation X.200: Reference Model of Open Systems Interconnection for CCITT Applications.

9 M.Dąbrowski, "Problemy sieci ISDN", w materiałach KST'87, Ref.A4.

10 P.Mermelstein, "G.722: A new CCITT Coding Standard for digital Transmission of Wideband Audio Signals", IEEE Com. Mag., No1, Jan.,1988

11 Migues M. Soto G.: Gestaltung eines Paketvermittlungsnetzes, Leistungsfahigkeit und zugehorige Einflussgrossen. Elektrisches Nachrichtenwesen. Nr1,1980

12 Proceedings of the Third Int. Conf. on Software Eng. for Telecommunications. Switching Systems, Helsinki, June, 1978.

13 McDonald J.C: Fundamentals of Digital Switching. Plenum Press, Nowy Jork 1983.

14 Hubuterne G: Ecoulement du trafic dans les autocomutateurs. Masson, Paryż, 1985.

15 Avisienis A: Fault-Tolerant Systems. IEEE Trans. on Computers, vol. C-25, 1976.

16 Błaszyk J. i inni: System komutacji elektronicznej E10. WKŁ, Warszawa, 1977.

17 Van Malderen R: System 12 Digital Exchange - Review of Fundamental Concepts. The Technical Journal of ITT, No 1/2, 1985.

18 Jarociński M., Dąbrowski M.: Centrala telefoniczno-teleinformatyczna SYNCOM -

DX2000. Materiały Krajowego sympozjum Telekomunikacji KST'86, część III, Rynia, wrzesień, 1986.

19 Farber N.: A Model for Estimating the Real-time Capacity of Certain Classes of Central Processors. ITC-6, Munich, 1970.

20 Jarociński M.: Katastrofy w sieci telekomunikacyjnej. II Konf. Naukowa: Systemy i sieci telekomunikacyjne, Warszawa, 1981.

21 Lemieux C.: Theory of Flow Control in Shared Networks and Its Application in the Canadian Telephone Network. IEEE Trans. on Communications, Vol. COM-29, No 4, 1981.

22 M.Dąbrowski, J.Jakubicki, Jarociński M.: "Speicherverwaltung in CHILL - Betriebssystem" 18. Fachkolloquium Informationstechnik, Dresden 1985.

23 CCITT: Recommendation Z.200 (CCITT High Level Language CHILL); Red Book, UIT, Geneva 1985.

24 Jarociński M., Sredniawa M.: "Język programowania CHILL"; WKŁ, Warszawa 1987 (w druku).

25 Dąbrowski M., Jakubicki J., Jarociński M.: "The Impact of Memory Management Strategy on CHILL Run-Time System Efficiency"; 4th CHILL Conference, Munich, Sept. 1986.

26 Williams M.C.: "The Efficiency of Executive Systems used in Telephony"; 5th SETSS, Lund 1983 (IEE Conf.Proc.223).

27 M. Jarociński, M. Średniawa - "Programowanie w języku CHILL", WKŁ 1987.

28 M.Dąbrowski, Komutacja Fotoniczna, Referat Plenarny A-5 w materiałach KST'89, Bydgoszcz, 1989

29 J.Gechter, P.O'Reilly, "Conceptual Issues for ATM", IEEE Network - The Mag. of Comp. Comm., No1, Jan., 1989

30 M.J.Rider, "Protocols for ATM Access Networks", IEEE Network - The Mag. of Comp. Comm., No1, Jan., 1989

31 A.S.Acampora, M.J.Karol, "An Overview of Lightwave Packet Networks", IEEE Network - The Mag. of Comp. Comm., No1, Jan., 1989

32 L.D. Hutcheson i inni, "Optical interconnects replace hardware", IEEE Spectrum, No3, March, 1987

33 T.S.Perry, "Intel's secret is out", IEEE Spectrum, No4, April1, 1989

34 Moustakas S.: The standardization of IEEE 802.3 compatible fiber optic CSMA/CD LAN - physical topologies. IEEE Comm. Mag. vol.25, No2,1987

35 D.Pitt, "Standards for the Token Ring", IEEE Network, No1, Jan., 1987

36 A.S.Acampora, M.J.Karol, "An Overview of Lightwave Packet Network", IEEE Network, No1, Jan., 1989

37 M.J.Karol i inni, "Input vs. Output Queueing on a Space Division Packet Switch", IEEE Trans. on Comm., No12, Dec, 1987

Jesienna Szkoła PTI

Mrągowo, listopad 1989

JAKOŚĆ OPROGRAMOWANIA

dr Jacek Irlik

Instytut Systemów Sterowania

ul.Armii Czerwonej 101

40-161 Katowice

1. Wstęp.

Zgodnie z potocznym znaczeniem jakość to "właściwość, rodzaj, gatunek, wartość; zespół cech stanowiących o tym, że dany przedmiot jest tym przedmiotem a nie innym; np.Dobra, zła jakość produkcji. Podnosić jakość wyrobów." (1)

Według terminologii EOQC (Europejska Organizacja d/s Sterowania Jakością) jakość to "stopień w jakim produkt zaspokaja wymagania użytkowników" (2).

Jakość produktów stanowi istotny przedmiot zainteresowania rozsądnego nabywcy.

Na ogół pragnie on, aby nabyty produkt:

- odpowiadał celowi i sposobowi zamierzonego użycia,
- działał we właściwy sposób,
- był bezpieczny w użyciu,
- był niezawodny,

i t.d.

Nabywca jest więc zainteresowany przede wszystkim gwarancją, że nabywany produkt funkcjonować będzie zgodnie z oczekiwaniami, których podstawą są najczęściej informacje na temat produktu udzielone przez producenta w postaci katalogów, folderów, instrukcji użytkowania itp.

Jakość produktów jest również przedmiotem zainteresowania ich producenta.

Rozsądny producent powinien dążyć do tego, aby produkt

- odpowiadał potrzebom, celom lub sposobom użycia,
- spełnił oczekiwania klientów,
- był wytworzony w sposób zgodny z właściwymi standardami lub uzgodnionymi z klientem specyfikacjami,
- nie naruszał w użytkowaniu zasad wynikających z ochrony zdrowia i życia ludzkiego, środowiska naturalnego i t.p.,
- mógł być oferowany po konkurencyjnej cenie,
- mógł być wytworzony po kosztach, które mogą zapewnić mu odpowiedni zysk.

Rozsądny producent uświadamia sobie, że w przypadku niezachowania należytej staranności w powyższym zakresie może ponieść straty w wyniku braku zaplanowanej sprzedaży lub nadmiernej liczby reklamacji.

Może też odpowiadać w przypadku, gdy sprzedany produkt złej jakości spowoduje zagrożenie lub szkody.

Rozsądny producent powinien przykładać należytą uwagę do zachowania jakości w całym cyklu produkcji po to, aby nie ponieść strat w wyniku zmarnotrawienia materiałów, konieczności powtórnego wykonania operacji i t.p.

W warunkach rynku konsumenta jakość produktów jest ważnym

czynnikiem wpływającym na pozycję producenta na rynku, a uzyskane atesty i znaki jakości mają wymierną wartość handlową.

Reasumując, w pewnym uproszczeniu można powiedzieć, że nabywcę interesuje głównie finalna jakość produktu.

Producent powinien być natomiast zainteresowany takim systemem zarządzania produkcją, który - zapewniając należytą jakość produktu - stanowić będzie o powodzeniu firmy.

Uświadomienie sobie, że jakość produktu to stopień, w jakim produkt zaspokaja wymagania użytkowników, nie pozwala na ograniczenie uwagi producenta jedynie do problemu jakości w samej produkcji. W ostatnich latach rozwinęła się koncepcja systemów "zarządzania przez jakość" (Quality Management Systems), które mają integrować takie sfery działań przedsiębiorstwa, jak:

a) sferę przedprodukcyjną, w ramach której dokonywane być powinny badania problemów i potrzeb użytkowników, analizy istniejących rozwiązań i ich przydatności, analizy marketingowe i t.p.,
b) sferę przygotowania produkcji, w ramach której sformułowane zostają koncepcje rozwiązań, stworzone projekty oraz zbudowane rozwiązania modelowe i t.p.,
c) sferę produkcji, w ramach której uruchomiona zostaje produkcja,
d) sferę obrótu towarowego, w ramach której następuje przygotowanie produktów do postaci umożliwiającej zakup i rozpoczęcie eksploatacji,
e) sferę poprodukcyjną, w ramach której utrzymywany jest kontakt z odbiorcami produktów, świadczone są usługi gwarancyjne i pogwarancyjne oraz zbierane informacje o stopniu zaspokajania potrzeb, jako sprzężenie zwrotne dla sfery przedprodukcyjnej - co umożliwia dobry rozwój.

Integracja powyższych sfer tworzy tzw. "pętlę jakości", dzięki której umożliwione zostaje dochodzenie, poprzez kolejne kroki rozwoju, do coraz lepszej zgodności pomiędzy ofertą producenta a potrzebami użytkowników.

Sfery przygotowania produkcji, produkcji oraz obrotu towarowego to sfery, które powinny być objęte systemem technicznej kontroli jakości kolejnych faz - od koncepcji do produktu finalnego.

Sfery przed i poprodukcyjne to sfery, które powinny być objęte właściwym systemem marketingu.

Całość integrowana jest systemem zarządzania przez jakość, którego istotą jest: właściwa struktura organizacyjna, układ odpowiedzialności, procedury postępowania, układ procesów oraz odpowiednie środki.

Rozważania dotyczące systemów zarządzania przez jakość, zwłaszcza tych ich elementów, które dotyczą technicznej kontroli jakości, abstrahują od określania konkretnych parametrów jakościowych, a także od wskazania konkretnych metod i zadań w toku kontroli.

Rozważania takie dotyczą wymagań w stosunku do organizacji systemu zarządzania, który pretenduje do miana systemu zarządzania przez jakość.

Konkretyzacja tych wymagań będzie tym większa im bardziej określony będzie rodzaj produktów, stanowiących przedmiot działania przedsiębiorstwa.

Pętla jakości jest szczególnie czytelna w przypadku takiego rodzaju produktu, jakim jest oprogramowanie. W modelowym ujęciu ewolucji programu (3) można znaleźć elementy poszczególnych, wymienionych powyżej pięciu sfer działalności, integrowanych w ramach systemu zarządzania przez jakość.

W sferach przygotowania produkcji, produkcji oraz obrotu towarowego realizowane będą np., określone w ramach modelowego ujęcia cyklu życia oprogramowania (4), procesy:

- koncepcji,
- specyfikacji wymagań,
- projektu,
- implementacji,
- testowania,
- instalacji,
- pielęgnacji.

Sfera przedprodukcyjna powinna obejmować:

- badanie i analizy problemów, istniejących rozwiązań, metod, narzędzi i t.p.,
- analizę informacji o stopniu satysfakcji użytkowników z rozwiązań wprowadzonych przez firmę do obrotu oraz sugestii co do pożądanych zmian,
- badania marketingowe.

Sfera poprodukcyjna powinna obejmować organizację serwisu wprowadzonego do obrotu oprogramowania oraz gromadzenie informacji niezbędnej dla sfery przedprodukcyjnej (propozycje nowych produktów), dla wprowadzenia niezbędnych zmian w sferach produkcyjnych (usunięcie wad,propozycje nowych wersji produktu).

Integracja wyżej wymienionych sfer znajduje w przypadku oprogramowania odzwierciedlenie w ujęciu wspomnianego modelu IEEE poprzez czynności, które podejmowane są w ramach procesów zarządzania (zwłaszcza procesów zarządzania jakością), a także tzw. procesów integracyjnych (zwłaszcza procesu weryfikacji i atestacji).

Graficzny schemat pętli jakości oprogramowania przedstawiono na rysunku. Procesy zarządzania obejmują tu proces inicjowania przedsięwzięcia, proces śledzenia i nadzoru nad przedsięwzięciem oraz proces zarządzania jakością. Procesy rozwojowe, to proces formułowania koncepcji, proces określania wymagań, proces projektowania, proces implementacji i testowania, proces instalowania, proces eksploatacji i zapewnienia serwisu. Procesy integracyjne to wreszcie proces zarządzania konfiguracją, proces weryfikacji i atestacji, proces rozwoju dokumentacji oraz proces szkolenia.

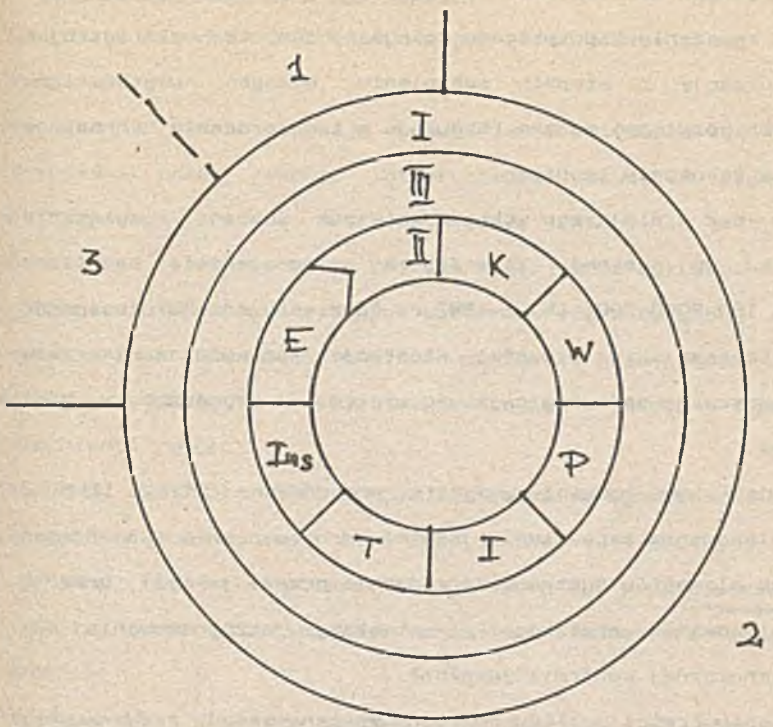
Model IEEE (4) określa zadania realizowane w toku wymienionych procesów oraz precyzuje niezbędną treść informacji przekazywanych pomiędzy procesami.

Wśród procesów rozwojowych znaczenie szczególne ma proces formułowania koncepcji. Podstawowymi zadaniami w tym procesie są:

- zidentyfikowanie potrzeb i pomysłów,
- sformułowanie potencjalnych rozwiązań,
- przeprowadzenie analizy wykonalności (feasibility studies),
- rozważenie możliwości przeniesienia (adaptacji istniejących rozwiązań),
- ostateczne sprecyzowanie potrzeb i pomysłów.

Informacje, w oparciu o które następuje sformułowanie koncepcji to m.in.:

- zmieniające się wymagania,
- żądania użytkowników oraz klientów,
- pomysły wynikające z toku rozwoju,
- informacje marketingowe,
- informacje o istniejących metodach i gotowych rozwiązaniach,



Rys. pętla jakości oprogramowania

1 - sfera przedprodukcyjna

2 - sfery produkcyjne

3 - sfera poprodukcyjna

I - procesy zarządzania

II - procesy rozwojowe

III - procesy integracyjne

- dane o własnych możliwościach i zasobach.

Szczególne znaczenie tego procesu polega na tym, że wykorzystując pełną informację o stopniu spełnienia wymagań użytkowników istniejących rozwiązań system formułuje w tym procesie koncepcję rozwiązań w założeniu lepszych.

Poniżej, w p.2 niniejszego wykładu omówione zostaną wymagania w stosunku do systemów zarządzania przez jakość, określone standardem ISO 9000-9004 (5) z 1987 r. Począwszy od 1987 r.agendy Wspólnoty Europejskiej prowadzą atestację systemów zarządzania przedsiębiorstwami software'owymi ze względu na zgodność z tymi wymaganiami.

W p.3 przedstawione zostaną wymagania proponowane przez IEEE w zakresie planowania zapewnienia jakości oprogramowania (6) (jeden z istotnych elementów systemów zarządzania przez jakość) oraz w zakresie planowania weryfikacji i atestacji oprogramowania (7) (system technicznej kontroli jakości).

W uzupełnieniu treści wykładu omówione zostaną zasady przyznawania produktom programowym w RFN znaku jakości. Przyznanie produktowi znaku jakości nie ma związku ze stwierdzeniem, czy produkt ma właściwie zorganizowany system zarządzania przez jakość. Stanowi jedynie potwierdzenie pewnego poziomu jakości produktu oferowanego w obrocie, o które producent może starać się niezależnie od tego, jak zorganizowana jest jego firma. Znak taki może mieć zastosowanie w szczególności w przypadkach , w których "manufakturalny" rozmiar przedsięwzięć software'owych stawia problem systemu zarządzania poza dyskusją.

2. Systemy zarządzania przez jakość.

2.1. Seria standardów ISO dotyczących jakości dzieli się na dwie grupy.

Standard ISO 9004 obejmuje wymagania w stosunku do systemów zarządzania przez jakość, integrujących wszystkie czynności w poszczególnych sferach, tworzących pętlę jakości i jest głównym przedmiotem zainteresowania podczas niniejszego wykładu.

W określeniu wymagań dla takich systemów nacisk położony jest na zapewnieniu przeświadczenia u producenta, że jego firma rozwijać będzie produkcję w sposób zapewniający jakość (wewnętrzne zapewnienie jakości).

Standardy ISO 9001 - 9003 precyzują minimalne wymagania w stosunku do systemu technicznej kontroli jakości u dostawcy, który niezbędny jest dla stworzenia u odbiorcy przeświadczenia o należytej jakości dostarczanych produktów (zewnętrzne zapewnienie jakości).

Standard ISO 9000 ma charakter przewodnika w stosunku do pozostałych.

2.2. Efektywny system zarządzania przez jakość powinien być oczywiście tak zaprojektowany, aby prowadził do zaspokojenia potrzeb odbiorcy produktów przy równoczesnym zapewnieniu interesu producenta.

System zarządzania przez jakość, którego ustanowienie polega na zdefiniowaniu:

- właściwej struktury organizacyjnej,
- układu odpowiedzialności,
- procedur postępowania,

- układu procesów oraz
- odpowiednich środków,

powinien wynikać z przyjętej przez producenta polityki w zakresie jakości. Polityka taka powinna jasno określać główne czynniki wpływające na jakość wyrobów danego producenta i jej poszczególne elementy takie, jak : zgodność z celem zamierzonego użycia, sposób działania, bezpieczeństwo w użytkowaniu czy niezawodność.
System powinien być wprowadzony w przedsiębiorstwie w taki sposób, aby był przejrzysty, zrozumiały dla wszystkich uczestników procesu zarządzania.

2.3. W systemie zarządzania przez jakość istotną rolę odgrywa jego właściwe udokumentowanie.
Omawiany standard stawia wymagania, aby dokumentacja systemu składała się przynajmniej z następujących elementów wymienionych poniżej.
1) Podstawową formą dokumentu niezbędnego dla opisu i wdrożenia systemu jest "Podręcznik jakości" (Quality Manual).
- Podręcznik ten powinien zawierać dokładny opis systemu, jego struktury i wzajemnych powiązań elementów.
- Powinien on określać wszystkie elementy dokumentacji tworzonej w procesie zarządzania oraz zasady ich identyfikacji, gromadzenia, dystrybucji i pielęgnacji.
- Powinien on ponadto określać zasady wprowadzania wszelkich zmian i uzupełnień systemu.

2) W przypadku zamiaru wprowadzenia do produkcji nowego wyrobu (podejmowania nowego przedsięwzięcia) powinien zostać opracowany "Plan zapewnienia jakości"(Quality plan), uwzględniający zasady

systemu.

W ramach tego planu należy zdefiniować :

a) parametry jakościowe, które należy osiągnąć,

b) rozkład kompetencji i odpowiedzialności w poszczególnych fazach przedsięwzięcia,

c) procedury, metody i instrukcje postępowania, które będą miały zastosowanie w poszczególnych fazach przedsięwzięcia,

d) plan czynności związanych z testowaniem, inspekcjami, lustracjami oraz innymi formami weryfikacji i atestacji wyników poszczególnych faz przedsięwzięcia,

e) zasady wprowadzania zmian w planie w trakcie przedsięwzięcia, jeśli zajdzie tego konieczność,

f) inne środki niezbędne dla uzyskania należytej jakości.

Należy zwrócić uwagę na szczególne znaczenie tego rodaju planu w przypadku produkcji oprogramowania.Tworzenie każdego nowego programu (systemu) jest formą wprowadzenia do produkcji nowego "wyrobu" (nowym przedsięwzięciem). W p.3 niniejszego wykładu omówione zostaną standardowe wymagania IEEE w stosunku do "Planu zapewnienia jakości oprogramowania" (Software Quality Assurance Plan). oraz w stosunku do "Planu weryfikacji i atestacji oprogramowania" (Software Verification and Validation Plan). Pierwszy z nich obejmuje omawiany powyżej "Plan zapewnienia jakości". Drugi- element d) tego planu.

3) Wyniki wszystkich przeprowadzanych czynności kontrolnych t.j. testów,inspekcji, lustracji i t.p. powinny być rejestrowane, tworząc tym samym ważny element dokumentacji systemu.

2.4. Omawiany standard ISO wymaga, aby system zarządzania przez

jakość był poddawany systematycznym lustracjom. Lustracja powinny stwierdzić, w jakim stopniu poszczególne elementy systemu wpływają na osiągnięcie właściwego poziomu jakości. Lustracje takie powinny odbywać się zgodnie z planem ustalonym przez kierownictwo firmy.

Plan lustracji powinien określać:

- czynności poddawane lustracjom,
- kwalifikacje personelu dokonującego lustracji,
- podstawę lustracji (np. reorganizacja, nasilenie nieprawidłowości i t.p.),
- sposoby ewidencjonowania wyników lustracji, wniosków oraz zaleceń.

Lustracje powinny obejmować:

- struktury organizacyjne,
- procedury techniczne i administracyjne,
- personel, narzędzia pracy i materiały,
- produkty wynikowe,
- dokumentację.

Personel przeprowadzający lustracje powinien być niezależny od personelu wykonującego czynności w ramach samego systemu zarządzania.

Należy stworzyć możliwości niezależnego przeglądu i oceny systemu. Począwszy od 1987 r.atestacji systemów zarządzania przez jakość w firmach software'owych dokonują: BSI w Wielkiej Brytanii oraz KEMA w Holandii.

2.5. Omawiany standard określa dalej szczegółowe wymagania w stosunku do systemu zarządzania przez jakość w odniesieniu do poszczególnych zadań odgrywających istotną rolę w pętli jakości.

Odnosi się w szczególności do zadań:

- marketingu (cz.7),
- specyfikowania i projektowania (cz.8),
- zaopatrzenia (cz.9),
- produkcji (cz.10),
- kontroli toku produkcji (cz.11),
- technicznej kontroli produktów (cz.12),
- kontroli sprzętu pomiarowo-kontrolnego (cz.13),
- w sferze poprodukcyjnej (cz.16)

Określa ponadto wymagania dotyczące ujęcia przez system zagadnień:

- postępowania w przypadku stwierdzenia nieprawidłowości w toku produkcji (cz.14),
- postępowania korekcyjnego (cz.15),
- dokumentacji dotyczącej jakości (cz.17),
- kwalifikacji personelu (cz.18),
- ostrzegania użytkownika przed ewentualnymi zagrożeniami (cz.19),
- zastosowanie metod statystycznych w kontroli jakości (cz.20).

Wymagania te należy traktować jako elementy, które należy uwzględnić w ramach systemu. powinny one znaleźć np. odzwierciedlenie w Podręczniku jakości (p.2.3.1), planach zapewnienia jakości (p.2.3.2), rejestrach czynności kontrolnych (p.2.3.3) i t.p.

Tworzenie oprogramowania jest produkcją specyficzną ze względu na niematerialny charakter jej przedmiotu. Upraszczając sprawę można powiedzieć, że cały proces prowadzący do budowy prototypu sprowadza się do zadań specyfikowania i projektowania, a produkcją jest wielokrotne kopiowanie tego prototypu.

Zagadnienie jakości w zaopatrzeniu sprowadza się np. do

zagwarantowania należytej jakości oprogramowania narzędziowego lub oprogramowania, które stanowić będzie środowisko tworzonego produktu.

W związku z powyższym poświęcimy większą uwagę jedynie głównym wymaganiom stawianym przez omawiany standard w stosunku do zadań specyfikowania i projektowania zwracając przy tym uwagę, że - jeśli chodzi o oprogramowanie - konkretyzacja wymagań w stosunku do planów zapewnienia jakości oraz związanych z nimi zadań i dokumentacji omawiana będzie w p.3 niniejszego wykładu, dotyczącym wymagań IEEE.

Standard ISO wymaga np.,aby:

1) dla każdego projektu określony został program zadań z wyraźnie określonymi fazami projektu i punktami kontrolnymi - zgodnie z rodzajem przedmiotu projektowania,

2) dla każdej fazy projektu określone były metody oraz kryteria dopuszczenia lub odrzucenia,

3) dla każdej fazy projektu określony był taki sposób dokumentowania jej wyników, który zapewni ich wykorzystanie w sposób jasny i zrozumiały - zarówno w dalszych fazach projektu, jak i w celu dokonania korekt faz poprzednich,

4) w przygotowaniu programu dokonana została analiza, które fazy projektu mają szczególne znaczenie dla uzyskania właściwej jakości, a w szczególności dla zmniejszenia ewentualnych zagrożeń związanych z użytkowaniem produktu.

Ponadto wymagane jest np., aby system zarządzania określił tryb postępowania związany z wydawaniem, zmianami oraz wykorzystaniem poszczególnych składników dokumentacji projektowej. Obowiązujące w tym zakresie zasady składają się na "system zarządzania

konfiguracją" (configuration management).

Warto w tym miejscu wspomnieć, że ISO przyjęła standardy dotyczące sposobu dokumentowania toku projektów w zakresie aplikacyjnych systemów oprogramowania. Są to :

- Guidelines for the dokumentation of computer - based application systems, ISO 6592 - 1985,
- User documentation and cover information for consumer software packages, ISO 9127 - 1988.

2.6. W dalszym toku niniejszego wykładu przedstawione zostaną wymagania IEEE w stosunku do planów zapewnienia jakości oprogramowania oraz planów weryfikacji i atestacji oprogramowania. Są to przykłady konkretyzacji wymagań, o których była mowa w p.2.3. powyżej. Stanowią one równocześnie wynik przyjęcia określonego modelu w procesie tworzenia oprogramowania.

3. Plan zapewnienia jakości oprogramowania.

3.1. Jak wspomniano powyżej (p.2.3), rozpoczęcie nowego przedsięwzięcia w zakresie oprogramowania powinno być poprzedzone sporządzeniem planu zapewnienia jakości (SQAP). Plan taki powinien składać się z następujących części :

1) Cel i zakres planu,
2) Dokumenty związane,
3) Zarządzanie,
4) Dokumentacja,
5) Standardy, praktyki, konwencje,
6) Przeglądy i lustracje,
7) Zarządzanie konfiguracją,

8) Ewidencja nieprawidłowości i działanie korekcyjne,

9) Narzędzia, techniki i metodologie,

10) Nadzór nad kodem wynikowym,

11) Nadzór nad nośnikami,

12) Kontrola dostawców,

13) Gromadzenie, pielęgnacja i podtrzymywanie dokumentacji toku przedsięwzięcia.

Plan powinien zawierać wszystkie trzynaście części, a jeżeli któraś z nich nie ma zastosowania w danym przypadku należy to - pod nagłówkiem części - wyraźnie napisać.

Treść poszczególnych części jest następująca:

3.1.1. Pierwsza część planu powinna zawierać informacje składające się na określenie celu i zakresu jego sporządzenia. W szczególności powinna odpowiadać na pytania:

- jakich produktów (z ustanowieniem nazw i skrótów) plan dotyczy?,
- jakie jest zamierzone użycie oprogramowania, którego dotyczy plan i w jakim stopniu oprogramowanie ma charakter krytyczny (tzn. jego użycie może spowodować zagrożenie)?,
- czy plan jest tworzony ze względu na wewnętrzne wymagania firmy czy jest wynikiem żądania zamawiającego?,
- jakie dokumenty są podstawą planu (np. tylko omawiany standard IEEE/ANSI 730-1981 lub dodatkowe wymagania branżowe)?,
- czy są jakieś uzasadnienia dla odstępstw od wymienionych wyżej standardów?.

3.1.2. W części drugiej planu powinna znaleźć się lista dokumentów powoływanych w którym kolwiek miejscu planu.

3.1.3. Trzecia część planu powinna określać strukturę organizacyjną, zadania i układ odpowiedzialności związane z realizacją planu.

Określając strukturę organizacyjną należy szczególną uwagę poświęcić kwestii relacji elementów związanych z realizacją zadań planu do elementów związanych z samym tworzeniem oprogramowania. Należy ponadto określić kompetencje w zakresie zatwierdzania planu i jego ewntualnych zmian oraz w zakresie zatwierdzania zgodności uzyskanych wyników z założeniami planu.

Określając zadania należy dla każdego z nich sprecyzować kryteria rozpoczęcia i zakończenia. Jeżeli nie opracowano planu rozwoju oprogramowania (SDP) (który jest fakultatywnym elementem dokumentacji SQAP,por. cz.4) należy w tej części określić harmonogram zadań składających się na cykl realizacji przedsięwzięcia.

Określając układ odpowiedzialności należy wskazać elementy struktury organizacyjnej oraz konkretne osoby odpowiedzialne za nadzór nad realizacją poszczególnych zadań i ich uprawnienia do zatwierdzania dokumentów tworzonych w ich trakcie.

3.1.4. Czwarta część planu zapewnienia jakości oprogramowania ma szczególne znaczenie.

Dotyczy ona bowiem dokumentacji, która:

- powinna zostać sporządzona przed rozpoczęciem przedsięwzięcia i stanowić załącznik planu,
- powinna powstać w toku jego realizacji.

Dokumentację tę należy wymienić w omawianej części planu oraz określić w niej wymagania w stosunku do treści i formy

dokumentacji, która ma powstać w toku rozwoju oprogramowania. Omawiany standard nr 730-1984 określa wymagania w stosunku do treści i formy dokumentacji, którą należy sporządzić równocześnie ze sporządzeniem SQAP.

Określa on również wymagania minimalne, które powinny zostać uwzględnione przy sporządzaniu wymagań w stosunku do dokumentacji, która ma powstać dopiero w toku realizacji przedsięwzięcia.

Dokumentacja ta dotyczy toku:

- tworzenia i rozwoju oprogramowania,
- jego weryfikacji i atestacji,
- eksploatacji,
- pielęgnacji oprogramowania.

Dokumentacja, która powinna zostać sporządzona przed rozpoczęciem realizacji przedsięwzięcia to:

- plan weryfikacji i atestacji oprogramowania (SVVP).

Zaleca się ponadto sporządzenie:

- planu rozwoju oprogramowania (SDP),
- planu zarządzania konfiguracją (SCMP),
- podręcznika standardów i procedur, które należy stosować w toku tworzenia oprogramowania.

Dokumentacja, która powinna powstać w toku realizacji przedsięwzięcia i dla której należy określić wymagania to przynajmniej:

- specyfikacja wymagań w stosunku do oprogramowania (SRS),
- dokumentacja projektowa (SDD),
- raport z weryfikacji i atestacji (SVVR),
- dokumentacja użytkowa.

Szczególne znaczenie ma specyfikacja wymagań (SRS). Podstawowym

bowiem kryterium odbioru wytworzonego oprogramowania (atestacji) jest stwierdzenie, że kod wynikowy oprogramowania realizuje wszystkie funkcje w sposób określony w tej specyfikacji (por.3.2). Należy zwrócić uwagę, że niezależnie od wymagań, które precyzuje omawiany przez nas standard, ustanowione zostały standardy odnoszące się do niektórych z wymienionych rodzajów dokumentacji. Wymagania w stosunku do dokumentacji SRS określa standard ANSI/IEEE nr 830-1984.

Wymagania w stosunku do dokumentacji SDD określa standard IEEE nr 1016-1987.

Wymagania w stosunku do planu weryfikacji i atestacji określa standard ANSI/IEEE nr 1012-1986. Wymagania te omówione zostaną poniżej, w p.3.2 niniejszego wykładu.

Wspomniano ponadto powyżej, w p.2.6 niniejszego wykładu, że wymagania w stosunku do dokumentacji użytkowej określone zostały standardem ISO.

3.1.5. W piątej części planu należy określić:

- standardy,praktyki i konwencje, które należy stosować w toku tworzenia i rozwoju oprogramowania,
- sposoby, w jaki sprawowany będzie nadzór nad ich przestrzeganiem.

Wspomniane standardy, praktyki i konwencje powinny zostać omówione w rozbiciu na poszczególne planowane fazy w toku realizacji przedsięwzięcia.

Tak więc na przykład dla fazy sporządzania wymagań (SRS) należy określić stopień formalizacji języka opisu poszczególnych wymagań, wykorzystywane metody opisu, modele i t.p.

Dla fazy projektu należy określić np. konwencje co do tworzenia

identyfikatorów, opisu struktur danych, opisu struktury programów, interfejsów międzymodułowych i t.p.

Zalecenia dla fazy implementacji powinny uwzględniać np. informacje na temat hardware'u, języka(ów) programowania, maksymalnego rozmiaru modułów, zasad etykietowania, rozmieszczenia tekstów w listingach i t.p.

Szczególną uwagę należy poświęcić omówieniu zaleceń w omawianym zakresie dla fazy testowania. Dotyczy to testowania poszczególnych jednostek, testów integracyjnych, testów systemu oraz testów odbioru. Należy zwrócić uwagę, że szczegółowe zalecenia w tym zakresie ustanowione są w standardzie ANSI/IEEE nr 829-1983.

Równie istotne jest sformułowanie zaleceń w omawianym zakresie w stosunku do fazy sporządzania dokumentacji oprogramowania. Zalecenia te w znacznym stopniu wynikają z wymagań, o których mowa jest w poprzedniej części planu.

3.1.6. W tej części planu omówione powinny zostać planowane przeglądy i lustracje dotyczące toku realizacji przedsięwzięcia oraz sposób ich dokonywania i dokumentowania.

Dla każdego przeglądu czy lustracji należy określić:

- jego przedmiot oraz sposób przeprowadzenia,
- dokumenty przedkładane oraz dokumenty tworzone w jego wyniku,
- kryteria oceny,
- uczestników tych czynności oraz osobę odpowiedzialną za sporządzenie dokumentów wynikowych,
- sposób wykonania ewentualnych zaleceń wynikłych z tych czynności.

Omawiany standard wymaga, aby zaplanowano i dokonano przynajmniej:

- przeglądu (rewizji) specyfikacji wymagań,
- przeglądu (rewizji) projektu wstępnego,
- przeglądu (rewizji) projektu szczegółowego,
- przeglądu (rewizji) planu weryfikacji i atestacji,
- lustracji funkcjonalnej oprogramowania wynikowego,
- lustracji fizycznej oprogramowania wynikowego,
- lustracji międzyoperacyjnych,
- lustracji w trybie nadzoru.

Omawiany standard określa minimalne kryteria pozytywnej oceny dla poszczególnych przedmiotów przeglądu czy lustracji.

3.1.7. Ta część SDAP ma zawierać plan zarządzania konfiguracją. Plan taki ma określić metody identyfikacji poszczególnych elementów całości tworzonego oprogramowania oraz ich statusu, a także zasady wprowadzania i dokumentowania zmian. Ta część może być pusta, jeżeli plan zarządzania konfiguracją (SCMP) jest odrębnym dokumentem załączonym do planu zapewnienia jakości (por.p.3.1). Wymagania szczegółowe co do treści i formy SCMP określone zostały standardem ANSI/IEEE nr 828-1983.

3.1.8. W ósmej części planu zapewnienia jakości należy określić zasady śledzenia, ewidencjonowania i usuwania nieprawidłowości, a także ustalić osoby odpowiedzialne za przestrzegania tych zasad.

3.1.9. Ta część planu powinna zawierać informacje na temat narzędzi, technik i metodologii wykorzystywanych w toku tworzenia i rozwoju oprogramowania.

Należy w niej na przykład wymienić konkretne środki udostępnione

przez system operacyjny, debuggery, generatory tekstów i tym podobne narzędzia programowe przewidziane do wykorzystania w realizacji przedsięwzięcia.

3.1.10. Ta część planu powinna określić metody postępowania oraz środki przedsięwzięte dla ochrony wersji wzorcowych poszczególnych elementów oprogramowania. W szczególności powinna określać organizacyjne zasady ochrony przed wprowadzaniem niepowołanych zmian.

3.1.11. Podobnie, jak w przypadku części dotyczącej ochrony wzorcowych wersji oprogramowania, w części jedenastej SQAP należy określić zasady ochrony nośników przed niepowołanym dostępem oraz uszkodzeniem czy degradacją.

3.1.12 W tej części planu należy określić środki zastosowane w celu zapewnienia, że elementy oprogramowania dostarczane przez poddostawców będą wytworzone według zasad wynikających z omawianaego planu zapewnienia jakości. Jako minimum wymagań należy przyjąć, że poddostawca przygotuje i przedłoży do wglądu własny plan zapewnienia jakości. Należy wówczas określić zasady dokonywania lustracji w toku realizacji tego planu u poddostawcy lub uczestnictwa w lustracjach, które z tego planu wynikają.

3.1.13. Ostatnia część planu zapewnienia jakości powinna określić zasady archiwowania i ochrony wszelkich dokumentów, które powstają w toku jego realizacji. Dla każdego rodzaju dokumentów należy określić okres przechowywania, zasady dostępu, środki podjęte dla

ochrony przed utratą i t.p.

3.1.14. Omówiony plan zapewnienia jakości oprogramowania jest przykładem jednego z istotnych dokumentów, które stanowią podstawę systemu zarządzania przez jakość, o których była mowa w p.2 niniejszego wykładu.

Niezależnie od sporządzenia planu zapewnienia jakości oraz dokonania jego oceny, przed firmą stoi problem praktyczny wdrożenia jego zasad do realizacji. Rozwiązując ten problem należy oczywiście dokonać realnej oceny umiejętności personelu, przewidzieć odpowiednie szkolenia, dokonać bilansu zasobów i t.p. Dokonywane w toku realizacji planu lustracje mogą również być podstawą sugerowania jego zmian, a kierownictwo firmy powinno określić zasady ich dokonywania.

3.2. Jednym z dokumentów wymaganych jako element planu zapewnienia jakości oprogramowania jest plan weryfikacji i atestacji. Najogólniej mówiąc plan taki powinien określić metody oraz konkretne zadania w celu stwierdzenia, że:

- wymagania sprecyzowane w ich specyfikacji (SRS) są prawidłowo odzwierciedlone w projekcie oprogramowania (SDD),
- projekt oprogramowania jest prawidłowo zaimplementowany, co ma odzwierciedlenie w kodzie wynikowym,
- kod wprowadzony do komputera realizuje funkcje w sposób zgodny ze specyfikacją wymagań (SRS).

Wspomniano powyżej w p.3.1.4, że treść i forma planu weryfikacji i atestacji oprogramowania jest przedmiotem standardu ANSI/IEEE nr 1012-1986. Standard ten precyzuje ogólny schemat planu oraz

ustala m.in. minimum zadań weryfikacji i atestacji, które należy wykonać dla tzw. oprogramowania krytycznego (critical software), t.j. oprogramowania, którego wady mogą spowodować zagrożenia lub doprowadzić do strat materialnych czy społecznych.

Plan weryfikacji i atestacji powinien - dla każdego z zadań w nim wymienionych - określać:

- dokumenty niezbędne dla jego wykonania,
- kryteria oceny,
- treść i formę raportu z jego wykonania,
- osoby uczestniczące w wykonaniu zadania oraz uprawnione do zatwierdzania raportów.

Poniżej wymienimy minimalny zestaw zadań dla poszczególnych faz tworzenia i rozwoju oprogramowania w modelowym układzie faz cyklu życia oprogramowania.

3.2.1 Dla fazy opracowywania koncepcji należy dokonać oceny jej dokumentacji wynikowej. Dokonując tej oceny należy zwrócić uwagę na stopień uwzględnienia potrzeb użytkownika, uwarunkowań środowiska sprzętowo- programowego oraz krytycznego charakteru oprogramowania.

3.2.2. W fazie opracowania specyfikacji wymagań należy:

- dokonać oceny dokumentacji wynikowej tej fazy (SRS),
- opracować plan dla testów systemu,
- opracować plan dla testów odbioru.

Dokonując oceny SRS należy stwierdzić m.in.:

- czy specyfikacja spełnia wymagania formalne (t.j. jest jednoznaczna, zupełna, wewnętrznie niesprzeczna, czytelna, łatwa

do śledzenia),

- czy odzwierciedla tezy zawarte w koncepcji.

Przez testy systemu rozumie się testy prawidłowości funkcjonowania całości oprogramowania w środowisku sprzętowym ze względu na wymagania projektu. Testy odbioru przygotowywane są pod kątem wymagań użytkownika, które określone są w ich specyfikacji i których pozytywne wykonanie stanowić będzie kryterium odbioru oprogramowania.

3.2.3. W fazie projektu należy:

- dokonać oceny dokumentacji projektowej (SDD),
- opracować plan dla testów modułów,
- opracować plan dla testów integracji,
- zaprojektować testy.

Dokonując oceny SDD należy stwierdzić spełnienie przez dokumentację wymogów formalnych oraz uwzględnienie w projekcie wszystkich wymagań wyspecyfikowanych w SRS.

Przez testy modułów rozumie się testy weryfikujące działanie poszczególnych modułów programowych zgodnie z funkcjami opisanymi w projekcie.

Testy integracji mają za zadanie rozstrzygnąć prawidłowość współdziałania modułów programowych w kolejnych krokach integracji składowych oprogramowania - aż do uzyskania całości.

3.2.4. W fazie implementacji należy:

- dokonać oceny i weryfikacji opracowanego kodu źródłowego,
- dokonać oceny dokumentacji programów źródłowych,
- przygotować dane dla wszystkich rodzajów testów,

- wygenerować procedury testów modułów integracji oraz systemu zgodnie z opracowanymi projektami,
- przeprowadzić testy dla poszczególnych modułów.

Ocena oraz weryfikacja opracowanego kodu powinna stwierdzić zgodność dokonanej implementacji z opracowanym projektem. Dokumentacja programów źródłowych powinna odpowiadać wymaganiom ustalonym i określonym w cz.5 planu zapewnienia jakości oprogramowania (p.3.1.5).

3.2.5. W fazie testowania należy :
- wygenerować procedury testów odbioru zgodnie z opracowanym projektem,
- przeprowadzić testy integracji, systemu oraz testy odbioru.

3.2.6. W fazie instalacji należy przeprowadzić końcową lustrację konfiguracji systemu a w tym dokonać oceny dokumentacji użytkowej oraz sporządzić raport z weryfikacji i atestacji.

3.2.7. W fazie eksploatacji i pielęgnacji plan weryfikacji i atestacji powinien przewidywać zadania:
- oceny stwierdzonych w trakcie eksploatacji nieprawidłowości,
- oceny proponowanych zmian w oprogramowaniu,
- wynikającej z konieczności dokonania zmian w oprogramowaniu rewizji planu weryfikacji i atestacji,
- powtórzenia wcześniejszych zadań w związku z dokonanymi w oprogramowaniu poprawkami i zmianami.

3.3. Przedstawiony powyżej schemat planu zapewnienia jakości

oprogramowania stanowi w założeniu opis istotnego elementu systemu zarządzania przez jakość. Precyzuje on podstawowe punkty zarządzania przedsięwzięciem. Sam schemat abstrahuje od konkretnych metod czy narzędzi, wymaga jednak, aby były one w odpowiedni sposób ustalone,opisane i udokumentowane.

W odniesieniu do realiów większości podejmowanych w Polsce przedsięwzięć w zakresie oprogramowania opisane wymagania mogą wydawać się nadmiernie biurokratyczne.

Znaczenie świadomości istnienia takich wymagań oraz wiedzy w tym zakresie poddaję pod dyskusję.

4. Przykład znaku jakości dla oprogramowania.

Intencją niniejszego wykładu było uświadomienie, że jakość oprogramowania, pojmowana jako stopień zaspokojenia potrzeb jego użytkowników jest celem, którego osiągnięcie wymaga panowania nad cyklicznym procesem obejmującym badania, wytworzenie, upowszechnienie i analizę wyników eksploatacji. Panowanie nad tym procesem wymaga znacznego wysiłku związanego z przestrzeganiem ustalonych reguł organizacji procesu wytwarzania oraz marketingu.

Ograniczenie rozważań na temat jakości oprogramowania do określenia parametrów jakościowych gotowego produktu programowego spłyca zagadnienie. Możliwe jest oczywiście określenie kryteriów, jakie powinien spełniać produkt, aby kwalifikował się do nadania mu znaku jakości. Kryteria takie odpowiadać mogą np. kryteriom, które przyjmowane są w trakcie dokonywania lustracji funkcjonalnej i fizycznej oraz testów odbioru - zgodnie z wymaganiami, o których mowa była w p.3 niniejszego wykładu.

Kryteria takie pozostawiają poza obszarem zainteresowania kwestie

organizacji procesu wytwarzania.

Kryteria takie mogą być wystarczające dla potencjalnego użytkownika z jednym wyjątkiem.

Abstrahowanie od organizacji procesu wytwarzania może postawić pod znakiem zapytania zdolność producenta do prowadzenia właściwego rozwoju produktu.

Kryteria takie określone zostały np. przez zachodnioniemiecki Związek Jakości Oprogramowania, organizację zrzeszającą kilkadziesiąt poważnych firm komputerowych i zaaprobowane przez RAL (Instytut Jakości w RFN) (8). W ramach tego związku funkcjonują stacje, które są autoryzowane przez RAL do nadawania produktom programowym znaku jakości.

Przedmiotem prowadzonych badań kwalifikacyjnych i oceny jest program oraz dokumentacja użytkowa.

Kwalifikacji programu dokonuje się na postawie testowania, które stwierdzić ma, że program realizuje prawidłowo wszystkie funkcje wymienione w dokumentacji.

Ocenia się ponadto program pod kątem wygody użytkowania, przejrzystości dialogów, jednolitości zasad komunikowania i t.p.

Przeprowadzając ocenę dokumentacji analizuje się, czy informuje ona użytkownika o funkcjach programu i zakresie jego zastosowania w sposób zupełny oraz dostatecznie precyzyjny.

Analiza kryteriów RAL prowadzi do wniosku, że odpowiadają one wymaganiom określonym przez standardy dotyczące jakości.

5. Literatura.

1. Słownik Języka Polskiego, t.I, PWN W-wa 1982,

2. EOQC Glossary of terms used in quality control, 1976,

3. M.M.Lehmann,L.A.Belady,Program Evolution,AP (1985),

4. Standard for Software Life Cycle Processes,IEEE P1074 (1988),

5. Quality Systems, ISO 9000-9004 (1987),

6. Software Quality Assurance Plans, IEEE 730-1984,

7. Software Verification and Validation Plans, IEEE 1012-1986,

8. Software Gutesicherung, RAL-GZ901, 1989.

Jesienna Szkola PTI
Mragowo, listopad 1989

PROGRAMOWANIE W LOGICE
Krótkie Wprowadzenie

Jan Małuszyński
Linköping University
Department of Computer and Information Science
582-83 Linköping, Sweden

1. Wstęp

Programowanie w logice liczy juz sobie ponad 15 lat od czasu uruchomienia pierwszego interpretatora. W dalszym ciągu jednak jest ono przedmiotem żywych, często emocjonalnych, dyskusji oraz dziedziną intensywnych badań. Celem tego wykladu jest krótkie omówienie podstawowych pojęc programowania w logice oraz próba scharakteryzowania aktualnych kierunków rozwoju. Sposób przedstawienia tematu oparty jest na przygotowywanej do druku książce [6]. Czytelnikowi zainteresowanemu techniką programowania w Prologu można polecić [10] jako jeden z wielu podręczników w tej dziedzinie. Matematyczne podstawy programowania w logice omawia szczegółowo monografia [5].

Na wstępie zilustrujemy na przykładzie podstawową rożnicę między programami logicznymi a programami w językach algorytmicznych. Program w języku algorytmicznym składa się z instrukcji, które opisują kroki obliczeniowe wykonywane następnie przez maszynę. Program służy zwykle do przekształcenia danych początkowych na wyniki, a więc określa pewną funkcję lub relację na danych. Na przykład algorytmiczny program sortowania liczb naturalnych określa funkcję na ciągach liczb. Dla uproszczenia przyjmijmy że są to ciągi bez powtórzeń elementów. Dany ciąg wejściowy w wyniku operacji obliczeniowych opisanych w programie zostaje przekształcony na ciąg rosnący złożony ze wszystkich elementów ciągu pierwotnego. Problem sortowania można opisac następująco:

1. Wynikiem sortowania ciągu pustego jest ciąg pusty.
2. Jeśli ciąg jest postaci $a\,\alpha$ gdzie a jest pierwszą liczbą ciągu to wynikiem jego sortowania jest ciąg $\alpha_1\,a\,\alpha_2$ gdzie α_1 jest wynikiem sortowania podciągu liczb mniejszych od a a α_2 jest wynikiem sortowania podciągu liczb większych niż a.

Powyższe sformułowanie problemu sortowania może być uważane za program logiczny. Zauważmy, że nie jest to ciąg instrukcji do wykonania lecz (rekurencyjna) definicja dwuargumentowej relacji "sortowanie" na ciągach liczb. Jest to relacja pomiedzy wejściowym i wynikowym ciagiem liczb. Relacja ta jest funkcją. Ogólnie, program logiczny jest zbiorem zdań oznajmujących zapisanych w pewnym języku formalnym. Najbardziej rozpowszechnionym jezykiem programowania w logice jest Prolog. Przykładowe zdania (1) i (2) można zapisać w Prologu (składnia edynburska) w następujący sposób:

```
sortowanie([],[]).
sortowanie([A|Alfa], Wynik) :-  podciagi(A,Alfa,Mniejszy,Wiekszy),
                                sortowanie(Mniejszy,Alfa1),
                                sortowanie(Wiekszy,Alfa2),
                                zlozenie(Alfa1,[A|Alfa2],Wynik).
```

Formalny zapis zdania (2) wymaga dodatkowego komentarza. Oprócz dwuargumentowej relacji *sortowanie* występują tu jeszcze nazwy dwóch innych relacji: *podciagi* i *zlozenie*. Relacja *podciagi* ma cztery argumenty. Pierwszy z nich jest liczbą a pozostałe ciągami liczb bez powtórzeń. Relacja składa się z takich czwórek *<a, alfa, alfa1, alfa2>* ze *alfa1* jest podciągiem *alfa* złożonym ze wszystkich elementów mniejszych od *a*, natomiast *alfa2* jest podciągiem *alfa* złożonym ze wszystkich elementów większych od *a*, np. <3, [5,1,7, 6,4,3], [5,7,6,4], [1] >. Relacja *zlozenie* jest trójargumentowa. Jej argumentami są ciągi liczb. Składa się ona z trójek *<alfa, beta, gamma>* takich że ciąg *gamma* może być otrzymany przez dopisanie ciągu *beta* do ciągu *gamma*, np.<[1,7,3],[2,4,5],[1,7,3,2,4,5]>. Zdanie (2) jest zdaniem warunkowym, które możemy odczytać jak następuje.

Jeśli

Mniejszy i *Wiekszy* są podciągami ciągu *Alfa* wyznaczonymi przez *a*,
Alfa1 jest wynikiem sortowania ciągu *Mniejszy*, i
Alfa2 jest wynikiem sortowania ciągu *Wiekszy*, i
Ciąg *Wynik* powstaje przez dopisanie liczby *A* i ciągu *Alfa2* do *Alfa1*

to

Wynik jest wynikiem sortowania ciągu złożonego z *A* i *Alfa*.

Jest to uscislone sformulowanie zdania (2).

Podsumowując rozważany przykład możemy stwierdzić że program logiczny opisuje nie obliczenie lecz relacje. Zachodzi tu pewne podobieństwo z relacyjnymi bazami danych. Jednakże tradycyjne bazy danych są zawsze skończone a relacje opisywane przez program logiczny mogą być nieskończone, jak w rozważanym przykładzie. Podobnie jak w przypadku bazy danych z programu logicznego korzysta się zadając pytania dotyczące opisanych relacji. W stosunku do relacji sortowanie można zadać pytanie o posortowaną wersję danego ciągu ,np. [3,1,7]. W Prologu miałoby ono postać

```
? sortowanie([3,1,7],X).
```

a uzyskana odpowiedz miałaby postać

```
X=[1,3,7].
```

W następnym przykładzie pokażemy również inne rodzaje pytań. Przykład opisuje pokrewieństwo pomiędzy osobami. Interesująca nas relacja *rodzic* wiąże rodziców z dziećmi i jest przedstawiona w postaci następującej podanej explicite bazy danych.

```
rodzic(anna,boleslaw).
rodzic(mieszko, boleslaw).
rodzic(boleslaw,mieszko11).
rodzic(mieszko, dobrogniew).
rodzic( jagna, dobrogniew).
rodzic(dobrogniew, scibor).
```

Oprocz tego wiadomo że, jesli *X* jest rodzicem *Y* i *Y* jest rodzicem *Z* to *Z* jest wnukiem *X*, co można zapisać jak następuje

```
wnuk(X,Z) :- rodzic(X,Y),rodzic(Y,Z).
```

Program w Prologu złożony z powyższej bazy danych i z reguły definiującej relację *wnuk* może odpowiedzieć na następujące przykładowe zapytania (odpowiedzi podane są przy każdym zapytaniu):

"Czy Boleslaw jest dzieckiem Mieszka?"
?rodzic(boleslaw,mieszko). yes.

"Kto jest dzieckiem Mieszka?"
?rodzic(mieszko, X). X=boleslaw; X=dobrogniew

"Kto jest rodzicem Dobrogniewa?"
?rodzic(X,dobrogniew). X=mieszko; X=jagna.

"Kto jest wnukiem Mieszka?"
?wnuk(mieszko,X). X=mieszkoII; X=scibor.

"Kto jest czyim rodzicem?"
?rodzic(X,Y). X=mieszko, Y=boleslaw; X=anna, Y=boleslaw; ...

Można również zdawać złożone pytania, np.

"Z kim Mieszko ma dzieci?"
?rodzic(mieszko,X), rodzic(Y,X). X=boleslaw, Y=anna;
X=dobrogniew, Y=jagna.

"Kto jest wnukiem Mieszka a nie jest dzieckiem Bolesława?"
?wnuk(mieszko,X), not rodzic(boleslaw,X). X=scibor.

2. Wybrane pojęcia logiczne.

Celem tego rozdziału jest wyjaśnienie związku miedzy logiką a programowaniem w logice. Przykładowy program sortowania powstał w wyniku formalnego zapisu zdań opisujących relację *sortowanie*. Formalizacja zdań języka naturalnego jest tradycyjnym przedmiotem zainteresowania logików. Na wstępie omówimy pokrótce język formalny stanowiący podstawę programowania w logice.

2.1. Język programów logicznych

Zdania oznajmujące języka naturalnego opisują relacje pomiędzy pewnymi obiektami, np.

(1) Każda matka kocha swoje dziecko.
(2) Kasia jest matką Zbyszka.

Z dwóch powyższych zdań skłonni jesteśmy wnioskować, że

(3) Kasia kocha Zbyszka.

Zdania (1) i (2) odnoszą się do pewnego zbioru (universum) obiektów, w tym przypadku osób, na którym określone są dwuargumentowe relacje "być matką" i "kochać". Zdania te mogą być prawdziwe lub nie w opisywanej sytuacji. Jeśli są one prawdziwe to prawdziwe musi być także zdanie (3). Przedmiotem badania logiki są reguły które umożliwiają (mechaniczne) konstruowanie wniosków z przesłanek. Poszukuje się reguł poprawnych, tj. takich które gwarantują prawdziwość konstruowanych wniosków w przypadku prawdziwości przesłanek.

Postulat mechanizacji procesu wnioskowania wymaga nałożenia ograniczeń na składnię języka. Analizując przykładowe zdania widzimy, że język musi zawierać następujące elementy:

- symbole do oznaczania elementów universum, takich jak Kasia czy Zbyszek; symbole te nazywają się *stałymi logicznymi:* w języku programów logicznych będziemy w tym celu używać identyfikatorów - nazw rozpoczynających się od małej litery np. *kasia, zbyszek*;

- symbole do oznaczania relacji, np. *kocha, jest-matką* itp; symbole te nazywają się *literami predykatywnymi*, lub krótko *predykatami*. Każdy predykat ma przypisaną liczbę naturalną określającą liczbę jego argumentów.

Najprostsze zdania takie jak "Kasia kocha Zbyszka" przedstawiane będą w postaci formuł atomowych, w tym przypadku *kocha(kasia,zbyszek)*. Zdanie (1) ma jednak bardziej skomplikowaną strukturę, gdyż odwołuje się ono nie do pojedynczych konkretnych obiektów uniwersum lecz do wszystkich elementów. Mówi ono bowiem, że dla każdej pary elementów *X* i *Y*, jeśli *X* jest matką *Y* to *X* kocha *Y*. Widzimy więc, że język formalny musi zawierać również symbole *zmiennych indywiduowych* przebiegających universum. W naszym języku zmienne będziemy reprezentować przez identyfikatory rozpoczynające się od dużej litery. Tak więc język dopuszcza zdania takie jak *matka(X,Y)* i *kocha (X,Y)*. W naszym przykładzie zdania te połączone są spójnikiem zdaniowym "jeśli - to" - który w języku formalnym reprezentowany będzie przez :- z odwróceniem kolejności. Ponadto zwrot "dla każdego *X*" będzie reprezentowany przez symbol kwantyfikatora ogólnego ∀. Przykładowe zdanie (1) może być teraz zapisane jako:

∀X ∀Y (kocha(X,Y) :- matka(X,Y)).

Oprócz spójnika zdaniowego "jeśli" używać będziemy również spójników: dwuargumentowego "i",reprezentowanego przez przecinek, oraz spójnika jednoargumentowego "nie", reprezentowanego przez "not". Na przykład zdanie

"Kto pracuje i nie dogadza sobie ten jest bogaty"

można zapisać w postaci

∀X (bogaty(X) :- pracuje(X), not dogadza(X,X)).

Niekiedy universum może być wyposażone w funkcje. W tym przypadku język zawiera litery funkcyjne reprezentujące funkcje. Stałe logiczne w tym kontekście mogą być uważane za zeroargumentowe litery funkcyjne. Wprowadzenie liter funkcyjnych umożliwia reprezentację elementów universum w języku za pomocą bardziej ogólnej konstrukcji zwanej *termem*. Zmienne i stałe są szczególnymi przypadkami termów. Przykładem termu może być wyrażenie *para(adam,ewa)*. Reprezentuje ono małżeństwo Adama i Ewy, jeśli stała *adam* reprezentuje osobę Adam, stała *ewa* osobę Ewa a litera funkcyjna *para* - funkcję przekształcającą pary osób w małżeństwa.

Omówione wyżej przykłady podsumujemy teraz podając w notacji BNF składnię programów logicznych.

<program logiczny> ::= <program bez negacji>|<program normalny>
<program bez negacji> ::= <klauzula definitywna> |
<klauzula definitywna> <program bez negacji>
<program normalny> :: <klauzula normalna> |
<klauzula normalna> <program normalny>
<klauzula definitywna> ::= <formuła atomowa>. |
<formuła atomowa> :- <ciąg formuł atomowych>.
<ciąg formuł atomowych> ::= <formuła atomowa> |
<formuła atomowa><ciąg formuł atomowych>
<formuła atomowa> ::= <predykat>(<argumenty>).

<argumenty> ::= <term> | <term>,<argumenty>
<term> ::= <stała> | <zmienna> | <litera funkcyjna>(<argumenty>)
<klauzula normalna> ::= <formuła atomowa>. |
<formuła atomowa> :- <ciąg literałów>.
<ciąg literałów> ::= < literał> | < literał><ciąg literałów>
<literał> ::= <formuła atomowa> | not<formuła atomowa>

Zauważmy, że każdy program bez negacji jest szczególnym przypadkiem programu normalnego. Reguły dotyczące składni elementów leksykalnych, tj. predykatów, zmiennych oraz liter funkcyjnych zostały pominięte. Język programów logicznych zdefiniowany przez powyższe reguły jest podzbiorem języka Prolog. Zauważmy, że nie obejmuje on przykładu sortowania dyskutowanego we wstępie. Jezyk ten jest jednocześnie podzbiorem języka logiki predykatów pierwszego rzędu (z zastosowaniem pewnych specyficznych konwencji notacyjnych obowiązujących w Prologu). W szczególności dotyczy to użycia kwantyfikatora ogólnego w klauzulach. Zakłada sie, ze dla każdej zmiennej X występującej w klauzuli, klauzula poprzedzona jest kwantyfikatorem $\forall X$. W związku z tym nie ma potrzeby umieszczania tego kwantyfikatora jawnie w klauzuli. Klauzula w postaci formuły atomowej nazywa sie *klauzulą jednostkową*.

Jak wspomniano wyzej,z programów logicznych korzysta się za pomocą *zapytań*. Oprócz języka programów zdefiniujemy więc język zapytań.

<zapytanie> ::= ?<ciąg literałów>.

2.2. Modele programow logicznych

Program logiczny jest zbiorem formuł logiki predykatów pierwszego rzędu. Semantyka programu moze więc być określona w terminach logiki. Podstawowym pojeciem jest tutaj pojecie *interpretacji*. Jak już wspominaliśmy, intencją autora programu logicznego jest opisanie pewnej struktury relacyjnej. Tak więc symbole występujace w programie mają reprezentować, odpowiednio elementy tej struktury, funkcje i relacje. To przypisanie obiektow strukturv symbolom języka nazywa się interpretacją, Interpretacja określa wartość logiczną każdego zdania w programie. Aby to wyjaśnić odwołamy się do składni naszego języka. Podstawowymi jego elementami są termy i formuły atomowe. Każdy term bez zmiennych reprezentuje pewien element uniwersum interpretacji:

- Element reprezentowany przez stałą jest wyznaczony bezposrednio w definicji interpretacji.

- Rozważmy term postaci $f(t_1,...,t_n)$. Litera funkcyjna f reprezentuje pewną n-argumentową funkcję F. Zalozmy, ze termy $t_1,...t_n$ reprezentuja odpowiednio elementy $e_1...e_n$ uniwersum interpretacji. Wowczas cały term reprezentuje wartość funkcji F dla argumentów $e_1,..e_n$. Na przykład jeśli stała *zero* reprezentuje liczbę naturalną 0, litera funkcyjna s reprezentuje funkcję następnik to term $s(zero)$ reprezentuje liczbę 1.

- Możemy teraz zdefiniowac wartość logiczną formuł atomowych bez zmiennych. Kazda z nich jest postaci $p(t_1,...,t_n)$, gdzie p jest n-argumentową literą predykatywną dla pewnego $n>0$. W danej interpretacji litera p reprezentuje n-argumentową relację P. Formula jest prawdziwa wtedy i tylko wtedy gdy n-tka wartosci $<e1,..en>$ reprezentowanych przez termy $t_1..t_n$ należy do relacji P.

- Z kolei mozemy już teraz zdefiniowac wartość logiczną formuł konstruowanych z formuł atomowych bez zmiennych za pomocą spojników zdaniowych. Niech p i q beda formułami, ktorych wartosc logiczna w danej interpretacji I jest znana. Wowczas:

(1) formuła *not p* jest prawdziwa w interpretacji *I* wtedy i tylko wtedy gdy formuła *p* nie jest prawdziwa w tej interpretacji;
(2) formuła *(p,q)* jest prawdziwa w interpretacji *I* wtedy i tylko wtedy gdy zarówno formuła *p* jak i *q* są prawdziwe w tej interpretacji;
(3) formuła *(p:-q)* jest prawdziwa w interpretacji *I* wtedy i tylko wtedy gdy formuła *p* jest prawdziwa w interpretacji *I* lub formuła *q* nie jest prawdziwa w tej interpretacji.

- Wartosc logiczną formuł z kwantyfikatorami określimy przy upraszczającym założeniu, że każdy element universum ma odpowiednią stałą ktora go reprezentuje. W naszym przypadku jedynymi kwantyfikowanymi formułami są klauzule programu. Niech *K* bedzie klauzulą programu logicznego. Jej instancją nazwiemy formułę otrzymaną z *K* przez zastąpienie w *K* każdej zmiennej przez pewien term. (Wielokrotne wystąpienia zmiennej zastępowane są przez wystąpienia tego samego termu). Instancję otrzymaną przez zastąpienie zmiennych przez stałe będziemy nazywali *instancją podstawową*. W przypadku nieskończonego uniwersum każda klauzula ze zmiennymi ma nieskończenie wiele instancji. Oczywiście instancje podstawowe klauzuli nie zawierają już kwantyfikatorów i ich wartości logiczne w danej interpretacji mogą być określone na podstawie uprzednio podanych reguł. Klauzula jest prawdziwa w danej interpretacji wtedy i tylko wtedy gdy wszystkie jej instancje podstawowe są prawdziwe w tej interpretacji.

Możemy już teraz sformułować podstawowe pojęcie *modelu* programu logicznego. Modelem nazywa się każda interpretacja, w której wszystkie klauzule programu są prawdziwe. Pojęcie to ma dość oczywistą intuicję. Punktem wyjścia dla każdego programu logicznego jest pewna struktura relacyjna, którą programista stara się opisać, przypisując elementy języka obiektom tej struktury, a więc korzystając z pewnej interpretacji. Program logiczny powinien poprawnie opisywać strukturę, innymi słowy, przyjęta interpretacja powinna być modelem programu. Program jest jednak napisem w języku formalnym i ma (nieskończenie) wiele interpretacji. Nawet jeśli opisywana struktura jest jego modelem, również wiele innych interpretacji może być jego modelami.

2.3. Model standardowy programu bez negacji.

Istnienie modelu programu logicznego jest zagwarantowane przez jego składnię. Warto jednak zwrócić uwagę na pewien szczególny model ukazujący związek programów logicznych z bazami danych. Podobnie jak w programie logicznym tak i w relacyjnej bazie danych obiekty zewnętrznego świata reprezentowane są przez napisy. Najprostszym sposobem interpretowania tych napisów może być przyjęcie, że reprezentują one same siebie. W programie logicznym dziedziną takiej interpretacji byłby więc zbiór wszystkich termów bez zmiennych, które da się utworzyć z alfabetu liter występujących w programie. Litery funkcyjne reprezentują przy tym funkcje tworzące termy z termów. Takie interpretacje nazywają się interpretacjami Herbrandta. W przypadku relacyjnej bazy danych jak również każdego programu logicznego bez liter funkcyjnych dziedzina taka jest skończonym zbiorem stałych. Interpretacja musi także wiązać predykaty z relacjami. W tym przypadku są to relacje na termach i mogą być zdefiniowane poprzez podanie zbioru formuł atomowych: relacja związana z literą *p* zawiera *n*-tkę termów $<t_1,..t_n>$ wtedy i tylko wtedy gdy formuła $p(t_1,...,t_n)$ należy do tego zbioru. Można udowodnić, że jeśli dwa zbiory formuł atomowych są modelami danego programu to również i ich przecięcie jest modelem. Oznacza to w praktyce że dla każdego programu istnieje najmniejszy model Herbrandta. Jak można go skonstruować i jaka jest jego intuicja? Oczywiście wszystkie instancje bez zmiennych każdej klauzuli jednostkowej muszą należeć do tego modelu. Następnie, jeśli $a:-b_1,..b_k$ jest instancją bez zmiennych pewnej klauzuli i $b_1,..,b_k$ należą do modelu to również *a* musi do niego należeć. Wychodząc ze zbioru wszystkich instancji bez zmiennych klauzul jednostkowych i dodając niezbędne elementy otrzymamy w granicy najmniejszy model. Dla przykładu z p.1 najmniejszy model ma postać:

```
rodzic(mieszko,boleslaw),      rodzic(anna,boleslaw),
rodzic(mieszko,dobrogniew),    rodzic(jagna,dobrogniew),
rodzic(boleslaw,mieszko11),    rodzic(dobrogniew,scibor),
wnuk(mieszko,mieszko11),       wnuk(mieszko,scibor)
wnuk(anna,mieszko11),          wnuk(jagna,scibor).
```

Relacyjna baza danych może być uważana za zbiór klauzul jednostkowych, które wyznaczają natychmiast najmniejszy model. Intuicyjnie oznacza to, ze przyjmujemy tutaj tzw. "założenie zamkniętego świata": informacje których nie ma w bazie danych uważamy za nieprawdziwe. Klauzule jednostkowe programu logicznego mogą być uważane za jawną część opisywanej bazy danych. Do tego dochodzą elementy najmniejszego modelu opisywane przez pozostałe klauzule. Tak więc najmniejszy model można uważać za rozwiniętą bazę danych obejmującą wszystkie informacje zawarte w programie. Wszystkie inne informacje uważa się w nim za nieprawdziwe.

2.4. Konsekwencje logiczne programu.

Program logiczny używany jest do odpowiadania na zapytania dotyczące opisanego modelu. Zapytanie ma postac ciągu (koniunkcji) literałów. Jednakże w języku logiki nie ma zdań pytających. Jaka jest więc logiczna treść zapytania? Jest to negatywne stwierdzenie. Zadaniem systemu jest znalezienie kontrprzykładu który obalałby to stwierdzenie. W rezultacie poszukuje się więc instancji zapytania, która byłaby prawdziwa w opisanym modelu. Jednakże program "opisuje" wszystkie swoje modele a nie tylko ten jeden, o którym myślał programista. W związku z tym wymaga się by znaleziona instancja zapytania była prawdziwa we wszystkich modelach programu. W terminach logiki formuła o tej własności nazywa się *konsekwencją logiczną* zbioru formuł, w tym przypadku - programu. Pojęcie to definiuje się następująco. Formuła *F* nazywa się konsekwencją logiczną zbioru formuł *FF* jeśli każdy model *FF* jest modelem *F*. Warto dodać, że najmniejszy model Herbrandta dla programu logicznego jest zbiorem wszystkich atomowych formuł bez zmiennych które są konsekwencjami logicznymi programu. Ten wynik uściśla stwierdzenie poprzedniego paragrafu, mowiące o tym, że model ten przedstawia jawnie całą informację zawartą w programie.

3. Od logiki do programowania.

3.1. Jak szukać odpowiedzi na zapytanie.

Rozważymy teraz sposób wnioskowania umożliwiający znajdowanie odpowiedzi na zapytania. Ograniczymy się najpierw do programów i zapytań bez negacji. Omówienie reguły wnioskowania używanej przez systemy programowania w logice rozpoczniemy od przykładu. Rozważmy zapytanie

"Kto jest wnukiem Mieszka?"

skierowane do programu z p.1. Ma ono postać

```
?wnuk(mieszko,X).
```

a w logice odczytuje się jako formułę

```
∀X not wnuk(mieszko,X)
```

czyli

(1) formuła *not p* jest prawdziwa w interpretacji *I* wtedy i tylko wtedy gdy formuła *p* nie jest prawdziwa w tej interpretacji;
(2) formuła *(p,q)* jest prawdziwa w interpretacji *I* wtedy i tylko wtedy gdy zarówno formuła *p* jak i *q* są prawdziwe w tej interpretacji;
(3) formuła *(p:-q)* jest prawdziwa w interpretacji *I* wtedy i tylko wtedy gdy formuła *p* jest prawdziwa w interpretacji *I* lub formuła *q* nie jest prawdziwa w tej interpretacji.

- Wartosc logiczną formuł z kwantyfikatorami określimy przy upraszczającym założeniu, że każdy element universum ma odpowiednią stałą ktora go reprezentuje. W naszym przypadku jedynymi kwantyfikowanymi formułami są klauzule programu. Niech *K* bedzie klauzulą programu logicznego. Jej instancją nazwiemy formułę otrzymaną z *K* przez zastąpienie w *K* każdej zmiennej przez pewien term. (Wielokrotne wystąpienia zmiennej zastępowane są przez wystąpienia tego samego termu). Instancję otrzymaną przez zastąpienie zmiennych przez stałe będziemy nazywali *instancją podstawową*. W przypadku nieskończonego uniwersum każda klauzula ze zmiennymi ma nieskończenie wiele instancji. Oczywiście instancje podstawowe klauzuli nie zawierają już kwantyfikatorów i ich wartości logiczne w danej interpretacji mogą być określone na podstawie uprzednio podanych reguł. Klauzula jest prawdziwa w danej interpretacji wtedy i tylko wtedy gdy wszystkie jej instancje podstawowe są prawdziwe w tej interpretacji.

Możemy już teraz sformułować podstawowe pojęcie *modelu* programu logicznego. Modelem nazywa się każda interpretacja, w której wszystkie klauzule programu są prawdziwe. Pojęcie to ma dosc oczywistą intuicję. Punktem wyjścia dla każdego programu logicznego jest pewna struktura relacyjna, którą programista stara się opisać, przypisując elementy języka obiektom tej struktury, a więc korzystając z pewnej interpretacji. Program logiczny powinien poprawnie opisywac strukturę, innymi słowy, przyjęta interpretacja powinna być modelem programu. Program jest jednak napisem w języku formalnym i ma (nieskończenie) wiele interpretacji. Nawet jeśli opisywana struktura jest jego modelem, również wiele innych interpretacji może być jego modelami.

2.3. Model standardowy programu bez negacji.

Istnienie modelu programu logicznego jest zagwarantowane przez jego składnię. Warto jednak zwrócić uwagę na pewien szczególny model ukazujący związek programów logicznych z bazami danych. Podobnie jak w programie logicznym tak i w relacyjnej bazie danych obiekty zewnętrznego świata reprezentowane są przez napisy. Najprostszym sposobem interpretowania tych napisów może być przyjęcie, że reprezentują one same siebie. W programie logicznym dziedziną takiej interpretacji byłby więc zbiór wszystkich termów bez zmiennych, które da się utworzyć z alfabetu liter występujących w programie. Litery funkcyjne reprezentują przy tym funkcje tworzące termy z termów. Takie interpretacje nazywają się interpretacjami Herbrandta. W przypadku relacyjnej bazy danych jak również każdego programu logicznego bez liter funkcyjnych dziedzina taka jest skończonym zbiorem stałych. Interpretacja musi także wiązać predykaty z relacjami. W tym przypadku są to relacje na termach i mogą być zdefiniowane poprzez podanie zbioru formuł atomowych: relacja związana z literą *p* zawiera *n*-tkę termów $<t_1,...t_n>$ wtedy i tylko wtedy gdy formuła $p(t_1,...,t_n)$ należy do tego zbioru. Można udowodnić, że jeśli dwa zbiory formuł atomowych są modelami danego programu to również i ich przecięcie jest modelem. Oznacza to w praktyce że dla każdego programu istnieje najmniejszy model Herbrandta. Jak można go skonstruowac i jaka jest jego intuicja? Oczywiscie wszystkie instancje bez zmiennych kazdej klauzuli jednostkowej musza należec do tego modelu. Następnie, jeśli $a:-b_1,..b_k$ jest instancją bez zmiennych pewnej klauzuli i $b_1,...b_k$ należą do modelu to również *a* musi do niego należec. Wychodząc ze zbioru wszystkich instancji bez zmiennych klauzul jednostkowych i dodając niezbędne elementy otrzymamy w granicy najmniejszy model. Dla przykładu z p.1 najmniejszy model ma postac:

rodzic(mieszko,bolesław), rodzic(anna,bolesław),
rodzic(mieszko,dobrogniew), rodzic(jagna,dobrogniew),
rodzic(bolesław,mieszko!!), rodzic(dobrogniew,ścibor),
wnuk(mieszko,mieszko!!), wnuk(mieszko,ścibor)
wnuk(anna,mieszko!!), wnuk(jagna,ścibor).

Relacyjna baza danych może być uważana za zbiór klauzul jednostkowych, które wyznaczają natychmiast najmniejszy model. Intuicyjnie oznacza to, że przyjmujemy tutaj tzw. "założenie zamkniętego świata": informacje których nie ma w bazie danych uważamy za nieprawdziwe. Klauzule jednostkowe programu logicznego mogą być uważane za jawną część opisywanej bazy danych. Do tego dochodzą elementy najmniejszego modelu opisywane przez pozostałe klauzule. Tak więc najmniejszy model można uważać za rozwiniętą bazę danych obejmującą wszystkie informacje zawarte w programie. Wszystkie inne informacje uważa się w nim za nieprawdziwe.

2.4. Konsekwencje logiczne programu.

Program logiczny używany jest do odpowiadania na zapytania dotyczące opisanego modelu. Zapytanie ma postać ciągu (koniunkcji) literałów. Jednakże w języku logiki nie ma zdań pytających. Jaka jest więc logiczna treść zapytania? Jest to negatywne stwierdzenie. Zadaniem systemu jest znalezienie kontrprzykładu który obalałby to stwierdzenie. W rezultacie poszukuje się więc instancji zapytania, która byłaby prawdziwa w opisanym modelu. Jednakże program "opisuje" wszystkie swoje modele a nie tylko ten jeden, o którym myślał programista. W związku z tym wymaga się by znaleziona instancja zapytania była prawdziwa we wszystkich modelach programu. W terminach logiki formuła o tej własności nazywa się *konsekwencją logiczną* zbioru formuł, w tym przypadku - programu. Pojęcie to definiuje się następująco. Formuła *F* nazywa się konsekwencją logiczną zbioru formuł *FF* jeśli każdy model *FF* jest modelem *F*. Warto dodać, że najmniejszy model Herbrandta dla programu logicznego jest zbiorem wszystkich atomowych formuł bez zmiennych które są konsekwencjami logicznymi programu. Ten wynik uściśla stwierdzenie poprzedniego paragrafu, mowiace o tym, że model ten przedstawia jawnie całą informację zawartą w programie.

3. Od logiki do programowania.

3.1. Jak szukać odpowiedzi na zapytania.

Rozważymy teraz sposób wnioskowania umożliwiający znajdowanie odpowiedzi na zapytania. Ograniczymy się najpierw do programów i zapytań bez negacji. Omówienie reguły wnioskowania używanej przez systemy programowania w logice rozpoczniemy od przykładu. Rozważmy zapytanie

"Kto jest wnukiem Mieszka?"

skierowane do programu z p.1. Ma ono postać

?wnuk(mieszko,X).

a w logice odczytuje się jako formuła

∀X not wnuk(mieszko,X)

czyli

"Dla kazdego X nieprawdą jest że X jest wnukiem Mieszka",

a więc poprostu

"Mieszko nie ma wnukow".

Skorzystamy teraz z programu by znaleźć kontrprzykład. Klauzule programu maja postac *A:-B* ("Jeśli B to A"). Klauzulę jednostkowa *A* można uważać za zdanie postaci "Jeśli *prawda* to *A* ". Można udowodnić że zdanie postaci "Jeśli *B* to *A* " jest logicznie równoważne zdaniu "Jeśli nie *A* to nie *B* ", tj. oba te zdania maja tą samą wartość logiczną w każdej interpretacji. Ponadto jeśli zdania "nie *A*" i "jesli nie *A* to nie *B* " są prawdziwe, to prawdziwe musi być również zdanie "nie *B* ". Naszym punktem wyjściowym jest zdanie

"Dla każdego *X*, *X* nie jest wnukiem Mieszka."

Odpowiednia klauzula programu mówi, że

"Jesli *X* nie jest wnukiem Mieszka to
nie jest prawdą, że Mieszko jest rodzicem *Y* i *Y* jest rodzicem *X*. ".

Wnioskujemy więc:

"Nie jest prawdą że Mieszko jest rodzicem *Y*, lub
nie jest prawda że *Y* jest rodzicem *X*."

Korzystajac z bazy danych widzimy że Mieszko jest rodzicem Bolesława, pierwsze zdanie jest więc nieprawdziwe w każdym modelu programu. Z kolei sprawdzamy w bazie danych, że Boleslaw jest rodzicem Mieszkall a więc i drugie zdanie jest nieprawdziwe, jesli *Y* to Bolesław. Z założenia, że *X* nie jest wnukiem Mieszka otrzymaliśmy sprzeczność i przy okazji stwierdziliśmy, że Mieszkoll jest wnukiem Mieszka.

W celu precyzyjnego opisania procesu wnioskowania zilustrowanego powyższym przykładem wprowadzimy pojęcie *podstawienia*. . Podstawieniem nazywa się skończony zbiór par postaci $\{X_1/t_1,\ldots,X_n/t_n\}$ gdzie X_i $i=1,..,n$ są różnymi zmiennymi a t_i jest termem różnym od X_i. Podstawienie można uważać za skończoną reprezentację funkcji przyporzadkowującej zmiennym termy. Zmienne nie występujące w podstawieniu są przyporządkowane samym sobie. Podstawienie służy do konstruowania instancji formuł i termów. Zastosowaniem podstawienia σ do formuły (termu) E nazywać będziemy instancję E otrzymaną przez zastąpienie każdego wystąpienia zmiennej przez term przyporządkowany jej przez podstawienie σ. Zastosowanie podstawienia σ do E oznaczać będziemy $E\sigma$. Niech $\sigma=\{X_1/s_1,\ldots,X_n/s_n\}$ oraz $\tau=\{Y_1/t_1,\ldots,Y_m/t_m\}$ będą podstawieniami. Ich złożeniem nazywa się podstawienie otrzymane ze zbioru $\{X_1/s_1\tau, \ldots X_n/s_n\tau, Y_1/t_1,\ldots,Y_m/t_m\}$ przez usunięcie wszystkich par $X_i/s_i\tau$ takich ze $X_i=s_i\tau$ oraz wszystkich par Y_j/t_j takich że Y_j należy do $\{X_1,\ldots X_n\}$. Mówimy, że podstawienie σ jest *bardziej ogólne* niż τ, jeśli istnieje podstawienie ρ takie ze $\sigma=\tau\rho$. Unifikatorem formuł atomowych (termów) E_1 i E_2 nazywa się podstawienie σ takie że $E_1\sigma=E_2\sigma$. Para formuł (termów) nazywa się parą *unifikowalną* jeśli istnieje unifikator tej pary elementów. Dla każdej pary unifikowalnej istnieje najogólniejszy unifikator.

Korzystajac z powyższych pojęć i wyników można teraz przedstawić ogólny schemat wnioskowania użytego w powyższym przykładzie do znalezienia wnuka Mieszka. Każdy proces wnioskowania rozpoczyna się od zapytania, które jest ciągiem formuł atomowych. W zapytaniu wybiera się jedną z formuł do dalszej analizy. Dla wybranej formuły poszukuje się w programie klauzuli której lewa strona byłaby z nią unifikowalna. Zarówno wybrana formuła atomowa jak i lewa strona klauzuli mogą zawierać zmienne. Zakłada się, że zmienne klauzuli nie występują w zapytaniu. W przeciwnym przypadku można je przemianować na nowe zmienne spełniające ten warunek. Nowe zapytanie tworzy się przez zastąpienie wybranej formuły atomowej prawą strona klauzuli z przemianowanymi zmiennymi i zastosowaniu do tak

otrzymanej formuły wyznaczonego uprzednio najogólniejszego unifikatora. Schematycznie krok wnioskowania można przedstawić za pomocą następującej reguły:

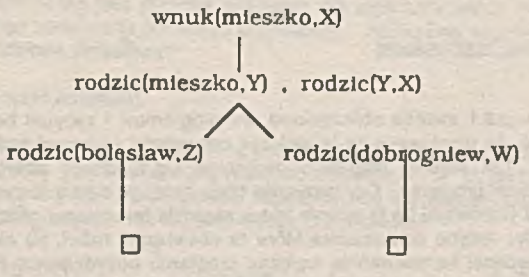
$$\frac{?A_1,...A_k,...A_n \quad A{:-}B_1,...,B_m}{?(A_1,...A_{k-1},B_1,...,B_m,A_{k+1},...A_n)\sigma}$$

gdzie $A{:-}B_1,...,B_m$ jest klauzulą programu ze zmiennymi przemianowanymi tak by nie zawierały żadnych zmiennych zapytania $A_1,...A_k,...A_n$

σ jest najogólniejszym unifikatorem A_k i A.

Wnioskowanie kończy się sukcesem po uzyskaniu w kolejnym kroku zapytania pustego. Przypuśćmy, że stało się to w kroku *n*. Oznaczmy przez $s_1,..s_n$ unifikatory otrzymane w kolejnych krokach wnioskowania. Ich złożenie $s_1...s_n$ ograniczone do zmiennych pierwotnego zapytania nazywa się *odpowiedzią obliczoną*. Wnioskowanie może się również zakończyć porażką: zachodzi to wówczas gdy lewa strona żadnej klauzuli nie da się zunifikować z formułą wybraną w danym kroku. Proces wnioskowania może być również nieskończony.

Warto zwrócić uwagę na to, że opisany schemat wnioskowania jest niedeterministyczny: zapytanie może składać się z wielu formuł atomowych i lewa strona wielu klauzul może unifikować się z wybraną formułą atomową. W praktyce dążymy do tego aby znaleźć wszystkie odpowiedzi na zadane pytanie. Czy jest to możliwe? Teoretycznie tak, ale względy efektywności systemu zmuszają do rezygnacji ze spełnienia tego postulatu. Problem szukania odpowiedzi może być przedstawiony jako systematyczne przeszukiwanie naogół nieskończonego drzewa. Konstrukcję tego drzewa rozpoczniemy od wprowadzenia deterministycznej reguły wyboru formuły atomowej z zapytania. Reguła taka może być funkcją na zapytaniach. Niekiedy wybór uzależnia się dodatkowo od historii obliczenia. Regułę taką nazwiemy *regułą obliczenia*. Odpowiedni wynik teoretyczny mówi, że znalezienie wszystkich odpowiedzi jest możliwe bez względu na przyjętą regułę obliczenia. Prosta reguła obliczenia przyjęta w języku Prolog polega na wybieraniu pierwszej formuły atomowej zapytania. Dla ustalonej reguły obliczenia, programu i zapytania początkowego zdefiniujemy *drzewo obliczeń*, zwane także *drzewem SLD* (Rys. 1).



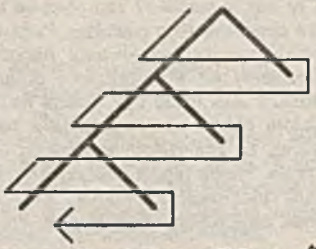
Rys.1. Drzewo SLD

Wierzchołkami drzewa są zapytania a każdą krawędź jest przyporządkowana klauzuli programu. Korzeń drzewa jest zapytaniem początkowym. Krawędzie wychodzące z każdego wierzchołka odpowiadają klauzulom, których lewa strona unifikuje się z formułą atomową wybraną w danym wierzchołku. Każda taka krawędź prowadzi do nowego zapytania otrzymanego w wyniku unifikacji. Problem poszukiwania odpowiedzi na dane zapytanie sprowadza się teraz do problemu przeszukiwania drzewa obliczeń określonego przez to zapytanie i wybraną regułę obliczeń. Każda istniejąca odpowiedź przedstawiona jest w drzewie przez skończoną gałąź zakończoną zapytaniem pustym. Tak więc teoretycznie każdą z odpowiedzi można osiągnąć w

skończonym czasie. Drzewo może mieć jednak również gałęzie nieskończone. Nawet w tym przypadku przeszukiwanie drzewa wszerz jak to wyjaśnia rys. 2b gwarantuje znalezienie każdej odpowiedzi. Okazuje się jednak, że metoda przeszukiwania wszerz jest bardzo nieefektywna. Z tego powodu istniejące implementacje Prologu stosują metodę przeszukiwania wgłąb, wyjasnioną na rys.2a. W tym przypadku jednak osiągnięcie każdej odpowiedzi w skończonym czasie nie jest zagwarantowane: przeszukiwanie wgłąb gałęzi nieskończonej drzewa może rozpocząć się przed znalezieniem istniejacych odpowiedzi.

a)

b)



Rys.2 . Metody przeszukiwania drzewa SLD: a) wgłąb; b)wszerz

3.2. Zupełność metody obliczeniowej

Przedstawiona w p.3.1 metoda obliczeniowa dla programow i zapytań bez negacji jest poprawna w tym sensie, że uzyskiwane za jej pomocą odpowiedzi są prawdziwe we wszystkich modelach programu. Tak więc w rezultacie otrzymuje się instancję zapytania, które są konsekwencjami logicznymi programu. Czy wszystkie takie instancje da się otrzymać za pomocą przedstawionej metody? Odpowiedź na to pytanie jest w zasadzie twierdząca; własność tą nazywa się *zupełnością* metody. Jedyne ograniczenie które tu obowiązuje mówi, że niekiedy zamiast formuły atomowej *A* będącej konsekwencją logiczną programu otrzymuje się bardziej ogólną formułę , której instancją jest *A*. Jeśli na przykład powiemy "wszyscy są zadowoleni", co zapiszemy:

```
zadowolony(X).
```

a ponadto wiadomo, że w grupie mamy studentów Kowalskiego i Kwiatkowskiego:

```
student(kowalski).
student(kwiatkowski).
```

to konsekwencją logiczną programu beda formuły: zadowolony(kwiatkowski) i zadowolony(kowalski). Jednakże nasza metoda obliczeniowa na zapytanie "kto jest zadowolony?"

? zadowolony(Y).

nie potrafi odpowiedzieć X=kowalski. Obliczenie zwiąże bowiem zmienną Y ze zmienna X co oznacza, że kazdy Y jest zadowolony.

Szczególnym przypadkiem drzewa jest skończone drzewo obliczenia nie zawierające żadnej odpowiedzi. Wszystkie liscie takiego drzewa sa formułami atomowymi, ktore nie unifikuja się z żadnymi klauzulami programu. Drzewo takie będziemy nazywac *drzewem porażek*. Przypuśćmy, że drzewo obliczeniowe jakiegoś zapytania jest drzewem porazek. Zupełnosc metody obliczeniowej gwarantuje wowczas że żadna instancja tego zapytania nie jest konsekwencja logiczną programu. Innymi słowy, oznacza to, że na zadane pytanie nie ma pozytywnej odpowiedzi. W takich przypadkach system Prologu odpowiada "no".

3.3. Kłopoty z negacją

Dotychczas rozważaliśmy programy i zapytania bez negacji. W praktyce negacja jest jednak często potrzebna. Wróćmy do przykładowego zapytania z negacja z p.1. (Kto jest wnukiem Mieszka a nie jest synem Bolesława). System Prologu znajduje odpowiedź na to pytanie w następujący sposób. Pierwszą część zapytania nie ma negacji i odpowiedź na nią uzyskuje się za pomocą metody opisanej w p.3.1. Pierwszą uzyskaną odpowiedzią jest *X=mieszkoII*. Następne zapytanie jest negatywne: czy MieszkoII nie jest synem Bolesława? Zamiast tego system zapytuje "Czy MieszkoII jest synem Bolesława?" i uzyskuje twierdzaca odpowiedz, ktora wyklucza to rozwiązanie. W związku z tym następuje próba znalezienia innej odpowiedzi na pierwszą część zapytania. Tą odpowiedzią jest *X=Scibor*. Kolejne zapytanie "czy Scibor jest synem Bolesława?" nie uzyskuje twierdzacej odpowiedzi. Drzewo obliczenia jest w tym przypadku drzewem porażek i system odpowiada "nie" , co oznacza pozytywna odpowiedź na pytanie "czy Scibor nie jest synem Bolesława?" i w rezultacie uzyskanie odpowiedzi na cale zapytanie.

Powyższy przykład ilustruje następująca metodę obliczeń stosowaną w Prologu do zapytań z negacją. Każdy negatywny literał *not A* wybrany przez regułę obliczeniową daje początek nowemu obliczeniu dla zapytania *A*. Jeśli to obliczenie odniesie porażkę traktuje się ją jako sukces obliczenia dla *not A* i wznawia się poprzednio zatrzymany proces obliczeniowy. Sukces obliczenia dla *A* traktuje się jako porażkę obliczenia dla zapytania *not A*. Przypadek sukcesu wymaga dodatkowego komentarza popartego przykładem. Rozważmy negatywne zapytanie "Czy Mieszko nie jest rodzicem Bolesława?"

?not rodzic(mieszko,boleslaw).

Zgodnie z opisaną metodą obliczeniową problem zostanie sprowadzony do poszukiwania odpowiedzi na zapytanie "Czy Mieszko jest rodzicem Bolesława?" . Uzyskana odpowiedz "tak" prowadzi do odpowiedzi "nie" na oryginalne zapytanie. Co się jednak stanie gdy oryginalne zapytanie zawiera zmienne, np. "Kto nie jest rodzicem Bolesława?"

? not rodzic(X,boleslaw).

W tym przypadku pomocnicze zapytanie ?rodzic(X,boleslaw). odnosi sukces, co nie znaczy, ze należy odpowiedzieć "nie" na oryginalne zapytanie. Przykład pokazuje,że zaproponowana metoda działa zgodnie z naszymi intencjami tylko w przypadku gdy zanegowane zapytanie nie zawiera zmiennych.

Opisana wyżej metoda obliczeniowa nosi nazwę "negacja przez porażkę" i stanowi podstawę implementacji programów z negacją w Prologu. Powstaje jednak pytanie czy uzyskiwane w ten sposób odpowiedzi są poprawne logicznie, a więc czy są prawdziwe we wszystkich modelach programu. Okazuje się, że tak nie jest. Rozważmy ponownie odpowiedź "Scibor" na zadane uprzednio zapytanie "Kto jest wnukiem Mieszka a nie jest synem Bolesława?" Dodajmy do

najmniejszego modelu Herbrandta formuły atomowe rodzic(bolesław,scibor), wnuk(anna,scibor). Łatwo sprawdzić, ze otrzymana w ten sposob interpretacja jest również modelem programu. W modelu tym zdanie "Scibor nie jest synem Boleslawa" jest nieprawdziwe. Oczywiście nasza nowa interpretacja nie jest na pewno modelem opisywanym przez programistę: Scibor ma w niej dwóch ojców. Jeśli jednak chcemy korzystac z odpowiedzi konstruowanych przez system musimy w jakis sposob uzasadnić ich logiczną poprawnosc. Takie uzasadnienie wiąże się z problematyką baz danych. W przypadku relacyjnej bazy danych przyjmuje się naogół, że informacje nie zawarte w bazie są nieprawdziwe. W przypadku programu logicznego bez negacji "informacje zawarte w programie" mozna utożsamić z najmniejszym modelem Herbrandta, który jest jednocześnie zbiorem wszystkich formuł atomowych bez zmiennych, które są konsekwencjami logicznymi programu. Te rozważania prowadzą do postulatu, by zamiast rozważać wszystkie modele programu, ograniczyć się z góry do pewnej klasy modeli, lub nawet do jednego modelu. W przypadku programow bez negacji, dopuszczających zapytania z negacja byłby to najmniejszy model Herbrandta. Pewna klasa programow z negacja umożliwia rownież konstrukcję "naturalnego" modelu. Wykracza to jednak poza ramy tego wykładu. Bardziej rozpowszechnione jest jednak inne podejście do problemu negacji. Oparte jest ono na obserwacji, że w mowie potocznej zdanie warunkowe używane jest często zamiast zwrotu "wtedy i tylko wtedy". Na przykład mowiąc "Jeśli będzie pogoda pojadę na wycieczkę." mam zwykle na myśli stwierdzenie "Pojadę na wycieczkę wtedy i tylko wtedy gdy będzie pogoda". Program logiczny jest zbiorem zdań warunkowych. Cóż się stanie, jeśli potraktujemy go jako zbiór zdań typu "wtedy i tylko wtedy" ? Wymaga to przede wszystkim pewnego przekształcenia programu, gdyż ten sam predykat występuje naogół w lewych stronach wielu klauzul. Różne klauzule opisują z zasady alternatywne przypadki, które w łącznym opisie powinny byc połączone spojnikiem "lub". Nie wdając sie w formalny opis tej transformacji zilustrujemy ją na przykładzie rozważanego uprzednio programu. Program zawiera dwa predykaty "rodzic" i "wnuk". Kazdy z nich ma być teraz określony za pomocą jednego zdania typu "wtedy i tylko wtedy". Definicja predykatu "wnuk" nie nastręcza żadnych trudności, gdyż ma on tylko jedna klauzulę, w której wystarczy zamienić implikację na spójnik "wtedy i tylko wtedy". Mamy więc:

"Z jest wnukiem X wtedy i tylko wtedy gdy X jest rodzicem Y i Y jest rodzicem Z . "

Teraz klauzula programu

```
wnuk(X,Z) :- rodzic(X,Y),rodzic(Y,Z).
```

będzie uważana za zapis powyższego zdania. Pozostały predykat "rodzic" jest opisany przez wiele klauzul, które łącznie odczytujemy jak następuje:

"X jest rodzicem Y wtedy i tylko wtedy gdy
X=mieszko i Y = bolesław, lub
X=anna i Y=bolesław,lub
itd."

Zauważmy, ze przy tej konstrukcji posługujemy się nowym predykatem = .
Formalnie, transformacja programu wprowadza jeszcze dodatkowe zdania opisujące ten nowy predykat, które tu jednak pominiemy. Zbiór zdań uzyskany w wyniku transformacji pierwotnego programu nazywa sie *uzupełnieniem* tego programu. Transformację stosuje się nie tylko do programow bez negacji, jak w powyższym przykładzie, lecz do wszystkich programów normalnych. Służy ona do objaśnienia w terminach logiki znaczenia odpowiedzi konstruowanych za pomocą przedstawionej metody obliczeniowej. Okazuje sie bowiem ze instancje zapytania konstruowane w ten sposob są konsekwencjami logicznymi uzupełnienia programu. Intuicyjnie odpowiedzi odnoszą się więc do przypadku gdy zdania warunkowe programu odczytuje się jako zdania typu "wtedy i tylko wtedy gdy". Wynik ten nie rozwiązuje jeszcze wszystkich kłopotów związanych z negacją, gdyż zakłada, że reguła obliczeniowa nie może wybrać zanegowanego zapytania ze zmiennymi. W implementacjach Prologu założenie to nie jest naogół przestrzegane, co może prowadzić do niepoprawnych logicznie odpowiedzi. Przyjęcie tego założenia jest (jedną) z przyczyn dla których metoda obliczeniowa dla programów z negacją nie jest zupełna. Ilustruje to przykład zapytania "Kto nie jest rodzicem Bolesława?" Zapytanie to ma wiele poprawnych

odpowiedzi, lecz nie da się ich skonstruować za pomocą opisanej metody.

3.4. Od logiki do Prologu

Programy normalne wraz z opisaną wyżej metodą obliczeniową stanowią jądro języka Prolog. Ten rozdział zawiera skrótowe omówienie niektórych innych elementów tego języka.

3.4.1. Odcięcie.

Często się zdarza, że drzewo obliczenia zawiera wiele gałęzi porażek a tylko jedną lub niewiele odpowiedzi. Zaniechanie przeszukiwania takiego drzewa po znalezieniu wszystkich odpowiedzi może znacznie usprawnić proces obliczeniowy. Język Prolog daje taką możliwość w postaci konstrukcji zwanej odcięciem. W celu wyjaśnienia tej konstrukcji odwołamy się do pojęcia drzewa obliczenia. Etykietami wierzchołków drzewa obliczenia są zapytania. W etykiecie każdego wierzchołka N wyróżniony jest jeden literał A, wybrany w danym kroku przez regułę obliczeniową w celu znalezienia odpowiedniej klauzuli. Przypuśćmy, że literał A nie występował w zapytaniu początkowym . Wobec tego jest on instancją literału B_j występującego w ciele pewnej klauzuli $H :- B_1,...,B_j,...,B_n$. Lewa strona tej klauzuli została zunifikowana z literałem wyróżnionym pewnego wierzchołka N' w gałęzi łączącej korzeń drzewa z wierzchołkiem N. Wierzchołek N' będziemy nazywać *wierzchołkiem pochodzenia* literału A. Konstrukcja zwana odcięciem może wystąpić w klauzulach programu jako specyficzny literał postaci !. System Prologu konstruuje drzewa obliczenia metodą wgłąb od lewej do prawej. Każda krawędź odpowiada zastosowaniu pewnej klauzuli do wyróżnionego literału. Porządek krawędzi określony jest przez porządek tekstowy klauzul w programie. Osiągnięcie liścia drzewa powoduje nawrót procesu poszukiwania i powrót do poprzedniego wierzchołka (Rys. 2a). Jeśli wychodzą z niego nieodwiedzone dotychczas gałęzie, następuje wznowienie przeszukiwania wgłąb. Wprowadzenie odcięcia powoduje modyfikację tego procesu. Przy przeszukiwaniu wgłąb literał odcięcia wybrany w jakimś wierzchołku odnosi natychmiastowy sukces i jest pomijany. Jednakże nawrót do tego wierzchołka powoduje pominięcie w procesie przeszukiwania jego nieodwiedzonych gałęzi. Zamiast tego następuje natychmiastowe przejście do wierzchołka jego pochodzenia. Zastosowanie odcięcia ilustruje następujący przykład. Dodajmy do przykładu rodziny następujące klauzule.

```
matka(X,Y) :- rodzic(X,Y),kobieta(Y).
kobieta(anna)
kobieta(jagna).
```

Drzewo obliczenia dla zapytania "Kto jest matką Bolesława?" zawiera gałąź porażki związaną z zapytaniem "Czy Mieszko jest kobietą?". Gałąź tą można wykluczyć z procesu przeszukiwania umieszczając odcięcie w definicji relacji "matka":

```
matka(X,Y) :- rodzic(X,Y),kobieta(Y),!.
```

3.4.2. Listy

Wiele językow programowania używa pojęcia listy jako standardowej struktury danych. Podstawową strukturą danych w programach logicznych są termy. Listy można łatwo reprezentować jako termy. Ponieważ jednak w wielu zastosowaniach korzysta się z list, Prolog wprowadza specjalną standardową notację dla list, uwalniając użytkownika od wprowadzenia

własnej reprezentacji. Lista pusta jest w Prologu reprezentowana jako []. Każda niepusta lista ma jeden element wiodący, dołączony do pozostałej części, która jest listą. Tak więc, jednoelementową listę z elementem *a* można uważać za term .(a ,[]) gdzie . jest dwuargumentową literą funkcyjną konstruującą listy. W notacji Prologu term ten można przedstawić jako [a |[]], lub jako [a]. Ogólnie, *n*-elementową listę o elementach $a_1,...,a_n$ można przedstawić jako $[a_1,...,a_n]$ lub $[a_1 |[a_2,...,a_n]]$ lub $[a_1,a_2, |[a_3,..,a_n]]$ ...lub $[a_1,..,a_n |[]]$. Tak więc | pełni rolę operatora konkatenacji, który dopisuje na początku istniejącej listy poprzedzający go ciąg elementów. Ta konwencja notacyjna została użyta we wprowadzającym przykładzie sortowania p.1.

Lispowe operacje na listach mogą być zdefiniowane w Prologu tylko jako relacje, np.:

```
car([X|Xs],X).
cdr([X|Xs],Xs).
```

3.4.3. Arytmetyka

Operacje arytmetyczne mogą być łatwo zdefiniowane za pomocą programów logicznych. Jednakże w praktyce nie jest to wygodne z różnych powodów. Aby to wyjaśnić spróbujmy opisać dodawanie i mnożenie na liczbach naturalnych za pomocą programu logicznego. Obiekty universum muszą być reprezentowane za pomocą termów. Niech 0 reprezentuje liczbę zero a funktor s funkcję następnika. Liczby naturalne będą więc reprezentowane przez termy 0, s(0),s(s(0)),....,itd. Operacje dodawania i mnożenia są funkcjami ale program logiczny pozwala definiować tylko relacje. Wobec tego dwuargumentowe funkcje będziemy opisywać jako trójargumentowe relacje, w których trzeci argument jest wynikiem operacji na dwóch pierwszych argumentach. Wiadomo, że operację dodawania i mnożenia spełniają następujące aksjomaty Peano:

```
0+X=X
 s(X)+Y = s(X+Y)
 0*X=0
 s(X)*Y=X*Y+Y
```

W zapisie relacyjnym przyjmują one następującą postać:

```
plus(0,X,X).
  plus(s(X),Y,s(Z)) :- plus(X,Y,Z).
 razy(0,X,0).
  razy(s(X),Y,Z) :- razy(X,Y,W),plus(W,Y,Z).
```

Powyższy program może być używany w Prologu. Na przykład, aby dodać 2 i 3 trzeba zapytać ?plus(s(s(0)),s(s(s(0))),X). a otrzymana odpowiedź będzie X=s(s(s(s(s(0))))). Program może być również używany do odejmowania i dzielenia, np. ?plus(X,s(0),s(0)) da w odpowiedzi X=0. Można go również używać do obliczeń symbolicznych, np. dodając 2 do jakiejkolwiek liczby X (?plus(s(s(0)),X,Y)) otrzymamy Y = s(s(X)). Jednakże taka realizacja arytmetyki jest w praktyce nieprzydatna. Zauważmy, że:

- reprezentacja liczb jest nieczytelna;
- obliczenia nie korzystają z arytmetyki komputera i są dlatego bardzo nieefektywne, np. dodanie liczb *n* i *m* wymaga *n*+1 kroków obliczenia;
- relacyjny zapis operacji uniemożliwia tworzenie wyrażeń arytmetycznych, takich jak w innych językach. Na przykład aby obliczyć wartość wyrażenia 2+3*4 trzeba samemu wprowadzać nowe zmienne reprezentujące wartość podwyrażeń; odpowiednie zapytanie miałoby postać:

```
?razy(s(s(s(0))),s(s(s(s(0)))),X),plus(X,s(s(0)),Y).
```

Prolog daje dostęp do arytmetyki maszynowej, wprowadzając wyrażenia arytmetyczne i specjalne konstrukcje języka umożliwiające obliczanie ich wartości. Liczby naturalne są reprezentowane przez numerały, takie jak 0, 1, 1989, podobnie jak w innych językach programowania. Z punktu widzenia logiki numerały są stałymi. Wprowadza się ponadto szereg operacji arytmetycznych. Są one reprezentowane przez standardowe litery funkcyjne, takie jak +,-,*,/,mod używane w notacji infiksowej. Tak więc można tworzyć wyrażenia arytmetyczne, np. X+1 , 2*3-5*X itp. Z punktu widzenia logiki wyrażenia te są termami. Dostęp do arytmetyki maszyny dają predykaty standardowe, z których najczęściej używany jest predykat *is* oraz predykaty równości i nierówności. Na przykład podwojenie maksymalnej z dwóch danych liczb X i Y może być opisane w Prologu w następujący sposób:

```
p(X,Y,Z) :- X<Y, Z is 2*Y.
p(X,Y,Z) :- Z is 2*X.
```

Powyższy program korzysta z faktu, że klauzule będą używane w kolejności ich wystąpienia w programie. Pierwsza klauzula spowoduje porównanie liczb X i Y. W momencie wywołania tego zapytania zmienne X i Y muszą być związane z wyrażeniami arytmetycznymi bez zmiennych. Niespełnienie tego warunku spowoduje sygnalizację błędu i przerwanie obliczeń. Jeśli warunek jest spełniony to wartości wyrażeń będą obliczone i porównane. Niespełnienie nierówności powoduje porażkę i po nawrocie próbę zastosowania drugiej klauzuli. W kolejnym kroku obliczenia wybranym literałem będzie Z is 2*Y. Konstrukcja standardowa *is* powoduje obliczenie wartości wyrażenia 2*Y i związanie jej ze zmienną Z. (Spełnienie warunku na aktualne wiązanie zmiennej Y jest też wymagane ale w tym przykładzie jest zagwarantowane poprzez wcześniejsze użycie zmiennej Y).

3.4.4. Inne konstrukcje Prologu

Prolog ma jeszcze wiele innych konstrukcji. Należą do nich konstrukcje:
- wprowadzania i wyprowadzania danych;
- przekształcania programu;
- badania aktualnych wiązań zmiennych;
- działania na zbiorach odpowiedzi

i in.

Różne implementacje Prologu mogą jeszcze zawierać dodatkowe specyficzne konstrukcje. Bardzo przydatnym mechanizmem dostępnym w wielu systemach Prologu jest mechanizm gramatyczny zwany gramatyką klauzul definitywnych DCG. Formalizm DCG jest rozszerzeniem formalizmu gramatyk bezkontekstowych, używanego powszechnie do opisu składni języków programownia. Reguły DCG mogą występować w programie Prologu i są automatycznie tłumaczone na równoważne klauzule, które umożliwiają zarówno analizę syntaktyczną jak i generowanie konstrukcji języka opisanego przez podaną gramatykę. Proces analizy syntaktycznej bądź generacji może być wszczęty poprzez proste zapytanie.

Przedstawienie tych konstrukcji wykracza poza ramy niniejszego opracowania.

4. Aspekty praktyczne.

Do końca lat siedemdziesiątych Prolog był raczej przedmiotem badań niż językiem programowania. Do tej pory zainteresowanie przemysłu programowaniem w logice jest dość ograniczone. Z drugiej strony wiadomo, że do sprzedaży trafia coraz większa liczba coraz bardziej zaawansowanych komercjalnych systemów Prologu, co zdawałoby się świadczyć o rosnącym zapotrzebowaniu. Również liczba podręczników i monografii na temat programowania

w logice rośnie bardzo szybko, przy czym niektóre z nich mają już wznowienia, np. [1], [3], [5].

Wydaje się, że główną zaletą programowania w logice jest wysoki poziom abstrakcji. Oznacza to, że nie ma istotnej różnicy między programem a formalnym opisem problemu. W szczególności, program logiczny nie zawiera w zasadzie (z wyjątkiem odcięcia) konstrukcji określających kolejność wykonywania kroków obliczeniowych. Ogólnie panuje zgodna opinia, że program w Prologu jest z reguły znacznie krótszy niż odpowiedni program w jakimkolwiek innym języku programowania. Jednocześnie czas sporządzenia takiego programu jest często znacznie krótszy. Za wysoki poziom języka płaci się stosunkowo niską efektywnością obliczeń i dużym zapotrzebowaniem na pamięć. Według opracowania [8] dla typowych problemów czas wykonywania skompilowanego programu Prologu (Quintus Prolog) jest 2-4 razy dłuższy niż odpowiedniego programu w języku algorytmicznym. Nawet jeśli te dane są optymistyczne, to panuje ogólna opinia o przydatności Prologu do szybkiego konstruowania prototypów projektowanych systemów.

Wydaje się, że przydatność Prologu jest różna dla różnego typu zastosowań. Według [8] Prolog stwarza duże możliwości w następujących dziedzinach:

- Inżynieria oprogramowania i projektowanie sprzętu; szczególną zaletą Prologu jest tutaj łatwość programowania analizy syntaktycznej w oparciu o DCG, łatwość porównywania wzorców oraz możliwość formułowania i korzystania z reguł. Przykłady zastosowań wymienione w [8] zawierają:
  . Różne kompilatory Prologu,
  . Kompilatory języka ADA,
  . System transformacji programów Cobolu w programy strukturalne Cobolu
  . System AUNT (A Universal Netlist Translator) tłumaczący między sześcioma różnymi językami do opisu układów(Microelectronics Center of North Carolina MCNC),
  . System VERIFY do badania czy projekt układu jest zgodny ze specyfikacją (Schlumberger Palo Alto).
  . Programy Prologu użyte do projektowania 32 bitowego mikroprocesora CLIPPER przez firmę Fairchild. Przy projektowaniu układu sądzono że wstępny projekt nie ma żadnych błędów, jeśli chodzi o zależności czasowe. Napisano jednak w Prologu system testujący. Zajęło to dwa dni. Program wykrył błędy w projekcie i od tego czasu jest stale stosowany do innych projektów. Assembler dla CLIPPERa był początkowo napisany w C lecz zawierał błędy i trudno go było poprawić. Napisano więc inny assembler w Prologu, co zajęło również dwa dni, znacznie mniej niż praca nad assemblerem w C. Grupa zajmująca się testowaniem arytmetyki CLIPPERA napisała w Prologu kilkustronnicowy generator testów , który wyprodukował 35000 linii kodu assemblera obejmującego wszystkie przypadki do testowania. Używając tego kodu znaleziono 3 błędy w projekcie.

- Zastosowania w bazach danych.

Tutaj konkurencją dla Prologu są tzw. języki 4 generacji. Prolog ustępuje im pod względem efektywności przewyższa je jednak elastycznością. Zastosowanie Prologu wydaje się celowe, jeśli wymaga się dokonania złożonej analizy zawartości bazy danych. Prolog góruje tutaj nad innymi językami tym, że sam jest językiem relacyjnym, co ułatwia formułowanie problemu. Wymienione w [8] przykłady wdrożonych zastosowań obejmują:
.System analizy rynku na podstawie bazy danych o klientach,
.System sterowania taśmą produkcyjną na podstawie zamówień,
.System wykrywania oszustw dla bazy danych z transakcjami.

-Zastosowania do Systemów Ekspertyzy

Podstawą takich systemów jest wiedza uzyskana od ekspertów i przedstawiona w postaci reguł. Zadaniem systemu jest odpowiadać na postawione pytania korzystając z reguły. W tym celu system musi umieć wyciągać wnioski z reguł, a więc mieć wbudowany pewien mechanizm wnioskowania. Prolog, mając wbudowany ogólny system wnioskowania i mechanizm mechanizm sterowania jest dobrze przystosowany do tego rodzaju zadań. Dodatkową zaletę stanowi łatwość budowania alternatywnych systemów wnioskowania w oparciu o technikę

metainterpretacji (np. budowanie interpretera Prologu w Prologu). Łatwo jest również skonstruować specjalny język do porozumiewania się systemu z użytkownikiem, korzystając z mechanizmu DCG. Wymienione w [8] przykłady są bardzo liczne i dotyczą głównie rynku skandynawskiego. Obejmują one m. in:

.System do ciągłego nadzoru nad turbogeneratorem wykonany na zamówienie firmy ASEA;

.System do poszukiwania przyczyn wadliwego działania pomp cieplnych dla firmy VVS;

.System do lokalizacji błędów w małych centralach telefonicznych, stosowany przez Televerket, Szwedzki Urząd Telefonów;

.System ciągłego nadzoru okrętowych silników Diesla; system analizuje dane w czasie pracy silnika i w razie potrzeby ostrzega o nieprawidłowościach diagnozując ich przyczyny. System został napisany w języku Turbo-Prolog, i korzysta z procedur obliczeniowych w Fortranie. System został uruchomiony w półtora miesiąca. Pracowali nad nim 3 specjaliści w dziedzinie silników , przy czym żaden z nich nie znał uprzednio Prologu i czas pracy nad systemem obejmuje także naukę Prologu.

Prolog nie nadaje się do zastosowań :

- które wymagają głównie złożonych obliczeń numerycznych,
- dla których czas wykonania jest czynnikiem krytycznym,
- które wymagają by program miał bezpośredni nadzór nad pamięcią.

5. Niektóre kierunki badań.

Badania nad programowaniem w logice dążą w swym głównym nurcie do rozwiązania problemów zasygnalizowanych w poprzednich punktach. Dość dobry obraz aktualnych kierunków badań dają prace dorocznych konferencji nt. programowania w logice, np. [4] oraz roczniki czasopism Journal of Logic Programming i New Generation Computing.

5.1. Dążenie do zwiększenia efektywności wykonywania programów.

Próby w tym kierunku dotyczą ulepszenia kompilatorów, poprzez lepszą analizę statyczną programu oraz optymalizację kodu. Nadzieje pokłada się również w implementacjach na istniejących maszynach równoległych oraz w konstrukcji komputerów, których architektura byłaby dostosowana do mechanizmów obliczeniowych programowania w logice.

5.2. Ulepszanie języków programowania w logice.

Prolog jest językiem relacyjnym. Utrudnia to definiowanie i korzystanie z funkcji. Proponuje się różne języki łączące cechy programowania w logice z cechami języków funkcyjnych, takich jak LISP, ML lub Miranda. Bada się podstawy logiczne łączenia programów logicznych z procedurami napisanymi w innych językach programowania.

Programy logiczne stanowią tylko pewien podzbiór logiki predykatów. Proponuje się języki programowania w pełnej logice predykatów pierwszego rzędu, a nawet rozszerzenia wyższych rzędów. Bada się sposoby realizacji negacji w programach logicznych, które usunęłyby trudności omawiane w p.3.3.

Programy logiczne nie dają możliwości opisywania współdziałających procesów. Rozwija się języki programowania podobne do Prologu, lecz zawierające koncepcje procesu. Typowymi reprezentantami takich języków są Concurrent Prolog [9] i Parlog [2].

5.3. Badania nad dedukcyjnymi bazami danych

Bada sie tutaj zwiazki pomiedzy programowaniem w logice i relacyjnymi bazami danych. Dąży się do opracowania efektywnych metod obliczeniowych dla programow logicznych o bardzo dużej liczbie klauzul jednostkowych. Szczególną uwagę poświęca się programom bez litor funkcyjnych. Wiele podstawowych prac poświęconych teoretycznym aspektom dedukcyjnych baz danych mozna znaleźć w [7].

5.4. Metodologia programowania

Aczkolwiek program logiczny może być uważany za specyfikację, to jednak niekoniecznie opisuje on to co zamierzał opisać programista. Tradycyjna problematyka poprawnosci programu powraca tutaj w nieco innym ujęciu. Badania dotyczace problematyki poprawnosci obejmują dowodzenie poprawności, transformacje programów logicznych oraz metody lokalizacji błędów.

6. BIBLIOGRAFIA

[1] Clocksin.W., Mellish,C.: *Programming in Prolog* (3rd edition) Springer-Verlag 1987

[2] Gregory S.: *Parallel logic programming in PARLOG.* International Series in Logic Programming, Addison-Wesley, 1987

[3] Kluzniak,F.,Szpakowicz,S. *Prolog for Programmers*, Academic Press 1985 (pierwsza wersja ukazała sie w języku polskim)

[4] Levi G., Martelli M. (eds.): *Logic Programming*, Proc. 6th Int. Conf., MIT Press ,1989

[5] Lloyd,J. *Foundations of Logic Programming* (2nd edition), Springer-Verlag 1986

[6] Maluszynski, J. Nilsson,U.: *From Logic to Programming*, Wiley and Sons, (In print)

[7] Minker, J. (ed) : *Foundations of Deductive Databases and Logic Programming*, Morgan Kaufmann 1987

[8] Ostlund, G.: *Industriella tillampningar av logikprogrammering*, Mekanforbundet, Stockholm 1988

[9] Shapiro, E.Y. *Concurrent Prolog: Collected Papers*, Vol.1-2, MIT Press, 1988

[10] Sterling.L, Shapiro.,E. *The Art of Prolog*. MIT Press (1986)

Jesienna Szkoła PTI
Mrągowo, listopad 1989

System TEX w ręku autora, wydawcy i drukarza

Marek Ryćko
Instytut Podstaw Informatyki
Polskiej Akademii Nauk
PKiN, skr. poczt. 22
00-901 Warszawa

1. Wstęp

TEX (wym. *tech*) jest profesjonalnym systemem służącym do wykonywania składu drukarskiego. Napiszę o tym więcej w dalszej części pracy. Chciałbym zacząć od opowiedzenia fascynującej, moim zdaniem, historii jego powstania.

W latach sześćdziesiątych nie tylko w Polsce, ale również na całym świecie, dominującą techniką druku była *typografia*, polegająca na odbijaniu stron z matryc, które w całości lub linijkami odlewano z metalu. Była to technika niezbyt różniąca się od składania takich matryc z osobnych czcionek.

Profesor Donald Knuth, wybitny matematyk i informatyk amerykański, pracował wówczas nad swoim wielkim dziełem *Sztuka programowania komputerów* (*The Art of Computer Programming*). Całość zaplanowana była na siedem tomów. Kiedy pierwsze trzy z nich były już wydane i trwały prace nad czwartym, do drukarń amerykańskich zaczęła się wkradać rewolucja. Rezygnowano z typografii i wprowadzano technikę fotoskładu, polegającą na naświetlaniu promieniem lasera odpowiednich punktów kliszy fotograficznej i tworzeniu w ten sposób obrazu całej strony tekstu. Powstała klisza wykorzystywana była potem w dalszych etapach procesu drukarskiego.

Sama zmiana techniki nie wystarczała jednak do tego, żeby powstające książki były ładniejsze, niż dotychczas. Knuth przeraził się, kiedy zobaczył dostarczone mu do korekty złożone nową metodą drugie wydanie któregoś z tomów jego dzieła. Szczególnie niezadowolony był z powodu niewłaściwie dobranych odstępów, zwłaszcza w formułach matematycznych. Komputery (a właściwie niedoskonałe programy), które zastąpiły fachowców od składu, nie umiały poradzić sobie z tak subtelnym zadaniem jak odpowiednie spacjowanie. Ponadto kształty liter i innych znaków drukarskich, użytych do złożenia książki, Knuth określił jako „okropne", w porównaniu z tymi, których użyto do złożenia pierwszego wydania. Autor *Sztuki programowania* nie wyobrażał sobie, iż ten oraz dalsze tomy jego dzieła mogłyby być tak wydane.

Wtedy zaczęła się jego wielka przygoda poligraficzna. Knuth przerwał prace nad *Sztuką programowania* i poświęcił najbliższe dziewięć lat swojego życia na zrobienie systemu komputerowego, który sterowałby wykonywaniem składu drukarskiego. W ten sposób powstał TEX. Po zakończeniu prac nad TEXem twórca powrócił do przerwanego zajęcia.

W swoich wspomnieniach Knuth opisał dwa główne cele, jakie mu przyświecały, kiedy przystępował do prac nad TEXem. Pierwszym z nich było dojście do najwyższej jakości, jaką kiedykolwiek osiągnięto w drukowaniu dokumentów. Drugim celem było stworzenie takiego systemu, który byłby niezależny od postępów w technice drukarskiej. Chodziło o to, aby mimo nowych odkryć, niezmieniony system mógł być używany nawet po stu latach.

Niełatwo jest stwierdzić, czy Knuthowi udało się osiągnąć zamierzone cele. Żeby docenić w pełni TEXa trzeba posiadać sporą wiedzę poligraficzną. Zdaniem fachowców powstanie tego systemu jest równie ważnym wydarzeniem, co wynalazek Gutenberga.

2. Podstawowe wiadomości o TEXu

System TEX jest programem typu wsadowego. Żeby go użyć należy najpierw przygotować plik z tekstem przeznaczonym do złożenia oraz dodatkowymi komendami wyrażonymi w specjalnym języku. Komendy te informują o dodatkowych cechach składu. TEX czyta tak przygotowane dane i generuje drugi plik, gdzie umieszczone są szczegółowe informacje o położeniu wszystkich znaków na każdej stronie. Ten plik nosi nazwę „DVI", co jest skrótem angielskiego określenia „*device independent*" (*niezależny od urządzenia*).

Istotnie, wynik wykonanego przez TEXa składu, zakodowany w postaci pliku „DVI" może być następnie przetworzony przez jeden z programów towarzyszących systemowi TEX. Każdy z takich programów służy do posyłania złożonego tekstu na inne urządzenie zewnętrzne. Można więc obejrzeć wynik pracy TEXa na ekranie, wydrukować na drukarce mozaikowej, laserowej lub wykorzystać do bezpośredniego naświetlenia specjalnej kliszy fotograficznej przy pomocy fotonaświetlarki.

System TEX został zaprojektowany i napisany w taki sposób, że nie wykorzystuje się w nim cech żadnego konkretnego komputera. Zakodowany został w pewnej łatwo przenośnej wersji języka PASCAL (por. 3.2). Dzięki takiemu podejściu wkrótce po powstaniu jego podstawowej wersji zostały wykonane instalacje tego systemu na różnych komputerach. Obecnie trudno byłoby znaleźć na świecie typ komputera, na którym nie ma działającego systemu TEX.

2.1. Dane wejściowe

Plik zawierający dane o tym, jak TEX ma wykonać skład, zawiera zwykle przede wszystkim znaki przeznaczone do złożenia, np. tekst książki. Oprócz liter, cyfr i innych znaków dostępnych zwykle na klawiaturze komputera, można zażyczyć sobie umieszczenia w składanym tekście innych znaków poprzez napisanie odpowiednich komend. Aby uzyskać efekt „Æsop's Œuvres" należy napisać w pliku danych dla TEXa: `\AE sop's \OE uvres`, gdzie komendy `\AE` i `\OE`, zaczynające się od „\" i zakończone odstępami, oznaczają polecenia złożenia nietypowych znaków „Æ" i „Œ"

Przełączenie z prostego kroju pisma na *kursywę* następuje poprzez zapisanie w pliku danych dla TEXa komendy `\it`. *Podobnie można użyć komendy* `\bf` **aby włączyć skład tłustym drukiem. Komenda** `\sl` *inicjuje składanie pismem pochyłym. Powrót do pisma prostego następuje przy pomocy komendy* `\rm`.

Język systemu TEX zawiera setki komend i można go rozszerzać o dalsze. Wiele jest potrzebnych bardzo często; inne wykorzystywane są tylko w niektorych przypadkach. Nie trzeba znać ich wszystkich, aby korzystać z systemu. Początkujący użytkownik może zacząć pracę z TEXem już nawet po krótkim wprowadzeniu i następnie w miarę potrzeby uzupełniać swoją wiedzę.

2.2. Trochę wiedzy, żeby docenić TEXa

W system TEX wbudowano ogromną ilość wiedzy matematycznej, informatycznej i poligraficznej. Piszę „wbudowano" ponieważ oprócz autora systemu, prof. Knutha, mam na myśli kilku jego współpracowników, którzy wspomagali prace nad TEXem rozpracowując pewne tematy i robiąc przy okazji doktoraty.

Jednym z takich tematów był problem automatycznego łamania linijek tekstu przy wykonywaniu składu akapitu. Trudność z łamaniem linii polega m.in. na tym, że jeśli decyzję o przejściu do następnego wiersza algorytm podejmuje dla każdej linijki osobno, to może w pewnym momencie nie znaleźć odpowiedniego miejsca podziału. Algorytm opracowany przez Knutha i Plasse działa w sposób dużo bardziej przemyślny. Rozważane są wszystkie możliwe podziały akapitu na linie i każdy z tych podziałów oceniany jest pod względem wielu cech wpływających na elegancję składu. Brane są pod uwagę np. takie cechy, jak to czy międzywyrazowe odstępy w sąsiednich linijkach mają zbliżone do siebie szerokości, czy nie nastąpiło zbyt wiele podziałów wyrazów itp. Dzięki algorytmowi zastosowanemu w TEXu, spacjowanie akapitów jest bardzo równomierne (w istocie najbardziej równomierne z możliwych). Można też wykorzystywać elastyczność metody składania akapitów do osiągania specjalnych efektów. Niniejszy akapit został złożony z dodatkowym poleceniem, by ostatnia linijka kończyła się dokładnie na prawym marginesie. TEX wykonał polecenie, bez konieczności dokonywania jakichkolwiek zmian w treści akapitu.

Twórca TEXa jest matematykiem i zadbał w swoim systemie bardzo starannie o to, by to, co zwykle sprawia najwięcej trudności zecerom — formuły matematyczne — wykonywało się w TEXu szczególnie łatwo, a uzyskiwane efekty były najlepsze, jakie można osiągnąć. Dzięki znakomitemu przystosowaniu TEXa do takiego składu, ogromna ilość matematyków na świecie zaczęła używać tego systemu do pisania swoich prac i książek. Amerykańskie Towarzystwo Matematyczne przyjęło TEXa jako swój standard.

Żeby pisać wzory w tekście składanym TEXem, trzeba nauczyć się nazw podstawowych komend, które sterują procesem składu. Początek i koniec formuły matematycznej zaznacza się w danych dla TEXa znakiem dolara. Wzór $P = x^2 + 3ax - 7$ zapisuje się jako `$P=x^2+3ax-7$`. TEX automatycznie i bardzo starannie dobiera odstępy między znakami, m.in. wokół znaków relacji i symboli oznaczających operacje. Najprostsze wzory matematyczne są bardzo łatwe do zakodowania. Ułamki, np. $\frac{1}{2}$, uzyskujemy pisząc `$1\over2$`. Pierwiastek $\sqrt[3]{x}$ zapisujemy w języku TEXa jako `$\root 3 \of x$`.

Oczywiście bardziej skomplikowane formuły wymagają bogatszego ję-

zyka. Nie jest on trudny do nauczenia i po nabraniu pewnej wprawy można pisać w sposób zupełnie naturalny. Czytelnik może zechcieć spróbować domyślić się które komendy i symbole następujących danych dla TEXa:

```
$p_1(n)=\lim_{m\to\infty}\sum_{\nu=0}^\infty
        \bigl(1-\cos^{2m}(\nu!^n\pi/n)\bigr)$
```

posłużyły do złożenia poniższego wzoru:

$$p_1(n) = \lim_{m\to\infty} \sum_{\nu=0}^{\infty} \left(1 - \cos^{2m}(\nu!^n \pi/n)\right)$$

(G. H. Hardy, *Messenger of Mathematics* 35 (1906), 145–146.)

Bardziej wprawny użytkownik TEXa umie poradzić sobie nawet z takimi diagramami:

$$\begin{array}{ccccccccc}
 & & & & & & 0 & & \\
 & & & & & & \downarrow & & \\
0 & \longrightarrow & \mathcal{O}_C & \xrightarrow{\iota} & \mathcal{E} & \xrightarrow{\rho} & \mathcal{L} & \longrightarrow & 0 \\
 & & \| & & \downarrow{\scriptstyle\phi} & & \downarrow{\scriptstyle\psi} & & \\
0 & \longrightarrow & \mathcal{O}_C & \longrightarrow & \pi_*\mathcal{O}_D & \xrightarrow{\delta} & R^1 f_* \mathcal{O}_V(-D) & \longrightarrow & 0 \\
 & & & & & & \downarrow{\scriptstyle \theta_* \otimes \gamma^{-1}} & & \\
 & & & & & & R^1 f_*(\mathcal{O}_V(-iM)) \otimes \gamma^{-1} & & \\
 & & & & & & \downarrow & & \\
 & & & & & & 0 & &
\end{array}$$

Jeśli ktoś często posługuje się wzorami podobnymi do siebie, nawet tak złożonymi jak powyższy, może przystosować język systemu TEX tak, by składanie ich nie sprawiało większej trudności.

3. Komu i do czego TEX się może przydać?

System TEX jest wykorzystywany do codziennej działalności przez ogromne rzesze naukowców, zwłaszcza informatyków i matematyków, na całym świecie. Piszą oni przy jego pomocy prace naukowe, raporty, listy itp. Dzięki poczcie

elektronicznej mogą przesyłać między sobą prace w postaci plików tekstowych, które po przetworzeniu systemem TEX w docelowym miejscu przybierają formę identyczną jak w miejscu nadania.

Chciałbym zwrócić uwagę na dwa istotne obszary zastosowań TEXa. Omówię rolę i miejsce tego systemu w procesie wydawniczym oraz możliwości wykorzystania go przez programistów.

3.1. TEX w procesie wydawniczym

Spróbujmy prześledzić typową drogę, jaką musi w obecnych czasach w Polsce pokonać tekst książki od miejsca gdzie się zrodził, czyli głowy autora, do miejsca, gdzie nadano mu ostateczną formę, tj. drukarni. Załóżmy przy tym, że autor, co się coraz częściej zdarza, przygotowuje swój tekst przy użyciu komputera.

Napisana przez autora książka, w formie jaką jej nadał przy pomocy swojego systemu składania tekstów, wędruje do wydawnictwa. Ponieważ w takiej postaci zwykle nie odpowiada ona normom tzw. *maszynopisu wydawniczego*, przepisywana jest ponownie przez maszynistkę w redakcji, na zwykłej maszynie do pisania. Maszynopis jest opisywany ołówkiem przez redaktora technicznego. W wyniku tej przemiany tekstu, tam gdzie autor zastosował np. *kursywę*, pojawia się zwykła maszynowa czcionka podkreślona linią falistą. Dopiero teraz maszynopis (z pewną ilością błędów wprowadzonych przez maszynistkę) przekazywany jest drukarni.

W drukarni następuje (ponowne) złożenie tekstu. Przepisywany jest on przez operatora fotoskładu, linotypistę lub innego fachowca. Wprowadza on dalsze błędy do tekstu i zużywa kolejne miesiące które stanowią o długości cyklu wydawniczego. Tekst, poprzez redakcję, która dokonuje własnej korekty, wędruje też do autora, który musi odnaleźć w książce bledy dodane przez maszynistkę i drukarnię. Cykl korekty autorskiej, redakcyjnej oraz poprawiania błędów przez drukarnię powtarza się kilkakrotnie, przy czym nierzadko oprócz usunięcia starych błędów, drukarnia wprowadza nowe.

Po szczęśliwym zakończeniu tego procesu, który trwa czasem kilka lat, książka jest drukowana i ukazuje się w sprzedaży. Jeśli to była książka z dziedziny informatyki, to są duże szanse na to, że jest już niezupełnie aktualna.

Najwyższy czas powiedzieć jak TEX może ułatwić i przyspieszyć tę trudną drogę. Przyjrzyjmy się idealnemu obrazowi współpracy autora, wy-

dawcy i drukarza. Obraz ten, mimo że idealny, jest równocześnie zupełnie realny.

Pierwszym etapem byłaby oczywiście praca autora. Pisałby on tekst książki na swoim komputerze, nadając jej przy pomocy TEXa formę zaleconą mu przez wydawnictwo, z którym byłby w kontakcie. Najlepiej, jeśliby wydawnictwo oprócz ustnych zaleceń dotyczących np. rozmiarów strony, odstępów między wierszami, kroju i wielkości czcionek, przygotowało przy pomocy zatrudnionych w nim fachowców odpowiednie rozszerzenie systemu TEX. Pozwoliłoby to autorowi skoncentrować się na treści i strukturze logicznej książki, podczas gdy przystosowany do tego celu system w pewnym sensie „reprezentowałby" redakcję i współpracował z autorem. Autor oglądałby napisany przez siebie tekst na ekranie komputera i dokonywał próbnych wydruków na drukarce swojego komputera.

Po zakończeniu prac autorskich, tekst książki w postaci pliku danych dla TEXa wędrowałby na dyskietce lub przy pomocy kabla z komputera autora do komputera wydawcy. Wydawca mógłby wykonać natychmiast skład książki i obejrzeć ją na ekranie lub wydrukować na swojej — prawdopodobnie lepszej niż autora — drukarce. Złożenie kilkusetstronicowej książki i wydrukowanie jej do pierwszej korekty nie powinno zająć na odpowiednio dobrym sprzęcie więcej niż ok. 2 godzin. Teraz następowałby proces sprawdzania tekstu i korygowania go. Wszystko odbywałoby się na redakcyjnym biurku, bez żadnego udziału drukarni, jedynie w porozumieniu z autorem. Po każdej zmianie nowa wersja książki byłaby natychmiast gotowa do dalszej ewentualnej korekty.

Kiedy redakcja i autor byliby już usatysfakcjonowani kształtem ich dzieła, ostateczna wersja książki — znowu na dyskietce lub kabelkiem — przekazywana byłaby do drukarni. Tam miejscowy komputer przy pomocy odpowiedniego programu wchodzącego w skład systemu TEX dokonywałby naświetlenia klisz fotograficznych na fotonaświetlarce i te klisze byłyby natychmist użyte do wydrukowania książki. Żadne decyzje podjęte w redakcji nie byłyby tu już zmieniane. Dzięki TEXowi forma książki utworzona na fotonaświetlarce byłaby identyczna z tą, którą redakcja oglądała na swoim ekranie i drukowała na swojej drukarce; jedyną różnicą byłaby wyższa jakość techniczna uzyskana na sprzęcie znajdującym się w drukarni. Niepotrzebna byłaby już jakakolwiek dodatkowa komunikacja między redakcją i drukarnią. Druk książki mógłby rozpocząć się tego samego dnia, którego dostarczono plik z jej tekstem.

Ten piękny obraz nie jest utopią. Wiele wydawnictw na świecie przy-

jęło już jako swój standard tę formę pracy lub jakąś jej odmianę, dzięki czemu książki wydawane są szybko, elegancko i zawierają mniej błędów drukarskich. Niebagatelną cechą tego procesu jest też jego taniość. Wykorzystywany sprzęt jest standardowy i nie musi być specjalnie przystosowywany do celów wydawniczych. Cena oprogramowania (systemu TEX i programów pomocniczych) jest na ogół niższa niż ceny innych systemów służących do podobnych celów.

Należy mieć nadzieję, że polskie wydawnictwa zechcą podążyć za światowymi trędami, ku swojej własnej wygodzie i zadowoleniu czytelników.

3.2. W czym TEX może się przydać programistom?

Ważną cechą TEXa odróżniającą go od wielu innych systemów składu jest fakt, że wszystkie komendy sterujące jego pracą zapisane są w pliku tekstowym. Pod pewnym względem można to uznać za wadę: ci, którzy przyzwyczajeni są do obsługiwania programów przy pomocy myszy i okien błyskających na ekranie mogą czuć się rozczarowani. Jednak cecha ta pozwala na coś niezwykłego: pracą TEXa może sterować nie człowiek, lecz inny program.

Program może jako wynik zapisać plik, który będzie zawierał pewien przeznaczony do złożenia tekst oraz odpowiednie komendy dla TEXa. Pewne warianty wykorzystania tej cechy systemu mogą być bardzo użyteczne w pracy programisty. Można zastosować TEXa do wspomagania pisania programów, i tworzenia ich dokumentacji oraz używać TEXa do eleganckiego formatowania wyników pracy tych programów.

Wykorzystywanie TEXa przy pisaniu programów i dokumentacji zostało zapoczątkowane prez samego twórcę systemu. We wstępie napisałem, że Knuth przerwał pracę nad swoją książką, żeby stworzyć narzędzie do wykonania jej składu. To tylko część historii. Sama praca nad systemem TEX zaczęła się również od stworzenia pewnego narzędzia. Żeby utworzyć tak duży program, Knuth uznał, że musi najpierw zrobić odpowiedni system programowania, który ułatwiłby mu to zadanie. Tak doszło do powstania systemu WEB.

Knuth twierdzi, że gdyby jedynym skutkiem wieloletniej pracy włożonej w TEXa było powstanie systemu WEB, to efekt ten byłby wart całej tej pracy. WEB został pomyślany jako system ułatwiający jednoczesne programowanie i tworzenie dokumentacji.

Zwykle w czasie tworzenia programu, wpisuje się do jego tekstu komentarze, które mają ułatwić czytanie i rozumienie algorytmów. Równocześnie, często na osobnych kartkach, zapisuje się pewne dodatkowe informacje,

np. opis struktur danych, większe dowody porawności algorytmów itp. Ideą systemu WEB jest umieszczanie zarówno programu jak i jego pełnej dokumentacji w jednym pliku. Taki plik może następnie być automatycznie przetworzony na „czysty" program, m.in. poprzez usunięcie komentarzy, a następnie skompilowany i wykonany. Może też być przetworzony na zestaw danych dla TEXa i wtedy TEX wykonuje bardzo czytelny skład programu i dokumentacji, dodając m.in. automatycznie wykonany skorowidz wszystkich identyfikatorów użytych w tekście programu oraz spis treści.

Cały program TEX został napisany w systemie WEB i wraz z zawartą w nim dokumentacją złożony (przez samego siebie) i wydany w formie książkowej. Książka „*TEX: The Program*" stanowi tom B napisanej przez Knutha pięciotomowej serii poświęconej komputerowemu składaniu tekstów.

Wydaje się, że bardzo istotnym, choć mało jeszcze rozpoznanym, zastosowaniem TEXa może być formatowanie wyników pracy programów. Łatwo sobie wyobrazić, że program generujący, powiedzmy, dane statystyczne, może przedstawiać rezultaty w różnej formie. Najprostszą są oczywiście ciągi liczb drukowane na drukarce komputera. Inną jest prezentacja wyników w postaci graficznej. Nie rezygnując z przedstawienia wyników w postaci liczbowej można je uczynić bardzo czytelnymi, jeśli zapisze się je najpierw w postaci pliku danych, a następnie przetworzy odpowiednio przystosowanym programem TEX. Takie informacje po wydrukowaniu mogą być wykorzystane bezpośrednio, np. jako część książki.

Znakomitym i często wykorzystywanym pomysłem jest połączenie TEXa z bazą danych. Program obsługujący bazę dba wtedy tylko o przechowywanie, uaktualnianie i selekcję informacji. TEX przejmuje na siebie nadawanie odpowiedniej formy wyciągom z tej bazy. Można w ten sposób obsługiwać listy adresowe, listy bibliograficzne i oczywiście wiele innych.

4. Podsumowanie

System TEX wciąż zdobywa popularność na całym świecie. Używany jest w dziesiątkach krajów świata, w różnych językach. Został również przystosowany do języka polskiego i jest wykorzystywany m.in. do składania książek i czasopism. Co najmniej jedna z działających na naszym rynku firm trudniących się składem specjalizuje się w obsłudze TEXa. Mam nadzieję, że TEX w Polsce znajdzie uznanie użytkowników i będzie z powodzeniem wykorzystywany.

5. Podziękowania

Największe podziękowania chciałbym złożyć na ręce Bogusława Jackowskiego, mojego najbliższego współpracownika, z którym w ciągu ostatnich dwóch lat spędziłem setki godzin zgłębiając razem sekrety TEXa oraz przystosowując go do języka polskiego. W dużym stopniu dzięki pracy z Bogusiem poznałem TEXa. Z przyjemnością nadal obaj zajmujemy się intensywnie tym systemem.

Dziękuję również Prof. Andrzejowi Bliklemu za zaproszenie mnie do wygłoszenia referatu w Mrągowie i do napisania tego artykułu. Jego życzliwe zainteresowanie moją pracą nad TEXem było mi bardzo pomocne.

Szósta Jesienna Szkoła PTI
Mrągowo, listopad 1989

JAKIE KORZYŚCI Z KOMBINATORYKI MOŻE MIEĆ PROGRAMISTA

MACIEJ M.SYSŁO
Instytut Informatyki Uniwersytetu Wrocławskiego
ul. Przesmyckiego 20, 51-151 Wrocław

Streszczenie

Wiele problemów rozwiązywanych z pomocą komputerów przyjmuje postać obliczeń na zbiorach skończonych (dyskretnych), których elementy są dodatkowo powiązane zależnościami strukturalnymi, tworzącymi na przykład graf lub częściowy porządek. Projektowanie takich obliczeń polega najczęściej na jednoczesnym wyborze odpowiedniego algorytmu i dostosowanych do niego struktur danych. Za kryterium ich wyboru przyjmuje się efektywność wykonywania zaprojektowanych obliczeń.

Programista ma dzisiaj do swojej dyspozycji wiele systemów ułatwiających tworzenie oprogramowania (na przykład języki turbo i systemy baz danych), które, często kosztem głębszej analizy problemu, umożliwiają szybkie otrzymywanie ostatecznych produktów.

Na tym tle, w pierwszej części wykładu, autor dzieli się swoimi uwagami na temat powodów niskiego notowania informatyki wśród innych nauk i rażąco niskiego miejsca jakie zajmują programiści wśród informatyków.

W drugiej części wykładu jest przedstawione zagadnienie z dziedziny komputerowo zintegrowanego wytwarzania (CIM). Po krótkiej charakterystyce samej dziedziny, jest omówiony model zagadnienia i zadania jakie stają przed programistą biorącym udział w jego rozwiązaniu.

W trzeciej i ostatniej części, algorytm Dijkstry rozwiązywania problemu najkrótszych dróg służy do zilustrowania ewolucji kopca. Omówiono tam ponadto pojęcia samoorganizujących się struktur danych i zamortyzowanego kosztu obliczeń

Wykład ten jest popularnym omówieniem wybranych aspektów czasowej i pamięciowej złożoności obliczeniowej problemów matematyki dyskretnej.

1. Dlaczego nas tak nisko cenią, czyli o notowaniach informatyków i programistów.

Pozycję programisty w opiniach wielu informatyków (a tym bardziej w oczach przedstawicieli innych nauk a zwłaszcza matematyków) można przyrównać do miejsca jakie informatyce przydziela się wśród innych dziedzin nauki. Ta opinia odnosi się przede wszystkim do sytuacji w Polsce.

Informatyka wyrastała niemal niezależnie z dwóch, całkowicie odmiennych dziedzin: matematyki i elektroniki. Dalej, ograniczamy się do jej znaczenia związanego z tym pierwszym pochodzeniem. Przez długie lata, informatyka dzieliła los wielu innych zastosowań matematyki i można powiedzieć, że była dziedziną niegodną zainteresowań czystego matematyka. Przyczyn takiego stanu rzeczy można upatrywać, po pierwsze w rodzaju matematyki uprawianej i wykorzystywanej przez informatyków, a po drugie - w interdyscyplinarności tej dziedziny. Chociaż angielska nazwa informatyki *Computer Science* zawiera w sobie słowo Nauka, z trudem przebija się ona jako samodzielna i równoprawna dziedzina wśród innych nauk, i najczęściej spotykamy ją związaną z innymi specjalnościami takimi jak automatyka czy robotyka. Pewnymi wyjątkami są niektóre działy informatyki, zdobywające sobie miejsce wśród innych specjalności matematycznych dzięki określeniu *matematyczny* (np. matematyczne podstawy obliczeń), lub w kombinacji z ugruntowaną dziedziną matematyczną (np. logika dla informatyków).

Los informatyki w opiniach specjalistów od matematyki ciągłej (i abstrakcyjnej) dzieli *matematyka dyskretna* (ang. discrete mathematics), najpowszechniej uprawiany rodzaj matematyki przez informatyków. Powodem jest przede wszystkim zakres tej dziedziny ograniczony do problematyki na zbiorach dyskretnych, a najczęściej skończonych. A cóż ciekawego można udowodnić o skończonej przestrzeni rozwiązań.

Ostatnio pojawiło się nowe określenie rodzaju matematyki, a mianowicie **matematyka konkretna** (ang. concrete mathematics). Słowo concrete można utworzyć w następujący sposób CONtinuous and disCRETE. Jest to więc pewna propozycja ugody, przynajmniej w nazwie, między matematyką ciągłą i dyskretną. Chodzi

jednak nie tylko o nową nazwę, ale przede wszystkim o zakres znaczeniowy tego pojęcia. Autorami pierwszego podręcznika poświęconego matematyce konkretnej [8] są Graham, Knuth i Patashnik. We wstępie do swojej książki tak uzasadniają powstanie tej nowej dziedziny. Dla informatyków, wykształcenie matematyczne jakie odbierają jest niewystarczające do pełnego rozumienia wszystkich aspektów procesu tworzenia programów. Te braki ma wypełnić matematyka konkretna, która początkowo była pomyślana jako antidotum na matematykę abstrakcyjną, gdyż konkretne rezultaty klasyczne zaczęły być wypierane z obowiązującego zakresu wiedzy matematycznej przez nową falę abstrakcyjnych idei, popularnie zwanych nową matematyką. Matematyka abstrakcyjna jest piękna i pożyteczna, ale jej wyznawcy zaczęli uważać, że pozostała matematyka jest niepotrzebna, gorsza i niewarta poświęcania uwagi. Zachłystywanie się abstrakcją powoduje, że wielu matematyków nie potrafią dostrzec piękna w rozwiązywaniu problemów ilościowych (liczbowych) i nie docenia znaczenia technik obliczeniowych w matematyce. Matematyka konkretna ma wprowadzić równowagę między abstrakcją i obliczeniami. A oto zawartość wspomnianej książki: sumy, zależności rekurencyjne, elementarna teoria liczb, współczynniki Newtona i inne liczby specjalne, funkcje tworzące, prawdopodobieństwo dyskretne, metody asymptotyczne (oszacowania). W prezentacji materiału, główny nacisk jest położony na techniki posługiwania się zależnościami, a nie na twierdzenia o istnieniu lub rozważania kombinatoryczne. Jak widać z powyższego wyliczenia zagadnień, zakres matematyki konkretnej różni się od powszechnie przyjmowanego zakresu matematyki dyskretnej, do której najczęściej zalicza się takie dziedziny jak: operacje na zbiorach, rachunki logiczne, kombinatorykę i teorię grafów, zbiory uporządkowane, porządkowanie i przeszukiwanie.

Programiści wśród informatyków uchodzą za osoby najbardziej oddalone od nauki, w tym przede wszystkim od matematyki. Jest to opinia zarówno osób spoza informatyki jak i informatyków z kręgów *computer science*. Myślę, że tej sytuacji są także winni sami informatycy, nie wykluczając programistów. Panuje przekonanie, podtrzymywane przez rzesze informatyków, że z pomocą komputera można policzyć wszystko, mniejsza o metodę. Obecny zalew mikrokomputerów potęguje jeszcze ten pogląd, gdyż coraz więcej niespecjalistów ma do nich dostęp, bez odpowiedniego, a czasem najmniejszego przygotowania. W wielu przypadkach cała edukacja informatyczna odbywa się przy klawiaturze z pomocą oprogramowania, które w danej chwili znajduje się w komputerze.

Autorzy Matematyki Konkretnej, motywując powstanie swojego dzieła, zarzucili matematyce abstrakcyjnej, że nie oferuje informatykom potrzebnej im wiedzy matematycznej. Ośmielam się wystąpić tutaj z poważniejszym oskarżeniem, tym razem wobec informatyków (a zwłaszcza programistów), że nie wykorzystują dostarczanej im wiedzy teoretycznej w swoich tworach programistycznych. Jest to między innymi powodem niskiej oceny poziomu ich prac.

Przy okazji chciałbym także stanąć w obronie programistów i ponarzekać na ich złe traktowanie przez rodzimych autorów i wydawnictwa. Ukazuje się wiele książek, w tytule których można znaleźć, że są "...dla programistów". Niestety, nie są to pozycje, na które najbardziej oczekują programiści. Wiele z nich ma po prostu na celu dostarczenie uzupełniającej wiedzy teoretycznej. Książki w języku polskim o programowaniu i dla programistów można podzielić z grubsza na dwie kategorie: o językach programowania lub o wybranych technikach i narzędziach programowania. Przynależność do tej pierwszej kategorii jest dość oczywista, zaś do drugiej zaliczyć można na przykład książki Wirtha i Dijkstry oraz Aho i wsp. Brak jest całkowicie pozycji zorientowanych na rozwiązywanie problemów. Nie chodzi tutaj tylko o przykładowe zadania akademickie, które są dobrą ilustracją jednego algorytmu lub klasy metod, ale o problemy rzeczywiste, z którymi na codzień stykają się programiści. Powinno dołożyć się wszelkich starań, by jak najprędzej ukazały się u nas opracowania Bentleya [4] i Kernighana [9] i [10], zawierające gotowe przepisy dla programistów.

Poza nielicznymi wyjątkami, nie ukazują się w kraju książki uzupełnione dyskietkami z kodami zawartych (lub tylko omówionych) w nich programów. Nie słyszałem, by któreś z wydawnictw planowało utworzenie odpowiedniego działu. (Złożona w WNT 5 lat temu propozycja przetłumaczenia książki z programami i wydania towarzyszącej jej dyskietki została odrzucona przez Wydawnictwo.) Zasłanianie się brakiem prawnych regulacji dotyczących ochrony oprogramowania jest zasłoną dymną. To są dwie różne sprawy. Chronić należy *rynek* i *przemysł oprogramowania*, których na dobrą sprawę u nas nie ma. Teksty kodów publikowane w książkach, są chronione tym samym prawem autorskim co książki, z których pochodzą. Mogą więc być tak samo używane jak fragmenty książek. Czy zamieszczając czyjeś twierdzenie w swojej książce, która następnie jest wydawana w 5 tys. egzemplarzy, płacimy autorowi twierdzenia część swojego honorarium?

Dlaczego mielibyśmy postępować inaczej z fragmentami kodów wziętych z książek? Publikowane w książkach stają się publiczną własnością. Trzeba jak najprędzej umożliwić i ułatwić programistom korzystanie z takich źródeł wiedzy i oprogramowania.

2. Czubek góry lodowej, czyli rzut oka na komputerowo zintegrowane wytwarzanie.

W tej części wykładu zostanie omówiona bardzo uproszczona wersja zagadnienia, będącego przedmiotem zainteresowania komputerowo zintegrowanego wytwarzania. Zanim jednak opiszemy samo zagadnienie i pojawiające się w nim problemy dla programistów, scharakteryzujemy bardzo ogólnie czym się zajmuje* ta dziedzina.

2.1. Komputerowo zintegrowane wytwarzanie

Dziedzina *komputerowo zintegrowanego wytwarzania* (ang. Computer Integrated Manufacturing - CIM) powstała w odpowiedzi na potrzeby współczesnego rynku towarów konsumpcyjnych, por. [12] i [14]. Wzmożona po II wojnie światowej masowa produkcja towarów przeżywa dzisiaj szereg trudności wynikających z technologicznych, organizacyjnych i społeczno-politycznych barier. Z kolei, bariery te są istotnymi stymulatorami nowych technologii i rozwiązań organizacyjnych. Wyróżnić można trzy podstawowe grupy trudności:

1. Mało elastyczna produkcja masowa.
Żywotność produktów i recesja na rynku zmuszają producentów do dużego różnicowania produkcji. Dla przykładu, coraz trudniej jest spotkać na szosie dwa identyczne mercedesy. Nawet należące do tej samej klasy pojemności, różnią się między sobą od istotnych detali w silniku po ozdoby, nie mówiąc już o wyposażeniu wnętrza, które klient może wybrać wg swoich upodobań, tak jak to robi na przykład przy urządzaniu mieszkania.

2. Rosnąca złożoność produktów.
Mały samolot ma 30 tys. części a lokomotywa - 200 tys., z których wiele jest także częściami złożonymi. Produkcja staje się więc w dużej mierze planowaniem przepływu zasobów, czyli koordynacją dostawców, optymalizacją wykorzystania materiałów i gospodarką zapasami.

3. Ograniczone możliwości wykorzystywania sił roboczej.
Wynika to z rosnącej złożoności produkcji i rosnących wymogów

odnośnie dokładności, by tym zagwarantować wysoką jakość produktów. Z powodu barier socjologiczno-psycholgicznych, człowiek nie jest w stanie zapenić produkcji bezodpadowej.

Te trzy grupy trudności nie pozwalają starymi metodami udoskonalać produkcji, jej dokładności i jakości. Potrzebne są nowe technologie i nowe podejścia do całego procesu, od projektowania aż po dostarczenie produktu końcowego. Największe oczekiwania wiąże się z mikroelektoniką, a ogólniej z nową technologią informacji, w której upatruje się szansę na odejście od mało elastycznej produkcji masowej. Zanim to nastąpi musi jednak dojść do uproszczenia komunikacji komputerowej i standaryzacji elementów systemów komputerowych, w tym także oprogramowania, por [13].

Koncepcja CIM wypływa stąd, że warunkiem pozostania konkurencyjnym na rynku jest szybka i elastyczna odpowiedź na zmieniające się żądania klientów i zmieniającą się technologię wytwarzania. CIM jest zintegrowanym systemem sterowania produkcją, ściśle powiązanym z systemami inżynierskimi i zarządzającymi, które go wspierają i nim kierują. Krytycznym elementem w takim kompleksie jest zdolność wszystkich programów (podsystemów) do korzystania ze wspólnych danych.

Ostatnio pojawiło się inne jeszcze pojęcie, tzw. *elastyczny system produkcyjny* (ang. Flexible Manufacturing System - FMS). Jest to podsystem w CIM, który jedynie automatyzuje niektóre etapy całego procesu produkcyjnego w jednej firmie, podczas gdy CIM łączy ze sobą wszystkie etapy produkcji i pozwala na koordynację działalności w różnych sferach produkcji: w projektowaniu (CAD), wytwarzaniu (CAM i CAE), zarządzaniu (MRP i EDS) i finasowaniu. Systemy CIM nie ograniczają się do pojedynczych zakładów, ale obejmują swym działaniem całe korporacje lub gałęzie produkcji.

2.2. Pewne zagadnienie w CIM

Wyobraźmy sobie fabrykę produkującą urządzenia gospodarstwa domowego takie jak roboty, miksery, młynki, wirówki, ekspresy, noże elektryczne, żelazka, podgrzewacze, itp, itd. Na przykład Philips, AEG czy NEC produkują te i masę innych urządzeń. Wymienione wyżej przyrządy mają wiele elementów wspólnych (chociaż nie zawsze identycznych) jak na przykład silniki, instalację elektryczną, termostaty, żarówki, i prawie każde urządzenie ma elementy niespotykane w innych.

Elastyczny rynek zamówień, z odbiorcami stawiającymi warunki i wymogi odnośnie wielu cech towarów, nie pozwala na planowanie produkcji z dużym wyprzedzeniem i w długich seriach złożonych z identycznych towarów. Producent musi być przygotowany na każdy kaprys klienta. Powstaje więc pytanie w jaki sposób zorganizować produkcję by z jednej strony wyjść na przeciw wymaganiom rynku, a z drugiej wyjść na swoje. Wiadomo, że gros zysku uzyskuje się z masowej produkcji, gdzie raz zainwestowany kapitał, nawet przy niewielkim zysku jednostkowym, mnożony wielkością produkcji może przynosić stały, dość wysoki zysk.

By nie zajmować się wszystkimi szczegółami, uprościmy rzeczywiste zagadnienie i przyjmiemy, że produkcja, na przykład u Philipsa, polega na wytwarzaniu *produktów końcowych* ze zbioru *części elementarnych* według ściśle opisanego *przepisu wytwarzania*. Produkcja, począwszy od etapu projektowania aż po końcowy etap wysyłki gotowych produktów korzysta ze wspólnej dla wszystkich szczebli zarządzania *bazy danych* o wszystkich możliwych produktach i niezbędnych do ich wytworzenia elementach oraz o sposobach produkowania.

Dalej, gros uwagi poświęcimy właśnie tej bazie danych, gdyż jest ona najistotniejszym elementem systemu sterowania produkcją, zarówno na poziomie bezpośredniego wytwarzania jak i na najwyższym szczeblu zarządzania. Nie zajmujemy się jednak szczegółami impelmentacyjnymi. Powiemy jedynie o rodzajach infomacji jakie powinny być w niej zawarte, o powiązaniach, które muszą być także opisane oraz, co jest najważniejsze, o algorytmach efektywnego z niej korzystania.

Baza zawiera informacje o elementach biorących udział w produkcji. Można je podzielić na trzy klasy: części elementarne, elementy pośrednie i produkty końcowe. Zakładamy, że elementarne części są dostarczane z zewnątrz jednostki, którą się zajmujemy, a elementy pośrednie są uzyskiwane w rozpatrywanym procesie produkcji. Kilka uwag o każdej z klas elementów.

1. Części elementarne.
Pochodzą z innych firm, zatem należy je zamawiać, sprowadzać i ewentualnie przechowywać. Wiadomo dokładnie w których występują elementach pośrednich i produktach końcowych. Określona jest także dokładnie ich forma w tych produktach, mogą więc wymagać dodatkowego przetworzenia, np. ogwintowania, skrócenia, pomalowania, itp.

2. Elementy pośrednie i produkty końcowe.

Wiadomo z jakich i ilu składają się części elementarnych i innych elementów pośrednich oraz określony jest proces składania. W przypadku elementów pośrednich są także określone elementy pośrednie i produkty końcowe, które je zawierają.

W bazie danych, rekord opisujący jej pojedynczy element zawiera przynajmniej następujące informacje:
- nazwę elementu,
- symbole techniczne opisujące element i jego cechy (na te informacje może być przeznaczonych wiele pól, których składowymi są zwykle łańcuchy symboli),
- listę elementów z których jest wytwarzany (części elementarne lub inne elementy pośrednie),
- listę elementów, do wytwarzania których służy (inne elementy pośrednie lub produkty końcowe),
- opis procesu wytworzenia (tj. *składania*) zawierający między innymi opis zasobów potrzebnych do wytworzenia (siłę roboczą lub maszyny), kolejność i czasy wykonywania poszczególnych operacji.

Dla uzasadnienie dalszych rozważań, wymieńmy najważniejsze cele do jakich będzie wykorzystywana ta baza. Można je podzielić na dwie kategorie: produkcyjne i doradcze.

1. *Zadanie produkcyjne* jest poleceniem wyprodukowania określonej liczby elementów pośrednich lub produktów końcowych. Najczęściej, zadanie to ma być ukończone przed z góry określonym terminem, często nie pozwalającym na odtworzenie całego procesu produkcji. Z tego powodu, jest uzasadnione magazynowanie niektórych elementów składowych. Zwłaszcza, że produkty końcowe w zamówieniach charakteryzują się dużą rożnorodnością cech zewnętrznych, które zwykle są nadawane produktom w ostatniej fazie produkcji. Zadanie ma być wykonane w określonym terminie a przy tym wytwórca musi uzyskać z produkcji pewien dochód. Oczywiście, przed przystąpieniem do produkcji, zadania są pod tym względem analizowane.

2. *Zadanie doradcze* polega na szybkim uzyskaniu informacji o możliwościach wykonania zadań produkcyjnych, czyli o mocach produkcyjnych, stanie magazynów, itp. Na podstawie oceny tendencji rynku (gustów, mody, itp.) są planowane zakupy części elementarnych i jest prowadzona produkcja z wyprzedzeniem niektórych elementów składowych, co umożliwia później szybką odpowiedź na spodziewane zamówienia.

2.2.1. Kombinatoryczne cechy bazy danych

Stosujemy tutaj terminologię, która jest objaśniona w książce Lipskiego [11].

Rozpatrywana baza danych może już istnieć w jakiejś postaci lub jej struktura może być narzucona przez systemy nadrzędne, na przykład przez systemy zarządzania większymi jednostkami organizacyjnymi. Należy brać także pod uwagę, że z jednej strony pewne fragmenty bazy mogą być tymczasowo *martwe* (np. w przypadku zaniechania produkcji niektórych towarów), a z drugiej - baza powinna być otwarta na przyjmowanie nowych elementów w związku ze zmianami technologii lub pojawianiem się nowych odmian towarów oraz całkiem nowych produktów.

Nie powiedzieliśmy jeszcze nic o wielkości bazy. W przypadku takich kolosów jak Philips czy NEC, ich bazy zawierają setki tysięcy elementów, a w mniejszych fabrykach liczą dziesiątki tysięcy. W tej sytuacji, jakakolwiek obróbka bazy, zwykle korzystająca tylko z części zakodowanych informacji, musi być wykonywana na wydzielonych z bazy strukturach zależności między elementami, które dodatkowo są obciążone pewnymi charakterystykami ilościowymi (liczbowymi).

Podstawową cechą rozpatrywanej bazy jest jej *acykliczność*, tj. rekordy i relacje zależności między elementami tworzą graf skierowany (w skrócie *digraf*), który nie zawiera cykli skierowanych. Jest to tak oczywista własność tej struktury, że bardzo często programiści ... o niej zapominają i, w konsekwencji, nie korzystają z uproszczeń jakie umożliwia.

Przypomnijmy tutaj, że większość problemów na grafach można sklasyfikować w dwóch grupach:
- problemy mające wielomianowe algorytmy rozwiązywania, i
- problemy NP-zupełne, czyli takie, które nie można rozwiązywać dokładnie inaczej niż metodami pełnego przeglądu przestrzeni rozwiązań.

Należy jeszcze pamiętać, że algorytmy rozwiązywania problemów z pierwszej grupy mają złożoność będącą zwykle wielomianem niskiego stopnia, a w przypadku grafów acyklicznych - bardzo często pierwszego stopnia. Nie należy tej informacji zaniedbywać. W swoim wykładzie na I Szkole PTI zilustrowałem jak szybko dla dużych rozmiarów danych czas obliczeń wzrasta ze wzrostem stopnia wielomianu nawet o jeden.

Na grafie acyklicznym reprezentującym naszą bazę danych po-

winniśmy wykonać pewne czynności, które usprawnią (nawet o rząd wielkości) obliczenia wykonywane już w ramach rozwiązywania konkretnych zadań produkcyjnych lub doradczych. Przede wszystkim należy:

1. Wydzielić *składowe spójności*, czyli rozdzielić elementy, które nie wchodzą razem do żadnego produktu końcowego. Operacja ta powinna być wykonana na całym grafie. Po jej wykonaniu zaś, dalsze operacje na bazie, zarówno strukturalne jak i obliczeniowe mogą być wykonywane na poszczególnych składowych oddzielnie. Należy przewidzieć jednak, że ożywienie martwych fragmentów bazy lub zmiany w technologii mogą prowadzić do zmiany zależności strukturalnych, co z kolei może mieć wpływ na podział na składowe.

2. Wyznaczyć *uporządkowaną numerację* elementów (zwaną także topologicznym uporządkowaniem) i wydzielić części elementarne i produkty końcowe w każdej składowej oddzielnie.

Te dwie operacje na bazie, a raczej na acyklicznej zależności strukturalnej między jej elementami, należy wykonać właśnie w takiej kolejności, w przeciwnym bowiem razie elementy tej samej składowej mogłyby być porozrzucane po całym uporządkowaniu.

3. Utworzyć uporządkowane reprezentacje poszczególnych składowych korzystając z wyników kroków 1 i 2. Należy wziąć pod uwagę reprezentacje tablicową i wskaźnikową (pomijamy reprezentację macierzową jako niezmiernie rozrzutną). Ta pierwsza nadaje się niemal do wszystkich operacji na bazie z wyjątkiem jej modyfikacji przez usuwanie lub wstawianie elementów - wtedy musi być użyta ta druga. Niestety, reprezentacja wskaźnikowa ma tę wadę, iż za każdym razem gdy zachodzi konieczność przeniesienia jej do pamięci pomocniczej, musi być zamieniona na inny typ strukturalny.

Po zbudowaniu grafowego modelu zależności strukturalnych należy nałożyć nań:
- wielkości liczbowe odpowiadające elementom (wierzchołkom w digrafie) i parom elementów (łukom w digrafie) - wielkości te odpowiadają cechom elementów oraz określają ilościowe zależności między nimi (w przypadku łuków);
- zależności technologiczne między elementami wynikające z technologicznego opisu procesu produkcji.

Ten pierwszy typ zależności wpływa zwykle na zwielokrotnienie rozmiaru opisu - każdej nowej funkcji opisanej na wierz-

chołkach lub łukach odpowiada nowa tablica przyporządkowana wierzchołkom lub łukom, odpowiednio. Dlatego liczby, zarówno typu całkowitego jak i rzeczywistego, powinny być reprezentowane jak najoszczędniej. W obu przypadkach, odpowiednią reprezentację liczb wybieramy po przeanalizowaniu zakresu wartości parametrów i uwzględnieniu żądanej dokładności obliczeń. Często można zrezygnować ze standardowych reprezentacji liczb, zwłaszcza typu rzeczywistego, i zastąpić je oszczędniejszym sposobem kodowania.

W tym drugim zaś przypadku, sytuacja jest o wiele bardziej skomplikowana z następujących powodów:
- liczba maszyn poszczególnych typów może być poważnym wąskim gardłem w produkcji i należy optymalizować ich wykorzystania;
- elementy wykonywane (lub składane) na tych samych maszynach (lub stanowiskach pracy) mogą pochodzić z różnych składowych spójności. Zatem zależności technologiczne mogą prowadzić do innego podziału na składowe.

2.2.2. Zadania dla programistów

Omówimy teraz pokrótce zadania, z którymi musi umieć poradzić sobie programista (a raczej zespół programistów) pracujący nad systemem dla naszego przykładowego zagadnienia z komputerowo zintegrowanego wytwarzania. Zakładamy, że rozpatrywana sieć zależności jest spójna i ma n wierzchołków oraz m łuków.

a. Analiza i synteza zależności strukturalnych.

a.1. Podział digrafu na składowe spójności. Można zastosować algorytm przeglądania w głąb (metoda dfs) do grafu powstałego z digrafu naszej bazy przez zaniedbanie orientacji łuków. Złożoność $O(m)$.

a.2. Wyznaczenie uporządkowanej numeracji w składowych. Stosujemy przeglądanie składowych począwszy od części elementarnych i numerujemy wierzchołki w kolejności ich osiągania. Złożoność $O(m)$.

a.3. Podprogram realizujący a.2 może być rozbudowany do postaci dającej kompletną reprezentację sieciową bazy, tj. zawierającą wszystkie zależności strukturalne i liczbowe między elementami.

a.4. Wydzielenie pełnych poddigrafów dwudzielnych. Jest to operacja, która może prowadzić do znacznego zredukowania rozmiaru sieci. Jeśli np. znajdziemy poddigraf na $p+q$ wierzchoł-

kach, to w zredukowanej sieci wprowadzamy jeden nowy wierzchołek i zastępujemy $p*q$ łuków $p+q$ łukami. Niestety, znalezienie w sieci największego poddigrafu dwudzielnego jest problemem NP-zupełnym, dlatego muszą być stosowane algorytmy przybliżone. Algorytmy takie najczęściej znajdują największy podzbiór wierzchołków o możliwie największej części wspólnej ich zbiorów bezpośrednich następników w sieci. Złożoność takiej metody przybliżonej jest przenajmniej $O(n^3)$, ale nie gwarantuje ona otrzymywania zbyt dobrych rozwiązań. Dlatego trzeba stosować metody bardziej pracochłonne.

a.5. Wyznaczanie minimalnych cięć w sieci, tj. podzbiorów elementów blokujących maksymalny przepływ elementów od części elementarnych do produktów końcowych. Elementy wchodzące w skład cięcia powinny być obiektem szczególnej troski organizatorów produkcji, gdyż ich wytworzenie decyduje o powodzeniu całej produkcji. Cięcia w sieci wyznacza się stosując techniki przepływów. W najprostszej wersji algorytm ma złożoność $O(n^3)$.

b. Wyznaczanie liczbowych charakterystyk w sieci.

b.1. Wyznaczanie długości dróg lub szukanie optymalnych dróg. Reprezentacje sieci znalezione w punktach a.1-3 pozwalają stosować metody, które wykonują liczbę elementarnych operacji proporcjonalną do liczby przeglądanych elementów w sieci.

b.2. Wyznaczanie wielkości przepływów. Przepływy w naszej sieci odpowiadają przepływom elementów składowych w trakcie produkcji realizującej ustalone zamówienia. Zamówienia mogą pochodzić od odbiorców lub być tylko przygotowaniem do spodziewanych zamówień.

Zarówno w przypadku b.1 jak b.2 obliczenia mogą być wykonywane począwszy od części elementarnych lub od elementów pośrednich, z nałożonymi warunkami początkowymi i brzegowymi lub bez nich.

c. Szeregowanie zadań technologicznych.

Przez *zadanie technologiczne* rozumiemy tutaj czynność techniczną, która ma być wykonana na jednym lub zespole elementów. Może nią być: obróbka jednego elemntu przy użyciu jednej lub wielu maszyn, obróbka wielu elemntów jednocześnie (np. malowanie tym samym kolorem), składanie wielu elementów w większe jednostki, itp. Jako dodatkowe zasoby, zwykle w ograniczonych ilościach, występują tutaj maszyny, siła robocza (ludzie) i czas. Matematyczne modele tych zagadnień są przedmiotem zainteresowania teorii szeregowania, a w ogólności badań operacyj-

nych, [5]. Niestety, nawet najprostsze zadania w tej dziedzinie są NP-zupełne, musimy więc korzystać z metod heurystycznych. W odróżnieniu jednak od teorio-grafowych problemów NP-zupełnych, dysponujemy tutaj metodami pseudo-wielomianowymi (np. wykorzystującymi zasadę programowania dynamicznego) czyli takimi, które dla niewielkich wartości liczbowych parametrów działają bardzo efektywnie.

W trakcie wykładu, powyższe rozważania ogólne zostaną zilustrowane konkretnymi przykładami.

3. Co nowego piszczy w kopcach, czyli o samoorganizacji danych i amortyzowaniu kosztów.

W tej części wykładu omówimy równocześnie różne implementacje klasycznego algorytmu wyznaczania najkrótszych dróg w sieci (zadanie to jest rozwiązywane w systemach omówionych w poprzedniej części) oraz różne wersje struktury danych zwanej kopcem (a czasem stogiem, ang. heap). Opiszemy optymalną realizację algorytmu Dijkstry przy pomocy kopców Fibonacciego a po drodze zilustrujemy dwa nowe zjawiska, które pojawiły się ostatnio w projektowaniu i analizie struktur danych: dane samoorganizujące się i zamortyzowany koszt obliczeń.

3.1. Kopce

Kopcem nazywamy strukturę danych złożoną z rodziny elementów, którym są przyporządkowane wagi K. Zakładamy, że na kopcu H można wykonywać następujące operacje:

- insert(v,H) - dołącz element v do kopca H, który nie zawiera v,
- delet_min(H) - odszukaj i usuń z H element o najmniejszej wadze; jeśli kopiec H jest pusty, to zwróć null;
- make_heap(S) - utwórz kopiec złożony z elementów zbioru S.

Dodatkowo, mogą być wykonywane operacje:

- find_min(H) - znajdź w H element o najmniejszej wadze, ale nie usuwaj go,
- delete(v,H) - usuń element v z kopca H,
- meld(H_1,H_2) - zastąp rozłączne kopce H_1 i H_2 przez jeden nowy kopiec.

J.W.J.Williams, który wprowadził pojęcie kopca w 1964 roku, podał jego realizację w tablicy. Następnie, zastosował tę strukturę danych w algorytmie heapsort, który jest nadal jedną z najefektywniejszych metod porządkowania. Knuth, w trzecim tomie swojej biblii o sztuce programowania, nazywa kopce *kolejkami priorytetowymi*.

Najczęściej, kopce są realizowane za pomocą drzew (w szczególności binarnych, w których elementy znajdują się w wierzchołkach i każdy wierzchołek v, który nie jest korzeniem spełnia tzw.

warunek kopca: $\quad K(p(v)) \leq K(v)$,

gdzie $p(v)$ jest ojcem wierzchołka v. Wynika stąd, że element o najmniejszej wadze znajduje się w korzeniu drzewa. Zatem czas wykonania operacji find_min(H) w dowolnym kopcu H jest stały $O(1)$, czyli nie zależy od liczby elementów w H. Czasy wakonywania innych operacji zależą od struktury drzewa.

Załóżmy, że kopiec jest zbudowany na d-arnym drzewie T, tj. każdy wierzchołek tego drzewa z wyjątkiem jednego ma d dzieci lub żadnego. Jeśli drzewo takie zawiera n wierzchołków, to jego wysokość wynosi $\log_d n+O(1)$. Wynikają stąd następujące oszacowania pracochłonności poszczególnych operacji. Dołączenie elementu trwa $O(\log_d n)$, gdyż nowy element podwieszony do drzewa może przebyć drogę aż do korzenia. Usunięcie zaś elementu, w tym także elementu minimalnego, może wymagać aż $O(d\log_d n)$ operacji - d operacji potrzeba do znalezienia elementu o najmniejszej wadze wśród dzieci ustalonego wierzchołka, a takich kroków przy usuwaniu trzeba czasem wykonać $O(\log_d n)$.

W punkcie 3.2.2, omawiamy impelementację algorytmu Dijkstry z użyciem d-kopców. Następnie, w par. 3.3 krótko wspominamy o dwóch innych rodzajach kopców, a w par. 3.4 opisujemy kopce Fibonacciego oraz ich zastosowanie w realizacji algorytmu Dijkstry.

W dalszej części, drzewo z korzeniem jest reprezentowane przez rodzinę $(p(v)$: v jest wierzchołkiem drzewa, różnym od korzenia), gdzie $p(v)$ jest *ojcem* wierzchołka v, czyli bezpośrednim poprzednikiem na drodze z korzenia do v. W tej *rodzinnej* terminologii, v nazywamy *dzieckiem* wierzchołka $p(v)$.

3.2. Algorytm Dijkstry

Dijkstra podał w jednej pracy [6] (por. także [11] i [16]) zachłanną metodę otrzymywania w sieci z nieujemnymi wagami zarówno najkrótszego drzewa jak i drzewa najkrótszych dróg z ustalonego wierzchołka. (Te na pozór podobnie brzmiące problemy mają zazwyczaj różne rozwiązania, nawet w sieciach symetrycznych, co umyka uwadze wielu osób.) Zajmiemy się tym drugim problemem. Dana jest sieć skierowana $D_d=(V,U;d)$, gdzie $D=(V,U)$ jest digrafem a $d:U \Rightarrow R_{\geq 0}$ - jest nieujemną funkcją długości łuków. Zakładamy, że sieć D_d jest reprezentowana przez zbiór list $(N^+(v): v\in V)$, gdzie $N^+(v)$ jest listą następników wierzchołka v. Z kolei, każdy element $w\in N^+(v)$ jest reprezentowany przez rekord zawierający przynajmniej trzy pola, w których są umieszczone: w, $d(v,w)$ oraz wskaźnik do następnego elementu listy. Nasz problem, który oznaczamy przez $SP(s,V)$ polega na znalezieniu w sieci D_d długości najkrótszych dróg z ustalonego wierzchołka s do wszystkich pozostałych wierzchołków w sieci oraz wierzchołków tworzących te drogi. Dowodzi się, że rozwiązaniem problemu $SP(s,V)$ jest drzewo skierowane T_s z korzeniem w s takie, że dla każdego $v\in V$, $v\neq s$, droga z s do v w T_s jest najkrótszą drogą z s do v w G_d. Algorytm Dijkstry wyznacza drzewo T_s (czyli określa wielkości $p(v)$, $v\in V$) w następujący sposób ($odl(v)$ jest długością najkrótszej drogi z korzenia s do wierzchołka v w rozwiązaniu częściowym):

Algorytm Dijkstry

Krok 0 (początkowy).

Przejmujemy: $Odl(s) \Leftarrow 0$ i $p(s) \Leftarrow 0$;
$Odl(v) \Leftarrow \infty$ i $p(v) \Leftarrow -1$ dla $v\in V$ i $v\neq s$;
$W \Leftarrow V-(s)$, $r \Leftarrow s$;

Krok 1 (iteracyjny).

1.1 Dla $v\in N^+(r)\cap W$ wykonaj
jeśli $odl(r) + d(r,v) < odl(v)$
to $odl(v) \Leftarrow odl(r) + d(r,v)$ i $p(v) \Leftarrow r$

1.2. Niech $min(odl(v): v\in W) = odl(w)$. Przyjmij $r \Leftarrow w$.

Słownie, w algorytmie Dijkstry drzewo najkrótszych dróg jest tworzone począwszy od wierzchołka s przez dołączanie w każdym kroku iteracyjnym wierzchołka najbliższego spośród wierzchołków jeszcze nie należących do rozwiązania. W szczególności, w kroku 1.1 są korygowane odległości do wierzchołków znajdujących się poza rozwiązaniem, a w kroku 1.2 jest wyzna-

czany najbliższy wierzchołek.

3.2.1. Implementacja algorytmu Dijkstry z użyciem tylko tablic.

Jeśli poza opisanymi wyżej wielkościami *odl* i *p* nie korzystamy z żadnych innych struktur danych, to całkowita złożoność algorytmu Dijkstry wynosi $O(n^2)$, por. [11], [15] i [16].

3.2.2. Implementacja algorytmu Dijkstry z użyciem kopca.

D.B.Johnson (por. [17]) zaproponował użycie dodatkowego d-kopca (z odpowiednio dobranym d) do pamiętania wierzchołków, które nie należą do rozwiązania, wraz z *odl* jako ich wagami. A oto realizacja algorytmu Dijkstry z użyciem kopca H.

```
procedure SP(s,V);
begin
   for v∈V do begin  odl(v) ⇐ ∞;   p(v) ⇐ -1 end;
   odl(s) ⇐ 0; p(s) ⇐ 0;
   H ⇐ make_heap({ });
   r ⇐ s;
   while r ≠ null do begin
1.    for v∈N+(r) do
 1.1.    if odl(r) + d(r,v) < odl(v) then begin
 1.2.       odl(v) ⇐ odl(r) + d(r,v);
            p(v) ⇐ r;
 1.3.       if v∈H then insert(v,H)
.4.                 else sift_up(v,H)
         end;  { if }
2.    r ⇐ delete_min(H)
   end  { while }
end.  { SP }
```

Krok 1.1 algorytmu realizuje instrukcja 1 a Krok 1.2 – instrukcja 2. Wyjaśnienia wymaga nieopisana dotychczas operacja sift_up(v,H). Otóż, w algorytmie Dijkstry, jeśli jest korygowana odległość do wierzchołka v (w inst. 1.2), to należy uwzględnić także ewentualną zmianę położenia tego wierzchołka w kopcu H. Jeśli v nie należy do kopca H, to dołączamy go w instrukcji 1.3, a jeśli należy, to musimy sprawdzić, czy odległość zmniejszona w instrukcji 1.2 nie spowoduje naruszenia warunku kopca. Za to jest odpowiedzialna operacja sift_up(v,H), która podnosi wierzchołek v w kopcu H tak wysoko jak pozwala

na to bieżąca wartość jego odległości. Operacja sift_up(v,H) wykonywana w n elementowym d-kopcu wamaga co najwyżej $O(\log_d n)$ operacji.

W powyższej procedurze jest wykonywanych $n-1$ operacji delet_min, $n-1$ operacji insert (gdyż każdy wierzchołek dokładnie raz trafia do kopca i raz jest usuwany jako minimalny), oraz co najwyżej $m-n+1$ razy sift_up (jest to liczba łuków w sieci bez drzewa T_s). Zatem całkowity koszt działania procedury SP wynosi $O(nd\log_d n+m\log_d n)$. Przyjmijmy $d=2+m/n$. W tym przypadku, procedura ma złożoność $O(m\log_{(2+m/n)} n)$. Zatem, dla digrafów rzadkich, czyli gdy m jest liniową funkcją zmiennej n, procedura SP działa w czasie $O(n\log n)$, natomiast dla digrafów pełnych jej złożoność jest porównywalna ze złożonością implementacji tablicowej.

3.2.3. Zamortyzowany koszt operacji.

Analiza implementacji algorytmu Dijkstry za pomocą d-kopców, którą z konieczności podaliśmy w wielkim skrócie, jest typowym rozumowaniem w analizie algorytmów, w przypadku rozpatrywania *pesymistycznej złożoności obliczeniowej*, czyli najmniej korzystnego układu danych i przebiegu obliczeń. Najpierw podaliśmy pesymistyczną złożoność pojedynczych operacji wykonywanych w algorytmie, a następnie, po wyznaczeniu górnych oszacowań na liczby poszczególnych operacji, obliczyliśmy całkowity koszt algorytmu sumując koszty wszystkich operacji.

Jak wykazują obliczenia testowe wykonywane z algorytmami, w wielu przypadkach jest to metoda bardzo zgrubna. Powodów tego należy upatrywać w tym, że nie wszystkie operacje tego samego typu kosztują tyle ile wynosi pesymistyczne oszacowanie ich czasu działania. Dla przykładu, podaliśmy, że koszt dołączenia elementu do d-kopca wynosi $O(\log_d n)$, gdy kopiec składa się z n elementów. Ale przecież nasz kopiec na początku jest pusty, a wszystkie elementy zawiera dopiero po $n-1$ krokach. Zatem, złożoność tych pierwszych umieszczeń w kopcu oszacowaliśmy daleko na wyrost.

Nie potrafimy jednak liczyć całkowitego kosztu algorytmu bez rozbijania go na prostsze operacje. Istnieje jednak pewna droga okrężna. Zamiast brania pod uwagę pesymistycznego kosztu wykonania pojedynczej operacji, możemy założyć, że tych operacji wykonujemy $O(n+m)$, tak jak jest w rzeczywistości w przypadku algorytmu Dijkstry, i policzyć średni koszt jednej ope-

racji w tym ciągu. By odróżnić tę analizę złożoności od analizy probabilistycznej, będziemy mówić o zamortyzowanym koszcie pojedynczej operacji, gdzie amortyzacja ma miejsce względem całego ciągu operacji. Niestety taka analiza nie jest możliwa w odniesieniu do d-kopców. Dopiero kopce rozbudowane o dodatkowe połączenia i operacje umożliwiają otrzymywanie takich oszacowań.

*Formalnym uzasadnieniem analizy kosztów zamortyzowanych może być następujące rozumowanie.

Załóżmy, że na rodzinie kopców X mamy do wykonania k operacji, gdzie k jest rzędu n, liczby elementów w kopcach. Przyjmijmy, że nieznany czas wykonania i-tej operacji wynosi t_i. Z wykonaniem tej operacji związmy *potencjał* Φ_i, który jest nieujemny i dodatkowo $\Phi_0=0$. Wielkość $\Phi_i-\Phi_{i-1}$ jest więc skokiem potencjału po wykonaniu i-tej operacji. *Koszt zamortyzowany* i-tej operacji definiujemy jako

$$a_i=t_i+\Phi_i-\Phi_{i-1},$$

a wtedy całkowity koszt zamortyzowany wykonania wszystkich k operacji wynosi

$$\Sigma a_i = \Sigma t_i + \Phi_k - \Phi_0.$$

Na podstawie założeń o funkcji potencjału, otrzymujemy następujące oszacowanie rzeczywistego kosztu wykonania k operacji

$$\Sigma a_i \geq \Sigma t_i,$$

czyli sumaryczny koszt zamortyzowany jest górnym oszacowaniem rzeczywistego kosztu algorytmu. Zasadniczy problem w analizie kosztów zamortyzowanych polega na odpowiednim doborze funkcji potencjałów umożliwiającej taką analizę.*

3.3. Samoorganizujące się struktury danych.

Koszty zamortyzowane udaje się wyliczyć dla tzw. samoorganizujących się struktur danych. Chociaż to pojęcie pojawiło się stosunkowo niedawno, bo w 1983 roku, samoorganizujące się struktury danych były stosowane już wcześniej. Ich podstawową cechą jest:

- uwzględnianie w sposobach wykonywania działań algorytmu wpływu, jakie mogą mieć na następne kroki, i

- wykonywanie pewnych operacji na strukturach danych wyłącznie na potrzeby przyszłych operacji algorytmu.

Pierwszą cechę można odnaleźć na przykład w 2-kolejkach stosowanych w implementacji algorytmu Bellmana-Moora dla problemu SP(s,V) w sieci o dowolnych długościach łuków, por. [16]. W zależności od częstości przyglądania, wierzchołki są dołączane do listy kandydatów na początku lub na końcu. Dzięki temu, w następnych krokach algorytmu, pierwszeństwo mają wierzchołki częściej odwiedzane, czyli te, które najbardziej wpływają na zmiany długości dróg.

Za drugą zaś cechę można uznać kompresję ścieżek wykonywaną w trakcie realizacji ciągu operacji FIND-UNION, por. [1]. Po wykonaniu operacji FIND(i), każdy wierzchołek j należący do ścieżki od i do korzenia jest podłączany bezpośrednio do korzenia. W ten sposób skracamy czas przyszłej operacji FIND(j).

W obu przypadkach, algorytmy, w których są wykorzystywane te struktury danych należą do najefektywniejszych w swojej klasie.

W wielu algorytmach, w których są stosowane kopce, podstawową operacją jest ich łączenie, meld(H_1,H_2). Łączenie d-kopców nie jest łatwe, z powodu konieczności zapewnienia stałej i takiej samej dla wszystkich wierzchołków liczby dzieci.

Kopcami, które pozwalają na łatwe i efektywne łączenie są kopce lewicowe (ang. leftist) i skośne (ang. skew), por. [17] i [3]. Te drugie są ponadto przykładem samoorganizujących się struktur danych.

Kopiec *lewicowy* jest kopcem zrealizowanym na drzewie lewicowym. Z kolei, *drzewo lewicowe* jest zbudowane na pełnym drzewie binarnym, czyli takim, w którym każdy wierzchołek ma 2 lub 0 dzieci. Elementy zbioru, który ma być reprezentowany, są umieszczane tylko w wierzchołkach pośrednich. Z każdym wierzchołkiem związany jest jego *rząd* $r(v)$, którym jest długość najkrótszej drogi z v do wierzchołka wiszącego. Pełne drzewo binarne jest drzewem lewicowym jeśli ma następującą własność: najbardziej skrajna droga z korzenia jest najkrótszą drogą z korzenia do wierzchołka wiszącego.

Łączenie dwóch kopców lewicowych H_1 i H_2 odbywa się w następujący sposób:
- scalić prawe drogi z H_1 i H_2 ustawiając ich wierzchołki w kolejności niemalejących wag,
- uaktualnić rzędy wierzchołków na otrzymanej drodze,

- wrócić do drzewa lewicowego przez ewentualną zamianę lewego poddrzewa z prawym na najbardziej prawej drodze.

Operacja ta trwa O(logn), gdyż jest to także oszacowanim długości prawych dróg w drzewach lewicowych.

Operacje insert, delete_min mogą być wykonane za pomocą łączenia odpowiednich kopców, dlatego trwają tyle samo. Podobną złożoność ma delete, ale dowód tego faktu nie jest już taki prosty. Operacje make_heap i find_min są tak samo pracochłonne jak w przypadku *d*-kopców.

Kopiec *skośny* różni się od kopca lewicowego tym, że nie wymaga liczenia i przechowywania w wierzchołka ich rzędów. Zamiast tego, w trakcie łączenia kopców, po scaleniu prawych dróg, zamienia się rolami lewy z prawym następnikiem dla każdego wierzchołka na najbardziej prawej drodze. Jest to operacja wykonywana przede wszystkim na potrzeby następnych operacji na danych i umożliwia analizę kosztów zamortyzowanych. To uproszczenie polegające na rezygnacji z funkcji rzędu ma wpływ na pesymistyczny koszt wykonywania operacji, ale koszty zamortyzowane pozostają takie same jak dla kopców lewicowych.

W trakcie wykładu, pojęcia struktur samoorganizujących się i sposób liczenia kosztów zamortyzowanych zostaną zilustrowane na przykładzie kopca skośnego,

3.4. Kopce Fibonacciego.

Kopce lewicowe i skośne nie pozwalają na dalsze zwiększenie efektywności algorytmu Dijkstry.

Przed dalszą dyskusją wprowadźmy jeszcze jedną operację wykonywaną na kopcach.

decrease_key(Δ,*v*,*H*) - poleca zmniejszyć wagę elementu *v* o Δ w kopcu *H*.

Tę operację możemy zastosować w procedurze SP usuwając sift_up i łącząc kroki 1.2 i 1.4. Zatem algorytm Dijkstry można zrealizować wykonując następujące operacje: make_heap, n-1 razy delete_min, n-1 razy insert oraz co najwyżej *m* razy decrease_key. Struktura danych, którą teraz opiszemy, pozwala wykonywać delete_min w czasie O(logn) a pozostałe operacje - w czasie stałym, tj. O(1). Koszt delete_min jest kosztem zamortyzowanym, czyli średnim względem ciągu O(n) takich operacji.

Kopiec *Fibonacciego H*, por. [7], zwany dalej w skrócie

F-kopcem, jest rodziną rozłącznych drzew spełniających warunek kopca. Dodatkowo, na elementach H są opisane dwie funkcje, a drzewa - uzupełnione o dadatkowe połączenia między elementami.

Liczbę dzieci wierzchołka v nazywamy jego *rzędem* i oznaczamy przez $r(v)$. Ponadto, każdy wierzchołek może być oznaczony lub nieoznaczony. Operacje na F-kopcach zapewniają, że wierzchołek o n następnikach ma rząd $O(\log n)$.

F-kopiec ma następującą impelemntację: Każdy wierzchołek zawiera wskaźnik do ojca i wskaźnik do jednego ze swoich dzieci. Wszystkie dzieci jednego wierzchołka tworzą dwukierunkową listę cykliczną. Każdy wierzchołek zawiera swój rząd i cechę. Wszystkie korzenie drzew tworzą także dwukierunkową listę cykliczną. Dostęp do kopca jest za pośrednictwem wskaźnika do korzenia zawierającego element o najmniejszej wadze. Dwukierunkowe listy cykliczne pozwalają usuwać z nich elementy i łączyć takie listy w czasie $O(1)$.

Operacja find_min(H) na F-kopcu H kosztuje $O(1)$. Łączenie dwóch F-kopców H_1 i H_2 wykonujemy przez połączenie list korzeni obu kopców i umieszczenie wskaźnika na mniejszym z elementów wskazywanym w H_1 i H_2. Zatem łączenie trwa $O(1)$. By włączyć nowy element v do F-kopca H, wystarczy utworzyć z tego elementu nowy kopiec H' i wykonać meld(H,H'). Zatem koszt insert(v,H) wynosi także $O(1)$. Wynika stąd również, że F-kopiec złożony z n elementów można utworzyć w czasie $O(n)$.

Operację decrease_key(Δ,v,H) wykonujemy następująco. Redukujemy najpierw wagę wierzchołka v o Δ. Jeśli v jest korzeniem, to sprawdzamy tylko czy nie otrzymaliśmy wierzchołka z mniejszą wagą. Jeśli v nie jest korzeniem, to odłączamy poddrzewo zaczepine w v od drzewa do którego należy. Wymaga to usunięcia v z listy dzieci wierzchołka $p(v)$ i zmiany wskaźnika z ojca na null. To odłączenie powoduje, że poddrzewo z korzeniem v staje się nowym drzewem w kopcu H. Zmniejszona waga w wierzchołku v nie psuje warunku kopca w poddrzewie zaczepinym w v, wystarczy więc porównać ją z dotychczasową minimalną wagą całego kopca. Podobnie można zrealizować operację delete. Zatem, czas wykonania obu operacji jest także stały, czyli $O(1)$.

Najbardziej pracochłonną operacją wykonywaną na F-kopcach jest delete_min. Powód jest dość oczywisty: po usunięciu elementu minimalnego, następny element minimalny może być gdziekolwiek w kopcu. Nie opiszemy tutaj jak ta operacja jest wykonywana - na wykładzie zostanie to pokazane na przykładzie.

Czas jej działania jest liczony w wersji zamortyzowanej, czyli z użyciem potencjałów i wynosi O(logn). Potencjałem w tym przypadku jest liczba drzew w kopcu. Zauważmy, że operacja insert zwiększa potencjał o 1 , a make_heap, find_min i meld nie zwiększają potencjału.

Źródłem nazwy omawianych tutaj kopców jest fakt, że wierzchołek rzędu k ma co najmniej F_{k+2} wszystkich następników, gdzie F_{k+2} jest $(k+2)$-gą liczbą Fibonacciego. Fakt ten jest wykorzystywany w dowodzie oszacowania czasu wykonania operacji delete_min.

Na koniec wróćmy do algorytmu Dijkstry. Stosując F-kopce do przechowywania wag *odl*, otrzymujemy realizację, której czas działania wynosi $O(n\log n+m)$. Zauważmy w tym miejscu, że jest to (w sensie asymptotycznym) kres możliwości dla tego algorytmu. Rozpatrzmy bowiem sieć będącą gwiazdą z n ramionami, którym przyporządkowano wagi x_i i zastosujmy do niej algorytm Dijkstry wybierając za wierzchołek s centrum gwiazdy. Kolejność w jakiej algorytm będzie przeglądał pozostałe wierzchołki wyznacza uporządkowanie elementów x_i. Z kolei wiadomo, że $n\log n$ jest dolnym oszacowaniem złożoności porządkowania n liczb. Składnika m także nie można zmniejszyć, gdyż wszystkie łuki w sieci mogą być brane pod uwagę w obliczeniach. Oczywiście te uwagi odnoszą się do algorytmu Dijkstry. Nie wyklucza to jednak możliwości istnienia szybszej metody dla problemu SP(s,F).

Podziękowania

Autor dziękuje Panu Jerzemu Żakowi z firmy ab:c Informatik GmbH (Ulm, RFN) za długie dyskusje nad zagadnieniem, które jest opisane w części drugiej.

Opracowanie to powstało w trakcie pobytu autora w grupie Profesora Rolfa Moehringa (Fachbereich Mathematik, Technische Univeritaet Berlin), latem 1989 roku.

Literatura

[1] A.V.Aho, J.E.Hopcroft, J.D.Ullman, *Projektowanie i Analiza Algorytmów Komputerowych*, PWN, Warszawa 1983.

[2] L.Banachowski, A.Kreczmar, *Elementy Analizy Algorytmów*, WNT, Warszawa 1982.

[3] L.Banachowski, A.Kreczmar, W.Rytter, *Analiza Algorytmów i Struktur Danych*, WNT, Warszawa 1987.
[4] J.Bentley, *Programming Perls*, Addison-Wesley, Reading Mass., 1986.
[5] J.Błażewicz, W.Cellary, R.Słowiński i J.Węglarz, *Badania Operacyjne dla Informatyków*, WNT, Warszawa 1984.
[6] E.W. Dijkstra, A note on two problems in connection with graphs, *Num. Math.* 1(1959), 269-271.
[7] M.L.Fredman, R.E.Tarjan, Fibonacci heaps and their use in improved network optimization algorithms, *J.ACM* 34(1987), 596-615.
[8] R.L.Graham, D.E.Knuth, O.Patashnik, *Concrete Mathematics*, Addison-Wesley, Reading Mass. 1988.
[9] B.W.Kernighan, P.J.Plauger, *Elements of Programming Style*, Addison-Wesley, Reading Mass. 1987.
[10] B.W.Kernighan, P.J.Plauger, *Software Tools in Pascal*, Addison-Wesley, Reading Mass. 1981.
[11] W.Lipski, Jr., *Kombinatoryka dla Programistów*, WNT, Warszawa 1989 (Drugie wydanie).
[12] P.G.Ranky, *Computer Integrated Manufacturing*, Prentice-Hall, Englewood Cliffs NJ 1986.
[13] A.Roobeck, M.Roscam Abbing, The international implications of CIM, *Int. J. CIM* 1(1988), 3-12.
[14] A.W.Scheer, *CIM - Computer Integrated Manufacturing. Computer Steered Industry*, Springer-Verlag, Berlin, Heidelberg 1986.
[15] M.M.Sysło, *Optymalizacja Kombinatoryczna*, Rozdz. 3 w: W.Kasprzak (red.), *Optymalizacja Dyskretna. Ekonomiczne Zastosowania*, PWE, Warszawa 1984, str. 71-140.
[16] M.M.Sysło, N.Deo, J.S.Kowalik, *Algorytmy Optymalizacji Dyskretnej z Programami w Pascalu*, PWN, Warszawa (tłumaczenie w przygotowaniu).
[17] R.E.Tarjan, *Data Structures and Network Algorithms*, SIAM, Philadelphia Pa. 1983.

Jesienna Szkoła PTI
Mrągowo, listopad 1989

LOGIKA I INFORMATYKA

Andrzej Szałas
Instytut Informatyki, Uniwersytet Warszawski
00-901 Warszawa, PKiN 850

1. Wstęp

Współczesna informatyka ma obecnie coraz silniejszy i coraz bardziej podstawowy wpływ na większość dziedzin działalności człowieka. Wpływ ten nie ogranicza się jedynie do nauki i techniki. Obejmuje on także szeroką sferę związaną ze zdrowiem, usługami, bankowością itd. Jednocześnie informatyka jest nauką bardzo młodą, a jej centralne zagadnienia, bezpośrednio związane z programowaniem jako takim, nie mają jeszcze wystarczająco silnego oparcia ani w tradycji, ani w dobrze ugruntowanych metodach formalnych, które miałyby natychmiastowe, a zarazem nietrywialne zastosowania w inżynierii oprogramowania. Z drugiej strony każdy, kto napisał choćby jeden program, lub korzystał z dużych produktów programistycznych (jak kompilatory, systemy operacyjne itd. itd.), wie jak bardzo proces tworzenia oprogramowania jest podatny na błędy. Zrozumiałe jest zatem dążenie do uzyskania rachunku formalnego, którego praktyczne stosowanie podniosłoby niezawodność oprogramowania. W tradycyjnych naukach inżynierskich rolę takiego rachunku formalnego spełnia np. analiza matematyczna, której aparat pozwala w wielu przypadkach na teoretyczne sprawdzenie poprawności przyjętych rozwiązań jeszcze przed przeprowadzeniem doświadczeń na prototypach. Wiele wskazuje na to, że w informatyce podobną rolę odegra logika. Dlatego też w dziedzinie logik informatycznych trwają od lat bardzo intensywne badania. Wprawdzie wyniki, uzyskiwane przede wszystkim w środowiskach akademickich, nie osiągnęły jeszcze w pełni formy jaką powinien mieć dobrze rozwinięty i ugruntowany aparat inżynierski, są one już na tyle znaczące iż ich omówieniu warto poświęcić nieco czasu. Taki aparat formalny może się bowiem rozwinąć jedynie podczas prób stosowania wyników teoretycznych w praktyce, niezależnie czy uzyskiwane przy tym doświadczenia będą pozytywne, czy nie.

Prace nad logikami programów rozwijały się w ścisłym powiązaniu z rozwojem metod i narzędzi programowania. Intuicyjne podejście do specyfikacji semantyki nowych konstrukcji programistycznych poniosło bowiem fiasko już w latach sześćdziesiątych (por. np. przypadek raportu Algolu 60). Okazało się bowiem, że odwoływanie się do intuicji zawsze dotyczy intuicji autorów projektów i dokumentacji, a nieporównanie rzadziej czytelników, mających inny życiorys zawodowy, pracujących najczęściej w innym środowisku. Co więcej, odwołania do

intuicji często miały charakter niezamierzony, wynikały z niedostrzegania przez autorów luk i nieścisłości we własnym rozumowaniu. Powtórzono wiele błędów popełnianych w historii rozwoju matematyki. Obrazowym przykładem jest np. następujące powtórzenie paradoksu Russela: rozważmy język programowania, w którym funkcje mogą być parametrami innych funkcji. Semantyka jest "intuicyjnie jasna" - każdorazowe wywołanie funkcji będącej parametrem formalnym zastępujemy wołaniem odpowiedniej funkcji danej jako parametr aktualny. Zdefinijmy teraz funkcję g(f), na którą składa się wykonanie instrukcji **if** f(f) = 0 **then** 1 **else** 0 **fi** (jeśli wartością funkcji f zastosowanej do samej siebie jest 0, to wartością g jest 1, w przeciwnym przypadku wartością g jest 0). Powyższa definicja funkcji g prowadzi do sprzeczności: g(g) = 0 *wttw, gdy* (wtedy i tylko wtedy, gdy) g(g) = 1. Sprzeczność tę można rozwiązać na wiele różnych sposobów (od dość topornego wprowadzenia zakazu definiowania w języku programowania takich właśnie funkcji, po znacznie subtelniejsze rozwiązania oferowane przez teorię typów, lub dopuszczenie jako parametrów aktualnych jedynie funkcji obliczalnych).

W wyniku prac nad technikami specyfikacji wypracowano wiele konkurencyjnych, bądź uzupełniających się podejść. W ich liczbie wymienić można np. podejście operacyjne, algebraiczne, denotacyjne i - oczywiście - logiczne. Ważnym problemem była nie tylko specyfikacja semantyki programów, ale także identyfikacja, wyrażanie i dowodzenie ich własności. Najbardziej naturalnym narzędziem do realizacji wszystkich tych celów w ramach jednego formalizmu są logiki. Użyliśmy tu liczby mnogiej, gdyż np. logiki wygodne w zastosowaniach do programów sekwencyjnych ponoszą fiasko w przypadku programów niedeterministycznych bądź współbieżnych. Odwrotnie, logiki obejmujące współbieżność mogą się okazać zbyt ogólnym narzędziem, przegrywając rywalizację z logikami skromniejszymi, za to wyspecjalizowanymi do rozwiązywania tych czy innych zagadnień. Podstawową rolę w tym wykładzie spełniać będzie logika klasyczna. Formuły klasyczne odzwierciedlają "statyczną" (niezależną od zmian środowiska) naturę wyrażanych pojęć. Można za ich pośrednictwem wyrażać wiele (nie wszystkie!) własności struktur danych. Z drugiej strony, dynamiczny charakter zjawisk występujących w informatyce wymaga rozszerzenia logiki klasycznej o nowe pojęcia. Główny nacisk położymy na omówienie rozmaitych rozszerzeń związanych z linią rozwojową narzędzi programistycznych determinizm (sekwencyjność) - niedeterminizm - współbieżność. Wskażemy także pewne istotne problemy metodologiczne, których źródłem są te rozszerzenia.

Rola logik w informatyce nie ogranicza się oczywiście jedynie do wspomnianych wyżej zagadnień, występuje ona bowiem w sposób mniej lub bardziej bezpośredni praktycznie we wszystkich warstwach oprogramowania - od bramek logicznych po programowanie w logice i sztuczną inteligencję. Omówienie wszystkich tych aspektów wymagałoby jednak obszernej monografii, znacznie przekraczającej rozmiary tego wykładu.

2. Co to jest logika?

W poniższym rozdziale omówimy pojęcie logiki. Przypomnijmy sobie najpierw jaką rolę odgrywa logika w procesie porządkowania rozumowania prowadzonego przez człowieka. Aby rozumowanie to nie było jałowe, najpierw musimy zdecydować się na jego przedmiot, ustalając jednocześnie język, w którym będzie przeprowadzane. Kolejnym krokiem jest nadanie ściśle

określonego znaczenia poszczególnym pojęciom przyjętego języka, unikając w ten sposób niejednoznaczności i wynikających z nich nieporozumień. Następnie musimy powiedzieć, jakie sądy (oceny) mogą być wypowiadane w ustalonym języku (na temat przyjętego przedmiotu rozumowania) i które z nich przyjmujemy za prawdziwe, a które za fałszywe. Możemy teraz patrzeć na przedmiot rozumowania jedynie poprzez pryzmat prawdziwości (fałszywości) wypowiadanych sądów. Dokonując tego typu abstrakcji definiujemy właśnie logikę.

2.1 Definicja: *Logiką* nazywać będziemy trójkę $L = <F,I,Sat>$, gdzie
(i) F jest *zbiorem formuł poprawnie zbudowanych* (sądów wypowiadanych w języku logiki L),
(ii) I - jest *klasą dopuszczalnych interpretacji* (definiujących możliwe znaczenia przedmiotów rozumowania),
(iii) *Sat* jest *relacją spełnialności* (określającej prawdziwość sądów przy danej interpretacji) - dla interpretacji I oraz formuły A: I *Sat* A oznacza, iż formuła (sąd) A jest prawdziwa w interpretacji I (mówimy też wtedy, że I jest *modelem* dla A).
Powiemy, że formuła A jest *konsekwencją* (semantyczną) zbioru formuł Z wttw, gdy każdy model dla Z (tzn. dla wszystkich formuł należących do Z) jest też modelem dla A. Zbiór konsekwencji zbioru Z oznaczamy przez Cn(Z).
Tautologią logiki L nazywać będziemy formułę prawdziwą w każdej interpretacji dopuszczalnej. Zbiór tautologii jest więc zbiorem konsekwencji zbioru pustego - Cn(Ø).//

2.2 Przykład: Przedmiotem rozważań klasycznej logiki zdań (oznaczanej dalej przez L_0) są związki między zdaniami łączonymi spójnikami negacji ($\sim$) oraz koniunkcji ($\wedge$). Zauważmy, że inne klasyczne spójniki zdaniowe można łatwo wydefiniować używając negacji i koniunkcji, np. alternatywę $p \vee q$ definiujemy jako $\sim(\sim p \wedge \sim q)$, implikację $p \Rightarrow q$ - jako $(\sim p) \vee q)$. Zbiór *formuł poprawnie zbudowanych*, F_0, definiujemy indukcyjnie:
(i) wyróżniamy zbiór V_0 *zmiennych zdaniowych* (w programowaniu reprezentują one typ *Boolean*, czasem oznaczany także jako *Logical*) i przyjmujemy, że zmienne zdaniowe są formułami;
(ii) jeśli p i q są formułami, to są nimi także $\sim p$ oraz $p \wedge q$.
Ponieważ interesuje nas (w L_0) przede wszystkim wartość logiczna zdań złożonych zależna jedynie od wartości logicznej zdań składowych, przyjmiemy że *interpretacjami dopuszczalnymi* (I_0) są wartościowania zmiennych zdaniowych, $I_0 = \{v_0 \mid v_0: V_0 \to \{true, false\}\}$. *Wartość formuły zdaniowej* p dla danego wartościowania v_0, $p(v_0)$, definiujemy indukcyjnie:
(i') dla zmiennej zdaniowej p: $p(v_0) =$ **true** wttw, gdy $v_0(p) =$ **true**;
(ii') dla dowolnej formuły q: $(\sim q)(v_0) =$ **true** wttw, gdy $q(v_0) =$ **false**;
(iii') dla dowolnych formuł p i q: $(p \wedge q)(v_0) =$ **true** wttw, gdy $p(v_0) =$ **true** oraz $q(v_0) =$ **true**.
Relację Sat_0 definiujemy następująco: v_0 *Sat* p wttw, gdy $p(v_0) =$ **true**. Warto zauważyć, że zarówno relacja Sat_0, jak i zbiór tautologii logiki L_0 są rozstrzygalne (tzn. istnieją algorytmy, które dla danych v_0 oraz p stwierdzają czy v_0 Sat_0 p (odpowiednio - czy p jest tautologią) - np. oparte na tzw. metodzie zero-jedynkowej).//

2.3 Przykład: Przedmiotem rozważań klasycznej logiki pierwszego rzędu, L_1, są własności funkcji i relacji określonych na pewnych zbiorach elementów, wyrażane przy pomocy kwantyfikatorów "dla każdego elementu" oraz "istnieje element". Zestaw funkcji i relacji (wraz ze zbiorem elementów) nazywamy *systemem relacyjnym*. Język logiki L_1, F_1, jest rozszerzeniem języka logiki L_0 (por. poprzedni przykład). Aby go zdefiniować, przyjmijmy, że dane są zbiory V (*zmiennych indywiduowych*), F (nazw funkcji, czyli *funktorów*) oraz P (nazw relacji, czyli *predykatów*). *Termem* nazywamy wyrażenie będące zmienną indywiduową, bądź zastosowaniem funktora do innych termów (postaci $f(t_1,...,t_n)$, gdy f jest funktorem n-argumentowym). Zbiór

formuł F_1 definiujemy teraz jako najmniejszy zbiór wyrażeń zawierający formuły klasycznego rachunku zdań (F_0 z poprzedniego przykładu), wszystkie formuły postaci $p(t_1,...,t_m)$, gdzie p jest m-argumentowym predykatem, a $t_1,...,t_m$ - termami, i zamknięty ze względu na następującą regułę:
(i) jeśli x jest zmienną indywiduową oraz A, B są formułami, to $\sim A$, $A \wedge B$ oraz $(\forall x)A$ ("dla każdego x zachodzi A") są formułami.
Oczywiście kwantyfikator "istnieje x" można łatwo zdefiniować przy pomocy kwantyfikatora $\forall x$: $(\exists x)A$ definiujemy jako $(\sim(\forall x)\sim A)$. Podobnie rozważać można tzw. *wielosortową logikę pierwszego rzędu*, w której dopuszcza się zmienne różnych typów. Powyższe definicje łatwo rozszerzyć do wersji wielosortowej. Podkreślmy, iż rozszerzenia tego dokonuje się jedynie ze względu na wygodę - z metodologicznego i teoretycznego punktu widzenia nie rodzi ono żadnych nowych problemów.
Zmienną x występującą w formule A nazywamy *wolną*, jeśli nie znajduje się ona w zasięgu kwantyfikatora ($(\forall x)$ lub $(\exists x)$), zaś *związaną*, jeśli nie jest ona wolna.
Klasa interpretacji dopuszczalnych, I_1, obejmuje wszystkie systemy relacyjne. Niech $R = <D,F,P>$ będzie ustalonym systemem relacyjnym, w którym D jest zbiorem elementów, zaś F oraz P odpowiednio zbiorami funkcji i relacji (dla uproszczenia przyjmujemy konwencję notacyjną, w której funktory i predykaty mają te same nazwy co funkcje i relacje w konkretnym systemie relacyjnym - w ten sposób nazwa funktora wyznacza jego interpretację; jeśli będziemy chcieli dokonać rozróżnienia między funktorem, a funkcją z R, tą ostatnią opatrywać będziemy indeksem R). Niech teraz $v: VU V_0 -> DU\{true,false\}$ będzie wartościowaniem zmiennych indywiduowych i zdaniowych (te pierwsze przyjmują oczywiście wartości z D, zaś drugie - true lub false). *Wartość termu* t w systemie R przy wartościowaniu v, $t_R(v)$, definiujemy indukcyjnie:
(i') $x_R(v) = v(x)$ dla zmiennej indywiduowej x;
(ii') $f(t_1,...t_n)_R(v) = f_R(t_{1R}(v),...,t_{nR}(v))$, dla n-argumentowego funktora f, odpowiadającej mu funkcji f_R w systemie R, oraz termów $t_1,...,t_n$.
Wartość formuły A w systemie R przy wartościowaniu zmiennych indywiduowych i zdaniowych v, $A_R(v)$, definiujemy indukcyjnie:
(i") $q_R(v) = v(q)$ dla zmiennej zdaniowej q;
(ii") $p(t_1,...t_m)_R(v)$ = true wtw, gdy $(t_{1R}(v),...,t_{mR}(v))$ jest elementem relacji p_R, dla m-argumentowego predykatu p, odpowiadającej mu relacji p_R w systemie R, oraz termów $t_1,...,t_m$ (m-argumentowe relacje traktujemy tu, jak zwykle, jako zbiory m-tek elementów);
(iii") $(\sim A)_R(v)$ = true wtw, gdy $A_R(v)$ = false, dla formuły A;
(iv") $(A \wedge B)_R(v)$ = true wtw, gdy $A_R(v)$ = true i $B_R(v)$ = true, dla formuł A, B;
(v") $((\forall x)A)_R(v)$ = true wtw, gdy dla każdego wartościowania v' różniącego się od v co najwyżej wartością zmiennej x: $A_R(v')$ = true.
Relację spełnialności *Sat*$_1$ definujemy następująco: R *Sat*$_1$ A wtw, gdy dla dowolnego wartościowania v: $A_R(v)$ = true. Zauważmy, że zbiór tautologii logiki L_1 jest częściowo rozstrzygalny, tzn. istnieje algorytm, który dla danej formuły A działa następująco:
- jeśli A jest tautologią to algorytm stwierdza ten fakt, w przeciwnym przypadku stwierdza, że nie jest lub się zapętla. (Algorytm tego typu nazywać będziemy *częściowym*.) Natomiast dla problemu spełnialności nie istnieje nawet częściowy algorytm.//

Klasyczna logika pierwszego rzędu pełnić będzie w naszym wykładzie podstawową rolę. Toteż, choć można założyć, że jest powszechnie znana, przytoczyliśmy jej definicję po pierwsze aby oswoić się z przyjętymi konwencjami, a po drugie - aby wykład ten był możliwie samowystarczalny. Istnieje ogromna ilość podręczników logiki klasycznej, w których znaleźć można dokładne jej omówienie. Zainteresowanych czytelników odsyłamy np. do pozycji [EFT'84, Grz'75, Mon'76]. Bardzo skondensowane, lecz warte polecenia omówienie podstawowych zagadnień znaleźć można również w (mini)monografii [Lyn'78].

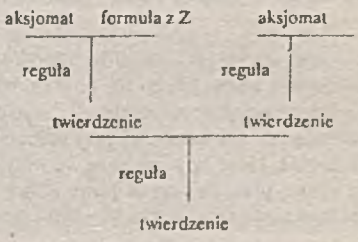
3. Systemy wnioskowania

Zauważny, że sama semantyczna definicja logiki nie dostarcza żadnych formalnych metod dowodzenia twierdzeń. Z drugiej strony, do przeprowadzania dowodów niezbędne jest jeszcze uzupełnienie logiki o takie metody. Do najpowszechniej przyjętych należą metody aksjomatyczne oparte na formalnych systemach wnioskowania (dowodzenia). Dowodzenie polega w nich na przyjęciu zestawu podstawowych aksjomatów i reguł wnioskowania, a następnie na wyciąganiu wniosków (twierdzeń) z aksjomatów bądź twierdzeń wykazanych już wcześniej.

3.1 Definicja: Przez *regułę wnioskowania* dla logiki $L = < F, I, Sat >$ rozumieć będziemy parę (Z,A), gdzie Z jest pewnym zbiorem formuł (podzbiorem F), zaś A jest formułą (regułę (Z,A) należy odczytywać "z prawdziwości wszystkich formuł ze zbioru Z można wywnioskować prawdziwość formuły A"). Zgodnie z tradycją regułę (Z,A) zapisywać będziemy $Z|\text{-}A$, Z nazywać jej *zbiorem przesłanek*, zaś A - *konkluzją*. Regułę o pustym zbiorze przesłanek nazywamy *aksjomatem*. *Systemem dowodzenia* (*wnioskowania*) dla logiki L nazywamy dowolny zbiór reguł wnioskowania.//

Jeśli zbiór twierdzeń uzyskiwanych przy pomocy systemu wnioskowania ma być nietrywialny (a zatem nieskończony), reguł wnioskowania musi też być nieskończenie wiele. Na szczęście zawsze podpadają one pod skończoną liczbę schematów. Przedstawiać je więc będziemy właśnie przy pomocy schematów. System o skończonej liczbie reguł, z których każda ma skończoną liczbę przesłanek nazwiemy *skończonym systemem dowodzenia*. Zauważmy ponadto, że definicja 3.1 obejmuje aksjomatyzacje w stylu zaproponowanym przez Hilberta. Innym, powszechnie używanym stylem jest rachunek sekwentów Gentzena. Ten ostatni jednakże nie pozwala na równie proste i naturalne rozszerzenie mechanizmów dowodzenia na teorie (niezbędne przy badaniu typów danych i przy weryfikacji programów liczących w oparciu o konkretne typy danych). Nie będziemy go więc tu bliżej omawiać, odsyłając zainteresowanych czytelników np. do [Tak'75].

3.2 Definicja: Dla danego zbioru formuł Z oraz systemu wnioskowania P, *dowód formuły A ze zbioru Z w systemie P* definiujemy jako drzewo D o gałęziach skończonej długości i wierzchołkach etykietowanych formułami, spełniające warunki:
(i) wszystkie liście D są etykietowane formułami ze zbioru Z lub aksjomatami logicznymi;
(ii) dla każdego wierzchołka n w drzewie D: albo n jest etykietowany przez formułę z Z, albo formułę B, dla której istnieje w P reguła postaci $S|\text{-}B$ oraz $S = \{C: C \text{ etykietuje syna wierzchołka n w } D\}$;
(iii) korzeniem D jest A.



Przykładowy schemat dowodu

Mówimy, że formuła A jest *twierdzeniem* uzyskanym ze zbioru formuł Z (*konsekwencją syntaktyczną* Z) w systemie wnioskowania P (ozn. $Z|\text{-}_P A$) wtw, gdy istnieje w P dowód A z Z.//

Punkty (i) oraz (ii) powyższej definicji można odczytać jako wspomniany wcześniej

warunek, że kolejne twierdzenia w procesie dowodzenia można uzyskać przy pomocy reguł wnioskowania albo bezpośrednio z aksjomatów, albo z twierdzeń wykazanych wcześniej. Punkt (iii) stwierdza, że końcowym twierdzeniem w procesie dowodzenia jest ta właśnie formuła, którą chcemy udowodnić.

3.3 Przykład: Poniższy zbiór (schematów) reguł wnioskowania (w tym aksjomatów A1-A4) jest systemem wnioskowania dla klasycznego rachunku zdań L_0 (por. przykład 2.2):
A1. $\vdash (p \vee p) \rightarrow p$; A2. $\vdash p \rightarrow (p \vee q)$; A3. $\vdash (p \vee q) \rightarrow (q \vee p)$; A4. $\vdash (p \rightarrow q) \rightarrow ((r \vee p) \rightarrow (r \vee q))$;
R1. $p, p \rightarrow q \vdash q$ (reguła odrywania);
R2. $p \vdash p(q/r)$, gdzie $p(q/r)$ oznacza formułę uzyskaną z p przez zastąpienie wszystkich wystąpień zmiennej zdaniowej q formułą r (*reguła podstawiania*).
Zauważmy, że p, q, r występujące w regułach wnioskowania (aksjomatach) A1-R2 pełnią rolę *metazmiennych*, przebiegających wszystkie formuły (wyjątkiem jest tu reguła R2, w której, jak to zaznaczyliśmy, q jest metazmienną przebiegającą jedynie zmienne zdaniowe).//

3.4 Przykład: Poniższy zbiór (schematów) reguł wnioskowania jest systemem wnioskowania dla klasycznej logiki pierwszego rzędu L_1 (por. przykład 2.3):
I1. aksjomaty A1-A4 oraz reguły R1-R2 z przykładu 3.3, z tą jednak różnicą, że metazmienne przebiegają teraz formuły logiki L_1;
I2. $\vdash (\forall x)A(x) \rightarrow A(x/t)$, gdzie t jest termem, zaś $A(x/t)$ oznacza formułę uzyskaną z A przez zastąpienie wszystkich wystąpień zmiennej indywiduowej x termem t;
I3. $A \rightarrow B \vdash A \rightarrow (\forall x)B$, gdy x nie jest zmienną wolną w A.//

3.5 Przykład: Udowodnijmy następujące twierdzenie L_1 (pomijając dla uproszczenia te części dowodu, które korzystają jedynie z tautologii L_0): $(\forall x)((A \vee B(x)) \rightarrow (A \vee (\forall x)(B(x)))$, gdy x nie jest zmienną wolną w A:
1. $(\forall x)(A \vee B(x)) \rightarrow (A \vee B(x))$ --- podstawienie (reguła 3.3.R2) w 3.4.I2;
2. $(\forall x)(A \vee B(x)) \rightarrow (\sim\sim A \vee B(x))$ --- podstawienie zastosowane do tautologii L_0;
3. $(\forall x)(A \vee B(x)) \rightarrow (\sim A \rightarrow B(x))$ --- podstawienie zastosowane do tautologii L_0;
4. $((\forall x)(A \vee B(x)) \wedge \sim A) \rightarrow B(x))$ --- podstawienie zastosowane do tautologii L_0;
5. $((\forall x)(A \vee B(x)) \wedge \sim A) \rightarrow (\forall x)B(x))$ --- reguła 3.4.I3;
6. $(\forall x)(A \vee B(x)) \rightarrow (\sim A \rightarrow (\forall x)B(x))$ --- podstawienie zastosowane do tautologii L_0;
7. $(\forall x)(A \vee B(x)) \rightarrow (A \vee (\forall x)B(x))$ --- podstawienie zastosowane do tautologii L_0.
Powyższy dowód można w oczywisty sposób zobrazować jako drzewo wymagane w definicji 3.2.//

Zauważmy, że systemy wnioskowania rozważane powyżej umożliwiają dowodzenie nie tylko tautologi, ale także twierdzeń będących konsekwencjami zbiorów formuł (własności). Jest to niezwykle ważna ich cecha, bowiem najczęściej interesują nas własności pewnej podklasy klasy interpretacji dopuszczalnych. "Wycinanie" tej podklasy uzyskujemy poprzez specyfikację interesujących nas własności przy pomocy zbioru formuł logiki, przyjmując że interesują nas twierdzenia, które są prawdziwe we wszystkich modelach wybranego zbioru formuł.Formuły tego zbioru przyjmujemy za nowe aksjomaty w systemie dowodzenia. Teraz do dowodzenia twierdzeń stosujemy system wnioskowania rozszerzony o te aksjomaty.

Dotychczas nie nakładaliśmy żadnych ograniczeń na systemy wnioskowania. Jasne jest jednak, że nie każdy taki system jest zadawalający. Na definiującego systemy wnioskowania czyhają (co najmniej) dwie pułapki. Przede wszystkim może się okazać, że posługując się systemem wnioskowania można dowodzić faktów nieprawdziwych (mówimy wówczas o niepoprawności systemu). Z drugiej strony można rozpatrzeć system o pustej liczbie reguł wnioskowania, w którym można nic wykazać. Należy więc zadbać o to, aby definiowany system

wnioskowania zapewniał (przynajmniej teoretyczną) możliwość wykazania prawdziwości rozsądnego zbioru własności (mówimy wówczas o pełności systemu). Ponieważ badania nad logikami programów doprowadziły do zdefiniowania różnych pojęć poprawności i pełności, pojęcia te omówimy bliżej później, przy okazji prezentowania różnych logik. Teraz możemy przytoczyć jedynie klasyczną definicję pełności i poprawności:

3.6 Definicja: Powiemy, że system wnioskowania P dla logiki L jest *poprawny* (*pełny*) wttw, gdy dla każdego zbioru formuł Z logiki L zbiór konsekwencji semantycznych Z, $Cn_L(Z)$, zawiera zbiór twierdzeń $C_P(Z)$, uzyskanych ze zbioru Z ($Cn_L(Z)$ jest zawarty w $C_P(Z)$)//

Ideałem klasycznym jest równość $C_P(Z) = Cn_L(Z)$, intuicyjnie oznaczająca, że wszystko co da się udowodnić jest prawdziwe oraz wszystko co jest prawdziwe da się udowodnić.

3.7 Przykład: Systemy wnioskowania dla L_0 i L_1 przedstawione w przykładach 3.3 i 3.4 są poprawne i pełne w klasycznym sensie (zdefiniowanym powyżej). Pełność systemu wnioskowania dla L_1 sugeruje następujący częściowy algorytm sprawdzania czy dana formuła jest tautologią:
- generuj wszystkie dowody (np. coraz dłuższe) i sprawdzaj czy dana formuła jest wnioskiem.
Ze względu na pełność, każda formuła prawdziwa ma (skończony) dowód, przy generowaniu wszystkich dowodów znajdziemy więc i jej dowód. Jeśli formuła nie jest prawdziwa, algorytm będzie oczywiście działał w nieskończoność. Opisana tu metoda, choć prosta koncepcyjnie, jest bardzo nieefektywna. Znacznie bardziej efektywne metody oparte są o formalizm Gentzena lub zasadę rezolucji.//

4. Typy danych

Formalna specyfikacja i analiza typów danych zajmuje ważną pozycję we współczesnych badaniach związanych z semantyką języków programowania i procesem weryfikacji programów. Zastanówmy się przez chwilę czym są typy danych. Przede wszystkim zauważmy, że typu danych nie określa jedynie zbiór wartości, jakie mogą być przyjmowane przez zmienne tego typu. Na przykład inne własności ma zbiór liczb naturalnych rozpatrywany jedynie z działaniem dodawania, a inne ten sam zbiór z działaniem mnożenia (różnice są tu o wiele głębsze niż się to może wydawać na pierwszy rzut oka - np. zbiór formuł (z dodawaniem) prawdziwych w zbiorze liczb naturalnych jest rozstrzygalny podczas gdy mnożenie powoduje iż staje się on nierozstrzygalny nawet częściowo). Podobnie, para liczb rzeczywistych może określać liczbę zespoloną (jeśli odpowiednio określimy działania dodawania i mnożenia), natężenie i napięcie prądu w przewodniku (jeśli określimy np. operację obliczania oporu przewodnika). Typ *stosy* różni od typu *kolejki* jedynie definicja operacji wstawiania elementu (na początek stosu, na koniec kolejki), podczas gdy same wartości tych typów (skończone ciągi elementów) nie różnią się od siebie. Tak więc dane różnych typów są rozróżnialne nie tylko przez ich wartości, lecz także przez operacje z nim związane. Odpowiednim formalizmem do modelowania (nie specyfikowania!) typów danych będą więc systemy relacyjne mające właśnie ich (typów) strukturę.

4.1 Przykład: Arytmetykę liczb naturalnych definiujemy jako system relacyjny $\underline{N} = <N;0,1,+,\cdot,^*,<,=>$, gdzie każdy elment N jest albo zerem, albo powstaje w wyniku

dodania do zera skończonej (!) liczby jedynek, zaś $+,-,*,<,=$ mają standardowe znaczenie.//

4.2 Przykład: Załóżmy, że mamy dany pewien typ E, o którym zakładamy jedynie, iż zdefiniowano w nim wyróżnioną stałą (funkcję zeroargumentową) ERROR oraz relację $=_E$ porównywania wartości (typu E). Stosy elementów typu E definiujemy jako system relacyjny (dwusortowy) $<E,S;ERROR,top,push,pop,empty,=_E,=_S>$, gdzie:
(i) sort E oraz stała ERROR i relacja $=_E$ są omówione wyżej;
(ii) S jest zbiorem skończonych (!) ciągów wartości typu E;
(iii) top(()) = ERROR; top($(e_1,...,e_n)$) = e1; ($(e_1,...,e_n)$ oznacza ciąg elementów $e_1,...,e_n$, zaś () - ciąg pusty);
(iv) push(e, $(e_1,...,e_n)$) = $(e,e_1,...,e_n)$;
(v) pop(()) = (); pop($(e_1,e_2,...,e_n)$) = $(e_2,...,e_n)$;
(vi) empty = ();
(vii) $(e_1,...,e_n) = (e_1',...,e_m')$ wtw, gdy n = m oraz $e_1 = e_1',...,e_m = e_m'$.//

4.3 Przykład: Typ drzew binarnych (każdy wierzchołek ma co najwyżej dwóch synów) definiujemy jako system relacyjny $<D;left,right,cons,empty>$, gdzie:
(i) D jest zbiorem wszystkich skończonych (!) wyrażeń (termów) zawierającym empty (drzewo puste) i spełniającym warunek, że jeśli do D należą d_1 i d_2, to należy także wyrażenie cons(d_1,d_2) (drzewo, którego lewym poddrzewem jest d_1, a prawym - d_2);
(ii) left(empty) = right(empty) = empty; left(cons(d_1,d_2)) = d_1; right(cons(d_1,d_2)) = d_2;
(iii) cons(d_1,d_2) = cons(d_1,d_2); empty = empty.
Dla odróżnienia elementów zbioru D od funkcji, nazwy tych pierwszych podkreśliliśmy.//

Podobne semantyczne definicje typów danych są wysoce niewygodne w procesie dowodzenia własności programów, nie są bowiem opisane w kategoriach pozostających na wystarczającym poziomie abstrakcji i przystających do metod wnioskowania omawianych w trzecim rozdziale. Skrajnym przykładem definicji tego typu byłoby wyznaczenie semantyki języka programowania poprzez działanie kompilatora. Dążymy zatem zawsze do zdefiniowania semantyki typu danych poprzez podanie najbardziej podstawowych własności danych i operacji w nim występujących. W zależności od siły przyjętej logiki i podanych w niej własności możemy uzyskać lepszy lub gorszy opis interesującego nas typu danych.

Logiczna specyfikacja typu danych składać się będzie zatem tylko z najważniejszych informacji o typie, bez wnikania w sposób jego implementacji. Do informacji tych należy po pierwsze sygnatura operacji (inaczej ich składnia, określająca dziedziny i przeciwdziedziny) w definicjach semantycznych występująca pośrednio, a po drugie zestaw aksjomatów precyzujących własności danych i operacji. Abstrahując od konkretnej implementacji (innymi słowy - od konkretnej semantycznej definicji typu danych) musimy znów zwrócić uwagę na problemy analogiczne do poprawności i pełności systemów wnioskowania. Odpowiednikiem poprawności jest w przypadku typu danych zapewnienie, iż specyfikacja ma jakąkolwiek implementację (istnieje dla niej model).Mówimy wówczas o niesprzeczności specyfikacji. Odpowiednikiem pełności jest zapewnienie, że specyfikacja umożliwi dowodzenie rozsądnego zbioru własności, czyli zawiera w sobie odpowiedź na rozsądny zbiór pytań jakie może postawić programista implementujący typ danych (mówimy wówczas o zupełności).

4.4 Definicja: Powiemy, że specyfikacja typu danych, której aksjomaty wyrażone zostały zbiorem Z formuł logiki $L = <F,I,Sat>$, jest *niesprzeczna* wtw, gdy Z ma model (istnieje interpretacja I taka, że I *Sat* Z). Powiemy, że specyfikacja ta jest *zupełna*, jeśli każda formula

bez zmiennych wolnych, bądź jej zaprzeczenie jest konsekwencją (semantyczną) zbioru Z.//

Zauważmy, iż specyfikacje budujemy, mając w głowie pewien model i dbając o to, aby wyrażane własności ten właśnie model opisywały. Przy poprawnym (zgodnym z zamierzeniami) wyrażaniu własności niesprzeczność uzyskujemy zatem praktycznie "za darmo" (za błędy tu popełnione trzeba jednak słono płacić!). Uzyskanie zupełnych specyfikacji jest problemem znacznie trudniejszym, często niemożliwym do rozwiązania. W literaturze rozważane są także inne, słabsze (a więc łatwiejsze do uzyskania) pojęcia zupełności, np. tzw. *wystarczająca zupełność* rozważana w kontekście badań informatycznych przez Guttaga (por. np. [Tur'82]), a będąca adaptacją pojęcia słabej reprezentowalności, znanego z logiki klasycznej (por. np. [Grz'75]). Zakłada się tu, że wystarczającą dla programisty wiedzą jest możliwość wykazania dla każdej operacji f typu danych wszystkich prawdziwych twierdzeń postaci $f(c_1,...,c_n) = d$, gdzie $c_1,...,c_n,d$ są termami stałymi (nie zawierającymi zmiennych). Podkreślmy w tym miejscu, iż kryterium wystarczającej zupełności stanowi program minimum (wynikający ze słabości klasycznej logiki L_1 - o czym za chwilę), bowiem w procesie dowodzenia własności programów często niezbędna jest istotnie szersza wiedza niż oferowana przez specyfikacje "wystarczająco zupełne".

4.5 Przykład: Sygnatura typu danych liczb naturalnych (4.1) jest następująca:
0, 1: -> N; +,*: NxN-> N; <, =: NxN-> {true,false}.
Jako aksjomaty specyfikacji tego typu przyjąć można np. zestaw aksjomatów Peano, wyrażonych w języku logiki pierwszego rzędu:
(i) $(\forall x,y)(x=x \wedge (x=y \rightarrow y=x) \wedge ((x=y \wedge y=z) \rightarrow x=z))$; (= jest relacją równoważności)
(ii) $(\forall x)((\neg 0=x+1) \wedge x+0=x \wedge x*1=x)$;
(iii) $(\forall x,y)((x+1=y+1 \rightarrow x=y) \wedge x+(y+1)=(x+y)+1 \wedge x*(y+1)=x*y+x)$;
(iv) $(A(0) \wedge (\forall x)(A(x) \rightarrow A(x+1))) \rightarrow (\forall x)A(x)$; (schemat aksjomatu indukcji);
(v) $(\forall x,y)((x>y \rightarrow (x-y=z \leftrightarrow x=z+y)) \wedge (\neg x>y \rightarrow x-y=0))$;
(vi) $(\forall x,y)(x>y \leftrightarrow (\exists z)x+z+1=y)$.
Powyższy zbiór aksjomatów (schematy aksjomatów traktować będziemy, jak zwykle, jako nieskończone zbiory aksjomatów powstałe przez podstawienie konkretnych formuł w miejsce metazmiennych reprezentujących formuły) nie jest zupełny (ale jest wystarczająco zupełny) - por. np. [Grz'75]. Można wykazać, że w logice klasycznej nie można podać (częściowo) rozstrzygalnego zbioru aksjomatów, który zapewniłby zupełność specyfikacji liczb naturalnych (por. np. [Grz'75]). Jeśli więc mamy specyfikację zupełną, nie potrafimy automatycznie rozstrzygnąć czy dana formuła jest aksjomatem specyfikacji.//

Powyższy przykład wskazuje na pewne poważne ograniczenie klasycznej logiki pierwszego rzędu w możliwości specyfikowania typów danych. Można pokazać także następujące fakty dotyczące tej logiki:
- każda specyfikacja ma modele niestandardowe (różne od zamierzonych) - por. tw. Skolema, Lowenheima, Tarskiego np. w [Grz'75];
- nie można wyrazić ani skończoności elementów typu danych (liczb naturalnych, stosów, drzew itd.) - por. np. [MS'87], ani skończoności jego dziedziny - por. tw. Trakhtenbrota, np. w [EFT'84];
- istnieją typy danych o tej własności, że można wykazać wszystkie twierdzenia postaci A(c), gdzie c jest dowolnym elementem dziedziny (stałą), zaś nie da się wykazać twierdzenia ogólnego $(\forall x)(A(x))$ - por. np. [Grz'75].
Jak pokażemy dalej, ograniczenia te nie dotyczą logik silniejszych niż L_1.

5. Dwa modele programów

Zastanówmy się teraz nad takim sposobem modelowania programów, który najlepiej przystawałby do rozważania ich własności. Musimy zatem zacząć przede wszystkim od identyfikacji interesujących nas własności programów. Zadanie to, wbrew pozorom, sprawić może wiele trudności. Na pierwszy rzut oka wydaje się, że podstawowe pytania o poprawność programu dotyczą zależności między jego danymi i wynikami - wejściem/wyjściem, (w tej klasie zagadnień zawiera się np. częściowa i całkowita poprawność). Takie podejście, jakkolwiek najbardziej naturalne w przypadku typowych programów sekwencyjnych, czy niedeterministycznych, z pewnością nie usatysfakcjonuje twórców takich programów, które z założenia nie mają się zatrzymać. Klasa tego typu programów jest bardzo szeroka - należą do niej systemy operacyjne, systemy rezerwacji miejsc lotniczych, kontrola przebiegu procesów przemysłowych itd. itd. W ich przypadku istotne jest pytanie o to jak zachowuje się program w trakcie obliczeń, a nie w momencie ich zakończenia. Zauważmy jeszcze, że podczas dowodzenia własności związanych z wejściem/wyjściem programu zależy nam jeśli już nie na całkowitym abstrahowaniu od przebiegu obliczenia, to przynajmniej na możliwie najmniejszym weń wnikaniu.

Wyodrębnić zatem możemy dwie podstawowe grupy własności - pierwsza dotyczy relacji wejścia/wyjścia programów, zaś druga - ich historii obliczeniowych. Tworzone są zatem także dwa podstawowe modele programów - model *relacyjny* i *interakcyjny*. Nazwa pierwszego z nich uzasadniona jest dążeniem do rozpatrywania programów jedynie z punktu widzenia ich relacji wejścia/wyjścia. Nazwa "model interakcyjny" wynika z kolei z faktu, iż własności wspomnianych systemów dotyczą przede wszystkim ich interakcji ze środowiskiem, np. w przypadku systemu operacyjnego istotne są przede wszystkim pytania w jaki sposób obsługiwane są żądania kierowane przez użytkownika, bądź żądania kierowane z jednej warstwy systemu do innych jego warstw.

5.1 Przykład: (model relacyjny) rozpatrzmy język programowania o składni zdefiniowanej przy pomocy następującej gramatyki (o symbolach nieterminalnych PROGRAM oraz FORMUŁA: PROGRAM:: = x: = t | FORMUŁA? | PROGRAM;PROGRAM | PROGRAM + PROGRAM | PROGRAM*, gdzie x jest zmienną, t - termem, zaś symbol nieterminalny FORMUŁA reprezentuje formuły otwarte (bez kwantyfikatorów) klasycznej logiki pierwszego rzędu (patrz przykład 2.3).
Z intuicyjnego punktu widzenia : = odpowiada podstawieniu, ? oznacza test (jeśli testowana formuła jest spełniona to obliczenie przechodzi dalej, w przeciwnym przypadku blokuje się), spójniki ; oraz + oznaczają odpowiednio złożenie sekwencyjne i niedeterministyczny wybór, zaś * - niedeterministyczną iterację (wybierz dowolną liczbę naturalną n i wykonaj program n-krotnie). W modelu relacyjnym semantykę programów definiujemy poprzez relację między wartościowaniami zmiennych na ich wejściu i wyjściu - rozpatrywane tu relacje będą więc zbiorami par wartościowań. Oczywiście obliczenia programu zależne są od przyjętego typu danych (por. 4). Dla ustalonego typu danych D z każdym programem P wiążemy jego relację wejścia/wyjścia, oznaczaną przez $W_D(P)$, następująco:
- $W_D(x:=t) = \{(v,v') \mid v'$ różni się od v co najwyżej wartością zmiennej x i $v'(x) = t_R(v)$ (wartość termu t, $t_R(v)$, definiowaliśmy w 2.3)};
- $W_D(G?) = \{(v,v) \mid G_R(v) = true$ (wartość formuły G, $G_R(v)$, definiowaliśmy w 2.3)};
- $W_D(P;Q) = W_D(P) \circ W_D(Q)$ (złożenie relacji, tzn. zbiór par (v,v"), dla których istnieje v' takie, że (v,v') jest elementem relacji danej jako pierwszy parametr złożenia, zaś (v',v") - jako drugi);
- $W_D(P+Q) = W_D(P) \cup W_D(Q)$ (zwykła suma relacji traktowanych jako zbiory par);

- $W_D(P^*) = (W_D(P))^*$ (zwrotne i przechodnie domknięcie relacji $W_D(P)$, tzn. (najmniejszy) zbiór będący rozszerzeniem $W_D(P)$ o pary postaci (v,v) i spełniający warunek, że jeśli (v,v') oraz (v',v") doń należą, to należy również (v,v")).

Powyższy zestaw programów, zaczerpnięty z logiki dynamicznej (por. np. [Har'78]), wystarcza do zdefiniowania innych typowych konstrukcji programistycznych, np.:

- instrukcja wyboru **if** G **then** P **else** Q **fi** definiowana jest jako [(G?;P) + (~G?;Q)];
- instrukcja iteracji **while** G **do** P **od** - jako [(G?;P)*;(~G?)];
- instrukcja dozorowana G1-> P1[]G2-> P2[]...[]Gk-> Pk - jako
  [(G1?P1) + (G2?;P2) + ... + (Gk?;Pk)].//

5.2 **Przykład:** (model interakcyjny) rozpatrzmy język programowania, będący rozszerzeniem języka rozpatrywanego w 5.1 o operator równoległości ||. W modelu interakcyjnym semantykę programów definiować będziemy poprzez zbiór ich możliwych obliczeń (skończonych lub nieskończonych ciągów wartościowań ich zmiennych). Podobnie jak poprzednio (5.1), obliczenia programu zależne są od przyjętego typu danych (por. 4). Dla ustalonego typu danych D z każdym programem P wiążemy zbiór ciągów, oznaczany przez $C_D(P)$, następująco:

- $C_D(x:=t) = W_D(x:=t)$ (W_D definiowaliśmy w 5.1);
- $C_D(G?) = W_D(G?)$;
- $C_D(P;Q) = C_D(P) \circ C_D(Q)$ (zbiór ciągów $(v_1,...,v_n,v_{n+1},...)$, takich że albo $(v_1,...,v_n,v_{n+1},...)$ jest obliczeniem P (elementem $C_D(P)$), gdy jest ono nieskończone, albo $(v_1,...,v_n)$ jest obliczeniem P, zaś $(v_n,v_{n+1},...)$ jest obliczeniem Q);
- $C_D(P+Q) = C_D(P) \cup C_D(Q)$ (zwykła suma zbiorów);
- $C_D(P^*) = \bigcup_{i \in \omega} C_D(P^i)$ (suma wszystkich zbiorów $C_D(P^i)$, gdzie P^i oznacza i-krotne złożenie (sekwencyjne) programu P, przy czym $C_D(P^0) = \{(v) \mid v$ jest wartościowaniem zmiennych$\}$;
- $C_D(P||Q) = C_D(P)||C_D(Q)$ (zbiór przeplotów ciągów z $C_D(P)$ oraz $C_D(Q)$, przy czym przeplotem ciągów q' oraz q" nazywamy ciąg q zawierający wszystkie (i tylko) elementy q' i q" oraz zachowujący porządek elementów wyznaczony przez q' i q", tzn. istnieją ciągi liczb naturalnych $i_1 < i_2 < ...$ oraz $j_1 < j_2 < ...$ takie, że $\{i_1,i_2,...,j_1,j_2,...\}$ jest zbiorem wszystkich liczb naturalnych, zaś $q' = (q_{i1},q_{i2},...)$ i $q'' = (q_{j1},q_{j2},...)$).//

6. Logika Hoare'a dla częściowej poprawności programów sekwencyjnych

Poniższy rozdział w całości dotyczyć będzie języka while-programów (zawierającego podstawienie (:=), złożenie sekwencyjne (;), instrukcję **if then else fi** oraz **while do od**). Rozważać będziemy więc programy sekwencyjne, deterministyczne (ich semantyka zdefiniowana została w przykładzie 5.1). Jak już wcześniej wspomnieliśmy, najbardziej adekwatnym modelem do rozważania własności programów sekwencyjnych jest model relacyjny. Interesujące nas własności programów dotyczą zatem ich relacji wejścia/wyjścia. Podstawowe pytania związane są zatem z dwoma zagadnieniami: po pierwsze - czy dla danych wejściowych program się zatrzymuje, a po drugie - czy (jeśli program się zatrzymuje) jego wyniki odpowiadają naszym oczekiwaniom. Problem stopu omówimy dalej, teraz zajmijmy się drugą z własności (zwaną częściową poprawnością).

Ustalmy pewną strukturę danych D. Zauważmy najpierw, że formuły logiki klasycznej L_1 (por. 2.3), interpretowane w D, wyznaczają zbiór tych wartościowań, przy których wartością tych formuł jest **true** (tzn. wartościowań spełniających te formuły). Zbiory wartościowań możemy

zatem opisywać formułami logiki L_I. To proste spostrzeżenie daje podstawę do rozważania własności programów przy pomocy par formuł tej logiki. Pierwsza z tych formuł opisuje dane wejściowe, zaś druga - wyjściowe.

6.1 Definicja: Powiemy, że program P jest *częściowo poprawny względem formuł A i B*, oznaczając ten fakt przez $\{A\}P\{B\}$, wttw gdy dla każdych danych wejściowych spełniających warunek A, wyniki (o ile istnieją) spełniają warunek B. Bardziej formalnie, *logiką Hoare'a dla częściowej poprawności while-programów* nazwiemy trójkę $L_H = <F_H, I_H, Sat_H>$, gdzie:
- F_H jest zbiorem formuł postaci $\{A\}P\{B\}$, gdzie A, B są formułami logiki L_I (por. 2.3) zaś P jest while-programem;
- I_H jest klasą systemów relacyjnych (por. 2.3);
- $R \; Sat_H \{A\}P\{B\}$ wttw, gdy dla dowolnych wartościowań zmiennych v: $(\{A\}P\{B\})_R(v) = true$, gdzie wartość formuły $\{A\}P\{B\}$ w systemie relacyjnym R i przy wartościowaniu zmiennych v, $(\{A\}P\{B\})_R(v)$, definiujemy następująco:
$(\{A\}P\{B\})_R(v) = true$ wttw, gdy $A_R(v) = true$ (por. 2.3) implikuje $B_R(v') = true$ dla dowolnego wartościowania v' takiego, że (v,v') należy do zbioru $W_D(P)$ (zdefiniowanego w 5.1).//

Zauważmy, iż ze względu na deterministyczny charakter rozważanych programów, każdy zbiór $W_D(P)$ jest albo pusty (gdy P się pętli), albo jednoelementowy (w przeciwnym przypadku).

6.2 Przykład: Poniższe warunki częściowej poprawności są prawdziwe (prawdziwość jest tu oczywiście wyznaczona przez Sat_H) w arytmetyce liczb naturalnych (por. 4.1):
$\{x = 0\}x := x + 1\{x = 1\}$, $\{x = x_0 \wedge y = y_0\}x := x + y; y := x + y\{x = x_0 + y_0 \wedge y = x_0 + y_0 + y_0\}$,
$\{x = x_0\}z := 0;$**while** $\sim z = y$ **do** $x := x + 1; z := z + 1$ **od**$\{x = x_0 + y\}$,
natomiast następujące nie są:
$\{x = 0\}x := x + 1\{x = 0\}$, $\{$**true**$\}x := x + 1\{x = 1\}$;
$\{$**true**$\}z := 0;$**while not** $z = y$ **do** $x := x + 1; z := z + 1$ **od**$\{x = x_0 + y\}$.//

Najważniejszym osiągnięciem Hoare'a było podanie systemu wnioskowania, umożliwiającego dowodzenie częściowej poprawności while-programów. Zestaw reguł tego systemu jest tak skonstruowany, że w każdej chwili dokładnie wiadomo, której reguły należy użyć w trakcie przeprowadzania dowodu - sposób postępowania wyznaczony jest przez składnię programu. W poniższej definicji przedstawiamy zestaw reguł podany przez Hoare'a [Hoa'69] i uważany zdecydowanie za najlepszy (i najbliższy praktyce programowania) aparat dowodzenia częściowej poprawności programów. Jest on również, mimo istotnych ograniczeń, doskonałym narzędziem specyfikacji.

6.3 Definicja: Przez system wnioskowania S_H dla logiki Hoare'a rozumieć będziemy następujący zbiór reguł wnioskowania:
(i) $\{A\}P\{C\}$, $C \rightarrow B \; |- \{A\}P\{B\}$; $A \rightarrow C$, $\{C\}P\{B\} \; |- \{A\}P\{B\}$;
(ii) $|- \{A(x/t)\}x := t\{A(x)\}$;
(iii) $\{A\}P\{C\}$, $\{C\}Q\{B\} \; |- \{A\}P;Q\{B\}$;
(iv) $\{A \wedge G\}P\{B\}$, $\{A \wedge \sim G\}Q\{B\} \; |- \{A\}$**if** G **then** P **else** Q **fi**$\{B\}$;
(v) $\{A \wedge G\}P\{A\} \; |- \{A\}$**while** G **do** P **od**$\{A \wedge \sim G\}$.
Formuła A występująca w regule (v) nazywana jest *niezmiennikiem* pętli **while**.//

6.4 Przykład: Ustalmy arytmetykę liczb naturalnych (por. 4.1), jako strukturę danych i udowodnijmy następujący prosty fakt:

$$\{x = y*(1+q)+r\text{-}y\}r := r\text{-}y; q := 1+q\{x = y*q+r\}.$$

Zgodnie ze składnią rozważanego programu, skorzystać musimy z reguły 6.3(iii). Aby było to możliwe, trzeba znaleźć "formułę pośrednią" C. Nietrudno zauważyć, że może nią być formuła $x=y*(1+q)+r$, mamy bowiem, zgodnie z aksjomatem na przypisanie 6.3(ii), że:
$\vdash\{x=y*(1+q)+r-y\}r:=r-y\{x=y*(1+q)+r\}$ oraz $\vdash\{x=y*(1+q)+r\}q:=1+q\{x=y*q+r\}$.
Teraz, zgodnie z regułą 6.3(iii), używając wyżej wykazanych faktów jako jej przesłanek, uzyskujemy dowodzony fakt.
Wykażmy teraz, częściową oprawność algorytmu znajdującego iloraz (zmienna q) i resztę z dzielenia (zmienna r) liczby x przez y:

$\{x\geq 0\wedge y>0\}$ $q:=0;\ \{x\geq 0\wedge y>0\wedge x=q*y+x\}$ (*)
$r:=x;\ \{x=q*y+r\wedge y>0\wedge 0\leq r\}$ (**)
while $r\geq y$ **do** $\{x=q*y+r\wedge 0<y\leq r\}$ $r:=r-y;\ q:=1+q$ **od**; (***)
$\{x=q*y+r\wedge 0\leq r<y\}$.

Dla uproszczenia wpisaliśmy formuły pośrednie, niezbędne przy dowodzie bezpośrednio w tekst programu, w postaci komentarzy. Aby wykazać (*) wystarczy zauważyć, że ze względu na własności struktury danych (liczb naturalnych) twierdzeniem jest formuła $x=0*y+x$, zatem $(x\geq 0\wedge y>0)\Rightarrow(x\geq 0\wedge y>0\wedge x=0*y+x)$ i skorzystać z drugiej z reguł 6.3(i) i aksjomatu 6.3(ii). Dowód (**) uzyskujemy przez natychmiastowe zastosowanie 6.3(ii). Formuła zapisana wewnątrz pętli (***) jest koniunkcją formuły (**) oraz warunku pętli. Pokazaliśmy wcześniej, że $\vdash\{x=y*(1+q)+r-y\}r:=r-y;q:=1+q\{x=q*y+r\}$. Po wymnożeniu nawiasów i redukcji (tu ingerują własności struktury danych) uzyskujemy:
$\vdash\{x=q*y+r\}r:=r-y;q:=1+q\{x=q*y+r\}$.
Łatwo też wykazać, iż $\{0<y\leq r\}r:=r-y;q:=1+q\{0<y\wedge 0\leq r\}$. Przyjmując zatem w regule 6.3(v) za G formułę $r\geq y$, zaś za A - formułę $x=q*y+r\wedge y>0\wedge 0\leq r$, dowodzone twierdzenie uzyskujemy jako bezpośredni wniosek.//

Można wykazać, że system wnioskowania S_H podany w 6.3 jest poprawny (por. 3.6) oraz że nie jest pełny. Co więcej, nie istnieje żaden pełny i skończony system dowodzenia dla logiki Hoare'a. Badania prowadzone nad systemem S_H doprowadziły do zdefiniowania nowego pojęcia pełności, a mianowicie tzw. relatywnej pełności (por. [Coo'78]). Podstawą tego pojęcia jest rozróżnienie pomiędzy wnioskowaniem o własnościach programów, a wnioskowaniem o własnościach struktur danych wykorzystywanych w trakcie obliczeń. Przyjmuje się przy tym, że program przeprowadza obliczenia w pewnej konkretnej strukturze danych, której własności wyrażane są formułami klasycznej logiki pierwszego rzędu L_1. Okazuje się, że istnieją programy i struktury o tej własności, że wprawdzie program (zawierający pętle) jest częściowo poprawny, ale nie istnieją niezmienniki niezbędne do udowodnienia tej poprawności. Rozważać jednak można zawężoną, lecz wystarczająco szeroką, klasę struktur danych (zwanych *strukturami* (albo *interpretacjami*) *eksprcsywnymi*), dla których można udowodnić istnienie niezmienników i wszystkch formuł pośrednich niezbędnych w dowodzie. Nie wnikając w szczegóły techniczne wspomnijmy tylko, że klasa ta składa się ze struktur skończonych (tj. mających skończoną dziedzinę) bądź arytmetycznych (tj. takich, które zawierają w sobie arytmetykę liczb naturalnych - por. 4.1). Te ostatnie zdefiniujemy bliżej omawiając pojęcie arytmetycznej pełności w kontekście logiki dynamicznej.

6.5 Definicja: Powiemy, że system dowodzenia P dla logiki $L=<F,I,Sat>$ jest *relatywnie pełny (poprawny)* względem klasy interpretacji $J\subset I$ wttw, gdy dla każdej interpretacji $J\in J$:

$J\ Sat\ A$ implikuje (jest implikowane przez) $Th_J\vdash_P A$,

gdzie Th_J jest zbiorem wszystkich formuł logiki L_1 prawdziwych w J.//

Zauważmy, iż relatywnie pełne systemy wnioskowania sprowadzają wnioskowanie o

programach do wnioskowania o własnościach struktur danych. Zakładając że, te ostatnie znane są programiście (trudno sobie wyobrazić, aby korzystał on ze struktur danych nie mając wystarczającej o nich wiedzy), uzyskujemy silny aparat dowodzenia własności w rozważanej klasie struktur. Można udowodnić następujące twierdzenie wskazujące, iż system Hoare'a dysponuje podobną siłą:

Twierdzenie 6.6: System S_H (por. 6.3) dla logiki Hoare'a (por. 6.1) jest relatywnie pełny względem klasy interpretacji ekspresywnych.//

Logika Hoare'a służy do dowodzenia częściowej poprawności, nie da się w niej bezpośrednio wyrazić własności stopu. Nie oznacza to jednak, iż nie da jej się użyć do dowodzenia tej własności. Istnieją bowiem kryteria, umożliwiające w dużej ilości przypadków dowodzenie, że dany program się zatrzymuje. Kryteria te wyrażane są jednak na meta-poziomie, a nie za pomocą formuł logiki, a zatem nie mogą być użyte jako narzędzie specyfikacji, a dowodzenie przy ich użyciu ma pośredni charakter. Ponadto nie są one ogólne tzn. istnieją programy, których własności stopu nie da się wykazać przy ich pomocy. Dlatego też nie będziemy ich tu bliżej omawiać, odsyłając zainteresowanych do bogatej literatury poruszającej ten temat (np. [AA'82, BK'82]). Problematykę związaną z zatrzymywaniem się programów omówimy dalej, przy okazji prezentacji innych logik.

7. Logika algorytmiczna - programy sekwencyjne

Rozdział ten, podobnie jak poprzedni, dotyczyć będzie języka while-programów. Jak już wcześniej wspomnieliśmy, najbardziej adekwatnym modelem do rozważania własności takich programów jest model relacyjny. Interesujące nas własności programów dotyczą zatem ich relacji wejścia/wyjścia. W logice algorytmicznej, wprowadzonej w 1970 roku a następnie intensywnie badanej i rozwijanej (por. [MS'87), odmiennie niż w logice Hoare'a, nie rozważa się wyspecjalizowanych formuł obejmujących wybraną klasę własności. W ich miejsce wprowadza się bardziej ogólne (a zarazem bardziej podstawowe) pojęcia, umożliwiające wyrażanie rozmaitych własności, a w ich liczbie częściowej poprawności oraz własności stopu.

Podstawowe formuły służące do wyrażania własności programów w logice algorytmicznej przyjmują postać PA, gdzie P jest programem, zaś A - formułą. Intuicyjnie formuła taka wyraża fakt iż dla dowolnych danych program P się zatrzymuje, a jego wyniki spełniają formułę A. Zauważmy w tym miejscu, że podobne znaczenie ma formuła *nww*(P,A) (*nww* jest skrótem od "najsłabszy warunek wstępny") rozważana przez Dijkstrę w kontekście programów niedeterministycznych. Dla naszych celów ciekawsze będzie jednak omówienie logiki algorytmicznej, historycznie wcześniejszej, a ponadto mającej dobrze rozwinięte podejście do specyfikacji typów danych oraz znacznie bardziej zaawansowane badania teoriodowodowe. Zainteresowanych pracami Dijkstry odsyłamy do książki [Dij'85].

7.1 Definicja: *Logiką algorytmiczną* nazwiemy trójkę $L_A = < F_A, I_A, Sat_A >$, gdzie:
- F_A definiujemy rozszerzając definicję formuł logiki L_I (por. 2.3) o regułę stwierdzającą, iż PA jest formułą L_A, jeśli A jest formułą logiki L_A, zaś P jest while-programem;
- I_A jest klasą systemów relacyjnych (por. 2.3);
- Sat_A jest rozszerzeniem Sat_I (por. 2.3) o regułę stwierdzającą, iż R Sat_A PA wtw, gdy dla

dowolnego wartościowaia zmiennych v: $(PA)_R(v) = true$, gdzie wartość formuły PA w systemie relacyjnym R i przy wartościowaniu zmiennych v, $(PA)_R(v)$, definiujemy indukcyjnie: $(PA)_R(v) = true$ wttw, gdy istnieje wartościowanie v' takie, że (v,v') należy do zbioru $W_D(P)$ (zdefiniowanego w 5.1) i $A_R(v') = true$.//

Zauważmy, że w logice algorytmicznej można wyrazić wiele ważnych własności while-programów, np.:
- własność stopu programu P: Ptrue;
- częściową poprawność programu P względem warunków A i B: $(A \wedge Ptrue) \rightarrow PB$;
- całkowitą poprawność programu P względem warunków A i B (oznaczającą, że dla danych spełniających A program P się zatrzymuje, a jego wyniki spełniają B): $A \rightarrow PB$.

Inną istotną cechą logiki algorytmicznej jest możliwość naturalnego wyrażania własności typów danych. Odrzuca się tu ogranicznie, iż specyfikacji dokonuje się przy pomocy formuł logiki klasycznej L_I, dopuszczając również inne formuły logiki algorytmicznej. Możliwość stwierdzenia, iż w specyfikowanym typie danych pewne programy zachowują się w ten czy inny sposób okazuje się bardzo ważna i istotnie zwiększa siłę specyfikacji.

7.2 Przykład: Pod koniec czwartego rozdziału wspomnieliśmy, że skończoność elementów typu danych (np. liczb natralnych i stosów - por. 4.1 i 4.2) nie da się wyrazić w logice L_I.Okazuje się, że własności te dają się łatwo wyrazić w logice algorytmicznej:
(i) formuła $(\forall x)y := 0;$**while** $\neg x = y$ **do** $y := y + 1$ **od true** stwierdza, iż każdy element x jest wynikiem skończonej ilości dodawań 1 startując od 0, a zatem wyraża fakt, iż każda liczba naturalna jest skończona;
ten sam fakt da się wyrazić na wiele różnych (lecz równoważnych) sposobów, które można wykorzystywać w dowodach zależnie od potrzeb, np.:
(i') **while** $x > 0$ **do** $x := x-1$ **od true**;
(i") $y > 0 \rightarrow$ **while** $x \geq y$ **do** $x := x-y$ **od true**;
(ii) formuła $(\forall x)$**while** $\neg x =$ **empty do** $x := pop(x)$ **od true** stwierdza, iż każdy stos x zamienia się w stos pusty (empty) po zdjęciu zeń skończonej ilości elementów ($x := pop(x)$), a zatem wyraża fakt, iż każdy stos jest skończony.
Okazuje się ponadto, iż dołączenie formuły (i) stwierdzającej własność stopu bardzo prostego programu do aksjomatów 4.5(i)-4.5(iii), 4.5(v), 4.5(vi) powoduje iż uzyskana specyfikacja jest zupełna (por. def. 4.4). Podobnie formuła (ii) jest podstawą do uzyskania zupełnej specyfikacji stosów opartej o aksjomaty:
(i') $(\forall x) \neg x = empty \rightarrow x = push(top(s),pop(s))$;
(ii') $pop(empty) = empty \wedge (\forall x,y)x = pop(push(y,x))$;
(iii') $top(empty) = ERROR \wedge (\forall x,y)y = top(push(y,x))$;
(iv') $(\forall x,y) \neg push(y,x) = empty$;
(v') formuła (ii).//

Można wykazać, że nie istnieje skończony, poprawny i pełny system wnioskowania dla logiki algorytmicznej. W przypadku logiki Hoare'a podobna własność doprowadziła do rozważania nieklasycznego pojęcia pełności. Podejście proponowane w przypadku bardziej skomplikowanej logiki, jaką jest logika algorytmiczna, polega na rezygnacji ze skończoności systemu wnioskowania:

7.3 Definicja: Przez system wnioskowania S_A dla logiki algorytmicznej rozumieć będziemy następujący zbiór reguł wnioskowania:

(i) reguły systemu wnioskowania dla L_I, podane w 3.4, z tą różnicą, że metazmienne przebiegają teraz formuły L_A;
(ii) $\vdash [x:=t]A \rightarrow A(x/t)$;
(iii) $\vdash P(A \wedge B) \leftrightarrow (PA \wedge PB)$; $\vdash P(A \vee B) \leftrightarrow (PA \vee PB)$;
(iv) $\vdash (P;Q)A \rightarrow P(QA)$;
(v) $\vdash$ (**if** G **then** P **else** Q **fi** A) $\leftrightarrow ((G \wedge PA) \vee (\neg G \wedge QA))$;
(vi) $\vdash$ (**while** G **do** P **od** A) $\leftrightarrow ((\neg G \wedge A) \vee (G \wedge P;$**while** G **do** P **od** A));
(vii) $A \rightarrow B \vdash PA \rightarrow PB$;
(viii) $\{$**if** G **then** P **else** Q **fi**$^i(\neg G \wedge A) \rightarrow B\}_{i \in N} \vdash$ (**while** G **do** P **od** A) $\rightarrow B$.//

Reguła (viii) jest nieskończona, bowiem jej zbiór przesłanek jest nieskończony (ponieważ zawiera wszystkie formuły postaci **if** G **then** P **else** Q **fi**$^i(\neg G \wedge A) \rightarrow B$, gdzie i jest dowolną liczbą naturalną).

7.4 **Przykład**: Jako przykład użycia systemu S_A rozważymy dowód faktu, iż:

$$\vdash \textbf{while } G \textbf{ do } K \textbf{ od true} \rightarrow \textbf{while } G \textbf{ do } K \textbf{ od} \neg G$$

Wykażemy indukcyjnie, że dla każdego $i \in N$:

$$\vdash \textbf{if } G \textbf{ then } K \textbf{ fi}^i \neg G \rightarrow \textbf{while } G \textbf{ do } K \textbf{ od} \neg G.$$

Dla i = 0 fakt ten jest oczywisty (korzystamy bezpośrednio z aksjomatu 7.3.vi). Pokażemy teraz, że z jego prawdziwości dla i wynika także prawdziwość dla i + 1. Mamy, z założenia indukcyjnego, $\vdash$ **if** G **then** K **fi**$^i \neg G \rightarrow$ **while** G **do** K **od** $\neg$ G. Na mocy reguły 7.3.vii (po przyjęciu za P programu **if** G **then** K **fi**) mamy więc:

$$\vdash \textbf{if } G \textbf{ then } K \textbf{ fi} ; \textbf{if } G \textbf{ then } K \textbf{ fi}^i \neg G \rightarrow \textbf{if } G \textbf{ then } K \textbf{ fi}; \textbf{while } G \textbf{ do } K \textbf{ od} \neg G,$$

czyli $\vdash$ **if** G **then** K **fi**$^{i+1} \neg G \rightarrow$ **if** G **then** K **fi**;**while** G **do** K **od** $\neg$ G. Rozważmy następnik implikacji. Na mocy 7.3.v jest on równoważny (G $\wedge$ K;**while** G **do** K **od** $\neg$ G) $\vee$ ($\neg$G $\wedge$ $\neg$G), a na mocy 7.3.vii - formule **while** G **do** K **od** $\neg$ G. Mamy więc:

$$\vdash \textbf{if } G \textbf{ then } K \textbf{ fi}^{i+1} \neg G \rightarrow \textbf{while } G \textbf{ do } K \textbf{ od} \neg G,$$

co kończy naszą indukcję. Do wykazania dowodzonego faktu wystarczy teraz skorzystać z reguły 7.3.viii.//

7.5 **Przykład**: Pokażmy, iż program obliczania ilorazu i reszty z dzielenia przedstawiony w przykładzie 6.4 zatrzymuje się. Fakt ten wyrazimy następująco:

$$(x \geq 0 \wedge y > 0) \rightarrow \textbf{q:=0;r:=x;while } r \geq y \textbf{ do r:=r-y; q:=1+q od true}.$$

Podobnie, jak w przykładzie 7.4 nietrudno wykazać poprawność następującej reguły (por. np. [MS'87]):
M**true** $\vdash$ **while** G **do** K **od true** $\rightarrow$ **while** G **do** K; M **od true**, gdzie zmienne z M nie występują ani w G, ani w K.
Wiemy, że $\vdash y > 0 \rightarrow$ **while** $x \geq y$ **do** x:=x-y **od true** (gdyż jest to aksjomat liczb naturalnych - por. 7.2.i"). Podstawiając w nim r w miejsce x, a następnie korzystając z klasycznego rachunku zdań, powyżej podanej reguły (przyjmując w niej za G formułę $x \geq y$, zaś za K i M odpowiednio r:=r-y i q:=1+q), oraz z reguł 7.3.ii, 7.3.vii łatwo uzyskamy dowodzony fakt.//

Poniższe twierdzenie wykazuje, iż przytoczony system wnioskowania osiąga klasyczny ideał - jest poprawny i pełny.

7.6 **Twierdzenie**: System wnioskowania S_A dla logiki algorytmicznej jest poprawny i pełny (w klasycznym sensie zdefiniowanym w 3.6).//

Zauważmy, że w praktyce dowodzenia często korzystamy z tzw. wtórnych reguł

wnioskowania, czyli reguł, które wprawdzie w procesie dowodzenia nie są niezbędne, ale które w wielu przypadkach skracają i uczytelniają przeprowadzane dowody. Taką regułą była np. reguła 7.4(i); mogą nimi być (odpowiednio zmodyfikowane co do składni) reguły Hoare'a itd. Warto podkreślić, że twierdzenie o pełności zapewnia dodatkowo, iż wszystkie poprawne wtórne reguły wnioskowania dają się udowodnić w systemie S_A.

8. Logika dynamiczna - niedeterminizm

Logika dynamiczna zdefiniowana została przez V. Pratta w połowie lat siedemdziesiątych. Jej rozszerzenie do logiki pierwszego rzędu wraz z badaniami teoriodowodowymi przedstawione zostało w [Har'78]. Logika dynamiczna ma wiele wspólnego z logiką algorytmiczną. Jej atrakcyjność polega na prostym i czytelnym wprowadzeniu niedeterminizmu (późniejsze badania niedeteminizmu w logice algorytmicznej poszły w zbliżonym, jednakże nieco innym kierunku) i zdefiniowaniu naturalnego systemu dowodzenia własności programów niedeterministycznych.

Wprowadzenie niedeterminizmu rodzi nowe pytania związane z pojęciem poprawności programu. Istnienie wielu możliwych wartościowań końcowych programu wymaga podjęcia decyzji czy program uważać będziemy za poprawny np. gdy dla każdego wartościowania wejściowego spełniającego warunek początkowy:
- istnieje obliczenie skończone, którego wartościowanie wyjściowe spełnia warunek końcowy (takie pojęcie poprawności uzasadnione jest definicją akceptowalności przez niedeterministyczne maszyny Turinga);
- każde obliczenie jest skończone i każde wartościowanie wyjściowe spełnia warunek końcowy (jest to chyba najbardziej naturalne rozszerzenie pojęcia poprawności zdefiniowanego dla programów sekwencyjnych, dające pewność co do wyniku bez względu na dokonywane niedeterministyczne wybory);
- każde wartościowanie wyjściowe spełnia warunek końcowy (jest to raczej odpowiednik częściowej poprawności).
W logice dynamicznej pierwsze i trzecie pojęcie jest wyrażalne bezpośrednio, zaś drugie - w klasie tzw. interpretacji arytmetycznych, do których to zawęża się badania teoriodowodowe.

8.1 Definicja: *Logiką dynamiczną* nazwiemy trójkę $L_D = <F_D, I_D, Sat_D>$, gdzie:
- F_D definiujemy rozszerzając definicję formuł logiki L_I (por. 2.3) o regułę stwierdzającą, iż $<P>A$ jest formułą L_D, jeśli A jest formułą logiki L_D, zaś P jest programem zapisanym w języku programowania z przykładu 5.1 (intencją formuły $<P>A$ jest wyrażenie istnienia obliczenia programu P, którego wartościowanie wyjściowe spełnia formułę A);
- I_D jest klasą systemów relacyjnych (por. 2.3);
- Sat_D jest rozszerzeniem Sat_I (por. 2.3) o regułę stwierdzającą, iż $R\ Sat_A <P>A$ wttw, gdy dla dowolnego wartościowania zmiennych v: $(<P>A)_R(v) = \mathbf{true}$, gdzie wartość formuły $<P>A$ w systemie relacyjnym R i przy wartościowaniu zmiennych v, $(<P>A)_R(v)$, definiujemy indukcyjnie:
$(<P>A)_R(v) = \mathbf{true}$ wttw, gdy istnieje wartościowanie v' takie, że (v,v') należy do zbioru $W_D(P)$ (zdefiniowanego w 5.1) i $A_R(v') = \mathbf{true}$.//

Powyższa definicja dopuszcza jedynie możliwość wyrażania istnienia obliczenia o pewnej

własności. Można jednak bezpośrednio wyrazić fakt, oznaczany przez [P]A, iż każde wartościowanie wyjściowe P spełnia warunek końcowy A: $[P]A \leftrightarrow \neg <P> \neg A$. Ponadto we wspomnianej klasie struktur arytmetycznych można wyrazić praktycznie wszystkie interesujące własności programów niedeterministycznych, a w ich liczbie drugie z rozpatrywanych wcześniej pojęć poprawności. Można także wyrazić wiele własności struktur danych.

8.2 Przykład: Oczywiście własności skończoności elementów typu danych, wyrażane w logice algorytmicznej, można też łatwo wyrazić w logice dynamicznej:
(i) formuła $(\forall x) <y:=0(y:=y+1)^*>x=y$

wyraża fakt,

iż każda liczba naturalna jest skończona; podobny fakt wyrażają formuły:
(i') $<(x:=x\text{-}1)^*>x=0$;
(i") $y>0 \rightarrow <(x:=x\text{-}y)^*>x<y$;
(ii) formuła $<(x:=pop(x))^*>x=$empty stwierdza, iż każdy stos x zamienia się w stos pusty (empty) po zdjęciu zeń pewnej skończonej ilości elementów, a zatem wyraża fakt, iż każdy stos jest skończony.//

Oczywiście logika dynamiczna, jako rozszerzenie logiki algorytmicznej, nie ma skończonego, poprawnego i pełnego systemu wnioskowania. Można podać dla niej poprawny i pełny system nieskończony, lub system skończony arytmetycznie pełny. Pojęcie arytmetycznej pełności zdefiniowane zostało przez Harela [Har'78] jako wariant relatywnej pełności (por. 6.5). Przytoczmy teraz system wnioskowania Harela.

8.3 Definicja: System wnioskowania S_D dla logiki dynamicznej składa się z następujących reguł wnioskowania (i aksjomatów):
(i) reguły (i aksjomaty) klasycznego rachunku zdań (por. 3.3), przy czym metazmienne przebiegają formuły L_D;
(ii) $[x:=t]A \leftrightarrow A(x/t)$; $[G?]A \leftrightarrow (G \rightarrow A)$; $[P;Q]A \leftrightarrow [P][Q]A$; $[P+Q]A \leftrightarrow ([P]A \wedge [Q]A)$;
(iii) $A \rightarrow B \mid\!- [P]A \rightarrow [P]B$; $A \rightarrow B \mid\!- \exists xA \rightarrow \exists xB$;
(iv) $A \rightarrow [P]A \mid\!- A \rightarrow [P^*]A$;
(v) $B \rightarrow \exists nA(n), A(n+1) \rightarrow <P>A(n) \mid\!- B \rightarrow <P^*>A(0)$, gdzie n jest zmienną reprezentującą liczby naturalne i nie występującą w P.//

Zdefiniujmy teraz pojęcie arytmetycznej interpretacji (struktury danych).

8.4 Definicja: Strukturę danych $R = <D,F,P>$ nazwiemy *arytmetyczną* wttw, gdy:
(i) D zawiera zbiór liczb naturalnych;
(ii) F zawiera funkcje 0, 1, +, * (o standardowym znaczeniu dla liczb naturalnych);
(iii) P zawiera relację nat: $D \times D \rightarrow \{true,false\}$, o tej własności, że $nat(d) = true$ wttw, gdy d jest liczbą naturalną;
(iv) P zawiera relację c: $D \times D \times D \rightarrow \{true,false\}$, kodującą ciągi elementów z D poprzez jeden element (tj. dla każdej liczby naturalnej n: $(\forall x_1,\ldots x_n)((nat(i) \wedge n \geq i) \rightarrow (c(x,i,y) \leftrightarrow x = x_i))$).//

System wnioskowania S_D nie jest poprawny w klasycznym sensie (por. 3.6), ale jest poprawny, jeśli rozpatrywać jako interpretacje dopuszczalne jedynie interpretacje arytmetyczne.

8.5 Definicja: Powiemy, że system wnioskowania P jest *arytmetycznie pełny (poprawny)* wttw, gdy jest relatywnie pełny (poprawny) względem klasy interpretacji arytmetycznych

(relatywną pełność i poprawność zdefiniowaliśmy w def. 6.5)//

8.6 Twierdzenie: System wnioskowania S_D dla logiki dynamicznej jest arytmetycznie poprawny i arytmetycznie pełny//

8.7 Przykład: Jako przykład wnioskowania wykażmy, iż w dziedzinie liczb naturalnych następująca iteracja:

$$(x:=x-1 + x:=x+1)^*$$

ma obliczenie skończone, kończące się z wartością $x=0$ (np. gdy zawsze wybieramy do wykonania instrukcję $x:=x-1$, aż do osiągnięcia wymaganego warunku). Fakt ten wyrazimy formułą $<(x:=x-1 + x:=x+1)^*>x=0$. Aby go wykazać, skorzystamy z reguły 8.3(v), w której za P przyjmiemy program $(x:=x-1 + x:=x+1)$, za B - **true**, zaś za A(n) - formułę $x=n$. Oczywiście $B \rightarrow \exists n A(n)$ jest przy tym założeniu trywialną własnością struktury danych (liczb naturalnych). Aby wykazać, że $A(n+1) \Rightarrow <P>A(n)$, czyli $x=n+1 \Rightarrow <x:=x-1 + x:=x+1>x=n$, skorzystamy z odpowiednich części aksjomatu 8.3(ii), praw klasycznego rachunku zdań oraz z definicji []:

$\vdash <x:=x-1 + x:=x+1>x=n \Leftrightarrow \neg[x:=x-1 + x:=x+1]\neg x=n \Leftrightarrow \neg([x:=x-1]\neg x=n \wedge [x:=x+1]\neg x=n) \Leftrightarrow ([x:=x-1]\neg x=n \vee \neg[x:=x+1]\neg x=n) \Leftrightarrow (<x:=x-1>x=n \vee <x:=x+1>x=n) \Leftrightarrow (x-1=n \vee x+1=n)$. Oczywiście (z własności liczb naturalnych) $\vdash x=n+1 \Rightarrow (x-1=n \vee x+1=n)$, co kończy dowód.//

9. Logika temporalna - współbieżność

Dotychczas omawialiśmy logiki, których formuły zawierać mogą programy. Tego typu logiki nazywane są *logikami endogenicznymi*. Z drugiej strony rozważać można także inne podejście, związane z tzw. *logikami egzogenicznymi*. W logikach tych nie występuje pojęcie programu. Własności programów bada się konstruując dodatkowy pomost pomiędzy rozważanymi pojęciami programistycznymi a samą logiką. Zaletą logik egzogenicznych jest możliwość rozważania wielu różnych narzędzi programistycznych bez konieczności zmiany logiki, podczas gdy w logikach endogenicznych (np. logice Hoare'a, logice algorytmicznej czy dynamicznej) mamy do czynienia z ustalonym raz na zawsze pojęciem programu. Za ogólność rozwiązania płacimy jednak koniecznością wykonywania kroku pośredniego związanego z definicją semantyki programów wyrażoną przy pomocy formuł wybranej logiki. Najpopularniejszymi logikami egzogenicznymi spotykanymi w informatyce są logiki temporalne (za logikę egzogeniczną można uznać także logikę klasyczną definiowaną w 2.3, jednakże ze względu na brak miejsca nie będziemy omawiać tego zagadnienia bliżej). W logikach temporalnych formuły służą do wyrażania własności obiektów i sytuacji zależnych od czasu. Logiki te rozszerzają bardziej tradycyjne podejście do weryfikacji programów oparte o model relacyjny. Rozszerzenie to wyraża się w możliwości rozważania modelu interakcyjnego (por. 5.2), bowiem programy zmieniają swoje stany w miarę upływu czasu (na czas patrzymy w sposób abstrakcyjny; np. w przypadku programów jest on odmierzany kolejnymi zmianami stanów spowodowanymi wykonaniami poszczególnych instrukcji).

Poszczególne logiki temporalne różnią się między sobą przyjętą strukturą czasu oraz przyjętą postacią formuł. W tym rozdziale omówimy najpopularniejszą z logik temporalnych, a mianowicie logikę z czasem dyskretnym, którego punkty odpowiadają liczbom naturalnym.

Nawet przy tak zawężonym pojęciu czasu rozważa się jeszcze dwie wersje logiki temporalnej. Pierwsza z nich (z tzw. *semantyką początkową*) zakłada iż formuła jest prawdziwa, gdy ustalimy stan początkowy (por. np. [MP'82]). Druga natomiast (z tzw. *semantyką normalną*) zakłada, iż prawdziwość formuły nie zależy od wyboru stanu początkowego (por. np. [Krö'87]). Obie te logiki mają wiele cech wspólnych, w szczególności pierwsza jest bardziej ogólna niż druga, toteż ją właśnie omówimy.

Dotychczas rozpatrywaliśmy jedynie logiki będące rozszerzeniem klasycznej logiki pierwszego rzędu. Z drugiej strony, poważna część badań nad logikami programów obejmuje ich wersje zdaniowe. W procesie dowodzenia spełniają one wżną pomocniczą rolę, podobną do tej jaką odgrywa klasyczny rachunek zdań w dowodach opartych o klasyczną logikę pierwszego rzędu. Ponadto atrakcyjność zdaniowych wersji logik programów wynika z faktu, iż ich zbiory tautologii są rozstrzygalne (lub co najmniej częściowo rozstrzygalne). Ze względu na ograniczoną objętość tego opracowania nie byliśmy w stanie prześledzić dokładniej problematyki zdaniowych logik programów. Zarysujemy ją jednak teraz, na przykładzie logiki temporalnej.

9.1 Definicja: *Strukturą Kripkego* nazwiemy zestaw $<D,W,w_0,R,I>$, gdzie:
(i) D jest niepustym zbiorem, zwanym *dziedziną*;
(ii) W jest *zbiorem stanów*, $w_0 \in W$ jest *stanem początkowym*;
(iii) R jest *relacją dostępności stanów*;
(iv) I jest *interpretacją*, tj. odwzorowaniem przyporządkującym każdemu stanowi system relacyjny, którego zbiorem wartości jest D.
Powiemy, że struktura $<D,W,w_0,R,I>$ jest *standardowa*, jeśli W jest zbiorem liczb naturalnych, $w_0 = 0$, zaś R jest zwykłą relacją $<$ określoną dla liczb naturalnych.//

Intuicyjnie, w_0 jest stanem początkowym, zaś uRv oznacza, że stan v jest w przyszłości w stosunku do stanu u.

9.2 Definicja: *Zdaniową logiką temporalną* nazwiemy trójkę $L_{T_0} = <F_{T_0}, I_{T_0}, Sat_{T_0}>$, gdzie:
- F_{T_0} definiujemy rozszerzając definicję formuł logiki L_0 (por. 2.2) o regułę stwierdzającą, iż AatB jest formułą L_{T_0}, o ile A oraz B są formułami logiki L_{T_0};
- I_{T_0} jest klasą struktur Kripkego, w których D = {true,false}, zaś systemy relacyjne przyporządkowywane stanom przez interpretacje, mają pusty zbiór relacji, a ich zbiór funkcji zawiera jedynie stałe, które dla wygody nazwiemy tak samo jak zmienne zdaniowe (por. 2.2) - będą one nadawać znaczenie (wartości) zmiennym w poszczególnych stanach;
- Sat_{T_0} definiujemy indukcyjnie (zakładając, że $K = <D,W,w_0,R,I>$):
  - dla formuły A nie zawierającej operatora at: $K\ Sat_{T_0}\ A$ wttw, gdy $v_0\ Sat_{T_0}\ A$ (por. 2.2), gdzie $v_0(p) = p_{I(w_0)}$ (przypomnijmy, że $p_{I(w_0)}$ oznacza wartość stałej p w systemie relacyjnym $p_{I(w_0)}$),
  - $K\ Sat_{T_0}$ AatB wttw, gdy istnieje w_1 takie, że $w_0 R w_1$ i $<D,W,w_1,R,I> Sat_{T_0} (A \wedge B)$ i dla każdego u jeśli $w_0 R u R w_1$, to $<D,W,u,R,I> Sat_{T_0} - B$.//

Intuicyjnie, formuła AatB oznacza, iż formuła B jest spełniona w pewnym stanie w przyszłości (przyjmując w_0 za chwilę bieżącą) oraz A jest spełniona w pierwszym takim stanie. Zauważmy, że inne typowe operatory temporalne można łatwo wyrazić przy pomocy operatora at, np.:
- w następnej chwili zajdzie formuła A, oznaczany przez nxtA, wyrażamy przez Aat true;

- kiedyś w przyszłości zajdzie formuła A, oznaczany przez smtA, wyrażamy przez true atA;
- zawsze w przyszłości zachodzi formuła A, oznaczany przez alwA, wyrażamy przez $\sim$smt$\sim$A.

Podobnie, z łatwością można wyrazić wiele interesujących własności programów, np.:
- własności poprzedzania:
  - zajście zdarzenia B jest poprzedzone zajściem zdarzenia A: $\sim$Bat(A v B);
  - powtarzanie żądania A wymusza odpowiedź B: alw(smtA)$\Rightarrow$smtB;
  - ciągłe zachodzenie żądania A wymusza odpowiedź B: alwA$\Rightarrow$smtB;
- własności bezpieczeństwa:
  - jeśli stan początkowy spełnia A, to B jest zawsze spełnione: A$\Rightarrow$alwB;
  - częściowa poprawność: A$\Rightarrow$alw(KONIEC$\Rightarrow$B), gdzie KONIEC jest formułą opisującą stan końcowy;
  - brak blokady w stanie st: alw(st$\Rightarrow$nxt$\sim$st);
- własności żywotności:
  - jeśli stan początkowy spełnia A, to B będzie kiedyś spełnione: A$\Rightarrow$smtB;
  - całkowita poprawność: A$\Rightarrow$(BatKONIEC).

Oczywiście takie zwroty z powyższego opisu, jak "zajście zdarzenia", "żądanie", "odpowiedź", "stan" muszą być wyrażone (i dają się wyrazić) formułami logiki temporalnej.

9.3 Definicja: System wnioskowania S_{T_0} dla zdaniowej logiki temporalnej składa się z następujących reguł wnioskowania (i aksjomatów):
(i) reguły (i aksjomaty) klasycznego rachunku zdań (por. 3.3), przy czym metazmienne przebiegają formuły L_{T_0};
(ii) $\sim$nxtA $\Leftrightarrow$ nxt$\sim$A; nxt(A$\Rightarrow$B) $\Rightarrow$(nxtA $\Rightarrow$ nxtB);
(iii) alwA $\Rightarrow$ nxt alwA;
(iv) AatB $\Rightarrow$ smtB; AatB $\Leftrightarrow$ nxt(A $\wedge$ B) v nxt($\sim$B $\wedge$ AatB);
(v) A |- nxtA;
(vi) A$\Rightarrow$B, A$\Rightarrow$nxtA |- A$\Rightarrow$alwB.//

9.4 Twierdzenie: Powyższy system wnioskowania dla zdaniowej logiki temporalnej jest poprawny i pełny.//

Warto tu jeszcze podkreślić, że zbiór tautologii zdaniowej logiki temporalnej jest rozstrzygalny. Natomiast zbiór tautologii logiki temporalnej pierwszego rzędu, którą za chwilę omówimy, nie jest nawet częściowo rozstrzygalny.

W temporalnej logice zdaniowej znaczenie wszystkich symboli (są nimi zmienne zdaniowe) jest zależne od czasu (takie symbole nazwiemy *lokalnymi*). Natomiast w logice pierwszego rzędu jest wygodne rozpatrywanie również symboli, których znaczenie jest ustalone, niezależne od upływu czasu (nazwiemy je *symbolami globalnymi*). W niniejszym opracowaniu ograniczymy się do przypadku, w którym symbolami globalnymi są pewne wyróżnione zmienne

9.5 Definicja: *Logiką temporalną pierwszego rzędu* nazwiemy trójkę $L_T = <F_T, I_T, Sat_T>$, gdzie:
- F_T definiujemy rozszerzając definicję formuł logiki L_{T_0} (por. 9.2) o możliwość operowania kwantyfikatorami i symbolami funkcyjnymi i relacyjnymi; L_T uzyskujemy zatem z L_{T_0} podobnie jak uzyskaliśmy klasyczną logikę pierwszego rzędu z logiki zdaniowej;
- I_T jest klasą par $<K,v>$, gdzie v jest wartościowaniem zmiennych globalnych, zaś K jest strukturą Kripkego, w której systemy relacyjne przyporządkowywane stanom przez

interpretacje, zawierają (co najmniej) stałe, nazwane tak samo jak zmienne lokalne - będą one nadawać znaczenie (wartości) tym zmiennym w poszczególnych stanach;
- Sat_T definiujemy rozszerzając definicję Sat_{T_0} w sposób będący naturalnym uogólnieniem definicji spełnialności dla logiki klasycznej: wartość zmiennych globalnych jest dana przez wartościowanie v, zaś symboli lokalnych przez interpretacje zależne od stanów struktury Kripkego.//

9.6 Przykład: Jako przykład definicji semantyki programów w logice temporalnej pierwszego rzędu przedstawmy semantykę while-programów. Założymy w tym celu, iż w języku są dostępne zmienne zdaniowe nazwane OK, gdzie K jest programem, reprezentujące sterowanie programu (OK oznacza, iż do wykonania gotowa jest instrukcja K). Semantykę programów wyznaczają następujące schematy formuł:
(i) alw(O[];K ⇒ nxt OK), gdzie [] oznacza program pusty;
(ii) alw(Oz: = t;K ⇒ (t = x ⇒ nxt(z = x ∧ OK)), gdzie z jest zmienną lokalną, zaś x - globalną;
(iii) alw(O if G then K else M fi;L ⇒ ((G⇒nxt OK;L) ∧ (−G⇒nxt OM;L)));
(iv) alw(O while G do K od;L ⇒ ((G⇒nxt OK;while G do K od;L) ∧ (−G⇒nxt OL)));
uzupełnione o formuły stwierdzające, iż w każdej chwili dokładnie jedna zmienna postaci OK przyjmuje wartość true oraz, że podstawienie jest jedyną instrukcją mogącą powodować zmianę zmiennych. Można wykazać, że powyższa specyfikacja jest zupełna, tzn. że każda własność temporalna spełniona przez dany program da się udowodnić z formuł uzyskanych w jego przypadku z podanych schematów.
Rozszerzmy teraz powyższą definicję na przypadek prostych programów współbieżnych (rozważać będziemy programy postaci $K_1||...||K_n$, gdzie $K_1,...,K_n$ są while-programami (semantykę || podaliśmy w 5.2)). Ponieważ poszczególne procesy (while-programy) działają w środowisku innych procesów, przyjmiemy iż wiele instrukcji może być gotowych do wykonania. Ponadto, zgodnie z przyjętą w 5.2 semantyką przeplotową, gotowość do wykonania instrukcji nie oznacza konieczności jej wykonania w następnej chwili - może się bowiem wykonać instrukcja gotowa do wykonania w innym procesie. Wprowadzimy zatem dodatkowe zmienne zdaniowe postaci inK, gdzie K jest while-programem, oznaczające, że w danej chwili wykonuje się instrukcja programu K. Uzyskamy zatem następującą specyfikację;
(i') alw(O[];K ⇒ (nxtOK)at in[];K);
(ii') alw(Oz: = t;K ⇒ ((t = x ⇒ nxt(z = x ∧ OK))at inz: = t;K);
(iii') alw(O if G then K else M fi;L ⇒ ((G⇒nxtOK;L) ∧ (−G⇒nxtOM;L))at (in if G then K else M fi;L);
(iv') alw(O while G do K od;L ⇒ ((G⇒nxt OK;while G do K od;L) ∧ (−G⇒nxt OL)) at in while G do K od;L).
Specyfikację tę należy uzupełnić o formuły swierdzające, iż w każdej chwili dokładnie jedna zmienna postaci inK przyjmuje wartość true oraz, że podstawienie jest jedyną instrukcją mogącą powodować zmianę zmiennych. Tak uzyskana specyfikacja jest zupełna, podobnie jak wcześniejsza, podana dla while-programów.//

9.7 Przykład: Logika temporalna jest także przydatna w specyfikacjach własności typów danych (por. [Sza'88]). Dla przykładu podajmy formułę wyrażającą skończoność liczb naturalnych:

$$z = 0 \wedge (\forall x)\overset{\text{alw}}{z} = x \Rightarrow \text{nxt}(z = x + 1) \wedge (\forall x)\text{smt}(z = x),$$

gdzie z jest zmienną lokalną, zaś x - zmienną globalną. Formuła ta stwierdza, że w kolejnych stanach struktury Kripkego wartościami zmiennej z są kolejne liczby naturalne (skończone), zaś każdy element z dziedziny przyjmuje kiedyś wartość równą z (a więc wartość pewnej liczby naturalnej).//

Dla logiki temporalnej pierwszego rzędu nie istnieje skończony poprawny i pełny system wnioskowania. Można podać system pełny nieskończony (por. [Sza'87]) oraz skończony arytmetycznie pełny (por. [Sza'87/88]). Istnieje jednak także inne interesujące pojęcie pełności, tzw. niestandardowa pełność, bazująca na wynikach Henkina związanych z prostą teorią typów (por. np. [Mon'76, Tak'75]), a do logik programów wprowadzona przez I. Nemetiego. Niestandardowość podejścia polega na dopuszczeniu tzw. niestandardowych obliczeń programów, tj. obliczeń, w których kolejne chwile czasu nie muszą odpowiadać liczbom naturalnym. O czasie przyjmuje się tylko pewien zestaw aksjomatów i dopuszcza dowolne modele tych aksjomatów jako możliwe pojęcia czasu. Tego typu podejście okazuje się bardzo silne i wydaje się, iż szczególnie dobrze pasuje do problematyki automatyzacji procesu dowodzenia twierdzeń o programach. W kontekście logiki temporalnej pojęcie niestandardowej pełności badane było np. przez Abadiego [Aba'88]. Ze względu na ograniczoną objętość niniejszego opracowania nie będziemy wnikali w techniczne szczegóły tego podejścia, poprzestając jedynie na zwróceniu nań uwagi, jest ono bowiem ważnym i ciekawym podejściem, alternatywnym dla rozwiązań prezentowanych wcześniej.

BIBLIOGRAFIA

[AA'82] S. Alagić, M.A. Arbib: *Projektowanie programów poprawnych i dobrze zbudowanych*, WNT 1982.

[Aba'88] M. Abadi: *The Power of Temporal Proofs*, SRC Report 30, Palo Alto, California 1988.

[BK'82] L. Banachowski, A. Kreczmar: *Elementy analizy algorytmów*, WNT 1982.

[Coo'78] S.A. Cook: *Soundness and Completeness of Axiom System for Program Verification*, SIAM J. Comput. 7, 1978, 70-90.

[Dij'85] E.W. Dijkstra: *Umiejętność programowania*, WNT 1985.

[EFT'84] H.D. Ebbinghaus, J. Flum, W. Thomas: *Mathematical Logic*, Springer-Verlag 1984.

[Grz'75] A. Grzegorczyk: *Zarys logiki matematycznej*, PWN 1975.

[Har'78] D. Harel: *First-order Dynamic Logic*, LNCS 68, Springer-Verlag 1978.

[Hoa'69] C.A.R. Hoare: *An Axiomatic Basis for Computer Programming*, CACM 12, 1969, 576-583.

[Krö'87] F. Kröger: *Temporal Logic of Programs*, Springer-Verlag 1987.

[Lyn'78] R.C. Lyndon: *O logice matematycznej*, PWN 1978.

[Mon'76] J.D. Monk: *Mathematical Logic*, Springer-Verlag 1976.

[MP'82] Z. Manna, A. Pnueli: *Verification of Concurrent Programs - the Temporal Framework*, w: *The Correctness Problem in Computer Science* (red. R.S. Boyer, J.S. Moore), Academic Press 1982, 215-373.

[MS'87] G. Mirkowska, A.Salwicki: *Algorithmic Logic*, PWN oraz D. Reidel Pub. Co. 1987.

[Sza'87] A. Szałas: *A Complete Axiomatic Characterization of First-Order Temporal Logic of Linear Time*, TCS 54, 1987, 199-214.

[Sza'87/88] A. Szałas: *Arithmetical Axiomatization of First-Order Temporal Logic*, IPL 26, 1987/88, 111-116.

[Sza'88] A. Szałas: *Towards the Temporal Approach to Abstract Data Types*, Fundamenta Informaticae 11, 1988, 49-64.

[Tak'75] G. Takeuti: *Proof Theory*, North-Hollang Pub. Co. 1975.

[Tur'85] W.M. Turski: *Metodologia programowania*, WNT 1982.

N O T A T K I

NOTATKI

N O T A T K I