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

Roman KONIECZNY

SPECYFIKACJA FORMALNA W PROJEKTOWANIU PROGRAMÓW KOMPUTEROWYCH

Streszczenie. Artykuł niniejszy dotyczy zagadnień formułowania specyfikacji programów komputerowych. Zagadnienie to staje się szczególnie ważne w przypadku projektowania złożonych systemów. O ważności tej problematyki świadczy duża liczba prac teoretycznych. Artykuł niniejszy stanowi próbę uporządkowania najważniejszych pojęć, definicji i konstrukcji, stanowiących formalne podstawy tworzenia specyfikacji programów komputerowych. Do zagadnień z tym związanych zaliczyć można: specyfikowanie funkcji, specyfikowanie operacji, pojęcie stanu oraz specyfikowanie typów danych. Podano przykłady różnych specyfikacji tego samego problemu.

W ciągu ostatnich kilku lat wzrosła znacznie liczba prac poświęconych zagadnieniu specyfikacji formalnej programów komputerowych. Zagadnienie to jest szczególnie ważne w przypadku projektowania dużych systemów, realizowanych przez kilkudziesięcioosobowe zespoły programistów.

1. Wprowadzenie

W konstrukcji oprogramowania można wyróżnić następujące fazy:

- ustalenie wymagań użytkownika,
- sformułowanie specyfikacji,
- wykonanie projektu,
- konstrukcja modułów programowych i ich testowanie,
- testowanie systemu,
- konserwacja (pielęgnacja) systemu.

Artykuł niniejszy dotyczy zagadnień formułowania specyfikacji programów komputerowych. Zagadnienie to staje się szczególnie ważne w przypadku projektowania złożonych systemów. O ważności tej problematyki świadczy duża liczba prac teoretycznych, których wyniki przedstawiono m.in. w publikacjach [1-2], [4-13] oraz [16-17]. Artykuł niniejszy stanowi próbę uporządkowania najważniejszych pojęć, definicji i konstrukcji, stanowiących formalne podstawy tworzenia specyfikacji programów komputerowych.

2. Specyfikacja programów - pojęcia podstawowe

W punkcie tym dokonano przeglądu zagadnień dotyczących specyfikowania funkcji oraz specyfikowania operacji.

2.1. Specyfikowanie funkcji

Jednym z podstawowych typów modułów stosowanych w realizacji oprogramowania są funkcje. Funkcję można uważać za regułę przypisującą argumentom pewne wartości. Na przykład:

$$f(x,y,z) \triangleq x * x + y + 3 * z$$

Jawna definicja funkcji zawiera lewą i prawą stronę, która rozdzielona jest symbolem definicji $\triangleq$. Lewa strona składa się z nazwy funkcji i z umieszczonej w nawiasach listy nazw stanowiących *parametry*. Prawa strona jest wyrażeniem zbudowanym z nazw parametrów, stałych, operatorów oraz nazw funkcji. Proces obliczania wyniku funkcji dla danych argumentów nazywany jest zastosowaniem funkcji do tych argumentów. Jednym ze sposobów ograniczenia rodzajów argumentów, jakie wolno stosować do funkcji, jest podanie opisu typu funkcji. Opis typu funkcji zawiera nazwę funkcji oraz listę nazw zbiorów, z których mogą pochodzić poszczególne argumenty. Bardzo pożytecznym, okazuje się, zapisanie nazwy zbioru dla ewentualnych wyników funkcji. Nazwę tę umieszcza się po znaku --> . Np.:

$$f : D1, D2, \ldots, Dn \dashrightarrow R$$

powyższy zapis dotyczy funkcji f , której n argumentów pochodzi ze zbiorów nazwanych odpowiednio $D1 \ldots Dn$, zaś wyniki należą do zbioru R . Zbiór możliwych argumentów funkcji zaznaczony w opisie typu nazywany jest *dziedziną* tej funkcji, zaś zbiór możliwych wyników nazywany jest jej *zakresem wartości (przeciwdziedziną)*. Jeżeli np. *Nat* oznacza zbiór liczb naturalnych, to opisem typu funkcji *f1* mógłby być:

$$f1 : Nat, Nat \dashrightarrow Nat \ .$$

Jawna definicja funkcji jest w pewnym sensie analogiczna do programu, gdyż podaje metodę obliczania funkcji. Sednem specyfikacji niejawnej jest określenie relacji między argumentami a wynikami funkcji, bez konieczności jawnego podawania sposobu obliczania tych wyników.

Funkcja, której zakresem wartości jest zbiór dwóch wartości TRUE i FALSE , nazywana jest *predykatem* . Zbiór tych dwu wartości jest zbiorem wartości boolowskich i oznaczany często jako *Bool* .

Poniżej pokazano standardowy zapis niejawnej specyfikacji funkcji [7,13]:

$f : D1, D2, \ldots, Dn \dashrightarrow R$
$pre\text{-}f(d1,d2,\ldots,dn) \triangleq \ldots$
$post\text{-}f(d1,d2,\ldots,dn,r) \triangleq \ldots$

gdzie:

pre-f: *D1, D2, ..., Dn --> Bool*

post-f: *D1, D2, ..., Dn, R --> Bool*

Warunek wejściowy jest predykatem dotyczącym argumentów funkcji. Warunek wyjściowy jest predykatem dotyczącym argumentów i wyniku.
Nazwy warunków wejściowych i wyjściowych tworzone są poprzez dodanie przedrostka (odpowiednio *pre-* lub *post-*) do nazwy specyfikowanej funkcji. Na przykład dla funkcji *sqrt* (pierwiastek kwadratowy) daje to:

sqrt: Real, Real --> Real

$pre\text{-}sqrt(x,t) = x \geq 0 \wedge t \geq 0$

$post\text{-}sqrt(x,t,r) \triangleq abs(r ** 2 - x) \leq t$

gdzie: *r* - oznacza wynik (ang. result), *abs* - oznacza znaną wcześniej funkcję *wartość bezwzględna* .

Ostatnim parametrem *r* w definicji predykatu *post-sqrt* jest wynik funkcji. Trzyczęściowa specyfikacja oznacza więc, że dla dowolnych argumentów określonych, jako poprawne (tj. spełniających warunek wejściowy *pre-sqrt*) każda funkcja *sqrt* zgodna z tą specyfikacją musi dawać wynik, dla którego (wraz z argumentami) warunek wyjściowy *post-sqrt* da wynik TRUE .
Formalnie można zapisać to następująco:

$$x_0 \in Real \wedge t_0 \in Real \wedge pre\text{-}sqrt(x_0, t_0) \Rightarrow post\text{-}sqrt(x_0, t_0, sqrt(x_0, t_0))$$

W [13] przyjęto konwencję notacyjną, że nazwy predykatów nie będących nazwami warunków wejściowych i wyjściowych, powinny zaczynać się od przedrostka *is-* (jest). Przyjęto również, że jeżeli w wyrażeniu obliczającym wartość funkcji jakiś składnik (element) używany jest kilkakrotnie - wówczas możliwy jest skrótowy zapis przy użyciu konstrukcji <u>let</u> (niech). W konstrukcji <u>let</u> określona jest pewna stała o znanej (lokalnie) wartości.

W celu uzyskania zwięzłości zapisów w notacjach specyfikacji stosowane są wyrażenia kwantyfikatorowe. Wyrażenie kwantyfikatorowe buduje się z następujących elementów: *kwantyfikator*, *zmienna związana*, *ograniczenie* i *treść*, w której zwykle co najmniej raz użyta jest zmienna związana.
Stosowane są następujące symbole kwantyfikatorów:

∀ - kwantyfikator ogólny - (oznacza: *dla każdych możliwych wartości ... zachodzi ...*),

∃ - kwantyfikator szczegółowy (egzystencjarny) - (oznacza: *istnieje pewna wartość ... , dla której zachodzi ...*) ,

∃!- kwantyfikator szczegółowy - (oznacza: *istnieje dokładnie jedna wartość ... , dla której zachodzi ...*).

Kwantyfikator egzystencjarny uważany jest za skrót alternatywy.

Wyrażenia logiczne zawierające kwantyfikatory stanowią fragment, tzw. *rachunku predykatów* .

Przykładowy zapis:

$$\forall (n \in (i:j)) (\text{is-even} (2*n))$$

może być przeczytany jako: *dla każdego* n *należącego do zbioru liczb całkowitych od* i *do* j *włącznie, zachodzi że wartość predykatu* is-even(2*n) *równa jest* TRUE (gdzie "even" oznacza funkcję określającą parzystość liczby całkowitej).

Specyfikacje formalne są bardzo ważnym narzędziem dla projektantów oprogramowania. Oprócz precyzji, jednoznaczności i zwięzłości zapisów, dodatkową korzyścią wynikającą z ich stosowania - jest możliwość przeprowadzenia na ich podstawie dowodu poprawności. *Poprawność* funkcji można rozważać tylko *względem specyfikacji*.
Mając formalną specyfikację niejawną oraz jawną definicję funkcji, można próbować skonstruować dowód, że ta druga spełnia wymagania pierwszej. Na przykład, mając daną specyfikację [13]

$$max : Int , Int \dashrightarrow Int$$
$$post\text{-}max (x,y,r) \triangleq r \geq x \wedge r \geq y \wedge (r = x \vee r = y)$$

i realizację

$$max (x,y) \triangleq \underline{\text{if}}\ x \geq y\ \underline{\text{then}}\ x\ \underline{\text{else}}\ y$$

można udowodnić, że

$$(\forall x,y \in Int) (pre\text{-}max (x,y) \Rightarrow post\text{-}max (x,y,max(x,y))))$$

Int oznacza zbiór liczb całkowitych.

Funkcje ogólnie można podzielić na: bezwarunkowe, warunkowe oraz rekurencyjne.

Najprostszą funkcją jest *funkcja stała*. Przykładowa specyfikacja niejawna takiej funkcji może mieć postać:

$$pi : Real \dashrightarrow Real$$
$$post\text{-}pi (x,r) \triangleq r = \pi$$

Real oznacza zbiór liczb rzeczywistych.

Oprócz funkcji stałej bardzo prostą funkcją jest też *funkcja identyczności*, tj. taka, której wynik jest dokładnie równy argumentowi. Specyfikacja niejawna funkcji identyczności jest następująca:

$$identity : Real \dashrightarrow Real$$
$$post\text{-}identity (x,r) \triangleq r = x$$

Oczywistą realizacją takiej funkcji jest: $identity (x) \triangleq x$.

Funkcje bezwarunkowe można porównać do programów "liniowych" (tj. bez instrukcji skoku, instrukcji warunkowych, czy pętli). Tzw. *funkcje warunkowe* można w pewnym sensie porównać do programów z rozgałęzieniami. Zasadniczym elementem używanym w definicjach funkcji warunkowych jest *wyrażenie warunkowe* . Przykładem realizacji wyrażeń warunkowych może być instrukcja

<u>if</u> W <u>then</u> I1 <u>else</u> I2

gdzie:
W - wyrażenie warunkowe, I1, I2 - alternatywne ciągi instrukcji.

Funkcje rekurencyjne różnią się jedynie tym od innych funkcji, że w ich definicji używana jest ta sama, definiowana właśnie funkcja.

Specyfikując funkcje, należy również pamiętać o "miejscach", w których funkcja może mieć wartości nieokreślone. Jedną z zalet stosowania kwantyfikatorów z ograniczeniami - jest zmniejszenie niebezpieczeństwa występowania nieokreślonych wartości wyrażeń.
Wartości nieokreślone można specyfikować używając słowa kluczowego UNDEFINED .

2.2. Pojęcie stanu

Wstępnym krokiem do definiowania programów jest zapoznanie się z notacją dotyczącą stanów np. [4], [5], [7], [13] . *Stan* jest zestawem pewnych wartości opatrzonych nazwami. Klasy dopuszczalnych stanów określa się, definiując nazwy (zmiennych) i zbiory wartości, które te zmienne mogą przybierać.

Definicja klasy obiektów składa się z nazwy nadawanej tej klasie, symbolu :: , i z właściwego opisu klasy. Opis ten stanowi lista pozycji, z których każda zawiera nazwę (interpretowaną, jako nazwa zmiennej) i określenie zbioru oddzielone od tej nazwy dwukropkiem. Np.:

Obiekt_typu_X :: N: Int, R: Real, B: Bool

Formuła ta oznacza, że każdy obiekt należący do klasy *Obiekt_typu_X* zawiera dokładnie trzy zmienne. Zmienne te mają nazwy *N,R* oraz *B*, i przybierają wartości odpowiednio ze zbiorów *Int*, *Real* oraz *Bool* .

Według konwencji [13] nazwy klas obiektów rozpoczynane są zawsze dużą literą, zaś następne litery są małe.

2.3. Specyfikowanie operacji

Pojedynczą instrukcję języka programowania, ciąg instrukcji lub też cały program można ogólnie nazwać *operacją* .

Typ funkcji przedstawiony w p.2.1. opisywany był za pomocą formuły

$$f : D \dashrightarrow R$$

Typ operacji może być opisywany $OP : Stan \dashrightarrow Stan$.
Specyfikacja operacji może być przedstawiana według następującego schematu, np.:

OP

stany : Stan,

gdzie OP oznacza nazwę operacji (zestaw dużych liter).

Specyfikacje funkcji (p.2.1.) zawierają dwie najistotniejsze części: warunek wejściowy (*pre-*) i warunek wyjściowy (*post-*). Te same części wymagane są dla operacji. Warunek wejściowy dla operacji będzie jednak predykatem dotyczącym stanu, zaś warunek wyjściowy predykatem dotyczącym dwóch stanów (początkowego i końcowego). Warunek wejściowy opisuje więc podzbiór wszystkich stanów, na którym operacja powinna działać prawidłowo. Za pomocą warunku wyjściowego określana jest żądana relacja między stanem początkowym a końcowym.

Specyfikacja operacji zawiera trzy zasadnicze elementy przedstawione poniżej (jako specyfikacja niejawna):

OP
stany : Stan
pre-OP (s) $\triangleq$ ...
post-OP (s,s') $\triangleq$ ...
gdzie:
Stan :: ...
pre-OP : Stan --> Bool
post-OP : Stan, Stan --> Bool

Znaczenie tej specyfikacji opisuje zależność:

$$(\forall s \in Stan) (pre\text{-}OP(s) \Rightarrow post\text{-}OP(s, OP(s)))$$

Oznacza to, że dla każdego stanu spełniającego warunek wejściowy, wykonanie operacji *OP* na tym stanie kończy się i dla tego stanu wejściowego oraz otrzymanego (wskutek wykonania operacji *OP*) stanu wyjściowego spełniony jest warunek wyjściowy. Należy zwrócić na wynikające z tej formuły żądanie, aby wykonanie operacji *OP* kończyło się dla dowolnego stanu spełniającego warunek wejściowy.

Klasy obiektów definiowane (tak jak stany, p.2.2) za pomocą reguły ze znakiem :: są w pewnym sensie analogiczne do struktur języka PL/I lub rekordów Pascala [13]. Operacje, będące tylko pojedynczymi funkcjami można również specyfikować, używając przedrostka *FN-* .

Dokonując podsumowania rozważań przedstawionych w niniejszym rozdziale, należy stwierdzić:

- programy wykorzystują i zmieniają stany,
- klasy obiektów, które mają być użyte jako stany, definiuje się za pomocą reguł z symbolem :: . Na przykład:
  Sfact :: N : Int , KN : Int , C : Int
  definiuje klasę obiektów
  Sfact = { < *n, kn, c* > | *n,kn,c* $\in$ *Int* } ;
- warunkiem wejściowym dla operacji nazywany jest predykat dotyczący wszelkich danych, od których może zależeć wykonanie tej operacji (np. stan początkowy i argumenty),
- warunkiem wyjściowym nazywany jest predykat dotyczący stanu wejściowego, argumentów, stanu wyjściowego i wyników.

- ażeby nie dopuścić do operacji na błędnych danych, należy w specyfikacji uwzględnić odpowiednie rozszerzenie warunku wejściowego (np. reakcja w postaci komunikatu o błędzie).

Specyfikacje programów za pomocą predykatów są nie tylko krótsze od samych programów, lecz także opisują wyłącznie te właściwości, które są potrzebne użytkownikowi. Języki programowania, pomimo iż są językami formalnymi, nie są dogodne do specyfikowania programów. Najistotniejszą przeszkodą w użyciu języków programowania do specyfikacji jest niemożność wyłożenia w nich tego, co chce się osiągnąć. Języki programowania przeznaczone są bowiem do definiowania sekwencji operacji.

3. Specyfikacja typów danych

W rozdziale zestawiono podstawowe konstrukcje abstrakcyjnych typów danych stosowanych w specyfikacji programów komputerowych. Są nimi: tablice, zbiory, rekordy i klasy, listy, odwzorowania, drzewa i lasy.

3.1. Tablice

Tablicą najogólniej można nazwać uporządkowane zestawienie elementów jakiegoś typu *T*. Tablica scharakteryzowana jest poprzez zbiór wskaźników (indeksów). Jeżeli zbiór ten jest 1-elementowy, mówi się, że tablica jest 1-wymiarowa (tzw. wektor); jeżeli zbiór ten jest *n*-elementowy - mówi się że tablica jest *n*-wymiarowa. Rozmiary tablic określane są poprzez zakresy ich dolnych i górnych indeksów. Wskaźniki umożliwiają *dostęp* do poszczególnych elementów tablicy.

Tablicowy typ danych (z uwagi na jego podstawowe znaczenie) jest najlepiej opisany w podręcznikach informatyki. Typ ten jest najpowszechniej stosowany przy realizacji oprogramowania komputerów.

W specyfikacjach programów komputerowych, w przypadku użycia typu tablicowego, niekiedy do nazwy dodawany jest przyrostek *-array* .

3.2. Zbiory

Zbiorem nazywane jest nieuporządkowane zestawienie elementów. Jedyną cechą elementu jest przynależność do zbioru. Konkretny zbiór można zdefiniować, wymieniając jawnie wszystkie jego elementy. Wykaz tych elementów ogranicza się nawiasami klamrowymi { } , na przykład: $\{ x_1 , x_2 , x_3 , \ldots \}$.

Należy pamiętać, że jeżeli elementy wymienione będą w innej kolejności, bądź niektóre z nich wymienione wielokrotnie - to zawsze otrzymany będzie ten sam zbiór.

Zbiór nie zawierający żadnego elementu nazywany jest *zbiorem pustym* i zapisywany { }

Zbiory można określać niejawnie, przez podanie predykatu stanowiącego

kryterium doboru elementów do zbioru. Następujący zapis $\{ f(x) \mid p(x) \}$ oznacza "zbiór wartości $f(x)$ dla wszystkich x spełniających $p(x)$ ". Zmienna x po prawej stronie pionowej kreski jest zmienną *związaną*. Po lewej stronie kreski można wyspecyfikować funkcję, którą należy zastosować do każdej wartości x spełniającej p .

Dla podzbiorów zbioru liczb całkowitych przyjmowany jest następujący skrótowy zapis: $\{ i : j \} = \{ x \mid i \le x \le j \}$,gdzie i, j są liczbami całkowitymi.

Zasadniczą operacją dotyczącą zbiorów jest sprawdzenie, czy dany element należy do zbioru.

$x \in S \Leftrightarrow$ TRUE wtedy i tylko wtedy, gdy x jest elementem zbioru S .

Stosowane jest także rozszerzenie tego zapisu:

$x, y \in S \Leftrightarrow (x \in S \wedge y \in S)$

Za pomocą operacji *jest elementem* można określić trzy operatory dające w wyniku zbiory.

Suma zbiorów: $S_1 \cup S_2 \triangleq \{ x \mid x \in S_1 \vee x \in S_2 \}$

Iloczyn (przecięcie zbiorów):

$$S_1 \cap S_2 \triangleq \{ x \mid x \in S_1 \wedge x \in S_2 \}$$

Różnica zbiorów:

$$S_1 - S_2 \triangleq \{ x \mid x \in S_1 \wedge \neg (x \in S_2) \}$$

Można także utworzyć zbiory złożone ze zbiorów (rodziny zbiorów). Czasem zachodzi potrzeba znalezienia sumy (ang. union) wszystkich zbiorów składowych rodziny. Służy do tego operacja *sumowania rodziny zbiorów*.

$$\underline{\text{union}}\ SS \triangleq \{ x \mid (\exists\ S \in SS)(x \in S) \},$$

gdzie SS jest tutaj zbiorem złożonym ze zbiorów. Np.:

$$\underline{\text{union}}\ \{ \{ e_1, e_2, e_3 \}, \{e_3, e_4\} \} = \{ e_1, e_2, e_3, e_4 \}$$

Następujące dwa operatory służą do sprawdzania, czy jeden zbiór jest podzbiorem bądź podzbiorem właściwym drugiego:

$S_1 \subseteq S_2 \triangleq x \in S \Rightarrow x \in S_2$

$S_1 \subset S_2 \triangleq S_1 \subseteq S_2 \wedge (\exists\ x \in S_2)(\neg (x \in S_1))$

Wynik tych dwu operatorów jest zawsze wartością logiczną.

Dwa zbiory uważane są za identyczne (równe), jeżeli zawierają one dokładnie te same elementy

$S_1 = S_2 \triangleq x \in S_1 \Leftrightarrow x \in S_2$

Czterem charakterystycznym zbiorom nadano specjalne nazwy używane w specyfikacjach [7],[13] :

$Bool = \{ \text{TRUE}, \text{FALSE} \}$

$Nat = \{ 1, 2, \ldots \}$

$Nat0 = \{ 0, 1, 2, \ldots \}$

$Int = \{ \ldots, -1, 0, 1, \ldots \}$

Ponadto dla liczb rzeczywistych stosuje się nazwę *Real*.

Dla zbiorów skończonych określony został jeszcze operator liczebności (ang. cardinality - card), dający w wyniku liczbę elementów zbioru - oznaczany jako $\underline{\text{card}}\ S$.

Pomiędzy elementami różnych zbiorów mogą zachodzić relacje. W specyfikacjach najczęściej rozważane są *relacje dwuargementowe*, tj. relacje między elementami dwóch zbiorów. Zapis $R(e_1,e_2)$ oznacza, że dwa elementy e_1 i e_2 są ze sobą w relacji R. Relacja dwuargumentowa może charakteryzować się własnościami: zwrotności, symetrii i przechodniości.
Relacja nazywana jest *zwrotną*, jeżeli każdy element jest w tej relacji z samym sobą $R(i,i)$, gdzie $i \in Int$. Relacja $\leq$ jest zwrotna, zaś relacja $<$ nie jest.
Relacja nazywana jest *symetryczną*, jeżeli dla każdej pary elementów będącej w tej relacji, para tych samych elementów w odwrotnej kolejności także jest w tej relacji

$$R(i,j) \Leftrightarrow R(j,i), \quad \text{gdzie} \quad i,j \in Int$$

Np. relacja $\neq$ jest symetryczna, zaś $\leq$ nie jest symetryczna.
Relacja nazywana jest *przechodnią*, jeżeli zachodzi ona dla każdych dwóch takich elementów, dla których istnieje trzeci pośredni element taki, że pary elementów: pierwszy i pośredni oraz pośredni i drugi są w tej relacji

$$R(i,j) \wedge R(j,k) \Leftrightarrow R(i,k), \quad \text{gdzie} \quad i,\ j,\ k \in Int$$

Relacja $<$ jest przechodnia, zaś relacja $\neq$ nie jest.

Każda relacja spełniająca wszystkie trzy powyższe wymogi nazywana jest relacją *równoważności* .

W specyfikacjach programów komputerowych, w przypadku użycia typu zbiorowego, do nazwy dodawany jest przyrostek *-set* .

3.3. Rekordy i klasy

Rekordem nazywany jest następujący typ danych:

$$Rek\text{-}nazwa :: \quad z_1 : A_1 \ , \ z_2 : A_2 \ , \ \ldots \ , \ z_n : A_n$$

gdzie: *Rek-nazwa* - nazwa danego typu rekordowego,
$z_1,\ldots,z_n$ - atrybuty (pola) rekordu,
$A_1,\ldots,A_2$ - typy elementarne lub inne typy składowe.

(Pojęcie rekordu koresponduje z matematycznym pojęciem produktu kartezjańskiego.)

Jedną z podstawowych konstrukcji związanych z pojęciem rekordu - jest specjalna funkcja kreująca (ang. make). W specyfikacjach operację kreowania (generacji) rekordu oznacza się jako mk (np. mk-*Rek-nazwa*). Operacja mk ma swój odpowiednik (new) w takich językach, jak PASCAL, SIMULA-67 czy LOGLAN.

Rozszerzeniem pojęcia rekordu jest pojęcie *klasy* (stosowane w językach SIMULA i LOGLAN). Klasą nazywany jest rekord wzbogacony o wykaz instrukcji (operacji) do wykonania. Klasę można wyspecyfikować następująco:

$$Klasa\text{-}nazwa :: \quad z_1 : A_1 \ , \ z_2 : A_2 \ , \ \ldots \ , \ z_n : A_n \ ;$$
$$OP_1 \ ; \ OP_2 \ ; \ \ldots \ ; \ OP_n \ ;$$

gdzie: $OP_1 \ldots OP_n$ - lista operacji danej klasy.

3.4. Listy

Listą nazywany jest dowolnie długi ciąg elementów, z których każdy charakteryzuje się kolejnym numerem (pozycją na liście). Listy stanowią, podobnie jak zbiory, pojęcie abstrakcyjne, gdyż w pełnym ujęciu zrealizowane są tylko w niewielu językach programowania. Listy można przedstawiać na wiele różnych sposobów. Najprostszym modelem listy jest tablica (stałego rozmiaru) i wskaźnik końca listy, najbardziej skomplikowanym - struktura listowa łączona wskaźnikami. W [4] nazywa się listy *n-tkami uporządkowanymi* albo ciągami.

Listy można określać jawnie wymieniając wszystkie ich elementy. Wykaz elementów listy ograniczany jest nawiasami kątowymi < >,

np. $< e_1, e_2, \ldots, e_n >$

Ponieważ z listy można wybierać poszczególne elementy na podstawie ich numerów kolejnych, porządek elementów w liście jest istotny (inaczej niż w przypadku zbiorów).

Lista nie zawierająca żadnego elementu nazywana jest *listą pustą* i oznaczana < > . Do rozkładu listy na części można wykorzystać m.in. operatory hd (ang. head - głowa) i tl (ang. tail - ogon). Operator hd daje w wyniku pierwszy element listy; operator tl daje w wyniku listę, jaka pozostaje z oryginalnej listy po usunięciu pierwszego elementu. Na typy elementów list nie nakłada się żadnych ograniczeń; mogą więc istnieć listy list, listy zbiorów itd. Operatory hd i tl mają ograniczoną dziedzinę - można je stosować tylko do list niepustych. Operator len (ang. length - długość) daje w wyniku liczbę elementów listy. Operator ten można zdefiniować następująco: len *list* ≙ if *list* = < > then 0 else 1 + len(tl *list*)

Wartość indeksu elementu musi zawierać się między 1 a długością listy. Operację wyboru elementu z listy można zdefiniować następująco:

list(*i*) ≙ if *i*=1 then hd *list* else (tl *list*) (*i* - 1)

Dla list można również określić operator *konkatenacji* (składania) o dwóch argumentach będących listami. Jego wynikiem jest lista złożona z wszystkich elementów pierwszego argumentu, a następnie z wszystkich elementów drugiego argumentu. Operator ten oznaczany jest symbolem || . Do konkatenacji większej liczby list służy operator conc (ang. concatenation), w pewnym sensie analogiczny do operacji sumowania rodziny zbiorów. Konkatenacja list jest operacją łączną.

Ostatnim operatorem dotyczącym list, jaki należy wymienić, jest operator tworzący zbiór wszystkich elementów listy, zdefiniowany w następujący sposób:

elems *list* ≙ { *list*(*i*) | 1 ≤ *i* ≤ len *list* }
(elems - elements - elementy), np. elems {1,2,1,4,3} = {1,2,3,4} .

Stosując listę w specyfikacjach do nazwy typu dołączany jest przyrostek *-list*. Jeżeli *X* jest daną klasą obiektów, to wyrażenie *L* = *X-list* oznacza, że każdy obiekt typu *L* jest listą o elementach typu *X*. Klasę list niepustych (tj. zwierających przynajmniej jeden element) oznacza się w następujący sposób: *L1* = *X-list1*
tak więc *l* ∈ *L1* ⇔ len *l* ≥ 1 ∧ (∀ *i* ∈ {1:len *l*}) (*l*(*i*) ∈ *X*)

Przy pomocy list można opisać stosy i kolejki. Przez *stos* (ang. stack) rozumie się typ danych pozwalający zapamiętywać i odczytywać wartości w taki sposób, że pobiera się zawsze wartość ostatnio zapamiętaną. Stosy spotyka się w bardzo różnych sytuacjach (np. przy obliczaniu wyrażeń). Najczęstszym ich zastosowaniem jest realizacja procedur rekurencyjnych za pomocą instrukcji iteracyjnej. *Kolejka* jest typem podobnym do stosu. Różnica polega na tym, że pierwszym elementem wyjmowanym z kolejki jest zawsze pierwszy element wprowadzony do niej.

3.5. Odwzorowania

Odwzorowanie jest specjalnym rodzajem funkcji. Pojęcie odwzorowania jest jednak bardziej ograniczone, niż funkcji. Dziedzina odwzorowania musi być bowiem skończona. Ponadto relacji między argumentami a wynikami nie ustala się za pomocą wyrażenia, lecz konstruuje się ją. Wskutek tych wymogów - można wykonywać na odwzorowaniach pewne operacje, których nie można określić dla funkcji.

Do notacji odwzorowań stosuje się nawiasy klamrowe. Odwzorowanie można zadać, wymieniając jawnie zestaw wszystkich par argument-wartość

[*d1* --> *r1*, *d2* --> *r2*, ... , *dn* --> *rn*] .

Należy pamiętać, że wszystkie *di* muszą być różne, gdyż odwzorowanie może mieć tylko jedną wartość dla danego argumentu. Kolejność zapisania poszczególnych par jest nieistotna. Wewnątrz danej pary wartość należącą do dziedziny zapisuje się po lewej stronie strzałki. Szczególny przypadek odwzorowania pustego zapisuje się [ ] .

Podobnie jak zbiory, również i odwzorowania można definiować niejawnie, pisząc [*x* --> *f*(*x*) | *p*(*x*)] . Na przykład:
[*x* --> *x* ** 2 | *x* ∈ { 1:4 }] = [1-->1, 2-->4, 3-->9, 4-->16]
W liście par składających się na odwzorowanie tylko elementy dziedziny muszą być różne.

Do znajdowania dziedziny odwzorowania, tj. zbioru elementów, dla których określona jest wartość odwzorowania, służy operator dom (and. domain - dziedzina).
Jeżeli *M* = [*x* --> *x* ** 2 | 1 ≤ *x* ≤ 3] to dom *M* = { 1, 2, 3 }

Odwzorowanie można zastosować do dowolnej wartości z dziedziny. Daje ono wówczas wynik - jest to drugi element z tej pary, w której pierwszym

elementem był dany argument. Operację zastosowania odwzorowania do argumentu można też traktować jako znajdowanie wartości w tabeli. Wynik odwzorowania dla wartości spoza dziedziny nie jest określony. Zbiór wartości odwzorowania otrzymuje się za pomocą operatora rng (ang. range - zakres wartości)

$$\underline{rng}\ map \triangleq \{\ map(d)\ |\ d \in \underline{dom}\ map\ \}$$

gdzie: *map* - mapping - odwzorowanie.

Odwzorowanie można także skonstruować z dwóch innych odwzorowań za pomocą operatora scalania (overwrite) oznaczonego znakiem . Jeżeli dziedziny mają przy tym wspólne, to drugi argument operatora ma priorytet względem pierwszego.

$$M1\quad M2 \triangleq [\ d \dashrightarrow r\ |\ d \in \underline{dom}\ M2 \wedge r = M2(d)\ \vee$$
$$d \in (\ \underline{dom}\ M1 - \underline{dom}\ M2\ ) \wedge\ r = M1(d)\ ]$$

Na przykład, jeżeli:

$m1 = [A \dashrightarrow 1\ ,\ B \dashrightarrow 2]$ i $m2 = [C \dashrightarrow 1]$ i $m3 = [A \dashrightarrow 2,\ D \dashrightarrow 0]$

to $m1\quad m2 = [A \dashrightarrow 1,\ B \dashrightarrow 2,\ C \dashrightarrow 1]$

$m1\quad m3 = [A \dashrightarrow 2,\ B \dashrightarrow 2,\ D \dashrightarrow 0]$

Odwzorowanie można *ograniczyć* do danego zbioru - dziedzina tak powstałego odwzorowania będzie przecięciem danego zbioru i poprzedniej dziedziny

$$M \mid S \triangleq [\ d \dashrightarrow M(d)\ |\ d \in (\ \underline{dom}\ M \cap S\ )\ ]$$

Zapis $M = D \dashrightarrow R$ definiuje klasę odwzorowań takich, że

$$(\ \forall\ m \in M\ )\ (\ \underline{dom}\ m \subseteq D \wedge \underline{rng}\ m \subseteq R\ )$$

3.6. Drzewa, lasy oraz grafy

Odwzorowania, których dziedzina pokrywa się z zakresem wartości, mogą służyć do reprezentowania wielu różnych struktur. Jeżeli odwzorowanie nie zawiera *cykli* (tj. ciągów elementów połączonych w zamknięte pętle), to za jego pomocą można reprezentować drzewa. Każdy element będzie odwzorowany na sąsiadujący element bliższy korzenia drzewa. Korzenie wyróżniają się tym, że są odwzorowane same na siebie. Ponieważ odwzorowanie może różnym elementom przypisać tę samą wartość (lecz nie na odwrót), gałęzie drzewa mogą łączyć się, lecz nie rozchodzić. Formalnie

$$F = Element \dashrightarrow Element \quad \text{oraz}$$
$$is\text{-}root:\ Element,\ F \dashrightarrow Bool$$
$$is\text{-}root\ (e,f) \triangleq f(e) = e$$

gdzie: *F* - forest - las (zbiór drzew) ; *is-root* - jest korzeniem.

W konwencji języka Pascal można napisać następującą definicję typu:

```
TYPE  node = ^element ;
      element = RECORD  a: A; up: node  END;
```

Drzewem jednowskaźnikowym nazywa się skończony zbiór obiektów typu *element* oraz zmienną typu *node*, o nazwie (np.) *root*, takie że dla każdego elementu

$q^\wedge$ tego zbioru następujące instrukcje mają skończone obliczenie:

$p := q$; WHILE $p \neq root$ DO $p := p^\wedge.up$;

Elementy drzewa nazywa się *wierzchołkami* [3]. Element wskazany przez wartość zmiennej *root* - *korzeniem drzewa*. Dla każdego wierzchołka drzewa $q^\wedge$, zbiór wierzchołków dla których wskaźnik *up* wskazuje na $q^\wedge$, nazywa się *bezpośrednimi następnikami* tego wierzchołka. Wierzchołki, które nie mają bezpośrednich następników nazywają się *liśćmi* drzewa. Wierzchołek $q^\wedge$ jest *następnikiem* wierzchołka $r^\wedge$, jeżeli instrukcje: $p := q$; WHILE $p \neq r$ DO $p := p^\wedge.up$; mają skończone obliczenie.

Zbiór następników wierzchołka $r^\wedge$ nazywa się *poddrzewem* o korzeniu $r^\wedge$, uporządkowana para wierzchołków - *krawędzią*, jeżeli pierwszy wierzchołek jest bezpośrednim następnikiem drugiego. *Ścieżką* w drzewie nazywa się ciąg $a_1,\ldots,a_k$ wierzchołków taki, że dla każdego $i = 2,\ldots,k$ para (a_{i-1},a_i) jest krawędzią w drzewie. (Dalsze definicje dotyczące drzew podaje m.in. lit. [3]).

Innym typem danych, które można przedstawić w tej grupie są *grafy zorientowane*. Grafem zorientowanym nazywa się parę (U,R), w której $U \in Nat$, R natomiast stanowi relację dwuargumentową $R \subseteq U \times U$. Elementy zbioru U nazywają się *wierzchołkami grafu*, elementy zbioru R - *krawędziami grafu*.

Jedną z metod reprezentowania grafu w programie jest, tzw. *lista incydencji*. Mianowicie dla każdego wierzchołka $i \in \langle 1:n \rangle$ buduje się jednokierunkową listę wierzchołków j incydentnych z i, to znaczy takich, że para (i,j) należy do R [3] .

3.7. Niezmienniki typów danych

Niezmiennikiem typu danych nazywany jest predykat zachodzący dla wszystkich obiektów tego typu, które można utworzyć za pomocą operacji dotyczących tego typu. Obiekty spełniające niezmiennik typu danych nazywane są *poprawnymi* [13].

Formalnie własność *inv* (ang. invariant) jest *niezmiennikiem*, jeśli spełnia następującą regułę:

$pre\text{-}OP(q,args) \wedge inv(q) \wedge post\text{-}OP(q,args,q',res) \Rightarrow inv(q')$

gdzie: q - stan, *args* - argumenty, *res* - wynik .

Stosowanie niezmienników znacznie ułatwia wybór właściwych modeli dla abstrakcyjnych typów danych. Wynika to z konieczności lepszego dostosowania typów danych określonych w specyfikacji do rzeczywistości. Często stosowanie niezmienników staje się wręcz niezbędne, gdyż rzeczywiste obiekty mogą być zbyt nieregularne w stosunku do czystych modeli matematycznych.

Przykładem niezmiennika może być następujący zapis:

niech *Data* określa stan, który ma zawierać daty

Data :: *ROK*: *Nat* , *MIES*: ⟨ *Sty*, *Luty*,... , *Gru* ⟩ , *DZIEŃ*: ⟨ 1:31 ⟩

Klasę poprawnych dat można scharakteryzować za pomocą następującego predykatu:

$$
\begin{aligned}
&\mathit{inv\text{-}data}\ (\ \langle\ r, m, d\ \rangle\ ) \triangleq \\
&\quad (\ m \in \{\ \mathit{Sty}, \mathit{Mar}, \mathit{Maj}, \mathit{Lip}, \mathit{Sier}, \mathit{Paz}, \mathit{Gru}\ \} \bullet 1 \le d \le 31\ ) \wedge \\
&\quad (\ m \in \{\ \mathit{Kwie}, \mathit{Czer}, \mathit{Wrze}, \mathit{Lis}\ \} \bullet 1 \le d \le 30\ ) \wedge \\
&\quad (\ m = \mathit{Luty} \wedge \neg\ \mathit{jest\text{-}przestępny}\ (r) \bullet 1 \le d \le 28\ ) \wedge \\
&\quad (\ m = \mathit{Luty} \wedge \mathit{jest\text{-}przestępny}\ (r) \bullet 1 \le d \le 29\ )
\end{aligned}
$$

3.8. Uściślanie danych i reprezentacja

W specyfikacjach używa się abstrakcyjnych typów danych. Jeżeli język programowania jest wyposażony w odpowiednie typy danych, to zadanie konstruktora jest ułatwione. Zazwyczaj jednak użyte typy danych są bardziej abstrakcyjne niż występujące w językach programowania i konieczne są dalsze kroki konstrukcji. Czynność określania realizacji dla abstrakcyjnego typu danych nazywana jest *uściślaniem typu danych*, zaś typ danych stanowiący tę realizację nazywany *reprezentacją* danego typu abstrakcyjnego. Mówi się, że operacje określone dla reprezentacji *modelują* operacje dotyczące typu abstrakcyjnego.

Przykładami reprezentowanych typów mogą być m.in. tablice, listy oraz zbiory. Tablice są w najprostszym przypadku gęste (tj. większość elementów ma wartości nietrywialne) i mają ustalone rozmiary. W czasie realizacji odwzorowuje się zazwyczaj ich elementy na jednowymiarowy wektor adresów. Macierze rzadkie (tj. duże tablice, w których np. około 5 % elementów jest róznych od zera) najlepiej uważać za odwzorowania indeksów w wartości. Listę można przedstawić jako jednowymiarową tablicę tylko wtedy, gdy znana jest jej maksymalna długość i gdy nie przewiduje się częstych operacji wstawiania elementów. Dla zbiorów istnieje znacznie większy wybór możliwości reprezentowania (np. tablice, wektory bitów, listy łączone).

Aby móc dowodzić poprawność uściślania danych, niezbędne jest zapisanie (udokumentowanie) relacji między abstrakcyjnym typem danych a jego reprezentacją. Używana do tego celu jest tzw. *funkcja odzyskiwania* [13]. Abstrakcyjne typy danych dobiera się tak, aby zawierały minimalną ilość informacji. Zawierają one wyłącznie informację niezbędną dla określenia znaczenia operacji. Elementy reprezentacji zawierają więc często więcej informacji niż elementy typu abstrakcyjnego. Te dodatkowe informacje mogą być nieistotne (jak np. w przypadku reprezentowania zbioru za pomocą listy) lub mogą służyć do lokalizacji właściwych informacji w reprezentacji. Funkcja odzyskiwania (ang. retrieve function) umożliwia odzyskanie z reprezentacji istotnej (abstrakcyjnej) informacji. Na przykład w przypadku list i zbiorów funkcję odzyskiwania można określić następująco:

$$
\begin{aligned}
&\mathit{retr\text{-}set} : \mathit{X\text{-}list} \longrightarrow \mathit{X\text{-}set} \\
&\mathit{retr\text{-}set}\ (\mathit{xl}) \triangleq \underline{\mathit{elems}}\ \mathit{xl}
\end{aligned}
$$

W przypadku uściślania operacji należy pamiętać, że operacje są zwykle określone przez warunki wejściowe i wyjściowe. Określenia takie dla danych argumentów dopuszczają pewien zakres wyników. Dla każdej operacji dotyczącej stanu abstrakcyjnego należy określić odpowiednią operację w stanie uściślonym. Nowa operacja *modeluje* operację abstrakcyjną, jeżeli można za jej pomocą wypełnić zadania operacji abstrakcyjnej. Operacja ta musi mieć wystarczająco dużą dziedzinę i dawać odpowiednie wyniki.

4. Przykłady różnych specyfikacji tego samego problemu

Zagadnienie to można zilustrować na przykładzie stosu. Elementy stosu można określić jako funkcje, a nie (zmieniające stan) operacje. Jedną z możliwych specyfikacji może być sformułowana za pomocą składni abstrakcyjnej:

```
Stack1 = [ Stack-level ]
Stack-level :: TOP : Element ,  REST : Stack1
NEWSTACK1 : --> Stack1
NEWSTACK1() = NIL
PUSH1 :  Stack1, Element  --> Stack1
PUSH1(st,el) ≙ mk-stack-level(el,st)
INSPECT1 :  Stack1 --> Element
INSPECT1(mk-stack-level(el,st)) ≙ el
pre-INSPECT1(st) ≙ st ≠ NIL
REMOVE1 :  Stack1 --> Stack1
REMOVE1(mk-stack-level(el,st)) ≙ st
pre-REMOVE1(st) ≙ st ≠ NIL
IS-EMPTY1 : Stack --> Bool
IS-EMPTY1(st) ≙ st = NIL
```

Oznaczenia: stack-level - poziom stosu, TOP - wierzchołek, REST - reszta, NEWSTACK - nowy stos, PUSH - włóż na stos, INSPECT - zbadaj, REMOVE - usuń, IS-EMPTY - jest puste.

Inną specyfikację tego samego problemu można sformułować za pomocą list. Pomijając opisy typu funkcji i warunki wejściowe można napisać:

```
Stack = Element-list
NEWSTACK() = < >
PUSH(st,el) ≙ <el> || st
INSPECT(st) ≙ hd st
REMOVE (st) ≙ tl st
IS-EMPTY(st) ≙ st = < >
```

Stos można też wyspecyfikować, stosując pojęcie stanu:

```
Stack2 ::  ST : Element-list ,  LICZNIK :  Nat

NEWSTACK2() ≙ mk-stack2(< >,0)
PUSH2(mk-stack2(st,licz),el)  ≙
          mk-stack2( if len st = licz then st || <el>
                     else modl (st, licz+1, el)), licz+1)
INSPECT2(mk-stack2(st,licz)) ≙ st(licz)
REMOVE2(mk-stack2(st,licz))  ≙ mk-stack2(st,licz-1)
IS-EMPTY(mk-stack2(st,licz)) ≙ licz = 0
```

Należy zauważyć, że w operacji *REMOVE2* nie zmieniana jest zawartość

listy, lecz zmiejszana jedynie wartość licznika.
Funkcja *modl* jest zdefiniowana następująco:

modl (*list*, *i*, *e*) ≙ <u>if</u> *i*=1 <u>then</u> <*e*> || <u>tl</u> *list*
<u>else</u> <<u>hd</u> *list*> || *modl* (<u>tl</u> *list*, *i*-1, *e*)

5. Uwagi końcowe

Prace nad realizacją oprogramowania komputerów według [7] można podzielić na następujące fazy:

- analiza (identyfikacja) problemu,
- specyfikacja,
- projektowanie na wysokim szczeblu abstrakcji,
- projektowanie na niskim szczeblu abstrakcji,
- kodowanie (w wybranym języku programowania),
- testowanie modułów,
- integracja modułów,
- test akceptacji całości i przekazanie do użytkowania.

W ciągu ostatnich kilku lat wzrosła znacznie liczba prac poświęconych zagadnieniu specyfikacji formalnej programów komputerowych. Zagadnienie to jest szczególnie ważne w przypadku projektowania dużych systemów, realizowanych przez kilkudziesięcioosobowe zespoły programistów. Błędy popełnione we wczesnym etapie specyfikowania nie są jeszcze zbyt kosztowe - i dzięki stosowaniu dowodów formalnych są stosunkowo łatwe do wykrycia. Ogólnie w specyfikacji formalnej można wyróżnić dwa zasadnicze podejścia: specyfikację algebraiczną - prezentowaną m.in. w pracach [7], [8], [9] oraz modelowo-zorientowaną, tzw. VDM (Vienna Development Method) - prezentowaną m.in. w [4],[5],[7],[12],[13].

Oprócz zagadnień omówionych w artykule należy jeszcze wyróżnić: dodatkowe elementy specyfikacji, zagadnienia dowodzenia poprawności oraz złożoność obliczeniową algorytmów.

5.1. Dodatkowe elementy specyfikacji

Dodatkowymi elementami są przede wszystkim operacje związane z komunikacją danego modułu programowego z otoczeniem (tj. wejście, wyjście, współpraca i powiązania międzymodułowe). Używane są tutaj różne słowa kluczowe, np. INPUT, OUTPUT lub operatory <u>rd</u> (czytaj) i <u>wr</u> (czytaj lub pisz) - w przypadku operacji wejścia- wyjścia; EXT, USES - w przypadku określania modułów współpracujących; UNIT, MODULE - w przypadku deklarowania nagłówków modułów; IMPORT, EXPORT, INTERFACE - w przypadku komunikacji i określania powiązań międzymodułowych.

Zestaw słów kluczowych specyfikacji formalnej oraz konwencję ich

stosowania ustala główny projektant oprogramowania.

5.2. Dowodzenie poprawności programów

Ponieważ języki programowania są językami formalnymi, dlatego dla zrealizowanych programów komputerowych mogą być przeprowadzone formalne dowody ich poprawności.

Mówi się, że algorytm K jest *semantycznie poprawny* względem warunków początkowego α i końcowego β, jeśli dla każdych danych wejściowych spełniających warunek α obliczenie algorytmu K dochodzi do punktu końcowego oraz końcowe wartościowanie zmiennych spełnia warunek β. Na równi z powyższym określeniem używane są równoznaczne terminy *całkowicie poprawny algorytm* oraz *algorytm zgodny ze specyfikacją* [3].

Niepoprawność algorytmu może być trojakiego rodzaju, a mianowicie dla pewnych danych wejściowych obliczenie algorytmu albo dochodzi do punktu końcowego, ale wyniki nie spełniają warunku końcowego, albo zatrzymuje się w punkcie niekońcowym tego algorytmu albo też jest nieskończone. W związku z tym poprawności algorytmu K względem warunków początkowego α i końcowego β dowodzi się zwykle przez pokazanie, że algorytm K ma następujące trzy własności:

1) dla każdych danych wejściowych spełniających warunek początkowy α, jeżeli obliczenie algorytmu K dochodzi do punktu końcowego, to otrzymane wyniki spełniają warunek końcowy β;
2) dla każdych danych wejściowych spełniających warunek początkowy α obliczenie algorytmu K nie jest przerwane;
3) dla każdych danych wejściowych spełniających warunek początkowy α obliczenie algorytmu K nie jest nieskończone.

Jeżeli dla algorytmu K zachodzi warunek 1), to mówi się, że algorytm K jest *częściowo poprawny* względem warunków początkowego α i końcowego β. Jeżeli zachodzi warunek 2), to mówi się, że algorytm K ma *własność określoności obliczeń* względem warunku początkowego α. W przypadku gdy jest spełniony warunek 3) mówi się, że algorytm K ma *własność stopu* względem warunku początkowego α [3]. Podstawowym środkiem do przeprowadzenia tych dowodów jest metoda nazywana *metodą niezmienników*, która polega na opisywaniu własności wartościowań zmiennych otrzymywanych, gdy obliczenie algorytmu przechodzi przez wyróżnione punkty tego algorytmu. Dla dowodzenia własności stopu algorytmów stosowane są: *metoda liczników iteracji* i *metoda malejących wielkości*.

Poprawności algorytmów rekurencyjnych dowodzi się zazwyczaj stosując indukcję względem wielkości danych wejściowych, dla których algorytmy są realizowane. Nie zachodzi potrzeba dzielenia całego procesu dowodzenia poprawności algorytmu na odrębne etapy weryfikacji własności częściowej poprawności, określoności obliczeń i stopu.

5.3. Złożoność obliczeniowa algorytmów

Każde wykonanie algorytmu na komputerze wymaga pewnej ilości czasu pracy, jak również pewnej ilości miejsca pamięci. Okazuje się, że dla dużej liczby znanych algorytmów, czas ich działania zwiększa się zbyt szybko w miarę jak wzrasta wielkość danych wejściowych. Dokonana analiza pokazuje, że znajomość czasu działania algorytmu jest niezbędna, aby odpowiedzieć na pytanie, czy wymagany czas działania umożliwia stosowanie algorytmu na dostępnym sprzęcie.

W celu oszacowania czasu działania algorytmu zwykle szacuje się liczbę wszystkich jednostkowych operacji wykonywanych w trakcie realizacji algorytmu. Przykładowo, dla języka Pascal następujące operacje można traktować jako jednostkowe:

- wykonanie operatora arytmetycznego, relacyjnego lub logicznego;
- nadanie wartości zmiennej typu prostego lub wskaźnikowego;
- obliczenie wartości zmiennej indeksowanej, wskazywanej lub atrybutu rekordu;
- inicjalizacja wywołania procedury lub funkcji;
- przekazanie wartości parametru aktualnego;
- wykonanie instrukcji pustej, skoku, wejścia lub wyjścia.

Niech K jest algorytmem, dla którego zbiorem danych wejściowych jest zbiór D taki, że dla danej d ze zbioru D obliczenie algorytmu K dochodzi do punktu końcowego; niech $t(d)$ oznacza liczbę jednostkowych operacji wykonywanych przez algorytm K dla danej wejściowej d ze zbioru D. Odwzorowanie t jest funkcją ze zbioru danych D w zbiór liczb naturalnych. Funkcję t nazywa się *pełną funkcją kosztu* algorytmu K [3].

Dla każdego algorytmicznego problemu określa się *rozmiar danych tego problemu*. Wzrost pełnej funkcji kosztu algorytmów, rozwiązujących ustalony problem, mierzy się względem wzrostu rozmiaru danych. Oprócz terminu funkcja kosztu algorytmu K są używane również następujące określenia: *funkcja kosztu niepomyślnego przypadku*, *funkcja złożoności czasowej*, *złożoność czasowa*, *pesymistyczna zło- żoność czasowa*.

Przy analizie złożoności algorytmów dogodnie jest rozważać tylko pewne wyróżnione operacje jednostkowe, nazywane *operacjami dominującymi* algorytmu, o tej własności, że liczba tych wybranych operacji wykonywanych przez algorytm, traktowana jest jako funkcja danych wejściowych, jest dokładnie rzędu pełnej funkcji kosztu.

Oprócz złożoności czasowej algorytmów rozważa się również ich *złożoność pamięciową* [3].

LITERATURA

[1] ABRIAL J.R.: The Specification Language Z: Basic Library - *Oxford University Programming Research Group, Specification Group Working Paper, 1980* .
[2] AHO A.V., HOPCROFT J.E., ULLMAN J.D.: Projektowanie i analiza algorytmów komputerowych. *PWN, Warszawa 1983* .
[3] BANACHOWSKI L., KRECZMAR A.: Elementy analizy algorytmów. *WNT, Warszawa 1982* .
[4] BJORNER D., JONES C.B.(eds.): The Vienna Development Method: The Meta-Language - *Springer Verlag Lecture Notes in Computer Science*, No. 61, 1978 .
[5] BJORNER D., JONES C.B.: Formal Specification and Software Development - *Prentice-Hall, 1982* .
[6] BRADY J.M.: Informatyka teoretyczna w ujęciu programistycznym. *WNT, Warszawa 1983* .
[7] COHEN B., HARWOOD W.T., JACKSON M.I.: The Specification of Complex Systems - *Addison-Wesley Publishing Company, 1986*.
[8] GEHANI N.: Specifications: Formal and Informal - A Case Study - artykuł z tomu *"Software Specification Techniques"* - *Addison-Wesley Publishing Company, 1986*
[9] GUTTAG J., HORNING J.J.: Formal Specification as a Design Tool - artykuł z tomu *"Software Specification Techniques"* - *Addison-Wesley Publishing Company, 1986*.
[10] HEWITT C., BAKER H.: Actors and Continuous Functionals - *artykuł z tomu: "Formal Descriptions of Programming Concepts"* - *North-Holland Publishing Company , 1978* .
[11] HIPO - A Design Aid and Documentation Technique. *Form GC20-1851 IBM*.
[12] JONES C.B.: Specyfikacje a programy - *Informatyka nr 10 i 11-12 / 1987* .
[13] JONES C.B.: Konstruowanie oprogramowania metodą systematyczną. WNT, Warszawa 1984.
[14] KONIECZNY R.: O pewnych konwencjach notacji algorytmów programów komputerowych - *Zeszyty Naukowe Politechniki Śląskiej , Transport nr 6 / 1988* .
[15] KONIECZNY R., KRAWIEC S.: Wybrane zagadnienia opisu formalnego modelu systemu złożonego - *(w niniejszym zeszycie)* .
[16] TURSKI W.M.: Metodologia programowania - *WNT Warszawa 1982* .
[17] ZEIGLER B.P.: Teoria modelowania i symulacji - *PWN Warszawa 1984* .

FORMAL SPECIFICATION IN PROGRAMS DESIGN

Summary

The present paper refers to the problems of formulating computer program specifications. This problem becomes especially important in the case of complex systems design. A number of theoretical works testify to the importance of these problems.

The paper is an attempt of arranging the most important notions, definitions and structures making formal basis for creating the computer program specifications. Functions and operations specyfing, the notion of state and specyfing data types can be numbered among the problems connected with it.

Some examples of various specifications of the same problem have been

given.

During the last several years the number of works devoted to the problem of the computer program formal specification has considerably increased.

This problem is especially important in the case of projecting large systems realized by the groups of the tens of programing specialists.

FORMALE SPEZIFIKATION IN DER PROJEKTIERUNG VON RECHNERSPROGRAMMEN

Zusammenfassung

Der vorliegende Aufsatz befaßt sich mit den Problemen der Formulierung der Spezifikationen von Rechnerprogramen. Diese Problematik wird bei der Projektierung von Komplizierten Systemen besonders wichtig. Von der Wichtigkeit dieser Problematik zeugt eine grose Anzahl theoretischer Arbeiten.

Vorliegender Aufsatz stellt eine Probe des Ordnungsbringens in die wichtigsten Begriffe, Definitionen und Konstruktionen dar, die formale Grundlagen zur Schaffung der Spezifikationen von Rechnerprogrammen bilden. Zu diesen Problemen kann man zählen: Funktions- und Operationsspezifizierung, Zustandsbegriff sowie Spezifizierung von Datentypen. Es wurden Beispiele verschiedener Spezifikationen des gleichen Problem angegeben.

Im Laufe letzter paar Jahren wuchs bedeutend die Anzahl von Arbeiten, die dem Problem der formalen Spezifikation von Rechnerprogrammen gewidmed sind. Diese Problematik ist bei der Projektierung von großen Systemen, die durch Programmiererkollektive mit einigen zig Personen realisiert werden, besonders wichtig.

ФОРМАЛЬНАЯ СПЕЦИФИКАЦИЯ В ПРОЕКТИРОВАНИИ КОМПЬЮТЕРНЫХ ПРОГРАММ

Резюме

Статья касается формулировки спецификаций компьютерных программ. Вопрос этот становится особенно важным в случае проектирования сложных систем.

В статье дана попытка упорядочненя важнейших понятий, определений и конструкций, являющиеся формальной основой получения спецификаций компьютерных программ. К этим вопросам можно зачесть: спецификация функций, спецификация операций, понятие состаяния а также спецификация типов данных. Даны примеры пазличных спецификаций одной и тойже проблемы.