INFORMÁTICA 83

XVI CONGRESSO NACIONAL DE INFORMÁTICA

SÃO PAULO OUTUBRO 83

UMA COMPARAÇÃO ENTRE METODOLOGIAS DE DESENVOLVIMENTO DE PROGRAMAS

Raul Cesar Baptista Martins/Paulo Augusto Silva Veloso/Carlos José Pereira de Lucena

Rua Marquês de São Vicente, 225 - Departamento de Informática -PUC/RJ
CEP: 22453 - Rio de Janeiro - RJ
Tel.: 274.9922 r/385/386 - 274.4449

Palavras chaves: Teoria da programação, engenharia de programas, método dos transformadores de predicados, método de transformação de programas, tipo abstrato de dados, método de Jackson, método de Constantine-Yourdon.

Resumo:

São apresentados os seguintes métodos de derivação de programas:
- método de transformação de predicados (Dijkstra)
- método de transformação de programas (Bauer)
- método das estruturas de dados (Cowan e Lucena)
- método dos tipos abstratos de dados (Pequeno e Lucena)
- método de Jackson
- método de Constantine e Yourdon

Para cada uma das metodologias é apresentado um resumo sucinto da teoria,seguido de um exemplo simples que ilustra a aplicação da metodologia.

I. Introdução

Métodos de derivação de programas são influenciados tanto pelas linguagens usadas para expressar as especificações do programa quanto pela linguagem usada para a codificação do programa.
Algumas vezes são propostas linguagens de amplo espectro que permitem tanto a especificação quanto a codificação da solução de um problema e as metodologias de derivação de programas assim estabelecidas contêm na sua concepção mecanismos capazes de suportar os formalismos matemáticos usuais em especificação de problemas.
Outras vezes, já se antevendo que a solução do problema será programada em linguagem do gênero Pascal, já se propõem metodologias de derivação nas quais as especificações dos problemas sejam escritas de forma a permitir diretamente a transformação de especificação em programa.
Outra distinção entre as várias metodologias propostas é o objetivo almejado: resolver problemas ditos acadêmicos ou resolver problemas ditos práticos. Como os problemas que são considerados interessantes na área acadêmica e que dão origem a metodologias com forte embasamento teórico não são os que ocorrem no dia a dia nos ambientes práticos temos que aplicar para a solução desses problemas, ou metodologias não muito bem caracterizadas ou então não adequadas para os problemas em estudo.
Em Lucena[1], os métodos de desenvolvimento de programas são classificados pela forma como tratam os dados de um problema: métodos nos quais os dados são implicitamente especificados e métodos nos quais os dados são explicitamente especificados.

Nos métodos que tratam os dados implicitamente as especificações de entrada e saída do problema supõem que o programa manipulará tipos de dados conhecidos e as formalizações para estes tipos de dados são introduzidos de uma forma pragmática.
Os métodos que tratam explicitamente dos dados procuram partir da especificação das estruturas dos dados envolvidos nos problemas e derivar a estrutura do programa solução.
Na primeira categoria estão incluídos o método dos transformadores de predicado, (Dijkstra[2]) e o método de transformadores de programa (Bauer[3]).
Na segunda categoria encontram-se o método da axiomática de tipo de dados (Pequeno e Lucena[4,5]), o método da estrutura de dados (Lucena e Cowan[6,7,8]), o método de Jackson[9] e Warnier[10] e o método de Constantine e Yourdon[11].

II. O Método dos Transformadores de Predicados

A partir da proposta do trabalho de Floyd[12], semântica de linguagens de programação independente dos processadores, e de Haore[13], axiomática de programação, Dijkstra[7] desenvolveu a idéia que a definição proposta tanto pode ser utilizada para a análise (verificação) quanto para a síntese de um programa (derivação).
A metodologia pode ser resumida em:
i. especificam-se os predicados que determinam o conjunto de estados desejados (estados finais)
ii. procura-se construir os predicados que caracterizam um conjunto a partir do qual com transformações consegue-se atingir o conjunto de saída. As transformações possíveis são definidas como mecanismos de controle.
iii. tem-se o conhecimento do segmento de programa que se deseja sintetizar quando qualquer que seja o conjunto de estados de saída pode-se deduzir o conjunto de estados de entrada a partir do qual se consegue alcançar o conjunto dado. Um segmento de programa passa a ser denominado um transformador de predicado capaz de transformar uma pós-condição numa pré-condição.

Formalizando-se:
i. a condição P que caracteriza o conjunto de estados iniciais que após a ativação do segmento de programa S, transformador de predicado, resulta em um estado final que satisfaz a pós-condição R é chamada a pré-condição mais fraca e se escreve P = pf(S,R)
ii. se S é um transformador de predicado e Q e R são pós-condições tem-se:
a. ∀S, pf(S, falso) = falso

b. ∀S,Q,R, Q R então pf(S,Q) → pf(S,R)

c. ∀S,Q,R; pf(S,Q) ∧ pf(S,R) → pf(S, Q ∧ R)

d. ∀S,Q,R; pf(S,Q) ∨ pf(S,R) → pf(S, Q ∨ R)
iii. são possíveis os seguintes mecanismos de controle:
a. Skip: pf(Skip,R) = R

b. Abort: pf(Abort,R) = falso

c. Atribuição: pf(x ← Expr,R) = R(expr)

d. Composição: pf(S_1;S_2,R) = pf(S_1,pf(S_2,R)
e. Alternância:

if

$B_1 \rightarrow S_1$
⋮
$B_n \rightarrow S_n$

fi

pf(if,R) = $((B_1 \vee \ldots \vee B_n) \wedge$

$(B_1 \rightarrow pf(S_1,R)) \wedge$

...

$(B_n \rightarrow pf(S_n,R)$

f. Repetição

do

$B_1 \rightarrow S_1$
⋮
$B_n \rightarrow S_n$

od

pf(do,R) = $(Ho(R) \vee \ldots \vee H_k(R) \vee \ldots)$

onde

$H_0(R) = R \; e \neg BB, \; BB = B_1 v ... v B_n$

$\forall k > 0$

$H_k(R) = pf(if, H_{k-1}(R)) v H_0(R)$

Como **primeiro exemplo** aplica-se a metodologia a um trecho de programa que deve calcular a $\sqrt{n}$ onde n é inteiro. O cálculo de $\sqrt{n}$ só deve ser feito se $n \geq 0$ caso contrário o programa deve ser cancelado.

A pós-condição pode ser definida da seguinte maneira:

R: $(n = \sqrt{n} \wedge n \geq 0) v (n_{\sqrt{\ }} = \uparrow \wedge n < 0)$

Para isto considere-se um programa P_1: $n_{\sqrt{\ }} = \sqrt{n}$ e um programa P_2: $n_{\sqrt{\ }} = \uparrow$ e calculemos a pf de cada:

$pf(P_1,R) = (n_{\sqrt{\ }} = n_{\sqrt{\ }} \wedge n \geq 0) v (n_{\sqrt{\ }} = \uparrow \wedge n < 0) = n \geq 0$

$pf(P_2,R) = (\uparrow = \sqrt{n} \wedge n \geq 0)(\uparrow = \uparrow \wedge n < 0) = n < 0$

de P_1 e P_2 tem-se o programa P°

P: if $\lfloor n \geq 0 \rightarrow n_{\sqrt{\ }} \leftarrow \sqrt{n}$
$\lfloor n < 0 \rightarrow n_{\sqrt{\ }} \leftarrow \uparrow$

if

Como **segundo exemplo** aplica-se a metodologia a um trecho de programa que deve calcular o $\sum_{i=0}^{n} i$, isto é, a soma dos naturais $\leq n$.

A pós-condição R pode ser definida como:

R: soma = $\sum_{i=0}^{n} i \wedge n \geq 0 \wedge i \leq n$

Considere-se os programas

P_1: if $\lfloor n=0$ soma$\leftarrow$0

fi

P_2: if $\lfloor n>0$ e $i \leq n \rightarrow$ soma$\leftarrow$soma + i

fi

e calcula-se

$pf(P_1,R) = i=0 \wedge i=0 \rightarrow pf(soma \leftarrow 0, R)$

$= \{i=0 \wedge i=0 \rightarrow 0=0\} =$ verdadeiro

$pf(P_2,R) = n>0 \wedge i \leq n \rightarrow pf(soma \leftarrow soma+i, R)$

que por indução finita pode ser provado igual a verdadeiro.

compondo-se P_1 e P_2 tem-se

P: $\{n \in N\}$

soma$\leftarrow$0

$i \leftarrow 0$

do

$\lfloor i \leq n \rightarrow$ soma$\leftarrow$soma +i

od

$\{n \in N \wedge soma = \sum_{i=0}^{n} i\}$

III. O Método da Transformação de Programas

Desenvolver um programa por transformação consiste em especificar o problema que se deseja solucionar na mesma linguagem a ser usada no processo de programação. Para isto define-se uma linguagem de largo espectro, capaz de, como por exemplo, cálculo de predicados e funções, formular um problema, e como por exemplo, ponteiros e registros, representar concretamente a sua solução. Uma vez provada a correção da especificação inicial, utilizando-se de mecanismos de transformações bem definidos e corretos, transforma-se a formulação abstrata do problema numa formulação concreta da solução. Seguem-se alguns aspectos da definição de uma linguagem de largo espectro e de mecanismos de transformações devidos a Bauer[3]:

i. definições de modos e objetos
- modos primitivos nat,int,real,bool pré-definidos fora da linguagem
- modos atômicos definidos por enumeração
- produto cartesiano
mode m $(m_1,S_1,...,m_n,S_n)$ onde m_i é o modo componente e S_i o seletor

- array
produto cartesiano de um mesmo modo
- função com argumentos $x_1,...,x_n$ e modos $m_1,...,m_n$ que produz resultados de modos $r_1,...,r_k$ de corpo E é:
funct f= $(m_1x_1,...,m_nx_n)$ $(r_1,...,r_k)$:E

ii. definições de mecanismos de controle
- cláusula iterativa
do S; if B then E leave fi; T od

onde S e T são comandos, B é uma expressão booleana e E uma expressão. O mecanismo permite uma saída da iteração com resultado E sob condição B
- quantificadores são permitidos desde que restritos a um único modo. A expressão some mx:p(x) é uma expressão de escolha que produz um m objeto arbitrário tq p(x) é verdadeiro. A expressão that mx:p(x) é determinística e exige uma unicidade do elemento que atende p(x).
iii. mecanismos de transformação
- transformação de recursão em iteração

funct f = (mx)n:
if B_0 then t(x)

elif B_1 then $f(g_1(x))$
⋮
elif B_n then $f(g_n(x))$

else

$f(g_{n+1}(x))$

fi

⇧
⇩

funct f = (my):n

var mx: = y;
t(x)
while $\neg B_0$

do if B_1 then $x:=g^1(x)$
elif B_2 then $x:=g_2(x)$
⋮
elif B_n then $x:=g_n(x)$

else

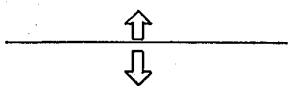
$x:=g_{n+1}(x)$

fi

od

- definição do while

while B do W od



if B then nop

else W; while B do W od fi

A idéia central do método de síntese de programas por transformação é a de que em geral uma definição recursiva de um problema de programação é a expressão mais natural da definição deste problema.
Como exemplo para a aplicação do método seja o mesmo cálculo do problema anterior soma =

$$\sum_{i=0}^{n} i \quad i \in N$$

i. obter a soma dos números naturais menores que um determina n.

ii. fun soma = (natn)nat:
if n=0 then 0
else n + soma(n-1)
fi
iii. transformando recursão em iteração
fun soma = (naty):nat

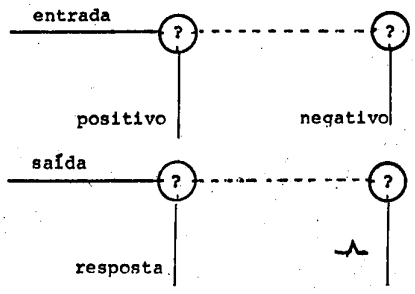
var nat n:=y;
soma:=0;
while ¬ n=0
do

soma:=soma + n
n:=n-1

od

IV - O Método da Estrutura de Dados

O ponto central do método devido a Cowan e Lucena [5, 7, 8] consiste em desenvolver programas nos quais exista uma correspondência um a um entre as estruturas dos tipos de dados característicos das entradas e saídas do programa e as estruturas de controle. Para alcançar este objetivo faz-se:
i. os dados de entrada e saída serão estruturados com mecanismos análogos aos de estruturação de programa, i.e., produto cartesiano, união discriminada e sequência.
ii. o programa passa a ser visto como um conjunto de produções que transforma dados de entrada em dados de saída ambos descritos de maneira apropriada.
iii. a solução do problema passa a ser dividida entre a procura da especificação dos dados e a da transformação das especificações.
Como primeiro exemplo seja o problema de derivar um trecho de programa que deve calcular a $\sqrt{n}$ onde n é um inteiro, a n só deve ser calculada se $n \geqslant 0$ caso contrário o programa deve ser cancelado.
i. especificação das entradas e saídas
type entrada = (positivo, negativo)
positivo = (inteiro $\geqslant$ 0)
negativo = (inteiro $<$ 0)
saída = (resposta, ↯)
resposta = (inteiro $\geqslant$ 0)
↯ = (mensagem erro)
ii. especificação gráfica das estruturas

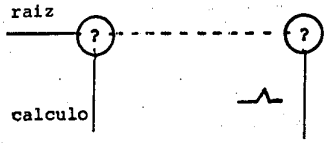


iii. estrutura do programa
programa raiz (x:entrada): saída

if testa(x)

then

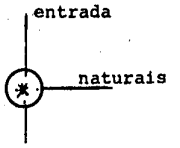
cálculo(resposta)

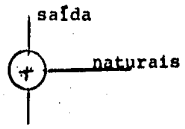


else
↯
fi
end

Como segundo exemplo seja o problema do cálculo da soma dos n primeiros naturais:
i. especificação de dados
type entrada = sequência de naturais

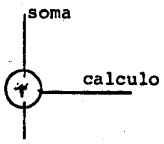
type saída = sequência de naturais





ii. especificação do programa
programa soma(x:natural): natural
while condição(x)

do cálculo od end isto é;



programa soma (x:natural):natural

soma←0

while x$<$n

do soma←soma + x

od end

V - O Método dos Tipos Abstratos de Dados

A metodologia dos tipos abstratos de dados ou da axiomática de tipo de dados é devida a Pequeno e Lucena [4,1].

Segundo seus autores a síntese de um programa pode ser obtida por:

i. escolha do tipo de dados adequada
ii. definição de propriedades dos tipos
iii. derivação do esquema de programa
iv. escolha da representação de dados e definição do tipo de dados desta representação
v. prova da correção da representação
vi. derivação dos cluster.
Uma linguagem de primeira ordem é a apropriada para i, ii e iii. Os axiomas que definem o tipo de dados são apenas os suficientes para permitir a verificação dessas propriedades, não necessariamente definindo completamente o tipo. A passagem iii, iv é obtida por interpretação entre teorias.
Para exemplo seja o problema de calcular a soma de inteiros menores que um determinado valor.
i. o tipo de dados será definido pela linguagem:

L=$<$menor, sucessor,+$>$
onde menor é um predicado binário indicando a relação usual de ordem em N
sucessor é uma função unária representada

a função sucessor S: N→N, isto é, S(n)=n+1
+ é um símbolo funcional binário indicando a operação de soma em N, isto é,
+: Nx N→N

o programa poderia ser descrito por:

$\{n \in N\} P(n,x) \{x=0 + suc(0) + ... + suc^n(0)\}$

As descrições dos termos da linguagem seriam:

menor
i. $x \not< Sy \leftrightarrow x < y$
ii. $x \not< 0$
iii. $x<y \;\; x=y \vee y \; x$
iv. $x< y \wedge y<z \rightarrow x<z$

sucessor

i. $S_x \neq 0$
ii. $S_x = S_y \rightarrow x=y$
iii. $y \neq 0 \exists x \mid y=Sx$
iv. $Sx \neq x$
$SSx \neq x$
⋮
$S^n x \neq x$

soma

i. $x + 0 = x$
ii. $x + S(y) = S(x+y)$
iii. $x + y + y + x$
iv. $(x+y)+z = x +(y+z)$

Uma primeira versão do programa seria:

n∈N
i←0
S←0
while ¬i<n
do
i←suc(i)
soma←S+i
S←soma
od
soma = 0+suc(0) +...+ $suc^n(0)$

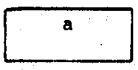
A prova da correção do esquema pode ser feita por indução finita e a escolha da representação depende apenas da linguagem em que o programa seja escrito.

VI - Metodologia de Jackson

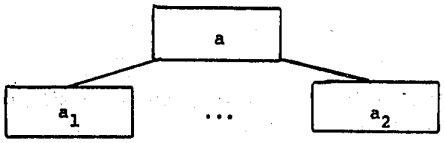
Jackson [9] em sua metodologia mostra que entradas e saídas de programas podem ser encaradas como linguagens infinitas (sobre um conjunto primitivo de dados). Para descrever estas entradas e saídas Jackson propõe uma árvore que pode ser usada apenas para representar linguagens regulares. Podemos inclusive mostrar que as árvores de Jackson são uma notação alternativa para expressões regulares usando definições de tipos de dados como sequência união discriminada e produto cartesiano.

Jackson propõe as seguintes representações:

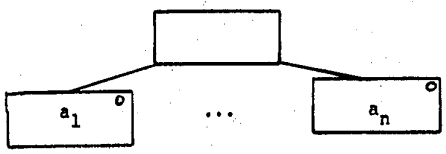
i. símbolo terminal



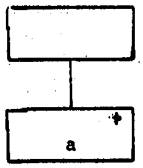
ii. concatenação de expressões regulares



iii. união discriminada de expressões regulares



iv. iteração de expressões regulares



A metodologia de Jackson pede que correspondência sejam detetadas entre a especificação da entrada e da saída em termos de correspondência entre subestruturas entre as duas especificações. Isto é feito de baixo para cima de tal forma que a transformação de um nó (de entrada para saída depende apenas de seus descendentes). Esta correspondência acaba por definir uma transformação de toda a especificão de entrada (árvore de entrada) na especificação de saída (árvore de saída).
Hughes [17] provou que para uma dada caracterização de gsm's pode-se mostrar que o método básico de Jackson dá origem a programas que são gsm computáveis.

A correspondência em que se apóia a metodologia acaba sendo equivalente a transformar nós da árvore de entrada na árvore de saída.

Formalmente:
i. seja I uma expressão de entrada definida num vocabulário Σ

ii. seja O uma expressão de saída definida num vocabulário Δ

iii. diz-se que O=output(I) se O pode ser derivado de I pela aplicação das seguintes regras, onde R e Q são expressões regulares

i. $R \in \Sigma$=output(R) $\in \Delta \cup \{\epsilon\}$ onde ∈ expressão saída

ii. output(R.Q) = output(R). output(Q)

iii. output(R Q) = output(R) output(Q)

iv. output(R*) = (output(R))*

v. R.∈=∈R=R

vi. R Q = Q R

vii. R R = R

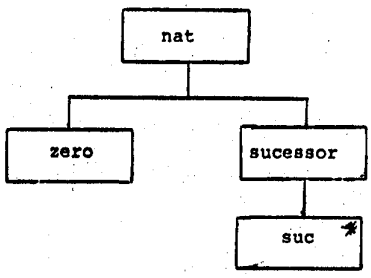
viii. (R*)* = R*

ix. $\epsilon^* = \epsilon$

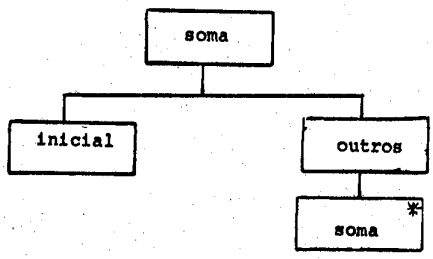
Definindo-se uma máquina sequencial generalizada (gsm) como sendo uma sextupla $(S, \Sigma, \Delta, \delta, \lambda, q_1)$ onde
i. S é um conjunto vazio não finito
ii. Σ é o alfabeto e entrada
iii. Δ é o alfabeto de saída
iv. $\delta : S \times \Sigma \rightarrow S$(função entrada)
v. $\lambda : S \times \Sigma \rightarrow \Delta^*$(função saída)
vi. q_1 elemento distinguido (estado inicial)

Tem-se que uma função $f: \Sigma^* \rightarrow \Delta^*$ é gsm computável se e somente se:
i. f preserva as subpalavras iniciais.
ii. f tem saídas limitadas em comprimento
iii. $f(\epsilon) = \epsilon$
iv $f^1(R)$ é regular desde que R seja regular.

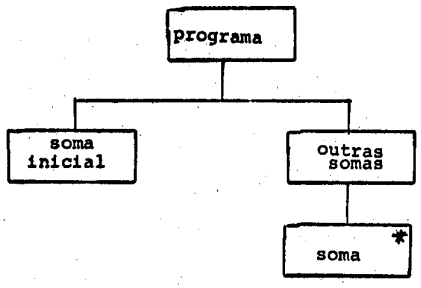
As expressões da metodologia básica de Jackson; backtracking, multithreading e ordenação F' violam a propriedade i enquanto que o conflito de fronteiras pode ser resolvido por decomposição de problemas.
Como exemplo tomemos de novo o problema de soma dos naturais.
i. Especificação de entrada



ii. Especificação de saída
Imaginamos a representação usual de naturais zero, suc^n(zero)

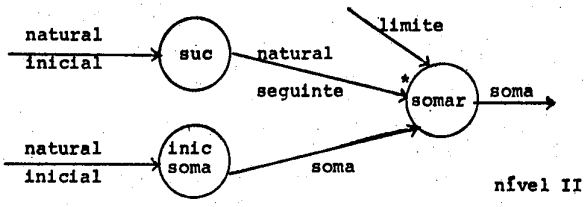


iii. Especificação do programa

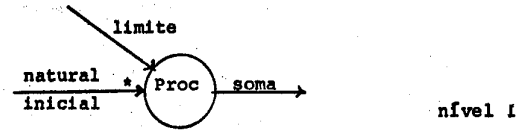


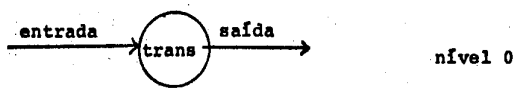

```
programa
natural←0

soma(inicial)←0
while natural≤n
do
soma←soma+natural
natural←natural + 1
od
end
```

VII - Metodologia do fluxo de dados
O propósito desta metodologia, devido a Constantine e Yourdon [II], é identificar as funções primárias a serem processadas, suas entradas e saídas de alto nível. Cria então módulos para: criar as entradas de alto nível, transformar entrada em saída e processar esta saída. Obviamente, o projeto de um fluxo de dados é um modelo de fluxo de informação.
Como outros modelos de fluxo de informação, a análise de transformação usa um modelo gráfico de computação. Este modelo é chamado de diagrama de fluxo de dados. Neste diagrama cada nó representada uma transformação de dados (que será realizada mais tarde por um módulo de programa). Os elementos de dados são representados por flechas conectando os nós. O exemplo abaixo mostra um diagrama de fluxo de dados com uma única entrada e uma única saída.
Uma transformação pode necessitar (ou aceitar) mais de uma entrada para produzir uma resposta. Um asterístico "*" ou o símbolo de disjunção "+" indica *, respectivamente, a necessidade simultânea de elementos ou uma situação mutuamente exclusiva..
Yourdon e Constantine propuseram uma metodologia de projeto de fluxo de dados que consta basicamente de quatro passos. Segue-se a proposta.
Passo 1: Consiste em expressar o problema como um fluxo de dados; os autores recomendam que o projetista deve ser preocupar primeiramente com o caminho principal dos dados que trata das entradas primárias.
Passo 2: Identificação dos elementos de dados inicial e final. Os elementos de dados inicial e final são os elementos de dados que são primeiramente removidos da entrada e saída, fisicamente falando.
Passo 3: Este passo é subdividido em quatro outros. Uma vez detetados os elementos de dados inicial e final de cada sistema temos de:
i. especificar um módulo principal que quando ativado deverá realizar todas as tarefas do sistema chamando seus subordinados.
ii. para cada elemento de dado inicial que alimenta a transformação central um módulo inicial é especificado subordinado ao principal.
iii. para cada elemento de dado final produzido pela transformação central é especificado um módulo final subordinado a esta transformação.
iv. para cada transformação central ou composição coesa* de transformações centrais, é especificado um módulo de transformação subordinado ao central que recebe do módulo principal as entradas apropriadas e as transforma nas também apropriadas saídas.
Yourdon e Constantine fazem notar que a este ponto existe uma simples, usualmente um para um, correspondência entre o fluxo de dados inicial e o diagrama de módulos associado a ele.
Passo 4: Neste passo fatora-se os módulos inicial, final e de transformação até que as entradas e saídas físicas sejam alcançadas e até que os detalhes dos módulos de transformação detetados durante a análise do problema também estejam descritos.
O objetivo principal desta análise de transformação é fazer a estrutura do programa refletir a estrutura da formulação do problema desde que feita sob o aspecto de um fluxo de dados.
Seja o problema de soma de n naturais. Sua expressão num diagrama de fluxo de dados pode ser exemplificada a seguir:

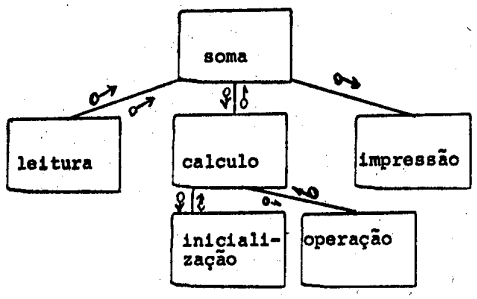


* Devido a não ser o objetivo principal, foi omitido muitos aspectos da metologia como coesão, acoplamento, robustez, etc...





que dá origem ao seguinte diagrama de estrutura de programa



que dá origem ao seguinte pseudo código

```
program soma
natural(inicial)←0
soma(inicial)←0
while natural≤n
do
```

```
soma←soma + natural
natural←suc(natural)

od

end
```

VIII - Conclusão

Existem outras maneiras de se desenvolver programas [16,17]. Na verdade esta é uma área aberta em pesquisa. Tanto de um ponto de vista prático, quanto de um ponto de vista teórico, construir programas corretos e de uma maneira semi-automática é um desafio. Talvez um desafio maior esteja em encontrar uma metodologia que tenha o rigor teórico de metodologias como a de transformadores de predicados ou a de transformadores de programas e a aplibilidade do método de Jackson ou do método de Constantine. Louve-se nesta área o trabalho de Hughes [17] procurando dar suporte teórico ao trabalho de Jackson.
Um trabalho de comparação deveria ao final recomendar alguma forma de apresentação como ideal. Contudo, todas as metodologias têm suas vantagens e o conhecimento global delas facilita e orienta o trabalho de programas.

Bibliografia

1. Lucena, C.J.P.; Análise e síntese de programas de computador; Editora Universidade de Brasília, Brasília, 188 páginas, 1982.
2. Dijkstra, E., "A discipline of programming", Prentice-Hall, 1976.
3. Bauer, F.L., Broy, M., Gnutz, R., Herse, V.; "Notes on the Project CIP: Towards a wide Spectrum Language to support Program Development by transformations", TVM-INFO-A-729, Universidade Técnica de Munique (TVM), Munique, 1979.
4. Lucena, C.J.P. e Pequeno, T.H.C.; Program Derivation Using Data Types: A case study; IEEE Transactions on Software Engineering, New York, Vol.SE-5, nº 6, 1979.
5. Cowan, D.D.; Graham, J.W.; Walch, J.V.; Lucena, C.J.P., A Data-directed Approach to Program Construction, Software Practice and Experience, New York, vol. 10, 1980.
6. Lucena, C.J.P. e Pequeno, T.H.C.; "Program Derivation Using Data Types: A Case Study"; IEEE Transactions on Software Engineering, vol. SE-5, nº 6, 1979.
7. Cowan, D.D.; Lucena, C.J.P; "A data directed Approach to Program Construction", Research Report CS-78-02, University of Waterloo, Waterloo, 1978.
8. Cowan, D.D.; Lucena, C.J.P.; "Some thoughts on the construction of Programs - A Data-Directed Approach Information Technology, J. Moneta, Ed. North-Holland, Publ Co., 1978.
9. Jackson, M.A., Principles of Program Design, Academic Press, London, 297 páginas, 1975.
10. Warnier, J.D.; Logical Construction of Programs; Van Nortrand Reinhold, New York, 1974.
11. Yourdon, E.; Constantine, L.L.; Structural Design: Fundamental of a Discipline of Computer Program and System Design; Prentice Hall, New Jersey, 473 páginas, 1978.
12. Floyd, R.W.; "Assigning meaning to programs", Proceedings of a Symposium on Applied Mathematics, vol. 19, American Mathematical Society, 1967.
13. Hoare, C.A.R., "An Axiomatic Basis for Computer Programming", Comm. ACM 12, 10, outubro d 1969.
14. Hoare, C.A.R., "Notes on Data Structuring", APIC Studies in Data Processing nº 8, Academic Press, London, 1972.
15. Lucena, C.J.P.; Martins, R.C.B.; Krahe, M.L.; "Projeto de Programas assistidos por Micro computadores", Anais do XIII Congresso Nacional de Processamento de Dados, Rio de Janeiro, pg. 309, 1980.
16. Martins, R.C.B.; Costa, H.J.L.; "Disciplina de Programação em Ambiente de Produção", Anais do XV Congresso Nacional de Processamento de Dados, Rio de Janeiro, pg. 890, 1982.