Anales PANEL'81/12 JAIIO
Sociedad Argentina de informática
e Investigación Operativa. Buenos Aires, 1981

UM METODO DE PROGRAMAÇÃO BASEADO NO MODELO DATA-FLOW

Gentil J. de Lucena Filho
Depto. de Sistemas e Computaçao
Universidade Federal da Paraíba

Maarten H. van Emden
Dept. of Computer Science
University of Waterloo

1. INTRODUÇÃO

Em programação, como em outras atividades, é útil distinguir-se entre fins e meios. Quando se escreve um programa, o fim é uma certa relação entre entrada e saída a ser computada pelo programa. Os meios são certas ações primitivas que, devidamente sequenciadas pelo programa, estabelecerão o fim desejado. No programa, o controle deste sequenciamento tanto pode ser especificado explicitamente (como, por exemplo, através de GO TO's) quanto implícitamente (como, por exemplo, através de WHILE's e IF's). Neste artigo, a palavra "controle" será usada para referir-se ao controle de sequenciamento de ações.

Um dos objetivos em metodologia de programação é possibilitar a produção de programas de cada vez mais alta qualidade, verificáveis, com uma quantidade de esforço, previsível, cada vez menor. Controle, há muito, tem sido identificado como uma fonte de dificuldade em programação. E.W. Dijkstra, por exemplo, é conhecido por ter chamado atenção para as desvantagens inerentes à especificação explícita de controle comparada à sua especificação implícita em construtos de programação gol-orientados (isto é, fim-orientados). Uma forma de melhorar métodos de programação é facilitar a especificação do fim (isto é, a relação entrada/saída) e requerer menos atenção aos meios (isto é, controle) através dos quais aquele é conseguido.

Ainda recentemente, Kowalski [6] caracterizou de forma precisa, o papel desempenhado por controle na especificação de algoritmos. Ele mostrou que, num certo formalismo ("programação

lógica"), é possível separar um algorítmo numa componente lógica e numa componente de controle. Kowalski mostrou que existem interpretadores "inteligentes" (mas, ineficientes) capazes de executar a lógica de um algorítmo com pouca ou quase nenhuma informação de controle. O controle requerido em programação convencional também pode ser usado em programação lógica; nesse caso, um interpretador "não tão inteligente" (mas, eficiente) é suficiente.

Neste trabalho procuramos explorar, esta separação entre lógica (fins) e controle (meios) em programação "data-flow" ("data-flow programming"). Uma significante classe de problemas de programação é naturalmente expressa em termos de "data-flow": a entrada e a saída são feixes de dados ("data streams"); tipicamente, o programa efetua uma transformação sobre o feixe de entrada ("input stream"). Um programa data-flow pode ser visto como um sistema de módulos, interconectados, com feixes de dados fluindo entre eles. A característica principal do modelo data-flow é que o sequenciamento das ativações/desativações dos vários módulos é controlado implícitamente pela maneira em que os módulos são conectados pelos feixes de dados. Especificamente, a ativação (ou desativação) de um módulo depende apenas da presença (ou ausência) de dados no seu feixe de entrada. Naturalmente, a fim de que esta forma implícita de controle seja efetuada automaticamente, necessário se faz o uso de uma linguagem especial, que chamamos de linguagem data-flow. Neste artigo, mostramos que o modelo data-flow é útil, mesmo no contexto de programação com uma linguagem de programação sequencial convencional. Nesse caso, embora (ainda) tenhamos que nos preocupar com os meios, podemos pelo menos "dividir para conquistar" ("divide-to-conquer"): primeiro concentramo-nos no fim e produzimos módulos funcionalmente semelhantes àqueles de um programa data-flow; então, concentramo-nos nos meios, isto é, programamos explicitamente o controle necessário para estabelecer corretamente a cooperação entre os módulos. Este controle, conforme veremos, é fácil de programar. Além disso, temos a vantagem de, pelo menos parcialmente, separar considerações de controle da lógica do programa.

2. O MODELO DATA-FLOW

Um programa data-flow consiste de módulos, cada um com zero ou mais feixes de entrada e zero ou mais feixes de saída. Um feixe pode ser, simultâneamente, entrada para um módulo e saída para outro. Nesse caso, dizemos que existe um elo ("data-link") dirigido de um módulo para outro. Estes elos constituem a única forma de interação entre módulos no modelo data-flow.

No método descrito neste artigo usamos uma linguagem de programação (de alto nível) convencional, ou seja, WATFIV [8]. Primeiro assumimos o modelo data-flow: escrevemos separadamente cada módulo que expressa apenas uma função entre seus feixes de entrada e de saída. Nestes módulos, construtos tipicamente convencionais tais como IF-THEN-ELSE e WHILE-DO são permitidos. As ca

racterísticas de data-flow são incorporadas nos módulos através de dois operadores: GET e PUT. Estes operadores definem operações sobre feixes da seguinte maneira: sejam GET_i e PUT_i operações sobre o feixe s_i num programa. A avaliação de $GET_i(x)$ resulta no valor TRUE no caso do feixe s_i estar não vazio e em FALSE caso contrário. No primeiro caso, tem-se como efeito colateral a atribuição a x do próximo elemento de s_i. No segundo caso, a chamada não terminará e o módulo chamante ficará bloqueado durante todo o tempo em que s_i permanecer vazio. A avaliação de $PUT_i(x)$ tem como único resultado a inclusão de x como último elemento de s_i. Via GET e PUT, cada módulo, respectivamente, pode assumir que o próximo elemento de um feixe de entrada não vazio está sempre disponível e que um elemento poderá sempre ser adicionado a um feixe de saída. Um feixe pode ser imaginado como uma fila sem limites. Para feixes de entrada finitos, um módulo pára após completar o processamento do seu último dado de entrada. Uma vez completos os módulos, os feixes são implementados através de definições, subprogramas, para os operadores PUT_i e GET_i. Além disso, pequenas modificações nos módulos, também se fazem necessárias.

A seguir demonstramos o método através de um exemplo simples de processamento de texto devido a Naur [9]. Nesta seção apresentamos apenas a componente lógica do algorítmo na forma de módulos data-flow. Nas seções seguintes, apresentamos a componente de controle a qual, através da implementação dos feixes, estabelecerá o controle requerido para interação entre os módulos.

2.1 O Exemplo

O problema é escrever um programa que lendo uma sequência de caracteres, S, imprime uma sequência de linhas, S'. A sequência de entrada S é tal que cada caracter ou é um BLANK (branco), um NLCR ("New Line and Carriage Return), ou um "caracter de palavra" (isto é, qualquer caracter que não seja BLANK, NLCR, ou "%", o qual marca o fim de S). Na sequência de saída, S', o número de linhas impressas deve ser o menor possível. Cada linha é uma sequência de "palavras" separadas, entre si, por um único branco e deve conter no máximo COMLIN (comprimento da linha) caracteres. Cada palavra é uma subsequência maximal de "caracteres de palavras". Requer-se ainda que a concatenação de sucessivas palavras contidas nas sucessivas linhas seja igual à sequência de sucessivos caracteres de palavras contidos em S.

A seguir construimos um programa data-flow com dois módulos: MOD1 e MOD2. A entrada para MOD1 é a sequência S. Sua função é "filtrar" palavras de S. Nesta tarefa, MOD1 inclui um branco delimitador entre cada duas palavras consecutivas. Esta sequência de palavras, saída de MOD1, constitui a entrada para MOD2 que, então, completa o processamento requerido. Ao conectar a saída de MOD1 à entrada de MOD2, estabelecemos um elo dirigido

de MOD1 para MOD2. Associado com este elo existe um feixe de dados sobre o qual as únicas operações possíveis são $PUT_1(x)$ (por MOD1) e $GET_1(x)$ (por MOD2).

(Nota 1: O feixe de dados associado com $PUT_1(x)$ e $GET_1(x)$ pode ser visto tanto como um feixe de palavras (caso em que x assumirá palavras com valores) quanto como um feixe de caracteres (caso em que x assumirá caracteres como valores). No primeiro caso, a tarefa de empacotar e desempacotar caracteres em palavras é feita em PUT_1 e GET_1, de modo que os módulos ficam mais simples. No segundo caso, esta tarefa é realizada nos módulos, de modo que a simplificação cai em PUT_1 e GET_1. Escolheremos a última alternativa, isto é, um feixe de caracteres. Achamos que "caracteres" sendo mais primitivos do que "palavras", são mais apropriados no sentido de uma eventual padronização na forma de interconexão entre módulos devendo, com isso, facilitar a ligação entre módulos já existentes e novos módulos.)

(Nota 2: Excetuando PUTs e GETs, todos os identificadores que aparecem num módulo são locais a ele.)

Os módulos, MOD1 e MOD2, com exceção dos PUTs e GETs são codificados em WATFIV e são apresentados nos Quadros 1 e 2, respectivamente.

```
C       DECLARAÇÕES
C       NOTEOI (NOT-END-OF-INPUT): INICIALIZADO COM .TRUE.
        WHILE (NOTEOI) DO
          NOTEOI = GET0(CAR)
          WHILE (NOTEOI.AND.(CAR.EQ.BLANK.OR.CAR.EQ.NLCR)) DO
            NOTEOI 1 GET0(CAR)
          END WHILE
C         NESTE PONTO, CAR OU É UM CARACTER DE PALAVRA OU %
          WHILE (NOTEOI.AND.CAR.NE.BLANK.AND.CAR.NE.NLCR) DO
            PUT1 (CAR)
            NOTEOI = GET0(CAR)
          END WHILE
          IF (NOTEOI) THEN DO
C           INSERE BRANCO NO FIM DA PALAVRA
            PUT1 (BLANK)
          END IF
        END WHILE
        STOP
        END
```

Quadro 1

```
C       DECLRAÇÕES
C       ARRAYS PALVRA E LINHA TEM, AMBOS, TAMANHO COMLIN.
C       PRESUPOE-SE QUE TODA PALAVRA NO TEXTO DE ENTRADA TEM
C       COMPRIMENTO MENOR OU IGUAL A COMLIN.
C       NOTEOI: INICIALIZAÇÃO COM .TRUE.
        INDLIN = 0
        WHILE (NOTEOI) DO
          COMP = 0
          NOTEOI = GET1(CAR)
          WHILE (NOTEOI .AND. CAR.NE.BLANK) DO
            COMP = COMP + 1
            PALVRA (COMP) = CAR
            NOTEOI = GET1 (CAR)
          END WHILE
          IF((INDLIN + COMP) .GT. COMLIN) THEN DO
            PRINT, (LINHA(I), I = 1, INDLIN)
            INDLIN = 0
          END IF
          I = 1
          WHILE (I .LE. COMP) DO
            INDLIN = INDLIN + 1
            LINHA (INDLIN) =  PALVRA(I)
            I = I + 1
          END WHILE
          IF (INDLIN .LT. COMLIN .AND. NOTEOI) THEN DO
            INDLIN = INDLIN + 1
            LINHA (INDLIN) = BLANK
          END IF
        END WHILE
        IF (INDLIN .GT. 0) THEN DO
          PRINT, (LINHA(I), I=1, INDLIN)
        END IF
        STOP
        END
```

Quadro 2

3. GERENCIAMENTO SISTEMÁTICO DE CONTROLE EM PROGRAMAS DATA-FLOW

O programa data-flow exemplificado neste artigo é "linear", isto é, o elo (único, no caso) existente entre os módulos impõe uma ordem linear entre os mesmos. Generalizando, dizemos que, um programa data-flow é linear, quando cada módulo tem no máximo um feixe de entrada e um feixe de saída.

Um feixe é, na realidade, uma fila com disciplina FIFO ("First-In-First-Out") sem dimensão prefixada. No modelo data -flow, qualquer subconjunto dos módulos, com velocidades relativas diferentes, pode ser simultâneamente ativado; a única restrição é que um módulo deve esperar quando de uma operação GET mal sucedi

da, isto é, sobre um feixe de entrada vazio. A seguir, introduzimos, sucessivamente, restrições adicionais necessárias no controle inter-módulos requerido por linguagens sequenciais convencionais.

R1. As filas tem ura dimensão limitada. Com isso, módulos também deverão esperar quando de uma operação PUT numa fila cheia.

R2. Em qualquer instante, apenas um módulo executa. A partir de agora, podemos dizer que um módulo "tem controle", ou não, e que um módulo "transfere controle" para outro módulo.

R3. Transferências de controle ocorrem apenas entre "vizinhos". Dois módulos são vizinhos quando existe um elo entre eles.

Das restrições acima segue-se que se módulo X transfere controle para módulo Y, então caso X seja reativado, esta reativação se dará devido a uma transferência de controle de Y para X. Além disso, esta transferência não deverá provocar uma nova ativação mas sim, um restabelecimento da última ativação. Isto significa que para cada módulo, em qualquer instante, existirá apenas uma ativação. Toda a informação de controle consistirá apenas de simples rótulos (marcando os pontos de retorno) em cada módulo. Este esquema de controle é consideravelmente mais simples que aquele usado com corotinas irrestritas, em que cada uma requer uma pilha de pontos de retorno. A próxima restrição, R4, estabelece uma estratégia de escalonamento ("scheduling strategy") que simplifica a implementação de filas.

R4. Cada módulo executa o máximo de tempo possível. Isto é, até que sua fila de entrada (saída) se torne vazia (cheia), o que ocorrer primeiro.

Esta restrição, implica na seguinte propriedade:

Propriedade 1: Quando um elemento é inserido numa fila, deleções não ocorrerão até que a fila se torne cheia (a não ser que a marca de fim de dados no feixe de entrada seja encontrada primeiro). Similarmente, quando um elemento é deletado duma fila, inserções não ocorrerão sem que a fila primeiro se esvazie.

Com esta propriedade, não precisamos mais de filas no sentido usual, o que não introduz quaisquer restrições no ordem de ocorrências de deleções e inserções. A partir de agora, nos referiremos a filas com "buffers".

Uma outra propriedade, agora decorrente do modelo data-flow, é o que chamamos de propriedade da "auto-suficiência":

Propriedade 2: O correto funcionamento de cada módulo ja

mais dependerá da presença simultânea de elementos num buffer. Noutras palavras, cada buffer poderá ter qualquer dimensão que seja pelo menos igual a dimensão de um elemento. Caso elementos seja necessários simultaneamente, o módulo em questão deverá prover armazenamento interno para tais.

4. UTILIZAÇÃO DAS RESTRIÇÕES R1, R2, R3 E R4.

Nesta seção descrevemos como usar as restrições R1, R2, R3 e R4 para completar a transformação dos módulos data-flow da Seção 2 num programaWATFIV. Conforme veremos, a modificação mais importante a ser feita nos módulos é a adição de subprogramas definindo as operações PUT e GET.

O i-ésimo módulo (i=2,...,n-1) de um programa data-flow (linear) obtém seus dados de entrada (via GET_{i-1}) do seu (i-1)-ésimo buffer e coloca sua saída (via PUT_i) no i-ésimo buffer. $Módulo_{i+1}$ é o "consumidor" de $módulo_i$; $módulo_i$ é o "produtor" do $módulo_{i+1}$. GET_i e PUT_i atuam sobre o mesmo $buffer_i$. Nem os módulos nem outros GETs e PUTs tem acesso do $buffer_i$. (Lembre que os módulos só tem acesso aos buffers indiretamente, via GETs e PUTs). $Módulo_1$ e $módulo_n$ são as interfaces do programa com o meio exterior. Assim sendo, GET_o e PUT_n são tratados separadamente

Quando $módulo_{i+1}$ requisita um dado de entrada, do ponto de vista de data-flow só existem duas possibilidades distinguíveis: ou o feixe de entrada é vazio ou não. Na realidade, existe ainda o caso em que embora o buffer esteja vazio o feixe de entrada ainda não se esgotou. Esta última distinção pode ser resolvida em GET_i e com isso $módulo_{i+1}$ permanece inalterado. Nesse caso, GET_i se encarregará de produzir o próximo elemento de entrada: se este estiver no buffer, GET_i tira-o de lá retornando-o para o $módulo_{i+1}$; caso contrário, GET_i primeiro ativa $módulo_i$ para encher o buffer. De maneira similar, a complicação de um buffer cheio pode ser resolvida em PUT_i: se $buffer_i$ não estiver cheio, então PUT_i insere-lhe o elemento imediatamente; caso contrário, PUT_i primeiro causará a reativação de $módulo_{i+1}$ para esvaziar o buffer.

Com WATFIV, o compartilhamento dos buffers será convenientemente implementado via áreas COMMON. O esquema (tipo corotinas) de transferência de controle intermódulos será simulado via o mecanismo CALL/RETURN e o GO TO computado. Nesta simulação deve-se levar em conta a assimetria (implicada pelo mecanismo CALL/RETURN) entre rotinas chamante e chamada. Isto é, toda instrução RETURN num subprograma chamado deverá ser precedido pelo registro do ponto de reativação ou seja, a próxima instrução

seguinte ao RETURN envolvido. Um GO TO no começo do subprograma se encarregará, quando o subprograma for chamado, de transferir o controle para o ponto de reativação previamente registrado (ou para o início do subprograma no caso de sua primeira chamada). Com isto, temos duas possibilidades a considerar:

(I) Produtores são ativados por CALLs, via GETs. RETURNs (simples) nestes GETs garantirão reativação correta dos consumidores (chamantes). RETURNs (aos GETs) nos produtores, todavia, deverão ser precedidos pelo registro do ponto de reativação tendo em vista a próxima vez que o produtor for reativado.

(II) Análogo a (I): substitua simultaneamente as palavras "produtores" por "consumidores", "consumidores" por "produtores", e "GETs" por "PUTs".

Uma outra escolha a ser feita é quanto à ativação inicial: quem deveria ser ativado primeiro, módulo$_n$ ou módulo$_1$? Por um lado, a ativação de módulo$_n$ causa uma "sucção" dos dados através dos módulos; por outro lado, a ativação de módulo$_1$ faz com que os dados sejam "empurrados" através dos módulos. Referir-nos-emos a esses casos como "modo sucção" e "modo empurrão", respectivamente.

Das quatro combinações de uma escolha entre as possibilidades (I) e (II) e uma escolha entre modo sucção e modo empurrão, apenas duas são viáveis: com (I), modo sucção deve ser escolhido, caso contrário todos os módulos não serão ativados com um único CALL. Análogamente, com (II), modo empurrão deve ser escolhido.

Em adição a todos as considerações de controle discutidas até agora, resta-nos decidir sobre o "critério de parada" do programa. Podemos escolher entre entrada vazia e saída completa. Escolheremos a primeira alternativa, entrada vazia (parece mais fácil de manipular num programa). Comparemos, agora, o modo sucção com o modo empurrão assumindo entrada vazia como critério de parada:

Modo empurrão: a execução de módulo$_1$ é interrompida por causa do fim da entrada. Desde que, nesse ponto, os buffers 1,...,n-1 ainda podem conter dados para processamento, reativações adicionais de módulo$_i$ (i=2,...,n) são necessárias.

Modo sucção: quando módulo$_1$ deteta o fim de entrada já existem chamadas ativas o módulo$_i$ (i=2,...,n), de tal forma que quando módulo$_n$ pára, buffer$_i$ (i=1,...,n) estão todos vazios.

Aparentemente, o modo sucção juntamente com a alternativa (I) é a combinação correta.

Neste ponto, procederemos à transformação dos módulos data-flow num programa WATFIV. Consideraremos dois casos: execução sequencial com e sem buffers.

4.1 Execução sequencial com buffers.

Os módulos (com exceção de módulo$_n$ que é especificado como programa principal - desde que estamos no modo sucção) são transformados em subprogramas função do tipo LOGICAL. O valor verdade de um módulo indicará se sua ativação produziu, ou não, pelo menos um elemento de dados de saída. Logo um resultado .FALSE. implica que o módulo já produziu toda sua saída em chamadas anteriores.

As operações PUT$_i$ e GET$_i$ (i=1,...,n-1) também são definidas como funções do tipo LOGICAL e têm a seguinte forma:

```
LOGICAL FUNCTION PUTi(x)
COMMON/FEIXEi/< atributos do FEIXEi>
<  outras declarações >
IF (< BUFFERi cheio > ) THEN DO
  PUTi = .FALSE.
ELSE DO
   < x torna-se último elemento de BUFFERi >
  PUTi = .TRUE.
END IF
RETURN
END
```

```
LOGICAL  FUNCTION  GETi(x)
COMMON/FEIXEi/< atributos do FEIXEi>
<  outras declarações >
IF (< BUFFERi vazio >)  THEN DO
   <ajusta ponteiros em BUFFERi>
   IF (.NOT. MODULOi) THEN DO
    GETi = .FALSE.
    RETURN
   END
END IF
<  x torna-se próximo elemento de BUFFERi >
GETi = .TRUE.
RETURN
END
```

Agora, de acordo com a combinação alternativa (I) e modo sucção, módulos consumidores são ativados via RETURNs e módulos produtores via CALLs. Isto implica não só que chamadas a GET num módulo não precisam ser modificadas, como também que algumas mo

dificações nos módulos em conexão com chamadas a PUT são necessárias: suponha que módulo$_i$ chama PUT$_i$ e este retorna .FALSE. como resultado. Então, após garantir que em sua próxima reativação sua primeira providência será concluir esta operação de saída mal sucedida, módulo$_i$ deveria retornar controle (via GET$_i$) ao consumidor módulo$_{i+1}$. Assuma ainda que existem m chamadas a PUT$_i$ no texto de módulo$_i$ e que $L_1,\ldots,L_{m+2}$ são rótulos não existentes em módulo$_i$. Seguem-se, então, as regras:

Em módulo$_i$, substitua

RG1: a k-ésima chamada a PUT$_i$(x)	por	L_k	IF(.NOT.PUTi(x)) THEN DO
			CHAVE = k
			RETURN
			END IF
			MODULOi = .TRUE

RG2: a instrução STOP	por	L_{m+2}	CHAVE = m + 2 RETURN

Além disso, imediatamente antes da primeira instrução executável de módulo$_i$,

RG3: insira	L_{m+1}	MODULOi = .FALSE. GOTO($L_1,\ldots,L_{m+2}$), CHAVE

onde CHAVE é uma variável do tipo INTEGER inicializada com m+1 em tempo de compilação.

O programa resultante da aplicação dessas regras aos módulos da Seção 2 é mostrado no Apêndice.

4.2 Execução sequencial sem buffers

Baseados no nosso exemplo, a obtenção de programas pelo método apresentado (até agora) caracteriza-se:(a) pela facilidade na obtenção dos módulos data-flow (a aproximação inicial) e (b) pela simplicidade e sistematização da transformação dos módulos data-flow em programas WATFIV.

Todavia, no contexto de programação sequencial, devido ao excessivo número de vezes em que dados são movidos através dos módulos, o uso de buffers como armazenamento intermediário tende a ser ineficiente. Observa-se, no nosso exemplo, que um caracter desde sua entrada até sua saída no programa, é movido cinco vezes!

A seguir usamos a propriedade da auto-suficiência dos módulos no sentido de obter um programa (sequencial) mais eficiente. Segundo aquela propriedade, o programa obtido anteriormente

deveria funcionar com um buffer de qualquer dimensão diferente de zero. Em particular, para buffers de dimensão um, a intermediação de GETs e PUTs é desnecessária. Nesse caso, o novo esquema de controle pode ser implementado modificando os módulos data-flow de acordo com as regras RG1', RG2', RG3' e RG4' dadas abaixo:

RG1'. Em módulo$_{i+1}$, substitua GET$_i$(x) por MODULOi(x).

RG2'. Em módulo$_i$, substitua PUT$_i$(x) por

```
                MODULOi = .TRUE.
                CHAVE = k
                RETURN
L_k             CONTINUE
```

RG3'. Imediatamente antes da primeira instrução executável de módulo$_i$, insira:

```
                INTEGER CHAVE/m+1/
                MODULOi = .FALSE.
                GO TO(L_1,...,L_m+2), CHAVE
L_m+1
```

Finalmente,

RG4'. Substitua, em módulo$_i$, STOP por

```
                CHAVE = m+2
L_m+2           RETURN
```

O resultado da aplicação dessas regras aos módulos da Seção 2, não é mostrado por limitações de espaço. O leitor, porém, fica convidado a checá-lo. (Note-se que, como antes, os módulos, com exceção de módulo$_n$, são transformados em subprogramas função do tipo LOGICAL).

5. CONCLUSÕES

A decomposição de um algorítmo em componentes lógica e de controle proposta por Kawalski [6] é expressada através de um exemplo simples de processamento de texto em que a componente lógica do algorítmo é especificada por módulos data-flow lineares e independentes. Partindo desta especificação, apresentamos um método através do qual restrições de controle são sistematicamente introduzidas, em duas diferentes maneiras, do modo a transformar os módulos em diferentes programas sequenciais, em WATFIV. Num caso, o programa resultante usa buffers para conectar os módulos e portanto é conceitualmente mais próximo do modelo data-flow. Os buffers reduzem a velocidade de execução do programa por causa dos múltiplos movimentos dos dados através dos módulos. Para módulos auto-suficientes, num contexto sequencial, os buffers não adicionam vantagem ao programa. Apesar disso, o ca-

ráter sistemático (e simples) na especificação de controle do programa é, a nosso ver, uma das vantagens do método. No outro caso, o programa resultante é obtido explorando-se a propriedade da auto-suficiência. Dados são transferidos diretamente entre os módulos eliminando-se, com isso o "overhead" no uso dos buffers. O programa é mais rápido, e até certo ponto, também aproxima-se do modelo data-flow.

No exemplo considerado, observamos ainda que o problema apresenta uma estrutura hierárquica: a saída requerida é uma sequência de linhas, cada uma das quais é uma sequência de palavras que, por sua vez, são sequências de caracteres. Comumente, problemas desse tipo são resolvidos com ninhos de laços("nested loops"): o laço mais externo produz as linhas ativando um laço interno que produz palavras processando caracteres. Um ponto a investigar é se, de alguma forma, módulos data-flow que manipulam dados hierárquicamente estruturados (como no nosso exemplo) podem ser combinados num programa monolítico similar à versão dos laços aninhados. Aparentemente, isto é possível quando os módulos são tais que as ocorrências de PUTs e GETs são limitadas a um certo número e certas posições dentro dos módulos. Para módulos arbitrários, todavia, a inter-relação dos dois modelos (data-flow e ninhos de laços) não parece óbvia.

Em continuidade ao trabalho, estamos agora desenvolvendo um "pré-processador data-flow" [7], cujo objetivo é implementar a transformação de módulos data-flow em programas WATFIV, não apenas de forma sistemática mas, também, automaticamente.

6. TRABALHOS RELACIONADOS

Há já algum tempo, trabalhos nas áreas de arquiteturas de de computadores e linguagens de programação vem sendo influenciados pelo modelo data-flow [2, 5]. Em [10], Peacock apresenta uma excelente "survey" de linguagens de programação ("hardware-orientadas") baseadas no modelo data-flow. Em [3], Kahn e McQueen propõem um atraente modelo computacional em que características específicas da arquitetura do processador são abstraídas da linguagem data-flow associada. Ainda nesta linha, apresentamos em [11] como usar o modelo data-flow no contexto de programação lógica. Na área de metodologia de programação, exemplos de trabalhos influenciados pelo modelo data-flow são encontrados em [1, 4].

7. REFERÊNCIAS

1. CUNHA, P.R.F. & MAIBAUM, T.S.E. - A Communications Data Type for Message Oriented Programming. Lecture Notes in Computer Science, Springer-Verlag, vol. 83, 1980.

2. DENNIS, J.B. - Programming Generality, Parallelism, and Computer Architecture. Proc. IFIP 68.

3. KAHN, G. & McQUEEN, D.B. - Coroutines and Networks of Parallel Processes. Proc. IFIP 77.

4. KERNIGHAN, B.W. & PLAUGER, P.J. - Software Tools. Addison-Wesley, 1976.

5. KOSINSKI, P.R., - A Data-Flow Programming Language. IBM Research Tech Report RC 4264, 1973.

6. KOWALSKI, R.A. - Algorithm = Logic + Control. Research Report, Dept. of Computation and Control, Imperial College, 1977.

7. LUCENA FILHO, G.J. & MENDES, A.M.M., - Um Preprocessador Data-Flow. (Em preparação).

8. MOORE, J.B. & MAKELA, L.J. - Structured FORTRAN with WATFIV. Reston Publishing Company, Inc., 1978

9. NAUR, P., - Programming by Action Clusters. BIT 9, 1966.

10. PEACOCK, J.K. - A Survey of Data-Flow Programming Languages. Master thesis, Dept. of Applied Analysis and Computer Science, Univ. of Waterloo, 1976.

11. Van EMDEN, M.H., LUCENA FILHO, G.J. & SILVA, H.M., - Predicate Logic as a Language for Parallel Programming. Submetido para publicaçao nas Proc. of the Logic Programming Workshop, Hungria, 1980.

```
$JOB GENTIL
$NOEXT
C* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
C     PROGRAMA PRINCIPAL: MODULO2
C     USA ROTINA 'INICIA' PARA INICIALIZAR FEIXES
C* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
      CHARACTER PALVRA(60), LINHA(60), BLANK/' '/, CAR
      INTEGER INDLIN, COMP, COMLIN/60/, I
      LOGICAL GET1, NOTEOI/.TRUE./
C
      CALL INICIA
      INDLIN=0
      WHILE(NOTEOI) DO
        COMP=0
        NOTEOI=GET1(CAR)
        WHILE(NOTEOI .AND. CAR .NE. BLANK) DO
          COMP=COMP+1
          PALVRA (COMP)=CAR
          NOTEOI=GET1(CAR)
        END WHILE
        IF((INDLIN+COMP) .GT. COMLIN) THEN DO
          PRINT 10,(LINHA(I),I=1,INDLIN)
          INDLIN=0
        END IF
        I=1
        WHILE(I .LE. COMP) DO
          INDLIN=INDLIN+1
          LINHA(INDLIN)=PALVRA(I)
          I=I+1
        END WHILE
        IF(INDLIN .LT. COMLIN .AND. NOTEOI) THEN DO
          INDLIN=INDLIN+1
          LINHA(INDLIN)=BLANK
        END IF
      END WHILE
      IF(INDLIN .GT. 0) THEN DO
        PRINT 10,(LINHA(I),I=1,INDLIN)
      END IF
   10 FORMAT(1X,120A1)
      STOP
      END
C* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
C                          MODULO1
C* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
      LOGICAL FUNCTION MOD1(IDUMMY)
      CHARACTER CAR,NLCR/'@'/,BLANK/' '/
      INTEGER CHAVE/3/
      LOGICAL GET0, PUT1, NOTEOI/.TRUE./
C
      MOD1=.FALSE.
      GO TO (20,30,10,40), CHAVE
   10 WHILE(NOTEOI) DO
        NOTEOI=GET0(CAR)
        WHILE(NOTEOI .AND. (CAR .EQ. BLANK .OR. CAR .EQ. NLCR))  DO
```



```
          NOTEOI=GET0(CAR)
        END WHILE
C       NESTE PONTO CAR OU E UM CARACTER DE PALAVRA OU $
        WHILE(NETEOI .AND. CAR .NE. BLANK .AND. CAR .NE. NLCR) DO
   20     IF(.NOT. PUT1(CAR)) THEN DO
            CHAVE=1
            RETURN
          END IF
          MOD1=.TRUE.
          NOTEOI=GET0(CAR)
        END WHILE
        IF(NETEOI) THEN DO
C       INSERE BRANCO NO FIM DA PALAVRA
   30     IF(.NOT. PUT1(BLANK))THEN DO
            CHAVE=2
            RETURN
          END IF
          MOD1=.TRUE.
        END IF
      END WHILE
      CHAVE=4
   40 RETURN
      END
C* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
C
      LOGICAL FUNCTION PUT1(CAR)
      COMMON/FEIXE1/BUF1(120),PRIM,ULT,TAMBUF
      CHARACTER BUF1,CAR
      INTEGER ULT,PRIM,TAMBUF
C
      IF((ULT+1).GT.TAMBUF) THEN DO
        PUT1=.FALSE.
      ELSE DO
        ULT=ULT+1
        BUF1(ULT)=CAR
        PUT1=.TRUE.
      END IF
      RETURN
      END
C* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
C
      LOGICAL FUNCTION GET1(CAR)
      COMMON/ FEIXE1/BUF1(120),PRIM,ULT,TAMBUF
      CHARACTER BUF1,CAR
      INTEGER PRIM,ULT,TAMBUF
      LOGICAL MOD1
C
      IF(PRIM.GT.ULT) THEN DO
        ULT=0
        PRIM=1
        IF(.NOT.MOD1(0)) THEN DO
          GET1=.FALSE.
          RETURN
        END IF
      END IF
```

```
      CAR=BUF1(PRIM)
      PRIM=PRIM+1
      GET1=.TRUE.
      RETURN
      END
C* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
C
      SUBROUTINE INICIA
      COMMON/FEIXE/BUF1(120),PRIM,ULT,TAMBUF
      INTEGER PRIM,ULT,TAMBUF
      CHARACTER BUF1
C
      READ,TAMBUF
      PRIM=TAMBUF
      ULT=0
      RETURN
      END
C* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
C
      LOGICAL FUNCTION GET0(CAR)
      CHARACTER CARTAO(80),CAR,EOI/'%'/
      INTEGER I/80/
C
      GET0=.TRUE.
      IF(I.GE.80) THEN DO
        READ(5,10)CARTAO
        I=0
      END IF
      I=I+1
      CAR=CARTAO(I)
      IF(CAR.EQ.EOI) GET0=.FALSE.
   10 FORMAT(80A1)
      RETURN
      END
$ENTRY
```