ESPECIFICAÇÃO, MODELIZAÇÃO E IMPLEMENTAÇÃO DE SOFTWARE

SINTESE E ESTUDO DE UM CASO CONCRETO

por

José A. Legatheaux Martins

UNL - 6/81 Março de 1981

Especificação, modelização e implementação de Software

Síntese e estudo de um caso Concreto (1)

por

José A. Legatheaux Martins

Departamento de Informática

Faculdade de Ciências e Tecnologia

Universidade Nova de Lisboa

Lisboa, Março de 1981

(1) Trabalho a ser submetido ao exame de habilitação à função de assistente

Perante um ataque geral vêr pág. 1.3 → objectivos. e o título do Cap. 2.

Dúvidas: a especificação funcional totalmente separada dos componentes tem interesse? Um utilizador não acaba por compreender melhor compreendendo a operacionalidade? A concepção de que a funcionalidade é suficiente chega?

Um ponto de vista central: Programar manipulam objectos, as entidades + operação sobre essas entidades.
→ fundamentamo-nos apenas numa visão extensional.

Existem 2 pontos de vista:
Uma escola "Americana" programática em bruta força → Linguagens sistemas de agenda.

Uma escola Europeia: Compreender — maneira de programação

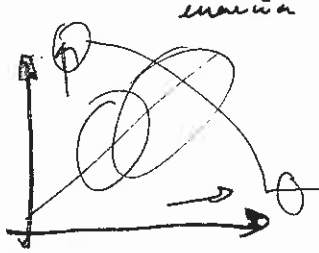


Diagrama de Arrow

→ na prática estas áreas não existem! (só para problemas de almanaque).

Lacunas: Especificação e os métodos de formalização ligados à Teoria de Dana Scott.

Em primeiro lugar desejo agradecer aos membros da equipa de Software Fundamental do D. I. da F.C.T. da UNL, J. Próspero dos Santos e Luís Monteiro, pelo enriquecimento deste trabalho, que a sua colaboração proporcionou.

Um segundo agradecimento vai para Nuno Lobo e Costa Pires pelo entusiasmo e dedicação que deram à tarefa de implementação do módulo de gestão de imagem.

Um agradecimento muito especial vai para a responsável do Grupo de Software Fundamental, Drª Madalena Quirino. Dentro de algum tempo passarão 10 anos sobre a altura em que fui seu aluno nas cadeiras de Sistemas Lógicos e Introdução aos Computadores na Faculdade de Ciências de Lisboa, daí para cá, prácticamente que nunca deixei de ser seu colaborador. Se hoje este trabalho revela uma preocupação metodológica a ela muito o devo. Desde os tempos em que ensinava ALGOL 60 (quando só se falava em FORTRAN) passando pela sua decisiva contribuição para o arranque da Licenciatura em Engenharia Informática e sobretudo para a definição do seu perfil moderno, que toda uma geração de licenciados em Informática muito lhe devem.

Um último agradecimanto vai para o pessoal da Secretaria do Departamento, com especial realce para a Srª Maria Etelvina Pires Duarte que com dedicação e eficiência proporcionaram a edição deste trabalho.

INDÍCE

3. - DESENHO DE UM SISTEMA CONCRETO - ESTUDO DO MONITOR GRÁFICO A INSTALAR NO GT42.

3. 1 - Apresentação do problema.

3. 2 - Soluções possíveis para implementar software de sistema.

3. 3 - Panorâmica geral da solução adoptada.

3. 4 - Modelização do módulo de gestão de imagem.

3. 5 - Conservação das equivalências semânticas - codificação.

3. 6 - Uma visão crítica da linguagem BLISS.

3. 7 - Resultados quanto à documentação.

3. 8 - Um método recomendável para testar módulos de um sistema - catalogação.

3. 9 - Avaliação dos resultados.

4. - BIBLIOGRAFIA

Anexo 1 - Uma solução possível para o módulo de gestão das comunicações.

Anexo 2 - Um modelo possível para o conceito de periférico lógico.

1 - INTRODUÇÃO

Desde o início do ano de 1980 que o autor, integrado numa equipa conjunta formada por elementos do Centro de Informática do LNEC e do Grupo de Software Fundamental do Departamento de Informática da UNL tem vindo a desenvolver um software gráfico de base que se destina a substituir o software fornecido com o computador do LNEC por se reconhecer que este último é inadequado a uma utilização eficiente das possibilidades do terminal gráfico disponível-GT42 da DEC.

Sendo o GT42 um terminal razoávelmente potente, visto ser controlado por um mini-computador de 16 bit's e 28K palavras de memória central, impunha-se desenvolver um software capaz de gerir a comunicação com o computador central, capaz de gerir de forma eficiente e dinâmica as imagens afixadas no écran e ainda capaz de gerir o diálogo com o utilizador através de periféricos lógicos, isto é, de alto nível.

Estas características, aliadas ao facto de se pretender obter um produto com características industriais, capaz de suportar a implementação de outras camadas de software destinadas a serem exploradas intensivamente durante vários anos e atendendo ainda a certas particularidades do projecto (1), tiveram como consequência lógica a necessidade de que o desenho do sistema seguisse um conjunto de técnicas do âmbito da Engenharia de Software e da Construção de Programas.

Assim, ele constitui um excelente motivo para uma reflexão sobre a actividade de especificar, projectar e implementar software. É essa reflexão, de caracter sintéctico, que aqui se dá conta.

Assim, a ideia fundamental que constitui a trave mestra desta sintese é a seguinte: A actividade de especifi-

---

(1) *A equipa é constituída por 7 pessoas, em que nenhuma delas dedica no total, mais de 20% do seu tempo a este projecto!*

car, modelizar e implementar software, ou de forma mais geral, a actividade de resolução de problemas através da informática, consiste em construir um modelo abstracto do problema que se pretende resolver e em seguida simular esse modelo, por um meio qualquer num computador, e a conclusão lógica que lhe está subjacente é que, a ferramenta ideal para a concretização dessa actividade seria constituída por:

a) Uma linguagem para exprimir esse modelo abstracto.

b) Um sistema automático que permita verificar a sua coerência e experimentar o seu funcionamento abstracto.

c) Eventualmente, um sistema que permita através de uma linguagem de programação transformar o modelo abstracto num modelo operacional de preferência de forma automática.(1)

Poderemos dizer que, em última análise, é à roda destas 2 questões e de todas as suas últimas consequências, que gira actualmente a investigação e o desenvolvimento nas seguintes áreas da informática:

a) Construção de Programas.

b) Engenharia de software.

c) Desenho e implementação de linguagens de alto e de muito alto nível.

d) Especificação e verificação de sistemas e programas.

e) Desenho e implementação de software com paralelismo.

---

(1) *Numa visão ainda pouco realista, as funções de b) poderiam dispensar totalmente as focadas em c), pois b) poderia executar a transformação automaticamente obtendo-se assim um modelo operacional eficiente e adequado, desde que se especificasse a interface entre os conceitos abstractos e os periféricos que estabeleciam a relação destes com o mundo real.*

Orientação do trabalho

Em comum a todas estas áreas e problemas, o conceito comum e essencial a reter é, a saber, que a abstracção é o principal meio conceptual pelo qual os problemas informáticos são resolvidos e que a linguagens, no sentido informático, são a principal ferramenta hoje à disposição dos projectistas.

É esta a orientação com que o presente trabalho foi desenvolvido.

Assim, é constituído por 3 capítulos.

Objectivos

a) No segundo capítulo é traçada uma panorâmica das principais frentes de desenvolvimento da Metodologia da Programação, das Linguagens de Alto Nível e da Engenharia de Software. Esta panorâmica é bastante informal e tem um caracter meramente sintético. Cada sub-capítulo, com a respectiva bibliografia, são outros tantos temas a aprofundar. Referimos desde já, que os importantes temas: Primitivas de Sincronização e Validação de Programas não são práticamente referidos.

b) No terceiro capítulo é mostrado como as mesmas ideias podem ser aplicadas num contexto não ideal, isto é, onde não se dispõe das ferramentas atrás enunciadas. É neste capítulo que é feito um estudo das soluções adoptadas à luz dos princípios enunciados no capítulo anterior.

Uma lacuna que convém desde já referir, é que o sistema não foi desenhado segundo uma especificação formal. Uma aplicação absolutamente rigorosa dos princípios expostos no capítulo 2, obrigaria a que o sistema fosse em primeiro lugar formalizado e que em seguida se derivasse dessa especificação formal o código dos programas que o compoem, demonstrando simultaneamente que a implementação é equivalente à especificação.

Finalmente, é apresentada uma extensa bibliografia anotada.

A generalidade dos pontos apresentados no 2º capitulo constituem de forma sintéctica e informal, um ponto da situação no que se refere ao estado da investigação e desenvolvimento da ciência informática em cada um dos temas a que o título do ponto se refere. Não existe em nenhum deles qualquer contribuição original, à excepção talvez, da forma como estão organizados e sistematizados.

A sistematização do método de tradução de um modelo numa implementação através da aplicação dos 4 princípios apresentação do ponto 2.12:

- Conservação das equivalências semânticas,
- conservação do princípio da boa documentação,
- conservação de segurança expressa no modelo, e
- utilização do sistema de macro processamento

é, enquanto sistematização em 4 principios fundamentais de métodos particulares e dispersos na literatura, uma contribuição parcelar, ainda que muito secundária, à Engenharia de Software.

Todo o capítulo 3 corresponde a uma aplicação viva dos métodos da engenharia de um sistema, com especial realce para o último ponto.

A recolha bibliográfica, extensamente referênciada neste texto, anotada e organizada por temas, é em si mesma, um trabalho preliminar muito útil a qualquer actividade de estudo, investigação ou ensino do tema a que se refere este trabalho.

Este título é de facto mais correcto!

2 - ESPECIFICAR, MODELIZAR E IMPLEMENTAR SOFTWARE E SUA RELAÇÃO COM LINGUAGENS

2.1 - Características do produto software

Quando um Engenheiro Civil projecta uma ponte, numa visão simplista, ele utiliza componentes, por exemplo vigas. São conhecidos os métodos para determinar o modelo abstracto dessas componentes (momentos de enércia, etc.), e as propriedades de todo o sistema (peso total que o tabuleiro pode suportar, etc.). Assim como o inverso, dadas as características da ponte, determinar as características das suas componentes.

Ainda que o engenheiro possa trabalhar absolutamente na base do modelo abstracto da ponte e das suas componentes, é quase sempre possível reconhecer na implementação da ponte essas componentes, assim como todo o raciocínio abstracto do engenheiro é fecundado pela sua percepção física desses objectos.

O plano final da ponte pode ser sempre retido num desenho único relativamente pequeno, seguido eventualmente de outros planos de pormenor.

Infelizmente, os produtos software, ainda que conceptualmente possam ser desenhados da mesma forma têm um caracter radicalmente diferente.

Não há nenhuma pessoa que perante o desfilar do código binário existente na memória de um computador, o qual constitui a única realidade física de um sistema informático, seja capaz de reconhecer as componentes desse sistema, mesmo que tenha um razoável conhecimento da sua estrutura. Um sistema informático é descrito pelas suas componentes principais!

O produto software tem um caracter absolutamente abstracto.

Seria desejável que fosse possível desenhar numa fol-

ha de papel um plano completo de um sistema informático. Infelizmente, na maior parte dos casos não o sabemos ainda fazer.

Seria desejável que conhecessemos os métodos para reter as propriedades abstractas quer do conjunto, quer das suas componentes. Infelizmente na maior parte dos casos não o sabemos ainda fazer.

Por outro lado é necessário ter presente que desenhar e implementar um sistema informático não é o mesmo que fazer um programa.

Um programa que vai funcionar para ser explorado sistemáticamente, por exemplo num contexto industrial, custa cerca de três vezes mais que o mesmo programa feito meramente por exemplo. /2.1.1/. Isto torna-se evidente pois esse programa tem de estar no seu essencial certo e expeperimentado, documentado e apto a ser mantido.

Um programa que vai funcionar como componente de um sistema mais geral custa em média cerca de 3 vezes mais que o mesmo programa funcionando isoladamente ./2.1.1/. Porque esse programa vai interagir com outros programas através de diversas interfaces e tem de ser capaz de resistir a erros de funcionamento existentes no ambiente onde funciona.

Um programa industrial, sub-componente de um sistema, custa cerca de 9 vezes mais que o mesmo programa feito meramente por exemplo ./2.1.1./.

É fácil imaginar que a complexidade de um sistema cresce exponenciamente com o número de interfaces das suas componentes, o que não é necessáriamente o caso dos outros ramos da Engenharia.

No entanto, a situação é contraditória /2.1.2/. O objectivo dos outros ramos da Engenharia são sob certos pontos de vista, em geral infinitamente mais complexos que os

problemas colocados em alguns sistema informático. Basta pensar na infinidade de factores que podem influenciar uma reacção química e que em geral se desprezam pois a prática demonstrou, talvez com alguns desastres, que são desprezáveis. Ora no software, salvo eventuais problemas de concorrência, é tudo determinista!

A forma de atacar e manipular essa complexidade é pois determinar quais as leis que regem o sistema informático e desenhá-lo com essas leis em mente, tal como nos outros ramos da Engenharia.

A dificuldade consiste em que essas leis, são em geral desconhecidas e têm que ser descobertas na sua generalidade.

2.2 - Especificar um problema

As especificações que conhecemos são em 99% dos casos, especificações informais, isto é, não formalizadas matemáticamente.

Quer se trate de especificar um programa, um procedimento, um sistema ou uma componente de um sistema, as suas especificações são do tipo:

Este procedimento insere na tabela um item, mantendo a mesma ordenada pela chave ....

Este programa permite desenhar o gráfico....

XYK34 é uma componente para manipulação de ficheiros, o switch /K permite passá-lo do formato standard para o formato estendido ....

No topo da pilha são colocados os parâmetros que a função passa a aceder indirectamente ....

Etc., etc.

Para compreender a especificação segue-se em geral um processo experimental. Ensaia-se e interpreta-se o resulta

do!

Ainda que as especificações informais sejam o dia a dia da informática, uma das virtudes da chamada "crise do software" foi pôr em evidência que isso é insuficiente. Uma reflexão aprofundada sobre o processo do desenvolvimento do software leva-nos à conclusão de que é absolutamente necessário ultrapassar esta fase artesanal, ainda que por agora não seja realista regeitar qualquer especificação não formalizada, pois trata-se no essencial de um campo ainda em investigação.

Para compreendermos bem o problema, é necessário começar por definir o que é especificar.

Especificar um objecto ou um sistema é descrever as suas propriedades externas. Qualquer pormenor da sua implementação é sob este ponto de vista irrelevante. Assim, especificar uma ponte é descrever as suas propriedades (por exemplo: resistência a sismos, fluxo de tráfego que permite, peso máximo que suporta etc.), sob este ponto de vista a sua estrutura, cor, etc. é irrelevante e em última análise estas propriedades são decorrentes das primeiras, tendo em consideração critérios objectivos (estrutura), ou simplesmente subjectivos (cor).

é descrever as suas propriedades funcionais.

Especificar um sistema informático é formalmente o mesmo, é descrever as suas propriedades essenciais, a forma como está implementado é irrelevante. No entanto a especificação de um sistema informático é muito mais difícil, porque o seu caracter abstracto, exige, para que a especificação seja coerente, completa e simultâneamente concisa, ausente de qualquer consideração experimental ou subjectiva, a utilização de instrumentos formais que a prática vem mostrando se encontrarem nas matemáticas não aplicadas. Dado que muitos sistemas (a generalidade) não pura aplicação em campos não formalizados.

Dúvida: será que o objectivo de tudo formalizar e de uma só vez não é idealista? Que se passa nos outros ramos?

Em contrapartida, não existe tradição de formalização matemática dos campos de aplicação actual da Informática com a excepção do cálculo numérico.

Serão estas exigências um exagero? É fácil concluir que não, se pensarmos que quem coloca o problema, quem especifica, quem resolve o problema e quem se vai servir do produto, podem ser pessoas distintas, com experiências, entendimentos e culturas completamente diferentes.

Este aspecto não é em geral tão dramáticamente importante noutros ramos. É óbvio que desde o Presidente da Câmara ao Automobilista, passando pelo Engenheiro, todos temos uma base comum suficiente para saber para que serve uma ponte e bastam alguns sinais de transito para especificar a interface desta com os seus utilizadores.

Nem sempre é assim!

Tradicionalmente os sistemas ou programas não são especificados desta forma. Não é preciso lembrar os efeitos deste estado de coisas: Manuais que presupõem que o leitor é um perito no assunto, que misturam o essencial com o particular, que não descrevem o efeito de certas opções. Implementações onde os implementadores tomaram opções que não são explicitadas e acabaram por se repercutir na própria especificação.

Postos perante estas dificuldades, poderemos dizer que se trata de um problema não resolvido. Não existem ainda métodos que permitam perante um problema construir a sua especificação. Existem isso sim, diversas propostas aplicadas a categorias de problemas particulares e certos métodos cuja generalidade é ainda duvidosa.

Horning

Outra forma de equacionar o problema é pô-lo na forma: dado um enunciado implicito, como passá-lo a um enunciado explicito? /2.2.1/.

Alguns dos métodos que são hoje em dia explorados são por exemplo: /2.2.1/, /2.2.2/.

1) Utilização da linguagem matemática

Dados a e b, determinar d tal que:

div (d,a) e div (d,b) e $\forall$ x tal que

div (x,a) e div (x,b) então div (x,d)

com div (x,y) <=> x mod y = 0

Isto é d é o máximo divisor comum de a e b.

A utilização da linguagem matemática com toda a sua generalidade é geralmente inadequada devido a que a sua enorme liberdade torna difícil e não automatizável a verificação da coerência, além de que invoca em geral um grande conhecimento matemático sub-entendido. A sua transformação num programa não segue nenhum método sistemático.

2) Utilização de apenas certos sub-conjuntos bem definidos da linguagem matemática

Exemplos:

a) funções recursivas com certas restrições:

O máximo divisor comum de a,b é o resultado da função mdc de $\mathbb{N} \times \mathbb{N}$ em $\mathbb{N}$, definida por:

mdc(a,b) ≡ se b=0 então a senão mdc (b, mod (a,b))

A sua transformação num programa é em geral conhecida.

b) Um sistema, ou uma operação, é uma relação binária entre o conjunto das entradas e o conjunto imagem. /2.2.1/, /2.2.3/.

Uma forma de exprimir esta relação é por exemplo a utilização do cálculo dos predicados da lógica.

F é uma função que aplica o produto cartesiano X^2 no conjunto {sim, não}. X é um dado conjunto de pessoas.

F(x,y) = falso se x e y não são irmãos do mesmo casamento e = verdadeiro se sim.

F(x,y) ≡ pai (x) = pai (y) ∧ mãe(x) = mãe (y)

pai (x) ≡ ...

mãe (y) ≡ ...

ou numa notação próxima do PROLOG:

F (x,y):-pai (x,k), pai (y,k), mãe (x,l), mãe (y,l)

Pai (...,...)

Mãe (...,...)

A passagem deste enunciado a um "programa eficiente" é complexo e não tem método geral.

3) Utilização de equações algébricas, definindo uma algebra.

Relacionar 2 com 3.

Um sistema pode ser visto como definido pelo conjunto de operações que este reconhece. Cada um desses operadores aplica um domínio num contradomínio que se tem que se especificar (sintaxe) e tem um significado (semântica).

Conjunto de pessoas P:

sintaxe: init () → P, cria o conjunto ∅

inserir: pessoa x P → P

retirar: pessoa x P → P

existe?: pessoa x P → {falso, verdadeiro}

vazio? : P → {falso, verdadeiro}

semântica (axiomas):

∀ x_1, x_2 ε pessoa e y ε P

inserir (x_1, inserir (x_2,y))= inserir (x_2, inserir (x_1, y))

inserir (x_1, inserir (x_2, y))= se x_1= x_2 então inserir (x_2,y)

retirar (x_1, init)= init

⋮

vazio? (init)= verdade

vazio? (inserir (x_1, y))= falso

A passagem deste enunciado abstracto a uma implementação é também complexa.

4) Modelo operacional ('Hoare')

Um outro método consiste em ver um sistema definido por um seu modelo matemático, um conjunto munido de uma estrutura algébrica.

Cada uma das operações é especificada pelas asserções que teem de se verificar antes da aplicação da mesma (pré-condições) e pelas asserções que se verificam depois da mesma (post-condições).

Avaliação do interesse deste método.

Conjunto de pessoas P:

P é um conjunto no sentido matemático

sintaxe: .....

semantica:

{retirar (x)} $P=\{x_1,\ldots,x_n\} \wedge n\geq 0 \wedge \forall_1 \leq i \leq n \; x_i \neq x$

isto é

pré (retirar (x)) ≡ Universo, ou seja, pode-se aplicar seja qual for o estado de P.

post (retirar (x)) ≡ $P=x_1,\ldots,x_n \wedge n\geq 0 \wedge \forall_1 \leq i \leq n \; x_i \neq x$

Este método torna, em geral, mais fácil a passagem a uma implementação.

5) Quando se trata de descrever sequências muito estruturadas de dados, um método muito útil e testado, é a utilização de gramáticas.

Uma forma, por exemplo, de especificar um sistema interactivo de gestão de pessoal é:

Especificar o input reconhecido por uma gramática contendo funções semânticas, as quais são por sua vez especificadas por um dos métodos c) ou d).

A passagem de uma gramática ao programa que reconhece a linguagem que esta gera é bem conhecida.

Todos estes métodos serão referidos nos pontos seguintes. (vêr também 2.3 e 2.7).

Um aspecto bastante importante de focar é que: especificar não é resolver um problema.

Em geral um programa numa linguagem algorítmica, representa todas as sucessões possíveis de estados de memória no tempo e que conduzem do input, estado de memória inicial, ao output, estado de memória final (se o programa estiver correcto e não entrar um ciclo, isto é, terminar), sendo cada passo elementar desta progressão desencadeado por cada instrução de afectação.

Todos estes caminhos possíveis, contêm inúmeros pormenores (estados de memória intermédios) irrelevantes para o problema em causa e que são uma estratégia particular para a resolução do problema. Por isso, é de todo desaconselhável utilizar uma linguagem de programação como linguagem de especificação.

Existe no entanto, a tendência de confundir a especificação com um programa numa linguagem de muito alto nível capaz da expressão de abstracções, isto é, capaz de permitir operações sobre objectos complexos definidos pelo utilizador.

Para terminar, convem salientar que a situação ideal da pessoa que especifica, seria dispor de uma linguagem bem precisa (no sentido das linguagens de programação) e bem definida (no sentido matemático) para exprimir a sua análise e dispôr também de um sistema automático de verificação da mesma e que permitisse retirar conclusões da especificação feita. (vêr também 2.11).

2.3 - Modelizar um problema

Modelizar um problema é definir os contornos da sua solução. um sistema razoávelmente complexo exige que se construa um seu modelo. Um modelo pode ainda não ser uma versão operacional do sistema, mas uma sua versão conceptual onde para além das suas propriedades externas são já retidos vários aspectos concretos de uma estratégia de uma dada solução.

não é !?

Trata-se de um passo intermédio entre a especificação e a implementação, onde é possível inferir, demonstrar ou verificar se as propriedades da especificação são ou não conservadas e também tomar decisões sobre aspectos de implementação concreta.

O modelo de um programa ou de um sistema pode ser, desde um simples esquema de programa, até um programa numa linguagem de programação com possibilidades de definições de abstracções.

Como adiante se verá a operação fundamental a aplicar no desenho de software consiste na sua sub-divisão em componentes. A principal operação que é necessário realizar é a operação de abstracção, isto é, definir um objecto pelas suas propridades, independentemente da sua concretização. Um programa ou um sistema é assim desenhado como um encadeamento de acções por uma cooperação sobre estas componentes abstractas. Componentes essas que de seguida serão por sua vez modelizadas por uma aplicação iterativa do método.

No exemplo do sistema interactivo focado atrás, um primeiro modelo do sistema seria conceber a existência de 2 componentes, uma que geria o diálogo e outra que geria o conjunto dos empregados. Repare-se que a especificação não obriga a esta opção, nem a implica, nem esta opção é contraditória com a especificação. Em seguida tomavam-se opções possíveis sobre a organização do conjunto dos empre

gados (ficheiro sequencial, acesso directo, sistema de hash-coding, etc.).

Novamente a situação se repete, subordinação do modelo à especificação...

Essa especificação, seria totalmente diferente desta especificação informal e incorrecta do mesmo sistema: XIZ3 é um sistema que gere ficheiros de empregados organizados em disco da seguinte forma... O que é que a secção de pessoal tem a ver com a organização dos ficheiros em disco?

Outra observação pertinente a fazer neste momento é a seguinte: uma das formas principais por que se traduz a operação de abstracção é conceber uma componente de um sistema como um conjunto do qual apenas são relevantes as operações (incluindo a sua sintaxe e semântica) que o manipulam.

Este tipo de abstração, que é hoje em dia essencial em informática, chama-se um tipo de dados abstracto.

Analisemos em pormenor um exemplo:

Dada uma palavra arbitrária de uma linguagem dada, contar o Nº de vezes que cada terminal aparece nessa palavra e listar como output a sequência ordenada dos terminais seguida do Nº de vezes que aparece no imput.

Especificação utilizando a linguagem matemática arbitrariamente:

input: Seja G uma gramática, T o conjunto dos terminais,
T está munido de uma relação de ordem $<$.
Seja $X \in L(G)$.

output: sequência $\langle (x_1, n_1), (x_2, n_2), \ldots, (x_n, n_n) \rangle$

tal que: $x_i \in T$, $n_i \in \mathbb{N}$

$$\forall 1 \leq i,j \leq n \quad i<j \Rightarrow x_i < x_j$$

Com $(x_i, n_i) \in$ à relação em $T \times \mathbb{N}$ tal que

$\{(x_i, n_i) : x_i$ aparece em X n_i vezes$\}$

Transformação da especificação num modelo.

Existe uma sequência de elementos de T,X:

A operação ler (t) permite tomar cada um dos elementos de X (t), pela ordem como se apresentam em X da esquerda para a direita. A operação Fim tem o valor falso ou verdadeiro conforme se tentou ler para além do último elemento de X, ou não.

Existe uma relação A:T×ℕ que inicialmente tem o valor vazio e que vai sendo transformada pela operação:

inserir (x)

a qual tem o seguinte efeito

Se $\forall (x_i, n_i) \varepsilon A \quad x_i \neq x$ então

inserir (x) tem por efeito transformar A em $A \cup (x,1)$

Se $\exists (x_i, n_i) \varepsilon A$ tal que $x_i = x$ então

inserir (x) tem por efeito transformar A em $A \setminus (x_i, n_i)$ e depois A passa a $A \cup (x_i, n_{i+1})$

A operação listar tem por efeito fornecer a sequência $<(x_1,n_1),(x_2,n_2),\ldots,(xn_1,n_n)>$

em que $(x_i, n_i) \varepsilon A$ e $\forall 1 \leq i,j \leq n \quad i<j \Rightarrow x_i < x_j$

Um modelo da solução que se poderia construir neste quadro seria: t←T;

```
ler (t);
enquanto não Fim fazer
      inserir (t);
          ler (t)
fim fazer;
   listar
```

Um outro modelo que se poderia construir numa linguagem informal poderia ser:

```
program contar;
module input;
defines ler, Fim;
   .
   .
   .
end input;
module relação;
    defines inserir, listar;
        .
        .
        .
end relação;
var t: T;
ler (t);
 while not Fim do
    inserir (t);
    ler (t)
 end while;
 listar
end contar;
```

Em qualquer dos casos o modelo foi construído à custa da definição de 2 abstracções, uma certa visão do input e a relação que se ía construindo. Ambos os modelos permitem verificar que os programas construiam a solução pretendida, pois a relação A vai tomando valores numa sucessão que partindo da relação $\emptyset$ termina com o valor pretendido. Essa sucessão é construida pelo ciclo enquanto, que termina pois X é finito, e em cada passo da iteração se verifica o invariante que se poderia informalmente enunciar por:

$(x_i, n_i) \varepsilon A$ sse x_i aparece n_i vezes na parte já lida de X. (1)

---

(1) *A demonstração seria feita por indução sobre o comprimento de X.*

Como listar é apenas uma mudança de representação de A sem lhe alterar o valor, poder-se-ía ignorá-la na demonstração de que aquele modelo conserva as propriedades essenciais da especificação.

De uma forma informal verifica-se no exemplo como as operações de passagem da especificação ao modelo foram introduzidas:

- divisão em componentes:
  module input,
  module relação.
- abstracção:
  input é defenido por ler e Fim.
  relação por inserir e listar.
- Fixação de uma estratégia de resolução:
  O ciclo constrói o valor final da relação A.

Esta primeira versão poderia por sua vez ir sendo transformada sucessivamente, noutros modelos mais próximos de uma implementação concreta, por uma aplicação iterativa do método. Haveria que seleccionar uma forma concreta de representar o input e uma forma concreta de representar a relação A, um array, uma árvore, um sistema de hash-coding, etc.

Todas estas opções estariam dependentes de outras tantas considerações independentes do enunciado essencial do problema.

Partir de uma especificação formalizada e a partir desta ir construindo sucessivos modelos de programas numa sequência que parte da especificação e termina num modelo operacional (programa) que implementa o sistema, de tal forma que todos os membros da sucessão conservem como invariante as propriedades formais da especificação é por enquanto um objectivo no seu essencial teórico.

Isso não invalida que, mesmo trabalhando de uma for

ma informal, desenhar um sistema não seja percorrer essa sucessão.

Existem bastantes tentativas de passar sistematicamente de uma especificação a um programa ou um modelo de programa.

A passagem de um enunciado recursivo a um programa é hoje em dia razoávelmente conhecida /2.3.1/, /2.3.7/.

A visão de um programa como uma sucessão de estados de memória foi bastante explorada por Hoare. Através de Pré e Post condições é possível estabelecer os axiomas que definem o efeito da aplicação de cada instrução ou operação. A partir deste método foi não só possível axiomatizar a programação, como linguagens /2.3.2/,/2.3.3/.

Nesta mesma linha, a construção descendente de programas a partir de asserções para a sua verificação, está muito desenvolvida com aplicações práticas. /2.3.4/, /2.3.5/, /2.3.8/, /2.3.15/.

O método de especificar um tipo de dados ou uma estrutura de dados através de pré e post condições sobre as operações (descrevendo o seu efeito sobre um modelo matemático do mesmo), foi também muito desenvolvido por Hoare. /2.3.10/, /2.3.12/, /2.3.13/. Uma sua generalização é apresentada em /2.3.14/.

A noção de um tipo de dados como uma algebra, para a qual se definem as operações e os axiomas que descrevem o seu significado semântico, é muito vulgarizada por Guttag. /2.3.17/, /2.3.18/, /2.3.19/, /2.3.20/, /2.3.21/, /2.3.22/, /2.3.23/, /2.3.24/, /2.3.25/.

Os métodos para a transformação de uma especificação algébrica num programa, estão a ser intensivamente estudados, /2.3.27/, /2.3.29/.

Recentemente, para ajudar à modelização de problemas e sistemas e no sentido de ajudar o projectista a se

leccionar quais as representações mais adquadas ao problema, diversas propostas surgiram para associar funções custo às operações de um tipo abstracto /2.3.30/, /2.3.31/.

Uma apresentação conjunta e comparativa dos métodos ditos "à Guttag" e "à Hoare", pode ser consultada em /2.3.32/.

A utilização da lógica como linguagem de especificação não coloca problemas à modelização e implementação, visto que a primeira, uma vez escrita em PROLOG /2.3.33/ é desde logo um programa executável. No entanto ao dar de uma só vez o salto que vai da especificação à implementação, a linguagem tem por efeito introduzir uma estratégia de resolução standard para todos os problemas, o que, aliado a outras faltas de amadurecimento do PROLOG o torna actualmente apenas adequado a uma estreita categoria de problemas.

Uma comparação do método de especificação algébrico, isto é, "à la Guttag", com método de especificação por cláusulas lógicas é feito em /2.3.34/.

A análise de problemas através de gramáticas e a sua transformação em programas é bem conhecida dos "gente da compilação". /2.3.35/, /2.3.36/.

A utilização sistemática de gramáticas para obter programas eficientes, em gamas de aplicações fora da compilação é um campo em desenvolvimento. /2.3.37/, /2.3.38/, /2.3.39/, /2.3.40/, /2.3.42/, /2.3.43/.

Um aspecto extremamente importante a reter é o seguinte: todos os métodos que atrás foram focados têm como objectivo fundamental passar de uma especificação a um programa. No final obtem-se um programa certo! Não é preciso testá-lo, pois demonstrou-se que está certo! Ainda que não se possa afirmar que é impossível aplicar esses métodos ao desenho de Software em larga escala, a verdade

é que esta afirmação está razoavelmente próxima da verdade no momento actual. No entanto, as linguagens modernas que incluem a possibilidade da definição de tipos abstractos e que verificam em tempo de compilação a coerência da utilização desses tipos, são elas próprias sistemas automáticos de auxílio à modelização, pois as abstracção que se fazem sobre os objectos envolvidos no sistema, têm expressão directa em objectos da linguagem, cujo complilador verifica a correção da sua utilização.

→ A semantica operacional nos Prologs não pode nunca ser ignorada (a declaratividade não chega)

Na prática corrente, estas linguagens são o núcleo central de sistema de ajuda à especificação, modelização e implementação de Software. É claro que para tal, elas possibilitam também a expressão de processos concorrentes e fornecem primitivas de sincronização (ver também 2.9, 2.10 e 2.11).

Um contexto ideal onde se deveria desenvolver a actividade de modelização de sistemas é pois a da existência de uma linguagem para exprimir esse modelo, servida por um sistema de ajuda à modelização que verifique da coerência do modelo e que pudesse ajudar à demonstração da verificação das propriedades da especificação (ver também 2.11).

→ Pelo menos no sentido de evitar os erros de distracção!

2.4 - Implementar a solução de um problema

Uma vez obtido um modelo da solução de um problema, das duas uma, ou esse modelo encontra-se expresso numa linguagem de programação disponível, e nesse caso a sua compilação seguida da "linkagem" e "carregamento" são uma implementação do sistema, ou então não se encontra expresso numa linguagem de programação ou esta não se encontra disponível, neste caso é necessário traduzi-lo numa linguagem de programação disponível. (ver também 2.12).

A implementação, em qualquer hipótese, tem de evoluir de acordo com o modelo que implementa e portanto de acordo com a especificação.

É a nível da implementação que se têm de considerar aspectos como a velocidade de execução, volume de dados envolvidos, etc. se estes não foram já claramente estabelecidos e resolvidos ao nível das etapas anteriores.

2.5 - Métodos de desenho de software

O início da década de 70 foi marcada pela palavra chave - programação estruturada.

Desenhar um programa passou a deixar de ser o estabelecimento de um conjunto arbitrário de operações, para se passar a reconhecer que existem padrões tipícos de composição de acções:

Composição sequencial, composição condicional, composição iterativa , composição concorrente.

Um programa deve ser construído como uma composição destes tipos de encadeamento de acções. (Dijk 68)

A principal razão para esta regra não é a não utilização da instrução goto. Um programa estruturado pode ter goto's desde que estes sirvam aquelas composições. A principal razão é que, com excepção da composição concorrente, é possível determinar formalmente quais as condições que se verificam após cada uma daquelas composições, desde que se conheçam as condições que se verificam antes delas.

O mesmo não se pode dizer de um programa construído por um conjunto arbitrário de "goto's" indiscriminados /2.6.10/.

Metodológicamente, um programa que use apenas aqueles padrões de controle, é mais fácil de entender e mais fácil de construir sistemáticamente.

Por outro lado, o método da análise descendente por refinamentos sucessivos (step wise refinement) foi formulado de uma forma sistemática. /2.3.11/, /2.5.1/.

Segundo este método, a análise (estabelecimento do modelo) de um problema é conduzida nos seguintes termos: estabelece-se uma primeira solução numa linguagem abstracta para uma máquina abstracta cujos recursos são óptimos para a solução do problema, (por exemplo a máquina abstracta que permitia percorrer a palavra X (ver 2.3) pelas o-

perações ler e fim, e a máquina abstracta que geria a relação A através das operações inserir e listar) em seguida, essas máquinas abstractas são de uma forma iterativa implementadas sobre outras máquinas abstractas, até se chegar à linguagem de programação disponível (a implementação).

Estas abstracções são em geral expressas em termos de procedimentos, no entanto, o conceito de procedimento como definição de uma operação abstracta introduzida pelo utilizador para expandir a linguagem disponível não é suficiente. É necessário introduzir abstracções estruturalmente mais complexas. Uma máquina abstracta fornece operações sobre uma determinada estrutura de dados. Estamos perante o problema da estruturação abstracta dos dados (Data structures, data abstraction), /2.3.13/, /2.3.12/.

O que é uma estrutura de dados? Trata-se de um conjunto qualquer, munido de operações que o manipulam. Repare-se que este pode por sua vez ser constituído por outros sub-conjuntos.

Mas um conjunto munido de operações é um tipo de dados. Trata-se portanto de uma extensão do conceito de tipo de dados das linguagens de programação. Numa primeira fase, ainda se procurou introduzir nas linguagens, as estruturas de dados tipicas (clássicamente apenas se conhecia o vector). Os trabalhos de Hoare em /2.3.12/ e os trabalhos de Wirth /2.5.1/ desembocaram (por exemplo) na linguagem Pascal. /2.5.2/.

Mas é necessário ir mais longe. É preciso dar a possibilidade de se utilizar no desenho, abstracções arbitrárias, das quais apenas são relevantes as operações que as manipulam, independentemente da sua forma de concretização. Introduz-se assim o conceito de tipo de dados abstracto /2.3.19/, /2.3.22/, /2.3.25/

Assim, chega-se a uma fase em que a concepção de sof

tware é realizada segundo 2 operações fundamentais:

A sua divisão em componentes e abstracção, isto é, a definição dessas componentes pelas suas propriedades relevantes, isto é, as operações características dessa abstracção, seja ela uma máquina abstracta, seja ela um tipo de dados abstracto. /2.5.3/, /2.5.4/, /2.5.5/, /2.5.6/.

Algumas concretizações destes conceitos foram introduzidas nos pontos anteriores. No sistema interactivo de gestão do pessoal, a componente que geria o diálogo é uma máquina abstracta, a componente que geria os empregados através das operações init, inserir, etc., é um tipo de dados abstracto.

Um aspecto que convem desde já realçar, é que sob o ponto de vista estritamente formal não existe distinção entre máquina abstracta e tipo de dados abstracto, pois ambos são um objecto abstracto de que apenas se conhecem as operações que o manipulam. do ponto de vista metodológico sim.

No entanto, sob o ponto de vista metodológico essa distinção é relevante, uma máquina abstracta está muito mais adequada à representação de objectos que interagem com o exterior através de periféricos por exemplo.

O Conceito de máquina abstracta veio também clarificar a construção de sistemas complexos, utilizando 1 ou várias linguagens. Uma outra visão do conceito de máquina abstracta, é o conceito de camada de software, isto é, uma máquina abstracta com as suas primitivas de manipulação, ou seja, as suas operações de acesso.

Diversos sistemas passaram a ser modelizados por camadas concêntricas, cada uma das quais é uma máquina abstracta. Uma visão tipíca deste tipo é a de um sistema de operação constituído por diversas camadas que vão desde a máquina fisíca até à camada da JOB CONTROL LANGUAGE, acessível ao utilizador que tem assim à disposição os ser-

viços prestados pelo sistema /2.5.7/, /2.5.8/.

O conceito de camada de software é uma extensão do conceito clássico de package (por exemplo de sub-rotinas).

A estrutura dos sistemas, concebidos segundo os métodos até agora expostos, é necessariamente, altamente hierarquizada.

Como a evolução dos conceitos, está intimamente ligada à evolução das linguagens nas quais eles se podem exprimir, e onde por sua vez suportamos o nosso raciocínio, é natural que todos estes métodos impliquem mudanças sensiveis na concepção das linguagens de programação - ver o ponto seguinte.

Outro aspecto introduzido pela interação do formalismo e da metodologia é a concepção de módulo. Um sistema é composto por componentes, cada componente tem uma definição, definição essa, constituída por sua vez à custa de outras definições. Um módulo é um agregado de definições, este agregado pode ir desde a definição de um conjunto de constantes à definição de uma ou mais máquinas ou tipos abstractos.

Com os módulos introduziu-se também o conceito de importação/exportação (import/export). Diz-se que um módulo importa as definições que são feitas em módulos que lhe são exteriores e exporta as definições que põe à disposição dos outros módulos.

O Mundo da metodologia da programação e da concepção de sistemas está na mais alta efervescência - ver as referências já feitas e ainda /2.3.6/, /2.3.14/, /2.3.26/ /2.3.41/, /2.5.9/, /2.5.10/.

2.6 - As linguagens modernas de programação - Uma introdução

Uma linguagem de programação reflecte o "estado da arte" da programação na época em que foi proposta.

Os nossos conceitos e raciocinios têm de se exprimir numa linguagem. Essa linguagem condiciona por sua vez a nossa fluência quer de expressão, quer de raciocinio. /2.6.4/, /2.6.16/.

Necessáriamente, toda a evolução na metodologia da programação teria de se reflectir nas linguagens de programação. O Assembler permite exprimir fácilmente o modelo conceptual da estrutura de um computador, nunca da estrutura de um sistema informático.

FORTRAN e COBOL (finais da década de 50) são a pré-história de todo este processo. Nos finais da década de 60, a estruturação de um programa em termos de algorítmos (estrutura do controle) e estruturas de dados por estes manipulados é amadurecida. PL/I, Símula 67, Pascal e Algol 68 são representantes por excelência deste modo de ver as coisas. /2.6.1/, /2.6.2/, /2.5.2/, /2.6.3/.

A partir dos primeiros anos da década de 70, a ideia de que um sistema é concebido por uma hierarquia de abstracções põe com extrema clareza a seguinte questão: Uma linguagem é tanto mais adequada quanto as abstracções usadas para desenhar um sistema ou programa são categorias sintácticas e semânticas da mesma.

Este aspecto, aliado ao facto de que a evolução da compilação, permitiu fazer com que em tempo de compilação a coerência da utilização dessas abstrações seja testada, levou ao aparecimento de toda uma nova geração de linguagens protótipos capazes de servir a expressão das mesmas.

Vê o pont. 2.10

Sob o ponto de vista metodológico e da Engenharia de Software, reconheceu-se também que a linguagem pode

ser um excelente meio de disciplinar a actividade do projectista e do implementador. A total liberdade permitida pelo assembler ou no essencial, permitida por todas as linguagens de "alto baixo" nível é indesejável, pois, a linguagem deve traçar uma via que condicione a forma de expressão dos conceitos, disciplinando o próprio programador. /2.6.5/, /2.6.6/, /2.6.7/, /2.6.8/, /2.6.9/, /2.6.10/ /2.6.20/.

Com estas duas ideias em mente, a saber, uma linguagem deve permitir exprimir nela as abstracções usadas para o desenho de um sistema e "impor" essa via ao seu utilizador, várias equipas universitárias fizeram propostas de novas linguagens de programação.

Que é que essas linguagens trazem de novo?

De uma forma geral a possibilidade da expressão de abstracções complexas do tipo:

- tipos de dados abstractos.
- máquinas abstractas.
- módulos com o conceito de importação/exportação
- conceito de encapsulamento de dados e algoritmos (information hidding)

Algumas delas introduzem também tentativas de facilitar a verificação de programas.

De uma forma geral são providas de mecanismos de verificação em tempo de compilação da coerência da utilização das definições introduzidas (por exemplo "type checking").

ALPHARD, CLU, EUCLID, SETL, MEFIA, MODULA são outras tantas palavras chave que traduzem esse capítulo da história das linguagens, /2.6.11/, /2.6.12/, /2.6.13/, /2.6.14/, /2.6.15/, /2.6.17/, /2.6.18/.

Por outro lado assistiu-se a um crescer do interesse pelas linguagens declarativas, isto é, mais adaptadas à

expressão de problemas que à expressão da sua solução, ou seja, linguagens adaptadas à especificação de problemas como por exemplo o PROLOG /2.3.33/.

Finalmente, resta referir que a necessidade de uma linguagem de programação nova, para substituir no mundo industrial as "velhas" linguagens, foi já reconhecida pelos grandes consumidores de software. Assim o Departamento de Defesa dos E.U.A. lançou o chamado "Projecto DOD", o qual conduziu a um concurso internacional para a definição de uma linguagem, com as características das modernas linguagens de programação e capaz de substituir o COBOL, FORTRAN, etc.

Esse projecto conduziu à aprovação da linguagem ADA, que não sendo uma linguagem protótipo, é já uma passagem a limpo dos conceitos que se encontram suficientemente amadurecidos para terem uma aplicação industrial /2.6.19/. Espera-se que a mesma se encontre disponível nas principais marcas de computador nos anos de 1982/83.

2.7 - Linguagens de especificação

Linguagens de especificação são linguagens de definição de problemas. Um problema especificado numa linguagem deste tipo é explicitado, não resolvido. Assim estas linguagens têm, de uma forma geral, um carácter declarativo e não procedimental!

Não se pode considerar a linguagem matemática, com toda a sua generalidade, como uma linguagem de especificação. É necessário dispor-se de uma linguagem no sentido informático, isto é, com regras precisas e em número relativamente pequeno, para que qualquer afirmação feita tenha um significado preciso e imediatamente evidente, sem qualquer tipo de interpretação contextual.

Discutir com Monteiro

Teóricamente, para uma linguagem com estas características, é possível fazer um sistema automático que verifique a coerência e completude da especificação. Trata-se como é óbvio de temas de investigação.

É neste sentido que por exemplo, a linguagem intodu zida por Guttag ou o Prolog, podem ser entendidos como linguagens de especificação. É também evidente, que todas as considerações de carácter engenheiral que se podem fazer às linguagens de programação, podem também ser estendidas a esta categoria de linguagens, isto é, não basta demonstrar que a linguagem tem uma síntaxe e uma semântica bem defenida e eventualmente potente. É necessário também, demonstrar a sua aplicabilidade, a legibilidade das especificações obtidas, a sua adequação à descrição da maioria dos problemas, a maior ou menor facilidade com que dela se pode derivar uma implementação, a sua adequação à descrição de regras de sincronização, etc.

É sobretudo no campo das bases de dados que se têm obtido maiores avanços, pois neste campo desde há vários anos que a enorme multidão de dados, relações, conjuntos,

etc. em jogo, assim como os elevados custos das implementações, tem colocado na primeira linha o estudo destas questões. /2.7.2/, /2.7.3/, /2.7.4/, /2.7.5/.

De qualquer forma o assunto está muito longe de se poder considerar em vias de arrumado. Inúmeros campos de aplicação da informática têm colocado e sugerido novas possibilidades e técnicas a utilizar. Em /2.7.1/ poderá ser encontrada uma panorâmica.

Sob o ponto de vista da especificação, um aspecto que importa não menosprezar é o facto de as linguagens modernas com as suas possibilidades de expressão de abstrações, poderem permitir obter um modelo tão altamente estruturado que os aspectos operacionais e não relevantes da especificação podem ser reduzidos ao mínimo. Nesse sentido, ainda que não sejam linguagens cuja função natural seja a especificação, permitem na prática obter resultados muito apreciáveis.

Sobre o conjunto destes problemas ver também o ponto 2.11.

2.8 - Características do software

Talvez para lá do ano de 2 000, o engenheiro informático sentar-se-à, com o utilizador, ao lado de um sistema automático e, em diálogo com este, irá construindo uma especificação do problema. No final "carregará no botão" e "do outro lado" sairá o sistema completo, eficiente, optimizado e de acordo com a especificação. Até lá os sistemas informáticos continuarão a ser construídos à mão e ao que parece absorvendo cada vez mais mão-de-obra.

O custo de um sistema informático é uma função complexa de vários factores. De qualquer forma um conjunto de características distinguem um bom dum mau software. A sua definição ainda é bastante dependente do contexto concreto e de aspectos subjectivos, no entanto existe já um razoável consenso. /2.8.1/, /2.8.2/, /2.1.1/, /2.1.2/, /2.3.11/.

- Simplicidade na definição e conforto na utilização
  Um bom sistema deve ser bem defenido, com recomendações para uma utilização adequada. Não devem haver aspectos vagos nem indefinidos. Deve ser fácil de compreender e de utilizar, controlando sempre os erros no seu uso e não ficando, em qualquer hipótese, num estado indefinido.

- Simplicidade, legibilidade e modularidade
  Com vistas à sua construção e manutenção, um bom sistema deve ser fácil de compreender. Deve ser decomposto em parcelas bem definidas e realizando tarefas específicas. Estas componentes devem ser suficientemente pequenas para que possam ser bem compreendidas e a sua documentação deve clarificar o papel que desempenham no conjunto.
  Para estudar e desenhar um programa estudam-se primeiro as suas componentes. Uma componente é sempre definida pela tarefa que executa, nunca pela forma

como a executa. Esta pode em certa fase do desenho ou do estudo ser ignorada. O estudo do programa no seu conjunto, é o estudo da forma como as suas componentes interagem. Pode-se assim controlar o programa ignorando os seus detalhes.
A documentação e a organização do programa ou do sistema, deve ser feita de tal forma, que quem o estude possa recompor o traço da análise por refinamentos sucessivos, isto é, compreenda claramente qual a sucessão de abstracções pela qual o sistema foi modelizado e implementado.
A construção de programas "com truques" é a negação da engenharia de software.

- Adaptabilidade

Um programa de grande dimensão, custa tanto a desenvolver que se pretende em geral explorá-lo durante anos. Durante esse período de tempo novas adaptações vão surgir, geralmente a cargo de outras pessoas. Um programa bem estruturado deve ser bem planeado e bem documentado de modo a facilitar essas adaptações.
A adaptabilidade é preparada pela decomposição e pela documentação. Modificar o programa é modificar as componentes responsáveis pela tarefa que vai ser alterada ou acrescentada.

- Segurança

Compreender logicamente um programa ou um sistema é meio caminho andado para que ele esteja certo. No entanto, ele conterá sempre inevitávelmente erros, a não ser que seja construído segundo um método de demonstração formal, o que evidentemente, não só é em geral irrealista, como se o for à mão, a segurança obtida não pode ser absoluta.

Para além da demonstração formal, as principais técnicas a aplicar são: os testes sistemáticos, os testes em tempo de compilação e os testes em tempo de execução.

Os testes sistemáticos são morosos e caros e como diz uma frase célebre: "eles servem para mostrar a presença de erros, mas nunca a sua ausência".

Os testes em tempo de execução, isto é, os testes gerados directamente pelo compilador, podem-se revelar extremamente caros pois degradam a velocidade de execução.

Os testes em tempos de compilação são portanto os mais adequados ao processo do desenvolvimento do software. É por este motivo que é muito importante dispormos de linguagens de programação nas quais seja possível exprimir directamente as abstracções usadas para resolver o problema. Quando isso acontece, é possível construir compiladores que verificam a coerência daquelas expressões (Exemplo: "type checking", declarações etc).

Por outro lado a linguagem deve ter mecanismos de impor fronteiras a cada abstracção, permitindo especificar claramente os objectos manipulados em cada situação e evitando que uma alteração num ponto, venha a ter repercussões não previstas ("import/export", "information hidding", etc). "side effects"

- Portabilidade

Quando se desenvolve um programa é desejável que ele possa correr em diversas máquinas. Uma forma de conseguir este aspecto é reduzir os aspectos dependentes da máquina e do seu sistema de operação a módulos bem definidos.

Outro aspecto muito importante, é dispor de

linguagens bem definidas e sem ambiguidades. Um bom sistema não deve utilizar indiscriminadamente possibilidades não standard de uma determinada instalação. Quando se trabalha numa linguagem com deficiências de standardização ou zonas pouco claras, nunca se deve utilizar toda a linguagem na implementação, mas, restringirmo-nos àquelas zonas estáveis e bem conhecidas, /2.8.3/, /2.8.4/.

- Eficiência

A eficiência reduz os custos da computação. Dada a actual descida abrupta dos custos do hardware é perigoso sacrificar as restantes qualidades em nome desta.

É muito frequente entre nós, a ideia errada de que um programa eficiente tem de ser codificado em assembler.

Por um lado é necessário separar claramente o processo da análise, do processo da codificação. Um programa pode ser "analisado em Alphard e codificado em assembler".

No entanto é preciso ter presente que além do assembler ser a antitese da portabilidade, um programa cuja especificação caiba numa folha de papel A_4 e cujo modelo expresso numa boa linguagem de alto nível, caiba em 10 folhas de papel A_4, pode expandir para assembler em mais de uma centena de páginas de A_4.

O carácter sintético das linguagens de alto nível, facilita a análise de conjunto dos programas, logo facilita a optimização global, enquanto que o assembler facilita a optimização local. Esta última espécie de optimização, é em geral realizada eficientemente por um bom compilador optimizante.

De qualquer forma, quando a complexidade de um programa é muito grande, as zonas significativas sob o ponto de vista da eficiência são restritas (10 a 20%, é nelas que se deve investir.

Até à alguns anos, o último reduto do assembler eram as instruções de controle directo do hardware. Também neste campo este começa a ser realisticamente batido - ver por exemplo, as linguagens MODULA e ADA.

2.9 - Linguagens e abstracção

1- Abstracção de operações

A abstracçãode uma operação pode-se em geral exprimir essencialmente por 2 vias: procedimentos e funções.

Um procedimento é da natureza de uma instrução. A sua execução é equivalente à expansão "in line", do código do seu corpo, com as relevantes substituições realizadas, isto é, decorrentes da sua interface (parâmetros formais e actuais).

Uma função é da natureza de uma expressão, isto é, a sua activação denota o valor que calcula em função dos seus parâmetros actuais.

As linguagens modernas tendem a sistematizar esta distinção de natureza e a sofisticar as possibilidades.

Assim, ADA por exemplo, proibe explicitamente que a função introduza qualquer tipo de efeitos laterais (side-effects), que se traduz na exigência de que todos os seus parâmetros são de entrada e não é possível dentro do corpo da função afectar valores a outras variáveis que não as variáveis locais. Deste modo não só se força que a abstracção que a função implementa seja expressa, como se restringe as possibilidades da expressão, a essa mesma abstracção.

Os side-effects são perniciosos porque impossibilitam, uma compreensão clara e independente do contexto.

Assim, dada a definição matemática de f(a,b):

f(a,b)+C é sempre igual a C+f(a,b)

No entanto, se os side-effects estiverem presentes, esta afirmação pode não ser verdadeira, pois, nada nos garante que um side-effect de f não seja modificar o valor de C, complicando extraordinàriamente a compreensão e / ou verificação do programa.

ADA, dá a possibilidade de introduzir "funções" tendo por efeito a modificação de variáveis globais, nomeadamente a

"aliasing"

través de parâmetros formais de tipo variável ou por side-effects, mas, apenas possibilita a expressão da abstracção deste tipo de "funções" através dos chamados procedures que devolvem um valor.

Por outro lado, a forma de passagem dos parâmetros num procedimento são hoje em dia bastante sofisticados, quer no sentido da correcta explicação da interface que implementam, quer ainda, no sentido de uma implementação extremamente eficiente a nível de código objecto.

Ainda em ADA, por exemplo, um parâmetro formal pode ser classificado por:

in - parâmetro formal de entrada.
É proíbida qualquer afectação ao mesmo dentro do corpo do procedimento.
O parâmetro efectivo é uma expressão, que é calculada uma e uma só vez à entrada.

out- parâmetro formal de saída.
É proíbido que este parâmetro figure numa expressão no corpo do procedimento. É controlado se lhe é afectado um valor no corpo do mesmo. O parâmetro efectivo é uma variável (em geral passada em endereço)!

in out- Parâmetro formal de entrada saída.
Parâmetro efectivo é uma variável (em geral passada em endereço)! Total liberdade de manipulação do pârametro formal.

Os parâmetros de entrada, podem ter valores por dedefeito, os quais são assumidos numa forma de activação especial como valores dos parâmetros actuais.

Essa forma de activação especial permite omitir parâmetros de entrada, assim como especificar a correspondência parâmetro formal parâmetro actual, explicitamente e não só por posição.

2- Abstracção por nomeação

Um outro tipo de abstração que se pode exprimir em todas estas linguagens é a definição de um valor ao qual se associa um nome. A presença do nome em qualquer expressão é um substituto para o próprio valor, que pode ser simples, por exemplo:

```
Pi = 3.1415 ...
```

ou complexo, por exemplo:

```
inicialização = (o, o, o, o, o, o,)
        : array [1 ..6] of integer
```

A nomeação de uma constante é particularmente útil e segura, sobretudo tendo em atenção futuras modificações.

Recursividade

Trata-se de um mecanismo de abstracção muito potente, também presente em todas as linguagens modernas. Ele pode ser aplicado a algorítmos mas, infelizmente, ainda não pode ser geralmente aplicado a dados.

3- Parametrização ou unidades de programa genéricas

Em geral, a parametrização é uma forma de dar ainda maior potência a uma abstração. Os procedimentos são parametrizados, pois implementam uma operação abstracta sobre os seus parâmetros. No entanto, a forma clássica de parametrização de procedimentos é ainda muito pobre, pois, exige uma adequação demasiado restrictiva dos parâmetros actuais aos parâmetros formais.

Matemáticamente, a definição de produto cartesiano é independente do número de componentes de um vector, assim como da sua representação (através de um vector no sentido informático ou de um record por exemplo).

A forma clássica de implementação de procedimentos não é compatível com esta generalidade.

Por outro lado, as abstracções que são modelo de um

sistema são em geral parametrizadas em função de outras suas componentes /2.9.30/. Por exemplo, uma pilha, é uma estrutura abstracta cuja caracterização funcional enquanto tipo de dados abstracto é parametrizada pelo tipo das suas componentes: pilha de inteiros, pilha de reais, pilha de pilha de inteiros, pilha de conjuntos de array's, etc. Seja qual for o tipo das componentes, o conjunto de axiomas que caracterizam a pilha é constante se for parametrizado pelo tipo das componentes.

As linguagens modernas permitem a expressão de abstracções genéricas parametrizadas não só por valores, mas também por outros tipos.

A parametrização por valores é já bem conhecida. Por exemplo em Pascal:

```
const  min =     0;
       max = 1 000;

var A: array[ min .. max ]of integer;
```

A parametrização por tipos permite por exemplo tipos abstractos genéricos. Isto é, definir uma fila independentemente das suas componentes. Tal é possível nas linguagens ADA, Alphard, CLU, etc. /2.9.31/.

Um problema mais complexo é o da expressão de operações de percurso do conjuntos definidos genéricamente /2.9.23/.

De qualquer forma, a parametrização de componentes é ainda um tema de investigação se se pretender associar esta à verificação em tempo de compilação das instanciações concretas dessas componentes para um dado parâmetro efectivo. /2.9.31/.

Possivelmente, no futuro, muitos outros tipos e formas de objectos poderão ser parametrizados pois a prática da modelização e especificação mostra a sua necessidade

4 - Expressão de processos concorrentes e sua sincronização

As linguagens modernas tendem a permitir a expressão de processos concorrentes e respectiva sincronização.

Assim Modula e Concurrent Pascal permitem exprimir os conceitos de processo e monitor, Path Pascal /2.9.32/, os conceitos de processo e Path expression e ADA os conceitos de processo e "rendêz-vous", por exemplo.

Abstracção através de tipos de dados

Deixâmos para o fim, propositadamente, a expressão de abstracções de tipos de dados por ser esta forma de expressão que maior atenção tem merecido nos últimos tempos.

O conceito de tipos de dados simples, foi rápidamente estendido para o conceito de tipo de dados estruturado (Pascal, Algol 68, PL/I, etc).

No entanto, nestas linguagens, a forma de estruturação é sempre uma combinação de mecanismos de estruturação pré definidos e fixos (array, record, set, ficheiro,etc). A definição de estruturações ou abstracções de dados de complexidade arbitrária é vedada. Por exemplo, em Pascal não existe nenhuma forma de definir o tipo de dados pilha, munido das operações push, pop, top, empty.

de forma segura e adequada.

No entanto, é possível exprimir esta abstracção através de um array, uma variável global e 5 procedimentos (init, pop, top, push e empty). Só que nada impede o programa de manipular directamente esta representação, por outra via diferente daqueles procedimentos, destruindo a coerência da mesma.

Havia pois que encontrar uma forma de definir tipos de dados (conjunto de valores, eventualmente estruturados, munido de um conjunto de operações de manipulação) de complexidade e estrutura arbitrária com o mesmo grau de segurança e verificação que um tipo de dados pré-definido.

A solução deste problema foi conseguida por 2 vias essenciais, através de mecanismos de protecção (information hidding) e através da abstracção de dados (Data abstraction).

O que é essencial às 2 soluções é que a abstracção é uma caixa negra da qual apenas interessa ao utilizador a forma como tem acesso às suas operações e a respectiva especificação, por exemplo, uma pilha é uma estrutura abstracta cujas operações de acesso são init, push, pop, top e empty, e ainda mais importante, é-lhe vedado qualquer outro tipo de acesso à caixa negra. O implementador da abstracção, pelo contrário, tem uma visão interna da caixa negra, isto é, da forma como esta está implementada.

Assim, nas linguagens com possibilidades de information hidding essa expressão é feita através do conceito de módulo (Modula-module, Euclid-module, ADA-Package). Nestas linguagens um módulo aparece como um conjunto de definições (tipos, constantes, variáveis, procedimentos), alguma das quais podem ser exportadas, isto é, colocadas à disposição do utilizador. O módulo por sua vez, pode importar definições do ambiente onde é declarado.

Assim, para definir um tipo de dados, define-se um módulo que exporta um conjunto de operações, as operações de manipulação, e um tipo, o tipo no qual é representado o tipo de dados. Internamente ao modulo são representadas outras operações, estruturas, etc. auxiliares. O utilizador pode declarar variáveis do tipo exportado, mas estas só são manipuláveis pelos procedimentos exportados.

O módulo pode no entanto não exportar nenhum tipo, neste caso ele pode servir qualquer outro objectivo de estruturação do desenho, por exemplo, definição de um agregado de constantes que parametrizam um sistema.

A diferença fundamental nas diferentes realizações

é a potência com que o conceito é implementado. Se se permite parametização, instanciações várias, grande sofisticação das interfaces, explicitação dos pontos de vista do utilizador e do implementador, etc.

Em ADA, por exemplo, é possível exportar procedimentos cujos identificadores assumem a forma da notação convencional de operadores: "+", "*" etc.

As variáveis do tipo exportado podem então figurar em expressões construídas com a notação convencional a+b*c.

b) Nas linguagens cuja orientação é a abstracção de dados, um módulo é sempre e apenas um agregado de dados e operações. A instanciação múltipla existe e o efeito da definição é sempre permitir a declaração de variáveis cujo tipo é o tipo assim definido e cujos operadores são as operações também assim definidas. As realizações são múltiplas (Simula e Concurrent Pascal-Class, Clu-cluster, Alphard-form). As diferenças essenciais residem no run-time system exigido, na potência da expressão e no grau de sofisticação com que a especificação e a representação do tipo são explicitadas.

Históricamente, tem interesse referir que todas as expressões de abstracções de informação têm origem no conceito de class em Símula, o qual, foi sucessivamente passado a limpo nas novas propostas.

2.10 - Linguagens, fiabilidade e segurança da programação

De que forma se pode garantir a segurança de um sistema? Concebendo-o por componentes correspondentes às diferentes abstracções necessárias ao desenho, verificando cada uma das componentes e verificando o conjunto.

De que forma as linguagens podem ajudar a segurança? Permitindo a expressão das abstracções envolvidas (ponto 2.9) e testando a coerência dessa expressão.

Os erros podem ser introduzidos por muitas vias. Desde a especificação, à incorrecta utilização dos módulos, passando pela incorrecta codificação e compreensão da linguagem, até aos erros por distração, tudo é possível.

Assim, um dos principais princípios de detecção de erros por via das linguagens é o princípio da redundância. Para que o compilador possa verificar o código é necessário introduzir redundâncias. Por exemplo, a declaração de uma variável é muitas vezes redundante com a sua utilização, pois, pode-se deduzir da sua utilização, qual o tipo a que pertence. É o que faz o FORTRAN com as declarações implícitas que muitos "quebra cabeças" criam aos programadores. Pequenas variações no léxico dos identificadores, podem conduzir a erros muito difíceis de detectar, se o compilador assume declarações implícitas.

Uma linguagem deve ser simples e conter um número mínimo de conceitos, a partir do qual seja possível ir criando estruturadamente conceitos mais complexos mas partindo sempre, na medida do possível, de um conjunto mínimo inicial. A Simplicidade é uma propriedade muito importante porque facilita a compreensão e a utilização.

Por outro lado, conceitos diferentes, devem ter formas de expressão diferentes para pôr em evidência as diferenças.

A linguagem não deve dar possibilidade de expressão

sem qualquer tipo de limitações. Assim, esta facilita a imposição de regras de estruturação e restringe a possibilidade de se criarem expressões "potencialmente geradoras de erros". /2.10.2/.

Exemplos de possibilidades proibídas em algumas linguagens modernas, no sentido de incrementar a segurança da programação:

- proíbição de goto's, /2.6.9/.
- proíbição da introdução de side-effects.
- Dar a possibilidade da passagem de parâmetros explicita e não por posição (ADA).
- Não existência de conversões implicítas ou standard entre tipos - levando sempre o type checking às últimas consequências.

A outra forma através da qual as linguagens podem ajudar à detecção de erros é a exigência da clara definição de todos os objectos manipulados. Assim, a definição de tipos de dados, parametrização de procedimentos, assim como regras de coerência precisas, muitas das quais podem ser testadas em tempo de compilação, permitem analisar a coerência das construções expressas e verificar em tempo de compilação se aquelas regras são mantidas coerentes (type-checking, parameter-checking, etc).

A verificação destas coerências deve sempre ser testada, na medida do possível, em tempo de compilação. Algumas daquelas construções ainda não se podem testar em tempo de compilação, por exemplo, a garantia de que uma variável se mantem no intervalo do tipo a que pertence (range-checking). Em torno destas questões existe hoje um grande trabalho de investigação, tentando introduzir nas linguagens possibilidades de dedução estática se aquelas propriedades são mantidas. Trata-se, digamos assim, da tentativa de mecanizar certos aspectos de verificação formal /2.6.10/, /2.10.4/.

Outra forma de ajudar à verificação dos programas é introduzir asserções. Uma asserção é uma condição que tem de ser verificada. Em tempo de execução, sempre que o controle passa pela asserção, a condição é testada, se falhar, o programa é abortado e uma mensagem emitida. Estes métodos, são pesados pois degradam muito o tempo de execução. Um grande trabalho de investigação é também feito no sentido de tentar deduzir estáticamente se as asserções são ou não verificadas. De qualquer forma, ter a possibilidade, de compilar opcionalmente o programa, com geração de código para teste do range-checking e das asserções, pode já ser uma grande ajuda.

Um outro exemplo de segurança, é a verificação pelo compilador, se uma variável é utilizada numa expressão antes de ser inicializada. Outra forma de obviar a este tipo de erro, é definir valores de inicialização pré-definidos para todas as variáveis, em função do tipo a que pertencem, o que nem sempre é aconselhável.

Uma outra forma de a linguagem ajudar à segurança da programação é a introdução das possibilidades de estruturas de controle sofisticadas, assim como de estruturação de componentes, procedimentos, tipos de dados, módulos, processos, etc.

Estas possibilidades tendem a permitir exprimir na linguagem os modelos das componentes dos sistemas sem a necessidade de o programador os implementar por vias "atravessadas", assim, o programa reflecte a organização do desenho e nesse sentido é mais apto à verificação, compreensão e manutenção /2.6.20/.

A modulorização, nomeadamente, é um dos métodos mais sofisticados de evitar a proliferação de variáveis globais com o seu consequente perigo por falta de segurança, /2.10.3/, /2.6.5/, /2.9.17/.

Mais recentemente, algumas linguagens permitem a especificação de "exception handlers", isto é, blocos ou módulos que entram em execução, se uma determinada excepção (erro) foi detectada, /2.10.2/. Estas excepções podem ser desencadeadas pelo hardware (por exemplo um overflow, uma pane) ou pelo software (por exemplo, range failure ou asserção falhada). Um exception handler permite especificar qual a acção a desenvolver quando a excepção é desencadeada, assim como a forma de continuação, aborto ou resincronização do processamento em curso. (ADA é um exemplo de linguagem com este tipo de possibilidades).

Finalmente, é também necessário referir as linguagens cujo desenho facilita a verificação manual ou mecânica da verificação formal da correção do programa (Euclid e Alphard por exemplo), /2.6.14/, /2.9.13/, /2.9.23/, /2.9.28/.

2.11 - Sistemas de ajuda à especificação e à programação

A evolução da engenharia de software, em paralelo com a metodologia da programação e as linguagens de alto nível, levou à entrada em cena dos chamados sistemas de "ajuda à especificação" e dos chamados "sistemas de ajuda à programação", ("programming environment").

Os sistemas de ajuda à programação não são uma novidade total, um bom compilador, servido por um bom editor e um sistema de time-sharing, são um ambiente de trabalho absolutamente imprescindível para o desenvolvimento eficiente de software.

A novidade consiste na sofisticação dos meios. Um sistema de ajuda à programação, tal como começa hoje em dia a ser desenhado /2.11.1/, /2.11.2/, é constituído por um sistema interactivo cujo núcleo central é um compilador de uma boa linguagem, (o compilador deverá ser incremental de preferência), um bom editor de texto e uma base de dados de componentes de programas.

A maioria das linguagens modernas permitem a compilação separada das unidades de programa (procedimentos, módulos, tipos de dados abstractos, etc). Estas unidades de programa são compiladas, arquivadas na base de dados e catalogadas. Através de uma linguagem de interrogação, o programador pode aceder às unidades de programa arquivadas, conhecer as suas interfaces externas, realizar buscas de quais as unidades cuja interface é tal, ou cuja semântica é tal.

Um programa é composto pela assemblagem de um conjunto de unidades compiladas separadamente. O processo da assemblagem é controlado pelo sistema sendo a sua coerência testada. Não existe duplicação de código fonte entre programas.

Em sistemas mais sofisticados, a base de dados é hie

rarquizada e a informação nela contida sujeita a regras de privacidade. Isto é, o programador tal pode usar o módulo tal, mas não pode alterá-lo, pois este é da responsabilidade de outro programador que responde pela sua coerência. Um processo automático de encaminhamento de mensagens entre programadores, permite avisar os utilizadores de um módulo das eventuais repercussões de uma alteração que lhe faz o seu responsável.

Noutros sistemas, utilizam-se editores especiais que reconhecem a sintaxe da linguagem, estes editores impedem, por um lado, que se escreva código sintáticamente errado, e por outro, impõem regras de documentação, formatação dos programas, etc. Em geral eles são o "front end" de um compilador incremental /2.11.3/.

No sistema poderão finalmente ser introduzidos utilitários de ajuda à programação, é o caso da utilização de filtros e verificadores que asseguram certas normas da programação /2.8.4/ ou o caso de utilitários que transformam programas recursivos em iterativos, ou ainda programas escritos sobre certo compilador, para programas aceites por outro compilador /2.11.4/. UNIX

Os sistemas de transformação de programas também se podem incluir nesta categoria. /2.11.5/.

Os sistemas de ajuda à especificação, são sistemas com contornos semelhantes, mas onde a linguagem é uma linguagem de especificação. O sistema ajuda à verificação da coerência da especificação, assim como à sua transformação num modelo operacional /2.11.6/, /2.11.7/.

Certos sistemas misturam informalmente as funções de um e de outro tipo de sistemas, já que como referimos atrás, as linguagens modernas tendem a ser utilizadas muitas vezes como linguagens de especificação /2.11.8/.

Finalmente, convem também lembrar que os sistemas de

ajuda à construção de compiladores, conhecidos desde há mais de 10 anos, são também sistemas de ajuda à especifi_ cação e programação. Para o desenho de um compilador, a sintaxe da linguagem, formalizada e expressa em termos de BNF por exemplo, é uma linguagem de especificação que um sistema especial (Parser checking and generation) analisa e transforma num programa (Parser). Infelizmente, o mesmo não se pode dizer (por enquanto) da semântica das linguagens de programação.

2.12 - E quando alguns constrangimentos impõem a utilização de linguagens de baixo nível?

Em certas circunstâncias, alguns constrangimentos impedem que a implementação seja feita em linguagens de alto nível. Excluindo à partida o critério eficiência (por demasiado simplista) a implementação poderá ter de ser feita em assembler ou num seu derivado. São exemplos mais correntes desta circunstância, não se dispor de outra linguagem ou as linguagens de que se dispõe, não permitem o acesso a certos aspectos baixo nível do computador em questão (ver também o ponto 3.2).

Noutras circunstâncias, critérios de portabilidade, interface com outros software ou ainda de interface com o sistema de operação poderão impor o FORTRAN ou o COBOL como linguagem a utilizar.

Nestes casos, deve-se mesmo assim, trabalhar num modelo expresso numa linguagem moderna e se o sistema for complexo e a linguagem estiver disponível, eventualmente noutro computador, então o modelo deve ser suficientemente pormenorizado para ser compilado, pois, este processo, ajudará à deteção de muitas das suas incorreções. Em certas circunstâncias, este aspecto poderá ser levado mais longe, tornando o modelo operacional e simulando o seu funcionamento noutro ou no mesmo computador. Este parece-nos ser o método por excelência para desenvolvimento de software para máquinas nuas, isto é, sem sistema de operação próprio.

Em seguida, o modelo deve ser traduzido à mão na linguagem de programação disponível.

Para executar tal tradução deve-se estabelecer uma cuidadosa norma de tradução do modelo na implementação. Os princípios base para desenhar essa norma são:

→ Conservação das equivalências semânticas, isto é, as abstrações expressas no modelo têm uma certa

semântica. A cada uma delas corresponderá na implementação, uma estrutura de código standard que conserva invariante o significado semântico da mesma.

→ Conservação do princípio da boa documentação, isto é, o modelo exprime de uma forma mais concisa, a implementação final, esta deve ter uma documentação que, acompanhada do modelo, conserve como invariante o princípio da boa documentação, a saber, a documentação permite recompor fácilmente o traço da análise descendente por refinamentos sucessivos. Neste sentido o modelo acompanhará o software que modeliza para a sua fase da manutenção.

→ O standard deve ser feito de tal forma que a segurança da programação, conseguida no modelo, seja conservada na implementação na medida do possível.

→ Nas linguagens dispondo de sistema de macro processamento, este deve ser utilizado ao serviço de uma correspodência, o mais efectiva que possível, entre o código do modelo e o código da implementação. /2.12.1/, /2.12.2/.

Uma aplicação deste método encontra-se descrita no Capítulo 3.

Em /2.8.2/, no capítulo final, encontra-se um exemplo da aplicação de um princípio semelhante, para desenho de um "Real-Time Scheduler".

Em /2.12.3/, descreve-se uma técnica semelhante.

Um aspecto a realçar é que, tanto quanto possível, a linguagem seleccionada não deve ter um run-time system, relevante no modelo, cuja concretização seja demasiado complexa e inadequada. Um bom exemplo de características negativas deste tipo, é a utilização da classe do Simula para concretização de tipos abstractos utilizando as declarações VIRTUAL e demasiadas instanciações dinâmicas.

Outro aspecto a tomar bastante em consideração é a utilização explicíta, das primitivas de sincronização da linguagem de alto nível. É claro que em todo o procedimento de carácter metodológico, o bom senso constitui sempre um aspecto a não menosprezar.

Outro exemplo de técnica a aplicar, ainda no contexto deste capítulo, é a utilização de prê-processadores /2.12.4/ ou de normas standard de expressão das figuras de controle através de branch's e goto's nas linguagens onde elas não estão presentes. A programação sistemática de sistemas interactivos com instruções LEAVE é de todo recomendável.

2.13 - Condução do projecto Software

Fazer software com características industriais, é muito diferente de fazer um programa, mas quando a dimensão do mesmo ultrapassa a "dimensão de 1 gabinete", isto é, quando o número de pessoas envolvidos ultrapassa a dimensão individual, para entrar na dimensão de uma equipa, as coisas mudam ainda mais de figura.

Em /2.1.1/, /2.13.1/, /2.13.2/, é feito um estudo exaustivo destes aspectos. Para alguns deles convem chamar a atenção.

O primeiro aspecto a ter em consideração é que o tempo de execução de um projecto não é igual ao tempo total requerido por 1 pessoa a dividir pelo número de pessoas que nele trabalham. Isto só seria verdade num sistema em que cada pessoa trabalhasse numa componente e estas fossem totalmente disjuntas.

A interacção entre as componentes de um sistema, leva à necessidade de uma profunda interacção entre os membros de uma equipa. Juntar abruptamente mais pessoas à equipa, para resolver os problemas de calendário, pode redundar numa catástrofe por se gerar um processo de "feed-back" positivo que pode conduzir a um aumento incontrolável da confusão.

Daqui se retira também a conclusão de que a interface entre as componentes dum sistema deve merecer uma especial atenção no processo da sua modelização e ser reduzida ao mínimo.

Outro dos aspectos a reter, é que as equipas devem ser altamente hierarquizadas. No topo está um número mínimo de pessoas que especifica e concebe a estrutura fundamental do modelo. É necessário partir de um número relativamente pequeno de pessoas, porque estas têm de ser as mais qualificadas (o peso das suas opções é enorme) e têm de ter

uma visão de conjunto.

Especificar um sistema e conceber a sua estrutura fundamental é um processo que envolve sempre uma notável experiência, capaz de avaliar as repercussões das decisões a tomar.

No entanto, a concepção de que existem aqueles que analisam a especificação, aqueles que tomam opções de concretização e aqueles que codificam, é uma visão errada e perigosa para a qualidade final do produto.

Os condutores de projectos de software têm de ter a capacidade de desempenhar qualquer daquelas funções, para poderem avaliar correctamente o peso das suas decisões e saberem ouvir e dirigir aqueles que executam tarefas com menor grau de liberdade.

O conjunto da equipa é visto como uma hierarquia em que a linguagem é semelhante e de preferência única. O método por excelência de comunicação não é uma montanha de papéis abstractos, o papel dos responsáveis não é produzir dossiers e dossiers de directivas capazes de serem entendidas por um codificador, mas o papel de produzirem directivas ajustadas, e de empreenderem o diálogo no sentido de elevar os dirigidos à compreensão das mesmas e à sua concretização de uma forma viva.

O trabalho numa equipa deve ser muito bem preparado e só se passa à execução quando o processo do desenho amadureceu suficientemente. Não se deve ceder à pressão dos "homens de acção imediata", os implementadores devem aproveitar estes tempos mortos, para afinarem as suas ferramentas, conhecerem bem as linguagens em que trabalham, implementando instrumentos de ajuda, etc.

A condução do projecto de software exige circulação de documentos, estes devem ser constituídos com um editor de texto e estarem sempre com uma versão única acessível a

toda a gente. As suas alterações são inevitáveis. Os documentos dactilografados, posteriormente cheios de anotações em geral incompreensíveis, são substituídos por versões em disco às quais se anexa um histórico de alterações.

A equipa tem uma estrutura que se deve adaptar à estrutura do próprio sistema em construção. A condução e controle do trabalho deve ser feita segundo um planeamento que inclui: período para terminar a especificação, período para desenho e codificação e período para testes. os testes devem ser conduzidos segundo cadernos de testes a elaborar pelos responsáveis, e os módulos, desenhados segundo o princípio da desconfiança mútua, isto é, testando, pelo menos em modo "debug", os pressupostos que a especificação faz sobre as interfaces com os outros módulos. O controle não deve ser vago. Uma das formas principais de evitar que "grão a grão" se vá criando um intolerável atraso sobre os planos previstos, é executar controles onde não se podem admitir frases como:

- Planeamento 90% acabado, etc.
- Debugging 99% completo, etc. (?!)

É necessário controlar na base de:

- Especificações assinadas e consideradas definitivas por todos.
- Código fonte 100% completo, editado e arquivado.
- A versão actual passa todos os testes do caderno.

Dentro da equipa, para além da hierárquia e da divisão de trabalho que reflecte a estrutura do sistema em construção, deve haver uma outra divisão de trabalho("the cirurgical team "). À frente do trabalho estão aqueles que dirigem, isto é, os programadores principais, assistidos por programadores assistentes, (se quizerem escrevam programador com letra grande, mas, desde o princípio deste texto que a única coisa de que estamos a falar é de PROGRAMAÇÃO). Ambos são assistidos por responsáveis de Tarefas especializadas:

- o especialista na linguagem que se está a usar.
- o responsável de catalogação das versões e testes, e seu arquivo.
- o responsável pelas ferramentas (software) de apoio.
- o responsável pelo trabalho administrativo (edição e actualização da documentação).

Evidentemente que várias destas funções podem estar concentradas numa única pessoa.

Julgamos oportuno fazer aqui a seguinte observação.

Modernamente, o processo do desenvolvimento do software é visto como conduzido por especialistas altamente multifacetados e capazes de trabalharem desde a especificação à codificação e teste. A divisão artificial das funções, e sobretudo o facto de se colocar na condução de projectos de software, pessoas sem a formação adequada, leva a que se distorçam as possibilidades da informática, quer rebaixando as suas potencialidades, reproduzindo no computador modelos em geral mais adequados a tratamentos manuais, quer rebaixando as funções dos programadores a meros codificadores, em geral pouco atritos a mudança e inovações.

O resultado final é um software de péssima qualidade e custos altíssimos, uma proliferação notável de "métodos de tirar rendimento da mediocridade", etc. Este estado de coisas deverá ser no essencial modificado colocando a INFORMÁTICA a dirigir a informática, isto é, utilizando no essencial os métodos expostos neste texto e acabando com a condução de projectos informáticos por pessoas de formação inadequada, possibilitando simultaneamente a exploração das reais possibilidades dos profissionais.

Este aspecto é ainda mais crítico a nível da especificação. Em geral, passar de um caderno de encargos, oralmente expresso por um "cliente", à especificação funcio

nal de um sistema é um processo complexo onde a capacidade do informático é decisiva, pois:

- O "cliente", em geral não conhece nem o seu problema rigorosamente, nem as possibilidades da informática para resolução (e mesmo muitas vezes explicitação e racionalização) do mesmo.
- O "cliente" tende a reproduzir no sistema informático a sua forma de proceder manual.
- O "cliente" não é em geral capaz de distrinçar um modelo abstracto e funcional do seu mundo real, de uma montanha de pormenores que apenas "parametrizam" esse mundo real (frequência de tratamento, constrangimentos de tempo, etc).

Assim, tornar o informático no essencial dependente desta visão, é errado e tem tido por resultado principal a tentativa de reprodução no computador das formas de tratamento manual dos problemas, não utilizando as possibilidades e a oportunidade da informática para racionalização e enriquecimento de uma actividade em geral feita manualmente. A actividade principal do informático, deve ser nessa fase, ajudar o seu "cliente" à explicitação desse modelo abstracto e funcional do seu mundo real.

Para a correcta condução desta actividade, ainda durante a especificação, pode ser feita uma implementação "à la gardere" desse modelo para que o "cliente", em conjunto com o informático, possa aferir da adequação do mesmo. Só depois de um acordo estabelecido em torno do comportamento do modelo é que se passa a uma fase em que se consideram todos os outros constrangimentos (volumes e frequências dos tratamentos).

Esta forma de funcionamento tem um efeito extremamente "pedagógico" sobre o "cliente", ao mesmo tempo que ajuda o informático a separar toda a fase de análise funcional da fase de modelização e implementação definitivas.

3.- DESENHO DE UM SISTEMA CONCRETO - ESTUDO DO MONITOR GRÁFICO A INSTALAR NO GT42.

3.1 - Apresentação do problema

Desde há vários anos que a experência do LNEC e da UNL na utilização do PICTURE-BOOK para controlar o terminal gráfico GT42 tem revelado a inadequação desta camada de software.

Decidiu-se assim implementar conjuntamente um software gráfico de controle do GT 42 cujas características gerais deveriam ser:

- Possibilidades de definição e construção de imagens estruturadas.
- Manipulação das imagens ou suas componentes, dinâmicamente.
- Transformações de imagem.
- Gestão dinâmica da memória pela libertação do espaço ocupado por componentes de imagem, no entretanto tornadas inúteis.
- Definições de macros-estruturas de componentes de imagens-SIMBOLOS.
- Definição e controle dos periféricos lógicos, realizando automáticamente todas as funções elementares de diálogo com o utilizador (correcção, eco e sinalização).
- Adequação geral a uma instalação fácil dos softwares gráficos de alto nível.
- Transformação de imagem em tempo real.
- Controle de erros, quer do utilizador da consola gráfica, quer dos programas de aplicação.
- Gestão das comunicações com o computador central.
- Parametrização em função das necessidades do utilizador.
- Adaptação a futuras expansões do hardware do terminal (memória e periféricos).

- Qualidade de tipo industrial.

Nesse sentido iniciaram-se os trabalhos com uma equipa formada por J. Duarte Cunha pelo LNEC e M. Quirino e J. A. Legatheaux Martins pela UNL. Esta equipa veio sendo sucessivamente acrescida pelos alunos da UNL, Nuno Lobo e Costa Pires, e por Carlos Cotta e Antônio Inês pelo lado do LNEC.

3.2 - Soluções possíveis para implementar Software de Sistema

O sistema em causa é constituído por um mini computador cuja memória é partilhada por um processador gráfico ao qual estão conectados diversos periféricos ("Light Pen", teclado, etc). O mini computador, por sua vez, assegura também o diálogo, através de um protocolo, com o computador central, DEC-10. Neste contexto, trata-se claramente de um software de sistema.

Quais são as soluções hoje em dia disponíveis para implementar um software com estas características:

a) Escrita do sistema em Assembler:
   Esta solução permite satisfazer plenamente as qualidades eficiência e as características de controle do hardware. Todos os outros objectivos são em geral sacrificados.
   Esta solução só é aconselhável se o sistema for simples e de pequena dimensão.

b) Utilização simultânea de código em alto e baixo nível:
   Outra solução é escrever em Assembler a parte do sistema que é claramente dependente do hardware e as restantes partes em código de alto nível.

   Esta solução é bastante adequada se se dispuser de um compilador com características industriais, permitindo inserir código assembler ou "sistem call's". Em qualquer hipótese é necessário assegurar que as funções redundantes do "Run Time System" da linguagem de alto nível não são carregados com o código final.

   A adopção desta solução necessita que o código assembler se restringa a pequenas rotinas, bem definidas e identificadas e nunca se expanda pelo conjunto do código.

A qualidade final do conjunto pode ser razoável.

c) Escrita do sistema totalmente em código de alto nível:
Com o aparecimento das linguagens da família CONCURRENT PASCAL, MODULA e ADA, é possível escrever um sistema completo para uma máquina nua, controlando toda a periferia que se deseje.
Esta é a solução a adoptar por excelência se um compilador para estas linguagens estiver disponível. O critério eficiência não é' em si um argumento contra esta hipótese. Outros aspectos como a adequação ao acesso ao hardware estão hoje em dia no fundamental resolvidos, quer pela introdução dos conceitos de "binding" (1) das abstracções feitas com componentes hardware, quer pela introdução de zonas onde o "type-checking" é relaxado (2) resumindo-se à comparação da dimensão das representações dos tipos envolvidos.

d) Escrita do sistema numa linguagem de implementação de sistema. (BLISS por exemplo).
Estas linguagens são super essemblers munidos dos conceitos de modularidade, estruturas de controle e procedimentos. Esta simbiose de conceitos alto e baixo nível deve ser considerada uma solução transitória devido à sua heterogeneidade. Usada com extremos cuidados é uma solução adequada.

Um aspecto a ter presente à partida, é que a fusão dos conceitos de alto e baixo nível, impede o compilador de controlar a coerência do código (Não há "type-checking" , controle da passagem de parâmetros, ou controle de "side-effects"). É

---

(1) - Notas de pé de página cujo

(2) - texto se encontra na contra face desta página

por esta via que vão entrar a maioria dos erros no sitema.

e) Escrita em código de baixo nível, partindo de um modelo alto nível, traduzido à mão segundo um standard de tradução.

Solução no seu essencial descrita em 2.12.

(1) Falando de linguagens de programação o termo inglês "binding" é em geral utilizado com o sentido de uma aplicação dos conceitos abstractos de uma linguagem de alto nível numa linguagem de baixo nível (linguagem fonte e objecto). Isto é, o "binding" de A é B se B é a concretização do conceito A.

(2) Nas linguagens de alto nível os compiladores verificam estáticamente todos os aspectos da coerência do programa. Quando se indica claramente que se pretende que aqueles testes não sejam realizados, utiliza-se em inglês um termo, cuja tradução literal para português, é "relaxar".

3.3 - Panorâmica geral da solução adoptada

Analisando o conjunto de funções do sistema, cedo se reconheceu que este deveria ser composto por 4 componentes. Uma componente residente no PDP11 que asseguraria todas as funções de construção, manipulação e transformação das imagens, uma componente residente no PDP11 que asseguraria todas as funções de gestão dos periféricos de input, uma componente residente no PDP11 que asseguraria a comunicação com o computador central e uma componente residente no DEC 10 que asseguraria a interface com o programa de aplicação e a comunicação com o PDP11.

Essas componentes são, necessáriamente, outras tantas abstracções funcionais que podem ser modelizadas e implementadas de acordo com esta visão.

Assim, a componente de gestão de imagem é um tipo de dados abstracto que é caracterizado pelas operações de construção, manipulação, transformação e destruição de imagens e suas componentes. São estas operações, com a sua definição, que interessam para a sua especificação. Uma sua primeira versão foi feita informalmente em /3.3/.

A componente de gestão do input tem um modelo mais complexo e ainda não suficientemente aprofundado

No seu essencial ele é uma máquina abstracta que põe à disposição do programa de aplicação o conceito periférico lógico e gere o diálogo com o utilizador. Sob este ponto de vista é uma componente bem definida. Uma sua primeira especificação informal é feita em /3.6/.

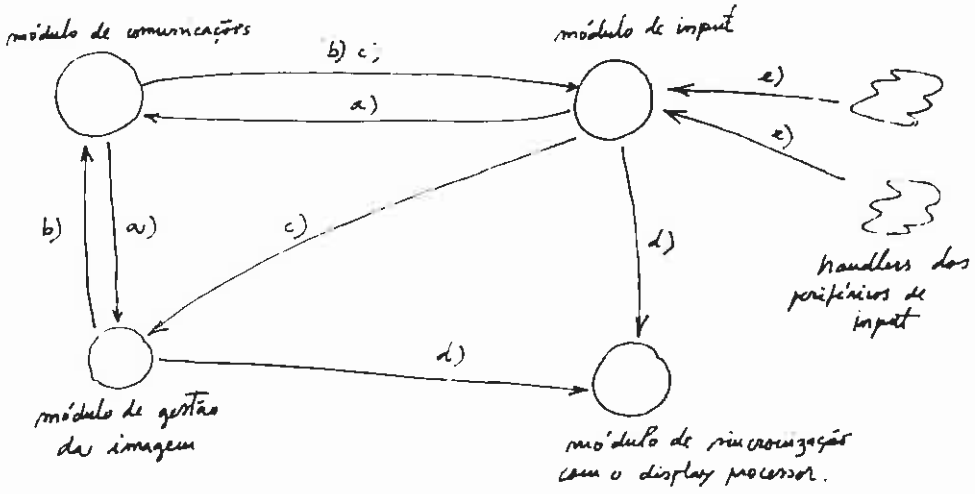
A componente de gestão do diálogo com o DEC-10 é no seu essencial uma máquina abstracta constituída por um autómato que reconhece o protocolo da transmissão e desencadeia as operações reconhecidas nas outras componentes e põe à disposição destas as operações necessárias para a transmissão de mensagens para o computador central

Na implementação cada componente vai traduzir-se num módulo.

As abstracções assim desenhadas, são a primeira fase do processo de análise do sistema por refinamentos sucessivos. Cada um dos módulos é visto pelos outros, apenas pelas operações que põem à disposição destes. O módulo, conjunto de definições, apenas "mostra" aos outros o que dele é relevante. Todos os pormenores da forma como essa implementação é feita são-lhes escondidos e, na implementação, o acesso a esses pormenores internos por outra via que não seja a da interface definida deve ser proibida.

Num contexto ainda teórico, o desenho deveria ser conduzido especificando formalmente aquele conjunto de módulos em termos das operações que exportam! No caso concrecto para cada uma daquelas abstrações foi definida a sua interface informalmente.

Outro aspecto a ter presente no desenho é o grafo de relações entre as componentes. Neste grafo, um arco tem o significado: A → B é equivalente a A usa uma operação definida por B. Ou numa linguagem decorrente dos conceitos de módulo, importação e exportação: A, importa e utiliza uma definição exportada por B.



Grafo de relação dos módulos

Nesse grafo as etiquetas têm o seguinte significado:

a) Desencadear uma operação pedida pelo programa de aplicação.
b) Desencadear a resposta a uma operação pedida pelo programa de aplicação ou em resposta ao utilizador.
c) Desencadear modificações da imagem em resposta a pedidos do utilizador.
d) Operações de sincronização com o Display processor
e) Desencadear operações em resposta a acções do utilizador.

Para a modelização e implementação deste sistema impunha-se a utilização de uma linguagem com capacidade de expressão de tipos de dados abstractos, definição de handlers de periféricos e expressão da concorrência. Na ausência duma tal linguagem disponível para o sistema em causa, optou-se por conduzir a modelização e a implementação em linguagens diferentes. Para a modelização foi seleccionado o Pascal por razões que serão expostas em 3.4. Para a implementação foi seleccionado o BLISS 11, por se dispôr de um cross-compiler do DEC 10 para o PDP11.

3.4 - Modelização do módulo de gestão de imagem

O Módulo de gestão de imagem /3.3/, /3.2/, /3.5/, /3.4/ é caracterizado por um conjunto de operações que definem o conceito de imagem, componente de uma imagem, componentes gráficas dessas componentes de imagem, etc, assim como operações de manipulação (apagar, acender,...) e transformação (translacionar,...) dessas componentes.

O seu desenho exige a concretização dos conceitos sobre um conjunto relativamente extenso de estruturas de dados, manipuladas por algorítmos não necessariamente triviais (buscas por dicotomia, garbage collection, optimização da display-file, etc). Poder-se-ia ter tomado a opção de passar da definição à sua implementação em BLISS. A posteriori, parece-nos que essa opção teria sido errada, preferiu-se antes trabalhar na base de um modelo em Pascal(ao Pascal é feita informalmente uma extensão com os conceitos de módulo e de importação e exportação).

Esse modelo, é no seu essencial um programa /3.3/ que indica a forma como a implementação vai ser feita.

Porquê o Pascal? Porque, das linguagens de alto nível disponíveis no DEC-10, acrescentada dos conceitos de módulo, importação e exportação, era aquela que permitia um modelo conceptualmente suficientemente realista para serem nele investigadas as opções a tomar na implementação.

Poder-se-ia ter trabalhado manualmente numa linguagem de mais alto nível. Esta opção teria 2 defeitos. Não dispormos de um compilador que ajudasse à verificação de parte da coerência do modelo. Não permitia investigar tão fácilmente, da adequação das soluções escolhidas.

A implementação de um módulo com estas características é tão dependente do material e das suas particularidades, que é necessário investigar a adequação dos algorítmos e estimar o conjunto das soluções. O modelo em Pascal

permitiu não só optimizar todas as estruturas de dados, como os algorítmos, antes de qualquer implementação. Essa investigação foi feita analisando um código legível e claro, o que seria mais difícil sobre o código BLISS.

Por outro lado, a semântica do Pascal é fácil de traduzir em BLISS, pois a zona do "run time system" da linguagem utilizada no modelo foi reduzida ao minímo.

A utilização deste modelo foi também fundamental para a análise das pré-condições que cada operação exige antes da sua execução, o que é fundamental para garantir a segurança de funcionamento do sistema.

Assim, o primeiro modelo /3.3/, foi traduzido numa primeira implementação /3.2/, a qual confirmou no essencial a justeza das opções tomadas. Um segundo modelo, corrigido de acordo com a experiência adquirida /3.5/, foi então desenhado e a implementação final realizada /3.4/.

Em /3.5/ encontra-se no Capítulo 3 o modelo do módulo de gestão de imagem, a definição em Pascal da sua interface e a definição do módulo de sincronização com o "Display Processor". No anexo-2 são apresentadas as relações destes módulos com os restantes módulos do sistema.

No conjunto, estes módulos correspondem a cerca de 60% do sistema residente no PDP11.

3.5 - Conservação das equivalências semânticas

O processo da tradução do modelo em BLISS, foi realizado manualmente segundo uma "norma de tradução" /3.5/ Anexo 1, desenhada segundo os princípios expostos em 2.12.

No que toca às equivalências semânticas, os pricípais problemas foram encontrados em torno de 2 pontos, a tradução das expressões e a tradução dos tipos de dados, já que as características gerais da linguagem BLISS (ver o ponto 3.6) facilitaram toda a tradução dos conceitos de módulo com importação e exportação, assim como da generalidade das instruções de controle.

Em BLISS não existe o conceito de instrução, mas apenas o de expressão. No que toca à tradução das instruções Pascal em BLISS, de uma forma geral o problema não se coloca, pois existem formas de tradução imediata e relativamente segura, com excepção da instrução de afectação.

A afectação em BLISS não é uma instrução, mas um operador. Assim o cálculo de uma expressão pode desencadear múltiplas afectações. Tantas quanto se queira.

Este aspecto, cria problemas delicados, pois é através das afectações que se introduzem "side-effects". (A chamada a um procedimento ou função só introduz side effects se estas contiverem instruções de afectação).

Este aspecto exigiu que se introduzisse uma séria restrição às expressões a utilizar. A essas expressões restringidas passámos a chamar "Expressões à Pascal".

Por via desta restrição, introduziu-se uma forma normalizada de simular a instrução de afectação do Pascal que pelo menos tem a vantagem de explicitar o "side-effect".

Por outro lado, em BLISS não existe o conceito de tipo de dados, o que tem como consequência imediata o facto de não existir o conceito de tipo dos parâmetros formais de um procedimento ou função.

Estes 2 aspectos, exigiram um especial cuidado na norma. No que toca aos tipos de dados, o que é essencial é que a equivalência semântica se traduz no seguinte facto: a declaração de uma variável corresponde a uma alocação de memória de dimensão necessária para representar o tipo de dados em causa.

No ponto 3.j do Anexo I de /3.5/ é feito um estudo cuidadoso da forma de representar cada tipo de dados e de o manipular.

Um conjunto de MACROS é introduzido para facilitar a manipulação de tipos "enumerated". Quanto aos tipos "sub--range", opcionalmente o código assegura que são verificadas asserções sobre os limites de variação dos valores das variáveis daqueles tipos.

Finalmente, no que toca à passagem de parâmetros, parece-nos ser a zona em que os resultados são mais fracos.

Um conjunto de regras, normaliza a forma como a passagem de parâmetros deve ser feita.

3.6 - Uma visão crítica da linguagem BLISS

Bliss 11 /3.7/ é uma linguagem de implementação de sistema para as máquinas da família PDP 11, sendo a sua característica fundamental a tentativa de encontrar um equilíbrio adequado entre o poder de expressão das linguagens de alto nível e o controle do programador sobre a forma concreta que assume o código objecto após carregado. O programa deve ser altamente optimizável e aproveitar bem os recursos da arquitectura a que se destina. BLISS 11 garante 100% a qualidade eficiência mas relativamente menos a qualidade segurança.

As suas principais características são:

(a) Ausência de tipos de dados com excepção de vector de "bit's" (1 6 ou 8 bit's). A interpretação das propriedades de uma unidade de memória são dependentes do contexto e podem ser:

- inteiro com ou sem sinal.
- caracteres.
- booleano.
- endereço
- componente de uma estrutura.
- nome de um procedimento

Para a definição de estruturas de dados, apenas se pode declarar o seu comprimento e o algorítmo de acesso a uma palavra. Vector é pré-definido.

O valor de uma expressão é sempre o de um vector de 1 6 ou 8 bits, dependente sempre do contexto.(1)

Não existe qualquer forma de "type-checking", todas as instruções sintáticamente correctas têm uma interpretação contextual válida.

(b) O valor de uma variável é o seu endereço. Um operador, ("contents of") permite aceder ao seu conteúdo. Através de uma variável e por aplicação su

(1) Eventualmente um sub campo de um destes vectores.

cessiva deste operador, pode-se aceder virtualmente a toda a memória.

(c) Todas as construções da linguagem são expressões (para permitir uma alta optimização). No que toca à expressão de padrões de controle a linguagem é muito satisfatória. Não existe instruçãode "goto". As situações onde a utilização de 'goto's' é recomendável, são bem resolvidas com as diferentes expressões de saída abrupta (LEAVE, EXIT, RETURN...).

Não existe instrução de afectação, mas sim, operador de afectação ("store"). Este pode figurar indiscriminadamente em qualquer ponto de uma expressão, desencadeando "side-effects" incontroláveis.
Para reforçar a optimização, o cálculo de uma expressão é sempre tentado em tempo de compilação. Este aspecto permite compilação condicional e parametrização da dimensão das estruturas.

Em tempo de compilação são ensaiados rearranjos do código no sentido da sua optimização (independência de acções para optimização da afectação de registos por exemplo), o que, ligado à liberdade da introdução de "side-effects", exige uma manipulação extremamente cuidadosa do operador "store".

(d) Ausência total do conceito de tipo de um parâmetro formal. Este, para todos os efeitos é um valor (endereço), a única forma de passagem de parâmetros explicítamente permitida.

(e) Presença dos conceitos de módulo, importação e exportação. O que permite uma programação altamente modular. Acontece que, no entanto, devido à ausência de "type-checking" e de tipo dos parâmetros, as verificações feitas sobre as interfaces dos módulos, estão relativamente empobrecidas e não podem ser consideradas satisfatórias.

Os pontos sublinhados são quanto a nós as características negativas do BLISS.

Costuma-se argumentar em favor destas soluções, que a programação de sistema não tolera o "type-checking" nem a proibição de manipulação directa de endereços. Posteriormente ao projecto que deu origem ao Bliss, foram inventadas as formas de ultrapassar estes problemas.

Para tal basta introduzir construções especiais nas linguagens para que, no cálculo de certas expressões, o "type-checking" seja ignorado e se confine à comparação da dimensão de representação dos tipos envolvidos. (CONCURRENT PASCAL Por exemplo)

Quanto à manipulação de endereços, ela é necessária, apenas dentro dos "handlers" de periféricos. Introduzindo na linguagem o conceito de "handler" e a forma de aceder aos registos o problema também é ultrapassado (MODULA por exemplo).

3.7 - Resultados quanto à documentação

Um software com estas características tem de ser altamente documentado visto se preverem inúmeras alterações durante a sua exploração. A presença de vários erros não deve também constituir uma surpresa.

A documentação foi bem preparada desde o início de todo o desenho. Assim um investimento relativamente pesado numa especificação muito completa, sintéctica e semiformalizada /3.3/, permitiu que esta se viesse a desdobrar em modelos e especificações, /3.3/, /3.5/, sistemáticamente mantidos em disco, com versão única e munidos de ficheiro histórico de alterações. /3.5/-Anexo 3.

Para a manutenção desta documentação, permanentemente actualizada e disponível, (cerca de 60 páginas para o módulo de gestão de imagem, bastante mais de 100, se incluirmos o código Bliss) foi fundamental a utilização de um sistema em "time-sharing" munido de um editor de textos.

A organização da documentação foi preparada para permitir uma utilização fácil a:

- quem vai fazer os manuais do utilizador.
- quem vai manter o código do módulo.
- quem vai escrever os outros módulos e necessita de conhecer a especificação das interfaces.

Assim, /3.5/ contem informação capaz de responder às questões:

- como se deve utilizar o módulo.
- qual a sua interface no DEC-10.
- quais as asserções a verificar à entrada de cada operação e quem as verifica.
- qual a comunicação exigida por cada operação.
- qual a interface em Bliss com o módulo no PDP11.
- qual o modelo do módulo.
- qual a norma segundo o qual o modelo foi traduzido na implementação.

- qual a história das diferentes alterações feitas à especificação e à implementação.

Toda esta documentação é concebida para acompanhar o módulo durante toda a sua vida. Devendo ser mantida constantemente coerente, com especial realce para o modelo, pelas pessoas encarregues da sua manutenção.

Um resultado interessante de constatar, /3.5/ - Anexo - 1, ponto 4 - Normas de documentação em Bliss, é que a documentação principal dos programas é o próprio modelo. A documentação do código em Bliss resume-se à explicitação de quais as zonas do modelo que este implementa, para aquelas zonas em que a " norma " é insuficinte, ou seja, representação de tipos de dados e passagem de parâmetros.

Um conjunto de MACROS permite aumentar a legibilidade do código BLISS. Para mais pormenores sobre as técnicas usadas ver /3.5/ - Anexo - 1.

3.8 - Um método recomendável para testar módulos de um sistema - catalogação

Quando construimos um sistema e queremos garantir que a implementação está correcta, um dos métodos que em geral se aplica, é o método dos testes sistemáticos.

Na ausência de provas formais é este o método a que temos de recorrer. No entanto é preciso ter presente que este método, em si, é insuficiente.

Dê que maneira proceder então? Em primeiro lugar trabalhar na base das técnicas descritas no conjunto deste trabalho. Conceber um sistema estruturadamente por componentes. A análise da especificação das componentes e da forma como interagem é meio caminho andado para fazer um sistema certo. Em segundo lugar, testar individualmente as componentes e só depois a sua forma de interacção. As componentes devem ser testadas individualmente e implementadas segundo o princípio da desconfiança mútua. Em terceiro lugar fazer revisões constantes dos modelos, soluções e código escrito (de preferência por pessoas diferentes dos responsáveis directos pelo seu desenho). Em quarto lugar prestar especial atenção aos módulos atreitos a erros. São estes módulos, que devido à sua complexidade, ou grande interacção, introduzem a maioria dos erros. Finalmente, em quinto lugar utilizar uma linguagem adequada, ao serviço da modelização e segurança.

No que toca aos testes das componentes individuais, um método a aplicar é a utilização de "debugers on-line".

Quanto a nós, um "debuger on-line" é insuficiente, pois tem tendência a dirigir a atenção para o aspecto microscópico da implementação e a esbater o seu aspecto estrutural.

Construir um programa especial, para teste interactivo de um módulo, que assegure através de comandos que o utilizador pode simular qualquer sequência de operações pos

síveis, mostrando para cada uma delas o resultado da sua aplicação, parece-nos um método adequado ao teste de módulos com as características do módulo de gestão de imagem.

Um tal programa pode ser munido de comandos que mostrem o estado corrente de certas estruturas internas do módulo, ajudando assim a um raciocínio sobre a estrutura do mesmo. Durante os testes de integração pode-se revelar muito útil.

Em seguida, devem-se especificar baterias de testes, dirigidas sobretudo aos pontos críticos do sistema, isto é, ao funcionamento dos módulos atreitos a erros. O programa e as baterias de testes devem acompanhar o sistema na sua fase de manutenção. Para cada nova alteração é exigido que o sistema passe a bateria de testes.

O programa e os testes farão parte de um catálogo de testes e versões do sistema.

Em /3.1/ é descrito um tal programa e uma técnica semelhante aplicada ao software GRI-GRI.

3.9 - Avaliação dos resultados

Globalmente parece-nos ajustado fazer desde já o balanço de que as técnicas utilizadas na estruturação do sistema tiveram repercussões positivas sobre a sua simplicidade, adaptabilidade, segurança e eficiência.

Nomeadamente, no que toca ao módulo de gestão de imagem, parece-nos que a utilização do modelo em Pascal, permitiu em fase de especificação e modelização, incrementar aquelas qualidades desta componente do sistema, o que seria mais difícil, trabalhando directamente, sobre Assembler por exemplo.

O conjunto das técnicas usadas constitui quanto a nós um ensaio coroado de êxito das ténicas expostas nos pontos finais do capítulo anterior.

Qualquer avaliação séria tem, no entanto, de ser feita após a fase de integração das componentes e entrada em exploração, tomando em consideração critérios objectivos, geralmente traduzidos em termos de custos.

Em /3.5/ - Anexo 4, são apresentados alguns resultados que permitirão (no futuro) realizar esse balanço.

4 - BIBLIOGRAFIA

A bibliografia encontra-se organizada de acordo com os pontos dos capítulos anteriores. Uma determinada referência é introduzida num ponto segundo os seguintes critérios.

a) O ponto é o ponto onde uma primeira referência lhe é feita.

b) A referência não é referida no texto, mas enquadra-se naquele sub-título.

As referências com anotações em Inglês são extraídas do artigo: D.M. Dungan
Bibliography on data types
SIGPLAN NOTICES (ACM)
Novembro 1979

2.1 - Características do produto software

/2.1.1/ - Brooks
the Mithycal Man-Month
Addison Wesley, 1974

'Uma visão bastante realista da Engenharia de software. apresentada pelo responsável principal pelo desenho de todo o software do IBM 360. Extremamente documentado e com exemplos'.

/2.1.2/ - C.A.R. Hoare
Computer programming as an engineering discipline. Electronics & Power - Agosto 1973.

'Os programas podem ser conceptualmente mais simples que qualquer outro engenho humano. De que forma fazê-los ter a qualidade daqueles? Demonstrando que o programa está certo'.

2.2 - Especificar um problema

/2.2.1/ - C. Pair
La programmation: de l'enoncé au programe.
Congresso 1978 da AFCET.

'Uma panorâmica dos principais métodos de passagem para um enunciado explicito e daí a um programa - muito resumido'.

/2.2.2/ - C. Pair
Quelques propositions pour un langage de trés haut niveau
Centre de Recherche en Informatique de Nancy. 1976

'L'effort des "langages de trés haut niveau" est de s'intéresser plus à l'anoncé des problemes qu'à la maniére de les résoudre'.

/2.2.3/ - P. Guerreiro

Un Modele Relationnel pour les programmes non-deterministes.
Rapport DEA-Informatique.
Université de Grenoble. France. 1979

2.3 - Modelizar um problema

/2.3.1/ - J. Arsac
La Recursivité
Departamento de Informática da UNL (em vias de publicação)

'Panoramica através de exemplos, da utilização de enunciados recursivos e da sua passagem a programas interativos'.

/2.3.2/ - C.A.R. Hoare
An axiomatic basis for computer program-

ming CACM 10/1969 pag. 576 e em /2.3.6/

'Ver as referencias neste ponto deste texto'.

/2.3.3/ - N. Wirth, C.A.R. Hoare
An axiomatic definition of the Language Pascal.

'idem'.

/2.3.4/ - Backouse, R. C.
Sintax of Programming Languages
Prentice Hall, 1979

'Diversos algorítmos de grafos são implementados e demonstrados pelo método das asserções'.

/2.3.5/ - S. Alagic, M. A. Arbib
the Design of well - Structured and correct Programs. Springer Verlag.1980

'Muitos exemplos de programação descendente com verificação pelo método axiomático à Hoare, o qual é apresentado na introdução'.

/2.3.6/ - David Gries - Editor
Programming Methodology
Springer Verlag. 1978

'Conjunto de artigos de fundo todos da autoria de membros do Grupo 2.3 da IFIP - -Metodologia da programação'.

/2.3.7/ - J. Darlington, R.M. Burstall
A system which Automatically Improves Programs in /2.3.6/.

'Um sistema de tranformação de equações recursivas em programas eficientes é apresentado'.

/2.3.8/ - C.A.R.Hoare
Proof. of a program: FIND
in /2.3.6/

'exemplo como em /2.3.5/'.

/2.3.9/ - R.T. Yeh - Editor
Current trends in programming Methodology- - vol. IV Data Structuring
Prentice Hall. 1978

'Conjunto de artigos de fundo sobre o titulo'.

/2.3.10/ - C.A.R. Hoare
Proof of correctness of data representations.
in /2.3.6/.

'Artigo clássico'.

/2.3.11/ - O. J. Dahl, E. W. Dijkstra, C. A. R. Hoare
Structured Programming
Academic Press. 1972

'O livro que apresentou pela primeira vêz as pedras base de todo o desenvolvimento do software nos últimos anos'.

/2.3.12/ - C.A.R. Hoare
Notes on Data Structuring
in /2.3.11/.

'Artigo clássico sobre a axiomatização de tipos de dados à Hoare. Resultados na origem da linguagem Pascal'.

/2.3.13/ - C.A.R. Hoare
Data Structures
in /2.3.9/, idem /2.3.12/.

/2.3.14/ - J. Boulenger et all
Aspects de la notion de type en programma

tion.

IRIA - Grenoble. 1975.

'Generalização do trabalho de Hoare em /2.3.12/'.

/2.3.15/ - C.A.R Hoare et all

Proof of a recursive program: Quicksort. 1971.
The Computer Journal vol. 14 nº4

/2.3.16/ - C.A.R. Hoare

Proof a a structured program:
'the sieve of Erastotenes'. 1973.

the computer Journal vol.15 nº4.

/2.3.17/ - M.R. Levy

Some Remarks on abstract data types.
SIGPLAN Julho 1977.

'Especificação à Guttag'.

/2.3.18/ - J. Guttag.

Notes on type Abstraction (Version 2).
IEE transactions on software Engineering.
Janeiro 1980.

/2.3.19/ - J. Guttag, J.J. Horning.

the algebric Specification of abstract data types.
Acta informática 10 (1978) e in /2.3.6/.

/2.3.20/ - J. Guttag, E. Horovitz e D.R. Musser.

the Design of data type specifications.
University of Southern California and in /2.3.9/.

/2.3.21/ - S. Kamin

Some definitions for algebraic data type specifications.

SIGPLAN Maio 1979.

'Fundamentos Matemáticos da especificação algébrica'.

/2.3.22/ - J. Guttag.
Abstract data types and the Development of Data Structures.
CACM Junho 1977.

'da especificação à implementação'.

/2.3.23/ - M.E.Majster.
Limits of the "Algebraic" specification of abstract data types.
SIGPLAN Outubro 1977.

/2.3.24/ - M.C. Gaudel et all.
Semantics of Procedures as an algebraic abstract data type.
IRIA - Voluceau - Rapport de Rechercge nº 334 - 1978.
'Aplicação da especificação algébrica à descrição da semântica de uma lingagem de programação'.

/2.3.25/ - J. Guttag.
Notes on type abstraction.
in /2.3.26/

/2.3.26/ - F.L. Bauer (Editor).
Program construction.
Lecture Notes in computer Science Nº 69
Springer Verlag, 1978.

'conjunto de lições apresentadas sobre verificação, transformação e desenho de programas num curso realizado em Marktoberdorf em 1978'.

/2.3.27/ - J.P. Finance
De la Specification abstraite d'une donné à sa representation un memoire,
C.R.I.N - Nancy. 1978.

'Da especificação à Guttag à implemen-

tação'.

/2.3.28/ - Panorama des langages d'aujourd'hui. Groupe Programma tion et langages de l'AFCET bulletin nº 8. 1979.

/2.3.29/ - D. Bert .
Specification Algebrique des types abstraits et certification de Programmes .
in /2.3.28/.

/2.3.30/ - J.L. Remy .
Construction, évaluation et amélioration Systématiques de structure de données.
RAIRO - Informatique théorique vol. 14 nº 1 - 1980 .

'Intrudução de custos nas operações'.

/2.3.31/ - T.L. Booth e C.A. Wielek .
Perfomance Abstract Data Types as a tool in Software Perfomance Analysis and Design.
IEEE Transaction on Software Engineering - Março 1980.

'idem /2.3.30/'.

/2.3.32/ - J.C. Derniame e J.P.Finance .
Types abstraits de données:Specification, Utilisation et réalisation .
CRIN - NANCY. 1980

'Apresentação dos métodos algébricos e com pré e post condições e sua relação com as linguagens de programação'.

/2.3.33/ - D. Warren, L.M. Pereira, F. Pereira
Prolog, the Language and its Implementation Compared with LISP. ACM Symp. on AI

and Programming Languages - Rochester 1977.

/2.3.34/ - M.H. Van Emden et all .
Equations compared with clauses for specification of abstract data types. 1979. University of Waterloo, Ontario CANADA.

' O método algébrico é comparado com o método clausal.
Concluindo que o primeiro é um caso particular so segundo.
No final é apresentada a especificação completa de uma base de dados'.

/2.3.35/ - Compiler construction, an advanced course Lecture Notes in Computer Science nº 22 . Springer Verlag .

'Um clássico da compilação'.

/2.3.36/ - D. Knuth.
Top Down Syntax Analysis .
Acta Informatica 1, 79-110 (1971)

'Uma visão geral do Parsing de gramáticas'.

/2.3.37/ - Griffiths, Cunin .
Programmation dirigeé par les données (CRIN- Nancy). 1980.

'Gramáticas e automatos são usados para desenhar sistemáticamente programas de gestão de stocks'.

/2.3.38/ - D. Coleman .
the Systematic Design of file-processing Programs. Software - Practice and Experience vol.7 pág.371 1977 .

'Orientação identica a /2.3.37/'.

/2.3.39/ - J.C. Reynolds.
Programming with transition Diagrams, in /2.3.6/.

'idem anterior, incluindo provas de programas'.

/2.3.40/ - Peter Naur.
Control - Record - driven processing. In /2.3.41/.

'idem anterior'.

/2.3.41/ - R.T. Yeh - Editor.
Current Trends in Programming Methodology. vol.I Prentice Hall. 1977.

'orientação semelhante a /2.3.6/'.

/2.3.42/ - A.I. Wasserman, S.K. Stinson.
A Specification method for interactive Information Systems - in /2.7.1/.

'idem /2.3.37/'.

/2.3.43/ - José A. Legatheaux Martins, Luis Monteiro.

- Universidade Nova de Lisboa -
- CIUNL. 1981
Linguagens formais e autómatos.

'Em vias de publicação, orientação identica a /2.3.37/'.

/2.3.44/ - Burstall, R. and Goguen, J. 1977, "Putting Theories Together to Make Specifications", Proceedings of the 1977 International Joint Conference on Artificial Inteligence, pp. 1045-1058.
This paper presents a proposal for a language in which one may describe theories and which arose from attempts to generalise methods of building up programs in

terms of abstract data types.

/2.3.45/ - Lucena, C. and Pequeno, T. 1977 "A View of the Program Derivation Process Based on Incompletely Defined Data Types: A Case Study", Tech. Rep., Pontificia Univresidade Católica, Depto. de Informáti ca, Rio de Janeiro, Brazil. Not avaible to author.

/2.3.46/ - Liskov, B. 1975, "Data Types and Program Correctness", SIGPLAN NOTICES, Vol. 10, No. 7, pp. 16-17.
This note discusses the impact of user--defined data types on program provability.

/2.3.47/ - Jones, C. B. 1979, "Constructing a Theo ry of a Data Structure as an Aid to Pro gram Development", Acta Informática, Vol. 11, Fasc. 2, pp. 119-137.
An extension to a method of developing programs via abstract data types is illustrated which aids in proving the cor rectness of programs.

/2.3.48/ - Jones, D. W. 1978, "A Note on some Limi ts of the Algebraic Specification Method", SIGPLAN NOTICES, Vol. 13. No. 4 pp. 64-67.
This note gives an algebraic definition for a data type previously posed as being undefinable by that method.

/2.3.49/ - Goguen, J., Thatcher, J., Wagner, E. and Wright, J. 1975, "Abstract Data Types as Initial Algebras and the correctness of Data Abstractions," Proceedings of the Conference on Computer Graphics,

Pattern Recognition, and Data Stucture, pp. 89-93.
This paper summarizes an initial algebra approach to data types and gives several examples which illustrate how that approach may be used to define data types and to validate the correctness of their implementations.

/2.3.50/ - Goguen, J., Thatcher, J. and Wagner, E. 1976, "An Initial Algebra Approach to the Specification, Correctness, and Implementation of Abstract Data Types," Report RC6487 ($26817), IBM Watson Research Center, Yorktown Heights, N. Y. and in Current Trends in Programming Methodology Volume IV, Prentice-Hall, Inc., Englewood Cliffs, N. J., 1978.
This paper deals with some of the major algebraic issues underlying abstract data types, presents several data type specifications with correctness proofs and explains the issues of error messages and implementations.

/2.3.51/ - Gehani, N. 1977, "More About Abstract Data Types, "Tech. Rep. 133, State University of New York, Buffalo, N. Y.
This paper describes an algebraic technique for the formal specification of abstract data types, considers the problem of the equivalence of arbitrary specifications of abstract data types and deals with the problem of selecting an appropriate implementation.

/2.3.52/ - Wegbreit, B. and Speitzen, J. 1976, "Pro

ving Properties of Complex Data Structures", J. ACM, Vol. 23, No. 2, pp. 389--396.
A method for proving properties of programs which may contain user-defined types is studied.

/2.3.53/ - Kieburtz, R. 1976, "programming without Pointer Variables", Proceedings of the Conference on Data: Abstraction, Definition and Structure, SIGPLAN Notices, Vol. 8, No. 2, pp. 95-107.
The inclusion of a class of data abstractions based on recursively defined data types is considered as an alternative to the use of pointer variables in programming.

/2.3.54/ - Majester, M. 1977, "Data Types, Abstract Data Types and their Specification Problem", TUM-INFO-7740, Technical University of Munich and in Theoretical Computer Science, Vol. 8, No. 1, 1979, pp. 89-127.
A denotational Specification technique for abstract data types is formulated and compared with other techniques.

/2.3.55/ - Nakajima, R., Honda, M. and Nakahara, H. 1978, "Describing and Verifying Programs with Abstract Data Types", Proceedings IFIP Working Conference on Formal Description of Programming Concepts, Norh--Holland, pp. 527-556.
Not available to author.

/2.3.56/ - Pequeno, T. and Lucena, C. 1978, "An Approach for Data Type Specification and

Its Use in Program Verification", Tech. Rep., Pontificia Universidade Catolica, Depto. de Informatica, Rio de Janeiro, Brazil.
An axiomatic approach for data type specification and representation is given which is based on mathematical logic and which aids in verifying programs using abstract data types.

/2.3.57/ - Rosenchein, S. and Katz, S. 1977, "Selection of Representations for Data Structures", Proceedings of the Symposium on Artificial Intelligence and Programming, SIGPLAN Notices, Vol. 12, No. 8, pp. 147-154.
A model is outlined to aid programmers in choosing efficient realizations for their abstract data structures.

/2.3.58/ - Ross, D. 1976, "Toward Foundations for the Understanding of Type", Proceedings of the Conference on Data: Abstraction, Definition and Structure, SIGPLAN Notices, Vol. 8, No. 2, pp. 63-65.
A philosophical view of some of the questions involving data abstraction and type is given.

/2.3.59/ - Zilles, S. 1974, "Algebraic Specification of Data Types", Project MAC Progress Report 11, MIT, Cambridge, Mass., pp. 28-52. Not available to author.

/2.3.60/ - Zilles, S. 1975, Data Algebra: A Specification Technique for Data Structures, Ph. D. Dissertation, Project MAC, MIT, Cambridge, Mass.

Not available to author.

2.5.- Métodos de desenho de Software

/2.5.1/ - N. Wirth.
Program development by stepwise refinement - in /2.3.6/ 'A análise descendente'.

/2.5.2/ - N. Wirth et all.
Pascal - User Manual and Report. Springer Verlag. 1974.
'Relatorio de definição da linguagem Pascal'.

/2.5.3/ - Liskov, Zilles.
Programming with abstract data types.
SIGPLAN NOTICES Set. 1974.
' Metodologia da programação em termos de tipos de dados '.

/2.5.4/ - J. B. Morris.
Programming by Sucessive Refinement of Data Abstractions. Software Practice and Experience Vol. 10 1980.
' idem /2.5.3/ '.

/2.5.5/ - D. D. Cowan et all.
A Data - directed approach to Program Construction. Software P. and Experience Vol. 10 1980.
' idem /2.5.3/ '.

/2.5.6/ - L. Flon.
Program Design With Abstract data types
Carnegie Mellow University.1975.
' idem /2.5.3/ '.

/2.5.7/ - Language Hierarchies and Interfaces.
Lecture Notes in Computer Science Nº 46.
Springer Verlag. 1976.

' Uma coleção de artigos sobre metodologia da programação e sua relação com as linguagens'.

/2.5.8/ - F. L. Bauer - Editor.
Software Engineering.
Lecture Notes in Computer Science Nº 30
Springer Verlag. 1975 .
' Uma visão geral dos metodos de desenho de software, sua relação com linguagens, assim como aspectos relacionados com portabilidade, modularidade, etc '.

/2.5.9/ - Le VAN KIET.
The Module: a tool for Structured Programming. ETH Zurich - Dissertation Nº 6153 - - 1978. 'Apresentação do conceito de Modulo em Modula; sua comparação com outros meios de estruturação de programas, assim como discussão da sua implementação'.

/2.5.10/ - D. L. Parnas,
Information distribution aspects of the design methodology. Proc. IFIP Congres 1971 Vol. 1.
— , On the criteria to be used in decomposing systems into modules. CACM Dez. 1972.
— , A technique for software module specification with examples. CACM Maio 1972
' Um conjunto muito importante de artigos por um dos principais introdutores do conceito de modularidade '.

/2.5.11/ - Banatre, M., Couvert, A., Herman, D. and Raynal, M. 1977, " Types abstraits et Pluralite de Leurs Representations a L'exe-

cution", Program Transformation, 3rd International Symposium on Programming, Paris, Dunod, pp. 263-278.
This paper describes the problem of managing several concrete representations for the same abstract data type at run--time and presents a mechanism to solve that problem.

/2.5.12/ - Bergstra, J. 1978, "What is an Abstract Data type?", Information Processing Letters, Vol. 7, No. 1, pp. 42-43.
This note defines abstract data type from a general operational point of view and discusses some advantages of the generality of such a definition.

/2.5.13/ - Chand, D. and Yadav, S. 1978, "on the Application of Data Abstraction Facilities", Proceedings of the 1978 ACM Annual Conference, Washington, D. C., pp. 639-645.
This paper describes the application and teaching of data abstraction facilities in the construction of programs.

/2.5.14/ - Dennis, J. 1975, "An Example of Programming with Abstract Data Types", SIGPLAN Notices, Vol. 10, No. 7, pp. 25-29.
This article shows how the concept of abstract data types can simplify the construction of probably correct programs.

/2.5.15/ - Fischer, A. and Fischer, M. 1973, "Mode Modules as Representations of Domains", Conference Record of the ACM Symposium on the Principles of Programming Languages, Boston, Mass., pp. 139-143.

This paper presents some mechanisms which allow programs to more closely model problem domains.

2.6 - As linguagens modernas de programação - uma introdução

/2.6.1/ - F . Veillon et all .
Cours de programmation en langage PL/1
Colin . Paris. 1971 .
' Curso sobre a linguagem PL/1.

/2.6.2/ - O. J. Dahl et all .
Common Base Language (Simula).
Norwegian Computing Center .
'Documento oficial de definição do Simula 67'.

/2.6.3/ - Van Winjngaarden et all .
Report on the Algorithmic language
ALGOL 68 .
Mr 101 - 1969
' Documento oficial de definição do Algol 68'

/2.6.4/ - José A. Legatheaux Martins .
Linguagens de alto nível .
U.N.L. . 1980
'Panorâmica actual de linguagens de programação'.

/2.6.5/ - W. Wulf et all .
Global variables considered harmfull.
SIGPLAN NOTICES, 8, 2, 28-34, 1973
' As variáveis globais são perigosas porque possibilitam side-effects '

/2.6.6/ - J.T. Schwartz .
Program Genesis and the design of program

ming languages. in /2.3.9/.

/2.6.7/ - J. J. Horning.
Programming languages for Relieble Computing, Systems, in /2.3.26/.
'De que maneira a linguagem facilita a qualidade final dos sistemas informáticos'.

/2.6.8/ - M. Griffiths.
Programming Methodology and Language Implications, in /2.3.26/.
'idem /2.6.7/'.

/2.6.9/ - E. W. Dijkstra.
GO-TO Considered harmful.
CACM Vol. 11 Nº3 (Março 1968)
'O início da cruzada contra o goto'

/2.6.10/ - Th. A. Zoethout et all.
Why the goto is harmfull.
SIGPLAN NOTICES 1979
'Uma explicação dos perigos do Goto em termos de teoria da computação'.

/2.6.11/ - Ph. Jorrand.
Bibliographie sommaire pour un cours sur les langages de programmation - UNL.
Set. 1978.
'Conjunto de artigos sobre as linguagens de alto nível modernas'.

/2.6.12/ - W. Wulf (Ed.)
An informal definition of alphard. 1978
Carnegie Mellon University, CS-78-105.
'Uma introdução à linguagem ALFHARD'

/2.6.13/ - B. LISKOV.
An introduction to CLU - in /2.6.11/.
'Uma introdução à linguagem CLU'

/2.6.14/ - J.J. Horning.
A case Study in Language Design. Euclid —, Verification of Enclid Programs, in /2.3.26/.
'Uma apresentação da linguagem EUCLID'.

/2.6.15/ - K. Kennedy.
An introduction to the set theoretical language SETL.
Comp. and Maths with Appls. Vol 1 Pergamon Press, in /2.6.11/.
'Apresentação da linguagem SETL'.

/2.6.16/ - P. Y. Cunin.
Fiabilité et Sécurité de programmas
Thése - CRIN - Nancy. 1979.
'Apresentação da linguagem MEFIA'.

/2.6.18/ - N. Wirth.
Modula: a language for Modular programming.
Software Practice and Exp. Vol. 7 3-35 (1977)
'Apresentação da linguagem MODULA'.

/2.6.19/ - J. D. Ichbiah et all.
Preliminary ADA reference Manual
SIGPLAN NOTICES Junho 1979
'Apresentação da linguagem ADA'.

/2.6.20/ - M. Quirino e J. Legatheaux Martins
Implicações das Linguagens de programação modernas nos projectos de Software.
1º Congresso da API e UNL. 1980.
'Apresentação subordinada ao título com exemplos em PASCAL, MODULA e ADA'.

/2.6.21/ - Gannon, J. 1976, "Data Types and Program

ming Reliability: some Preliminary Evidence, "Proceedings of the MRI Syposium on Computer Software Engineering, Vol. 24, Polytechnic Institute of New York, pp. 367-376.
The role of data types in achieving reliable programs is explored.

/2.6.22/ - Gries, D. and Gehani, N. 1977, "Some Ideas on Data Types in High Level Languages", Comm. ACM, Vol. 20, Nº 6, pp. 414-420.
Several issues related to data types are explored such as iterating over the elements of any finite set, generic procedures, and type conversions.

/2.6.23/ - Guariano, L. 1978, "The Evolution of Abstraction in Programming Languages", Department of Computer Science, Carnegie-Mellon University, Pittsburgh, Pa.
This paper discusses how the use of abstraction in programming languages assists the programmer and traces the development of support for the abstraction of objects and of control constructs in programming languages.

/2.6.24/ - Horning, J. 1976, "Some Desirable Properties of Data Abstraction Facilities"; Proceedings of the Conference on Data: Abstraction, Definition and Structure, SIGPLAN Notices, Vol. 8, Nº 2, pp. 60-62
This note explores the advantages provided by data abstraction facilities in a programming language in light of the advantages provided by procedures.

/2.6.25/ - Nordstrom, B. 1978, "Programming with Abstract Data Types, Some Examples", Proceeding of the 1978 ACM Annual Conference, Washington, D. C., pp. 646-654. Some inadequacies of data type facilities in currently well-used programming languages are illustrated via examples.

2.7 - Linguagens de especificação

/2.7.1/ - Proceedings of the conference on Specifications of Reliable Software .
IEEE - Nº 79c = 1401 - 9c . 1979 .
' Ver o ponto 2.7 deste texto '

/2.7.2/ - J. Abrial.
École d'ètê d'informatique de l'IRIA .
Methode et langage de specification .
Support du cours de J. R. ABRIAL
— , Z: A Specification language .
'Apresentação da linguagem de especificação Z com aplicaçoês e exemplos'.

/2.7.3/ - C. Rolland et all .
Le pilotage de l'evolution des systemes d'information. Congrês INFORSID 79 Aix en Provence .
'Sistema completo de especificação funcional, temporal e algêbrica do modelo de um sistema de informação tipo relacional'.

/2.7.4/ - Ana Sales .
Utilisation de la notion de type abstrait Generique en bases de données relationelles.
Congrês AFCET 80 .
'Extensão dos tipos de dados abstractos para a especificação e implementação de

sistemas informáticos como bases de dados relacionais'.

/2.7.5/ - Rui Barbosa.
Functional Specification of data bases--UNL. 1979.
'Um subconjunto da linguagem Z é usado como método de especificação funcional de uma base de dados. É dado um exemplo de gestão de um Departamento Universitário'.

/2.7.6/ - Bergstra, J., Ollongren, A. and Van der Weide, Th. P. 1977, "An Axiomatization of the Rational Data Objects", Proceedings of the 1977 International FCT-Conference, Poznan-Kornik, Poland, pp. 33--38.
This paper introduces the notion of a data type system which is a generalization of Guttag's notion of abstract data types.

/2.7.7/ Ehrig, H., Kreowski, H. and Padawitz, P. 1978, "Stepwise Specification and Implementation of Abstract Data Types", Fifth Colloquium, Udine, Italy, Automata, Languages and Programming, Lecture Notes in Computer Science 62, Springer-Verlag, pp. 205--226.
The algebraic approach to the specification and implementation of abstract data types is extended to study problems of stepwise specification and implementation.

2.8 - Características do Software

/2.8.1/ - C. A. R. Hoare

The Quality of Software - Guest Editorial.
Software Pract. and Experience Vol. 2 103-105 1972.
'17 Boas características do software são apresentadas'.

/2.8.2/ - Per Brinch Hansen.
The architecture of concurrent programs Prentice Hall. 1976.
'Na introdução deste livro são apresentadas várias boas características do software e sua relação com as linguagens'

/2.8.3/ - P. J. Brown - Editor.
Software Portability - An advanced course.
Cambridge University Press. 1977.
'Uma panoramica dos problemas de portabilidade e sua relação com linguagens'.

/2.8.4/ - M. Griffths.
Verifiers and Filters - in /2.8.3/
'Para construir um programa estável deve-se apenas utilizar um sub-conjunto estável das linguagens disponíveis. Esse sub-conjunto deve ser fixado num standard de programação. Um programa especial, verificará se o sub-conjunto é respeitado'.

/2.8.5/ - Linden, T. 1976, "The Use of Abstract Data Types to Simplify Program Modifications", Proceedings of the Conference on Data: Abstraction, Definition and Structure, SIGPLAN Notices, Vol. 8, Nº2, pp. 12-23.

This paper includes an extensive introduction to the concept of abstract data types and shows that a program is likely to be much easier to modify if it is structured using abstract data types as the basic unit of modularity.

2.9 - Linguagens e abstração

/2.9.1/ - Barnard, D. and Elliott, W., eds. 1977, "Tech. Rep. 82, Computer Systems Research Group, University of Toronto, Toronto, Ontario and in SIGPLAN Notices, Vol. 13, No. 3, 1978, pp. 34-84.
Several papers from this report include a discussion of the various data type facilities available in EUCLID as well as a comparison of type facilities between EUCLID and PASCAL and between EUCLID and MODULA.

/2.9.2/ - Chang, E., Kaden, N. and Elliott, W. 1978, "Abstract Data Types in EUCLID", SIGPLAN Notices, Vol. 13, Nº 3, pp. 34-40.
This paper assesses the data abstraction facilities provided by EUCLID, examines two ways of instantiating EUCLID modules, and discusses features of modules not included in the language.

/2.9.3/ - Conradi, R. 1976, "Further Critical Comments on PASCAL, Particulary as a Systems Programming Language", SIGPLAN Notices, Vol. 11, Nº 11, pp. 8-25.
This paper includes a critical review of data types in PASCAL.

/2.9.4/ - Lampson, B., et al. 1977, Report on the Programming Language EUCLID, SIGPLAN No

tices, Vol. 12, Nº 2.
This report contains a description of parameterized types in EUCLID.

/2.9.5/ - Lampson, B., et al. 1978, "Revised Report on the Programming Language EUCLID", Tech. Rep. CSL78-2, Xerox Research Center. Not available to author.

/2.9.6/ - Lecarme, O. and Desjardins, P. 1974, "Reply to a Paper by A. N. Habermann on the Programming Language PASCAL", SIGPAN Notices, Vol. 9, Nº 10, pp. 21-27.
Several criticisms of PASCAL concerning subranges and types are considered.

/2.9.7/ - Lecarme, O. and Desjardins, P. 1975, "More Comments on the Programming Language PASCAL", Acta Informática 4, pp. 231-243.
This article also includes a critical discussion of subranges and types in PASCAL.

/2.9.8/ - Liskov, B. 1974, "A Note on CLU", in CLU Design Notes, Project MAC, MIT, Cambridge, Mass.
The motivation underlying the development of CLU is discussed and an informal description of its data type facilities is given.

/2.9.9/ - Liskov, B. 1976, "An Introduction to CLU", Computation Structures Group Memo 136, MIT, Cambridge, Mass.
Not available to author.

/2.9.10/ - Liskov, B., Snyder, A., Atkinson, R. and Schaffert, C. 1977, "Abstraction Mechanisms in CLU", Comm. ACM, Vol. 20, Nº8, pp. 564-576.

This paper illustrates the use of data, procedural, and control abstraction by means of programming examples and shows how CLU modules are used to implement those abstractions.

/2.9.11/ - Liskov, B., et al 1978, "CLU Reference Manual", Computation Structures Group Memo Nº 161, Laboratory for Computer Science, MIT, Cambridge, Mass.
This manual includes a discussion of the data type definition facilities available in CLU.

/2.9.12/ - London, R., Shaw, M. and Wulf, W. 1976, "Abstraction and Verification in ALPHARD: A Symbol Table Example", Technical Reports, USC Information Sciences Institute, Marina del Rey, Calif. and Carnegie-Mellon University, Pittsburgh, Pa.
This paper designs, implements and verifies a general symbol table mechanism which illustrates the use of programmer defined abstractions.

/2.9.13/ - London, R., et al. 1978, "Proof Rules for the Programming Language EUCLID", Acta Informática 10, pp. 1-26.
Hoare-style proof rules are given which are applicable only to legal EUCLID programs and which deal with the various data types availables in EUCLID.

/2.9.14/ - Johnson, R. and Morris, J. 1976, "Abstract Data Types in the MODEL Programming Language", Proceedings of the Conference on Data: Abstraction, Definition and Structure, SIGPLAN Notices, Vol. 8,

Nº 2, pp. 36-46.
This paper describes the characteristics and suggested uses of abstract data types in the MODEL programming language.

/2.9.15/ - Habermann, A. 1973, "Critical Comments on the Programming Language PASCAL", Acta Informatica e, pp. 47-57.
This paper discusses some problems with PASCAL, including those caused by the confusion of ranges, types and structures.

/2.9.16/ - Wasserman, A. 1975, "Issues in Programming Language Design - An Overview", SIGPLAN Notices, Vol. 10, Nº 7, pp. 10--12.
This note identifies data types and abstraction as important issues in current work in language design.

/2.9.17/ - Morris, J., Jr. 1973, "Protection in Programmig Languages", Comm. ACM, Vol. 16, Nº 1, pp. 15-21.
This paper includes a discussion on protecting user-defined types from type violations.

/2.9.18/ - Palme, J. 1973, "Protected Program Modules in SIMULA 67", FOAP Report C8372-M3 (E5), Research Institute of National Defense, Stockholm, Sweden.
Not available to author.

/2.9.19/ - Palme, J. 1976, " New Feature for Module Protection in Simula", SIGLAN Notices, Vol. 11, Nº 5, pp. 59-62.
This note describes an extension for

protecting SIMULA classes from side-effects.

/2.9.20/ - Popek, G., et al. 1977, "Notes on the Design of EUCLID", Proceedings of an ACM Conference on Language Design

/2.9.21/ - Schaffert, C., Snyder, A. and Atkinson, R. 1975, "The CLU Reference Manual", Project MAC, MIT, Cambridge, Mass.
Not available to author.

/2.9.22/ - Scheifler, R. and Snyder, A. 1977, CLU Information Package, Computation Structures Group Memo Nº 154, Laboratory for Computer Science, MIT, Cambridge, Mass.
Not available to author.

/2.9.23/ - Shaw, M. 1976, "Abstraction and Verification in ALPHARD: Design and Verification of a Tree Handler", "Proceedings of the Fifth Texas Conference on Computing Systems, pp. 86-94.
Techniques for specifying and verifying abstractions in ALPHARD are illustrated by means of developing a program for manipulating the parse tree of an arithetic expression.
This paper introduces a way of specializing loop statements in ALPHARD to operate on abstract entities without explicitly depending on the representation of those entities.

/2.9.24/ - Welsh, J., Sneeringer, W. and Hoare, C. A. R. 1977, "Ambiguities and Insecurities in PASCAL", Software, Practice and Experience 7, pp. 685-696.

This paper includes a discussion of the equivalence of types in PASCAL.

/2.9.25/ - Wirth, N. 1975, "An Assessment of the Programming Language PASCAL", Proceedings of the 1975 International Conference on Reliable Software, SIGPLAN Notices, Vol. 10, No. 6, pp. 23-30.
Some problems with PASCAL related to features such as the concept of type union are discussed in light of program reliability.

/2.9.26/ - Wulf, W. 1974, "ALPHARD: Towards a Language to Support Structured Programs", Department of Computer Science, Carnegie-Mellon University, Pittsburgh, Pa.
Not available to author.

/2.9.27/ - Wulf, W., London, R. and Shaw, M. 1976, "Abstraction and Verification in ALPHARD: Introduction to Language and Methodology", Technical Reports, Carnegie-Mellon University, Pittsburgh, Pa. and USC Information Sciences Institute, Marina del Rey, Calif.
The data abstraction facilities of the ALPHARD programming language and the associated verification methodology are described.

/2.9.28/ - Wulf, W., London, R. and Shaw, M. 1976, "An Introduction to the Construction and Verification of ALPHARD Programming", IEEE Transaction on Software Engineering, Vol. Se-2, Nº 4, pp. 253-265.
This paper introduces ALPHARD by developing and verifying a data structure defi

nition and also shows how each language construct contributes to the development of the abstraction.

/2.9.29/ - Panorama des Languages d'aujourd'hui - - Carsege 14-22 Maio 1979 - Colóquio,bulletin du Groplan de l'AFCET Nº 8
'Panorama geral das linguagens modernas' equivalente a /2.3.28/.

/2.9.30/ - Paul Jacquet.
Abstraction et Genericité,
in /2.9.29/.

/2.9.31/ - K. Proch.
Parametrisation des unités de Compilation en A. T. M. Tése de DEA - Université Nancy.
'A expressão de tipos de dados genéricos em ATM. Contem uma comparação com as linguagens CLU e Alphard'.

/2.9.32/ - R. H. Campbell et all. An overview of Path Pascal design
SIGPLAN Set.80
'Path Pascal=Pascal+modulos+processos concurrentes+Path expressions'.

2.10 - Linguagens, fiabilidade e segurança de programação

/2.10.1/ - Gannon, J. and Horning, J. 1975, "The Impact of Language Design on the Production of Reliable Software", Proceedings of the 1975 International Conference on Reliable Software, SIGPLAN Notices, Vol. 10, Nº 6, pp. 10-22.
General language design principles are applied to specific language features such as hierarchies of data and type-

checking to enhance the reliability of programs.

/2.10.2/ - J. J. Horning.
Programming languages for Reliable Computing systems.
in /2.3.26/.
'Uma apresentação completa de todo o problema da segurança da programação, desde os aspectos humanos, linguagens, abstração, error-recovery, verificação e trabalho com as linguagens clássicas'.
Equivalente a /2.6.7/.
Ver também /2.6.21/.

/2.10.3/ - M. Loyer.
Comparaison des structures modulaires de quelques languages de programmation , in /2.9.29/.
'Comparação dos aspectos de modularidade, protecção, visibilidade e diferença de pontos de vista do utilizador e do implementador em Simula, Modula, Mesa e ADA'.

/2.10.4/ - G. J. Williams.
Programm checking.
University of Rocherter. 1979
'A dedução estática das propriedades das variáveis de um programa'.

2.11 - Sistemas de ajuda à especificação e à programação

/2.11.1/ - A. N. Haberman .
The Gandalf Project .
Carnegie Mellon Computer Science Research Review. 1979
'Descreve o projecto de um sistema de a

ajuda à programação em ADA'.

/2.11.2/ - R. Minot .

ATM: Un systeme de fabrication de programmes basé sur les concepts de modularité et type abstrait. These - CRIN - - Nancy. 1980 .
'Descreve um sistema de ajuda à programação baseado numa linguagem experimental definida para o efeito-ATM'

/2.11.3/ - V. Douzeau - Gouge et all .
Programming environments based on structured editors: the Mentor experience .
INRIA - Julho 1980
'Descreve-se um editor que reconhece a estrutura sintáctica da linguagem a editar - MENTOR'.

/2.11.4/ - B. Malése .
Manipulation des programmes Pascal au niveau des Concepts du language.
Thése - Université d'Orsay - Junho 1980
'Um sistema baseado em MENTOR para manipulação de programas Pascal'.

/2.11.5/ - J. Darlington and R. M. Burstall .
A system which automatically improves programs - in /2.3.6/
Equivalente à referência /2.3.7/.

/2.11.6/ - D. R. Musser .
Abstract data type Specification in the AFFIRM system .
IEEE Transaction on Software Eng. Jan. 1980 .
'Sistema interactivo para desenho de programas utilizando a especificação à

Guttag e à Hoare'.

/2.11.7/ - M. Demuynck et all.
Systeme ZAIDE d'aide à la Specification
Congrês AFCET 1980.
'Sistema de ajuda à especificação da linguagem Z'.

/2.11.8/ - R. Schneider e J. P. Thomesse
Les Réseaux de Petri dans l'aide à la conception d'applications temp reel et réparties.
Congresso AFCET 80.
'Sistema de ajuda à modelização e implementação de aplicações repartidas'.

2.12 - E quando alguns constrangimentos impõem a utilização de linguagens de baixo nível

/2.12.1/ - Jim Welch .
Structured programming in Macro Assembly Languages. Software Practice and Exp. Vol.8 371-376 1978 .
'Um conjunto de macros para programação estruturada em Macro11 - DEC PDP11'.

/2.12.2/ - Luís Arriaga da Cunha, João Duarte Cunha .
Algumas notas sobre utilização de MACROS em linguagens de assembler.
CI - LNEC, Lisboa, 1977
'Idem anterior para MACRO10'.

/2.12.3/ - ——— . Software for The Plotter Complot DP -8.
LNEC, Lisboa, 1978
'A implementação de software de controlo de um plotter com um modelo em Simula'.

/2.12.4/ - David E. Boddy.
Structured FORTRAN - With or without a preprocessor SIGPLAN NOTICES - Abril 1977.
'Um pré processador permitindo a programação em FORTRAN com estruturas de controle modernas é descrito. O pré processador gera automaticámente

o programa FORTRAN STANDARD
Com if's e goto's'.

2.13 - Condução do projecto Software

/2.13.1/ - J. N. Buxton.
Software Engenieering, in
/2.3.6/

/2.13.2/ - W. M. Turski.
Software Engenieering - some
principles and problems, in
/2.3.6/

'Em ambos os artigos é apresentada uma visão semelhante, mas mais condensada, à do Prof. Brooks em /2.1.1/'.

3. - Desenho de um sistema concreto - estudo do monitor gráfico a instalar no GT42

/3.1/ - José A. Legatheaux Martins. Apresentação da implementação de GRI-GRI no sistema DEC10+ +GT42. - CIUNL 4/79, Lisboa, 1979

_____, GRI-GRI (Logiciel graphique interactive de base) Manual do Utilizador para o DEC-10 (Versão-1) Junho de 1979 - CIUNL-3/79, Lisboa, 1979

_____, A implementação do Software gráfico interactivo de base, GRI-GRI, no DEC10, balanço e perspectivas. Julho de 1979 - CIUNL - 6/79, Lisboa, 1979

/3.2/ - Nuno Lobo e Costa Pires. Implementação do Modulo de Gestão de imagem do Monitor gráfico para o PDP11 do LNEC (Projectos SAGRA-8o - UNL e SGL - LNEC) CIUNL-14/80 Lisboa, Novembro, 1980
'Primeira versão do módulo'.

/3.3/ - J. Cunha, J. Legatheaux Martins, M. Quirino
Construção de um monitor Gráfico, I-parte - Gestão de imagens (Proj. SAGRA80 - UNL e SGL - LNEC)
CIUNL-6/80 Lisboa, Julho,1980
' Primeira especificação do módulo

módulo de gestão de imagem'.

/3.4/ - Nuno Lobo, Costa Pires.
Implementação final do Módulo de Gestão de imagem do Monitor gráfico para o PDP11 do LNEC (Projectos SACRA-80 - UNL e SGL - LNEC - Em vias de publicação) Lisboa, 1981.
'Segunda versão do módulo'.

/3.5/ - J. A. Legatheaux Martins
Construção de um Monitor Gráfico.
Documentação final da gestão de imagem (Projectos SACRA80 - UNL e SGL-LNEC - em vias de publicação) Lisboa, 1981.
'Documentação final dos módulos relacionados com a gestão de imagem'.

/3.6/ - J. Cunha, J. Legatheaux Martins, M. Quirino.
Primeira especificação do módulo de gestão dos periféricos (Relatório interno do LNEC em vias de publicação)
Lisboa 1981.
'Primeira especificação daquele módulo'.

/3.7/ - Jerry L. Apperson et all.
BLISS11 programer's manual
Computer Science Department.
Carnegie Mellon University.
Pittsburgh (Distribuído pela DECUS) 1972.

/3.8/ - K. L. Heninger.
Specifying Software Requirements for Complex Systems: New Techniques and their application.
IEEE transactions on Soft. Engenieering - Jan. 1980

'São descritas técnicas concretas a aplicar na especificação e documentação de sistemas complexos. No caso, um programa de ajuda à pilotagem de um caça supersónico. Descrição do input, descrição do output, relações entre ambos em termos de conditions e events'.

/3.9/ - C. A. R. Hoare.
Monitors: an operating system structuring concept.
CACM - Outubro 1974

'Um artigo clássico de introdução dos monitores'.

/3.10/ - P. Brinch Hansen.
Structured Multiprogramming.
CACM Julho 1972
'Primeira proposta de notações linguísticas para os Monitores'.

/3.11/ - P. Brinch Hansen.
The programming language Concurrent Pascal in /2.3.6/

'A primeira apresentação da linguagem Concurrent Pascal, pri-

meira linguagem com o conceito de Monitor'.

/3.12/ - R. C. Holt et all.
Structured Concurrent Programming with Operating System aplications. Addison Wesley, Toronto . 1978 .

'Introdução aos sistemas de operação baseada no essencial no conceito de monitor expresso na linguagem CSP/K'.

/3.13/ - P. le Guernic, M. Raynal .
L'expression du controle' des accés concurrents aux objects. in /2.3.28/.
Carsége 1979.
'As regiões críticas, monitores, expressões de caminho, contadores de estado e serializadores são comparados sob o ponto de vista de potência de expressão e métodos de provas'.

/2.6.25/ - Nordstrom, B. 1978, "Programming with Abstract Data Types, Some Examples", Proceeding of the 1978 ACM Annual Conference, Washington, D. C., pp. 646-654. Some inadequacies of data type facilities in currently well-used programming languages are illustrated via examples.

2.7 - Linguagens de especificação

/2.7.1/ - Proceedings of the conference on Specifications of Reliable Software .
IEEE - Nº 79c = 1401 - 9c . 1979 .
' Ver o ponto 2.7 deste texto '

/2.7.2/ - J. Abrial.
École d'été d'informatique de l'IRIA .
Methode et langage de specification .
Support du cours de J. R. ABRIAL
— , Z: A Specification language .
'Apresentação da linguagem de especificação Z com aplicaçoẽs e exemplos'.

/2.7.3/ - C. Rolland et all .
Le pilotage de l'evolution des systemes d'information. Congrés INFORSID 79. Aix en Provence .
'Sistema completo de especificação funcional, temporal e algébrica do modelo de um sistema de informação tipo relacional'.

/2.7.4/ - Ana Sales .
Utilisation de la notion de type abstrait Generique en bases de données relationelles.
Congrés AFCET 80 .
'Extensão dos tipos de dados abstractos para a especificação e implementação de

sistemas informáticos como bases de dados relacionais'.

/2.7.5/ - Rui Barbosa.
Functional Specification of data bases--UNL. 1979.
'Um subconjunto da linguagem Z é usado como método de especificação funcional de uma base de dados. É dado um exemplo de gestão de um Departamento Universitário'.

/2.7.6/ - Bergstra, J., Ollongren, A. and Van der Weide, Th. P. 1977, "An Axiomatization of the Rational Data Objects", Proceedings of the 1977 International FCT-Conference, Poznan-Kornik, Poland, pp. 33--38.
This paper introduces the notion of a data type system which is a generalization of Guttag's notion of abstract data types.

/2.7.7/ Ehrig, H., Kreowski, H. and Padawitz, P. 1978, "Stepwise Specification and Implementation of Abstract Data Types", Fifth Colloquium, Udine, Italy, Automata, Languages and Programming, Lecture Notes in Computer Science 62, Springer-Verlag, pp. 205--226.
The algebraic approach to the specification and implementation of abstract data types is extended to study problems of stepwise specification and implementation.

2.8 - Características do Software

/2.8.1/ - C. A. R. Hoare

The Quality of Software - Guest Editorial.
Software Pract. and Experience Vol. 2 103-105 1972.
'17 Boas características do software são apresentadas'.

/2.8.2/ - Per Brinch Hansen.
The architecture of concurrent programs Prentice Hall. 1976.
'Na introdução deste livro são apresentadas várias boas características do software e sua relação com as linguagens'

/2.8.3/ - P. J. Brown - Editor.
Software Portability - An advanced course.
Cambridge University Press. 1977.
'Uma panoramica dos problemas de portabilidade e sua relação com linguagens'.

/2.8.4/ - M. Griffths.
Verifiers and Filters - in /2.8.3/
'Para construir um programa estável deve-se apenas utilizar um sub-conjunto estável das linguagens disponíveis. Esse sub-conjunto deve ser fixado num standard de programação. Um programa especial, verificará se o sub-conjunto é respeitado'.

/2.8.5/ - Linden, T. 1976, "The Use of Abstract Data Types to Simplify Program Modifications", Proceedings of the Conference on Data: Abstraction, Definition and Structure, SIGPLAN Notices, Vol. 8, Nº2, pp. 12-23.

This paper includes an extensive introduction to the concept of abstract data types and shows that a program is likely to be much easier to modify if it is structured using abstract data types as the basic unit of modularity.

2.9 - Linguagens e abstração

/2.9.1/ - Barnard, D. and Elliott, W., eds. 1977, "Tech. Rep. 82, Computer Systems Research Group, University of Toronto, Toronto, Ontario and in SIGPLAN Notices, Vol. 13, No. 3, 1978, pp. 34-84.
Several papers from this report include a discussion of the various data type facilities available in EUCLID as well as a comparison of type facilities between EUCLID and PASCAL and between EUCLID and MODULA.

/2.9.2/ - Chang, E., Kaden, N. and Elliott, W. 1978, "Abstract Data Types in EUCLID", SIGPLAN Notices, Vol. 13, Nº 3, pp. 34-40
This paper assesses the data abstraction facilities provided by EUCLID, examines two ways of instantiating EUCLID modules, and discusses features of modules not included in the language.

/2.9.3/ - Conradi, R. 1976, "Further Critical Comments on PASCAL, Particulary as a Systems Programming Language", SIGPLAN Notices, Vol. 11, Nº 11, pp. 8-25.
This paper includes a critical review of data types in PASCAL.

/2.9.4/ - Lampson, B., et al. 1977, Report on the Programming Language EUCLID, SIGPLAN No

tices, Vol. 12, Nº 2.
This report contains a description of parameterized types in EUCLID.

/2.9.5/ - Lampson, B., et al. 1978, "Revised Report on the Programming Language EUCLID", Tech. Rep. CSL78-2, Xerox Research Center. Not available to author.

/2.9.6/ - Lecarme, O. and Desjardins, P. 1974, "Reply to a Paper by A. N. Habermann on the Programming Language PASCAL", SIGPAN Notices, Vol. 9, Nº 10, pp. 21-27.
Several criticisms of PASCAL concerning subranges and types are considered.

/2.9.7/ - Lecarme, O. and Desjardins, P. 1975, "More Comments on the Programming Language PASCAL", Acta Informática 4, pp. 231-243.
This article also includes a critical discussion of subranges and types in PASCAL.

/2.9.8/ - Liskov, B. 1974, "A Note on CLU", in CLU Design Notes, Project MAC, MIT, Cambridge, Mass.
The motivation underlying the development of CLU is discussed and an informal description of its data type facilities is given.

/2.9.9/ - Liskov, B. 1976, "An Introduction to CLU", Computation Structures Group Memo 136, MIT, Cambridge, Mass.
Not available to author.

/2.9.10/ - Liskov, B., Snyder, A., Atkinson, R. and Schaffert, C. 1977, "Abstraction Mechanisms in CLU", Comm. ACM, Vol. 20, Nº8, pp. 564-576.

This paper illustrates the use of data, procedural, and control abstraction by means of programming examples and shows how CLU modules are used to implement those abstractions.

/2.9.11/ - Liskov, B., et al 1978, "CLU Reference Manual", Computation Structures Group Memo Nº 161, Laboratory for Computer Science, MIT, Cambridge, Mass.
This manual includes a discussion of the data type definition facilities available in CLU.

/2.9.12/ - London, R., Shaw, M. and Wulf, W. 1976, "Abstraction and Verification in ALPHARD: A Symbol Table Example", Technical Reports, USC Information Sciences Institute, Marina del Rey, Calif. and Carnegie-Mellon University, Pittsburgh, Pa.
This paper designs, implements and verifies a general symbol table mechanism which illustrates the use of programmer defined abstractions.

/2.9.13/ - London, R., et al. 1978, "Proof Rules for the Programming Language EUCLID", Acta Informática 10, pp. 1-26.
Hoare-style proof rules are given which are applicable only to legal EUCLID programs and which deal with the various data types availables in EUCLID.

/2.9.14/ - Johnson, R. and Morris, J. 1976, "Abstract Data Types in the MODEL Programming Language", Proceedings of the Conference on Data: Abstraction, Definition and Structure, SIGPLAN Notices, Vol. 8,

Nº 2, pp. 36-46.
This paper describes the characteristics and suggested uses of abstract data types in the MODEL programming language.

/2.9.15/ - Habermann, A. 1973, "Critical Comments on the Programming Language PASCAL", Acta Informatica e, pp. 47-57.
This paper discusses some problems with PASCAL, including those caused by the confusion of ranges, types and structures.

/2.9.16/ - Wasserman, A. 1975, "Issues in Programming Language Design - An Overview", SIGPLAN Notices, Vol. 10, Nº 7, pp. 10--12.
This note identifies data types and abstraction as important issues in current work in language design.

/2.9.17/ - Morris, J., Jr. 1973, "Protection in Programmig Languages", Comm. ACM, Vol. 16, Nº 1, pp. 15-21.
This paper includes a discussion on protecting user-defined types from type violations.

/2.9.18/ - Palme, J. 1973, "Protected Program Modules in SIMULA 67", FOAP Report C8372-M3 (E5), Research Institute of National Defense, Stockholm, Sweden.
Not available to author.

/2.9.19/ - Palme, J. 1976, " New Feature for Module Protection in Simula", SIGLAN Notices, Vol. 11, Nº 5, pp. 59-62.
This note describes an extension for

protecting SIMULA classes from side-effects.

/2.9.20/ - Popek, G., et al. 1977, "Notes on the Design of EUCLID", Proceedings of an ACM Conference on Language Design

/2.9.21/ - Schaffert, C., Snyder, A. and Atkinson, R. 1975, "The CLU Reference Manual", Project MAC, MIT, Cambridge, Mass.
Not available to author.

/2.9.22/ - Scheifler, R. and Snyder, A. 1977, CLU Information Package, Computation Structures Group Memo Nº 154, Laboratory for Computer Science, MIT, Cambridge, Mass.
Not available to author.

/2.9.23/ - Shaw, M. 1976, "Abstraction and Verification in ALPHARD: Design and Verification of a Tree Handler", "Proceedings of the Fifth Texas Conference on Computing Systems, pp. 86-94.
Techniques for specifying and verifying abstractions in ALPHARD are illustrated by means of developing a program for manipulating the parse tree of an arithetic expression.
This paper introduces a way of specializing loop statements in ALPHARD to operate on abstract entities without explicitly depending on the representation of those entities.

/2.9.24/ - Welsh, J., Sneeringer, W. and Hoare, C. A. R. 1977, "Ambiguities and Insecurities in PASCAL", Software, Practice and Experience 7, pp. 685-696.

This paper includes a discussion of the equivalence of types in PASCAL.

/2.9.25/ - Wirth, N. 1975, "An Assessment of the Programming Language PASCAL", Proceedings of the 1975 International Conference on Reliable Software, SIGPLAN Notices, Vol. 10, No. 6, pp. 23-30.
Some problems with PASCAL related to features such as the concept of type union are discussed in light of program reliability.

/2.9.26/ - Wulf, W. 1974, "ALPHARD: Towards a Language to Support Structured Programs", Department of Computer Science, Carnegie-Mellon University, Pittsburgh, Pa.
Not available to author.

/2.9.27/ - Wulf, W., London, R. and Shaw, M. 1976, "Abstraction and Verification in ALPHARD: Introduction to Language and Methodology", Technical Reports, Carnegie-Mellon University, Pittsburgh, Pa. and USC Information Sciences Institute, Marina del Rey, Calif.
The data abstraction facilities of the ALPHARD programming language and the associated verification methodology are described.

/2.9.28/ - Wulf, W., London, R. and Shaw, M. 1976, "An Introduction to the Construction and Verification of ALPHARD Programming", IEEE Transaction on Software Engineering, Vol. Se-2, Nº 4, pp. 253-265.
This paper introduces ALPHARD by developing and verifying a data structure defi

nition and also shows how each language construct contributes to the development of the abstraction.

/2.9.29/ - Panorama des Languages d'aujourd'hui - - Carsege 14-22 Maio 1979 - Colóquio,bulletin du Groplan de l'AFCET Nº 8
'Panorama geral das linguagens modernas' equivalente a /2.3.28/.

/2.9.30/ - Paul Jacquet.
Abstraction et Genericité,
in /2.9.29/.

/2.9.31/ - K. Proch.
Parametrisation des unités de Compilation en A. T. M. Tése de DEA - Université Nancy.
'A expressão de tipos de dados genéricos em ATM. Contem uma comparação com as linguagens CLU e Alphard'.

/2.9.32/ - R. H. Campbell et all. An overview of Path Pascal design
SIGPLAN Set.80
'Path Pascal=Pascal+modulos+processos concurrentes+Path expressions'.

2.10 - Linguagens, fiabilidade e segurança de programação

/2.10.1/ - Gannon, J. and Horning, J. 1975, "The Impact of Language Design on the Production of Reliable Software", Proceedings of the 1975 International Conference on Reliable Software, SIGPLAN Notices, Vol. 10, Nº 6, pp. 10-22.
General language design principles are applied to specific language features such as hierarchies of data and type-

checking to enhance the reliability of programs.

/2.10.2/ - J. J. Horning.
Programming languages for Reliable Computing systems.
in /2.3.26/.
'Uma apresentação completa de todo o problema da segurança da programação, desde os aspectos humanos, linguagens, abstração, error-recovery, verificação e trabalho com as linguagens clássicas'.
Equivalente a /2.6.7/.
Ver também /2.6.21/.

/2.10.3/ - M. Loyer.
Comparaison des structures modulaires de quelques languages de programmation, in /2.9.29/.
'Comparação dos aspectos de modularidade, protecção, visibilidade e diferença de pontos de vista do utilizador e do implementador em Simula, Modula, Mesa e ADA'.

/2.10.4/ - G. J. Williams.
Programm checking.
University of Rocherter. 1979
'A dedução estática das propriedades das variáveis de um programa'.

2.11 - Sistemas de ajuda à especificação e à programação

/2.11.1/ - A. N. Haberman.
The Gandalf Project.
Carnegie Mellon Computer Science Research Review. 1979
'Descreve o projecto de um sistema de a

ajuda à programação em ADA'.

/2.11.2/ - R. Minot .

ATM: Un systeme de fabrication de programmes basé sur les concepts de modularité et type abstrait. These - CRIN - - Nancy. 1980 .
'Descreve um sistema de ajuda à programação baseado numa linguagem experimental definida para o efeito-ATM'

/2.11.3/ - V. Douzeau - Gouge et all .
Programming environments based on structured editors: the Mentor experience .
INRIA - Julho 1980
'Descreve-se um editor que reconhece a estrutura sintáctica da linguagem a editar - MENTOR'.

/2.11.4/ - B. Malése .
Manipulation des programmes Pascal au niveau des Concepts du language.
Thése - Université d'Orsay - Junho 1980
'Um sistema baseado em MENTOR para manipulação de programas Pascal'.

/2.11.5/ - J. Darlington and R. M. Burstall .
A system which automatically improves programs - in /2.3.6/
Equivalente à referência /2.3.7/.

/2.11.6/ - D. R. Musser .
Abstract data type Specification in the AFFIRM system .
IEEE Transaction on Software Eng. Jan. 1980 .
'Sistema interactivo para desenho de programas utilizando a especificação à

Guttag e à Hoare'.

/2.11.7/ - M. Demuynck et all.
Systeme ZAIDE d'aide à la Specification
Congrés AFCET 1980.
'Sistema de ajuda à especificação da linguagem Z'.

/2.11.8/ - R. Schneider e J. P. Thomesse
Les Réseaux de Petri dans l'aide à la conception d'applications temp reel et réparties.
Congresso AFCET 80.
'Sistema de ajuda à modelização e implementação de aplicações repartidas'.

2.12 - E quando alguns constrangimentos impõem a utilização de linguagens de baixo nível

/2.12.1/ - Jim Welch .
Structured programming in Macro Assembly Languages. Software Practice and Exp. Vol.8 371-376 1978 .
'Um conjunto de macros para programação estruturada em Macro11 - DEC PDP11'.

/2.12.2/ - Luís Arriaga da Cunha, João Duarte Cunha .
Algumas notas sobre utilização de MACROS em linguagens de assembler.
CI - LNEC, Lisboa, 1977
'Idem anterior para MACRO10'.

/2.12.3/ - ——— . Software for The Plotter Complot DP -8.
LNEC, Lisboa, 1978
'A implementação de software de controlo de um plotter com um modelo em Simula'.

/2.12.4/ - David E. Boddy.
Structured FORTRAN - With or without a preprocessor. SIGPLAN NOTICES - Abril 1977.
'Um pré processador permitindo a programação em FORTRAN com estruturas de controle modernas é descrito. O pré processador gera automaticámente

o programa FORTRAN STANDARD
Com if's e goto's'.

2.13 - Condução do projecto Software

/2.13.1/ - J. N. Buxton.
Software Engenieering, in
/2.3.6/

/2.13.2/ - W. M. Turski.
Software Engenieering - some
principles and problems, in
/2.3.6/

'Em ambos os artigos é apresentada uma visão semelhante, mas mais condensada, à do Prof. Brooks em /2.1.1/'.

3. - Desenho de um sistema concreto - estudo do monitor gráfico a instalar no GT42

/3.1/ - José A. Legatheaux Martins. Apresentação da implementação de GRI-GRI no sistema DEC10+ +GT42. - CIUNL 4/79, Lisboa, 1979

_____, GRI-GRI (Logiciel graphique interactive de base) Manual do Utilizador para o DEC-10 (Versão-1) Junho de 1979 - CIUNL-3/79, Lisboa, 1979

_____, A implementação do Software gráfico interactivo de base, GRI-GRI, no DEC10, balanço e perspectivas. Julho de 1979 - CIUNL - 6/79, Lisboa, 1979

/3.2/ - Nuno Lobo e Costa Pires. Implementação do Modulo de Gestão de imagem do Monitor gráfico para o PDP11 do LNEC (Projectos SAGRA-8o - UNL e SGL - LNEC) CIUNL-14/80 Lisboa Novembro, 1980
'Primeira versão do módulo'.

/3.3/ - J. Cunha, J. Legatheaux Martins, M. Quirino
Construção de um monitor Gráfico, I-parte - Gestão de imagens (Proj. SAGRA80 - UNL e SGL - LNEC)
CIUNL-6/80 Lisboa, Julho,1980
'Primeira especificação do módulo

módulo de gestão de imagem'.

/3.4/ - Nuno Lobo, Costa Pires.
Implementação final do Módulo de Gestão de imagem do Monitor gráfico para o PDP11 do LNEC (Projectos SACRA-80 - UNL e SGL - LNEC - Em vias de publicação) Lisboa, 1981.
'Segunda versão do módulo'.

/3.5/ - J. A. Legatheaux Martins
Construção de um Monitor Gráfico.
Documentação final da gestão de imagem (Projectos SACRA80 - UNL e SGL-LNEC - em vias de publicação) Lisboa, 1981.
'Documentação final dos módulos relacionados com a gestão de imagem'.

/3.6/ - J. Cunha, J. Legatheaux Martins, M. Quirino.
Primeira especificação do módulo de gestão dos periféricos (Relatório interno do LNEC em vias de publicação)
Lisboa 1981.
'Primeira especificação daquele módulo'.

/3.7/ - Jerry L. Apperson et all.
BLISS11 programer's manual
Computer Science Department.
Carnegie Mellon University.
Pittsburgh (Distribuído pela DECUS) 1972.

/3.8/ - K. L. Heninger.
Specifying Software Requirements for Complex Systems: New Techniques and their application.
IEEE transactions on Soft. Engenieering - Jan. 1980

'São descritas técnicas concretas a aplicar na especificação e documentação de sistemas complexos. No caso, um programa de ajuda à pilotagem de um caça supersónico. Descrição do input, descrição do output, relações entre ambos em termos de conditions e events'.

/3.9/ - C. A. R. Hoare.
Monitors: an operating system structuring concept.
CACM - Outubro 1974

'Um artigo clássico de introdução dos monitores'.

/3.10/ - P. Brinch Hansen.
Structured Multiprogramming.
CACM Julho 1972
'Primeira proposta de notações linguísticas para os Monitores'.

/3.11/ - P. Brinch Hansen.
The programming language Concurrent Pascal in /2.3.6/

'A primeira apresentação da linguagem Concurrent Pascal, pri-

meira linguagem com o conceito de Monitor'.

/3.12/ - R. C. Holt et all.
Structured Concurrent Programming with Operating System aplications. Addison Wesley, Toronto . 1978 .

'Introdução aos sistemas de operação baseada no essencial no conceito de monitor expresso na linguagem CSP/K'.

/3.13/ - P. le Guernic, M. Raynal.
L'expression du controle' des accés concurrents aux objects. in /2.3.28/.
Carsége 1979.
'As regiões críticas, monitores, expressões de caminho, contadores de estado e serializadores são comparados sob o ponto de vista de potência de expressão e métodos de provas'.

ANEXO 1

ANEXO 1

Uma solução possível para o módulo de gestão das comunicações

Num sistema como o monitor gráfico discutido neste trabalho estão presentes problemas de sincronização e protecção de recursos. Como é óbvio, a presença dos periféricos do PDP11 (lápis de luz, teclado, terminal alternativo, linha de comunicações e relógio de tempo real) colocam problemas deste tipo.

Para a sua solução é necessário dispor-se de:

- um mecanismo que distribua o tempo do processador pelos diferentes processos, assim como o consequente mecanismo de comutação de estado do processador.
- uma forma de exprimir regras de encadeamento das operações, isto é, primitivas de sincronização.

Estas funções são em geral asseguradas por um "real-time-scheduler".

Um modelo conceptual adequado para descrever este tipo de problemas consiste, por exemplo, em descrever o sistema em termos de monitores /3.9/, /3.10/.

Dentro deste modelo, todos os módulos que encapsulam dados (por exemplo, o módulo de gestão de imagem encapsula a "display file" e as tabelas necessárias à sua gestão) têm de assegurar que todas as operações que exportam, e que permitem manipular esses dados são indivisíveis e que também se vários processos acedem a um mesmo módulo deste tipo (monitor), um conjunto de filas de espera manipuladas pelas operações "wait" e "signal" asseguram que os acessos são apenas executadas nos momentos adequados.

Partindo da hipótese que não nos interessa complicar demasiado o sistema com a introdução do "real time

rão o mesmo até que esse tratamento termine (Exemplos: transmissão do último byte de uma operação Poli vector ou de uma translação em tempo real, ou ainda o "interrupt" de confirmação de uma operação de entrada que desencadeie a transparência de uma fila de espera do PDP 11 para o DEC 10)

No que se segue pressupomos que esta hipótese é ergonómica e funcionalmente compatível com o que se pretende do sistema.

Concepção da linguagem de comunicações

As mensagens transmitidas no sentido DECIO-PDP11 e vice versa têm a forma abstracta:

<cabeça> <byte>*<fecho>

Podendo cada um dos elementos <cabeça>, <byte> ou <fecho>, ser eventualmente composto por mais de 1 byte (1). Existirá necessáriamente no elemento <cabeça> um byte com o número interno da operação. Conceptualmente poderemos sempre ver a mensagem transmitida pelo DEC-10 para o PDP11 como a chamada de um procedimento cujo nome vem codificado na cabeça e cujos parâmetros de entrada estão em <byte>*. Uma resposta do PDP 11 para o DEC 10 é também a resposta do procedimento cujo nome segue na cabeça e cujos parâmetros de saída vão em <byte>*.

As operações desencadeadas pelo DEC10 no PDP11 terão sempre uma mensagem de retorno. Eventualmente, serão enviadas mensagens do PDP11 para o DEC10 que não correspondem a nenhuma resposta a uma operação, mas sim, uma resposta do PDP11 a uma acção (assincrona) do utilizador sobre um periférico (carregar em "carriage return" (CR) no segundo terminal por exemplo).

---

(1) byte: Conjunto de 8 bit's. Unidade de informação mínima de comunicação do PDP11 com o DEC 10.

beça com o número de operação. A linha de output é gerida em BUSY WAYTING *).

Logo temos:

```
módulo gestão_de_comunicações;
    importa definições do hardware e para
            metrizações várias;
    exporta enviar;
    procedure handler_linha_input_DEC10;
        (* procedure cuja execução é de-
        sencadeada pela chegada de um by
        te pela linha e cujo modelo é  o
        de um autômato que analisa a se-
        quência de bytes na entrada *)
    módulo control_comunicações;
      importa definições várias e parame-
        trizações do hardware;
      exporta guardar_número_operação,
              put,
              fecho,
              enviar,
              status;
      type buffer = array [1.. max] of byte;
      var buffer_in, buffer_out: buffer;
        operação,
        contador,
        status: byte; (* estado do autóma
        to *)
      procedure guardar_número_operação
        (N: byte);
            begin
             operação: = N; contador: =0
            end;
      procedure put (N: byte);
            begin
             contador: = contador + 1;
```

```
                2: begin
                     .
                     .
                     .
                     .
                     .
                   etc.
                   end (* case *).
                 end (* fecho *);
               begin (* inicialização do con-
               trolo de comunicação *)
                   status: = 1
               end modulo controle_comunica-
               ções;
           end modulo gestão_de_comunicação;
```

Para analisarmos o autómato handler_linha_input, de talhemos a linguagem de comunicação

<mensagem>::= <cabeça><byte> * <fecho>.

<cabeça>::= α A <byte>.

<fecho>::= α B .

<byte>::= α α | qualquer byte diferente de α.

Um byte especial, α, assinala um controlo, seguido de A é o início, seguido de B o fecho. Para transmitir um byte arbitrário e se quizer transmitir eventualmente o próprio α, transmite-se αα.

Estas definições são também tomadas em consideração por SEND-TO-DEC10.

Um autómato que analisa o input de acordo com aquela linguagem e que executa as funções semânticas adequadas será:

scheduler" e consequente pilha para cada processo, podemos fixar as seguintes formas de trabalho.

a) Os únicos processos presentes são:
   - Um processo de inicialização de todo o sistema e que após a inicialização ficará eternamente num ciclo de espera passiva.
   - Os "handlers" (1) dos periféricos de entrada cuja entrada em execução é assegurada pelo PDP 11.

b) Todas as operações exportadas por monitores são de tal forma desenhadas que para os casos em que "handlers" distintos as executam (directa ou indirectamente), encontrarão sempre as estruturas de dados coerentes.

Nesta forma de trabalho, a hipótese a) dispensa o mecanismo de comutação de estado (usamos apenas o já fornecido pelo PDP11), a hipótese b), exige apenas que as operações exportadas pelo monitor sejam indivisíveis visto que dispensa as operações "wait" e "signal".

Ora se associarmos a todos os "handlers" a prioridade mais alta esta condição está garantida pois durante a sua execução o processador não aceita interrupções.

Esta forma de trabalho, simples de implementar com a linguagem Bliss, degrada o carácter assincrono do sistema, pois todas as interrupções que encontrem o sistema num estado tal que o seu tratamento seja complexo, bloquea

---

Nota - Os periféricos de saída serão geridos em "BUSY WAYTING" (2) o que dispensa a introdução de "handlers" suplementares.

(1) - "handler" de um periférico, rotina de tratamento de um "interrupt" do periférico.

(2) - "BUSY WAYTING" ou ciclo de espera activo.

Estes aspectos recomendam vivamente que todas as operações exportadas pelos módulos do PDP11 e que correspondem a operações desencadeadas no DEC10, tenham sempre por uma questão de facilidade, apenas 2 parâmetros, "Buffer"de entrada e "Buffer" de saída.

O módulo que gere as comunicações com o DEC-10 é composto por um autômato de análise dos bytes recebidos pela linha que desencadeia operações sobre um módulo de controle da comunicação,local a toda esta componente do sistema.

Este módulo, por sua vez, exporta todas as operações necessárias ao autômato para proceder de acordo com o que recebe da linha assim como uma operação que pode ser assincronamente desencadeada pelos módulos que gerem as entradas, para transmissão de uma mensagem (não de retorno) para o DEC10.

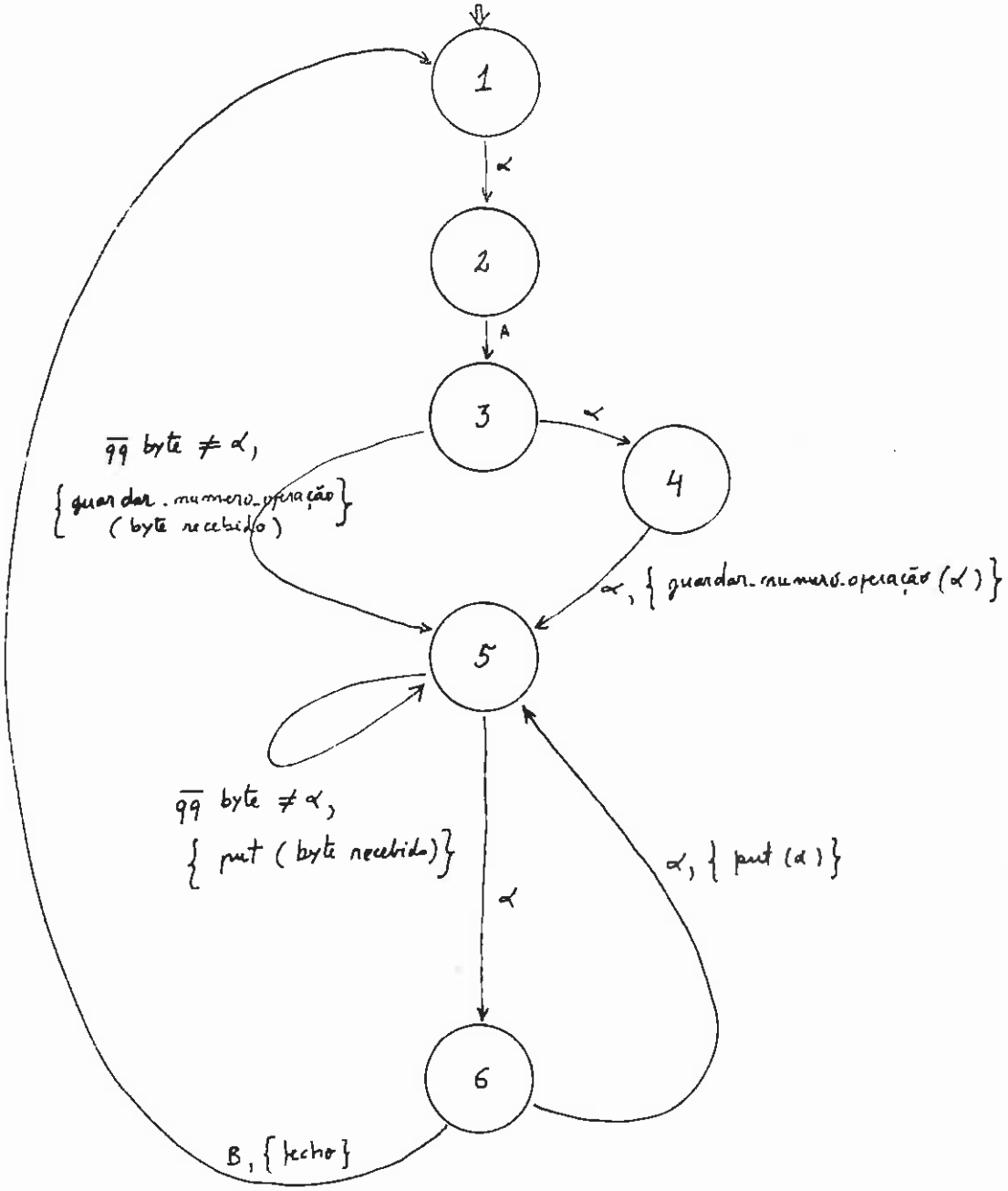
As operações são:

```
procedure guardar_número_operação (N:byte);
        (* Esta operação permite ao autómato registar o
           número da operação recebido e inicializar um
           buffer de input*)
procedure put (N: byte);
        (* Esta operação permite ao autómato acrescen-
           tar um byte ao buffer de input*)
procedure fecho;
        (* Esta operação desencadeia a execução da ope
           ração interna cujo número foi registado, ten
           do por parâmetros um "buffer de input". Quan
           do esta termina, fecho assegura a transmis-
           são do "buffer de output" para o DEC-10 (men
           sagem de retorno), através do procedure en-
           viar *).
procedure enviar (b: buffer; n, op: byte);
                (* Os n bytes presentes em buffer, são trans
                   mitidos  para o DEC10 antecedidos por uma ca
```

```
            buffer-in [contador]: = N
        end;
procedure enviar (b: buffer; n, op:
        byte);
        var I: byte;
        procedure send_to_DEC10 (b:
        byte);
        (* envia para o DEC10 em "bu-
        sy wayting" o byte b *);
    begin


        ( * enviar a cabeça da trans-
        missão *)
        for I: = 1 to N do send-to-
        -DEC 10 (buffer[I]);

        ( * enviar fecho *)
    end (*enviar*);
procedure fecho;
    begin
        case operação of
          1: begin
               incialização (buffer-
               -in, buffer-out);
               (* número interno da o
               peração de incializa-
               ção da gestão da ima-
               gem *)
               enviar(buffer-out, 1, 1);
               (* retorno é só 1 byte
               com o código de erro*)
               end;
```



Para o transformar num algorítmo consultar /2.3.43/, por exemplo.

ANEXO 2

ANEXO 2

Um modelo possível para o conceito de periférico lógico

Em computação gráfica interactiva, concebe-se em geral, que o programa de aplicação dialoga com o utilizador por meio de uma consola gráfica. (Por exemplo, o sistema em discussão é uma consola gráfica sofisticada). Deste modo, tal como qualquer sistema interactivo, terá funções de entrada e saída.

As funções de saída são, em geral, associadas à gestão de imagem. As entradas são, em geral, materializadas sobre uma entidade abstracta chamada "periférico lógico" (1).

Assim , o módulo de entradas deste sistema, tem por funções essenciais gerir periféricos lógicos de entrada e a sua análise foi motivação para uma reflexão sobre o conceito de "periférico lógico".

Os meios fisícos utilizados para o utilizador comunicar com o programa são, em computação gráfica, muito diversificados (teclados de funções, lápis de luz, ...), ultrapassando o quadro clássico do terminal alfanumérico.

Por outro lado, classicamente a comunicação do programa com o terminal alfanumérico faz-se, caracter a caracter, linha a linha ou, quando muito, por um conjunto de linhas. Em computação gráfica essas funções invocam geralmente objectos abstractos que podem não ter correspondência directa com um periférico fisíco.

A definição de periférico lógico, é, em si, um problema delicado se se quizer descrevê-lo de uma forma independente do material disponível, pois um conjunto dado de periféricos lógicos pode ser implementável sobre um conjun

---

(1) "Lógical device"

operação de signal( ) na "condition".

Formalmente um monitor implementa um tipo de dados abstracto cujas operações de manipulação são as operações exportadas que, por definição, se executam em exclusão mútua. Adicionalmente, os objectos "condition", munidos das operações Wait ( ) e signal ( ), permitem exprimir formas de sincronização mais complexas, dos processos que partilham o tipo de dados. (1)

Para aumentar a segurança da programação com monitores, as operações wait e signal só podem ser executadas dentro do monitor que contem a condition sobre a qual aquelas se aplicam.

Uma expressão linguística possível para um monitor pode ser a seguinte:

```
monitor identificador do monitor;
   uses lista de constantes, tipos e procedimentos
        importados;
   defines lista de procedimentos exportados;
   const...;
   type ...;
   condition ....;
         (*conditions privativas do monitor*)
   module identificador do módulo;
   uses...;
   defines;
      .
      .
      .
   end module identificador;
```

(1) No essencial as conditions servem para evitar formas de espera activa ("busy wayting").

ANEXO 2

Um modelo possível para o conceito de periférico lógico

Em computação gráfica interactiva, concebe-se em geral, que o programa de aplicação dialoga com o utilizador por meio de uma consola gráfica. (Por exemplo, o sistema em discussão é uma consola gráfica sofisticada). Deste modo, tal como qualquer sistema interactivo, terá funções de entrada e saída.

As funções de saída são, em geral, associadas à gestão de imagem. As entradas são, em geral, materializadas sobre uma entidade abstracta chamada "periférico lógico" (1).

Assim , o módulo de entradas deste sistema, tem por funções essenciais gerir periféricos lógicos de entrada e a sua análise foi motivação para uma reflexão sobre o conceito de "periférico lógico".

Os meios fisícos utilizados para o utilizador comunicar com o programa são, em computação gráfica, muito diversificados (teclados de funções, lápis de luz, ...), ultrapassando o quadro clássico do terminal alfanumérico.

Por outro lado, clàssicamente a comunicação do programa com o terminal alfanumérico faz-se, caracter a caracter, linha a linha ou, quando muito, por um conjunto de linhas. Em computação gráfica essas funções invocam geralmente objectos abstractos que podem não ter correspondência directa com um periférico fisíco.

A definição de periférico lógico, é, em si, um problema delicado se se quizer descrevê-lo de uma forma independente do material disponível, pois um conjunto dado de periféricos lógicos pode ser implementável sobre um conjun

---

(1) "Lógical device"

to muito variável de meios fisícos.

Básicamente, poderemos dizer que um periférico lógico é uma máquina abstracta que permite ao utilizador comunicar ao programa de aplicação um conjunto de valores de um determinado tipo de dados. A forma como a comunicação desse conjunto é feita ao programa (sincrona, assincronamente) é também característica do estado do periférico lógico.

Numa implementação concreta, o periférico lógico assegura ao utilizador um conjunto de possibilidades ergonómicamente desejáveis como, emendar erros, aperceber-se do valor dado, emendá-lo eventualmente, confirmar um dado fornecido, terminar a sequência, etc.

Para a realização destas funções de "assistência ao diálogo", o periférico lógico não só comunica com periféricos fisícos de entrada, como também com periféricos fisícos de saída, tal como a acção de leitura de um carácter pelo terminal alfanumérico desencadeia sempre a acção de ecoar o caracter lido no écran do mesmo.

Basicamente, a máquina abstracta periférico lógico, pode ser activada em um de vários estados. A forma de activação condiciona a forma de sincronização entre a máquina abstracta e o programa de aplicação. No essencial, na activação é especificado um cardinal máximo para o conjunto de dados que o utilizador deve ou pode fornecer. Após a activação, o utilizador é notificado deste valor e da acção que se espera ou que pode desenvolver. A sequência de acções do utilizador é então especificada por

((<acção de comunicar dado> . (<acção de o emendar><acção de comunicar dado>)*
. <acção de confirmação>)*(<terminação> +λ)

Por outro lado, as sequências de operações que o programa de aplicação pode fazer sobre um periférico lógico

são:

(⟨activação⟩ ⟨leitura⟩* ⟨desactivação⟩)*

Na activação, além de se especificar o cardinal máximo do conjunto de dados a receber, é especificado um modo de activação.

Os modos podem ser:

a) A pedido ("Request") (1)
Se o periférico é activado em modo "Request", o programa de aplicação é bloqueado até que o utilizador forneça, ou N dados, confirmados um a um, ou termine a sequência antes de fornecer N dados.
Após o desbloqueamento, a operação de leitura do periférico lógico, permite ao programa receber os valores fornecidos pelo utilizador ordenados por ordem de chegada.

b) Amostragem ("Sample"):
Se o periférico é activado em modo "Sample", o utilizador pode fornecer, confirmar, emendar, etc, tantos dados quantos queira. Qualquer acção de terminar é ignorada. O programa de aplicações não fica bloqueado. Sempre que executar a operação de leitura do periférico lógico, é-lhe fornecido o último dado comunicado pelo utilizador, isto é, o valor corrente do seu "buffer".

c) Acontecimento ("Event"):
Se o periférico é activado em modo "Event",o programa de aplicação não fica bloqueado e pode continuar a execução. O utilizador pode fornecer até N dados, confirmados um a um. A

---

(1) No que se segue usar-se-à a nomenclatura inglesa.

acção de Terminar é ignorada. O programa de aplicação sempre que executa uma leitura, recebe o conjunto de dados já fornecido até ao momento. Após a leitura, o utilizador pode fornecer outro conjunto de N dados e assim sucessivamente.

d) Interrupção ("Interrupt"):

Se o periférico é activado em modo "Interrupt", o programa de aplicação não fica bloqueado e pode continuar a execução.

O utilizador pode fornecer até N dados, confirmados um a um, ou terminar antes de N dados. Quer o utilizador termine, ou confirme o enésimo dado, um "handler" lógico do device lógico e da responsabilidade do programa de aplicação é activado.

Após a activação, a operação de leitura fornece os dados no entretanto comunicados pelo utilizador. Esta pode ser desencadeada pelo "handler" lógico ou pelo programa de aplicação.

Na falta de uma especificação adequada desta máquina abstracta, apresentaremos um seu modelo baseado no conceito de monitor. Para tal segue-se uma apresentação breve do conceito de monitor.

Conceito de Monitor

O conceito de monitor foi introduzido e desenvolvido por Hoare e Brinch Hansen, /3.9/, /3.10/, /3.11/.

Algumas linguagens modernas incluem o conceito de monitor, por exemplo: concurrent Pascal /3.11/, Modula

/2.6.18/ e CSP/K /3.12/. Em /3.13/ uma extensa comparação do conceito de Monitor com outros conceitos de expressão de sincronização é apresentada.

Informalmente, um monitor pode ser visto como um módulo que encapsula um conjunto de estruturas de dados e algoritmos.

O monitor exporta também um conjunto de operações que permitem aos processos manipular a estrutura de dados que o monitor encapsula.

Adicionalmente é garantida a indivisibilidade de cada operação exportada, isto é, se um processo está a executar uma operação do monitor, num ambiente de pseudo paralelismo, qualquer outro processo que tente executar uma operação do mesmo fica automáticamente bloqueado até que o primeiro complete a execução. Outras regras de sincronização mais complexas podem ser expressas através da possibilidade de o monitor conter "conditions", isto é, filas de espera. Um processo que executa a operação wait (condition), fica bloqueado na fila de espera associada à "condition" e liberta o monitor que a contem para que outros processos possam executar operações do monitor. Um processo que execute a operação signal (condition), fica bloqueado e o processo que está à cabeça da fila de espera associada à "condition" entra em execução.

Assim, um monitor pode ser visto como uma caixa negra à qual existem associadas tantas filas de espera como operações exportadas e "conditions" do monitor. Em cada momento um e um só processo pode estar executando algorítmos do monitor. Todos os outros processos que no mesmo instante necessitam de utilizar os recursos que o monitor fornece, das duas uma, ou são bloqueados nas filas de espera associadas à entrada nas operações exportadas, ou se encontram bloqueados numa fila de espera associada a uma "Condition" do monitor, esperando que outro processo execute uma

operação de signal( ) na "condition".

Formalmente um monitor implementa um tipo de dados abstracto cujas operações de manipulação são as operações exportadas que, por definição, se executam em exclusão mútua. Adicionalmente, os objectos "condition", munidos das operações Wait ( ) e signal ( ), permitem exprimir formas de sincronização mais complexas, dos processos que partilham o tipo de dados. (1)

Para aumentar a segurança da programação com monitores, as operações wait e signal só podem ser executadas dentro do monitor que contem a condition sobre a qual aquelas se aplicam.

Uma expressão linguística possível para um monitor pode ser a seguinte:

```
monitor identificador do monitor;
   uses lista de constantes, tipos e procedimentos
        importados;
   defines lista de procedimentos exportados;
   const...;
   type ...;
   condition ....;
         (*conditions privativas do monitor*)
   module identificador do módulo;
   uses...;
   defines;
      .
      .
      .
   end module identificador;
```

(1) No essencial as conditions servem para evitar formas de espera activa ("busy wayting").

```
var ...;
  (* dados encapsulados pelo monitor*)
procedure .....; (*procedimentos *;
begin
  (* operações de inicialização do monitor*)
end monitor identificador;
```

Modelo de Periférico lógico através de monitores

Um periférico lógico é um monitor que exporta 2 interfaces (2 conjuntos de operações), uma com o programa de aplicação e o seu "handler" lógico, outra com os "handlers" dos periféricos fisícos de input, e importa uma interface, a interface com os periféricos fisícos de saída.

Cada periférico lógico é parametrizado pelo tipo de dados que o caracteriza e que passaremos a designar por:

monitor-type.

O estado do monitor pode ser um dos 5 valores que a seguir se definem:

type estado = (Request, Sample, Interrupt, Event, Closed).

Definição das operações importadas e que dão acesso ao monitor que assegura a comunicação com os periféricos fisícos de saída:

a) procedure output-prompting (st: estado, N: integer); (1)
   Acção: o utilizador é notificado da activação do periférico lógico no estado st e do número de operações de entrada que pode executar.
b) procedure output-echoing (d: monitor-type); (2)
   Acção: Assegura o eco em resposta a uma acção de fornecer um dado pelo utilizador.
c) procedure output-erasing (3)
   Acção: "apagar" o dado anteriormente fornecido pelo utilizador.
d) procedure output-confirming; (1)
   Acção: "Confirmar" o último dado recebido.
e) procedure output-overflow;
   Acção: O utilizador é notificado que a fila

---

(1) "prompting" é a acção de um periférico notificando que espera uma acção determinada do utilizador.

(2) "Echoing" é a acção de assegurar um eco em resposta a uma acção do utilizador.

(3) "Erase" apagar.

(1) "Confirming" confirmar

de espera não comporta mais dados.

f) procedure output-wayting-to (N: integer);
Acção: O utilizador é notificado de quantos dados se espera que forneça ainda.

g) procedure output-close;
Acção: O utilizador é notificado de que o periférico lógico foi dasactivado.

Definição das operações exportadas para comunicação com o programa de aplicação.

a) procedure open (....);
Acção: O periférico lógico é activado num estado determinado.

b) procedure ler (...);
Acção: O programa de aplicação lê os dados acumulados no periférico lógico.

c) procedure close
Acção: O periférico lógico é desactivado

d) procedure wayting-interrupt;
Acção: Operação que deve ser desencadeada pelo handler lógico do periférico após se ter colocado o periférico em modo Interrupt.

Definição das operações exportadas para comunicação com os handlers dos periféricos fisícos de entrada. Estas operações são desencadeadas pelos periféricos fisícos de entrada quando as correspondentes acções são realizadas pelo utilizador.

a) procedure data-event (...); (1)
Acção: Operação a realizar pelo handler do periférico fisíco quando o utilizador dá 1 dado.

b) procedure erasing-event;
Acção: idem em resposta a uma acção de emen

---

(1) "Event" acontecimento, acção.

dar o dado

c) procedure termination-event;
Acção: idem para uma terminação

d) procedure help-event;
Acção: idem quando o utilizador executa uma acção de pedir esclarecimentos.

Modelo do periférico lógico:

```
Monitor periférico-lógico (monitor-type: type);
uses estado, output-prompting, output-echoing, out-
     put-confirming, output-overflow, output-wayting
     -to, output-close;
defines open, read, close, wayting-interrupt, data-
     event, erasing-event, confirmation-event, help-
     event;
const maximo-da-fila = ... ;
     (* dimensão máxima da fila de espera *)
condition wayting-in-request-mode,
     wayting-in-interrupt-mode;
module fila-espera (componente: monitor-type; max:in
     teger);
     (* exporta as operações de manipulação da fila
     de espera e é parametrizado em termos do tipo
     das componentes e do seu comprimento máximo *)
defines  reset,
         put,
         data, (*devolve o conteúdo da fila*)
         número; (* devolve o número actual de ele-
                    mentos na fila*)
end module fila-espera;
var
```

```
        max-data-events: 1 .. maximo-da-fila;
        buffer: monitor-type;
        st: estado;
        buffer-status: (emptv, available, confirmed);
    procedure open (st1: estado; N: 1.. maximo-da-fila);
        begin
        if st <>closed then ERROR
        else
        reset;
        st: = st1;
        if st <> sample then max-data-events:=end if;
        buffer-status:= empty;
        output-prompting (st, N);
        if st = request then wayt (wayting-in-request-
        -mode) end if
        end if
    end open;
    procedure wayting-interrupt;
        begin
              wayt (wayting-in-interrupt-mode)
        end wayting-interrupt;
    procedure close;
        begin
        if st =closed then ERROR
        else
            output-close;
            st:= closed;
            buffer-status:= empty
        end if
    end close;
    procedure read (var N: 1.. maximo-da-fila; d: fila);
        begin
            if st =closed then ERROR
            else if st = sample then
```

```
        if buffer-status = confirmed then
                              N: = 1
        else N:= 0 end if;
             d:= buffer
   else
        N: = number;
        d: = data;
        reset
   end if
end read;

   procedure data-event (d: monitor-type);
        begin
           if st <> closed then
                output-ecoing (d);
                buffer: = d;
                buffer-status: = available
           end if
        end data event;
   procedure erasing-event;
        begin
           if st <> closed and buffer-status = available
                                                 then
              output-erasing (buffer);
              buffer-status: = empty
           end if
        end erasing-event;

   procedure confirmation-event;
        begin
          if st = closed or buffer-status <>available then  return
               (* do nothing *) end if;
          buffer-status:= confirmed;
          output-confirming (buffer);
          if st<>sample and number < max-data-events then put (buf-
                                                              fer)
```

```
                else output-overflow end if;
            if number = max-data-events then
                if st = interrupt then signal (wayting-in-interrupt-
                                                              -mode)
                else if st = request then signal(wayting-in-request-
                                                              -mode)
                end if
            end if
        end confirmation-event;

procedure termination-event;
    begin
        if st = request then
            signal (wayting-in-request-mode)
            else if st = interrupt then
                    signal (wayting-in-interrupt-mode)
            end if
    end termination-event;
procedure help-event;
    begin
        if st in [request + interrupt] then
            output-wayting-to (max-data-events      - number)
        end if
    end help;




begin  (* inicialização do periférico lógico *)
    st : = closed  ;
    buffer-status : = empty
end monitor periférico lógico;
```

Conclusões

O conceito de monitor permite de uma forma adequada captar a abstracção do conceito lógico independente de qualquer intrepretação funcional suportada num certo hardware concreto.

O Modelo garante também a independência dos periféricos fisícos, nomeadamente dos handlers dos mesmos, tornando assim irrelevante para a descrição do periférico lógico qualquer particularidade de um certo material.

O modelo sugere uma implementação numa linguagem na qual se possam exprimir monitores e handlers de periféricos, como, mesmo realizando a implementação noutro tipo de linguagens, permite inferir a estrutura modular do código a desenhar.

Por outro lado, este exercicío põe em evidência a necessidade de nas linguagens de programação de sistema se disporem de possibilidades de expressão de unidades de compilação genéricas. Essas possibilidades são em geral fornecidas apenas para a expressão de tipos de dados genéricos. Neste caso, uma aplicação concreta, mostra que aquela possibilidade devia ser estendida a outro tipo de conceitos. Finalmente, é necessário verificar da adequação do modelo, ensaiando com diversos tipos de periféricos, munidos de diversas operações e implementados em diversos materiais.

É evidente que a implementação de todos os periféricos lógicos possíveis, ou geralmente propostos nas normas de standardização, com todas estas possibilidades não são suportadas pela maioria das consolas actualmente existentes.

Também acontece que certo tipo de periféricos fisícos sugerem interacções com o utilizador que tornam supérfluas certas funções.