10.03

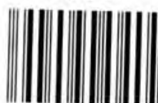
60

Paulo Alberto de Azeredo

Tratamento de Exceções em Ambientes Modulares

Tese apresentada ao Departamento de Informática da PUC/RJ, como requisito parcial para a obtenção do grau de Doutor em Ciências.

Orientador: Arndt von Staa

SABi


Departamento de Informática

Pontifícia Universidade Católica do Rio de Janeiro

Rio de Janeiro, 5 de dezembro de 1980

U F R G S
BIBLIOTECA
CPD/PGCC

Resumo

É estudado o problema de tratamento de exceções, com vistas a ambientes modulares. É feita uma caracterização inicial do problema, sendo identificados os requisitos que um mecanismo de tratamento de exceções em ambientes modulares deve possuir. A seguir, são discutidas e analisadas as propostas recentemente feitas para resolver o problema, sendo, cada uma delas, confrontada com os requisitos antes mencionados. É proposto, então, um mecanismo que assegura tais requisitos. A ênfase principal dada ao mecanismo proposto é a sua aplicabilidade a ambientes modulares e sua capacidade de verificação.

Abstract

The exception handling problem is studied, especially in the case of modular environments. The problem is characterized, and the requirements that a mechanism for handling exceptions in such environments must have are identified. Current approaches for dealing with the problem are analized and evaluated with respect to those requirements. A mechanism for handling exceptional conditions in modular environments that fits the requirements is proposed. The main characteristics of the mechanism are its applicability in modular environments and the enforcement of verifiability.

Dedico este trabalho
ao amor de minha esposa,

HELENA,

e de minhas filhas,

LUCIANA e GABRIELA.

Agradecimentos

Ao professor Arndt von Staa, pela orientação recebida.

Aos professores Carlos José Pereira de Lucena e Paulo Augusto da Silva Veloso, pelas valiosas sugestões e contribuições feitas a este trabalho.

Aos demais professores do Departamento de Informática.

Ao professor Manoel Luiz Leão, que me iniciou, no ano de 1967, na carreira de processamento de dados e que não se furtou, ao longo destes anos, de dar todo o apoio e incentivo que me permitiram concluir mais esta etapa de minha formação.

Aos amigos e colegas Clésio Saraiva dos Santos e José Mauro Volkmer de Castilho, cujas companhias, durante o doutoramento, renovaram e solidificaram antigas amizades.

Aos colegas e depois amigos Manoel Agamemnon Lopes, Tatuo Nakanishi, Atendolfo Pereda Bôrques e Bernardo Lula Júnior.

À Pontifícia Universidade Católica do Rio de Janeiro, pela hospitalidade com que me recebeu em seu seio.

À Universidade Federal do Rio Grande do Sul, principal financiadora deste projeto.

Ao Conselho Nacional de Desenvolvimento Científico e Tecnológico - CNPq e à IBM do Brasil, pelo apoio financeiro complementar.

E, finalmente, agradeço a meus pais e, acima de tudo, a Deus.

Sumário

Resumo i

Abstract ii

Dedicatória iii

Agradecimentos iv

Sumário vi

1 - Introdução 1

2 - Exceções 9

2.1 - Definição 9

2.2 - Terminologia 12

2.3 - Classificação 14

2.4 - Confiabilidade de Programas 33

3 - Modularidade 37

3.1 - Introdução 37

3.2 - Módulos 39

3.3 - Requisitos da Programação Modular 45

3.4 - Modularidade em ADA 53

3.5 - Tratamento de Exceções em Ambientes Modulares 58

4 - Requisitos 64

4.1 - Estratégias de Tratamento 65

4.1.1 - Local de Tratamento 65

4.1.2 - Associação Tratador-Exceção 68

4.1.3 - Retomada do Processamento 70

4.2 - Requisitos de um Mecanismo 72

Sumário

5 - Trabalhos Recentes 76

5.1 - [Berry, Kemmerer, von Staa, Yemini 79] 77

5.2 - [Goodenough 75c] 79

5.3 - [Horning, Lauer, Melliar-Smith, Randell 74] . 83

5.4 - [Ichbiah 79] 85

5.5 - [Levin 77] 88

5.6 - [Liskov, Snyder 79] 91

5.7 - [Parnas 72b] 94

5.8 - [Wulf 75] 95

5.9 - Comentários 98

6 - O Mecanismo Proposto 111

6.1 - O Modelo do Mecanismo 112

6.2 - Descrição do Mecanismo 126

6.3 - Método de Especificação 139

6.4 - Modelo de Implementação 141

7 - Validação do Mecanismo 153

7.1 - Verificabilidade 155

7.1.1 - Exemplo 157

7.2 - Encapsulamento 161

7.3 - Construtibilidade 162

7.4 - Acoplamento 163

7.5 - Adequação ao Ambiente 164

7.6 - Outros Exemplos 166

8 - Conclusões e Sugestões 171

Sumário

Bibliografia 183

Capítulo 1 - Introdução

As metodologias de programação recentemente desenvolvidas têm por objetivo criar métodos de produção de programas que permitem a obtenção de produtos úteis, econômicos, confiáveis, duráveis e eficientes [Bauer 72]. Nesse sentido, várias sugestões foram feitas a fim de que se pudesse atingir este objetivo. As mais importantes contribuições são as metodologias de programação estruturada, desenvolvimento 'top-down' e decomposição modular [Myers 78], [Dijkstra 72], [Mc Gowan, Kelly 75], [Wirth 73]. Estas metodologias visam criar meios para reduzir a complexidade dos programas a níveis compatíveis com a habilidade humana de compreendê-la, isto é, criar meios de exprimir, de forma natural, os componentes relevantes do problema real, sem criar artifícios para captá-los, uma vez que estes artifícios acrescentam, desnecessariamente, novos problemas à solução e, consequentemente, à sua compreensão.

Entre as ferramentas utilizadas nesse sentido, está o conceito de abstração, que permite que se definam objetos abstratos que capturem um dado conceito que ocorre no problema real. O conceito de abstração, que pode captar tanto aspectos de controle como de dados, associado à disciplinas de desenvolvimento de programas, nos permite atingir os objetivos das metodologias de programação. É necessário, entretanto, que a linguagem na qual será implementada a

abstração possua mecanismos não só para implementar a abstração em sí, mas também para permitir uma implementação natural do conceito, de tal modo que, ao se examinar a estrutura da implementação de uma abstração, se reconheça, nela, a estrutura do conceito implementado.

Quanto às estruturas de controle da linguagem, pode-se dizer que sua importância se torna secundária, se se dispõe de um mecanismo eficaz para a implementação de abstrações, pois de posse destas abstrações a organização do controle adquire relevância menor.

Assim, o que se deseja é a possibilidade de se definir unidades funcionais que implementam, cada uma delas, um conceito existente no problema real. Estas unidades funcionais, que denominamos <u>módulos</u> têm, como objetivo principal, permitir a diminuição da complexidade da solução, uma vez que o problema pode ser dividido em problemas menores e independentes um dos outros, cada um sendo expresso por um módulo. Na verdade, a técnica de produção de programas por meio de módulos, denominada <u>programação modular</u>, permite o desenvolvimento de programas complexos através da composição de soluções simples e independentes. Esta característica de independência permite, por sua vez, a reutilização dos módulos já produzidos, em outros problemas nos quais ocorra o mesmo conceito por ele implementado.

A partir da catalogação de um módulo, ele se torna disponível aos usuários, como uma nova função existente na linguagem, permanecendo a sua implementação invisível ao usuário. Este conhece as características do módulo apenas pela sua especificação, que indica o quê ele implementa e como usá-lo. Este encapsulamento da solução cria, por outro lado, a necessidade de haver mecanismos de proteção no módulo, para assegurar que ele será usado conforme o especificado ou, de uma forma geral, indicar a ocorrência de qualquer coisa anormal no módulo, seja qual for a sua origem.

Ao conjunto de 'coisas anormais' que podem ocorrer durante a execução de um módulo, damos o nome de <u>exceções</u>, as quais constarão da especificação do módulo, para indicar em que situações elas ocorrem.

As exceções não correspondem, necessariamente, a êrros no módulo, mas a qualquer evento para o qual exista a previsão de sua ocorrência e seja necessária uma resposta que não esteja prevista no código normal do módulo.

Entenda-se por 'código normal' o código necessário para implementar o conceito que o módulo respresenta. Por extensão, o código necessário para responder à uma exceção será denominado de 'código de exceção', que pode ser provido tanto pelo usuário como pelo próprio módulo, conforme o caso.

Independentemente de quem proveja o código de exceção, é necessário existir um mecanismo de controle que permita implementar este conceito.

Recentemente, várias propostas de mecanismos de controle de exceções foram apresentados, tentando resolver, cada uma à sua maneira, o problema antes mencionado [Berry, Kemerrer, von Staa, Yemini 79], [Goodenough 75c], [Horning, Lauer, Melliar-Smith, Randell 74], [Ichbiah 79], [Levin 77], [Liskov, Snyder 79], [Parnas 72b], etc.

Vemos, entretanto, que cada uma destas propostas apresenta algumas desvantagens (veja capítulo 5), quer seja por resolver parcialmente o problema, isto é, por sugerir uma solução para determinados casos de exceções, ou por propor soluções que requerem que se abra mão de importantes conceitos, os quais se sabe que sua utilização é uma grande contribuição ao desenvolvimento de módulos, tal como o encapsulamento da implementação.

As soluções parciais acima mencionadas parecem ter origem na tentativa de apresentar soluções que não acrescentem muitos novos detalhes à linguagem. De fato, a proliferação de detalhes e opções de uma linguagem é uma característica conflitante, pois ao mesmo tempo que acrescenta poder à linguagem, a torna mais complexa, desestimulando o seu uso, isto é, a linguagem termina pecando por excesso.

Com esses problemas em mente, propomos , neste trabalho, um mecanismo de tratamento de exceções que atende a dois objetivos principais: ser adequado a ambientes modulares e permitir a verificabilidade dos módulos que utilizam o mecanismo (estes aspectos são discutidos no capítulo 4).

Organização.

Este trabalho é composto por oito capítulos, sendo que o primeiro corresponde a introdução, onde é identificado o problema central, ou seja, a necessidade de se dar especial atenção ao tratamento de exceções, principalmente quando se desenvolve programas sob a metodologia de decomposição modular. Ainda na introdução é apresentado o objetivo principal da tese.

O capítulo 2 contém, inicialmente, uma pequena discussão sobre as definições propostas para o termo exceção. A seguir, propomos uma nova definição que exprime com mais fidelidade o sentido que é dado ao termo exceção, neste trabalho.

Ainda no capítulo 2 são identificados os diferentes tipos de exceções que normalmente ocorrem em programação sendo, ainda, proposta uma classificação para estas exceções. O objetivo da classificação é mais no sentido de fornecer uma orientação ao projetista de módulos de programas no que diz respeito a exceções, do que tentar esgotar as diferentes classes de exceções que eventualmente possam ocorrer. Por

isso, não pretendemos que a classificação proposta seja completa. Pensamos que pesquisas futuras possam contribuir nesse sentido.

O capítulo 2 finda com a discussão sobre a importância do tratamento de exceções na confiabilidade dos programas.

No capítulo 3 são discutidos diferentes aspectos referentes a modularidade e tratamento de exceções. O capítulo inicia com uma discussão de algumas definições existentes na literatura para o termo <u>módulo</u>, para, a seguir, propor nova definição. Na seção que se segue, são identificados e discutidos os requisitos da programação modular. São apresentados, ainda neste capítulo, os principais recursos existentes na linguagem ADA [Ichbiah 79] para a implementação de módulos. Esta apresentação é feita com o objetivo de mostrar uma linguagem que implementa modularidade conforme com o que é caracterizado neste mesmo capítulo.

Finalmente, o capítulo encerra com a caracterização do que vem a ser o tratamento de exceções em ambientes modulares e em ambientes não-modulares.

No capítulo 4 são identificados e discutidos os requisitos principais que um mecanismo de tratamento de exceções em ambientes modulares deve possuir.

O capítulo 5 contém um resumo das propostas mais recentes sobre o assunto. Ao final desse capítulo as

diferentes propostas são analisadas à luz dos requisitos formulados no capítulo 4.

O capítulo 6 propõe um mecanismo que atende aos objetivos formulados. O capítulo inicia com a apresentação do modelo do mecanismo, no qual uma série de conceitos, apresentados informalmente ao longo do trabalho, são formalizados. A seguir é feita a apresentação do mecanismo propriamente dito. A apresentação é feita em termos do modelo apresentado, sendo usadas regras de inferência para explicar a semântica dos comandos sendo, também, proposta uma sintaxe para as construções do mecanismo. Além da sintaxe e da semântica do mecanismo, propomos um método de verificação dos módulos que usam as contruções do mecanismo.

Ainda neste capítulo são sugeridos um método de especificação de módulos que usam o mecanismo, e um modelo de implementação para o mesmo.

O capítulo 7 mostra como o mecanismo proposto atinge os requisitos que foram inicialmente formulados. O processo de validação do mecanismo será conduzido de maneira informal, baseando-se, principalmente, na formulação de exemplos.

O último capítulo contém as conclusões do trabalho. Também nesse capítulo são feitas algumas sugestões para pesquisa futura nesta área.

O trabalho termina com a listagem da bibliografia utilizada no seu desenvolvimento.

Capítulo 2 - Exceçõoes.

2.1 - Definição

Por motivos históricos, o termo Exceção sempre esteve ligado a erros, principalmente aqueles provocados por defeitos do hardware [Levin 77]. Posteriormente, alguns autores estenderam o sentido da palavra, incluindo os erros devidos ao software [Avizienis 77], [Fregni, Ogus 74], [Levin 77], [Parnas 72c], [Ichbiah 79].

Segundo o manual da linguagem ADA [Ichbiah 79], no capítulo 12 - Exception Handling, o autor diz que

> "exceções são situações de erros que, embora raras, podem ocorrer, dado tempo suficiente".

Liskov [Liskov, Snyder 79] não dá, explicitamente uma definição para o termo, mas cita a palavra "unusual" ao se referir à exceções:

> "uma exceção indica que algo não usual (unusual) ocorreu".

A definição mais precisa dada até agora é a de Goodenough [Goodenough 75c]:

> "uma exceção é uma condição, detectada durante a execução de uma procedure, e devida a tentativa de executar uma determinada operação, tornando necessária a sinalização de sua ocorrência à procedure ativadora".

Esta definição, também utilizada por outros autores [Berry 79], [Melliar-Smith, Randell 76], sugere que exceções estão relacionadas com a execução de operações, isto é, que são problemas inerentes aos programas. Na verdade, a definição proposta por Goodenough é aplicável apenas a alguns casos de exceções, como adiante veremos.

Preferimos, no entanto, usar o termo exceção com um significado mais amplo do que aqueles que os autores citados procuram dar. No Novo Dicionário da Língua Portuguesa, de Aurélio Buarque de Holanda Ferreira, primeira edição, terceira impressão, encontramos a seguinte definição para exceção:

> "Exceção.[Do lat. exceptione.] S.f. 1. Ato ou efeito de excetuar. 2.Desvio da regra geral: seu caso constitui exceção. 3. Aquilo que se exclui da regra. 4. Exclusão (1): com exceção dele, foram todos à festa. 5. Privilégio, prerrogativa: Procura sempre valer-se das exceções. 6. Indivíduo cujo modo de agir ou de pensar difere do pensar comum. 7. ...".

O significado que daremos ao termo é semelhante a este, somente adaptado ao ambiente no qual estamos trabalhando. Assim, definiremos exceções da seguinte forma:

> "uma exceção é a ocorrência de um dado evento que identifica uma situação que exija um processamento diferente do normal".

O processamento normal, neste caso, corresponderia à regra, isto é, ao processamento aplicado aos casos em que não ocorram os eventos mencionados na definição.

Daremos, na sequência desta seção, a terminologia usada ao longo deste trabalho. Não se trata, aqui, de definição dos termos, mas apenas de caracterizar o sentido com o qual eles serão usados.

2.2 - Terminologia

a) detectar uma exceção

identificar a ocorrência de um evento que caracteriza a situação de exceção.

b) ponto de deteção

local onde é detectada uma exceção.

c) sinalizar uma exceção

provocar um sinal que indica a ocorrência de uma exceção.

d) ponto de sinalização

local onde é feita a sinalização de uma exceção.

e) sinalizador ou módulo sinalizador

módulo que contém o ponto de sinalização.

f) receptor ou módulo receptor

módulo que que recebe o sinal de exceção emitido pelo módulo sinalizador.

g) ativador ou módulo ativador

módulo que ativou o módulo sinalizador.

h) tratador.

código necessário para processar de forma adequada uma dada exceção. Este termo está sendo usado como uma tradução de "handler".

i) <u>tratar uma exceção</u>.

executar o tratador.

2.3 - Classificação das Exceções.

Nesta seção são identificados os diferentes tipos de exceções que ocorrem em programação, sendo proposta uma classificação segundo os aspectos de natureza, origem e causa.

2.3.1 - Natureza.

A natureza de uma exceção diz respeito à sua formação, isto é, às razões de sua ocorrência. A classificação das exceções segundo este aspecto não envolve, necessariamente, ambientes modulares, uma vez que ela mostra apenas como as exceções se formam.

A ocorrência de uma exceção na solução de um problema só pode ser devida a um de tres motivos: a exceção já existia no problema; ou foi introduzida artificialmente na solução ou, ainda, foi gerada pelo ambiente no qual se processa a solução.

Ao primeiro tipo, cuja existência é natural, damos a denominação de exceções do problema. Para caracterizar mais precisamente este tipo de exceção, podemos dizer que o problema contém uma exceção quando ocorre (ou pode ocorrer), neste problema, um fato real que foge das características principais do objeto. Como exemplos, citamos:

a) queda de uma ligação telefônica;

b) o fato de um ítem de um estoque de materiais atingir seu ponto de reposição;

c) uma "condição de alarme" em um processo industrial.

À classe das exceções que não existem no problema real, mas são introduzidas na solução, damos o nome de exceções da solução, que podem surgir em qualquer uma das diferentes fases do processo de desenvolvimento do programa. Assim, uma exceção da solução pode ser introduzida na especificação da solução ou na implementação ou, ainda, na representação dos dados usados na implementação.

Como exemplo de exceções da solução, citamos:

a) "integer overflow";

b) "stack underflow";

c) "divide by zero".

Finalmente, consideraremos o caso em que a exceção não existe no problema, nem na solução proposta para ele, mas sim no ambiente em que se desenvolve a solução. Denominamos de ambiente da solução, ao conjunto de recursos e procedimentos utilizados para conduzir a solução, tais como: operação, dados, energia elétrica, linhas de transmissão de dados, equipamentos em geral.

Quando um desses recursos ou procedimentos não contribui para a solução da forma desejada ou, então, é utilizado um recurso inadequado ao caso, e se este fato interferir, de algum modo na solução do problema, dizemos que ocorreu uma exceção do ambiente.

Como exemplo dessas exceções, citamos:

a) submeter os dados errados a um programa;

b) utilizar uma linha de transmissão inadequada;

c) deixar de colocar o "anel de gravação" em uma fita magnética a ser gravada;

d) usar um programa com finalidade diferente daquela para a qual ele foi projetado.

2.3.2 - Origem.

A origem de uma exceção refere-se à relação existente entre os módulos sinalizador e receptor. Em outras palavras, distingue entre os casos de exceções sinalizadas dentro do escôpo léxico do módulo receptor ou fora deste.

Tradicionalmente, a ocorrência de exceções sempre esteve ligada à execução de módulos [Levin 77], [Berry, Kemerrer, von Staa, Yemini 79], [Ichbiah 79], [Goodenough 75c], [Liskov, Snyder 79]. Além disso, sempre houve consenso de que uma exceção é causada no interior de um módulo em execução,

de modo que, no seu código ou no código de seu ativador devam estar previstas as ocorrências de <u>todas</u> as condições que resultam ou podem resultar em exceções.

Este entendimento é verdadeiro, mas não é completo, no sentido de que <u>nem todas</u> as exceções que afetam um módulo são sinalizadas no seu escôpo. Existe uma categoria de exceções que são sinalizadas fora do escôpo de um módulo e que, de alguma forma, interfere na sua execução.

Suponha o caso em que o usuário de um programa interativo deseje, por algum motivo, suspender a execução de um módulo em execução e, posteriormente, retomar a sua execução ou, então, fazer com que o módulo reinicie, possivelmente com novos parâmetros, ou deseje, ainda, abortar a sua ativação. Naturalmente, para interferir em um módulo da forma acima descrita é necessário que haja uma interferência externa sobre este módulo.

A esta classe de exceções denominamos, genericamente, <u>exceções exógenas</u>, que podem tanto ser causadas manualmente (pelo operador ou usuário) ou automaticamente.

Embora este tipo de exceção seja muito similar ao conceito de <u>interrupção</u>, preferimos não usar esta denominação, por se tratar de uso específico de interrupção em módulos e tambem pelo fato de que o conceito será usado com o sentido mais amplo do que o usual [Tanenbaum 76], [Madnick,

Donovan 74], [Mc Glynn 76], uma vez que um módulo, como veremos, poderá ser interrompido em pontos semanticamente razoáveis (definidos pelo usuário).

Quando, por outro lado, a sinalização de uma exceção ocorre dentro do escôpo léxico do módulo ativador, dizemos que ocorreu uma exceção endógena.

Os motivos que provocam a ocorrência de exceções endógenas, formam o último aspecto de classificação, que é a diferenciação destas exceções quanto à sua causa.

2.3.3 - Causa.

As exceções que ocorrem durante a execução de um módulo e que são sinalizadas pelo próprio módulo podem ser devidas a diferentes causas. Identificamos, inicialmente, aquelas exceções provocadas pela tentativa de executar uma operação com um operando que não pertença ao domínio da operação. Às exceções devidas a este motivo damos o nome de exceções de - domínio e correpondem, essencialmente, ao que Goodenough [Goodenough 75c] denomina de "domain failures". Como exemplo de exceções de domínio citamos:

a) tentativa de divisão por zero;

b) tentativa de resolver um sistema linear que não tem solução;

c) tentativa de aplicar a função "arcsen" sobre um número maior que 1.

A segunda classe de exceções endógenas são aquelas provocadas pela transformação efetuada sobre um dado. Isto significa que a ocorrência destas exceções está relacionada com a execução de uma operação que transforma um dado. Quando esta transformação é tal que a mudança do dado implica em uma mudança no seu estado, dizemos que ocorreu uma <u>exceção de estado</u>.

Antes de prosseguir no exame das exceções de estado, é conveniente caracterizar aquilo que denominamos de estado de um dado.

2.3.3.1 - <u>Estado de um Dado</u>.

Ao conjunto de definição de um tipo de dado, associamos um conjunto de estados, de tal forma que à cada elemento do primeiro conjunto corresponda um e somente um elemento do segundo e vice-versa.

Seja D = { v1, v2, v3, ..., vn }

o conjunto de definição de um tipo de dado T,

e seja S = { s1, s2, s3, ..., sn }

o conjunto dos estados associados a D.

Dizemos que um dado do tipo T está no estado si, se e somente se o seu valor é vi, isto é,

S = si <=> V = vi

Sobre o conjunto S de estados, definimos uma relação de equivalência R que determina uma partição P(R) de S.

Seja R[x] uma classe de equivalência formada pelo conjunto dos estados si que pertencem a S tais que:

R[x] = { si: x R si }

Uma partição P(R) de S é a família dos sub-conjuntos não vazios de S, cuja união é igual à S:

P(R) = {B: (E x)(B = R[x] & B ≠ ∅)}

Chamemos R[x] de <u>classe de estado</u>. Neste caso, podemos denominar a partição P(R) de <u>conjunto de classes de estados</u>.

Seja <u>op</u> uma operação da definição de um tipo de dado T. Dizemos que a operação efetua uma transformação sobre um dado <u>d</u> do tipo T, se o estado do dado <u>d</u> pode ser alterado pela operação. Usaremos a notação

s1 <u>op</u>(d) s2

para indicar a situação acima descrita, onde s1 é o estado do dado <u>d</u> <u>antes</u> da execução da operação e s2 o estado <u>após</u>.

Quando, após a execução da operação op(d), tivermos $s1 \neq s2$, dizemos que a operação provocou uma mudança no estado de d.

De forma análoga, podemos caracterizar a transformação em termos de mudança de classe de estados:

R[x1] op(d) R[x2]

Dizemos que um dado muda de classe de estado, quando ele passa de um estado s1 para o estado s2, tais que:

$s1 \in R[x]$ & $s2 \notin R[x]$

Retornando ao estudo dos problemas de exceções, dizemos, finalmente, que ocorre uma exceção de estado Es, quando uma operação sobre um dado d provoca uma mudança de classe de estado deste dado:

Es(d) <=> s1: $s1 \in R[x]$ op(d) s2: $s2 \notin R[x]$

isto é, quando a operação provocou uma mudança de classe de estado tal que:

s2: $s2 \notin R[x]$ & $s2 \in S$;

Neste caso, a operação é executada com sucesso, porem a transformação por ela causada é que se considera como sendo uma exceção. Estas exceções tambem podem ser usadas para dar significado a um resultado, ou para monitorar o andamento de um processo.

U F R G S
BIBLIOTECA
CPD/PGCC

Não caracterizaremos como sendo uma exceção de estado o caso em que

$s2 \notin S$

isto é, o caso em que o resultado produzido pela operação não pertence ao conjunto dos resultados possíveis. A ocorrência destes casos são devidos, como veremos adiante, a dois fatores:

a) erros do hardware;

b) erros de programação.

Podemos, no entanto, enriquecer o conceito de mudança de estado, se permitirmos a inclusão de outros domínios entre as partições que definem as classes de estado.

$s2 : s2 \notin R[x] \; \& \; s2 \in S \cup S'$

onde S' é a classe artificialmente incluida.

Como exemplo de exceções de estado citamos as seguintes:

a) a quantidade de um determinado ítem de estoque atingiu ou ultrapassou o ponto de reposição. Neste caso, o dado passou do estado "estoque normal" para o estado "fazer pedido de reposição", que constitui uma exceção que requer um processamento próprio;

b) a quantidade de pedidos de um dado produto ultrapassou o limite de produção daquele produto;

c) a quantidade de alunos que desejam se matricular em uma dada disciplina de um curso ultrapassou o limite admissível para uma turma, sendo necessário criar nova turma e nela alocar os excedentes.

d) "end of file".

A terceira classe de exceções endógenas congrega as exceções que surgem devido ao método usado para resolver o problema. Alguns problemas, principalmente aqueles da área da Análise Numérica só podem ser resolvidos se for usado um dado método de solução, isto é, alguns problemas não possuem uma solução geral que possa ser aplicada para qualquer caso.

As exceções causadas por este motivo, isto é, pela tentativa de utilizar um método inadequado de solução, são denominadas de <u>exceções do método</u>.

Observe que, para alguns problemas existe uma solução universal, isto é, um método de solução que pode ser aplicado em <u>qualquer</u> problema daquela natureza, como por exemplo, o método de eliminação Gaussiana [Albrecht 73] para inversão de matrizes pode ser usado para inverter <u>qualquer</u> matriz que possua inversa. Caso ocorra uma exceção na inversão de uma matriz por este método, certamente não será por causa da aplicabilidade do método.

Por outro lado, não se pode fazer afirmação idêntica para o caso do uso do método de Gauss-Seidel para a solução de sistemas lineares não-homogêneos [Albrecht 73]. Este método não é aplicável a qualquer sistema linear. Por exemplo, caso se queira resolver o sistema Ax=B, onde:

$$A = \begin{vmatrix} 1 & 2 & -2 \\ 1 & 1 & 1 \\ 2 & 2 & 1 \end{vmatrix} \qquad B = \begin{vmatrix} 1 \\ 3 \\ 5 \end{vmatrix}$$

usando uma implementação do método de Gauss-Seidel, certamente ocorreria uma exceção indicando que que o método não está convergindo para uma solução, embora esta exista [Albrecht 73]:

$$x = \begin{vmatrix} 1 \\ 1 \\ 1 \end{vmatrix}$$

Certamente esta exceção não ocorreria se o método de solução adotado fosse Jacobi [Albrecht 73]. Neste caso, a denominação de <u>exceções do método</u> deixa claro que a exceção ocorreu exclusivamente por se ter usado o método inadequado de solução.

A ultima classe de exceções endógenas que consideraremos é a classe das exceções provocadas pela ocorrência de erros devidos a falhas no funcionamento do "hardware" ou a defeitos no "software".

A ocorrência de erros de hardware e/ou software tem sido alvo de muitos estudos recentes [Randell 75], [Wulf 75], [Horning, Lauer, Melliar-Smith, Randell 74], [Avizienis 77], [Parnas 72b]. Estes autores concordam, todos sobre dois pontos:

1) por maior que seja o desenvolvimento tecnolóogico na área de hardware, os componentes de um computador têm probabilidade não-nula de falharem;

2) apesar do desenvolvimento de novas metodologias de programação, de técnicas de verificação e de testes sistemáticos, os erros residuais de programação continuarão a constituir um entrave à confiabilidade dos programas.

O estudo destes problemas gerou o conceito de "fault-tolerance". Diz-se que um sistema ou programa é tolerante à falhas, se ele é capaz de produzir resultados confiáveis, a despeito da ocorrência de erros.

Naturalmente, estes erros, quer sejam de software ou de hardware, são características indesejáveis de um sistema de computação, sendo, portanto, necessário prever-se, nos programas, a possibilidade de sua ocorrência, que irão gerar, por sua vez, exceções, às quais poderá ser dado o tratamento adequado que garanta a confiabilidade do programa.

Como vimos, os erros podem ter origem no software ou no hardware. Erros de software são aqueles causados por defeitos de programação, que não foram detectados nos processos de verificação e de testes.

Erros de hardware são causados por falhas no funcionamento do equipamento que, por sua vez, são provocadas por defeitos. O defeito no equipamento pode ser de projeto, de fabricação ou de desgaste. Quando o defeito de um equipamento se manifesta, ele provoca uma falha que, por sua vez, causa um erro no programa.

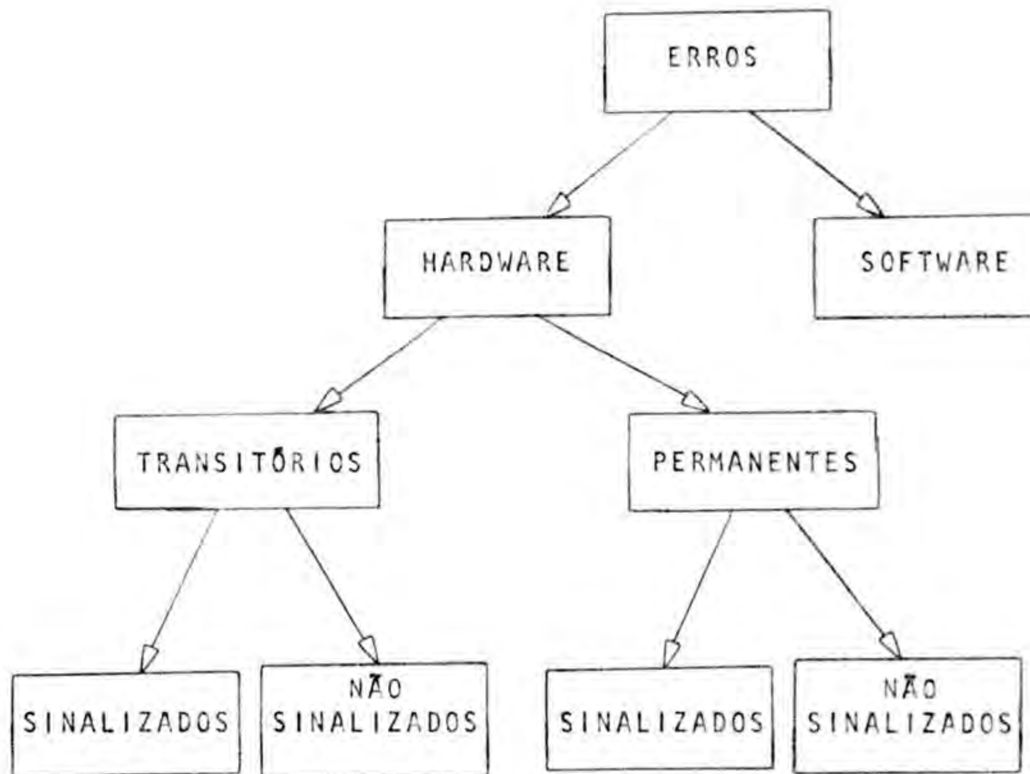
Os defeitos podem se manifestar de duas formas diferentes: de forma intermitente ou de forma permanente. A primeira causa erros chamados transitórios, no sentido de que, em condições idênticas futuras, eles poderão não se repetir. A segunda forma causa erros também denominados de permanentes, sentido de que, em condições idênticas futuras, o mesmo erro se repetirá. Sob o ponto de vista prático,

consideramos um erro como permanente, se ele ocorrer um dado número de vezes, pré-estabelecido para cada caso. Por exemplo, podemos estabelecer que ocorre um erro permanente de leitura de fita magnética, se a leitura falhar após, digamos, dez tentativas.

Além da divisão transitórios-permanentes, os erros de hardware são divididos entre <u>sinalizados</u> e <u>não-sinalizados</u>. Erros sinalizados são aqueles detectados pelo próprio hardware, provavelmente por um sistema de redundância [Avizienis 77], sendo a sua existência notificada programa que solicitou a operação, através de um mecanismo de interrupção. O erro mais comum desta categoria é o erro de paridade na transmissão de dados.

Os erros não-sinalizados são aqueles para os quais não existe um sistema de detecção, ficando a sua detecção, por conta do usuário.

A figura abaixo mostra, esquematicamente, os erros acima mencionados:



Detecção de Exceções.

Para fins de detecção, consideraremos os erros divididos em duas categorias: os sinalizados e os não-sinalizados, sendo que incluiremos, na segunda categoria, os erros de software, uma vez que, por se tratar de defeitos indesejados, a sua existência é desconhecida.

Note-se que a detecção de um erro não implica,

UFRGS
INSTITUTO DE INFORMÁTICA
BIBLIOTECA

necessariamente, na identificação da falha que o causou.

Erros Sinalizados.

Em se tratando de erros sinalizados, a sua detecção é trivial, uma vez que o próprio sistema reconhece a sua ocorrência e notifica o usuário. Resta ao usuário, apenas a tarefa de prever, em seu programa, a possibilidade de ocorrência deste tipo de erro (sinalizado), a fim de prover o tratamento adequado.

Erros não-sinalizados.

A detecção da ocorrência de um erro não-sinalizado em um programa não é uma tarefa simples, uma vez que sua natureza é desconhecida. Ao contrário dos erros sinalizados, cuja ocorrência pode ser prevista, os não-sinalizados são de difícil previsão, no sentido de que não se pode antecipar as circunstâncias nas quais eles ocorrerão, nem tampouco quando ocorrerão. A única certeza que temos sobre estes erros é que a sua ocorrência provavelmente prejudicará os resultados produzidos pelo programa.

Podemos, no entanto, detectar a ocorrência de um erro não-sinalizado, através de asserções sobre os resultados de uma computação. A verificação das asserções pode ser visto como um teste de aceitação do resultado ou resultados obtidos. Naturalmente, iremos supor que não ocorra nenhum erro não-sinalizado durante a verificação da asserção.

O projeto do sistema determinará não só a natureza como a granularidade das asserções.

A natureza refere-se ao quê será testado. A asserção não é necessariamente a mesma sugerida por Manna [Manna 74] para verificação da correção do programa.

A granularidade diz respeito à quantidade de computação efetuada entre duas asserções consecutivas. Se a granularidade for pequena, por exemplo uma computação, a ocorrência de um erro poderá ser não somente detectada, como também identificada. No caso contrário, isto é, quando existem muitas computações entre duas asserções consecutivas, também será possível detectar o erro, porém a sua identificação será mais difícil.

A identificação do erro é vantajosa, pois no caso de se tratar de um erro software, ele poderá ser corrigido. Por outro lado, a busca microscópica de erros produz a desvantagem de aumentar o custo do programa, pois certamente ele demandará maior tempo de projeto, de codificação, de execução, além de se aumentar, com isso, a probabilidade de serem introduzidos outros erros no programa.

A alternativa adequada é verificar unidades funcionais de programas, tais como módulos, blocos, procedures, etc, inserindo-se as asserções devidas após cada uma destas

unidades para verificar se ocorreram erros que afetaram os resultados produzidos, conforme o rigor desejado.

O quadro abaixo mostra os diferentes tipos de exceções em conjunto, para que o leitor possa ter uma visão global da classificação proposta.

<u>Classificação das Exceções</u>

1 - Quanto à natureza.

    1.1 - Exceções do problema.
    1.2 - Exceções da solução.
    1.3 - Exceções do ambiente.

2 - Quanto à origem.

    2.1 - Exceções exógenas.
    2.2 - Exceções endógenas.

3 - Quanto à causa.

    3.1 - Exceções de domínio.
    3.2 - Exceções de estado.
    3.3 - Exceções do método.
    3.4 - Erros.

2.4 - Exceções e confiabilidade de programas.

Confiabilidade (1)de software é um objetivo atingível? Depende do que considerarmos como sendo a confiabilidade.

William Wulf [Wulf 75] diz que é absolutamente seguro afirmar que o aumento da confiabilidade é o objetivo de todos. Diz, também, por outro lado, que não é seguro afirmar que todos concordam, ou mesmo compreendam, o que se entende por confiabilidade.

Segundo Wulf, a confiabilidade de programas não pode ser vista de forma absolutista: não é realista fixarmos metas absolutas de confiabilidade, como, por exemplo, totalmente confiável, uma vez que a confiabilidade do software não é caracterizada apenas pelos erros que ele contém, mas sim pelos efeitos produzidos por eles.

Glenford Myers [Myers 76] define a confiabilidade de programas como sendo

> "a probabilidade de que o programa irá funcionar, sem erros, durante um dado período de tempo, ponderado pelo custo, ao usuário, de cada erro que ocorrer"

Ou seja, a confiabilidade é uma medida do impacto causado pelos erros do programa, sobre o usuário.

----

(1) O termo confiabilidade está sendo usado, neste texto, com o sentido da palavra inglesa reliability.

Embora a confiabilidade dos programas seja uma medida difícil de ser obtida, pois não se conhece a probabilidade de que o programa irá funcionar sem erros durante um período T, o que se deseja, na maioria dos casos não é calculá-la, mas sim assegurar-se que ela seja a maior possível, para o caso em questão.

Para se conseguir este objetivo, devemos usar metodologias que, de forma empírica ou sistemática, asseguram o aumento da confiabilidade dos programas.

Ao nosso ver, qualquer tentativa que vise aumentar a confiabilidade de programas deve incluir:

a) o uso de metodologias de programação adequadas;

b) verificação de programas;

c) testes sistemáticos;

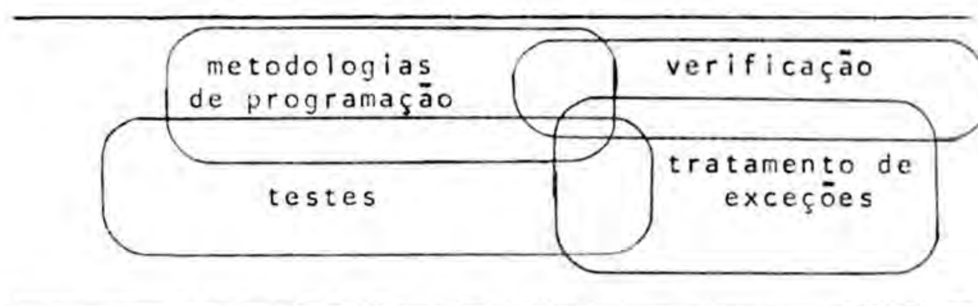
d) tratamento de exceções.

Estas quatro técnicas, que consideramos complementares, podem, se bem utilizadas, aumentar a confiabilidade dos programas aos níveis de tolerância que a aplicação exige.

Embora não se possa quantificar o aumento da confiabilidade dos programas, pode-se assegurar ([Avizienis 77], [Bauer 71], [Boehm 76], [Myers 76], etc) que a aplicação destas

técnicas diminue a probabilidade de ocorrerem falhas na sua execução.

A figura abaixo ilustra a contribuição dada por cada uma das técnicas citadas, na composição da confiabilidade de um programa.



A figura acima foi feita sem qualquer preocupação com proporções. A contribuição dada por cada uma das técnicas na confiabilidade varia de caso a caso.

Não nos deteremos na discussão de como o emprego de técnicas de verificação, testes e de metodologias de programação podem melhorar a confiabilidade de programas, pois este assunto tem sido bastante explorado na literatura (vide [Myers 76] e as referências citadas naquela obra).

Tratamento de exceções pode contribuir para a melhoria da confiabilidade de duas formas. A primeira atuando como base de uma metodologia de programação, pois se considerarmos a existência das exceções, em todas as suas diferentes formas, desde o início do projeto de desenvolvimento do programa, estaremos contribuindo para a criação de um produto

mais confiável. A segunda forma é utilizando o tratamento de exceções para tratar das falhas residuais de programação que, uma vez detectada a ocorrência de uma delas, pode ser manipulada no sentido de evitar que o programa entre em colapso, pois corrigí-lo seria muito difícil, uma vez que erros residuais são absolutamente imprevisíveis e, portanto, seria quase impossivel antecipar um tratamento para cada caso de erro que possa ocorrer. O que se tenta, então, é minimizar os seus efeitos, mantendo a integridade do programa, para que este possa prosseguir a sua execução, a despeito da ocorrência dos erros.

Capítulo 3 - Programação Modular.

3.1 - Introdução.

A programação modular é uma técnica cujo objetivo é permitir o desenvolvimento de partes de programas denominadas módulos, que implementam, cada uma delas, um dado conceito.

A divisão do programa em módulos permite, pois, diminuir a complexidade da solução a níveis compatíveis com a habilidade humana de compreendê-la. A técnica de produção de programas por meio de módulos, denominada programação modular, permite o desenvolvimento de programas complexos por meio da composição de soluções simples e independentes. Esta característica de independência permite, por sua vez, que os módulos sejam programados, verificados, testados, documentados e catalogados separadamente, de forma a permitir o seu uso sempre que o conceito por ele implementado se torne necessário no desenvolvimento de um programa ou outro módulo.

As vantagens do uso desta metodologia não residem apenas na redução da complexidade do programa. A programação modular, se bem feita, introduz, paralelamente, uma série de vantagens, que implicam no aumento da qualidade do "software" assim desenvolvido, tais como:

a) simplificação da verificação;

b) melhoria da legibilidade;

c) facilidade de extensão e documentação.

Espera-se obter, finalmente, um produto que possua as características de correção, confiabilidade, manutebilidade e extensibilidade. Estes fatores devem contribuir, no seu conjunto, para a diminuição do custo final dos programas.

A fim de se poder atingir estes objetivos, é necessário que os módulos produzidos possuam determinadas características, tais como: construtibilidade, encapsulamento da implementação, baixo acoplamento, alto grau de coesão e verificabilidade. Programas ou sistemas que possuem as características acima são denominados de <u>fortemente modulares</u>. Qualquer tentativa para resolver o problema de tratamento de exceções dentro da programação modular terá de considerar as características acima enunciadas.

3.2 - Módulos.

Na literatura da área da Ciência da Computação são encontradas várias definições de "módulo". Alguns autores divergem no que venha a ser exatamente um módulo, de tal forma que determinadas construções são consideradas, por uns, como sendo módulos, enquanto outros não as reconhecem desta forma.

A seguir, apresentaremos as definições de alguns autores, juntamente com comentários e comparações entre elas. Finalmente, é proposta uma nova definição de módulo.

Myers [Myers 78] define módulo como sendo:

"qualquer coleção de comandos executáveis que satisfazem os seguintes critérios:

1) é uma sub-rotina fechada;

2) pode ser chamada por qualquer outro módulo do programa;

3) tem a potencialidade de ser compilado separadamente."

Myers, nesta definição, não se preocupou em definir um módulo como sendo uma unidade inserida no concerto da programação modular. Esta metodologia professa, segundo o próprio Myers, que os módulos que compõem os programas modulares devem possuir determinadas características, a fim

de se poder atingir os objetivos da programação modular, tais como diminuição da complexidade e do custo do programa, melhorar a legibilidade, a manutebilidade, a extensibilidade, etc. Myers define módulos de forma ampla, para a seguir caracterizar o que é um módulo _bom_ e um módulo _ruim_.

Rigorosamente, a definição por ele proposta não é uma definição de módulo mas sim a de _implementação de módulo_, isto é, de como o conceito de módulo pode ser concretizado em um programa.

Segundo Constantine [Yourdon, Constantine 79],

> "um módulo é uma sequência lexicamente contígua de comandos contidos entre delimitadores com um identicador associado".

Novamente, esta é uma definição de como pode ser implementado um módulo. Os autores não se preocuparam, também, com as características funcionais que o objeto sendo definido deve possuir.

Na verdade, tanto Constantine como Myers, pretendem dizer como obter um objeto denominado módulo, através dos mecanismos e estruturas existentes nas linguagens de programação. Isto é, que recursos de linguagens devemos usar para obtermos um módulo.

Ademais, as definições apresentadas são conflitantes. Segundo Myers, um parágrafo COBOL, uma procedure interna PL/I

ou uma procedure PASCAL não podem ser consideradas como sendo exemplares de módulos, pois violam o terceiro critério da definição. Constantine, por outro lado, aceita as construções acima mencionadas como sendo módulos, pois sua definição não exige que os módulos possam ser compilados separadamente.

Afinal, o que é, então, um módulo? Um módulo é um conceito que pode ser usado no projeto de programas. Como implementar este conceito, não é relevante, desde que a implementação preserve as suas características. Segundo esta linha de pensamento, podemos escrever programas modulares até para uma máquina Turing-completa.

Fixar restrições de como implementar um módulo é equivalente a dizer que um programa que usa GO TOs não pode ser um programa estruturado, segundo a metodologia de programação estruturada. Sabe-se que esta afirmação não é necessariamente verdadeira. Se o programa foi projetado estruturadamente, e se estas estruturas foram implementadas usando-se construções do tipo IF-THEN-ELSE e GO TO, o programa não deixará de ser estruturado.

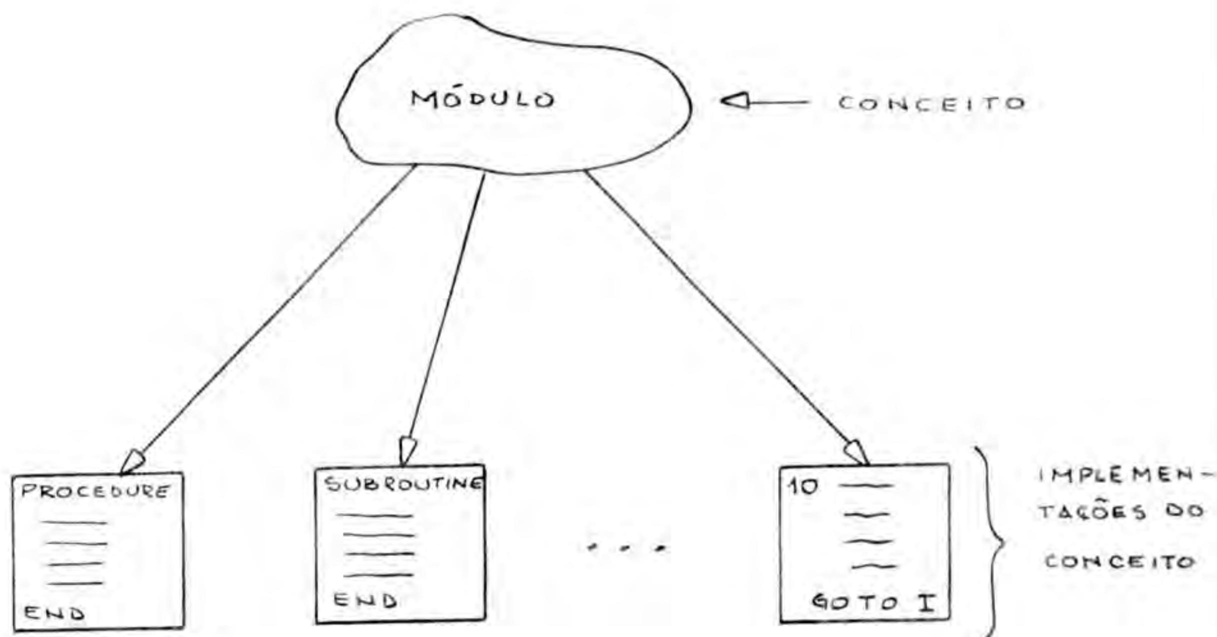
Isto posto, passamos a sugerir a definição abaixo para

U F R G S
BIBLIOTECA
CPD/PGCC

módulo:

> "módulo é uma unidade funcional de programa, projetada de acordo com determinados critérios e destinada a reunir-se ou ajustar-se a outras unidades análogas, de várias maneiras, para a formação de um todo harmônico e funcional".

Observe que esta definição é a do conceito de módulo. Nada é dito a respeito de como realizar este conceito. A figura abaixo ilustra a idéia apresentada.



Segundo a definição proposta, qualquer segmento de programa pode ser a implementação de um módulo, desde que ele represente o conceito do módulo projetado.

Um módulo é composto por um conjunto de funções (operações) e um conjunto de dados (memória) que podem ser,

cada um deles, mas não ambos, vazios. Dependendo da cardinalidade dos dois conjuntos e da interação existente entre eles, pode-se caracterizar diferentes categorias de conceitos implementados por módulos:

a) uma única função, sem memória. Ex.: sqrt (x), sin (x), etc.

b) uma única função, com memória. Ex.: random (x).

c) várias funções, sem memória. Ex.: conjunto de funções para manipulação de matrizes.

d) várias funções, com memória. Ex.: tipos abstratos de dados.

e) somente dados. Ex.: records Pascal, blockdatas Fortran.

Tendo em vista que estamos caracterizando módulos tendo em mente o tratamento de exceções em ambientes modulares, não incluiremos a última categoria como sendo uma geradora de módulos, pois um módulo composto somente de dados não gera, por si só, exceções. Para que um módulo desse tipo provoque exceções, é necessário que haja um módulo "driver" que, juntamente com o módulo de dados, forma um terceiro que se enquadra em uma das demais categorias.

Assim, somente consideraremos, para fins do estudo do tratamento de exceções, módulos "ativos", isto é, módulos que efetuam transformações. Por esta razão, não consideraremos os

módulos de dados, que são "passivos", isto é, que apenas sofrem transformações.

3.3 - Requisitos da Programação Modular.

Vimos que para se atingir os objetivos da programação modular é necessário que os módulos produzidos possuam determinadas propriedades, já anteriormente enumeradas: construtibilidade, encapsulamento da implementação, baixo acoplamento, alto grau de coesão e, finalmente, verificabilidade.

A propriedade de construtibilidade diz respeito ao contexto no qual o módulo será usado e ao nível de suposições feito na sua construção. Em qualquer dos dois casos, deseja-se que a influência do contexto e o nível de suposições seja o menor possível, ou mesmo inexistente. Isto significa que módulos que possuam a propriedade de construtibilidade possam ser combinados entre sí para formarem novos módulos, sem necessitar do conhecimento de detalhes de construção de cada um. Da mesma forma, a construção de um módulo deve ser feita de tal maneira que não seja necessário supor qualquer coisa sobre o contexto no qual ele será usado.

Existem dois aspectos relacionados com esta propriedade. O primeiro diz respeito ao nível linguístico que implementa o módulo, que deve possuir recursos para construir interfaces genéricas, isto é, que garantam o nível de construtibilidade exigido. O segundo aspecto está relacionado com a correção do módulo. Se pudermos abstrair (ou ignorar) o contexto no qual o módulo será usado, podemos, também, fazer afirmações

absolutas sobre o seu comportamento, isto é, podemos garantir a correção do módulo independentemente do ambiente no qual ele será usado, bem como podemos fazer uso (como lemas), dos critérios de sua correção, quando o utilizarmos no desenvolvimento de outros módulos.

A propriedade de encapsulamento da implementação permite liberar o usuário de um módulo do conhecimento dos detalhes irrelevantes de sua construção, isto é, dos detalhes que o usuário não precisa, não deseja ou não deve conhecer para poder usar o módulo. Esta limitação produz benefícios extremamente desejáveis, tais como: os programas (ou módulos) resultantes da composição de módulos que possuem esta propriedade são mais fáceis de desenvolver, de compreender, de manter e de serem provados corretos. O benefício secundário obtido é evitar que o usuário de um módulo explore a sua implementação, o que pode, eventualmente, conduzí-lo a um processo inadequado de solução do problema, isto é, evitar que a implementação (representação dos dados) "sugira" a solução do problema.

Esta propriedade, no entanto, só pode ser atingida através do nível linguístico que implementa o módulo.

A propriedade de acoplamento diz respeito à quantidade e natureza das informações que são trocadas entre os módulos. Esta propriedade, que a literatura denomina de "coupling" [Myers 78], está relacionada com a "dependência" física entre

dois módulos, no sentido de que, quanto maior for a quantidade de informações trocadas entre os módulos e quanto maior for a complexidade dessas informações, maior o acoplamento (e a dependência) entre eles. Naturalmente, módulos fortemente acoplados são indesejáveis, pois esta característica implica, necessariamente, em diminuição de outras propriedades, tais como construtibilidade e verificabilidade. O grau de acoplamento de dois módulos pode depender de vários fatores. O primeiro se refere à forma de ligação entre eles, isto é, do mecanismo que permite a transferência de controle de um módulo a outro e vice-versa. A situação desejável é aquela na qual existe uma quantidade mínima de pontos de entrada e saida de um módulo, suficientes para distinguir os diferentes usos (funções) do módulo. Além disso, a forma de efetuar a transferência deve ser mais disciplinada do que uma simples transferência de controle (GO TO), embora isto possa tornar a interface mais complexa.

O segundo fator que exerce influência sobre o grau de acoplamento é justamente a <u>complexidade da interface</u>. Esta complexidade representa o esforço, ao nível de usuário, necessário para expressar o acoplamento entre dois módulos. Em outras palavras, a tarefa de desenvolver a interface é, em parte, atribuida ao usuário, o que faz com que o acoplamento entre os módulos se torne mais forte.

O terceiro fator de influência é o que diz respeito ao tipo de informações transmitidas de um módulo à outro, ou seja, a finalidade das informações transmitidas a um módulo. Basicamente, as informações passadas de um módulo a outro podem ser usadas com dois fins distintos: como dado (variável), que pode ser operado e como controle que atua sobre a lógica do módulo. Tendo em vista que as informações de controle requerem o conhecimento de pelo menos parte da estrutura interna do módulo chamado, deve-se evitar, na medida do possível, a ocorrência deste tipo de informação.

Finalmente, o último fator que influencia o grau de acoplamento é o tempo em que é feita a ligação dos módulos (binding time). Quanto mais cedo é feita a ligação, menos flexível é o sistema pois o acoplamento entre os mó dulos não pode sofrer mudanças daí em diante. Por outro lado, quanto mais tarde (mais próximo do tempo de execução) for feita a ligação, maior a flexibilidade do sistema, pois os módulos envolvidos no processo de ligação podem ser mudados sem necessidade de se retornar ao instante da ligação.

A propriedade de coesão ou funcionalidade de um módulo é, basicamente, uma propriedade adquirida na fase de projeto de um módulo. Esta propriedade reflete o relacionamento, sob o ponto de vista funcional, dos elementos que compõem o módulo. Esta medida possue grande correlação com a medida de

acoplamento entre módulos, no sentido de que, quanto menor for o acoplamento, maior será a funcionalidade.

A literatura [Myers 78], [Yourdon, Constantine 79] tem proposto uma classificação para os diferentes níveis de coesão de módulos, conforme o tipo de associação interna dos seus elementos. Esta classificação distingue níveis de coesão que vão desde a coincidente, na qual a função não pode ser descrita em termos de um dado conceito (veja definição de módulo, neste capítulo), até a coesão funcional, na qual a função do módulo representa um único e bem definido conceito (operação ou abstração de dado) que possa ser útil para a composição de outros módulos ou programas.

Dizemos, então, que um módulo é funcional (ou possue alto grau de funcionalidade), se o seu nível de coesão for funcional.

Finalmente, a propriedade de <u>verificabilidade</u> se refere à capacidade de um módulo ser verificável, isto é, que seja possível desenvolver uma prova da correção do módulo com respeito a uma especificação ou conjunto de asserções, sem que seja necessário envolver, no processo de verificação, considerações sobre contexto no qual o módulo será usado (por esta razão os aspectos de construtibilidade, acoplamento e coesão são importantes). Se este objetivo (verificabilidade) for alcançado, é possível simplificar o processo de verificação de programas modulares a níveis aceitáveis, uma

vez que a tentativa de provar a correção de um programa monolítico como um todo pode se tornar tão complexa que o processo de verificação se torne proibitivo. Se for possível provar, separadamente, a correção de cada módulo que compõe um programa, então o seu processo de verificação pode ser feito examinando-se a sua estrutura e identificando-se:

1) as propriedades que o programa deve garantir;

2) as propriedades dos módulos usados.

A prova de correção considerará (1) como um conjunto de teoremas a ser provado e (2) como um conjunto de lemas que podem ser usados na prova dos teoremas.

Sabemos, no entanto, que este conjunto de propriedades depende fundamentalmente de dois fatores:

a) da habilidade do projetista em identificar conceitos que podem ser expressos sob a forma de módulos;

b) das características do nível linguístico no qual os conceitos são implementados.

Ou seja, as propriedades dependem do <u>projeto</u> e da <u>linguagem</u> usada para implementar os módulos.

O projeto exerce influência em praticamente todas as propriedades, principalmente no grau de coesão interna do módulo, pois este ítem depende quase que exclusivamente da

habilidade do projetista em escolher os módulos. Sabemos que uma escolha mal feita pode prejudicar de forma significativa a complexidade estrutural do programa resultante.

Quanto à linguagem, esta deve fornecer os meios que permitam o exercício das propriedades enumeradas. Devemos lembrar, no entanto, que o fator que imprime a característica de módulo à uma unidade de programa não é a linguagem usada, mas sim o fato da unidade captar e representar um dado conceito. O nível linguístico apenas acrescenta propriedades desejáveis aos módulos.

As propriedades que também dependem do nível linguístico são:

a) construtibilidade;

b) encapsulamento;

c) acoplamento;

d) verificabilidade.

Devemos acrescentar, ainda, uma característica à linguagem, para que ela permita (ou forneça recursos para permitir) a composição de módulos, a fim de que estes possam, conforme a definição proposta, "reunirem-se ou ajustarem-se um aos outros, formando um todo harmônico e funcional". A esta característica denominaremos de <u>componibilidade</u>.

A seguir, mostraremos como os recursos de linguagem podem contribuir na consecução dos objetivos da programação modular. Usaremos a linguagem ADA [Ichbiah 79] para ilustrar alguns casos. Cabe, antes, resaltar o fato de que os recursos de linguagem não garantem que os módulos tenham as propriedades que dela dependem. Estes recursos apenas permitem que o programador desenvolva módulos com aquelas propriedades.

3.4 - Modularidade em ADA.

O mecanismo da linguagem ADA que permite a implementação de módulos é o *package*. Este mecanismo permite a implementação de tres tipos de módulos:

1) módulos de dados;

2) grupo de funções (subprogramas) relacionados;

3) tipos de dados.

Tendo em vista o interesse particular no último tipo de módulo, nos deteremos apenas na descrição das facilidades existentes na linguagem para implementá-lo.

A implementação de um tipo de dado por meio de um *package* é feita em duas partes: a primeira, denominada *package specification*, contém a especificação do módulo, enquanto a segunda parte, denominada *package body* contém a implementação propriamente dita do módulo. A parte de especificação do módulo (visível ao usuário) contém a declaração dos dados que devem ser conhecidos do usuário e, também, a declaração das funções disponíveis ao usuário. A parte de implementação (invisível ao usuário) contém os dados locais do módulo e o corpo das funções especificadas na primeira parte, além de eventuais funções de apoio.

A linguagem permite que um programa (ou módulo) que use um dado módulo M seja compilado apenas em presença da parte

de especificação deste módulo. Enquanto a especificação de M se mantiver inalterada, o programa (ou módulo) que o usa não precisa ser recompilado, podendo, emquanto isso, haver várias redefinições da implementação do módulo. Este recurso é uma ferramenta importante tanto na fase de desenvolvimento como na manutenção do programa. O fato de se poder substituir a implementação de um módulo sem afetar o programa que o usa é que imprime a característica de construtibilidade ao módulo, ou seja, a única interface do módulo é a sua especificação.

Entretanto, para permitir este recurso, foi necessário que a definição dos dados de um módulo fosse feita na parte de especificação deste módulo, a fim de que o compilador dispusesse de informações suficientes para manipular as declarações de variáveis de um dado tipo e pudesse, também, decidir sobre alocação de memória para estas variáveis.

Como, no entanto, o usuário tem acesso amplo e irrestrito a todos os elementos constantes na especificação de um módulo, foi criado o conceito de private, que torna inacessíveis (e invisíveis) ao usuário os dados declarados na especificação, de tal maneira que eles só possam ser operados pelas funções definidas pelo módulo.

Quanto a acoplamento, a linguagem oferece recursos para que toda a troca de informações entre um módulo e o contexto no qual ele está situado seja feita por meio de parâmetros. Um recurso importante que permite diminuir o acoplamento é a

existência de variávaies "own" na implementação de um módulo, as quais retêm seu valor entre duas chamadas consecutivas do módulo. Em linguagens que não possuem este recurso, pode-se obter este efeito através de variáveis globais ou por meio de parâmetros. Em qualquer caso elas serão conhecidas fora do módulo, mesmo que isto não seja necessário sob o ponto de vista lógico, aumentando, assim, a troca de informações entre o módulo e o contexto que o contém.

Um outro recurso importante existente na linguagem, embora não seja um conceito novo, permite restringir o tipo de acesso aos parâmetros ao mínimo necessário. Para cada parâmetro transmitido de um módulo a outro deve ser declarado o tipo de acesso permitido ("read", "write" ou "read/write"). Um parâmetro que deva ser usado apenas como dado de entrada em um módulo é declarado como sendo do tipo <u>in</u>. Os parâmetros só de saida são declarados do tipo <u>out</u>, enquanto que aqueles que são usados indistintamente tanto como entrada como de saida devem ser declarados como <u>in</u> <u>out</u>.

Mostramos, a seguir, um exemplo extraido de [Wegner 79], que mostra o uso dos recursos da linguagem ADA para construir um módulo de definição do tipo abstrato "RATIONAL NUMBER".

```
package RATIONAL_NUMBER is
  type RATIONAL is private
  type POSINTGR is INTEGER range 1..INTEGER'LAST;
  function "=" (X, Y: RATIONAL) return BOOLEAN;
  function "+" (X, Y: RATIONAL) return RATIONAL;
  function "*" (X, Y: RATIONAL) return RATIONAL;
  function "/" (X   : INTEGER,
                Y   : POSINTGR) return RATIONAL;

  private
    type RATIONAL is
      record
        NUMERATOR   : INTEGER;
        DENOMINATOR : POSINTGR;
      end record
end

package body RATIONAL_NUMBER is
  procedure SAME_DENOMINATOR (X, Y: in out RATIONAL) is
    begin
      -- reduz X e Y ao mesmo denominador.
    end
  function "=" (X, Y: RATIONAL) return BOOLEAN is
    U, V: RATIONAL;
    begin
      U := X;
      V := Y;
      SAME_DENOMINATOR (U, V);
      return (U.NUMERATOR = V.NUMERATOR);
    end "=";

  function "+" (X, Y: RATIONAL) return RATIONAL is
    ...
    end "+";

  function "*" (X, Y: RATIONAL) return RATIONAL is
    ...
    end "*";

  function "/" (X   : INTEGER,
                Y   : POSINTGR) return RATIONAL is
    begin
      return (X, Y);
    end "/";
end RATIONAL_NUMBER;
```

O trecho de programa abaixo ilustra a criação de objetos do tipo RATIONAL e o uso destes objetos por meio das operações de igualdade, adição, multiplicação e atribuição.

```
declare
  use RATIONAL_NUMBER;
  X, Y, Z: RATIONAL := 1/1;   -- set initial value.

begin
  X := 3/4;
  Y := 6/8;
  if X = Y
    then Z := X * X;
    else Z := X + Y;
  end if;
end;
```

Após a execução do trecho de programa acima, Z terá o valor 9/16.

3.5 - Tratamento de Exceções em Ambientes Modulares.

Vimos anteriormente o conceito de tratamento de exceções. Verificamos que "tratar uma exceção" é executar um tratador de exceções que esteja associado à exceção (para maiores detalhes reporte-se ao capítulo 6, onde estes conceitos são formalizados). Sob o ponto de vista lógico, o tratamento de uma exceção corresponde à providência tomada em resposta a ocorrência de uma situação que foi definida como sendo de exceção.

Veremos, nesta seção, o significado da expressão "tratamento de exceções em ambientes modulares", que intitula este trabalho.

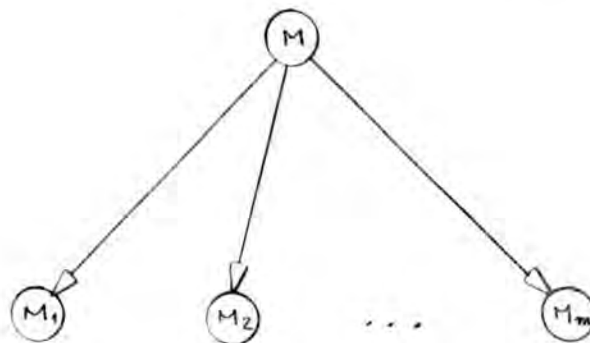
Caracterizamos, inicialmente, um ambiente modular como sendo o ambiente formado por um programa desenvolvido segundo a técnica programação modular. Sabemos que a execução de um programa modular produz uma hierarquia de ativações dos módulos, que reflete a composição usada para desenvolver o programa. Existem, pois, dois elementos que caracterizam um ambiente modular, segundo o sentido dado neste trabalho:

a) um conjunto de módulos Mi, cada um possuindo as propriedades antes enumeradas;

b) uma relação entre os módulos.

A estruturação dos módulos na forma de hierarquia dá origem ao conceito de <u>contexto</u>, que é fundamental no tratamento de exceções em ambientes modulares.

Seja M um módulo cuja execução gera uma hierarquia de ativações de módulos Mi, i=1,...,m, onde Mi são módulos que compõem M.

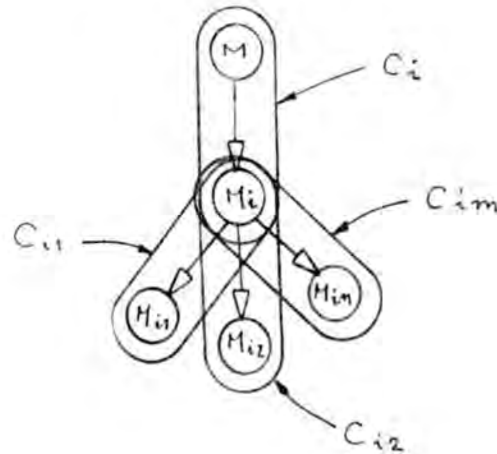


Cada um dos Mi, por sua vez, pode ser decomposto em módulos Mik_i, até o ponto em que eventualmente alguns desses módulos não sejam mais decomponíveis, aos quais denominamos de módulos primitivos.

Denominamos, então, de contexto de Mj o conjunto formado pelo módulo Mj juntamente com o módulo M que o ativou. Este contexto será usado, explorando não só o fato de que ele é composto pelos dois módulos, mas também, e principalmente, pelo fato de que o módulo Mj foi ativado pelo módulo M.

A figura abaixo ilustra a formação dos contextos aos

quais pertence um módulo Mi.



Para caracterizar o tratamento de exceções em ambientes modulares, vamos distinguir entre casos de exceções endógenas e exógenas.

No caso de exceções endógenas, é o tratamento conduzido dentro no contexto no qual a exceção foi causada e que procura preservar, ao mesmo tempo, as propriedades dos módulos que formam aquele contexto. Além disso, o tratamento procura explorar as características do contexto, isto é, procura se valer do fato de que o módulo Mi causou uma dada exceção quando ativado pelo módulo M em um ponto bem conhecido.

No caso de exceções exógenas, o tratamento não se restringe , necessariamente, ao contexto que gerou a exceção, sendo que os módulos sinalizadores e receptores são necessariamente diferentes e estão em execução simultânea (ou concorrente). Neste caso o tratamento da exceção deverá

apenas preservar, como no caso anterior, as propriedades do módulo receptor.

No caso de exceções endógenas, a decisão de restringir o tratamento da exceção ao contexto no qual ela ocorreu se deve ao fato de que este requisito é fundamental para manter as propriedades de encapsulamento, verificabilidade, acoplamento e coesão. Caso fosse permitido o tratamento fora do contexto da exceção, seria necessário que o módulo receptor conhecesse os detalhes do sinalizador. Entretanto, como neste caso o sinalizador não seria um componente direto do receptor, seria necessário que este conhecesse detalhes da implementação dos seus módulos componentes.

Para exemplificar, vamos supor o caso de um módulo A, composto pelos módulos B e C, sendo que B é composto pelo módulo D, dentre outros (vide figura adiante). Naturalmente, o módulo A nada sabe sobre os componentes de B e C, sendo que cada um desses, por sua vez, desconhece a composição um do outro. Isto significa que, a fim de que as propriedades dos módulos envolvidos neste exemplo sejam conservados, é necessário que as eventuais exceções ocorridas durante a execução de D sejam tratadas pelos módulos que estão no contexto de D, ou seja, D ou B.

Caso o tratamento das exceções de D seja feito no módulo C ou no módulo A, isto descaracterizaria o tratamento de exceções em ambientes modulares. Neste caso estaríamos diante

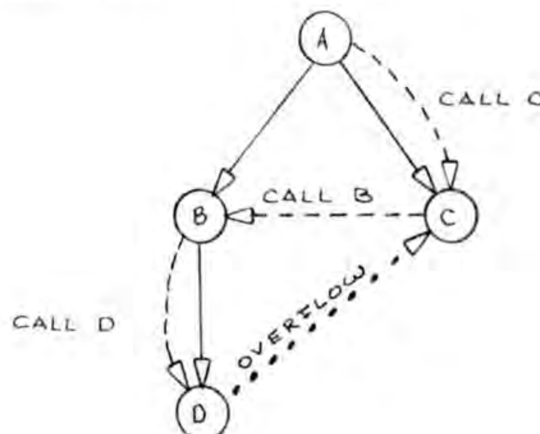
U F R G S
BIBLIOTECA
CPD/PGCC

daquilo que podemos chamar de <u>tratamento de exceções em ambientes não-modulares</u>.

O exemplo a seguir mostra um possível uso do mecanismo ON Condition da linguagem PL/I [Ibm] que proporciona um tratamento de exceções não-modular. Neste exemplo, a exceção <u>overflow</u> provocada na procedure D será tratada na procedure C. Para que este tratamento possa ser conduzido de forma adequada é necessário que C conheça detalhes da implementação de B.

```
A: PROC;
   B: PROC;
      D: PROC (X, Y, Z);
         X = Y + Z;
         END D;
      CALL D (X, Y, Z);
      END B;
   C: PROC;
      ON OVERFLOW BEGIN; ... END;
      CALL B;
      END C;
   CALL C;
   END A;
```

Esquematicamente, a situação caracterizada por este exemplo é a seguinte:



Neste caso, como que C pode tratar a exceção havida, se ele desconhece não só a existência de Y e Z como desconhece, também, o significado da operação que causou a exceção?

Capítulo 4 - Requisitos.

Neste capítulo são discutidos basicamente dois aspectos relativos ao tratamento de exceções em ambientes modulares. O primeiro é uma discussão sobre as diferentes formas de conduzir o tratamento das exceções, no que diz respeito às estratégias de solução. O segundo é a caracterização dos requisitos que um mecanismo de tratamemto deve assegurar.

4.1 - Estratégias de Tratamento.

Um dos requisitos que qualquer mecanismo de uma linguagem de programação deve possuir é o de adequação às variadas classes de problemas que ele propõe resolver, isto é, que o mecanismo possa ser usado para resolver problemas reais, de forma natural. Um requisito, como acima colocado, se torna extremamente difícil de se mostrar que é atingido, a menos que se conheça o universo dos problemas que o mecanismo se propõe s resolver, bem como as respectivas soluções destes problemas.

Ao invés de seguirmos esta linha, proporemos, no lugar de formas de tratamento de exceções, estratégias de tratamento, que acreditamos que, se bem usadas, possam conduzir à solução de problemas de tratamento de exceções em ambientes modulares.

4.1.1 - Local de Tratamento.

Existem duas alternativas de tratamento de exceções, no que diz respeito ao local de tratamento. A primeira é conduzir o tratamento no próprio ambiente onde ocorreu a exceção. Esta alternativa é viável quando se pode recolher, do referido ambiente, os elementos necessários para a condução do tratamento. Em alguns casos este tipo de tratamento, que denominamos de interno não só é possível, como obrigatório. Isto ocorre quando a exceção está

relacionada com aspectos de representação de dados, os quais são ignorados no exterior do módulo. Suponha, por exemplo, o caso de um módulo que utilize para representação física de um dado, uma lista linear duplamente encadeada. Suponha, ainda, que o módulo faça verificação de consistência interna dos dados, para assegurar a integridade física da estrutura. No caso de ser detectada, por exemplo, uma perda de um dos "links", pode ser gerada uma exceção que, obviamente, deve ser tratada internamente a este módulo, pois somente no seu interior é que a estrutura física do dado é conhecida.

A segunda alternativa, ainda no que se refere ao local de tratamento, é conduzí-lo no exterior do módulo que gerou a exceção. Esta alternativa deve ser usada nos casos em que o significado lógico da exceção só pode ser determinado fora do ambiente que a causou. É o caso, por exemplo, da exceção "invalid index" que indica que houve tentativa de acessar um elemento não existente em uma estrutura indexada. Obviamente, neste caso, o módulo que detectou a exceção não tem autonomia para tratar a exceção, pois não conhece o seu significado, dentro do contexto do programa. A alternativa de tratar a exceção fora do módulo que a gerou, a qual denominamos de tratamento <u>externo</u>, deve se restringir ao ambiente do módulo que ativou o sinalizador. não sendo permitida a propagação da exceção a níveis superiores. O motivo desta restrição reside no fato de que, devido a característica de encapsulamento da implementaçao, os módulos dos níveis superiores desconhecem a

existência dos descendentes do módulo ativador, ignorando, portanto, as exceções que que eles eventualmente venham a sinalizar.

Existem, portanto, dois locais de tratamento de uma exceção: interno e externo, sendo que o externo deve se restringir ao módulo que ativou o sinalizador. Em outras palavras, o tratamento de uma exceção deve ser feito dentro do "contexto" (conforme definido na seção 3.5) do módulo sinalizador.

4.1.2 - Associação Exceção-Tratador.

Um dos aspectos importantes no tratamento de exceções é o que diz respeito à associação entre exceções e tratadores, ou seja, com qual unidade sintática do módulo deve ser associado um tratador.

Este aspecto é importante, uma vez que, tendo sido feita a associação, sempre que ocorrer uma exceção na execução da unidade sintática, o respectivo tratador será executado.

Existem as seguintes unidades sintáticas (do módulo receptor) candidatas à associação com tratadores:

1) o módulo todo;

2) blocos;

3) comandos;

4) expressões;

5) operações.

A relação de candidatos acima está ordenada no sentido de representar, a primeira, uma associação "fraca", e a última uma associação "forte". Dizemos que uma associação é forte quando ele permite identificar, de forma clara, o significado lógico da exceção. Caso contrário dizemos que a associação é fraca.

Por exemplo, se associarmos a ocorrência das exceções do módulo que implementa a operação de multiplicação (*) ao comando abaixo:

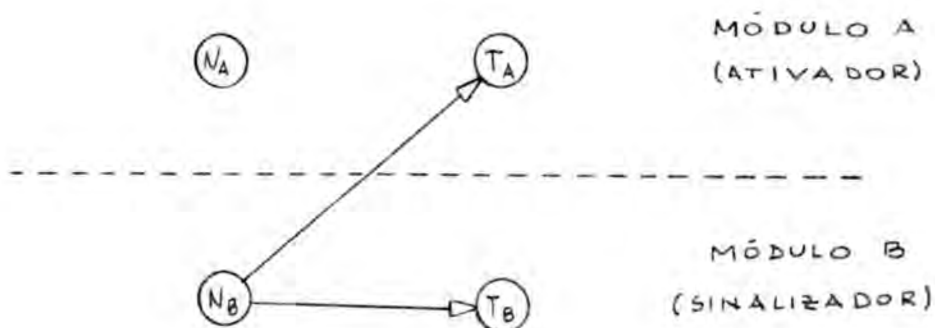
prod := a * b * c

não poderemos distinguir em qual ativação da operação * ocorreu a exceção, o que indica que o tratamento teria de ser mais genérico.

Por este motivo, é desejável que o tipo de associação entre exceções e tratadores seja o mais forte possível, ou seja, o caso em que se associa a ocorrência de exceções com ativações de operações.

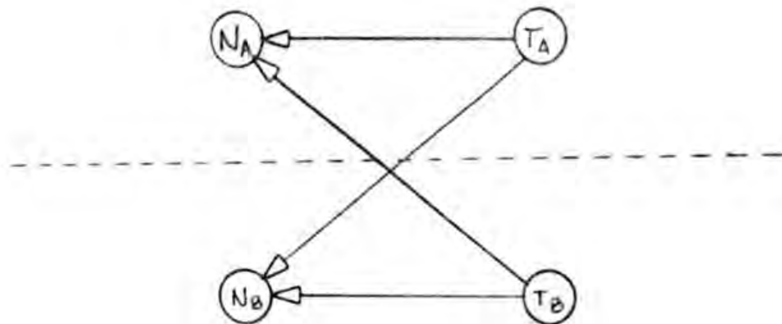
4.1.3 - Retomada do Processamento.

A retomada do processamento corresponde à fase seguinte ao tratamento de uma exceção. Podemos caracterizar o problema da seguinte forma: dizemos que um sistema está em estado de tratamento, se ele estiver executando um tratador de exceções. Caso contrário dizemos que o sistema está em estado normal. A figura abaixo caracteriza as diferentes transições que podem ocorrer dos estados normais (N) para os estados de tratamento (T) de dois módulos interligados.



Ainda se referindo a figura acima, suponha que o módulo B esteja no estado N_B. A ocorrência de uma exceção neste módulo acarreta uma das seguintes mudanças de estado: ou o novo estado é T_B(no caso de tratamento interno), ou é T_A(no caso de externo). O problema central em discussão é: qual (ou quais) a alternativa de mudança de estado, quando for concluido o tratamento da exceção?

A rigor, qualquer das transições mostradas na figura abaixo é possível, dependendo da exceção ocorrida e do tratamento dado.



A existência das transições T_A -> N_B e T_A -> N_A podem ser caracterizadas através do seguinte exemplo: suponha um módulo que implementa um método iterativo de solução de um problema da Análise Numérica. A cada iteração efetuada o módulo gera uma exceção que provoca a transição N_B -> T_A . O tratador da exceção examina o valor do erro cometido até então pelo processo iterativo e decide ou retomar a execução de B (T_A -> N_B) ou dar por encerrado o processo iterativo (T_A -> N_A).

No caso das transições T_B -> N_B e T_B -> N_A pode-se pensar em situações análogas.

As transições do tipo T_B -> N_B e T_A -> N_B definem o modelo de retomada denominado <u>modelo de continuação</u> (resumption model), enquanto que as demais (T_B -> N_A e T_A -> N_A) definem o <u>modelo de término</u> (termination model).

4.2 - Requisitos de um Mecanismo.

Nesta seção são formulados e justificados os requisitos que um mecanismo de tratamento de exceções em ambientes modulares deve possuir. Embora alguns requisitos sejam mais importantes do que outros, não apresentaremos nenhuma ordem de prioridade entre eles.

Os requisitos de um mecanismo para tratamento de exceções em ambientes modulares podem ser considerados em diferentes níveis:

a) requisitos específicos;

b) requisitos genéricos;

c) requisito de verificação.

Os requisitos específicos são aqueles que se referem aos objetivos do mecanismo. Uma vez que o mecanismo é especificamente projetado para o tratamento de exceções, espera-se que ele possa ser usado em uma variedade de problemas de tratamento de exceções. Este requisito aparece, então, como sendo a capacidade do mecanismo de permitir a aplicação das diferentes estratégias de tratamento de exceções, conforme foi colocado na seção 4.1 deste capítulo. Assim, os requisitos específicos do mecanismo são:

1) dar ao usuário a opção de escolha do local de tratamento da exceção;

2) permitir que exceções sejam associadas à operações (ativações de módulos);

3) dar ao usuário a opção de escolha da forma de retomada do processamento após o tratamento de uma exceção.

Os requisitos genéricos são, por sua vez, aqueles que se referem ao ambiente no qual ele será usado. Como o mecanismo é proposto para o uso em programação modular, espera-se que o seu uso permita preservar as propriedades dos módulos sinalizadores e receptores das exceções, conforme foi colocado no capítulo 3, principalmente aquelas propriedades que dependem do nível linguístico:

1) construtibilidade;

2) encapsulamento;

3) acoplamento.

A propriedade de acoplamento, como vimos anteriormente, diz respeito a quantidade e natureza das informações que são trocadas entre os módulos. Segundo o que foi discutido no capítulo 3 sobre este assunto, é desejável que o acoplamento entre módulos seja mínimo. Isto implica em minimizar a quantidade de informações trocadas entre os módulos e restringir a natureza destas informações a dados somente.

Entretanto, na hipótese de haver tratamento externo de exceções, é necessário aumentar o acoplamento entre os

módulos, tanto no que diz respeito a quantidade como a natureza das informações. A quantidade de informações será maior pois é necessário transmitir dados relativos a ocorrência da exceção. Além disso é necessário transmitir informações de controle que são usadas principalmente para a escolha do tipo de retomada do processamento após o tratamento da exceção (veja detalhes no capítulo 6).

Assim, no que diz respeito ao requisito de acoplamento, o mecanismo deve ser tal que o grau de ligação seja mantido em níveis toleráveis.

Este requisito pode ser atingido, se as trocas de informações específicas para o tratamento de exceções forem feitas de forma disciplinada.

Quanto ao encapsulamento, o mecanismo deve ser tal que não exija que o usuário de um módulo deva conhecer detalhes da implementação deste módulo para escrever os tratadores das exceções que ele provoca.

Finalmente, o último requisito genérico é o de construtibilidade. Este requisito estabelece que o mecanismo permita que seja relativamente simples a substituição de módulos (sinalizadores e receptores) do sistema, sem que seja necessário alterar de forma substancial os demais módulos.

Por fim, o requisito de verificabilidade. Este

requisito, como veremos a seguir, está fortemente relacionado com os requisitos antes formulados.

Este requisito é estabelecido no sentido de permitir que seja desenvolvido um processo de prova de um módulo, sem que se conheçam os tratadores externos das exceções do módulo. É fácil perceber que o processo de verificação de um módulo seria bastante simples, se fossem conhecidos, no momento da sua prova, os tratadores das suas exceções externas. Isto, no entanto, prejudicaria de forma significativa a propriedade de construtibilidade, pois não seria possível construir bibliotecas de "módulos provados".

Assim, é necessário não só dar a característica de verificabilidade ao módulo, como também desenvolver uma metodologia de verificação que assegure os demais requisitos.

Quanto a outros requisitos que eventualmente possam ser colocados, e que normalmente são invocados por outros autores, faremos uma abordagem no capítulo a seguir, onde são discutidas várias propostas recentes de mecanismos de tratamento de exceções.

Capítulo 5 - Trabalhos Recentes.

Nesta seção descrevemos, resumidamente, os trabalhos mais recentes na área em estudo, com dois objetivos. Primeiro caracterizar o estado da arte e segundo permitir uma apreciação crítica dos resultados até agora obtidos.

Embora os artigos a seguir citados não representem a totalidade dos trabalhos recentemente publicados, são aqueles que consideramos de maior relevância.

Não incluiremos neste trabalho a descrição de mecanismos tradicionais de tratamento de exceções em linguagens existentes. Este assunto pode ser encontrado em [von Staa 74] ou [Levin 77]. Também não incluiremos as propostas de [Dijkstra 75] (guarded commands) e [Back 80] (multi-exit statements), por considerá-los inadequados ao tratamento de exceções, embora seus autores afirmem que eles podem ser usados com este objetivo. A razão da sua inadequação reside no fato de que aqueles mecanismos não distinguem entre os casos "normais" e os casos de exceções o que, ao nosso ver, não invalida os mecanismos, mas faz com que eles produzam programas onde esta característica não fica claramente exposta.

Ao final do capítulo serão feitas críticas e comentários sobre os trabalhos aqui apresentados.

5.1 - "Toward Modular Verifiable Exception Handling". [Berry, Kemmerer, von Staa, Yemini 79]

Berry, ao abordar o problema do tratamento de exceções em ambientes fortemente modulares, sugere uma solução denominada Fully Protected Solution, na qual são definidos, dentro do módulo, um elenco de "handlers" para cada condição que possa ser detectada pelo módulo. O usuário, ao fazer uso deste módulo, associa cada condição com um dos "handlers" disponíveis.

Esta estratégia apresenta vantagens no sentido de que se torna mais fácil assegurar o encapsulamento do módulo, bem como provar a sua correção, pois as ações que se desenvolvem são todas interiores a ele. Por outro lado, introduz algumas desvantagens:

a) o projetista do módulo deve antecipar "handlers" suficientes para cada um dos diferentes tratamentos que o usuário queira dar à exceção;

b) a existência de muitas alternativas tende a confundir o usuário sobre qual delas deva utilizar;

c) finalmente, nem todos os tratamentos podem ser feitos no interior do módulo, se o contexto por ele definido não for suficiente para decidir qual o tratamento a ser dado, ou então, se o contexto definido pelo módulo não

for suficiente para desenvolver um dado tratamento que o usuário queira dar.

Numa segunda alternativa, tambem proposta por Berry, denominada Partially Protected Solution, é permitida a extensão de tipos através de parâmetros. Nesta solução, o módulo é aberto para permitir que um novo "handler", definido pelo usuário, possa ser acrescentado. Este novo "handler" poderá ter acesso à representação do tipo de dado.

A possibilidade de permitir ao usuário de um módulo a parametrização proposta, a fim de que o módulo seja moldado às suas necessidades, não é considerada como sendo uma boa solução, uma vez que a parametrização de condições requer a parametrização das operações que detectam as condições. Esta solução tambem provoca algumas desvantagens:

a) no caso de módulo que implementa uma abstração de dado, é necessário fornecer, ao usuário, detalhes da representação do dado, para que este possa codificar as operações. Isto viola as características de encapsulamento do módulo;

b) a especificação de uma operação pelo usuário pode invalidar a verificação da correção da implementação, requerendo, por parte do usuário, nova verificação;

c) novamente, a solução restringe o tratamento da exceção ao ambiente do módulo.

UFRGS
INSTITUTO DE INFORMÁTICA

5.2 - "Exception Handling: Issues and a Proposed Notation". [Goodenough 75c]

O artigo de Goudenough aborda os seguintes tópicos referentes à exceções: definição de condição de exceção (veja seção 2.1 deste trabalho), requisitos que os mecanismos de tratamento de exceções devem satisfazer e, finalmente, propõe um mecanismo que satisfaz os requisitos enunciados.

Quanto ao uso, o autor afirma que, em essência, exceções permitem ao usuário estender o domínio ou o contra-domínio (range) de uma operação, isto é, permite que o usuário generalize uma operação, de acordo com seu interesse. De um modo específico, exceções são usadas:

a) para permitir a correção de falhas de domínio ou contra-domínio ocorridas em operações;

b) para indicar o significado de um resultado;

c) para monitorar o andamento de uma operação.

Falhas de domínio ocorrem quando a operação não produz resultados válidos. Para tratar falhas desta natureza, o mecanismo deve dar ao "invoker" a capacidade de executar uma das seguintes ações:

a) abortar a operação;

b) tentar nova execução;

c) encerrar a operação, retornando resultados parciais.

Falhas de contra-domínio são aquelas que ocorrem quando os dados de entrada de uma operação falham em um teste de aceitabilidade. Nestes casos, o mecanismo deve permitir que o "invoker" receba informações suficientes sobre a falha ocorrida.

Quando exceções são usadas para dar significado a um resultado válido, é suficiente que o mecanismo permita que o "invoker" receba informações adicionais que descrevam o resultado da operação, quer seja através de variáveis de estado ou de códigos de retorno (parâmetros, cujo valor indica o tipo de resultado obtido).

Para o caso de monitoramento de operações, o mecanismo deve prever a possibilidade de notificar o "invoker" quando uma dada condição ocorrer. Este, após cada notificação recebida, deve poder retomar a execução da operação ou, então, cancelá-la. Este recurso pode ser alcançado com um mecanismo de co-rotina.

A ocorrência de uma exceção sempre está ligada à tentativa de executar uma operação. Entretanto, a associação entre tratadores e exceções é feita ao nível de comando.

Quanto a retomada do processamento normal (após o tratamento), Goodenough considera a existência de tres tipos de exceções:

a) ESCAPE - exceções que <u>obrigam</u> o encerramento da execução do sinalizador;

b) NOTIFY - exceções que <u>proibem</u> o encerramento do sinalizador;

c) SIGNAL - exceções que <u>facultam</u> ao usuário a escolha do tipo de retomada.

Todas as exceções devem ser declaradas como sendo de um dos tipos acima.

O tratamento de uma exceção do tipo ESCAPE pode ser encerrado pela execução de um dos seguintes comandos:

a) EXIT - prossegue a execução do comando ao qual está associado o tratador;

b) RETURN - encerra a execução do comando ao qual está associado o tratador;

c) ESCAPE - sinaliza outra exceção do tipo ESCAPE.

O tratamento de uma exceção do tipo NOTIFY deve terminar necessariamente com um comando RESUME, que provoca a continuação da execução do sinalizador.

Goodenough ainda sugere que uma operação possa ser usada apenas como o condutor do sinal de uma exceção sinalizada em níveis inferiores. Neste caso, a operação deve prever a ocorrência das exceções que devem ser re-sinalizadas para o

nível superior (veja discussão sobre este ítem ao final do capítulo 8).

5.3 - "A Program Structure for Error Detection and Recovery". [Horning, Lauer, Melliar-Smith, Randell 79]

O artigo descreve o mecanismo denominado Recovery Block, destinado a permitir a detecção e recuperação de erros de software.

O mecanismo é aplicável a uma linguagem com estrutura de blocos semelhante a do Algol. Assim, cada Recovery Block possue a mesma categoria sintática do bloco do Algol, diferindo, apenas, na sua estrutura interna.

Cada Recovery Block contém um bloco primário, um teste de aceitação e zero ou mais blocos alternativos. Sempre que o Recovery Block é ativado, é executado o seu bloco primário. Após a sua execução, é executado o teste de aceitação. Este teste pode ser visto como uma asserção sobre os efeitos da execução do bloco primário, não sendo necessário, entretanto, que a asserção seja no sentido de comprovar a correção do bloco segundo [Manna 74]. O projeto do sistema fixará o rigor do teste de aceitação.

Se o teste de aceitação tiver sucesso, o Recovery Block é dado por encerrado. Caso contrário é executado o primeiro bloco alternativo, se houver, que tentará executar uma ação corretiva do erro. Ao final da sua execução, o teste de aceitação é repetido. Este processo progride até que uma das ações abaixo ocorra:

a) o teste de aceitação é executado com sucesso;

b) não reste nenhum bloco alternativo a ser executado.

No primeiro caso, a execução do Recovery Block é dada por encerrada com sucesso, enquanto que no segundo caso o Recovery Block reportará fracasso ao bloco que o contém, uma vez que um Recovery Block pode ser um bloco primário ou um bloco alternativo de outro Recovery Block.

Um fator a destacar nesta proposta é que o projeto de qualquer bloco alternativo não é afetado pelo fracasso do bloco primário, ou seja, o bloco alternativo é executado como se o bloco primário (ou qualquer bloco alternativo anteriormente executado) jamais tivesse sido executado.

Quem garante esta propriedade é um mecanismo denominado cache, que restaura o ambiente para o estado encontrado imediatamente antes do início da execução do bloco primário.

5.4 - "Exception Handling in ADA". [Ichbiah 79]

O mecanismo de tratamento de exceções da linguagem GREEN foi projetado para, principalmente, responder à situações imprevistas de erro, detectadas durante a execução de um programa. As situações excepcionais incluem erros de hardware, software, situações de erro em operações "built-in" e aquelas definidas pelo usuário.

Assim, uma exceção pode ocorrer de duas formas:

a) quando for explicitamente provocada pelo programa, através do comando <u>raise</u> ou,

b) quando for implicitamente provocada pelo programa, devido a uma situação de erro prevista pelo sistema (divide by zero, out of range, etc).

Para cada condição de exceção que possa ocorrer no programa, deve haver um tratador correspondente. Este tratador pode estar associado à mesma unidade sintática (procedure, módulo, bloco) que causar ou detectar a exceção ou, então, à outra unidade sintática que a contenha. Tratadores são sempre associados às unidades sintáticas de tal forma que, através dele se possa acessar os elementos (dados, constantes, parâmetros, etc) desta unidade.

Quando ocorre uma exceção, seja através do comando <u>raise</u>

ou seja implicitamente, em uma unidade sintática, duas situações podem ocorrer:

1) se houver um tratador para a exceção ocorrida nesta unidade, ele é executado;

2) caso não haja um tratador, a exceção se propagará para o exterior da unidade da seguinte forma:

   a) se a unidade for um subprograma, a mesma exceção ocorrerá no seu ponto de chamada;

   b) se for um bloco, a mesma ocorrerá no ponto onde este finda.

Em ambos os casos (a) e (b), dizemos que a exceção se propagou.

Em qualquer um dos casos (1) ou (2), sempre que ocorrer uma exceção dentro de uma unidade de programa, esta unidade tem a sua execução encerrada.

A linguagem também prevê mecanismos para tratamento de exceções para o caso de processamento paralelo. Para este caso, o mecanismo sofre algumas limitações, tendo em vista que, pelo fato das relações entre "tasks" serem diferentes das relações existentes entre módulos, subprogramas ou blocos no caso de programação sequencial.

A limitação principal diz respeito à propagação das exceções. Se uma "task" não provê um tratador para uma exceção, não ocorrerá a propagação antes descrita. A "task" será simplesmente encerrada.

5.5 - "Programs Structures for Exception Handling". [Levin 77]

Em sua publicação, Levin propõe um mecanismo de tratamento de exceções de dois tipos: estruturais e de condições de fluxo. Exceções estruturais são aquelas relacionadas com o uso de uma entidade estrutural (dados), enquanto que as de condições de fluxo são relacionadas com a execução de funções. As exceções estruturais são caracterizadas pelo fato de que sua ocorrência é de interesse potencial de todos os usuários da entidade em questão, enquanto que as de fluxo de controle interessam apenas ao usuário que provocou diretamente a exceção.

Uma exceção pode ser associada a comandos, blocos ou chamadas de funções. A cada uma dessas exceções corresponde um tratador. A notação usada pelor autor é:

S [C: H]

para indicar que se ocorrer a exceção C durante a execução de S, será executado o tratador H. Embora a notação usada não deixe claro, o tratador H é considerado como parte integrante de S, tendo, em princípio, acesso aos elementos de S. Em H é permitido qualquer tipo de processamento, exceto efetuar transferências para fora (go to ou exit).

Quando ocorrer uma exceção, podem ser ativados mais de um tratador para processá-la. A escolha dos tratadores a serem chamados é feita em dois estágios. Inicialmente é

determinado um conjunto de tratadores elegíveis. A seguir é selecionado um subconjunto destes tratadores elegíveis, através da aplicação de uma das políticas pré-definidas de seleção. Este conjunto receberá, então, a oportunidade de processar a condição.

Um tratador H é dito elegível para tratar uma exceção E, com respeito a uma ocorrência O, se ele estiver associado a um contexto de programa que possa acessar a ocorrência O.

As políticas de seleção propostas pelo autor são as seguintes:

1) <u>sequential-conditional</u>: chamar, sequencialmente, os tratadores elegíveis enquanto persistir a condição de exceção;

2) <u>broadcast-and-wait</u>: chamar, em paralelo, todos os tratadores elegíveis e esperar que eles encerrem a sua execução, para prosseguir o processamento;

3) <u>broadcast</u>: chamar, em paralelo, todos os tratadores elegíveis e prosseguir, também em paralelo, o processamento.

O mecanismo obriga a retomada da execução do sinalizador em qualquer caso. O autor argumenta que isto é uma vantagem, pois se o tratador, por exemplo, forçar o término do sinalizador, a abstração do módulo pode não ser mantida.

Neste sentido, nem o tratador nem o sinalizador devem ser capazes de influenciar a execução um do outro.

A fim de evitar problemas de interferência de acesso aos dados comuns entre o tratador e o programa a ele associado, o autor adota uma solução simplista: quando um tratador é acionado, se o programa ao qual ele está associado estiver em andamento, a sua execução (do programa) é interrompida (suspensa) até o final da execução do respectivo tratador.

5.6 - "Exception Handling in CLU". [Liskov, Snyder 79]

O trabalho de Liskov e Snyder discute, inicialmente, os vários modelos de tratamento de exceções para, a seguir, apresentar a sintaxe e semântica (informal) do mecanismo de tratamento de exceções da linguagem CLU.

Os autores afirmam que os pontos principais no projeto de um mecanismo de tratamento de exceções são:

1) onde tratar uma exceção e

2) decidir se o sinalizador da exceção deve continuar a existir após a sinalização.

Quanto ao local do tratamento, ele deve ser feito em uma das procedures ativas por ocasião da sinalização. É excluído, deste conjunto de procedures candidatas, a própria procedure sinalizadora, uma vez que os autores consideram exceções como condições que a procedure sinalizadora não é capaz de tratar. Dentre os candidatos restantes, é escolhida a procedure ativadora, uma vez que os demais não possuem conhecimento da existência da exceção.

Quanto ao segundo ponto, os autores optaram pelo "termination model" (veja seção 4.1.3), baseados em um meio-termo entre o poder acrescentado à linguagem pela opção ao "resumptiom model" e complexidade daí resultante.

A sinalização de uma exceção em uma procedure CLU é feita pela execução de um comando signal, o qual encerra a execução da procedure e provoca o seu retorno à procedure ativadora.

Quanto à associação entre exceções e tratadores, são as seguintes as restrições:

1) as associações são estáticas;

2) tratadores só podem ser associados a comandos e não a expressões.

Estas decisões são, segundo os autores, baseadas em aspectos de legibilidade (no primeiro caso) e simplicidade do mecanismo (no segundo caso).

A associação é feita de acordo com a seguinte regra de sintaxe:

<statement> except <handler list> end

Este comando possue a seguinte interpretação: o <statement> sinaliza todas as exceções sinalizadas pelas procedures que ele contém, excluidas aquelas tratadas por comandos except internos. A <handler list> irá tratar algumas (ou todas) dessas exceções. O comando except, como um todo, sinaliza aquelas exceções do <statement> que não foram tratadas pela <handler list>, mais as exceções sinalizadas pelos tratadores da <handler list>.

Cada tratador da <handler list> contém o nome de uma ou mais exceções, seguido de uma lista de comandos que constituem o tratador propriamente dito. Pode haver transmissão de informações da procedure sinalizadora para o tratador através de parâmetros.

5.7 - "Response to Detected Errors in Well-Structured Programs". [Parnas 72b]

O artigo discute um método de manipulação de erros de execução em programas bem estruturados (programas desenvolvidos segundo [Dijkstra 72], [Wirth 73], etc).

Parnas considera que programas sempre contêm erros e, se quizermos que eles sejam confiáveis, devemos projetá-los com a previsão de tratamento de erros.

A proposta baseia-se em sistemas estruturados de forma hierárquica, uma vez que, segundo o autor, a recuperação de erros normalmente requer a ação combinada de níveis hierárquicos diferentes, pois a informação necessária para tratar adequadamente um erro somente é disponível em níveis hierárquicos superiores àquele que detectou o erro.

Basicamente o mecanismo consiste em chamar sub-rotinas definidas externamente ao módulo, especificado segundo [Parnas 72a]. Cada função de um módulo conterá, como parte de sua especificação, a lista de sub-rotinas de tratamento de erros, juntamente com a definição das condições nas quais as sub-rotinas serão chamadas.

Cada usuário poderá, então, prover as suas próprias sub-rotinas para os casos previstos no módulo que ele pretende utilizar. Além disso, Parnas propõe que os códigos do módulo (caso normal), de detecção e de recuperação sejam mantidos lexicamente separados, para facilitar as suas trocas.

5.8 - "Reliable Hardware-Software Architecture". [Wulf 75]

É descrita, neste artigo, a filosofia de tratamento de erros do sistema Hydra, desenvolvido no Computer Science Department, Carnegie-Mellon University. Hydra é o "kernel" do sistema operacional do multi-processador C.mmp,desenvolvido na mesma universidade.

Tendo em vista que o sistema desenvolvido deve ser altamente confiável, no sentido de que as falhas ocorridas possam ser recuperadas (coverage), o sistema dispõe de mecanismos para tratamento de erros.

Os objetivos do sistema Hydra, no que diz respeito a confiabilidade, podem ser resumidos da seguinte forma:

- o sistema deve ser capaz de detectar qualquer falha ocorrida;
- deve ser possível limitar os danos que possam advir de uma falha;
- tornar as falhas transparentes, se possível;
- se não for possível tornar uma falha transparente, então o sistema deve ser capaz de:

  a) colocar o sistema em um estado consistente, de tal forma que a recuperação da falha seja possível;

b) transmitir a ocorrência da falha aos níveis hierárquicos superiores, os quais podem recuperar, de forma inteligente, as falhas transmitidas tornando, assim, transparente a ocorrência da falha para os demais níveis superiores.

Para atingir os objetivos acima, o sistema Hydra é construido de forma modular, onde cada módulo implementa uma abstração. Por outro lado, cada módulo é responsável pela manutenção da integridade da abstração que ele implementa, através da verificação dos objetivos antes mencionados.

Com relação a um módulo, são identificadas tres situações de erro que podem exigir uma ação corretiva:

a) na chamada do módulo (erro de parâmetro);

b) na execução do módulo (consitência de dados);

c) na chamada de módulos do nível inferior, os quais podem estar reportando situações de erro (erro na chamada de funções). Esta situação pode ser devida a ocorrência tanto de (a) como de (b).

Na verdade Wulf não propõe, no artigo, um mecanismo de tratamento de exceções. Ele apenas expõe a técnica utilizada para conduzir o tratamento. Como o sistema foi escrito na linguagem BLISS, o tratamento se resume naquilo que a linguagem oferece para este fim:

a) parâmetros com códigos de erro;

b) chamadas de sub-rotinas (como em [Parnas 72b]);

c) "BLISS Signals".

5.9 - Comentários.

Examinando as propostas acima, podemos concluir os seguintes fatos:

1) existem duas grandes famílias de propostas:

a) uma tende a considerar exceções como sendo uma técnica de programação ([Levin 77], [Berry, Kemmerer, von Staa, Yemini 79], [Goodenough 75]), permitindo que sejam executadas ações de tratamento da exceção e seja, após isso, retomado o processamento normal.

b) Uma segunda família tende a restringir as exceções a eventos que podem ser considerados (em algum sentido) como sendo erros ou, pelo menos, condições de terminação do processamento, isto é, condições que, se ocorrerem, em uma dada unidade do programa, a sua execução é terminada. O controle é passado para um tratador da exceção, porém nunca retornará ao ponto onde a exceção ocorreu. O tratador pode decidir reiniciar a mesma sequência de ações sob melhores condições em um nova chamada, e nunca por meio de retomada de processamento. Entre estas propostas encontramos: [Horning, Lauer, Melliar-Smith, Randell 74], [Ichbiah 79] e [Liskov, Snyder 79].

2) O mecanismo proposto por Levin [Levin 77] permite produzir programas verificáveis. Entretanto, o que dá esta característica ao mecanismo é o fato de ele estar dentro da linguagem Alphard [Wulf 74]. Se quizessemos implementar o seu mecanismo em outra linguagem, o que é possível, seria necessário incorporar, também, algumas características da linguagem Alphard.

Por outro lado, o mecanismo por ele proposto é o único que permite associar exceções a objetos, enquanto os demais, inclusive este que está sendo proposto neste trabalho, só permitem associar exceções à unidades de programas.

Na verdade, o efeito do mecanismo de Levin pode ser obtido pelo aqui proposto, se considerarmos a existência de exceções exógenas.

3) Quanto ao aspecto de propagação da exceção, existem duas categorias de mecanismos:

a) aqueles que restringem os efeitos da exceção ao ambiente onde ela ocorre: [Berry, Kemmerer, von Staa, Yemini 79] e [Horning, Lauer, Melliar-Smith, Randell 74];

b) aqueles que permitem a propagação dos efeitos de uma exceção para fora do ambiente onde ela ocorre. Esta categoria contém dois modelos: "single-level" e

"multi-level". O "single" permite a propagação até um nível hierárquico acima, enquanto o "multi-level" permite a propagação a varios níveis acima. No primeiro caso encontramos [Liskov, Snyder 79] e [Ichbiah 79]. No segundo caso temos [Goodenough 75], [Levin 77] e [Wulf 75].

4) De uma maneira geral, os mecanismos são especializados, no sentido de que eles foram projetados para uma finalidade principal, o que prejudica ou impossibilita o seu uso em outros casos. A seguir damos alguns exemplos de problemas de tratamento de exceções que não podem ser resolvidos pelos mecanismos mencionados:

   a) qualquer problema que exija o encapsulamento da implementação e exija, ao mesmo tempo, tratamento externo para as exceções: [Berry et al. 79];

   b) qualquer exceção que exija retomada do processamento do sinalizador: [Horning et al. 74] e [Ichbiah 79];

   c) monitoramento de processos: [Levin 77] e [Liskov, Snyder 79].

5) Somente algumas propostas permitem o tratamento externo. Dentre auelas que permitem, encontramos dois casos diferentes:

a) as que <u>exigem</u> que o tratamento seja externo [Liskov, Snyder 79], [Levin 77];

b) as que <u>permitem</u> que o tratamento seja externo [Ichbiah 79], [Goodenough 75]. Neste caso, o tratamento será externo <u>somente</u> <u>se</u> não for provido nenhum tratador interno.

Não concordamos com o critério adotado pelas propostas que pré-determinam o local do tratamento da exceção, uma vez que o significado lógico de uma exceção, na maioria dos casos, só é conhecido pelo usuário da abstração e, por conseguinte, é o único que tem os meios de determinar a sua forma de tratamento. Além disso, nestes casos, os recursos necessários para conduzir o tratamento só podem ser obtidos em um ambiente externo. Citamos, aqui, o mesmo exemplo citado por [Levin 77] para caracterizar o problema de interpretação de uma exceção para fins de tratamento: o que significa a ocorrência da exceção "symbol absent" em uma operação de consulta à uma tabela de símbolos durante um processo de compilação de programas? Depende do contexto. Se se tratar de compilação da parte declarativa do programa (declaração de variáveis), trata-se de uma situação normal, que exige um tipo de tratamento. Caso, entretanto, esta exceção ocorrer durante o processo de compilação da parte procedural do

programa, trata-se, então, de erro (variável não declarada), o que exigirá um tratamento diferente.

Este exemplo caracteriza claramente que o mecanismo não pode forçar o usuário a um dado tipo de tratamento, como sugerem algumas propostas.

Quanto à adequação das propostas analizadas, aos requisitos que foram formulados no capítulo 4, mostraremos alguns pontos que cada uma das propostas deixa de satisfazer.

1) Goodenough.

As principais falhas dessa proposta estão nos aspectos de:

a) verificabilidade: o autor não menciona as regras que permitem verificar a correção de programas que usam o mecanismo.

b) o mecanismo permite o tratamento em ambientes modulares, porém gera um alto grau de acoplamento em determinados casos, como no caso em que o autor sugere transmitir variáveis de estado ou parâmetros de retorno, para indicar o significado de um resultado válido de uma operação.

c) a proposta não inclui o tratamento de exceções exógenas.

2) Berry.

A principal falha desse mecanismo, no que diz respeito à adequação aos requisitos formulados, reside no fato de que o usuário deve conhecer a implementação e a representação dos objetos do módulo para escrever os tratadores. Além disto, o mecanismo só permite tratamento interno.

3) Horning (Recovery Blocks).

Como o próprio nome do mecanismo sugere, ele é orientado para programas estruturados sob a forma de blocos, sendo que a ocorrência de exceções é associada a execução de um bloco. Este mecanismo é, na verdade, segundo o próprio autor, destinado a propiciar "fault-tolerance", não sendo indicado para o tratamento genérico de exceções.

Outras falhas do mecanismo: ele somente permite tratamento externo. O teste de aceitação é sempre o mesmo dentro de um Recovery Block, o que nem sempre é a situação desejável.

4) Ichbiah (ADA).

O mecanismo de tratamento de exceções da linguagem ADA cumpre grande parte dos requisitos aqui formulados. Sua principal falha, no entanto, está no fato de que

ele adota o modelo de encerramento da execução do módulo após a ocorrência de uma exceção. Isto o torna não usável para os casos de exceções que exijam a continuação da execução do módulo. O mecanismo permite o tratamento de exceções exógenas, mas apenas entre "tasks" que estejam associadas entre sí de forma direta.

5) Levin.

Sobre o trabalho de Levin, gostaríamos de fazer alguns comentários específicos, uma vez que o seu conteúdo se assemelha, a primeira vista, do conteúdo deste trabalho.

A proposta de Levin é no sentido de prover um mecanismo de tratamento de exceções compatível com a linguagem Alphard [Wulf 74]. Nesse sentido, Levin formulou os objetivos que o seu mecanismo deveria alcançar:

a) verificabilidade;

b) uniformidade;

c) adequação;

d) praticabilidade.

O primeiro objetivo foi alcançado tendo como suporte dois fatos, um dos quais já mencionado: o processo de verificação do mecanismo supõe que este está embutido dentro da linguagem Alphard, se valendo, assim, de toda a metodologia de verificação da linguagem. Levin não desenvolveu nenhum método de verificação de programas com exceções. Ele apenas estendeu o método de verificação da linguagem para incluir o mecanismo proposto. O segundo fato diz respeito ao fato de que, para provar a correção de um programa com tratamento de exceções, é necessário conhecer o comportamento dos tratadores.

O requisito de verificabilidade do mecanismo que propomos é feito com um objetivo. Primeiro, não supõe nenhuma liguagem hospedeira em particular, exigindo, apenas, que ela suporte o desenvolvimento de programas modulares, conforme colocado no capítulo 3. Por esta razão, o método de verificação incorpora os elementos necessários à verificação, sem apelar para a linguagem em si. Em segundo lugar, o mecanismo, por ser orientado para o caso de exceções em ambientes modulares, permite que se processe a verificação, sem a presença dos tratadores das exceções. Naturalmente que, neste caso, o processo de verificação terá de ser completado pelo usuário, quando este desenvolver o tratador. Entretan-

to, esta forma de verificar garante os requisitos de contrutibilidade e encapsulamento.

Com o requisito de uniformidade, Levin deseja que o seu mecanismo seja aplicável a todos os níveis de sistema (incluindo o hardware). O autor mostra, em seu trabalho, que este objetivo é atingido, através de uma série de exemplos que ele considera como sendo os casos "típicos" de vários níveis de sistemas.

Este mesmo requisito aparece no mecanismo que propomos sob o nome de "adequação ao ambiente". Este requisito exige o mecanismo seja aplicável no tratamento das exceções que ocorrem em ambientes modulares. Não fazemos, ao formular o mecanismo, restrições quanto ao seu nível de aplicabilidade. Restringimos seu uso apenas no que diz respeito ao método de programação usado, isto é, a programação modular, independentemente do nível de aplicação.

No terceiro requisito, que Levin denomina de adequação, é estabelecido que o mecanismo deve ter a capacidade de ser usado em uma variedade de problemas reais, de forma natural. Novamente, neste caso, o autor se vale dos mesmos exemplos usados para mostrar a uniformidade do mecanismo, para mostrar, desta vez, a sua adequação. É claro que este requisito é difícil, ou mesmo impossível, de ser "provado". Nada garante que o

elenco de exemplos sugeridos por Levin seja completo. Por exemplo, pelo fato de que o mecanismo exige sempre a retomada do sinalizador após o tratamento de uma exceção, ele não poderá ser usado de forma natural (como o autor propõe) para o caso de tratamentos que exijam o término da execução do sinalizador.

Ao contrário de Levin, neste trabalho é definido um conjunto de estratégias para tratamento de exceções que, mesmo não podendo provar que sejam completas, pensamos que possam ser adotadas na maioria dos casos de tratamento de exceções, principalmente em se tratando de ambientes modulares.

Finalmente, Levin propõe um objetivo final de praticabilidade, no sentido de que o mecanismo somente será de utilidade prática caso tenha uma implementação eficiente.

Não colocamos, em nosso trabalho, este objetivo na lista de requisitos do mecanismo proposto, por duas razões. Primeiro por entender que este requisito é genérico demais, ou seja, este é um requisito de qualquer mecanismo de uma linguagem, além do que, a avaliação da eficiência de um mecanismo é extremamente difícil de ser conduzida. No entanto, no caso de mecanismos de tratamento de exceções este requisito não adquire grande importância, uma vez que uma exceção é

um evento que, por sua natureza, ocorre raras vezes (embora isto não seja regra). O que deve ser exigido, em termos de eficiência, é que o mecanismo não acrescente "overhead" no tempo de execução do programa, quando não ocorrer exceções. Quanto a eficiência do mecanismo em sí, deve-se apenas exigir que ele seja compatível com a complexidade do seu efeito.

O segundo motivo pelo qual não incluimos o objetivo de praticabilidade (segundo Levin), é pelo fato de que, neste trabalho, apenas nos preocupamos com com o aspecto de implementabilidade, o que é mostrado no capítulo 6.

Assim, embora os requisitos das duas propostas se assemelhem em alguns pontos, tratam-se, claramente, de objetivos diferentes. Enquanto Levin propõe um mecanismo para ser usado em programas "Alphard-like" e principalmente para o caso de exceções associadas a objetos (shared data), nós propomos um mecanismo para programas modulares (como definido no capítulo 3) e que, além de assegurar as características de modularidede do programa, faça uso destas características para

conduzir o tratamento das exceções.

6) <u>Liskov</u> & <u>Snyder</u> (CLU).

Sobre o mecanismo de tratamento de exceções da linguagem CLU, verificamos que, embora ele possa ser usado para refletir as exceções ocorridas em módulos (clusters), ele não é adequado ao ambiente modulares, pois a associação entre exceções e e tratadores é feita através de comandos e não por meio de ativações de módulos. Também este mecanismo adota, por questões de simplicidade (segundo os autores) o modelo de terminação, o que o torna não aplicável aos casos em que é exigida a retomada do processamento do sinalizador. Além disso, não existe nenhum recurso no mecanismo o tratamento de exceções exógenas.

7) <u>Parnas</u>.

Embora a proposta de Parnas vise o tratamento de exceções em módulos, vemos dois defeitos principais no seu mecanismo:

a) o fato de que as exceções são tratadas por sub-rotinas (definidas pelo usuário) cujas chamadas já pertencem à especificação do módulo, o que significa que a associação entre exceções e tratadores é feita através de módulos e não de ativações de módulos, ou seja, não importa o local

de chamada de um módulo, suas exceções serão sempre tratadas da mesma maneira, mesmo que o significado lógico das exceções mude de um ponto para outro;

b) uma vez que as chamadas das sub-rotinas de tratamento são embutidas no módulo, o usuário terá de fornecer a ela todas as informações necessárias para o tratamento das exceções por meio de "áreas comuns", o que aumenta, sensivelmente, o acoplamento do sistema.

8) Wulf (Hydra).

Deixaremos de fazer comentários específicos sobre o trabalho de Wulf, uma vez que ele é apresentado mais como uma metodologia e não como uma proposta de mecanismo de tratamento de exceções.

Capítulo 6 - O Mecanismo Proposto

Neste capítulo mostraremos um mecanismo de tratamento de exceções projetado para atender os requisitos formulados anteriormente.

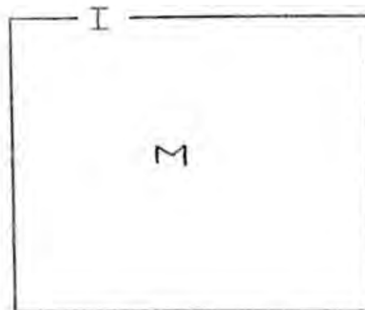
A descrição do mecanismo será feita sobre um modelo formal, a seguir apresentado. Este modelo será usado, também, para justificar determinadas construções e características do mecanismo proposto.

A apresentação do modelo será feita de forma gradual, isto é, partiremos de um modelo inicial simplificado, ao qual serão acrescentados novos elementos, à medida em que isto for necessário.

6.1 - O Modelo

Segundo o modelo que usaremos, um módulo é caracterizado por sua interface de comunicação (I). A interface de um módulo contem todos os elementos que são necessários para permitir a troca de informações com o corpo do módulo.

Esquematicamente, representaremos um módulo M da seguinte forma:



onde M é a implementação de um conceito (função ou abstração de dados).

Um módulo possue uma lista de zero ou mais parâmetros formais $\underline{x}$, aos quais podem ser associados parâmetros reais $\underline{a}$, em cada ativação do módulo.

Sejam P e Q asserções que envolvem os parâmetros $\underline{x}$. A especificação do conceito implementado por M é expresso pela fórmula:

$$P \{M\} Q \qquad (1)$$

cujo significado é o mesmo dado em [Hoare 69].

UFRGS
INSTITUTO DE INFORMÁTICA
BIBLIOTECA

A lista de variáveis livres que aparece em P ou Q está contida (ou é igual) à lista de parâmetros formais. Ou, em outras palavras, P ou Q não incluem variáveis globais. Além disso, as variáveis das listas $\underline{x}$ e $\underline{a}$ devem ser distintas, para evitar a ocorrência de "aliasing".

Seja então

<u>module</u> m(E); M

a declaração de um módulo, onde m é o seu nome, E é a especificação dos seus parâmetros formais e M é o corpo do módulo. Seja x a lista de parâmetros formais de m. A semântica da ativação de um módulo pode ser expressa, então, pela seguinte regra:

$$\frac{P \ \{M\} \ Q}{\forall x \ (P \ \{m(x)\} \ Q)} \qquad (2)$$

onde m(x) indica uma ativação do módulo.

Uma vez que P e Q consideram apenas os aspectos externos do módulo, é necessário incluir na regra de prova a ocorrência de "variáveis de estado", isto é, variáveis do tipo "own" utilizadas por módulos que possuem memória própria, corotinas, tabelas, etc.

Sejam R e S asserções que exprimem propriedades da memória do módulo. Então a semântica da ativação de um módulo com variáveis de estado é expressa pela regra:

$$\frac{P,R\ \{M\}\ Q,S}{\forall x,y\ (P,R\ \{m(x)\}\ Q,S)} \tag{3}$$

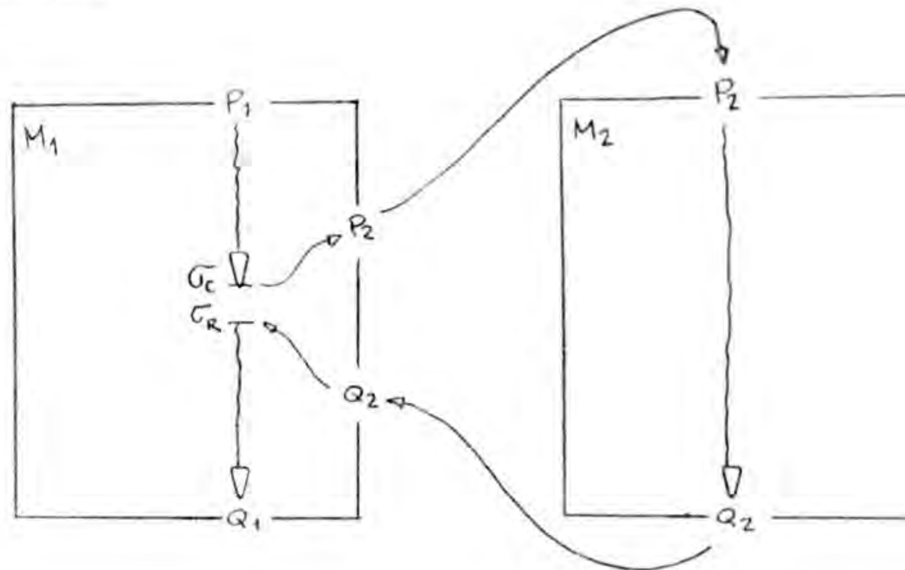
onde y é a lista de variáveis livres em R ou em S (ou em ambas).

Omitiremos, no entanto, nas próximas referências a especificção de um módulo, as asserções R e S com o objetivo de simplificação.

Os conceitos de construtibilidade (building blocks) e encapsulamento (hiding) são caracterizados pelo fato de que, uma vez tendo sido provada a correção do módulo, apenas os elementos que constituem a interface são suficientes para que o usuário possa fazer afirmações absolutas sobre o conceito implementado.

Vamos supor, agora, a existência de dois módulos M1 e M2, sendo M2 ativado por M1. Seja σ_c uma asserção que expressa uma condição de chamada de M2 e σ_R uma asserção que expressa uma condição de retorno, isto é, σ_c e σ_R são asserções sobre M1 que exprimem condições que devem se verificar para que M2 seja chamado e condições que deverão se verificar após a execução de M2, respectivamente. Esquemati-

camente temos:



A fórmula que exprime a chamada de M2 por M1 sob as referidas condições é:

$$\sigma_C \text{ ,P2 \{M2\} Q2, } \sigma_R \qquad (4)$$

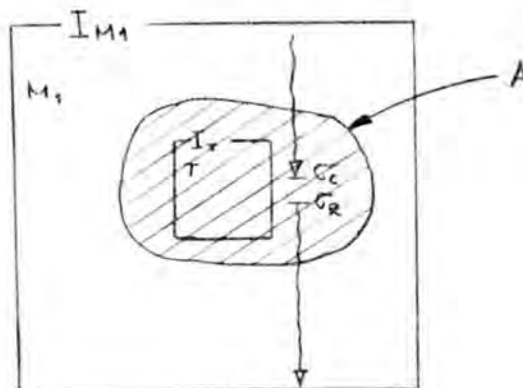
A situação caracterizada até agora é a situação que podemos denominar de "normal". Suponhamos, então, que durante a execução de M2 ocorra uma exceção E. A ocorrência desta exceção implicará na suspensão (temporária) da execução de M2, sendo executado um tratador T contido em M1 ou M2. Suporemos, inicialmente, que T esteja contido em M1, conforme abaixo caracterizado.

Seja o ambiente A, definido no ponto de chamada de M2. Este ambiente é caracterizado pelos seguintes elementos:

1) σ_C , σ_R

2) z : conjunto de objetos (variáveis) visíveis no ponto de chamada de M2.

3) um tratador T para a exceção E.



O tratador T pode ser considerado como um sub-módulo de M1, que possue, da mesma forma que um módulo, parâmetros formais $\underline{w}$ e asserções inicial X e final Y.

Entretanto, ao contrário de um módulo, um tratador tem variáveis globais a ele e ao ambiente A que o contém. Estas variáveis globais são os objetos acima denominados de z.

Da mesma forma que um módulo, a especificação do tratador pode ser expressa pela fórmula:

$$X \{T\} Y \quad (5)$$

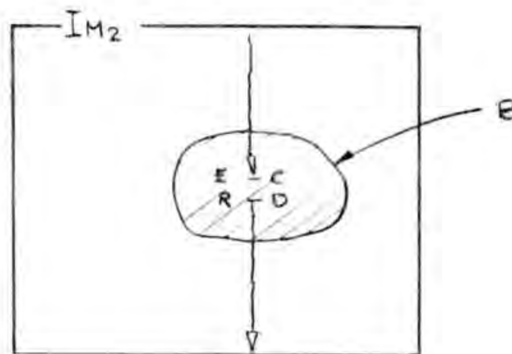
Seja L o "label" de entrada do tratador T. Se, durante a execução de M2 ocorrer a exceção E, associada à condição C,

será executado o tratador T, ou seja, a ocorrência de C implica em uma transferência para o ponto de A correspondente ao "label" L de T. Denotaremos esta situação pela fórmula:

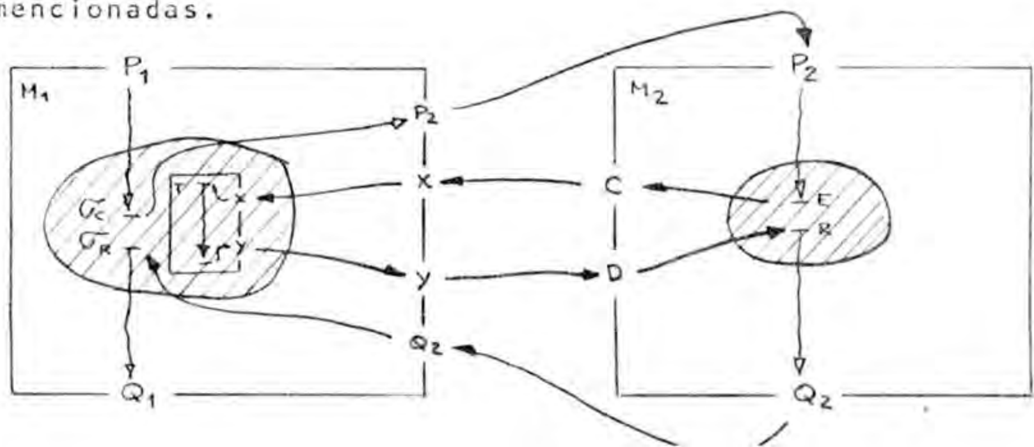
$$C \Rightarrow A(L) \qquad (6)$$

De forma análoga, o ponto de ocorrência de E em M2 define o ambiente B, ao qual também pertence o "label" R de retorno do tratador T, isto é, a ocorrência da condição Y (final) de T implica em uma transferência para o ponto R de B. A fórmula abaixo denota esta situação:

$$Y \Rightarrow B(R) \qquad (7)$$



A figura abaixo esquematiza, em conjunto, as ocorrências acima mencionadas.



Seja então

<u>tratador</u> t(K); L:T

a declaração de um tratador, onde t é o seu nome, K é especificação dos parâmetros formais, L é o seu "label" de entrada e T é o corpo do tratador.

A semântica de ativação do tratador T é expressa por:

$$\frac{X\ \{T\}\ Y}{\forall w,z\ (X\ \{t(w)\}\ Y)} \qquad (8)$$

onde t(w) indica uma ativação de t.

Provar, então, a correção de M2 consistirá em provar:

$$P2\ \{M2'\}\ \neg C,\ P2' \qquad (9)$$

$$P2'\ \{M2''\}\ Q2 \qquad (10)$$

$$P2\ \{M2'\}\ C \qquad (11)$$

$$(C)_{a}^{x} \Rightarrow (X)_{b}^{w} \qquad (12)$$

$$X\ \{T\}\ Y \qquad (13)$$

$$(Y)_{b}^{w} \Rightarrow (D)_{a}^{x} \qquad (14)$$

$$D\ \{S;M2''\}\ Q2 \qquad (15)$$

onde M2' corresponde a parte de M2 desde seu início até o ponto E. S é o comando cujo "label" é R e M2'' são comandos que se seguem. As fórmulas (9) e (10) correspondem aos casos em que não ocorre a condição C. As demais correspondem ao caso em que ocorre C.

Entretanto, a característica de construtibilidade implica no desconhecimento de T durante a prova de M2. Tendo em vista este fato, reformularemos o critério de correção de um módulo, incluindo, na parte referente ao tratamento da exceção a exigência da existência de um tratador T, desconhecido, tal que:

$$X \{T\} Y \tag{16}$$

$$(C)^{*}_{a} \Rightarrow (X)^{\omega}_{b} \tag{17}$$

$$(Y)^{\omega}_{b} \Rightarrow (D)^{*}_{a} \tag{18}$$

Denotaremos esta situação pela fórmula:

$$\exists T \; (C \overset{T}{\Rightarrow} D) \tag{19}$$

Assim, a semântica da execução de um módulo é expressa pela regra:

$$\frac{(P \; \{M'\} \; \neg C, P'; \; P' \; \{M''\} \; Q), \; (P \; \{M'\} \; C; \; D \; \{S; M''\} \; Q)}{P \; \{M\} \; Q} \tag{20}$$

desde que: $\exists T \; (C \overset{T}{\Rightarrow} D)$

Na regra (20) acima, M, M' e M'' têm significados

análogos a respectivamente M2, M2' e M2'' das fórmulas (9), (10) e (15).

Até agora supomos que durante a execução de um módulo só possa ocorrer uma exceção. Suponhamos, então, que possam ocorrer diferentes exceções Ei, cada uma delas associada a condições de ocorrência Ci, i=1,...,n. Isto implicará na existência de correspondentes ambientes Bi em M2.

O ambiente A de M1, por sua vez, será ampliado para conter tantos tratadores Tj quantos são as exceções Ei de M2 (veremos adiante que é possível termos menos tratadores do que exceções). Da mesma forma que anteriormente, cada tratador possuirá um "label" de entrada L(Tj); uma asserção inicial Xj e uma final Yj.

Neste caso, a ocorrência da exceção Ei sob a condição Ci implicará na transferência para o tratador cujo "label" L(Tj) é igual a Ei. A seleção e transferência para o tratador Tj é expressa pela seguinte fórmula:

$$Ci \Rightarrow A(L(Tj)) \mid Ei = L(Tj) \qquad (21)$$

A retomada da execução de M2 é expressa por:

$$Yj \Rightarrow Bi(R) \qquad (22)$$

Estendendo a regra (20) para conter este conceito, temos a seguinte regra de prova para um módulo:

$$\frac{(P\ \{Mi'\}\ \neg Ci, Pi';\ Pi'\ \{Mi''\}\ Q),\ (P\ \{Mi'\}\ Ci;\ Di\ \{Si;Mi''\}\ Q)_{i=1,..,n}}{P\ \{M\}\ Q} \qquad (23)$$

$$\text{desde que } \forall i \quad \exists T \quad (Ci \overset{T}{\Rightarrow} Di),\ i=1,..,n \qquad (24)$$

Vimos no capítulo 3 que diferentes tipos de exceções podem requerer diferentes formas de tratamento. Vimos, no entanto, que não é uma boa prática antecipar, através de construções do mecanismo, as formas genéricas de tratamento das exceções, uma vez que poderíamos incorrer no risco de sermos incompletos. Além disso, para fins de verificabilidade do módulo que provoca exceções, é necessáario conhecer-se a lista de possíveis respostas que o tratador possa dar a exceção. Não se trata de conhecer qual o tratamento que será dado, mas sim qual a forma de retomar a execução após o tratamento.

Com esta finalidade, ampliaremos os ambientes Bi para conter, cada um deles, diferentes pontos de retomada (labels) que denotaremos por:

$$Ri,k \text{ para } k=1,..,mi \qquad (25)$$

Estes "labels" correspondem aos possíveis pontos de M2 a partir dos quais pode ser retomada a execução domódulo após a ocorrência de Ei.

A cada um destes pontos estará associada uma asserção Di,k que deverá se verificar quando a retomada da execução de

M2 se se der a partir do ponto Ri,k, sendo que a escolha do ponto de retomada é feita pelo tratador da exceção correspondente.

Para tal, cada tratador Tj poderá terminar sua execução em diferentes condições Yj,k.

Desta forma, a seleção do ponto de retomada da execução de M2 e a consequente transferência pode ser expressa por:

$$Yj,k \Rightarrow Bi(Rk) \tag{26}$$

ou seja, se a execução de Tj terminar em Yj,k, a execução de M2 será reiniciada em Bi(Rk).

Com a introdução deste novo conceito, a regra de prova de um módulo passa a ser:

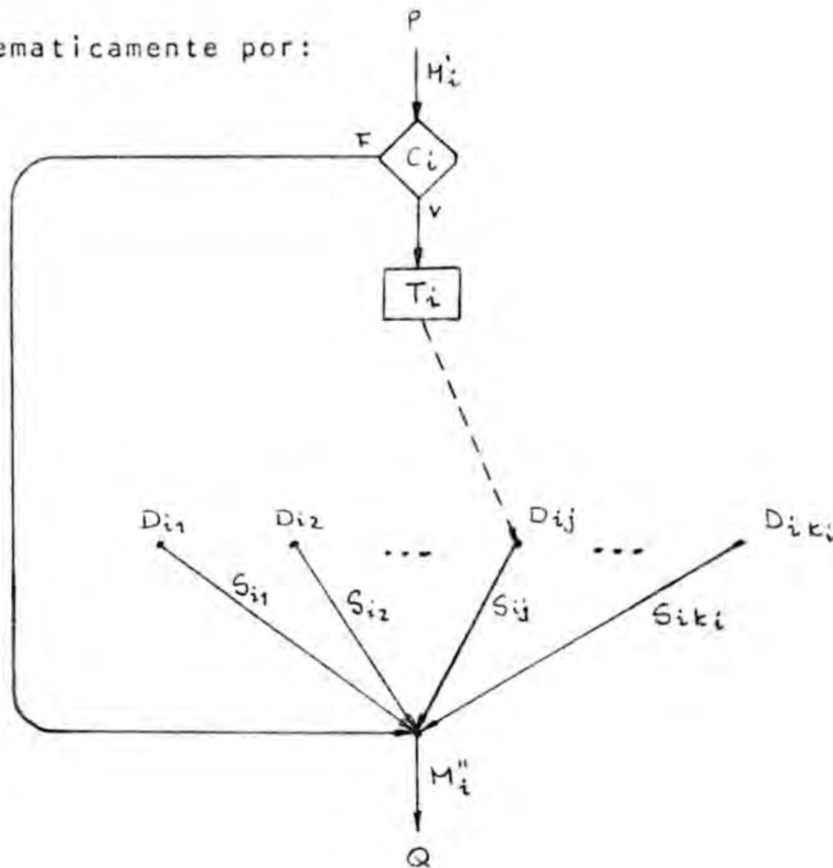
$$\frac{(P\ \{Mi'\}\ \neg Ci, Pi';\ Pi'\ \{Mi''\}\ Q),\ (P\ \{Mi'\}\ Ci;\ Dij\ \{Sij;Mi''\}\ Q)_{i=1,..,n;\,j=1,..,ki}}{P\ \{M\}\ Q} \tag{27}$$

desde que:

$$\forall i\ (i=1,\dots,n)\ \exists T\ \exists j\ (j=1,\dots,ki)\ (Ci \overset{T}{\Rightarrow} Di,j) \tag{28}$$

A situação caracterizada em (28) pode ser representada

esquematicamente por:



Por se tratar do desenvolvimento de um modelo de tratamento de exceções, estão sendo omitidos uma série de detalhes, os quais serão considerados por ocasião da apresentação do mecanismo propriamente dito, ainda neste capítulo. Nesse sentido, o modelo que estamos apresentando considera apenas situações "ideais", não levando em conta, por exemplo, a ocorrência de casos anômalos como a inexistência de um tratador para uma dada exceção ou, então, o caso de aborto, isto é, o caso em que não há retomada do módulo sinalizador da exceção.

Tratadores Internos.

Segundo o modelo até aqui desenvolvido, sempre que ocorrer uma exceção em um módulo, o tratamento da mesma será feito por um tratador textualmente contido no ambiente de chamada deste módulo. Podemos, no entanto, de forma análoga, caracterizar a existência de um ambiente AI pertencente ao módulo, definido por um conjunto de tratadores TIi (i=1,..,m), cada um deles identificado por "labels" L(TIi). Da mesma forma que no caso anterior, cada tratador é considerado como sendo um sub-módulo, possuindo parâmetros formais $\underline{x}$ e asserções iniciais e finais.

No caso de tratadores internos, temos as seguintes fórmulas para caracterizar os casos abaixo:

a) seleção e tranferência para tratador interno:

$$Ci \Rightarrow AI\ (L(TIi)) \mid L(TIi) = Ei \quad (29)$$

b) execução de tratador interno:

$$Ci\ \{TIi\}\ Di,j \quad i=1,..,n;\ j=1,..,ki \quad (30)$$

c) retomada da execução do módulo, após a execução de TI:

$$Di,j \Rightarrow B(Rj);\ i=1,..,n; j=1,..,ki \quad (31)$$

d) regra de prova do módulo:

$$\frac{(P\ \{Mi\}\ \neg Ci, Pi';\ Pi'\ \{Mi''\}\ Q),\ (P\ \{Mi'\}\ Ci;\ Ci\ \{TIi\}\ Dij;\ Dij\ \{Sij; Mi''\}\ Q);\ i=1,..,n; j=1,..,ki}{P\ \{M\}\ Q} \quad (32)$$

Veja que, neste caso, não é necessário o critério de existência (28) anteriormente mencionado.

6.2 - Descrição do Mecanismo.

Usaremos, para fins de apresentação do mecanismo proposto, os conceitos formulados no modelo ja apresentado. A descrição do mecanismo será feita através da apresentação simultânea de uma descrição informal do mesmo, da sua sintaxe (BNF) e da sua semântica (regras de inferência).

A semântica do mecanismo será apresentada usando-se a notação proposta por Ralph-Johan Back [Back 80]:

$$A \vdash P: S$$

onde A é um conjunto de "labels" L(Ti), (i=1,..,m). Cada "label" tem, associado a sí, uma asserção Xi. P é a pré-condição e S um comando do mecanismo. A fórmula estabelece que, se P é verdadeiro antes da execução de S, e se a execução de S terminar no "label" L(Ti) de A, então a asserção Xi deve se verificar.

Declaração de uma Exceção.

A declaração de uma exceção é usada com o fim de declarar a existência, dentro do módulo onde aparece a declaração, de um objeto do tipo $nexceção. Na declaração tambem é indicada a visibilidade da exceção.

As exceções detectadas e explicitamente sinalizadas pelo usuário podem ser de dois tipos, no que se refere ao alcance do sinal. Dizemos que uma exceção é visível ao ativador, se o seu sinal pode atingir o ativador. Caso contrário, isto é,

caso em que o efeito da sinalização da exceção seja restrito ao ambiente do módulo sinalizador, dizemos que a exceção é <u>invisível</u> ao ativador. Os conceitos de visibilidade e invisibilidade de uma exceção são relativos ao módulo que ativou o sinalizador. Portanto, no caso de exceções visíveis, a visibilidade de restringirá ao módulo ativador, permanecendo

O conceito de visibilidade de uma exceção é importante, pois as exceções da implementação devem ser invisíveis ao usuário, para permitir a preservação do encapsulamento da referida implementação. Este conceito também pode ser usado para implementar o princípio de "last wishes" proposto por [Ichbiah 79].

A declaração de uma exceção é feita da seguinte forma:

<u>exception</u> <nome> (<params>) <visibilidade>

<u>Sinalização de uma Exceção</u>.

A sinalização de uma exceção faz com que a execução do módulo sinalizador seja suspensa, para iniciar a execução do módulo de tratamento da exceção.

Ja vimos, durante o estudo do modelo anteriormente apresentado, o significado do tratamento interno e externo de uma exceção, bem como as regras de verificação de tratadores. Resta, pois, estabelecer os critérios usados pelo mecanismo

para selecionar entre tratador externo e interno, conforme a visibilidade da exceção.

A sinalização de uma exceção é feita através do comando

<u>raise</u> Ei (<parlist>)

onde Ei é o nome da exceção.

No caso de Ei ser invisível, o seu tratador será obrigatoriamente interno. Nesta caso, a regra de prova é:

$$\frac{Ci \Rightarrow AI\ (Ei)}{AI \vdash Ci:\ \underline{raise}\ Ei(a)} \qquad (33)$$

onde AI (Ei) representa a asserção associada a Ei no ambiente AI.

Para o caso de Ei ser uma exceção visível, devemos estabelecer condições de existência de tratadores. A existência de um tratador externo para Ei é definida por:

$$Pe = \exists\, TI \in A \mid (L(TI) = Ei) \qquad (34)$$

enquanto a existência de um tratador interno é definido por:

$$Pi = \exists\, TI \in AI \mid (L(TI) = Ei)\ \&\ \neg Pe \qquad (35)$$

O critério de escolha entre tratador externo e interno é expresso pela regra de prova abaixo:

$$\frac{(Ci\ \&\ Pe \Rightarrow A\ (Ei));\ (Ci\ \&\ Pi \Rightarrow AI\ (Ei))}{A,\ AI \vdash Ci:\ \underline{raise}\ Ei\ (a)} \qquad (36)$$

Ou, em outras palavras, o tratador interno somente será executado caso não exista um tratador externo para a exceção.

Este critério de escolha permite que se construam módulos nos quais todas as possíveis exceções tenham tratadores, independentemente do usuário prover ou não tratadores próprios. Naturalmente, na maioria dos casos, o tratamento poderá se resumir na emissão de mensagem de erro com consequente cancelamento do módulo, dado que um tratador interno não poderá, nestes casos, interpretar o significado das exceções.

O quadro abaixo resume o critério de escolha do tratador.

há tratador interno?	há tratador externo?	então o tratamento é
sim	sim	externo
sim	não	interno
não	sim	externo
não	não	não há

Na hipótese configurada na última linha do quadro acima, ocorrerá uma exceção implícita do sistema, denominada failure

As exceções do tipo failure são geradas no mesmo módulo onde ocorreu a exceção para a qual não foi provido nenhum tratador. Para uma exceção desse tipo, também podem ser

previstos os mesmos tratamentos de uma exceção do usuário. Exceções do tipo <u>failure</u> têm a seguinte forma:

<u>failure</u> (<nome-da-exceção>,<nome-do-módulo>,<histórico>)

onde <nome-do-módulo> é o nome do módulo onde a exceção foi detectada e <nome-da-exceção> é o mome da exceção para a qual não há tratador.

O terceiro parâmetro, <histórico>, é um 'string', que na primeira ocorrência de <u>failure</u> é vazio. Caso não seja previsto um tratamento para ele, será gerado um outro <u>failure</u>, no qual o seu parâmetro <histórico> será concatenado com os parâmetros do <u>failure</u> anterior, e assim por diante. Desta forma, a origem de um <u>failure</u> pode ser recomposta, mesmo que tenha sido detectado vários módulos acima do ponto onde foi originalmente causado.

Como exemplo, suponha um módulo 'inverte-matriz' que obtem a inversa de uma matriz dada e no qual está prevista a exceção 'divisão-por-zero'. Este módulo é chamado por outro, digamos 'resolve-sistema'. Em nenhum dos dois módulos é prevista qualquer tratamento para 'divisão-por-zero'. Caso esta ocorra, será gerada, no módulo 'inverte-matriz', a exceção:

<u>failure</u> ('divisão-por-zero','inverte-matriz',<u>nil</u>)

que pode ser interpretado da seguinte maneira:

'divisão por zero no módulo inverte-matriz'

caso não haja nenhum tratador previsto em 'resolve-sistema', será gerado outro failure:

failure ('failure','resolve-sistema',
'divisão-por-zero,inverte-matriz')

que pode ser interpretada como:

'falha no módulo resolve-sistema devida a ocorrência de divisão-por-zero no módulo inverte-matriz'.

A ocorrência de failure provoca a terminação da execução do módulo onde ocorreu a exceção.

O esquema acima descrito para a ocorrência de "failure" viola, em parte, o conceito de encapsulamento da implementação, uma vez que é exibida a cadeia de ativacção desde o ponto de ocorrência da excecção até o seu ponto de tratamento. Entendemos, no entanto, que este fato não prejudica o mecanismo, pois a ocorrência de "failure" pode ser entendida como sendo um erro e que, portanto, deve ser corrigido. Para isto se torna necessário apontar a origem do mesmo. Seria possível, no entanto, criar um comando do tipo "clear failure

list" para remover parte ou todo o histórico da lista de "failure".

Tratadores de Exceções.

Os tratadores de exceções, como foi caracterizado no modelo, podem ser internos ou externos, dependendo do ambiente no qual se situam. Os tratadores externos se situam no ambiente de chamada do módulo sinalizador das exceções. Para este caso, proporemos a seguinte sintaxe:

```
          <association> ::= <module-call> <exception-part-list>

<exception-part-list> ::= <exception-list> |
                          <exception-list> <exception-part-list>

     <exception-list> ::= "[" "on "<exception-name-list> <parlist>
                          "=>" <handler-name> ":" <handler-body>
                          "end" <handler-name> "]"

<exception-name-list> ::= <exception-name> |
                          <exception-name> <exception-name-list>

            <parlist> ::= "(" <parms> ")"| <empty>

              <parms> ::= <variable-name> |
                          <variable-name> "," <parms>
```

Esta descrição permite construções como a abaixo esquematizada:

```
call sub (...)
     [on E11, E12, E13, ... , E1a => h1: ... end h1]
     [on E21, E22, E23, ... , E2b => h2: ... end h2]
     [on E31, E32, E33, ... , E3c => h3: ... end h3]
     .
     .
     .
     [on En1, En2, En3, ... , Enm => hn: ... end hn]
```

Uma vez que as exceções acima só ocorrem uma de cada vez, a ordem das <exception-list> é irrelevante. É necessário, apenas, que a interseção das <exception-name-list> seja vazia, isto é, que cada exceção só apareça referenciada uma vez.

No caso de tratadores internos, a sintaxe pode ser semelhante, só mudando a associação com o ambiente, que no caso passa a ser o mesmo do módulo sinalizador:

```
<module> ::= <module-head>
             <exception-part-list>
             <module-body>
```

As regras de prova dos tratadores são as mesmas formuladas no modelo anteriormente apresentado.

Retomada do Processamento.

A retomada do processamento corresponde à transferência da execução do tratador para o módulo sinalizador, conforme mostrado no modelo. A retomada é provocada pela execução do comando:

return to Rj

cuja regra de prova é:

$$\frac{Xj \Rightarrow Bi\ (Rj)}{Bi \vdash Xj:\ \underline{\text{return}}\ \underline{\text{to}}\ Rj} \qquad (37)$$

A definição dos pontos Rj no ambiente B é feita segundo a seguinte regra de sintaxe:

```
<return-environment> ::= <label> ":" <statement>; |
                         <label> ":" <statement>;
                         <return-environment>
```

De acordo com o que foi proposto no modelo, o ambiente de retorno é fisicamente contíguo ao comando de sinalização raise, como no exemplo abaixo:

```
raise Ei (<parlist>)
    R1: S1;
    R2: S2;
     .
     .
     .
    Rn: Sn;
```

Tratamento sem Retomada.

Na formulação do mecanismo proposto, sempre supomos, até agora, que o modelo adotado fosse do tipo "resumption", isto é, após o tratamento de uma exceção, sempre ocorre a retomada da execução do módulo sinalizador da mesma.

Na verdade, para que o mecanismo seja completo (segundo um dos requisitos formulados), ele não deve impor nenhuma regra deste tipo ao usuário, isto é, deve deixar a ele a decisão de efetuar ou não a retomada da execução do sinalizador.

Na hipótese de ser adotada a opção de não retomar o processamento do módulo sinalizador, a semântica do comando de sinalização é descrita pelo axioma:

$$Ci \{ \underline{raise}\ Ei \}\ \underline{false} \qquad (38)$$

Usaremos, para indicar a opção de aborto, o comando:

<u>abort</u>

Da mesma forma que o comando <u>return to</u>, o comando <u>abort</u> só pode ser usado dentro de tratadores de exceções.

Seja V o ponto do ambiente A correspondente ao ponto de retorno de M2. A este ponto, como foi caracterizado no modelo, está associada a asserção Q2, que deve se verificar após a execução de M2. Seja, também, a asserção Ai que é verdadeira imediatamente antes da execução do comando <u>abort</u>. Então a regra de prova do comando <u>abort</u> é dada por:

$$\frac{Ai \Rightarrow Q2}{K \,\&\, A(V) \vdash Ai\text{: } \underline{abort}} \qquad (39)$$

onde A(V) denota a asserção associada ao ponto V em A. K é um predicado que denota a ocorrência de aborto de M2.

A existência de K é necessária para permitir ao usuário distinguir, a partir do ponto V, se houve ou não ocorrência de aborto, que corresponde ao "termination model" das linguagens ADA [Ichbiah 79] e CLU [Liskov, Snyder 79].

Note-se que qualquer que seja o modelo usado ("resumption" ou "termination"), a verificabilidade do módulo M2 fica assegurada, uma vez que é requerido que, em qualquer caso, se verifique:

$$P \{M\} Q$$

No caso de existência de um comando de aborto em um tratador, a asserção Q2, que deve se verificar tanto no caso de retorno normal como no caso de aborto do tratamento, é formada pela disjunção da condição de retorno de M2 e da condição de ocorrência de aborto. Q2 é, portanto, a "strongest pos-condition" do ponto V.

Exceções Exógenas.

Os recursos necessários para permitir o tratamento de exceções exógenas são mais simples, como veremos adiante.

No capítulo 2 dissemos que as exceções exógenas podiam ser vistas como sendo uma espécie de interrupção. Na verdade, tratam-se de interrupções qualificadas, dirigidas a um dado módulo, o qual fará executar um tratador, se a execução do módulo estiver em um ponto no qual o módulo pode perceber (sentir) a ocorrência da exceção. Assim, o único efeito de

controle produzido pela exceção é o de transferência para o tratador da exceção.

Como a sensibilidade do módulo às exceções é espacial, o efeito de uma exceção não é necessariamente imediato à ocorrência da mesma. O efeito (tratamento) ocorrerá quando e se a execução do módulo atingir o ponto sensível à exceção.

Uma exceção exógena pode ter parâmetros associados, os quais poderão ser utilizados pelo tratador. Estes parâmetros, transmitidos pelo módulo sinalizador, são do tipo "value" (sob o ponto de vista do módulo receptor) e só têm existência lógica dentro do tratador da exceção correspondente.

A definição dos pontos sensíveis às exceções exógenas pode ser feito de acordo com a seguinte sintaxe:

```
when Ei (...) =>
   [hi: Ti end hi]
```

Sempre que a execução do módulo atingir este comando, será executado o tratador Ti se a exceção Ei tiver ocorrido. Neste caso, após o tratamento da exceção, o seu sinal é "desligado". Caso a exceção não tenha ocorrido, a execução do módulo prossegue a partir do comando que se segue ao when.

A regra de inferência abaixo fornece uma descrição mais precisa do comando:

$$\frac{P \;\&\; Ei \;\{Ti\}\; Q,\; P \;\&\; \neg Ei \Rightarrow Q}{P \;\{\; \underline{\text{when}}\; Ei\; (\ldots)\; \ldots \;\}\; Q,\; \neg Ei} \qquad (40)$$

onde P e Q são as asserções que antecedem e sucedem, respectivamente, o comando when.

Cabe, aqui, uma advertência, no sentido de alertar o projetista de um módulo de que os "pontos sensíveis" às exceções exógenas devam ser tais que tenham significado lógico ao usuário, isto é, sejam pontos tais que possam ser identificados na especificação do módulo.

A prova da correção do tratador de exceções exógenas é (deve ser) desenvolvida pelo projetista do módulo que o contém, e é conduzida de forma similar a dos tratadores de exceções endógenas, exceto pelo fato de não ser necessário que os parâmetros sejam distintos entre sí, uma vez que todos são do tipo "value".

6.3 - Método de Especificação.

Nesta seção propomos, sumariamente, uma forma de especificação da parte relativa a exceções de um módulo. Por este motivo, não nos preocuparemos, pois foge do escopo deste trabalho, com a especificação do módulo em sí, a qual pode ser feita por várias propostas existentes, destacando-se, principalmente, o método proposto por Parnas [Parnas 72a].

A especificação das exceções deve, naturalmente, prover o usuário do módulo com informações suficientes para que este possa:

a) completar a verificação do módulo especificado, no que diz respeito ao tratamento externo das exceções endógenas;

b) conduzir o processo de verificação do módulo em desenvolvimento;

c) conhecer as condições nas quais o módulo sinaliza as exceções.

Naturalmente, a especificação deve ser tal que não mostre a implementação usada.

Assim, a especificação deverá conter:

a) as asserções iniciais e finais do módulo (P e Q);

b) as condições (Ci) de sinalização das exceções;

c) a declaração das exceções visíveis, mostrando os parâmetros que estas transmitem, bem com os seus tipos;

d) os pontos de retomada das exceções, com as respectivas asserções (Di);

e) o efeito dos tratadores internos das exceções visíveis, se houver.

A especificação pode ser feita, por exemplo, por meio de um esquema semelhante ao abaixo sugerido.

```
module m (<parlist>) is
   entry: P
   exit:  Q
   exceptions:
     C1 raises E1 (<parlist1>)
        R1: D1; R2: D2; ..., Rk1: Dk1
     C2 raises E2 (<parlist2>)
        R2: D1; R2: D2; ..., Rk2: Dk2

     ...

     Cn raises En (<parlistn>)
        R1: D1; R2: D2; ...; Rkn: Dkn
   end of exceptions
```

6.4 - Implementação.

Nesta seção é descrito um modelo de implementação do mecanismo proposto.

A implementação que é sugerida a seguir se baseia em um processo de pré-compilação, supondo uma linguagem do tipo Algol, que suporte a construção de co-rotinas [Pratt 75], sendo usado o comando resume para a retomada do processamento.

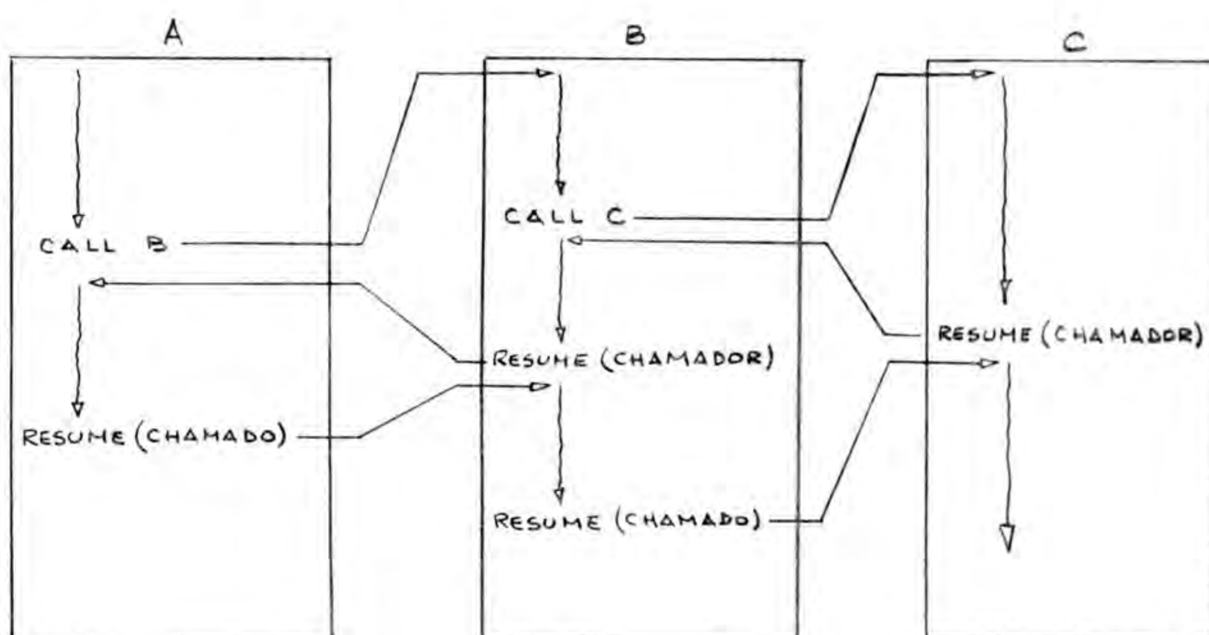
Normalmente um comando do tipo resume provoca a retomada do processamento do módulo que ativou este que executou o resume. Entretanto, tendo em vista que o mecanismo exige a possibilidade de efetuar a retomada tanto do módulo chamado como do chamador, suporemos a existência das seguintes variantes do comando resume:

a) resume (chamador);

b) resume (chamado);

O primeiro provoca a retomada de execução do módulo que ativou este que executou o comando, sem provocar a sua desativação. O segundo provoca a retomada da execução do módulo chamado, o qual já havia executado, anteriormente, um comando <u>resume</u> (chamador).

Para exemplificar, suponha que o módulo A chame B, que por sua vez chama C. A figura abaixo ilustra o efeito da execução dos comandos <u>resume</u>.



No modelo de implementação não nos preocupamos com outros fatores que não sejam exclusivamente funcionais. Assim, aspectos de eficiência e otimização não são considerados.

A implementação descrita a seguir baseia-se no método "branch table", que consiste em construir, após cada chamada de módulo, uma tabela que contenha uma entrada para cada exceção deste módulo a ser tratada no seu exterior. Além das entradas existentes para cada exceção listada, é acrescentada uma entrada extra para a exceção "failure", caso o usuário não tenha previsto a sua ocorrência.

Cada entrada nesta tabela é composta pelo nome de uma exceção e pelo seu tratador. A figura abaixo mostra o esquema desta tabela.

```
call sub (...)
  E1: tratador para E1
  E2: tratador para E2

  ...

  En: tratador para En
  Failure: tratador para "failure".
```

O uso da "branch table" permite uma facil implementação do retorno e da sinalização de exceções: o retorno do módulo chamado, isto é, o caso em que não ocorrem exceções na sua execução, corresponde a retomar o processamento do módulo ativador a partir do comando seguinte à tabela. No caso de ocorrência de exceções no módulo, o controle será transferido para o ponto da tabela que contenha o nome da exceção sinalizada ou, na falta deste, para o ponto correspondente à entrada "Failure".

Além da tabela, a chamada do módulo será precedida de comandos que montam uma lista contendo o nome de todas as exceções que estão previstas na chamada do módulo. Esta lista será passada para o módulo, para que este saiba quais, dentre as exceções que ele provoca, aquelas que terão tratamento externo.

Tendo em vista que o mecanismo exige, em algumas situações, a troca de informações entre o módulo sinalizador e o tratador das exceções, é acrescentada, à lista de parâmetros do módulo, uma outra lista cujos parâmetros serão usados para auxiliar a troca de informações. Assim, a lista de parâmetros será composta por duas sub-listas, como abaixo:

call sub (<parlist>,<controllist>)

onde <parlist> é a lista de parâmetros que o usuário especificou e <controllist> é a lista de controle.

A lista de controle é composta pelos seguintes elementos:

a) modo - do tipo "string". Seu conteudo indica se haverá retomada da execução do módulo sinalizador ou se ocorrerá a sua desativação (aborto);

b) rótulo - do tipo "string". Seu contéudo é o nome do ponto de retomada de execução do módulo sinalizador, caso esta seja a opção indicada em modo.

UFRGS
INSTITUTO DE INFORMÁTICA
BIBLIOTECA

c) lista-de-exceções - do tipo "lista". Contém a lista dos nomes das exceções para as quais existem tratadores externos;

d) nome-da-exceção - do tipo "string". Seu conteudo, se diferente de nil, indica qual a exceção que ocorreu;

e) parâmetros - do tipo "lista-de-parâmetros". Contém a lista dos parâmetros usados por todas as exceções da lista lista-de-exceções;

f) nome-do-módulo - do tipo "string". Usado para transmitir o nome do módulo sinalizador, no caso de ocorrência de "failure";

g) motivo - do tipo "string". Usado para transmitir o motivo da ocorrência de "failure";

h) histórico - do tipo "string". Usado para transmitir o histórico de "failure".

A partir dos elementos acima caracterizados, temos a

seguinte implementação da "branch table":

```
call sub (<parlist>,<controllist>)
  t: if nome-da-exceção ≠ nil then
       begin
         case nome-da-exceção of
           E1: begin {tratador para E1} end;
           E2: begin {tratador para E2} end;
           .
           .
           .
           En: begin {tratador para En} end;
           else: begin {tratador para "failure"} end
         end case;
         go to t    {verifica se há nova exceção}
       end
```

Para a implementação do comando <u>raise</u> são necessárias as funções:

$$\text{invisível (E)} \begin{cases} =T, \text{ se a exceção E foi declarada invisível.} \\ =F, \text{ caso contrário.} \end{cases}$$

$$\text{existe (E,l)} \begin{cases} =T, \text{ se a exceção E está na lista l.} \\ =F, \text{ caso contrário.} \end{cases}$$

Supondo, ainda, os seguintes endereços:

e = "entry point" do tratador interno de E.

c = endereço do comando seguinte ao <u>raise</u>.

A implementação do comando raise pode ser:

```
if invisível (E) or
   ¬ existe (E,1)
   then begin
           assign c to e';
           go to e          { executa tratador interno }
        end
   else begin
           nome-da-exceção := E;
           resume (chamador) { executa tratador externo }
        end.
```

Obs.: A variável e' é do tipo "label", sendo usada para permitir a retomada do processamento normal após o tratamento interno da exceção (vide a implementação de tratador interno, nas páginas seguintes).

A parte correspondente aos pontos de retomada da execução do sinalizador pode ser implementada de forma similar à parte dos tratadores, por meio do uso de "branch table", de forma idêntica à anteriormente descrita, na qual cada entrada corresponderá a um ponto de retomada.

É necessário, também, distinguir entre os casos de retomada através do comando return to e o caso de uso do comando abort. Esta distinção pode ser feita pelo parâmetro de controle modo. Caso modo = "return", o parâmetro rótulo conterá o "label" do ponto de retorno. Caso modo = "abort", será simplesmente executado um retorno (o que provocará a desativação do sinalizador) ao módulo ativador.

A implementação da "branch table" de retorno é mostrada a seguir:

```
c: case modo of
     "abort": return;
     "return": begin
                case label of
                  R1: S1;
                  R2: S2;

                  ...

                  Rn: Sn
                end case
   end case
```

Segundo o modelo proposto para implementar os pontos de retomada, é necessário que os comandos return to e abort sejam implementados como a seguir:

```
                     begin
                       modo := "return";
                       rótulo := "Ri";
return to Ri;    {     resume (chamado);
                       nome-da-exceção := nil {apaga exceção}
                     end

                     begin
                       modo := "abort";
                       set-aborted (chamado);
     abort;      {     resume (chamado);
                       nome-da-exceção := nil {apaga exceção}
                     end
```

Obs.: o endereço t corresponde ao ponto inicial da "branch table" dos tratadores de exceções.

Implementação de tratador interno da exceção E (cujo "entry point" é _e_), caso este seja fornecido pelo projetista do módulo sinalizador:

```
e: begin
     <corpo-do-tratador>
     go to e'
   end
```

Caso a exceção não tenha tratador interno (fornecido pelo projetista), será gerado o seguinte código:

```
e: begin
     nome-da-exceção := "failure";
     motivo := "Ei";
     nome-do-módulo := "nome-do-módulo";
     histórico := nil;
     return
   end
```

Caso não existe um tratador interno definido para "failure", será gerado:

```
failure: begin
           nome-da-exceção := "failure";
           nome-do-módulo := "nome-do-módulo";
           histórico := nil;
           return
         end
```

O nome das exceções declaradas _invisíveis_ é colocada em uma lista <lista-de-invisíveis> para ser consultada pelo módulo invisível (nome-da-exceção).

Exceções Exógenas.

O método aqui proposto para implementar as exceções exógenas utiliza um "vetor de interrupções" VI, onde cada elemento deste vetor é associado à uma exceção exógena declarada no módulo. Assim, cada ativação de um módulo terá um vetor de interrupção de tantos elementos quantos forem as exceções exógenas declaradas no mesmo módulo.

Cada entrada no vetor é composta de dois elementos:

VI (Ei) = <chave, parâmetros>

onde chave é uma variável lógica que indica se a exceção Ei ocorreu ou não. O componente parâmetros contém os parâmetros da exceção Ei.

Para identificar a ocorrência de uma exceção Ei, o módulo consulta o componente chave da entrada VI (Ei) do vetor:

chave = on => ocorreu a exceção Ei

chave = off => não ocorreu a exceção Ei

A implementação do comando when pode, então, ser feita do do seguinte modo:

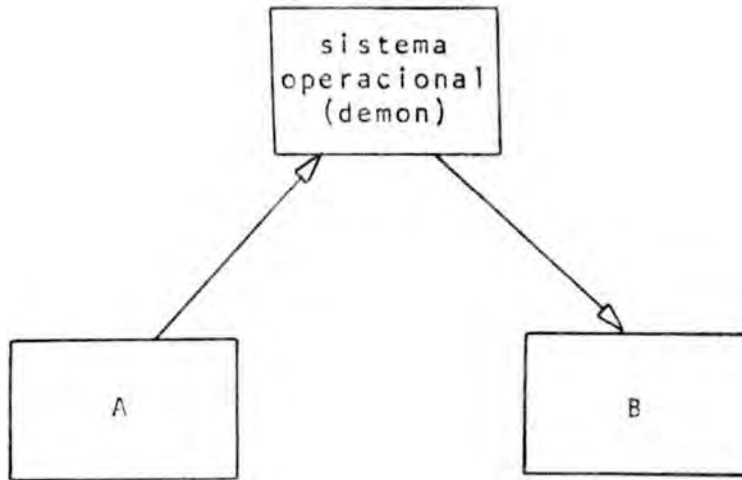
```
P (Si);
chave := obtêm-chave (VI, Ei);
if chave
  then begin
         desliga-chave (VI, Ei);
         <corpo-do-tratador>;
         V (Si)
       end
  else V (Si);
```

onde Si é uma variável semáfora associada à exceção Ei. P (Si) e V (Si) são as operações sobre variáveis semáforas, conforme [Dijkstra 68].

Como pode se ver pelo modelo proposto para implementar exceções exógenas, a re-sinalização de uma exceção não produzirá nenhum efeito enquanto não for iniciado o tratamento da sinalização anterior.

A tarefa de sinalizar uma exceção exógena é exercida pelo sistema operacional, o qual proverá, através de módulos "demon", a ligação entre os módulos sinalizador e sinalizado. Neste modelo, sempre que um módulo desejar sinalizar a ocorrência de uma exceção exógena em outro módulo, aquele fará o pedido ao módulo "demon" que, por sua vez, ligará a chave correspondente no vetor de interrupções do módulo receptor.

A figura abaixo esquematiza a situação onde o módulo A sinaliza uma exceção para o módulo B.


sistema
operacional
(demon)
A
B

Capítulo 7 - Validação do Mecanismo.

Mostraremos, neste capítulo, a validade do mecanismo proposto com relação aos requisitos formulados, isto é, que a proposta formulada neste trabalho atende aos requisitos colocados no capítulo 4, e que são os seguintes:

1) verificabilidade, no sentido de que deve ser possível verificar a correção dos módulos que utilizam as construções do mecanismo;

2) encapsulamento, no sentido de que o mecanismo deve assegurar meios que permitam manter o encapsulamento do módulo, isto é, que o uso do mecanismo não implique na violação desta regra, inclusive durante o processo de verificação;

3) construtibilidade, no sentido de que o uso do mecanismo deve permitir a construção de módulos sem ser necessário conhecer o ambiente no qual ele será usado;

4) acoplamento, no sentido de que o uso do mecanismo não deva aumentar a troca de informações entre os módulos de forma significativa;

5) adequação ao ambiente, no sentido de que ele possa ser usado para os problemas de tratamento de exceções em ambientes modulares.

Cabe, inicialmente, uma observação já anteriormente mencionada neste trabalho: estamos supondo que o mecanismo será usado em uma linguagem que suporte modularidade, conforme caracterizado no capítulo 3, sendo que algumas das características acima enumeradas são originalmente fornecidas pela linguagem hospedeira e não pelo mecanismo. A este cabe apenas preservá-las ou mantê-las em níveis toleráveis. Assim, nestes casos, mostraremos apenas que o mecanismo fornece meios que asseguram estes requisitos.

Veremos, também, que alguns requisitos (encapsulamento e construtibilidade) estão fortemente comprometidos com o requisito de verificabilidade, isto é, que aqueles requisitos são assegurados devido ao método de verificação proposto para o mecanismo.

7.1 - Verificabilidade.

Segundo o que foi proposto no capítulo 6, a prova da correção de um módulo M é conduzida em duas etapas distintas. Na primeira etapa de prova, conduzida pelo projetista do módulo, deve ser provado:

$$(P\ \{Mi'\}\ \neg Ci, Pi';\ Pi'\ \{Mi''\}\ Q),\ (P\ \{Mi'\}\ Ci;\ Dij\ \{Sij; Mi''\}\ Q),\ i=1,..,n; j=1,..,ki$$

onde a primeira parte da fórmula corresponde ao caso em que não ocorrem exceções, enquanto que a segunda parte corresponde à ocorrência de exceções.

Na hipótese de serem providos tratadores internos para as exceções Ei do módulo, o projetista ainda deverá provar a sua correção, provando que:

$$Ci\ \{\ Ti\ \}\ Di,j; \quad j=1,..,ki$$

Uma vez vencida esta etapa de prova, o módulo M pode ser devidamente catalogado para uso futuro (o que lhe imprime a característica de construtibilidade).

A segunda etapa de prova é conduzida por parte do usuário do módulo. Este deverá provar que os tratadores Ti que ele prover para as exceções Ei de M são tais que garantam que:

$$\forall i\ \exists Ti\ \exists j\ (Ci \overset{Ti}{=>} Di,j); \quad i=1,..,n;\ j=1,..,ki$$

Para tanto, o usuário deverá provar inicialmente que:

$$(Ci)^{*}_{a} \Rightarrow (Xi)^{\omega}_{b}$$

ou seja, que a asserção associada a ocorrência da exceção Ei implica na asserção inicial do tratador correspondente. A seguir, deve provar que a execução do tratador, sob condições iniciais Xi, ou termina em um ponto Yil tal que

$$Xi \; \{Ti\} \; Yi1 \vee Yi2 \vee \ldots \vee Yim_i; \quad (m_i \leq k_i)$$

$$\text{tal que:} \quad (Yil)^{\omega}_{b} \Rightarrow (Dij_{\ell})^{*}_{a}$$

$$\text{onde} \quad j_{\ell} \in \{ 1, 2, \ldots, k_i \}$$

sendo que a associação entre Yil e Dij é determinada pela semântica do comando <u>return</u> <u>to</u> (vide regra 37).

Ou, em outras palavras, o usuário deverá provar que as asserções finais de Ti implicam nas asserções associadas aos correspondentes pontos de retorno por ele selecionados, ou termina em aborto, caso em que deve ser provado que

$$Xi \; \{ \; Ti \; \} \; V$$

onde V é a "strongest post-condition" associada ao ponto de retorno do módulo que causou Ei.

7.1.1 - Exemplo

Suponha um módulo que, a partir de um vetor de dados X de n+1 elementos, obtêm um outro vetor Y cujos n+1 elementos, inteiros, indicam a ordem em que os elementos de X deveriam estar para que este vetor tivesse seus elementos em ordem crescente. É natural imaginarmos a situação em que o módulo encontra dois elementos de X cujos valores sejam iguais. Nestes casos, como o módulo não possue informações suficientes para dirimir a ordem na qual estes elementos devem aparecer, são causadas exceções para que o usuário da abstração decida a ordem dos referidos elementos.

A especificação do módulo é dada por:

```
module sort (X, Y: array [0..n] of integer;
             n: integer) is
  entry: n ≥ 0 & ∀i (0≤i≤n) Y[i] = i;
  exit : perm (Y, Y') & ∀ i (0≤i≤n) X[i] = X'[i] &
         ∀i (0≤i<n) X [Y' [i]] ≤ X [Y' [i+1]];
  exceptions:
         ∀i,j (0≤i≤n  &  0≤j≤n  &  i≠j)
         X [Y [i]] = X [Y [j]] raises tie (i,j);
         R: i = i'  &  j = j'  &
            (Y [i] = Y' [i]  &  Y [j] = Y' [j]) or
            (Y [i] = Y' [j]  &  Y [j] = Y' [i])
  end of exceptions
end sort.
```

A asserção inicial de sort especifica que os valores do

vetor Y sejam 0,1,...,n, além do que o valor de n não deve ser negativo.

A asserção final afirma que após a execução de sort, o vetor Y (denotado por Y') é uma permutação do mesmo vetor Y inicial, e que este mesmo Y indica a ordem dos elementos de X. O vetor X não tem seus elementos alterados.

Além disso, a especificação do módulo indica que, para todos os pares de valores de X que sejam iguais entre sí, será causada uma exceção indicando, através de Y, quais os valores de X que são iguais. A asserção associada ao ponto de retorno indica que, ou os valores Y [i] e Y[j] permanecem inalterados, ou que foram permutados. Ou seja, o usuário tem inteiro arbítrio para decidir em casos de empates.

Suponha, agora, o trecho de programa abaixo, que usa o módulo sort:

```
...
s := 50;
for i:=0 to s do
   begin read ( A [i] ); IND [i] := i end
   { s≥0  &  ∀i  (0≤i≤s) IND [i] = i }
   call sort (A, IND, s);
     [on tie (k, l) =>
         { (0≤k≤s) & (0≤l≤s) & (k≠l) & A [IND [k]] = A [IND [l]] }
         h: begin
               if V [IND [k]] ≤ V [IND [l]]
               { k = k'  &  l = l'  &
                 IND [k] = IND' [k] & IND [l] = IND' [l] }
                 then return to R
                 else begin
                          aux := IND [k];
                          IND [k] := IND [l];
                          IND [l] := aux;
               { k = k'  &  l = l'  &
                 IND [k] = IND' [l] & IND [l] = IND' [k] }
                          return to R
                      end
            end h]
         { perm (IND, IND') & ∀k (0≤k≤s)
          X [IND [k]] ≤ X [IND [k+1]] &
          ∀i (0≤i≤s) A [i] = A' [i] }
...
```

Mostraremos, a seguir, o processo de verificação do exemplo acima. Assumiremos a correção do módulo sort, isto é, não mostraremos como este módulo foi verificado. A referida prova pode ser encontrada em [Manna 74], de onde o exemplo foi inspirado.

O que deve ser provado, para completar o processo de verificação do módulo sort é o seguinte:

```
a) (∀i,j (0≤i≤n & 0≤j≤n & i≠j &
    A [IND [i]] = A [IND [j]]) =>
   (0≤i≤s & 0≤j≤s & i≠j &
    A [IND [i]] = A [IND [j]])
```

```
b) (0≤k≤s & 0≤l≤s & k≠l &
    A [IND [k]] = A [IND [l]]) { h(k, l) }
   (k=k' & l=l' &
    ((IND [k] = IND' [k] & IND [l] = IND' [l]) or
     (IND [k] = IND' [l] & IND [l] = IND' [k]))]

c) (i=i' & j=j' &
    ((IND [i] = IND' [i] & IND [j] = IND' [j]) or
     (IND [i] = IND' [j] & IND [j] = IND' [i]))] =>
   (i=i' & j=j' &
    ((IND [i] = IND' [i] & IND [j] = IND' [j]) or
     (IND [i] = IND' [j] & IND [j] = IND' [i]))]
```

As fórmulas acima podem ser facilmente provadas. No caso (a), basta usar a regra "dictum de omni" [Rasiowa 73] para provar a implicação.

A fórmula (b) pode ser provada, fazendo-se as substituições correspondentes às atribuições feitas pelo tratador, no termo consequente.

A prova da última fórmula também é trivial. Basta constatar a semelhançados termos.

7.2 - Encapsulamento.

O principal argumento que permite comprovar que o mecanismo proposto assegura meios de manter o encapsulamento, está no fato de que o usuário de um módulo, para desenvolver os tratadores externos que o módulo requer, e também para completar o processo de verificação do mesmo módulo, se vale apenas dos elementos que constam da especificação do referido módulo. Uma amostra deste fato foi dada no exemplo anteriormente apresentado. Naquele exemplo, propositadamente evitamos mostrar a implementação do módulo sort, o que não impediu que se construisse o tratador apropriado para a exceção tie e se completasse o processo de verificação do módulo.

7.3 - Construtibilidade

O mesmo argumento usado acima pode ser utilizado para mostrar que o uso do mecanismo não interfere na propriedade de construtibilidade do módulo, isto é, não torna o módulo especializado ao ponto de impedir que ele possa ser usado em outros ambientes. Pelo contrário, pensamos que o uso do mecanismo, especialmente no caso de tratamento externo aumenta a propriedade de construtibilidade, uma vez que a possibilidade de prover um tratador externo torna o módulo mais universal, ou seja, aumenta a sua faixa de aplicabilidade, o que não ocorreria, caso o tratamento da exceção fosse pré-fixado pelo projetista. Por exemplo, no caso do módulo sort a possibilidade do usuário decidir em casos de empate aumenta a gama de aplicações do módulo. Isto não ocorreria caso o critério de desempate fosse pré-estabelecido.

É claro que outras propostas para tratamento de exceções também possuem esta característica, mas não simultaneamente com as características de verificação e encapsulamento.

7.4 - Acoplamento.

O requisito formulado para o caso de acoplamento era no sentido de que o acréscimo de informações trocadas entre os módulos sinalizador e sinalizado não aumentassem de forma significativa o acoplamento já existente entre os módulos. Infelizmente a solução adotada na proposta implica no aumento da ligação entre os módulos, tanto no que diz respeito a controle como no que diz respeito a dados (variáveis). As informações de controle foram necessárias, uma vez que o mecanismo é essencialmente uma estrutura de controle. O acréscimo em troca de dados é permitido (não é obrigatório), no caso em que o módulo sinalizador deseje transmitir parâmetros para o tratador. Esta possibilidade foi incluida no mecanismo com o objetivo de permitir que a sinalização de uma exceção não se resumisse ao sinal de exceção em sí, mas que este sinal pudesse ser enriquecido com parâmetros, com o fim de especializá-lo.

Acreditamos que as vantagens resultantes do aumento do acoplamento compensem largamente as desvantagens daí resultantes.

7.5 - Adequação ao Ambiente.

O requisito de adequação é facilmente comprovável, uma vez que as construções existentes no mecanismo foram moldadas tendo em vista este requisito.

Como vimos no capítulo 4, o requisito de adequação ao ambiente foi colocado da seguinte forma:

1) permitir que o tratamento das exceções tanto pudesse ser interno como externo;

2) a associação entre exceção e tratador fosse feita por meio de operações (ativações de módulos);

3) permitir que o usuário optasse pela forma de retomada do processamento normal.

Como se vê, todos estes requisitos exploram a estruturação hierárquica que caracteriza um ambiente modular.

O requisito (1) é atendido, pois vimos, no capítulo 6 deste trabalho, que o usuário tem como excercer esta opção, sendo que o critério de escolha que o mecanismo adota para selecionar o tratador é dado pela fórmula:

$$(C_i \in P_e \Rightarrow \Lambda\ (E_i)),\ ((C_i \in P_i \Rightarrow \Lambda I\ (E_i))$$

No que se refere a associação entre exceção e tratador (ítem (2)), o mecanismo obriga que ela seja uma associação forte (veja definição do termo no capítulo 4), isto é, que a

localização do tratador seja tal que se torne possível identificar, de forma unívoca, qual a ativação do módulo (operação) que causou a exceção.

Quanto a opção de retomada (ítem (3)), o mecanismo não se fixa em um determinado modelo (termination, resumption), nem tampouco obriga que o projetista do módulo indique a forma de retomada de cada exceção. O usuário do módulo tem inteira liberdade de optar, de acordo com a conveniência do caso, pela forma de retomada.

É claro que estas flexibilidades dadas ao mecanismo o debilitam no sentido de que as asserções que podem ser provadas verdadeiras se tornam mais fracas.

7.6 - Outros Exemplos.

A seguir mostraremos outros exemplos do uso do mecanismo proposto neste trabalho. Para cada exemplo nos limitaremos ao enunciado do problema e ao programa em sí. Não mostraremos a correção dos módulos envolvidos.

Exemplo 1.

Suponha os módulos abaixo:

```
module gauss-seidel (A: array [1..n] of
                        array [1..n] of real;
                     B, X: array [1..n] of real is
                     ...

module jacobi (A: array [1..n] of
                  array [1..n] of real;
               B, X: array [1..n] of real is
               ...
```

que implementam a solução de sistemas lineares A X = B pelos métodos de Gauss-Seidel e Jacobi, respectivamente. Ambos os módulos podem causar as exceções:

singular e

não-converge

que indicam, respectivamente, que a matriz A de coeficientes é singular e que o método não converge para o sistema dado [Albrecht 73].

O trecho de módulo abaixo mostra o mecanismo sendo usado para selecionar o método de solução de um sistema linear A X = B.

```
...

call gauss-seidel (A, B, X);
  [on singular =>
   sing: begin
          raise singular;
           R: raise failure ('sistema singular',
                             'solução', nil)
         end
   end sing]

  [on não-converge =>
   gs: abort end gs]

 if aborted (gauss-seidel)
   then begin
          call jacobi (A, B, X);
           [on não-converge =>
            jac: abort end jac]

           if aborted (jacobi)
             then raise failure ('sistema sem
                                 solução por Gauss-Seidel
                                 e Jacobi', 'solução', nil)
        end

 print (A, B, X);

...
```

Observação: a função aborted (...) corresponde ao predicado K da fórmula (39) do capítulo 6.

Exemplo 2

Este segundo exemplo mostra o mecanismo sendo usado para controlar o andamento de um processo iterativo.

Suponha um módulo itera que implementa a solução de um problema através de um método iterativo. Este módulo provoca uma exceção iteração (erro) a cada iteração completada, onde erro é uma variável que indica a diferença entre as soluções

encontradas nas duas últimas iterações (ou entre o valor inicial do processo e a primeira iteração). Este módulo possue um tratador interno para essa exceção, o qual encerra a execução do módulo se erro ≤ delta, ou então causa a exceção não-converge (n), caso não for satisfeita a condição erro ≤ delta quando já tiverem ocorridas n iterações.

```
...
{tratador interno}
[on iteração (erro) =>
 it: if erro ≤ delta
       then return
       else begin
              itno := itno + 1;
              if itno ≥ n
                then raise não-converge (n)
                        R: {return actions for this exception};
                else continue
            end
 end it]
...
while T do
  begin
    {compute uma iteração}
    raise iteração (erro)
  R: end
end.
```

No caso deste exemplo, o usuário do módulo itera poderá prover ou não um tratador para a exceção iteração. Caso ele opte por um tratador próprio (como no caso abaixo), este será

executado sempre que ocorrer a referida exceção.

```
...

itno := 0; minerro := maxfloat;
call itera (...);
  [on iteração (erro) =>
   it: if erro ≤ delta
           then abort
           else if erro ≤ minerro
                   then begin
                          itno := itno + 1;
                          minerro := erro;
                          continue
                        end
                   else raise diverge (erro, minerro, itno)
                          R: {return actions for this exception}
   end it]
...
```

Exemplo 3.

Este exemplo mostra o caso de uma exceção exógena.

Suponha uma aplicação de "computer graphics" composta por um conjunto de módulos para construir figuras e projetá-las em um terminal CRT. Dentre os módulos que fazem manipulação de imagens está o módulo rotate (axis, period, angle), que produz uma rotação de um ângulo angle em torno do eixo axis a cada período period da figura que está sendo projetada na tela. Este módulo pode ser interrompido por tres exceções exógenas. A primeira, newparms, provoca a troca dos parâmetros do módulo rotate; a segunda, wait, provoca uma parada da execução do módulo (e consequentemente da rotação da figura) por um determinado espaço de tempo, enquanto que a

terceira, stop, provoca o fim da execução do módulo. A seguir mostramos os trechos sensíveis a estas exceções do referido módulo.

```
rotate (axis: line; period: seconds; angle: grades) is
...
repeat begin
          {compute novas coordenadas da figura}
          {projete novos pontos na tela}
          when newparms (newaxis, newperiod, newangle) =>
            [new: begin
                     axis := newaxis;
                     period := newperiod;
                     angle := newangle
                  end
             end new]

          when wait (time) =>
            [w: wait (time) end w]

          when stop =>
            [s: return end s]
       end
until F
end rotate.
```

UF
INSTITUTO D... [illegible] RMÁTICA
BIBLIOTECA

Capítulo 8 - Conclusões e Sugestões.

Foi apresentado, neste trabalho, um estudo sobre o tratamento de exceções sendo, ainda, proposto um mecanismo para o tratamento de exceções em ambientes modulares.

Apresentamos, inicialmente, um estudo sobre os diferentes tipos de exceções que normalmente ocorrem em programação, sendo proposta, a seguir, uma classificação destas exceções segundo critérios de natureza, origem e causa. Como mencionamos anteriormente, o objetivo principal da classificação foi no sentido de fornecer uma orientação ao programador, no que diz respeito a exceções. Entendemos ser necessários maiores estudos sobre o assunto, principalmente pelo fato de que a classificação que propomos representa a primeira tentativa nesse sentido.

A discussão sobre a definição de módulo, juntamente com a sugestão de definição apresentada representa, ao nosso ver, um novo enfoque sobre o assunto, uma vez que procuramos desvincular a noção de módulo da noção de implementação de módulo, ao contrário da maioria dos autores que normalmente confundem os dois conceitos. Uma conceituação similar a apresentada neste trabalho é encontrada em [Dennis 73].

Quanto aos requisitos formulados, procuramos nos restringir àqueles que dizem respeito especificamente ao objetivo da proposta, ou seja, os requisitos necessários para

permitir o tratamento de exceções em programas modulares, além, é claro, do requisito de verificabilidade. Quanto a este último requisito, acreditamos representar uma real contribuição, uma vez que a solução adotada para a verifiva-ção de módulos representa uma inovação na forma de conduzir o processo de prova. Normalmente, como foi caracterizado no capítulo 5, os métodos de verificação adotados pelos demais autores exigem o conhecimento do tratador para provar a correção do módulo. A metodologia de verificação que propomos não faz tal exigência, o que assegura, pois, a satisfação de requisitos como construtibilidade e encapsulamento, que de outra forma não poderiam ser satisfeitos.

Por outro lado, a desvantagem produzida pelo método de verificação proposto é no sentido de que ela produz provas mais "brandas", e que devem ser completadas pelo usuário do módulo. Acreditamos, no entanto, que esta desvantagem seja plenamente compensada pelas vantagens que o método introduz.

Quanto ao mecanismo em sí, pensamos que possa represen-tar uma boa contribuição à solução do problema de tratamento de exceções pelos seguintes motivos:

a) permite o desenvolvimento de programas com tratamento de exceções em etapas distintas. Na primeira o projetista se preocupa apenas com as condições normais do programa. Na segunda ele insere o código necessário para o tratamento das exceções.

Embora não tenhamos desenvolvido experiências nesse sentido, pensamos que este método de desenvolvimento de programas possa facilitar e simplificar a tarefa de programar, tornando-os, inclusive, mais legíveis e manuteníveis.

b) O mecanismo permite aumentar a confiabilidade dos programas, uma vez que a ocorrência de erros não previstos pode ser detectada por meio da construção "failure", juntamente com informações que permitem sua identificação. Esta característica é fundamental em aplicações em tempo real que não podem sofrer interrupções desastrosas, o que poderia ocorrer caso a ocorrência de erros provocasse o colapso do sistema. É claro que o mecanismo em sí não possue esta faculdade. Ele apenas fornece os recursos que permitem dar a característica de confiabilidade.

c) A prioridade dada ao tratador externo (quando existe interno para a mesma exceção) introduz um novo conceito, no sentido de que o projetista de um módulo pode projetar "default actions" para cada exceção. O usuário pode, por sua vez, examinando a especificação do módulo, optar pelo uso do tratador provido pelo projetista ou, então, projetar o seu próprio, de acordo com os seus interesses particulares ao caso. O usuário excerce esta opção apenas listando, após o ponto de

ativação do módulo, o tratador por ele definido, usando para isso a sintaxe própria.

Normalmente os mecanismos que permitem que o tratamento de uma exceção seja interno ou externo (p. ex. [Ichbiah 79]) dão prioridade de tratamento ao tratador interno. Efeito similar pode ser obtido pelo mecanismo aqui proposto, se a exceção for declarada "invisível", o que também representa um novo conceito ([Ichbiah 79] permite inibir a propagação de uma exceção). Uma exceção invisível pode ser usada com duas finalidades:

1 - obrigar que o tratamento seja interno;

2 - executar ações preliminares à sinalização de outra exceção, visível ao usuário, para produzir um efeito similar ao que [Ichbiah 79] denomina de "last wishes".

d) O mecanismo não obriga o projetista de um módulo a especificar a forma de retomada do processamento do módulo, como por exemplo em [Goodenough 75]. A decisão de como retomar o processamento é de inteiro arbítrio do usuário, embora em alguns casos ele tenha apenas uma opção de retomada, como por exemplo, no caso de uma exceção do tipo <u>eof</u>, cuja única alternativa é encerrar a execução do módulo de leitura.

Esta facilidade de escolha dada ao usuário é fruto do fato de que o mecanismo não impõe um modelo fixo de retomada ("resumption" ou "termination"), como por exemplo em [Horning et al. 74], [Ichbiah 79], [Levin 77], [Liskov, Snyder 79], etc.

e) O mecanismo permite desenvolver programas verificáveis de tal forma que ficam asseguradas as características de construtibilidade e encapsulamento.

Queremos registrar, finalmente, que as idéias formuladas neste trabalho foram testadas, com sucesso, no desenvolvimento de um sistema de gerência de entrada de dados para um mini-computador. O sistema foi projetado para permitir o controle de até 32 terminais, cada um deles executando um programa diferente de aquisição e crítica de dados.

<u>Sugestões para Pesquisa</u>.

Procuramos, ao longo deste trabalho, cobrir, na medida do possível, os quesitos propostos para a sua elaboração. Isto não significa, no entanto, que o assunto tenha se esgotado. Esperamos, entretanto, ter contribuido de forma significativa para o estudo do tema, sendo, no entanto, complementar este estudo em alguns aspectos, alguns dos quais listamos a seguir.

a) Estudo do problema de detecção de exceções. Este estudo deve considerar dois aspectos importantes e distintos:

a semântica de condições de exceção e a eficiência do processo. O primeiro diz respeito ao significado de uma condição de exceção. Nesse sentido, devem ser conduzidos estudos que permitam criarem-se meios do usuário especificar as condições que ele deseja caracterizar como sendo de exceção. Quanto à eficiência, o mecanismo de detecção deve ser tal que não provoque "overhead" no processamento, enquanto não ocorrerem as condições de exceção. Talvez recursos de "hardware", juntamente com o conceito de interrupção possam colaborar neste sentido.

b) Estudo do problema de tratamento de exceções em programação não-sequencial (co-rotinas, tasks, etc), com o objetivo de identificar formas de propagação e os locais de tratamento mais adequados.

c) Otimização do processo de implementação do mecanismo proposto, no sentido de evitar que o tempo de execução de um módulo seja influenciado pelo mecanismo, enquanto não ocorrer nenhuma exceção. Estamos convictos de que o uso de recursos de "hardware" (ou de microprograma) no processo de "binding" dos módulos pode ser de grande valia nesse sentido. Também recursos dessa natureza podem ajudar na implementação de exceções exógenas.

d) Desenvolver um estudo sistemático sobre a classificação proposta no capítulo 2, a fim de aperfeiçoá-la e

assegurar sua completeza. É possível que a identificação de novos tipos de exceções venham sugerir aperfeiçoamentos no mecanismo.

e) Melhorar a legibilidade dos programas, através da remoção dos tratadores de exceção do texto do programa. A liberação da obrigatoriedade de que o tratador esteja textualmente contido no ambiente de ativação do módulo sinalizador permitiria, inclusive, que os tratadores pudessem ser manipulados como módulos ordinários, os quais poderiam possuir as características de construtibilidade, encapsulamento, etc.

f) Desenvolver estudos no sentido de criar uma metodologia de programação baseada no uso de tratamento de exceções, que permitisse desenvolver programas corretos por construção. O método de verificação proposto neste trabalho já é orientado neste sentido. Também a sugestão formulada no ítem (a) deste capítulo contribue para este fim.

g) O aperfeiçoamento da parte do mecanismo para o tratamento de exceções exógenas, para permitir, também, o seu uso no caso de exceções (interrupções) que exigem tratamento imediato.

A razão da existência deste tipo de exceção pode ser substanciada nos seguintes casos:

1 - interrupções de proteção:

a) falta de energia;

b) multiprogramação.

2 - interrupção de acidente:

a) erro de "hardware/software";

b) erro de operação.

3 - interrupção de suspensão:

a) fim de tempo disponível;

b) "interrompa imediato".

A ocorrência de uma exceção exógena de tratamento imediato, ao contrário da de atendimento em pontos sensíveis, deve provocar a interrupção imediata da execução do módulo sinalizado. A exceção somente será sentida pelo módulo se esta não estiver "mascarada". O tratador da exceção poderá determinar o aborto do módulo em questão ou a retomada da sua execução.

Sugerimos as seguintes regras de sintaxe para as construções acima citadas:

<u>whenever</u> X - identificação do tratador da exceção X.

<u>mask</u> X - inibe a sinalização de X, até que

<u>unmask</u> X - libera a ocorrência de X.

O motivo pelo qual deixamos este ítem como sugestão reside no fato de que é necessário desenvolver grande esforço de pesquisa para desenvolver regras de prova para tal mecanismo.

Uma solução simples seria impedir qualquer tipo de acesso do tratador aos objetos do módulo sinalizado, de maneira que a ocorrência de uma exceção de atendimento imediato fosse transparente ao módulo. Neste caso, o tratador não teria como interferir na execução do módulo, não interferindo, pois, na sua correção. Por outro lado esta ferramenta teria um poder de expressão muito reduzido, não podendo ser usada, por exemplo, para implementar um sistema operacional.

Podemos, no entanto, imaginar que este mecanismo seja o caso limite do mecanismo proposto para exceções exógenas de tratamento em pontos sensíveis, onde os pontos sensíveis são todos os intervalos entre unidades sintáticas do módulo. É claro que, neste caso, o processo de verificação do módulo se tornaria extremamente longo.

Enfim, pensamos que pesquisas nesta direção proposta possam vir a contribuir para resolver de forma satisfatória o problema de exceções exógenas de tratamento imediato.

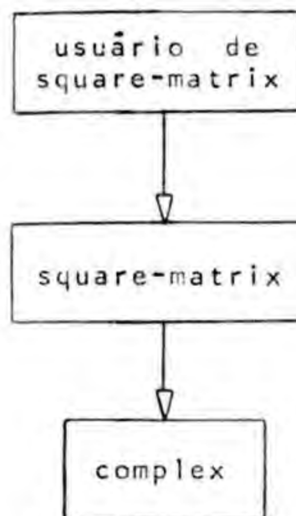
h) Ainda outro aspecto que deve ser pesquisado é o que diz respeito a ocorrência de exceções endógenas em tipos de dados parametrizados [von Staa 74]. Um tipo de dado parametrizado é aquele em que o tipo de um ou mais parâmetros é, por sua vez, um parâmetro, como no exemplo abaixo, onde é usada a linguagem CLU [Liskov, Zilles 74]:

```
square-matrix: cluster (element-type: type,
                        side-length: integer) is
  put-element, get-element, add, sub, ...;

  rep(type-param: type, side: integer) =
     (e-type: type; mat: array [1..side] of
                         array [1..side] of type-param);

  create
     m: rep(element-type, side-length);
     m.e-type := element-type;
     return m;
  end

  get-element: ... end
  put-element: ... end
  add: ........... end
  sub: ........... end
    .
    .
    .
end square-matrix.
```

Um objeto do tipo square-matrix pode ser criado, por exemplo, como abaixo:

```
mat1: square-matrix (complex, n);
```

que "cria" uma matriz n x n com elementos do tipo "complex" e que pode ser manipulada pelas operações definidas no cluster. Neste caso, teremos a seguinte organização:



Supondo que ocorra uma exceção no cluster "complex", este reportará a sua ocorrência ao cluster "square-matrix". Esta situação não é a ideal pois, neste caso, ao contrário do que foi estabelecido anteriormente, o módulo que possue elementos que permitem tratar a exceção não é o superior imediato mas sim o superior deste.

Neste aspecto, o mecanismo deve ser aperfeiçoado para permitir que a exceção trafegue diretamente do cluster "complex" ao módulo do usuário, uma vez que o cluster "square-matrix" não possue nenhum conhecimento das exceções que os clusters que implementam o

"element-type" possam sinalizar. Isto significa que a proposta de Goodenough [Goodenough 75c] para esta situação não é apropriada, pois na sua proposta é necessário que o módulo intermediário contenha as previsões das exceções que ele deve retransmitir, o que torna a proposta inviável para este fim.

Pensamos, finalmente, que as sugestões acima formuladas venham contribuir para o aperfeiçoamento do mecanismo e para o desenvolvimento da área.

Bibliografia

[Alagič, Arbib 78]

Alagič, Suad and Arbib Michael A.: "The Design of Well-Structured and Correct Programs". Springer-Verlag, New York, (1978).

[Albrecht 73]

Albrecht, Peter: "Análise Numérica - Um Curso Moderno". Livros Técnicos e Científicos Editora S.A., Rio de Janeiro, (1973).

[Anderson, Kerr 76]

Anderson, T. and Kerr, R.: "Recovery Blocks in Action: a System Supporting High Reliability". Technical Report #93. Computing Laboratory, University of New Castle upon Tyne, (july 1976).

[Arbib, Alagič 79]

Arbib, Michael A. nd Alagič, Suad : "Proof Rules for goto's". Acta Informatica 11, (1979), pp. 139-148.

[Avizienis 77]

Avizienis, Algirdas: "Computer Reliability and Fault-Tolerant Computers". Technical Report. Computer Science Department, University of California, (july 1977).

[Back 80]

Back, Ralph-Johan : "Exception Handling With Multi-Exit Statements". in Informatik-Fachberichte. Band 25: Programmiersprachen und Programmentwicklung. 6. Fachtagung, Darmstadt, 1980. Herausgegeben von H.-J. Hoffmann, Springer-Verlag, Berlin, (1980), pp. 71-82.

[Bauer 71]

Bauer, Friedrich L.: "Software Engineering". Proceedings of the IFIP Congress' 71 C.V. Freiman Ed., (1972), pp. 530-538.

[Berry, Kemmerer, von Staa, Yemini 80]

Berry, D.M., Kemmerer, R.A., von Staa, Arndt, and Yemini, S.: "Toward Modular Verifiable Exception Handling". Journal of Computer Languages, Vol 5, (1980), pp. 77-101.

[Black 80]

Black, Andrew P.: "Exception Handling and Data Abstraction". Computer Sciences Department. IBM Thomas J. Watson Research Center. RC 8059. New York, (january 1980).

[Boehm 76]

Boehm, Berry W.: "Software Engineering". IEEE Transactions on Computers, Vol. C-25, No. 12, (december 1976), pp.1126-1241.

[Chaudhary, Sahasrabuddhe 78]

Chaudhary, B.D. and Sahasrabuddhe, H.V.: "Suggestions about a Specification Technique". Sigplan Notices, Vol. 13, No. 12, (december 1978), pp. 25-28.

[Dennis 73]

Dennis, Jack B.: "Modularity", in Advanced Course on Software Engineering. Springer-Verlag, Berlin, (1973).

[Dijkstra 72]

Dijkstra, Edsger W.: "Notes on Structured Programming". in Structured Programming, Academic Press, New York, (1972).

[Dijkstra 75]

Dijkstra, Edsger W.: "Guarded Commands, Nondeterminacy and Formal Derivation of Programs". Communications of the ACM, Vol. 18, No. 8, (august 1975), pp. 453-457.

[Fregni, Ogus 74]

Fregni, Edson and Ogus, Roy C.: "Error Recovery Techniques in Computer Systems: a survey". Technical Note No. 42. Digital Systems Laboratory, Stanford University, (june 1974).

[Goodenough 75a]

Goodenough, John B.: "Structured Exception Handling". Conf. Rec., Second ACM Symposium on Principles of

Programming Languages. Palo Alto, California, (january 1975), pp. 204-224.

[Goodenough 75b]

Goodenough, John B.: "Exception Handling Design Issues". Sigplan Notices, Vol. 10, No. 7, (july 1975), pp. 41-45.

[Goodenough 75c]

Goodenough, John B.: "Exception Handling: Issues and a Proposed Notation". Communications of the ACM, Vol. 18, No. 12, (december 1975), pp. 682-696.

[Goos, Kastens 77]

Goos, G. and Kastens, U.: "Programming Languages and the Design of Modular Programs". Proceedings of the IFIP Working Conference on Constructing Quality Software, Novosibirsk, USSR, (may 1977), pp. 153-191.

[Guttag 76]

Guttag, J.V. et al.: "The Design of Data Type Specifications". Current Trends in Programming Methodology, Vol. IV, Raymond Yeh (Ed.), Prentice Hall, Englewood Cliffs, New Jersey, (1976), pp. 60-79.

[Hamlet 76]

Hamlet, R.G.: "High-level Binding with Low-level Linkers". Communications of the ACM, Vol. 19, No. 11, (november 1976), pp. 642-644.

UFRGS
INSTITUTO DE INFORMÁTICA

[Hoare 69]

Hoare, C.A.R.: "An Axiomatic Basis for Computer Programming". Communications of the ACM, Vol.12, No. 10, (october 1969), pp. 576-583.

[Hoare, Wirth 73]

Hoare, C.A.R. and Wirth, N.: "An Axiomatic Definition of the Programming Language Pascal". Acta Informatica 2, (1973), pp. 335-355.

[Horning, Lauer, Melliar-Smith, Randell 74]

Horning, J.J., Lauer, H.C., Melliar-Smith, P.M., and Randell, B.: "A Program Structure for Error Detection and Recovery". in Operating Systems, Gelenbe and Kaiser (Ed.), Springer-Verlag, New York, (1974), pp. 171-187.

[Ibm]

PL/I(F) Language Reference Manual, Form GC28-8201, IBM Corporation.

[Ichbiah 79]

Ichbiah, Jean D.: "Rationale for the Design of the ADA Programming Language". Sigplan Notices, Vol. 14, No. 6, (june 1979), part B.

[Ironman 77]

Department of Defense Requirements for High Order Computer Programming Languages. Revised "Ironman", july

1977. The Technical Requirements. Sigplan Notices, Vol. 12, No. 12, (december 1977), pp. 39-54.

[Levin 77]

Levin, Roy: "Programs Structures for Exception Handling". Ph.D. Thesis, Department of Computer Science, Carnegie-Mellon University, (1977).

[Liskov, Snyder 79]

Liskov, Barbara H., and Snyder, Alan: "Exception Handling in CLU". IEEE Transactions on Software Engineering, Vol. SE5, No. 6, (november 1979), pp. 546-558.

[Lucena, von Staa 78]

Lucena, C.J.P. e von Staa, Arndt: "Projeto de Programas - Aplicaçpões de Programação Estruturada em Cobol". Cadernos de Programação 1 - Serpro, Seção de Publicações Técnicas, Rio de Janeiro, (1978).

[Luckham, Pollack 80]

Luckham, D.C. and Pollack, W.: "ADA Exception Handling: An Axiomatic Approach". ACM Transactions on Programming Languages and Systems, Vol. 2, No. 2,(april 1980), pp. 225-233.

[Mac Laren 77]

Mac Laren, M. Donald: "Exception Handling in PL/I". Proceedings of the ACM Conference on Languages Design for Reliable Software. New Castle, (march 1977).

[Madnick, Donovan 74]

Madnick, Stuart E., and Donovan, John J.: "Operating Systems". Mc Graw Hill, New York, (1974).

[Manna 74]

Manna, Zohar : "Mathematical Theory of Computation". Mc Graw Hill, New York, (1974).

[Mc Glynn 76]

Mc Glynn, Daniel R.: "Microprocessors, Technology, Architecture, and Applications". John Willey, New York, (1976).

[Mc Gowan, Kelly 75]

Mc Gowan, C., and Kelly, J.: "Top-Down Structured Programming Techniques", Petrocelli/Charter, (1975).

[Melliar-Smith 77]

Melliar-Smith, P.M.: "Software Reliability: the role of Programmed Exception Handling". Proceedings of the ACM Conference on Languages Design for Reliable Software. New Castle, (march 1977).

[Mullery 75a]

Mullery, Alvin P.: "Error Tolerant Software: Some Principles". Computer Sciences Department. IBM Thomas J. Watson Research Center, New York, (may 1975).

[Mullery 75b]

Mullery, Alvin P.: "A Design for Error Tolerant Software". Computer Sciences Department. IBM Thomas J. Watson Research Center, New York, (june 1975).

[Myers 76]

Myers, Glenford J.: "Software Reliability Principles and Practices". John Willey, New York, (1976).

[Myers 78]

Myers, Glenford J.: "Composite/Structured Design". Van Norstrand Reinhold Company, New York, (1978).

[Newton 79a]

Newton, Roy: "On Exception Handling with the "Ironman" Languages, GREEN and REDL". Sigplan Notices, Vol. 14, No. 4, (april 1979), pp. 41-54.

[Newton 79b]

Newton, Roy: "Some Exception Handling Problems in Language Systems Displaying a Multi-Path Capability". Sigplan Notices, Vol. 14, No. 4, (april 1979), pp. 55-63.

[Parnas 72a]

Parnas, D.L.: "A Technique for the Specification of Software Modules with Examples". Communicataions of the ACM, Vol. 15, No. 5, (may 1972), pp. 330-336.

[Parnas 72b]

Parnas, D.L.: "Response to Detected Erros in Well-Structured Programs". Report CS205, Computer Science Department, Carnegie-Mellon University, (july 1972).

[Parnas 72c]

Parnas, D.L.: "On the Criteria to be Used in Decomposing Systems into Modules". Communications of the ACM, Vol. 15, No. 12, (december 1972), pp. 1053-1058.

[Popek, Horning, Lampson, Mitchell, London 77]

Popek, G.J., Horning, J.J., Lampson, B.W., Mitchell, J.G., and London, R.L.: "Notes on the Design of Euclid". Proceedings of the ACM Conference on Language Design for Reliable Software". New Castle, (march 1977).

[Randell 75]

Randell, B.: "System Structure for Software Fault-Tolerance". IEEE Transactions on Software Engineering, Vol. SE1, No. 6, (june 1975), pp. 220-232.

[Rasiowa 73]

Rasiowa, Helena : "Introduction to Modern Mathematics". North-Holland Publishing Company, Amsterdam, (1973).

[Suppes 72]

Suppes, Patrick: "Axiomatic Set Theory". Dover Publications Inc, New York, (1972).

[Tanenbaum 76]

Tanenbaum, Andrew S.: "Structured Computer Organization". Prentice Hall Inc., Englewood Cliffs, New Jersey, (1976).

[von Staa 74]

von Staa, Arndt: "Data Transmission and Modularity Aspects of Programming Languages". Ph.D. Thesis. University of Waterloo, (1974).

[Wasserman 77]

Wasserman, A.I.: "Procedure-Oriented Exception Handling". University of California, Laboratory of Medical Science, Report, (1977).

[Wegner 79]

Wegner, Peter : "Programming with ADA: An Introduction by Means of Graduate Examples". Sigplan Notices, Vol. 14, No. 12, (december 1979), pp. 1-46.

[Weinstock 73]

Weinstock, Charles B.: "A Survey of Protection Systems". Computer Science Department. Carnegie-Mellon University, (july 1973).

[Whitaker 78]

Whitaker, William A. Lt. Col., Usaf: "The US Department of Defense Common High Order Language Effort". Sigplan Notices, Vol. 13, No. 2, (february 1978), pp. 19-29.

[Wirth 73]

Wirth, Niklaus: "Systematic Programming - An Introduction". Prentice Hall, New York, (1973).
Também em Língua Portuguesa:
Wirth, Niklaus: "Programação Sistemática". Tradução de Paulo Augusto da Silva Veloso, Editora Campus, Rio de Janeiro, (1978).

[Wulf 74]

Wulf, William A.: "Alphard: Toward a Language to Support Structured Programs". Department of Computer Science, Carnegie-Mellon University, (1974).

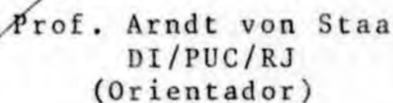
[Wulf 75]

Wulf, William A.: "Reliable Hardware-Software Architecture". Proceedings of the International Conference on Reliable Software, (april 1975), pp. 122-130.

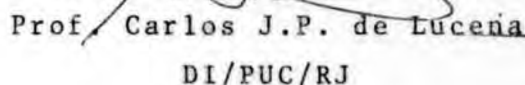
[Yourdon, Constantine 79]

Yourdon, E., and Constantine, L.L.: "Structured Design". Prentice Hall, Englewood Cliffs, New Jersey, (1979).

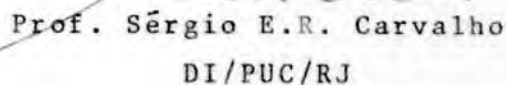
Tese apresentada ao Departamento de Informática da PUC/RJ, e aprovada pela Comissão Julgadora, formada pelos seguintes professores:



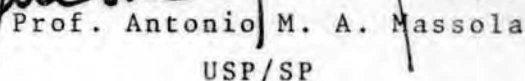
Prof. Arndt von Staa
DI/PUC/RJ
(Orientador)



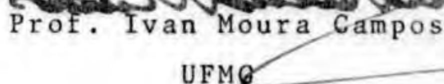
Prof. Carlos J.P. de Lucena
DI/PUC/RJ



Prof. Sérgio E.R. Carvalho
DI/PUC/RJ

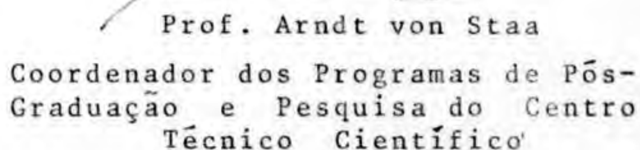


Prof. Antonio M. A. Massola
USP/SP



Prof. Ivan Moura Campos
UFMG

Visto e permitida a impressão
Rio de Janeiro, 05/12/80



Prof. Arndt von Staa
Coordenador dos Programas de Pós-Graduação e Pesquisa do Centro Técnico Científico