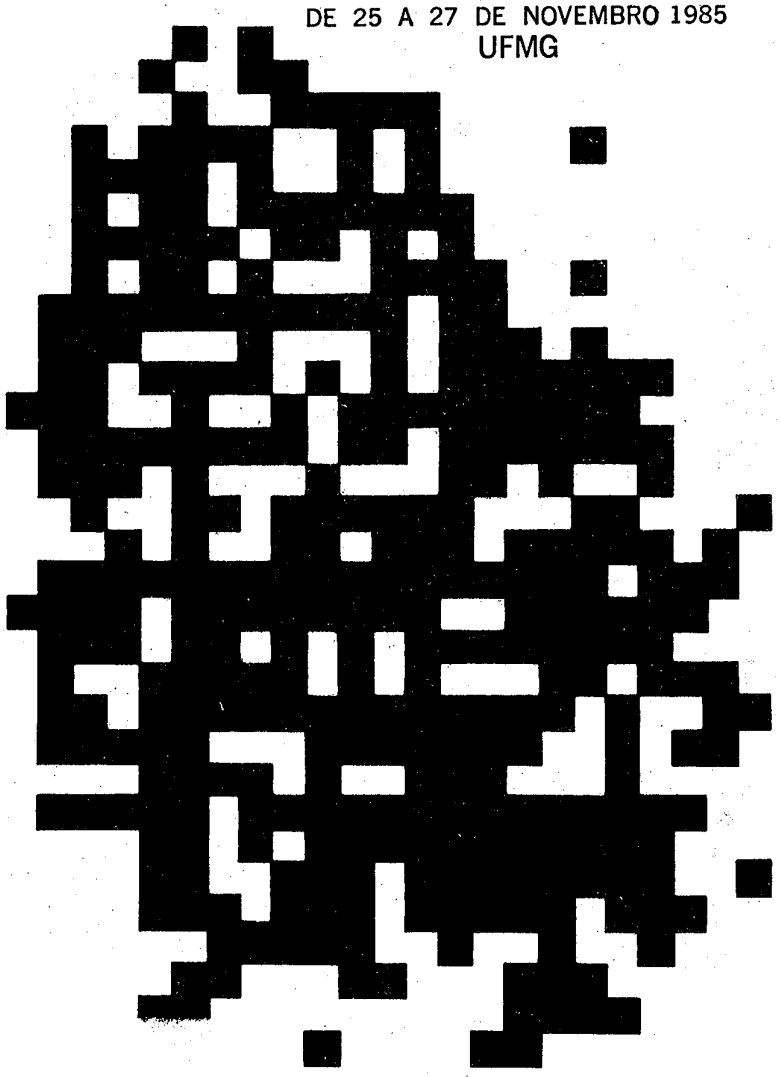
V SIMPÓSIO

SOBRE DESENVOLVIMENTO DE SOFTWARE BÁSICO

DE 25 A 27 DE NOVEMBRO 1985
UFMG



ANAIS

5º SIMPÓSIO SOBRE DESENVOLVIMENTO DE SOFTWARE BÁSICO

BELO HORIZONTE 25 A 27 DE NOVEMBRO DE 1985

A N A I S

PROMOÇÃO: SOCIEDADE BRASILEIRA DE COMPUTAÇÃO - SBC
. COMISSÃO ESPECIAL PARA LINGUAGENS E SISTEMAS DE PROGRAMAÇÃO
UNIVERSIDADE FEDERAL DE MINAS GERAIS - UFMG

PATROCÍNIO: CONSELHO NACIONAL DE DESENVOLVIMENTO CIENTÍFICO E TECNOLÓGICO - CNPq

SOBRE O USO DE MODULA-2 PARA PROGRAMAÇAO EM AMBIENTES DISTRIBUIDOS

Lidia Segre *

Michael Stanton #

* COPPE/UFRJ
Programa de Sistemas
Caixa Postal 68511
21945 Rio de Janeiro, RJ

Departamento de Informàtica
PUC-RJ
Rua Marquês de São Vicente, 225
22453 Rio de Janeiro, RJ

Resumo

Este trabalho relata os resultados de uma pesquisa sobre a construção de software distribuido analisando particularmente formalismos para os conceitos de paralelismo e confiabilidade.

Foi escolhida a linguagem Modula-2, à qual são propostas extensões para atender aos conceitos mencionados com o intuito de montar um ambiente de programação distribuida. São descritos neste trabalho a especificação e um modelo de implementação do uso de processos concorrentes, de mecanismos de comunicação e sincronização.

I - Introdução

Um dos objetivos deste projeto è o de conduzir uma pesquisa sobre a construção de software distribuido, incluindo também aplicações de tempo real, para executar em arquiteturas compostas por um conjunto de estações autònomas, sem memòria compartilhada, e interligadas através de um sistema de comunicação. Para esta classe de software è necessàrio propor formalismos para os

conceitos de paralelismo, configuração e confiabilidade.

O hardware de um sistema distribuido inclui diversos processadores e, portanto, suporta naturalmente a execução em paralelo de vàrios programas em estações distintas. Adicionalmente, o processador de uma estação poderà empregar tècnicas de multiprogramação. O conceito de programação concorrente engloba tanto o paralelismo real como o aparente. Devem ser incluidos tambèm mecanismos para comunicação e sincronização entre as atividades concorrentes.

Nos sistemas centralizados, a construção modular de programas significa que um programa è montado a partir de um conjunto de mòdulos desenvolvidos e compilados separadamente. A especificação dos mòdulos que compòem um programa, e de suas ligações, è chamada configuração, e linguagens como C/Mesa [Mitchell 1979] foram definidas para formalizar esta tarefa.

Para sistemas distribuidos, a configuração precisa incluir tambèm a localização física dos mòdulos nas diferentes estações. Outra diferença entre sistemas centralizados e distribuidos è que nestes últimos a configuração do hardware poderà ser alterada dinamicamente, obrigando portanto uma reconfiguração do software. A linguagem Conic [Sloman 1984] se destina a configuração de software em sistemas distribuidos.

A confiabilidade do software pode ser tratada atravès de duas abordagens complementares: prevenção de falhas e tolerância a falhas. A estrutura modular de software facilita a prevenção de falhas, jà que permite o desenvolvimento e a verificação de mòdulos de forma independente. Para que erros ocorridos durante a execução não ocasionem a paralização total do programa, è necessàrio ter mecanismos para intercepção de situações de exceção, e a consequente tomada de providências corretivas, entre as quais pode ser considerada a reconfiguração dinâmica de software.

Como ponto de partida para o trabalho sobre programação em sistemas distribuidos resolvemos selecionar como base uma linguagem de programação que jà atende a vàrias das preocupações citadas acima. Um enfoque diferente è dado em [Loques 1985], onde a linguagem Pascal è estendida sintatica e semanticamente para apoiar programação em sistemas distribuidos.

Entre as novas linguagens de programação que apareceram no final da última dècada està Modula-2 [Wirth 1982, Segre 1984a], que às caracterìsticas de Pascal acrescenta facilidades para o desenvolvimento modular (as quais separam a definição da interface de um mòdulo da sua implementação), para compilação separada, para programação concorrente e para a criação de tipos abstratos. Alèm disto, Modula-2 permite o manuseio direto dos mecanismos de endereçamento de memòria, de interrupções e de entrada/saìda em sistemas monoprocessador. Com estas caracterìsticas torna-se possìvel escrever todo o software de um sistema de computação nesta linguagem, como ilustràdo no caso do

computador Lilith [Wirth 1981].

Um aspecto importante desta linguagem è sua disponibilidade para alguns minicomputadores (PDP-11, VAX-11), e em particular para vàrios microcomputadores, inclusive o Apple II e o PC da IBM.

Modula-2 foi escolhida para este trabalho, no qual vão-lhe ser acrescentadas as extensões suficientes para atender aos conceitos de paralelismo e confiabilidade necessàrios para software distribuido, como foi discutido antes. Em relação a configuração os autores apresentam em outro artigo [Segre 1985] uma proposta de uma linguagem para especificar a configuração estàtica de um programa distribuido a partir de um conjunto de mòdulos escritos em Modula-2 com as extensões mencionadas.

Depois dos estudos realizados sobre o problema, chegou-se às seguintes conclusões:

a) as extensões a Modula-2, que seriam necessàrias para implementar os conceitos acima mencionados, poderão ser realizadas através de uma biblioteca de mòdulos escritos quase inteiramente em Modula-2 sem precisar alterar esta linguagem, e pela definição de uma linguagem de configuração.

b) a biblioteca citada em (a) conterà mòdulos para implementar programação paralela tanto dentro de uma estação como em estações distintas, com suporte para comunicação e sincronização, e para o tratamento de falhas.

Passaremos então a descrever a especificação das extensões propostas a Modula-2 e uma primeira forma de implementação destas extensões.

II - Especificação das extensões a Modula-2 para paralelismo e comunicação em sistemas distribuidos.

a) Processos Concorrentes

Modula-2 [Wirth 1982] foi projetada inicialmente para ser implementada num computador convencional com um único processador. Ela oferece algumas facilidades bàsicas para multiprogramação que permitem a especificação de processos quasi-concorrentes e de real concorrência no caso de dispositivos periféricos.

A palavra processo è utilizada em Modula-2 com o significado de co-rotina. Uma co-rotina è um programa sequencial, com suas pròprias variàveis particulares, que è executada em regime de quase concorrência junto com outras co-rotinas num único processador. O processador è comutado de uma co-rotina para outra através de um comando explicito de transferência de controle. A co-rotina que recebe controle continua sua execução no ponto da sua última suspensão através do comando de transferência de

controle.

Como em Modula-2 co-rotinas são consideradas facilidades de baixo nível, seu tipo associado e seus operadores fazem parte do pseudo-módulo chamado SYSTEM, presente em cada implementação de Modula-2. Para que outros módulos possam usar estas ferramentas eles devem importá-las a partir de SYSTEM.

Utilizando o conceito de co-rotina, é possível agora construir módulos que implementam o conceito de processo concorrente, tradicionalmente usado como elemento básico para a construção de software concorrente [Dijkstra 1968]. Estes módulos são funcionalmente equivalentes ao suporte de execução de linguagens tais como Modula [Wirth 1977], Pascal Concorrente [Brinch Hansen 1975], Pearl [DIN 1980] e ADA [DOD 1979]. Nestas linguagens são definidos modelos fixos para representar atividades concorrentes, mecanismos de comunicação e sincronização e para o tratamento de E/S. Esses modelos são implementados em software através de um chamado suporte de execução que geralmente não pode ser escrito na própria linguagem.

Existem diversos modelos para a definição de processos concorrentes e suas interações, alguns baseados no conceito de acesso direto a variáveis compartilhadas [Hoare 1974], e outros que utilizam a troca explícita de dados na forma de mensagens [Gentleman 1981]. Para cada modelo existem diversas opções de políticas de escalonamento. A linguagem Modula-2 nos dá condições de programar quase toda esta variedade de alternativas, sem ter que recorrer a linguagem de montagem. Na prática, isto é feito através da programação de um módulo que exporta os conceitos apropriados para o usuário (programador de software concorrente). Usando a divisão entre módulo de definição e módulo de implementação [Segre 1984b], os detalhes da implementação destes núcleos podem ser totalmente escondidos do usuário.

O núcleo proposto contém o conceito de processo, e as operações Criaprocesso e Fimprocesso que criam e destroem processos, respectivamente, e que poderão ser chamadas de fora do núcleo. As implementações serão vistas com mais detalhes em outra seção.

Num sistema distribuido cada estação suportará a execução multiprogramada de vários processos através de um núcleo próprio.

b) Comunicação e sincronização

Segundo Lauer e Needham [Lauer 1979], há dois principais modelos que são usados para combinar os conceitos de paralelismo, sincronização e comunicação e que definem um estilo de programação: sistemas orientados por troca de mensagens e sistemas orientados por chamadas de procedimentos.

Um sistema orientado por mensagens contém um pequeno número

estàtico de processos grandes, e è caracterizado pela facilidade de passar mensagens entre os processos. Os processos dispòem de primitivas para enviar, receber e esperar mensagens e examinar o estado das filas de mensagens. As estruturas de dados manuseadas por mais de um processo são enviadas em mensagens entre estes.

Um sistema orientado por chamadas de procedimentos contèm um nûmero grande de processos pequenos, nûmero este que pode variar rapidamente. Estes processos se comunicam atravès de compartilhamento direto das variâveis e da sincronização do acesso a elas. Isto è realizado, por exemplo, usando monitores para encapsular as variâveis compartilhadas, às quais se terà acesso atravès de chamadas de operações definidas no mesmo monitor. Quando um processo chama um procedimento de um monitor, ele pode ser obrigado a ficar esperando numa fila associada ao monitor, atè que outro processo executando uma operação do mesmo monitor o libere.

Lauer e Needham demonstraram que, sob certas restrições, existe uma dualidade entre os dois modelos, e que um programa realizado segundo um modelo tem seu correspondente direto no outro.

Consideramos em maior sintonia com o espìrito de Modula-2 a escolha do modelo de comunicação e sincronização entre processos baseado em chamadas de procedimentos. Na linguagem Modula-2 as interfaces dos mòdulos são procedurais e a interrelação entre eles è feita atravès de chamadas de procedimentos importados, à partir dos mòdulos que os definem e que os exportam para os outros mòdulos.

Os dois modelos de comunicação e sincronização citados aqui foram aplicados inicialmente a sistemas compostos por conjuntos de processos residentes num ûnico processador. Mas eles podem ser estendidos para sistemas de mais de um processador interligados de alguma maneira, tendo ou não memòria compartilhada. No nosso caso particular, estamos supondo uma arquitetura distribuida composta por um conjunto de processadores interligados, sem memòria compartilhada. Neste caso precisamos então estender nosso modelo baseado em chamadas de procedimentos acrescentando o conceito de chamada remota de procedimento (RPC), que serà utilizado como o principal mecanismo de comunicação entre estações [Nelson 1981].

O desenvolvimento do conceito de chamada remota de procedimento è uma consequência natural da evolução do trabalho sobre abstração de dados, de controle e de concorrência nas linguagens procedurais. Uma caracterìstica crucial das linguagens que incorporam o conceito de monitor, tais como Pascal Concorrente, Mesa e Modula è que seus processos compartilham memòria e a comunicação entre eles è feita atravès de chamadas locais de procedimentos de monitores. No caso de um sistema distribuido no qual os monitores estão alocados em espaços de endereçamento diferentes daqueles dos programas que os chamam, è necessàrio utilizar chamadas remotas de procedimentos.

Podemos definir uma chamada remota de procedimento como uma chamada onde o procedimento alvo está armazenado e será executado num computador diferente daquele que faz a chamada. A idèia principal è que as chamadas remotas de procedimentos em programas distribuidos se assemelhem e se comportem de forma exatamente igual às chamadas locais de procedimentos em programas tradicionais. Esta transparência sintática e semântica levanta uma sèrie de problemas que precisam ser encarados e resolvidos.

Os mecanismos de nomeação e ligação utilizados nas chamadas locais devem ser estendidos analogamente para as chamadas remotas. Outras caracteristicas das chamadas locais são mais difíceis de estender para chamadas remotas, tais como, por exemplo, a passagem de parâmetros e a propriedade de retorno garantido da chamada local. Na chamada remota de procedimento não podem ser passados ponteiros ou endereços em geral como argumentos dado que estes só têm validade no espaço de endereçamento de quem faz a chamada. Isto requer um tratamento especial para o uso de estruturas encadeadas e também para os parâmetros que são atualizados pelo procedimento chamado.

Garantir que uma chamada remota de procedimento sempre retorne è difícil num sistema distribuido onde tanto os computadores quanto a comunicação podem falhar arbitrariamente. Para isto è necessário implementar mètodos eficientes de deteção e tratamento de falhas que gerem mensagens de erros de forma anàloga ao caso de chamadas locais.

III - Especificação da implementação de paralelismo e comunicação

a) Núcleo de multiprogramação

O núcleo escolhido para ser implementado foi baseado no núcleo de Holt [Holt 1978] que provè os tipos usados para implementar o conceito de monitor.

No núcleo de Holt os processos podem estar num dos quatro seguintes estados:

- executando
- pronto para executar
- bloqueado por alguma condição
- bloqueado esperando o fim de uma operação de entrada/saída

Notamos que num monoprocessador só pode haver um processo executando de cada vez.

Os processos prontos são aqueles que estão em condição de executar e que estão esperando a sua vez para ganhar controle do processador. Eles formam uma fila chamada a fila de processos

prontos. Existe um processo ocioso que só executa quando todos os demais processos estiverem bloqueados, e ele é sempre o último da fila.

Os processos executam seu código sequencialmente e se comunicam e se sincronizam com outros processos através de acesso a variáveis compartilhadas, testando determinadas condições lógicas sobre essas variáveis e assinalando as condições quando estas forem satisfeitas. Para isto o núcleo implementa duas primitivas Espera e Avisa que atuam sobre variáveis do tipo condição. É definida uma variável do tipo condição para cada condição lógica e os processos que esperam por uma condição lógica são suspensos e inseridos na fila associada a ela.

Os processos também podem estar bloqueados esperando o fim da execução de alguma operação de entrada/saída associada a algum periférico. Em geral, haverá uma fila própria associada a cada periférico.

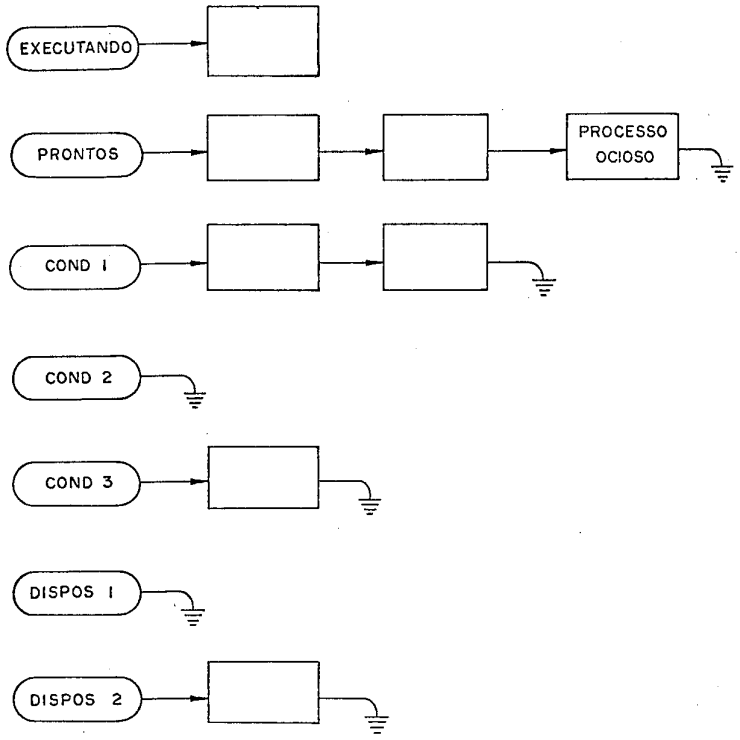


Figura 1: Filas de processos do núcleo de Holt

A estrutura de dados usada para compor as filas de processos è o descritor de processo, que guarda o contexto do processo (informação volàtil, tipicamente contida em registradores do processador), quando este estiver suspenso, e tambèm aponta para o pròximo descritor na mesma fila. Estas filas são ilustradas na figura 1 onde os retângulos representam os descritores de processo, e as ligações dos descritores estão representadas por setas.

Na nossa implementação cada processo tem um descritor de processo a ele associado com a estrutura da figura 2. Os processos estão ligados em filas e cada um contèm um ponteiro para o pròximo descritor da fila, no campo chamado seguinte. Os outros campos do descritor correspondem à àrea na qual se salva o contexto quando o processo for suspenso e às informações sobre a pilha associada ao processo.

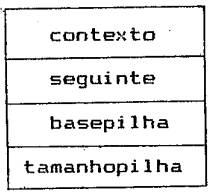


Figura 2: Descritor de processo

Existem as filas de processos, uma para cada condição, uma para cada perifèrico e uma para os processos prontos. Para estas filas foi definido um tipo chamado FilaProc que è exportado pelo mòdulo de forma opaca.

Passaremos a descrever as primitivas implementadas pelo mòdulo NucleoHolt.

O procedimento Criaprocesso chama a primitiva de Modula-2 que cria um processo, depois de ter alocado e inicializado o descritor do processo e tê-lo inserido no primeiro lugar da fila dos prontos para ser o pròximo a ser executado.

Quando um processo executa a primitiva Espera, ele è suspenso e inserido no final da fila associada à condição e o controle è transferido para o primeiro processo da fila dos prontos.

A primitiva Avisa testa a fila associada à condição. Se ela estiver vazia o processo continua a sua execução; caso contràrio, suspende o processo colocando-o no primeiro lugar da fila dos prontos e passa o controle para o primeiro processo da fila de condição.

A primitiva Filavazia testa a fila de condição e retorna o valor TRUE caso ela não possua nenhum processo pendurado.

As filas de condição são inicializadas através da primitiva Inicializa.

Existe uma primitiva para matar um processo quando acabar a execução de seu código. Ela libera a área do processo e passa o controle para o primeiro processo da fila dos prontos.

A inicialização do módulo consiste da alocação de um descritor de processo e a sua inicialização.

Nesta implementação cada monitor corresponde a um módulo que encapsula as estruturas de dados compartilhadas e condições associadas sobre as quais atuam as primitivas Espera e Avisa. As operações do monitor são implementadas através de procedimentos que são exportados e que serão executados com exclusão mútua. Esta exclusão mútua é garantida através do mecanismo de prioridade da linguagem Modula-2. É importante frisar que as primitivas Espera e Avisa somente devem ser usadas dentro de monitores para garantir a integridade das filas de condição.

O módulo programado para implementar o Núcleo de Holt, para poder ser utilizado em diferentes programas escritos em Modula-2, precisa ser uma unidade de compilação. Portanto ele deve constar de uma parte de definição, que representa a interface com os outros módulos, e contém os nomes dos procedimentos e tipos por ele implementados que podem ser importados pelos outros módulos, e de uma parte de implementação, que contém os detalhes dos tipos e os algoritmos correspondentes aos procedimentos que são exportados, além de variáveis, procedimentos e módulos locais e a inicialização do módulo.

Para programar o módulo precisamos importar do módulo SYSTEM os tipos e as primitivas que estão relacionadas com o tratamento de processos, vistos na seção anterior. Precisamos também importar primitivas pertencentes ao módulo que gerencia a memória, que supomos existir, para alocar e liberar áreas da memória.

Uma descrição mais completa desta implementação pode ser achada em [Segre 1984b].

b) Sistema de comunicação

Este é o nível que vai tratar das comunicações entre estações diferentes.

Num sistema distribuido a escolha do modelo de comunicação entre processos baseado em chamadas de procedimentos tem como consequência a necessidade de implementar chamadas remotas de procedimentos (RPC) como principal mecanismo de comunicação entre

estações [Nelson 1981], como já foi descrito anteriormente.

Para a implementação de chamadas remotas de procedimentos em sistemas distribuidos, foi escolhido o modelo que usa o conceito de "stub" [Birrell 1984, Nelson 1981] que apresentaremos a seguir. Ilustramos o caso de uma chamada feita por um módulo (usuário) de um procedimento contido num módulo remoto (servidor). Quando é executada a chamada remota, são envolvidos cinco componentes de programa: o usuário, o "stub" do usuário, o pacote de comunicação de RPC (chamado RPC runtime), o "stub" do servidor, e o servidor. A relação entre eles é ilustrada pela figura 3.

Para que uma chamada remota pareça formalmente igual a uma chamada local (propriedade de transparência já mencionada anteriormente) o usuário passa o controle para seu "stub", através de uma chamada local, ficando bloqueado até voltar o controle, depois da execução da chamada remota. O procedimento do servidor é ativado também através de uma chamada local feita por seu "stub". A descrição que segue explica a interrelação entre os dois "stubs".

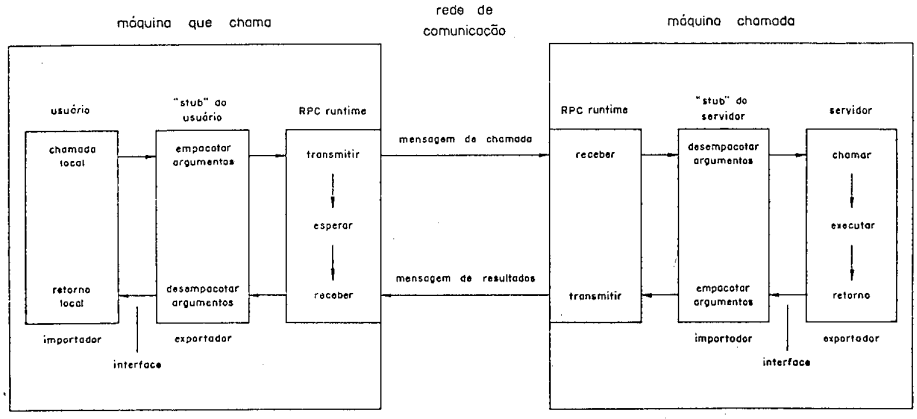


Figura 3: Esquema da chamada remota de procedimento

O "stub" do usuário é o responsável pelo gerenciamento da chamada remota de um procedimento, feita pelo usuário, empacotando o nome do procedimento junto com os argumentos da chamada numa mensagem que é enviada para a máquina onde reside o servidor contendo o procedimento chamado. Ele é encarregado também de receber uma mensagem com os resultados da execução do procedimento chamado, e de passá-los para o usuário que ficou bloqueado esperando por eles.

O RPC runtime da estação do usuário ativado pelo "stub" do usuário é o responsável pelo envio da mensagem de chamada para a máquina que contém o procedimento e pelo recebimento da mensagem de resultados da mesma máquina.

O RPC runtime da estação do servidor tem funções análogas ao da estação do usuário, com a diferença que é ele que ativa o "stub" do servidor. O RPC runtime da estação do servidor recebe a mensagem de chamada que contém o nome do procedimento a ser executado junto com os parâmetros necessários para cada chamada. Ele passa esta informação para o "stub" do servidor, que é o encarregado da identificação do conteúdo da mensagem de chamada, e de passar a chamada do procedimento correspondente ao servidor que o contém, para iniciar a sua execução.

Quando a execução do procedimento acaba, o "stub" do servidor recebe os resultados, empacota-os e passa-os para o RPC runtime do servidor; este se encarrega de enviá-los para o RPC runtime da estação do usuário que fez a chamada.

A execução da chamada remota prossegue através da execução do "stub" do usuário que recebe os resultados. Ele desempacota os resultados para enviá-los ao usuário que pode então continuar a execução do seu código, após a chamada remota ter sido concluida.

Para cada chamada remota será necessário então criar os códigos dos "stubs" correspondentes, um para o usuário (isto é, para quem faz a chamada) e outro para o servidor (isto é, para quem contém o procedimento chamado). Para melhor ilustrar este esquema apresentaremos na figura 4 um exemplo de um módulo cliente (um leitor) de um módulo servidor de arquivos que oferece diferentes operações a serem executadas sobre o arquivo. O leitor e o servidor de arquivos se encontram em nós diferentes. Este exemplo está esquematizado usando a linguagem Mesa.

Na etapa de configuração são geradas as tabelas contendo a informação sobre a localização dos módulos nas estações físicas de forma que o RPC runtime possa identificar para que estação precisa enviar a chamada a ser executada.

Essas tabelas podem estar centralizadas numa estação em particular, estar replicadas em cada estação ou estar divididas em tabelas contendo somente a informação necessária para cada estação em particular.

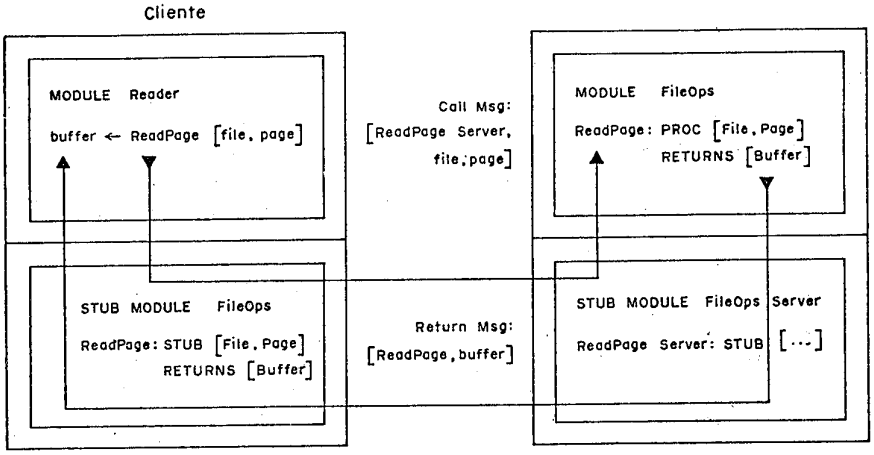


Figura 4: Chamada remota de procedimento implementada através do conceito de "stub"

Uma forma de implementar a execução das chamadas remotas de procedimentos numa estação è tratando as invocações de forma concorrente e não sequencial; esta forma tem se mostrado mais eficiente e mais segura [Nelson 1981]. Por exemplo, eficiência pode ser perdida pela serialização da execução de acessos ao mesmo módulo a partir de processos distintos. Em relação a segurança uma cadeia de chamadas remotas indiretamente recursivas provocará bloqueio perpètuo. Esta situação poderà acontecer também no caso de serem sincronizadas entre si chamadas independentes ao mesmo servidor.

Existem duas possibilidades para tratamento dessas chamadas: uma solução è criar um processo para cada chamada recebida e destruí-lo no final da execução; a outra è ter um número determinado de processos ociosos esperando pelas chamadas a serem atendidas. A primeira implica numa sobrecarga maior para criar e destruir processos enquanto que a segunda limita o número de chamadas a serem atendidas e obriga a manter processos ociosos no caso de não acontecerem chamadas.

A implementação de chamadas remotas de procedimentos através

de "stubs" requer a criação de um gerador de "stubs", tanto para o exportador (servidor) como para o importador (usuário). Em princípio, um "stub" de usuário implementa a mesma interface que o servidor que ele representa, e está totalmente especificado pelo módulo de definição do servidor. O gerador de "stubs" então processa este módulo de definição e gera código para converter cada chamada a um procedimento da interface do servidor remoto num envio de mensagem à estação remota, identificando o procedimento chamado e os argumentos da chamada. O endereço desta estação será obtido da tabela de módulos mantida na estação do usuário, que será elaborada e carregada durante a interligação da configuração.

Ao mesmo tempo o gerador de "stubs" monta o código do "stub" do servidor, que receberá as mensagens enviadas pelo "stub" do usuário convertendo-as em chamadas locais aos procedimentos apropriados do servidor. Em suma, a importação de uma interface remota, detetada na configuração de um programa, resulta na geração de um par de "stubs" que comunicam entre si os detalhes da chamada remota e do retorno.

O processamento dos argumentos de uma chamada remota requer cuidado. Certos tipos de argumentos, por exemplo, ponteiros, não podem ser enviados a uma outra estação e têm que ser desreferenciados. No caso específico de parâmetros de saída (parâmetros VAR de Modula-2), somos obrigados a implementá-los usando valor-resultado ("value-result"), isto é, ao parâmetro corresponde uma variável local do procedimento que será inicializada com o valor do argumento na hora da chamada e cujo valor final na hora do retorno será atribuído ao argumento. Os "stubs" têm que converter a lista de argumentos no texto da mensagem de chamada, e vice-versa. O gerador dos "stubs" fornece rotinas para empacotar/desempacotar argumentos, além de determinar se os argumentos devem ser transformados. Finalmente, o gerador de "stubs" deteta numa interface a existência de ítens inadmissíveis, como, por exemplo, a exportação de variáveis.

IV - Especificação e implementação de tratamento de falhas

a) Exceções

Na última seção vimos que a exigência de robustez de software distribuído, combinada com a transparência do conceito de chamada remota de procedimentos, leva inexoravelmente ao requisito de um mecanismo geral para a deteção e tratamento de falhas na execução de um programa. Propomos nesta seção uma maneira de acrescentar um mecanismo deste tipo para programas escritos em Modula-2, baseado no conceito de exceções. A exposição que segue é baseada em [Yemini 1985].

Chamamos operações os procedimentos exportados por um módulo. Cada operação é programada para executar alguma

computação sobre dados de entrada no seu domínio de definição. Estes dados são caracterizados pela afirmativa do caso normal de entrada da operação. A aplicação da operação a dados de entrada que satisfazem a afirmativa do caso normal de entrada prescrita deverá gerar um resultado que satisfaz a afirmativa de saída prescrita.

Exemplos de situações nas quais os dados de entrada não satisfazem a afirmativa do caso normal de entrada incluem, por exemplo:

- encontrar fim de arquivo numa operação de leitura,
- divisão por zero,
- desempilhamento de uma pilha vazia.

Supomos então que a operação vai verificar a afirmativa do caso normal de entrada, identificar aqueles casos onde não é válida esta afirmativa e avisar ao usuário da sua ocorrência. Estes casos se chamam exceções daquela operação, e dizemos que a operação é o assinalador destas exceções. Assinalar a exceção significa avisar ao usuário que foi detetada uma exceção.

Em geral, somente o usuário de uma operação sabe qual é o objetivo da sua aplicação, e, portanto, qual é o significado da deteção de uma exceção. Logo, cabe ao usuário de uma operação assinalada determinar o processamento das exceções, através de uma rotina chamada de tratador de exceção. Esta divisão de responsabilidades entre a operação assinaladora e seu usuário é de importância crítica para modularidade, pois, na sua ausência, aumenta o grau de acoplamento entre o usuário e operação por ele chamada. Em [Yemini 1985] é apresentada uma série de requisitos que devem ser satisfeitos por um mecanismo de tratamento de exceções para permitir uma resposta flexível por parte dos usuários à deteção de exceções sem comprometer a modularidade. Entre estes damos destaque aqui a respostas alternativas dos tratadores, parametrização dos tratadores, a propagação explícita de exceções e o papel verificador do compilador.

Respostas alternativas dos tratadores de exceções. Entre as respostas possíveis se encontram as seguintes:

1. prosseguir o assinalador,
2. terminar o assinalador,
3. executar novamente o assinalador,
4. propagar a exceção,
5. transferir controle.

Parametrização de tratadores de exceções. O tratador de exceção precisa ter parâmetros formais para permitir que o assinalador lhe passe informação específica a cada ocorrência.

Propagação explícita de exceções. Uma alternativa usada em linguagens como Ada [DOD 1979] é propagar exceções automaticamente através da cadeia de chamadas de procedimentos (cadeia dinâmica). Isto é considerado inapropriado, pois, em geral, cada nível de chamada representa um nível de abstração. A propagação de uma exceção por mais de um nível de abstração tem o perigo de encontrar um tratador que não saiba o contexto original da exceção e não possa ter uma resposta adequada. Para manter o desacoplamento entre os níveis de abstração, exigimos que uma exceção somente seja tratada pelo usuário da operação assinaladora. No caso de querer propagar uma exceção ao usuário do usuário, isto deve ser feito assinalando uma outra exceção relativa a este nível superior de abstração.

Papel do compilador. Deve-se aproveitar das regras de escopo e de tipos da linguagem para fazer verificação em tempo de compilação da consistência das interfaces e da proteção dos detalhes da sua implementação. Entre os itens verificados se incluem:

- o conjunto correto de argumentos de cada exceção assinalada;

- o conjunto correto de parâmetros formais de cada tratador de exceção;

- o requisito de propagação explícita de exceções, garantindo que exceções definidas por um assinalador só sejam assinaladas por ele.

b) O modelo de substituição

Yemini e Berry [Yemini 1985] propõem um novo mecanismo para tratar exceções chamado o modelo de substituição ("replacement model"). Neste modelo adota-se um enfoque baseado em expressões: um programa é uma expressão composta, e uma exceção é explicitamente associada a uma expressão - seu assinalador em potencial. A exceção do assinalador corresponde a uma subexpressão que não pôde ser avaliada no assinalador; o resultado do tratador substitui ou o resultado da subexpressão, ou o resultado da chamada do assinalador, o que motiva o nome "modelo de substituição".

No mesmo artigo, Yemini e Berry propõem extensões à linguagem Algol 68 [van Wijngaarden 1975], acrescida de pacotes análogos aos definidos em Ada [DOD 1979], para incorporar seu modelo de substituição. Estas extensões incluem

- a cláusula on ... no para definir um tratador de exceção

- a construção de saída replace

- extensões à sintaxe de cláusulas fechadas para indicar que elas podem assinalar exceções.

O seguinte exemplo, usado em [Yemini 1985] ilustra o mecanismo. O procedimento convert traduz um arranjo de números inteiros numa cadeia de caracteres. Caso haja código inválido, assinala a exceção badcode.

```
proc convert=(ref [] int code)string
                  signals(  # declara as exceções assinaladas#
                  exc(int)(char,string)badcode
                           ):
begin
   string s:="";
   for i from lwb code to upb code do
      int codei = code[i];
      s:= s+if codei<=maxchar and codei>=minchar
           then repr codei
           else badcode(codei) #assinala a exceção#
              fi
   od;
   s
end
```

Neste caso, convert pode assinalar a exceção badcode com um argumento do tipo int. O tratador da exceção ou pode retornar ao assinalador convert com um valor do tipo char, ou pode retornar ao usuário do convert com um valor do tipo string. Para especificar um tratador para a exceção badcode basta especificá-lo numa cláusula on ... no associada ao usuário de convert, como ilustrado a seguir.

```
proc usuário = ...
begin
   ...
   convert(nums);
   ...
end
on badcode = (int i)(char,string):
    <corpo do tratador de badcode>
no
```

Ilustramos duas possíveis formulações do tratador.

(i) implementação do prosseguimento do assinalador

```
on badcode = (int i)(char,string):
   "?" # substitui o caractere inválido #
no
```

(ii) implementação do têrmino do assinalador devolvendo um resultado ao usuário

```
on badcode = (int i)(char,string):
    "" replace # substitui a cadeia com caractere invàlido por uma
                    cadeia vazia #
no
```

Observamos que em (i) o resultado do tratador è do tipo char e em (ii) o resultado è do tipo string.

O modelo de substituição também permite propagar uma exceção para um nível de abstração mais alto. Isto è feito através do assinalamento de uma segunda exceção pelo tratador da primeira. Ilustramos isto com um exemplo adaptado de [Yemini 1985], que utiliza o mesmo procedimento convert que já foi definido acima.

```
proc intermediary = void  signals(exc(char,void)finish):
   begin
      ...
      print(convert(nums))
      ...
   end
   on badcode = (int i)(char,string):
         finish
   no;

proc end user = void:
   begin
      ...
      intermediary;
      ...
   end
   on finish = (char,void):
         skip replace
   no
```

Neste exemplo, o assinalamento da exceção badcode causa o assinalamento da exceção finish, que aqui retornarà controle para end user após a chamada do procedimento intermediary.

Antes de apresentar uma adaptação do modelo de substituição à linguagem Modula-2, devemos chamar atenção a vàrias das características importantes da proposta de Yemini e Berry.

(i) O assinalamento de exceções pode ser associado a qualquer cláusula fechada (por exemplo cláusula condicional, iteração, bloco) da linguagem. Neste caso a cláusula è precedida por uma cláusula signals.

(ii) Um tratador de exceção pode ser associado a qualquer cláusula fechada, através do acrèscimo no seu final da cláusula on ... no. A ligação entre o tratador e o

assinalador é determinada através do nome da exceção. Ao chamar convert, será ligado a sua ativação o tratador badcode que está visível no ponto da chamada. Esta ligação é portanto determinável estaticamente. Outra consequência é que esta ligação estática facilita o uso de tratadores padrão, que seriam usados em caso de omissão pelo usuário do assinalador.

(iii) O tratador tem as características de um procedimento, declarado localmente à cláusula fechada, a qual está associada sua cláusula on, e o assinalamento da exceção implica na sua chamada, possivelmente com a passagem de parâmetros. O retorno de controle do tratador se faz por duas alternativas: o retorno normal, ao ponto do assinalamento da exceção, ou o retorno ao usuário do assinalador (replace).

De maneira geral o modelo de substituição oferece grande flexibilidade no tratamento de exceções permitindo todos os tipos de resposta que identificamos como desejáveis no início desta apresentação. Ao procurar um mecanismo de exceções para usar com Modula-2 comparamos favoravelmente esta flexibilidade com, por exemplo, a proposta de Ada [DOD 1979]. Esta proposta, além de especificar o tratador junto com o assinalador, comprometendo a modularidade, só prevê a opção de terminar o assinalador.

Adicionalmente, como veremos na próxima seção, é relativamente fácil implementar os traços salientes do modelo de substituição sem alterar o compilador de Modula-2, através de convenções de programação e a provisão de um módulo de biblioteca que dá suporte às novas facilidades necessárias.

c) Adaptação do modelo de substituição para Modula-2

A proposta abaixo apresentada para incorporar o modelo de substituição à linguagem Modula-2 é necessariamente menos ambiciosa que a descrita acima, principalmente porque não nos permitimos sugerir qualquer alteração ao compilador que utilizamos. Por isto toda a proposta tem que se basear na definição de um módulo de biblioteca, que exporta os tipos e as operações necessários para implementar o tratamento de exceções. Módulos precisando incorporar exceções então importarão estes tipos e operações e passarão a utilizá-los segundo convenções a serem apresentadas neste artigo.

Algumas limitações que decorrem do uso de Modula-2 sem alterações dizem respeito a definição de um assinalador e à ligação entre uma exceção e seu tratador. Na proposta de Yemini e Berry, qualquer cláusula fechada pode declarar que assinala exceções. Em Modula-2 isto somente é fácil de fazer para procedimentos se fizermos a implementação óbvia de tratadores de exceções como procedimentos. A associação do tratador com a exceção também não pode manter a forma de Yemini e Berry. A única

opção parece ser especificar explicitamente como parâmetros da chamada de uma operação os tratadores que queremos associar às suas exceções. Isto requer que toda chamada cite os tratadores, excluindo assim o uso de um tratador padrão em caso de omissão ("default").

Outra dificuldade associada a esta solução è a proibição em Modula-2 da passagem de procedimentos locais como parâmetros de outros procedimentos, sendo permitidos como parâmetros somente procedimentos declarados no nível 0. Isto significa que um procedimento que implementa um tratador não tem acesso às variàveis locais e parâmetros do procedimento do usuário pelo qual foi especificado como tratador. Fica restrito, portanto, a variàveis globais a comunicação entre usuário e tratador.

Esta última restrição ocasiona algumas dificuldades para que possamos implementar a propagação de exceções. Neste caso, um tratador assinala uma exceção ligada a outro tratador passado como parâmetro a seu usuário. Como è extremamente importante poder propagar exceções, será necessário utilizar variàveis globais para estabelecer ligações entre exceções e tratadores. A inicialização e o acesso a estas variàveis globais faz parte das regras da programação de exceções desta proposta.

A proposta agora será descrita.

(i) Um tratador de exceção è um procedimento de nível 0, que será passado como parâmetro para a operação assinaladora. O assinalamento da exceção pela operação è feito através da chamada do tratador.

(ii) A saída do tratador por retorno ao assinalador è feito normalmente usando, quando for o caso, o comando RETURN para devolver um valor.

(iii) Para implementar a semântica de REPLACE (retorno ao usuário do assinalador), introduzimos o tipo ENVIRONMENT e dois procedimentos, RegisterException e Replace. Replace efetua a transferência desejada, que è especificada através de um parâmetro do tipo ENVIRONMENT que identifica o endereço de retorno e o ambiente de execução (registro de ativação). O valor deste parâmetro è obtido chamando RegisterException dentro da operação assinaladora. ENVIRONMENT, Replace e RegisterException são exportados pelo módulo Exceptions, cuja interface è dada a seguir.

```
DEFINITION MODULE Exceptions;

  FROM SYSTEM IMPORT ADDRESS;

  EXPORT QUALIFIED ENVIRONMENT, RegisterException, Replace;

  TYPE ENVIRONMENT;
```

```
  PROCEDURE RegisterException (VAR env:ENVIRONMENT);
      (*lembra  quem  chamou a operação assinaladora  da  exceção
        para poder retornar com Replace*)

  PROCEDURE Replace (env : ENVIRONMENT;
                     A : ADDRESS;
                     n : CARDINAL);
    (*retorna controle para o usuário da operação assinaladora  da
      exceção,  que foi guardado em env, devolvendo como resultado
      o valor guardado no endereço A e com tamanho n*)

END Exceptions.
```

Observamos que Replace tem dois parâmetros descrevendo o resultado, para permitir a passagem de um valor de tipo arbitrário.

Ilustramos agora a programação de exceções através dos exemplos de Yemini e Berry, traduzidos para Modula-2.

(a) uma operação asinaladora - convert

```
MODULE Abstraction;

  FROM Exceptions IMPORT ENVIRONMENT, RegisterException;
  FROM Strings IMPORT STRING, AssignNullString, ExtendString;
  EXPORT convert;

  TYPE Tratador=PROCEDURE (ENVIRONMENT,CARDINAL):CHAR;
  ...

  PROCEDURE convert (VAR code:ARRAY OF CARDINAL;
                     tr:Tratador):STRING;
  VAR i : CARDINAL;
      s : STRING;
      e : ENVIRONMENT;
  BEGIN
     RegisterException(e);      (*inicializar a exceção*)
     AssignNullString(s);
     FOR i:=0 TO HIGH(code) DO
        IF code[i] <= maxchar AND code[i] >= minchar
           THEN ExtendString(s,CHAR(code[i]))
        ELSE ExtendString(s,tr(e,code[i]))   (*assinalar*)
        END (*IF*)                           (*a exceção*)
     END (*FOR*);
     RETURN s
  END convert;
  ...

END Abstraction.
```

OBS: A variável local e è usada para guardar a informação sobre o ambiente do usuário de convert. E obrigatória a chamada de RegisterException antes de assinalar a exceção.

(b) a associação de tratadores com exceções

```
MODULE user;

  FROM SYSTEM IMPORT ADR, SIZE;
  FROM Exceptions IMPORT ENVIRONMENT, Replace;
  FROM Strings IMPORT STRING, AssignNullString;
  IMPORT convert;
  ...
  (*bad1 prossegue execução do assinalador da exceção badcode*)
  PROCEDURE bad1 (e:ENVIRONMENT; c:CARDINAL):CHAR;
  BEGIN RETURN "?" END bad1;

(*bad2 termina execução do assinalador da exceção badcode*)
  PROCEDURE bad2 (e:ENVIRONMENT; c:CARDINAL):CHAR;
  VAR s:STRING;
  BEGIN
     AssignNullString(s);
     Replace (e,ADR(s),SIZE(s))
  END bad2;

  ...
  PROCEDURE user1;
  VAR nums:ARRAY [1..20] OF CARDINAL;
  BEGIN
     ...
     convert(nums,bad1);    (*operação com tratador bad1*)
     ...
     convert(nums,bad2);    (*operação com tratador bad2*)
     ...
  END user1;
  ...

END user.
```

OBS: Mostramos aqui duas chamadas de convert, especificando tratadores diferentes. O tratador bad1 ilustra o retorno ao assinalador, e bad2 ilustra o retorno ao usuário do assinalador.

(c) propagação de uma exceção

```
MODULE Propagator;

  IMPORT convert;
  FROM Strings IMPORT PrintString;
  FROM Exceptions IMPORT ENVIRONMENT, RegisterException;
  EXPORT CallConvert;
  TYPE Tratador1 = PROCEDURE(ENVIRONMENT):CHAR
  VAR finish = RECORD                  (*variável global para *)
                 trat:Tratador1;       (*comunicar entre*)
                 e:ENVIRONMENT         (*CallConvert e sua*)
               END;                    (*rotina tratadora*)
```

V - Conclusões

O estudo descrito aqui apresenta propostas para alguns dos problemas associados ao uso de Modula-2 em sistemas distribuidos. Sugerimos um modelo para programação concorrente baseado em processos e monitores, incluindo o uso de chamadas remotas de procedimento, e outro modelo para o tratamento de exceções. Estes modelos seriam implementados na forma de módulos que estariam carregados em cada estação do sistema distribuido. Todos estes módulos, exceto aquele que dá suporte para o tratamento de exceções, podem ser escritos inteiramente em Modula-2.

As facilidades exportadas por estes módulos poderiam ser utilizadas através da importação das suas interfaces em caso de isto se fazer necessário. Deste modo a solução proposta não só se baseou no suporte explícito a modularidade provido por Modula-2 como também estendeu os conceitos disponíveis ao programador de forma modular.

Acrescentando a estas extensões a proposta da linguagem de configuração apresentada em [Segre 1985] chegamos a definir um ambiente adequado para o desenvolvimento de software distribuido em Modula-2.

Embora as extensões propostas a Modula-2 neste trabalho sejam quase independentes, existem alguns aspectos da sua interação a serem analisados mais profundamente; em particular o tratamento de falhas em ambientes paralelos e distribuidos.

A pesquisa descrita aqui faz parte de um projeto maior sobre ambientes de desenvolvimento modular de software distribuido, e será dado prosseguimento na direção apontada no parágrafo anterior.

****

Os autores agradecem o apoio dado a esta pesquisa pela FINEP, EMBRATEL e CNPq.

Bibliografia

[Birrell 1984] Birrell,A.D., Nelson,B.J., "Implementing Remote Procedure Calls", ACM Trans. Comp. Sys. 2, 1 (Fev. 1984).

[Brinch Hansen 1975] Brinch Hansen,P., "The programming language Pascal Concurrent", IEEE Trans. Soft. Eng. SE-1, 2, p.199-206 (1975).

[De Renner 1976] De Renner,F. e Kron,H.H., "Programming-in-the-

```
PROCEDURE TratadorBadcode (env:ENVIRONMENT; c:CARDINAL):CHAR;
BEGIN
   WITH finish DO
     RETURN trat (e)            (*propaga a exceção tratada*)
   END (*WITH*)
END TratadorBadcode;

PROCEDURE CallConvert (tr1:Tratador1);
VAR nums:ARRAY [1..20] OF CARDINAL;
BEGIN
   WITH finish DO               (*guardar o tratador e*)
        trat:=tr1;              (*o ambiente de retorno*)
        RegisterException(e)    (*numa variável global*)
   END (*WITH*);
   ...
   PrintString(convert(nums,TratadorBadcode));
   ...
END CallConvert;
...

END Propagator.
```

OBS: TratadorBadcode é associado a CallConvert por ter sido citado na chamada que este último faz a convert. Para permitir que TratadorBadcode assinale a exceção tr1, passada como parâmetro para CallConvert, precisamos usar a variável global finish, onde será guardado também o resultado da chamada de RegisterException.

Finalmente apresentamos um esboço da implementação do módulo Exceptions, que provê o suporte para o uso do modelo de substituição descrito aqui. Esta requer manuseio explícito da pilha de registros de ativação de procedimentos para poder implementar a operação Replace.

O tipo ENVIRONMENT representa um registro de ativação (R.A.), e a primitiva RegisterException devolve o R.A. do procedimento que o chama (a operação assinaladora). O R.A. por sua vez contém ponteiros para o R.A. e o endereço de retorno do seu usuário. A primitiva Replace recebe, como parâmetro, o R.A. da operação assinaladora e o valor a ser retornado e devolve controle e este resultado ao usuário da operação.

Evidentemente, a implementação destas primitivas depende da implementação de Modula-2 e, em geral, elas deverão ser programadas em linguagem de máquina.

No caso específico das implementações de Modula-2 baseados no compilador M2M [MRI 1983, Rodriguez 1985], a linguagem foi estendida para incluir procedimentos codificados em código-M. Nestas duas implementações é fácil acrescentar uma nova instrução do código-M, que seria a maneira mais simples de implementar a primitiva Replace.

large versus programming-in-the-small", IEEE Trans.Soft.Eng. SE-2,2 , p.80-86 (1976).

[Dijkstra 1968] Dijkstra,E.W., "Cooperating Sequential Processes", em F.Genuys(ed.), Programming Languages, Academic Press, New York, p.43-112 (1968).

[DIN 1980] DIN (Deutsches Institut fuer Normung), Programmiersprache PEARL-Full PEARL (1980).

[DOD 1979] DoD (U.S. Department of Defense) "Preliminary ADA reference manual", ACM Sigplan Notices 14, 6 PART A (1979).

[Gentleman 1981] Gentleman,W.M., "Message passing between sequential processes: the reply primitive and administrator concept", Soft.Pract.Exper. 11, p.435-466 (1981).

[Hoare 1974] Hoare,C.A.R., "Monitors: An operating system structuring concept", Comm. ACM 17, 10, p.549-557 (Out.1974).

[Holt 1978] Holt,R.C. et al., "Structured Concurrent Programming with Operating Systems Applications", Addison-Wesley, London, 262 p. (1978).

[Lauer 1979] Lauer,H.C., Needham,R.M., "On the duality of operating system structures", ACM Operating Systems Review 13, 2, p.3-19 (Abril 1979).

[Loques 1985] Loques Fo.,O.G.,Souza Fo.,V.S.R., "Uma arquitetura de software para sistemas distribuidos", submetido para apresentação no V Simpòsio sobre Desenvolvimento de Software Bàsico, a se realizar em Belo Horizonte (Nov.1985).

[Mitchell 1979] Mitchell J.G. et al., "Mesa language manual, version 5.0", Report CSL-79-3, Xerox PARC (1979).

[MRI 1983] Bingham,L. et al. "Modula-2 Handbook", Modula Research Institute, Provo, Utah (1983).

[Nelson 1981] Nelson,B.J., "Remote Procedure Call", Report CSL 81-9, Xerox PARC (1981).

[Rodriguez 1985] Rodriguez,N.R., Stanton,M.A., "A implementação de Modula-2 em microcomputadores nacionais - um relatòrio de progresso", submetido para apresentação no V Simpòsio sobre Desenvolvimento de Software Bàsico, Belo Horizonte (Nov. 1985).

[Segre 1984a] Segre,L.M., Stanton,M.A., "A linguagem Modula-2 e seu ambiente de suporte de programação", Anais do XVII Congresso Nacional de Informàtica, Rio de Janeiro (Nov.1984).

[Segre 1984b] Segre,L.M., Stanton,M.A., "Modula-2: suporte para o

desenvolvimento de software concorrente", Anais do XVII Congresso Nacional de Informàtica, Rio de Janeiro (Nov.1984).

[Segre 1985] Segre,L.M., Stanton,M.A., "Uma linguagem de configuração para o uso de Modula-2 para programação em ambientes distribuidos", submetido para apresentação no V Simpòsio sobre Desenvolvimento de Software Bàsico, a se realizar em Belo Horizonte (Nov.1985).

[Sloman 1984] Sloman,M. et al., "Building flexible distributed systems in Conic", Proc. SERC Distributed Computing 84 Conference, Bringhton (Set. 1984).

[van Wijngaarden 1975] van Wijingaarden,A. et al, "Revised report on the algorithmic language Algol 68", Acta Informatica, 5 (1975).

[Wirth 1977] Wirth,N., "Modula: a Language for Modular Multiprogramming", Soft.Pract.Exper.7, p.3-35, (1977).

[Wirth 1981] Wirth,N., "The personal computer Lilith", in A.I.Wassermann (ed.), Software Development Environments, IEEE Computer Society Press (1981).

[Wirth 1982] Wirth,N., Programming in Modula-2, Springer-Verlag, New York (1982).

[Yemini 1985] Yemini,S., Berry,D.M., "A modular verifiable exception-handling mechanism", ACM Trans.Prog.Lang.Syst. 7, 2 , p. 214-243, (Abril 1985).