UMA MÁQUINA VIRTUAL PARA ADA NUM SISTEMA COM MÚLTIPLOS PROCESSADORES

Este exemplar corresponde a redação final da tese defendida pelo Sr. OSVALDO ALVES DOS SANTOS e aprovada pela Comissão Julgadora.

Campinas, 12 de agosto de 1985

Prof.Dr. TOMASZ KOWALTOWSKI
Orientador

Dissertação apresentada ao Instituto de Matemática, Estatística e Ciência da Computação, UNICAMP, como requisito parcial para obtenção do Título de Mestre em Ciência da Computação.

ÍNDICE

AGRADECIMENTOS

Ao prof. Dr. Tomasz Kowaltowski, meu orientador, e aos demais professores e colegas do IMECC que me incentivaram na elaboração deste trabalho.

À Creusa M. D. Cavalini e as funcionárias da DPG/FUEM que datilografaram o texto manuscrito, com a maior paciência.

À FUEM, CNPq e CAPES pelo apoio financeiro.

ABSTRACT

Ada is a general purpose programming language, which includes features found in classical languages such Pascal and Algol, as well as feactures for modular programming system and real-time programming, which are usualy found only in specific languages.

We present in this thesis a proposal for a run - time system for Ada. Due to the complexity of the language, we treat only the concurrency and exception handling. We assume that the underlying computing system allows for multiple processors.

I - INTRODUÇÃO

A definição da linguagem de programação Ada foi desenvolvida pela Cii Honeywell Bull subsidiada pelo United States Department of Defense. A primeira versão foi proposta em 1979 [05] e [06]. Em 1983 foi proposta uma nova versão da definição[01], caracterizada principalmente pela remoção de chamadas de subprogramas externas às tarefas e da ativação explícita de tarefas.

O objetivo desta linguagem é cobrir uma ampla gama de aplicações, desde as facilidades oferecidas pelas clássicas linguagens de programação, tais como Pascal, bem como as facilidades geralmente encontradas somente em linguagens especializadas. Assim, a linguagem é uma moderna linguagem algorítmica com o usual controle de estruturas e com as facilidades de definir tipos e subprogramas. Ela também supre as necessidades de modularidade, sempre que dados, tipos e subprogramas possam ser agrupados. Ela trata a modularidade no seu sentido físico bem como, a facilidade de compilação em separado. Além destes aspectos, a linguagem permite programação em tempo real através de tarefas paralelas e exceções. Ela também permite programação de sistemas, proporcionando um controle preciso sobre a representação de dados e as propriedades dependentes do sistema. Finalmente, está definido na linguagem entrada e saída em alto nível e a nível de máquina.

Compiladores são basicamente programas que têm como entrada um programa em uma determinada linguagem de alto nível e produzem como saída o programa em linguagem de montagem ou em linguagem de máquina. Além de traduzir programas, os compiladores geralmente detectam determinados tipos de erros e tentam otimizar o código gerado.

No caso de certas linguagens, a compilação de um programa é uma tarefa muito complexa, se for considerada em um único passo. Por esta razão, o processo de compilação, geralmente, é dividido em diversas fases. A primeira fase, chamada análise léxica ("scanner"), tem por objetivo localizar os átomos do programa fonte. A saída do analisador léxico é um fluxo de átomos, que é passado para a próxima fase. A análise sintática ("parser"), tem por finalidade agrupar os átomos em estruturas sintáticas. Geralmente, a estrutura sintática pode ser considerada como uma árvore cujos nós internos representam as construções da linguagem e as folhas os átomos. A fase de análise semântica é caracterizada por gerar um fluxo de instruções simples a partir da estrutura produzida pelo analisador sintático, que poderá ser um código intermediário ou de montagem. A análise semântica verifica também a ocorrência de certos tipos de erros, tais como, compatibilidade de tipos, omissão de declarações, etc. Existem diversos estilos de código intermediário. As vantagens de gerar código intermediário em vez de montagem são principalmente a portabilidade e simplicidade; a principal desvantagem é a necessidade de uma nova fase (geração de código), tornando o processo de compilação mais lento. Alguns compiladores possuem uma fase que tenta aplicar transformações no código gerado na tentativa de produzir uma versão do programa menor e/ou mais rápida; esta fase é popularmente chamada de fase de otimização. Os programas objeto que são fre-

quentemente executados deveriam ser otimizados. A quantidade de fases e passos de um compilador, dependerão basicamente da aplicação do compilador e do "hardware" disponível.

Uma linguagem intermediária pode ser vista como a linguagem de uma máquina virtual, comumente chamada de sistema de execução para a linguagem ("run-time system"). Neste trabalho é apresentado uma máquina virtual para a linguagem Ada, independente da máquina objeto e com características de portabilidade. Para tal, foi proposta uma máquina virtual, chamada máquina-Ada. O estilo de linguagem intermediária adotado compreenderá um mnemônico da instrução contendo ou não um pequeno número de parâmetros. Este estilo foi adotado pela simplicidade de tradução, onde cada instrução corresponderá a uma macro instrução no sistema hospedeiro.

Note que, para a implementação real do compilador Ada faz-se necessário a criação de um ambiente, consequentemente necessitando de um sistema operacional específico.

A literatura que serviu como base para o trabalho, foi [01], que contêm a nova versão da definição da linguagem Ada. Em [02] encontra-se uma proposta de um sistema de execução para a linguagem de programação Ada, baseada em [05] e [06]. As diferenças entre [01], [05] e [06] são caracterizadas principalmente pela remoção de chamadas de subprogramas externas às tarefas e, da ativação explícita de tarefas. Um outro ponto a ser considerado é que em [02] a proposta é para um sistema hospedeiro com um único processador, e na proposta deste trabalho é considerado que o sistema hospedeiro poderá possuir um ou mais processadores. Ainda, sobre máquina virtual foram consultadas as obras [03]e[04]. Sobre sistema de execução e linguagens intermediárias foram consultadas as obras [07] e [08]. As obras [09], [10], [11] e [12]

foram consultadas para resolver os problemas de comunicação entre tarefas.

Este trabalho foi dividido em quatro capítulos e dois apêndices. No capítulo II é descrita a notação adotada nas montagens das construções da linguagem e na definição das instruções. No capítulo III é apresentado o sistema de execução para Ada, onde foi definida a estrutura da máquina virtual e o seu conjunto de instruções. Este capítulo também contém as sugestões de como implementar as diversas construções da linguagem. No apêndice I são definidas as diversas instruções usadas na parte concorrente da linguagem. Devido ao grande número de abreviações foi incluído no trabalho o apêndice II, que contém um dicionário das abreviações usadas no capítulo III e apêndice I, com referências.

II - NOTAÇÃO

Neste capítulo, é descrita a notação usada para montar as construções da linguagem e as instruções.

As construções da linguagem serão compostas de instruções e partes que dependerão da parte sequencial da linguagem e/ou que são tratadas em detalhes, oportunamente, no texto; estas partes serão apresentadas, especificando informalmente o código que deverá ser gerado.

As instruções serão compostas por um identificador contendo quatro letras que corresponderá ao mnemônico da instrução. Uma instrução poderá ser rotulada ou não. Os rótulos iniciarão com a letra R, seguido de letras e/ou dígitos e seguido de dois pontos. Ainda, as instruções poderão possuir parâmetros ou não, se possuir mais de um serão separados por vírgula. Nas instruções, uma frase após o carácter "/", até o final da linha, corresponderá a comentários, não tendo significado na montagem da instrução.

As instruções, como pode ser observado no apêndice II, são apresentadas na forma de algorítmos em português, com algumas características da linguagem Pascal.

Algumas instruções possuem operações não determinísticas, tais como: $\stackrel{?}{=}$ e $:\stackrel{?}{=}$. O operador $\stackrel{?}{=}$ é um operador booleano que significa que se o resultado da operação for falso, o resultado

será falso; caso contrário, se o resultado da operação for verdadeiro, o resultado será não determinístico, isto é, poderá ser verdadeiro ou falso. O operador $:=^{?}$ é uma atribuição não determinística, isto é, após a operação de atribuição, a variável do lado esquerdo do operador poderá ou não possuir o resultado da expressão especificada do lado direito; se não ocorrer a atribuição o valor da variável não será alterado.

Para referir-se a um endereço na memória de pilhas, é usado um registrador que aponta para a base ou para uma determinada posição na pilha, mais uma expressão que fornece o deslocamento. Algumas vezes, este deslocamento será dado através de um nome de campo do RA, que numa implementação real seria uma constante.

III - SISTEMA DE EXECUÇÃO

1. Introdução

Neste capítulo está descrito o sistema de execução proposto para Ada. Na secção 2 é apresentada a estrutura geral da máquina-Ada. Nessa secção ainda é relacionado um conjunto de rotinas que são partes do ambiente para o compilador. Na secção 3 é apresentada a proposta de como implementar as diversas construções da parte concorrente da linguagem. Na secção 4 são discutidas as implementações das exceções. Na secção 5 são discutidas algumas possibilidades de ocorrências de bloqueio, que poderão ser causadas devido a erros de programação.

2. Características Gerais da Máquina - Ada.

Nesta secção será descrita a estrutura básica e o funcionamento da máquina - Ada.

2.1. Organização da Memória

A memória da máquina - Ada será composta de três regiões distintas:

- Registradores especiais
- Memória de dados
- Memória de programas

Registradores Especiais

A seguir, será apresentado um conjunto básico de registradores eseciais:

CP - Contador de programa:

Conterá o endereço da próxima instrução para ser executada.

AFTP - Apontador para a fila de tarefas prontas:

Apontará para um vetor (AFTP (0 .. max)), que será usado como apontadores para as filas de tarefas prontas, de acordo com a prioridade.

ALTD - Apontador para a lista de tarefas dormentes.

IHD - Indicador da hora de despertar.

REL - Relógio de tempo real.

ARUC - Apontador para o registro de ativação (RA) da unidade corrente.

ARTC - Apontador para o RA da tarefa corrente.

AP - Apontador para o topo da pilha.

O conjunto de registradores relacionados a seguir, será um dos requisitos para melhorar o desempenho da máquina - Ada:

- Um conjunto de registradores para armazenar o vetor de reistradores de base.
- Registradores para executar operações no topo da pilha.

- Registradores de propósito geral que serão usados como temporários pelo conjunto de instruções da máquina - Ada.

Memória de Dados

A memória de dados será dividida, de maneira geral, em duas partes: memória de pilhas das tarefas e memória de variáveis dinâmicas.

Considerando as características da linguagem, tais como: natureza recursiva e estrutura de bloco, torna-se adequada uma estrutura de pilha para armazenar os dados. Devido ao potencial de multitarefas dos programas em Ada, a estrutura adotada será de multipilhas ("cactus stack"), onde cada pilha corresponde a uma tarefa. Ver Figura 1.

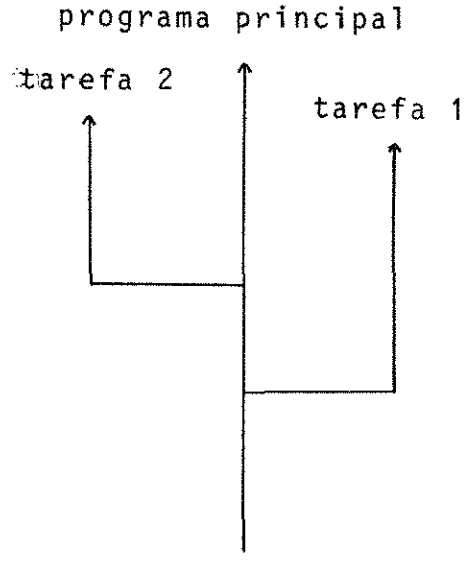


Figura 1: Memória de pilhas.

A pilha de uma tarefa T estará ligada a pilha de uma

tarefa S, que contém uma unidade U que ativou T, através do descritor de T residente no RA de U.

Nas pilhas serão armazenados os RA's das unidades que forem executadas. Opcionalmente, também poderão ser armazenados valores temporários de resultados de expressões aritméticas e das instruções da máquina - Ada.

As palavras de memória de cada pilha serão endereçadas linearmente, a partir de zero. Numa implementação real deverão ser levados em conta os limites de tamanho das pilhas. Se estes limites forem ultrapassados, deverá ser levantada a exceção "STORAGE-OVERFLOW", na unidade corrente.

A memória de variáveis dinâmicas será usada para armazenar objetos criados pelo alocador *new* . Sempre que for executado o alocador *new* , uma certa quantidade de memória será alocada para um objeto acessível por um determinado tipo de acesso. Estas áreas desaparecerão quando estes objetos tornarem-se inacessíveis. Para que estas áreas possam ser reutilizadas, poderá ser executado um algorítmo que tenderá recuperar esses espaços ("garbage collector").

Memória de Programas

Nesta região serão armazenados os códigos dos programas a serem executados. A memória de programas será linear; as instruções serão armazenadas a partir do endereço zero. Suporemos, por razões de exposição, que o tamanho dessa região será suficientemente grande para armazenar qualquer programa. O formato exato das instruções é irrelevante para esta discussão.

Desde que um programa esteja carregado/ligado, e os registradores devidamente inicializados, as instruções indicadas pe-

lo CP serão executadas até que seja encontrada uma instrução de parada.

No início de cada instrução, o valor do CP será incrementado assim, as instruções serão executadas sequencialmente, a menos que ocorra um desvio. As tarefas serão executadas até que ocorra uma interrupção ou seja terminada.

As instruções da máquina - Ada poderão ser interrompidas e a execução da instrução, quando a tarefa for reiniciada, deverá continuar a partir do ponto que ela foi interrompida.

2.2. Registros de Ativação

A implementação das unidades de programa estará baseada em registros de ativação, que guardarão os contextos das unidades. A implementação de chamada de entrada também necessitará de um RA simples para executar o comando *accept*. Os diversos RA's podem ser vistos na figura 2. Serão conhecidas as bases dos RA's acessíveis em um determinado contexto e o acesso às informações dos RA's serão relativas as suas bases. Para facilitar o endereçamento aos diversos campos dos RA's, suporemos que os campos terão tamanho fixo e múltiplos de uma palavra.

A seguir, serão descritos os campos que farão parte dos diversos RA's. Primeiramente, serão descritos os campos que são comuns a todos os RA's:

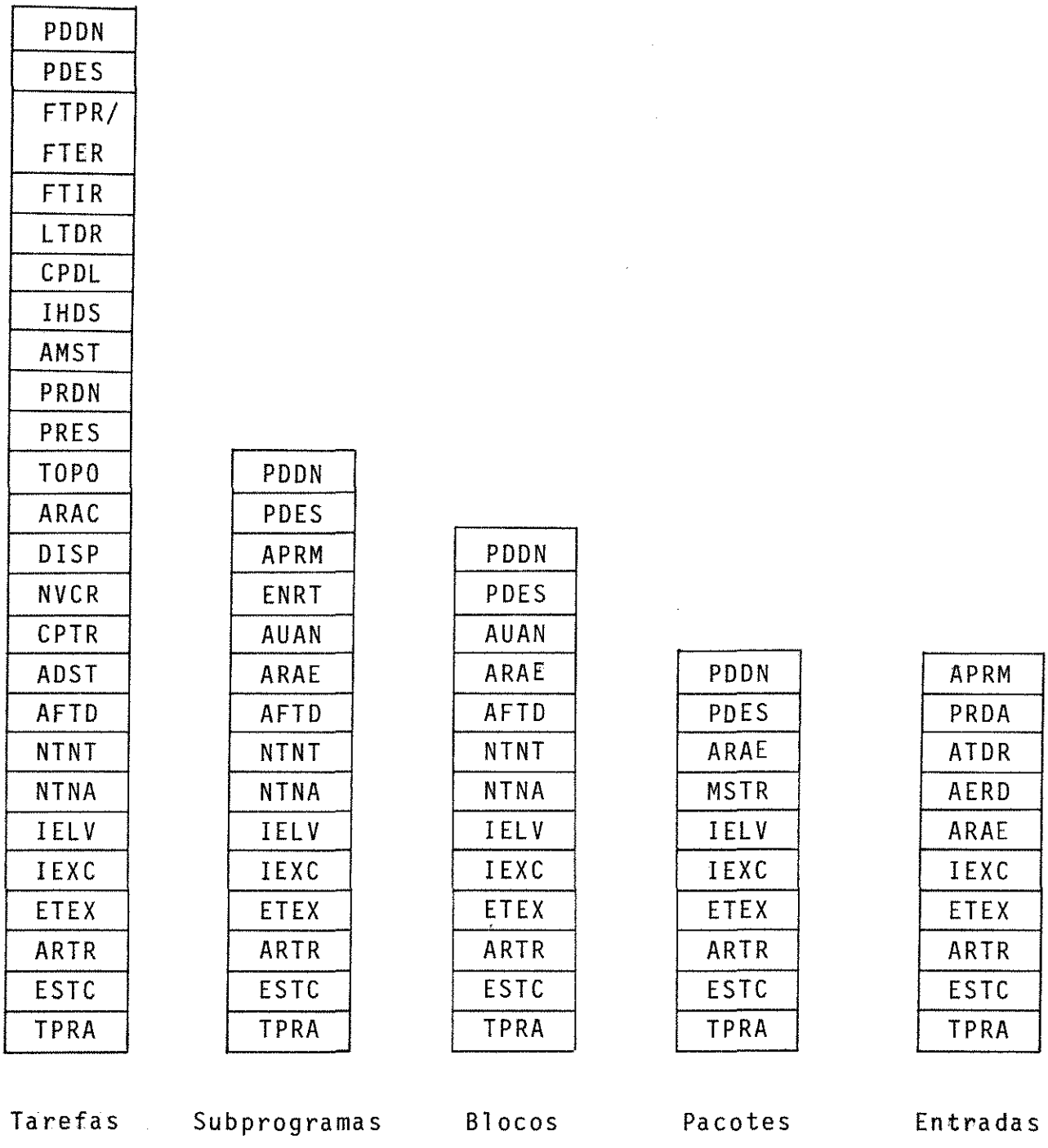


Figura 2: Registros de ativação.

TPRA - Tipo do RA:

- tarefas: TR
- subprogramas: SP
- blocos: BL
- pacotes: PC
- entradas: EN

Usado para identificar o RA, quando não for conhecido o contexto. Isto ocorre quando estão sendo marcadas como "anormal" as tarefas descendentes de uma tarefa abortada.

ESTC - Estado da tarefa no contexto:

Usado para restaurar o estado da tarefa no momento do retorno da execução para um determinado contexto.

ARTR - Apontador para o RA da tarefa:

Será usado para ter acesso ao RA da tarefa da qual este contexto faz parte. Se o contexto for de uma tarefa, este campo apontará para o seu próprio RA.

ETEX - Endereço do tratador de exceção:

Será usado principalmente para levantar/propagar exceções em unidades que não fazem parte do contexto corrente.

IEXC - Índice da exceção:

Será usado pelo tratador de exceções para tratar a exceção levantada.

A seguir serão descritos os campos que são comuns aos RA's das unidades de programa.

IELV - Indicador de exceção levantada:

Este campo será usado para indicar que existe uma exceção levantada.

PDES - Parte de declarações estáticas:

É a área onde serão armazenados os objetos cujas declarações são do tipo estático, isto é, inteiramente conhecidos em tempo de compilação e, apontadores para os objetos cujas declarações são de tipos dinâmicos. Objetos de tipo dinâmico serão conhecidos inteiramente somente em tempo de execução. Nesta área também serão armazenados os descritores das entradas (Figura 3), das famílias de entradas (descritores de vetores de descritores de entradas - Figura 4), descritores de tarefas (Figura 5) e os descritores de famílias de tarefas (descritores de vetores de descritores de tarefas - Figura 6). O tamanho da área PDES dependerá do número e do tipo das declarações das unidades.

PDDN - Parte de declarações dinâmicas:

Esta é a área onde serão armazenados os objetos cujos tipos são dinâmicos. O início desta área será variável, sendo conhecido somente após a compilação da parte declarativa da unidade.

A seguir, serão descritos os campos que fazem parte dos RA's de tarefas, subprogramas e blocos:

NTNA - Número de tarefas não ativadas:

Conterá um valor inteiro não negativo que, indicará o número de tarefas que dependem da unidade e estão em ativação.

NTNT - Número de tarefas dependentes da unidade, não terminadas:

Conterá um valor inteiro não negativo que, indicará o número de tarefas ativadas dependentes da unidade.

AFTD - Apontador para a fila de tarefas dependentes:

Apontará para o RA da primeira tarefa da fila de tarefas que dependem da unidade

A seguir serão descritos os campos que fazem parte somente do RA de tarefas:

ADST - Apontador para o descritor da tarefa:

CPTR - Contador de programa da tarefa:

Indicará o endereço do código da tarefa onde deverá reiniciar a execução, exceto quando for reativada devido a temporização ("timeout").

NVCR - Nível corrente:

Armazenará o nível léxico corrente quando a tarefa for interrompida.

DISP - Vetor de registradores de base:

Armazenará o vetor de registradores de base quando a tarefa for interrompida (se for implementado).

ARAC - Apontador para o RA da unidade corrente:

Armazenará o apontador para o RA da unidade corrente quando a tarefa for interrompida. Este campo deverá existir so-

mente se o aninhamento estático das unidades for feito através do campo ARAE, em vez do vetor de registradores de base.

TOPO - Topo da pilha:

Salvará o registrador AP quando a tarefa for interrompida.

PRES - Prioridade estática.

PRDN - Prioridade dinâmica:

A prioridade dinâmica de uma tarefa poderá ser diferente da estática, durante a execução de um comando *accept*, se a tarefa chamadora tiver maior prioridade que a tarefa chamada.

AMST - Apontador para a unidade mestre:

Apontará para a unidade que através da qual esta tarefa foi criada.

IHDS - Indicador da hora de despertar:

Conterá um valor que indicará o momento que a tarefa deverá ser despertada, sempre que for suspensa por meio da execução de um comando *delay*.

CPDL - Contador de programa alternativo da tarefa:

Conterá o endereço onde deverá reiniciar a execução, se a tarefa for reativada devido a temporização.

LTDR - Lista de tarefas dormentes:

Usado para formar a lista das tarefas que estão "executando" um comando *delay*.

FTIR - Fila de tarefas irmãs:

Usado para formar a fila de tarefas que dependem da mesma unidade.

FTPR/FTER - Fila de tarefas prontas/ fila de tarefas esperando rendezvous:

Dependendo do estado, uma tarefa poderá estar na fila de tarefas prontas ou na fila de tarefas esperando rendezvous ou em nenhuma das filas. Essencialmente, uma tarefa nunca poderá estar simultaneamente nas duas filas assim, esse espaço físico poderá ser compartilhado pelos dois campos.

FTPR: conterá um apontador que será usado para formar a fila de tarefas prontas. A fila de tarefas prontas será usada pelo escalonador, para ter acesso e indicar a ordem em que deverão ser atendidos os pedidos de processamento.

FTER: conterá um apontador que será usado para formar a fila de tarefas esperando rendezvous que, será usada para ter acesso e indicar a ordem que deverão ser atendidos os pedidos para realizar rendezvous.

A seguir será descrito o campo que faz parte dos RA's de subprogramas, blocos, pacotes e entradas:

ARAE - Apontador para o RA da unidade antecessora estática:

Poderá ser usada como uma forma opcional para se ter acesso ao RA da unidade antecessora estática. A outra forma de acesso poderá ser através do vetor de registradores de base.

A seguir será descrito o campo que faz parte dos RA's

de subprogramas e blocos:

AUAN - Apontador para o RA da unidade antecessora dinâmica.

A seguir será descrito o campo que faz parte dos RA's de subprogramas e entradas:

APRM - Área de parâmetros:

É a região onde serão armazenados os parâmetros.

A seguir será descrito o campo que faz parte somente do RA de subprogramas:

ENRT - Endereço de retorno:

Indicará o endereço, na memória de programas, onde deverá reiniciar a execução após o término do subprograma.

A seguir será descrito o campo que faz parte somente dos RA's de pacotes:

MSTR - Apontador para o mestre:

Conterá um apontador para o RA da unidade que é mestre do pacote corrente.

A seguir serão descritos os demais campos que fazem parte somente dos RA's de entradas:

AERD - Apontador para o descritor da entrada através da qual está sendo realizado o rendezvous.

ATRD - Apontador para o RA da tarefa com o qual está sendo realizado o rendezvous.

PRDA - Prioridade dinâmica anterior:

Indicará a prioridade que a tarefa possuía antes da execução do comando *accept* corrente.

Campos Auxiliares

A seguir serão apresentados os descritores de entradas e os descritores de tarefas.

Os descritores de entradas possuirão quatro campos. O primeiro campo especificará o tipo do descritor, que poderá ser "DE" (descritor de entrada simples) ou "DFE" (descritor de família de entradas).

O descritor de uma entrada simples terá o seguinte formato:

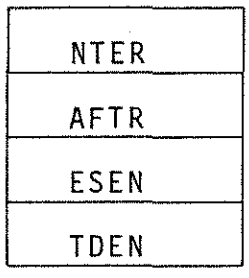


Figura 3: Descritor de entrada simples.

TDEN - Tipo do descritor de entrada.

ESEN - Estado da entrada:

Indicará se a tarefa está ou não esperando uma chamada para realizar rendezvous, através desta entrada (aberta ou fechada, respectivamente).

AFTR - Apontador para a fila de tarefas esperando rendezvous com

a entrada:

Apontará para a primeira tarefa da fila de tarefas esperando rendezvous com a tarefa, através desta entrada.

NTER - Número de tarefas esperando para realizar rendezvous com a tarefa, através desta entrada:

Usado principalmente pelo atributo COUNT.

O descritor de uma família de entradas terá o seguinte formato:

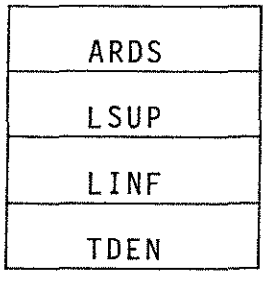


Figura 4 : Descritor de família de entradas.

TDEN - Tipo do descritor da entrada.

LINF - Valor do limite inferior do intervalo.

LSUP - Valor do limite superior do intervalo.

ARDS - Apontador para a região onde serão armazenados os descritores de membros da família de entradas.

Os descritores de membros de famílias de entradas são semelhantes aos descritores de entrada simples, exceto que, os descritores de membros de famílias de entrada não possuirão o campo TDEN.

Os descritores de tarefas poderão ter a forma de descritores de tarefas simples ou descritores de famílias de tarefas, de acordo com o tipo de declaração da tarefa.

O descritor de uma tarefa simples terá o seguinte formato:

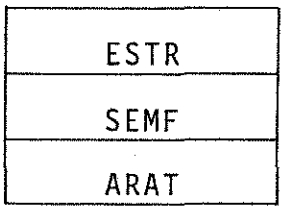


Figura 5: Descritor de tarefa simples.

ARAT - Apontador para o RA da tarefa.

SEMF - Semáfaro:

Alguns dos campos do descritor e do RA da tarefa deverão ser manipulados com exclusão mútua. O campo semáfaro indicará se estes campos estão liberados ou não (aberto ou fechado, respectivamente).

ESTR - Estado da tarefa:

Os estados que uma tarefa poderá ter, serão:

- em elaboração
- normal
- dormente
- unidade completada
- esperando rendezvous - ator
- esperando rendezvous - ator dormente
- esperando rendezvous - servidor
- esperando rendezvous - servidor dormente

- esperando rendezvous ou esperando término
- esperando término
- em rendezvous - ator
- em rendezvous - servidor
- anormal
- completado
- terminado

O descritor de uma família de tarefas terá o seguinte formato:

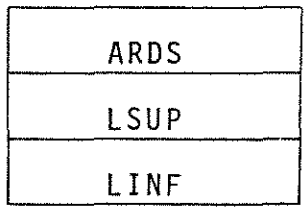


Figura 6: Descritor de família de tarefas.

O significado destes campos já foram apresentados em descritor de família de entradas .

O descritor de um membro de uma família de tarefas terá o mesmo formato do descritor de uma tarefa simples.

Por razões de exposição, os vetores de comprimento estático e dinâmico serão tratados da mesma forma, isto é, os vetores de comprimento estático serão tratados como dinâmicos. Note que esta forma de tratamento de vetores de tipo estático aumenta sensivelmente o tempo de execução.

2.3. Rotinas Auxiliares

As rotinas que farão o escalonamento de tarefas, que tratarão os conflitos de acesso à memória comum e que manipularão a lista de tarefas dormentes dependerão, em primeiro lugar

da arquitetura da máquina. Outras rotinas que manipularão filas e pilhas de tarefas dependerão do sistema de gerenciamento de memória e da estrutura de dados adotada.

Escalonamento de Tarefas

Tarefas paralelas (processadores lógicos paralelos) podem ser implementados em multicomputadores, multiprocessadores ou com entrelaçamento de um único processador.

Como possivelmente o número de processadores físico será menor do que o número de tarefas paralelas prontas para serem executadas em um dado instante, torna-se necessário algum meio de escalonar a execução das tarefas. A rotina que fará o escalonamento de tarefas para execução é chamada de "escalonador".

A política de compartilhamento dos recursos, adotada pelo escalonador, dependerá da implementação.

Sempre que, por qualquer motivo, for invocado o escalonador, deverá ser guardado o contexto da tarefa interrompida.

Sempre, antes do escalonador passar o controle do processador para uma tarefa, deverá ser restaurado o contexto da tarefa.

Conflitos de Acesso à Memória Comum

Os problemas que poderão ocorrer, durante a execução de comandos concorrentes, devem-se ao fato de que dois ou mais comandos, em tarefas distintas, podem tentar verificar e atualizar "simultaneamente" o mesmo campo ou RA de uma tarefa. Este é um problema clássico de regiões críticas.

Se no sistema hospedeiro existir um único processador, a maneira mais simples de resolver este problema é inibir, antes de iniciar a execução de uma região crítica, o conjunto de interrupções do processador que possibilitam rotinas manipular essa região crítica e, após a execução da região crítica habilitar essas interrupções.

No entanto, neste texto, estamos supondo que poderá existir mais de um processador no ambiente onde serão executados os programas. Neste caso, uma maneira de resolver o problema de controle de acesso é, estabelecer um único semáfaro aos campos do do RA e do descritor de uma tarefa. Então antes de iniciar a execução de uma região crítica que tem comandos que verificam ou atualizam campos, que podem ser manipulados por mais de uma tarefa "simultaneamente", do RA ou do descritor de uma tarefa T, onde T aponta para o RA dessa tarefa, deverá ser executada a rotina "entra na RC (DT)" (DT especificará o descritor da tarefa T), que verificará se a região crítica está livre ou ocupada. Se estiver livre, a rotina alterará o semáfaro para ocupado e iniciará a execução da região crítica, caso contrário, a tarefa será colocada em uma fila e esperará pela liberação da região crítica.

Os campos que deverão ser manipulados de maneira esclusiva são:

(a) Pertencentes à RA's de blocos, subprogramas e tarefas: NTNA, NTNT e AFTD.

(b) Pertencentes somente à RA's de tarefas: LTDR, FTIR, FTPR e FTER. Pertencentes aos descritores de entradas: ESEN, AFTR e NTER.

(c) Pertencentes aos descritores de tarefas: ESTR.

No final da região crítica deverá ser executada a rotina "sai da RC(DT)", que liberará o semáfaro do descritor apontado por DT.

As rotinas "entra na RC(DT)" e "sai da RC(DT)" poderão ser implementadas através das operações primitivas P e V de Dijkstra [09].

Tarefas Dormentes

Através da lista de tarefas dormentes estarão ligadas as tarefas que estiverem "executando" um comando *delay*. Deverão existir rotinas que verificarão o momento que uma tarefa deve ser acordada, por temporização . Estas rotinas dependerão do "hardware" disponível para esse fim. Existirão ainda as rotinas que farão a remoção de tarefas da lista de tarefas dormentes, independentemente de temporização e, as rotinas que farão a- inserção de tarefas na lista de tarefas dormentes, "retira - ALTD(T)" e "insere - ALTD(T)", respectivamente, onde T apontará para o RA da tarefa. Estas rotinas dependerão da estrutura de dados adotada para implementar a lista, que dependendo do número de tarefas na lista, poderá ser implementada através de lista ligada, "heaps" ou árvores.

Outras Filas de Tarefas

As outras filas de tarefas são: fila(s) de tarefas prontas, filas de tarefas esperando rendezvous e filas de tarefas irmãs. As rotinas que farão inserções e remoções da(s) fila(s) de tarefas prontas serão: "insere - AFTP(T)" e "retira - AFTP(T)", respectivamente; das filas de tarefas esperando ren-

dezvous serão:"insere - AFTR(E,T)" e "retira - AFTR (E,T)", respectivamente, onde E apontará para a entrada chamada e T apontará para o RA da tarefa chamadora. Estas rotinas dependerão da estrutura de dados que for usada para implementar as filas, que poderão ser circulares, com encadeamento simples ou com encadeamento duplo. A rotina "insere - AFTD(T)" fará a inserção de T na fila de tarefas irmãs de T (que serão dependentes da unidade corrente), a rotina "retira - AFTD(T)" fará a remoção de T da fila de tarefas irmãs de T (que serão dependentes da unidade corrente). A estrutura de dados adotada para implementar a fila de tarefas irmãs é circular com encadeamento simples.

Alocação de Pilhas

Como foi observado anteriormente, cada tarefa possuirá a sua própria pilha. A pilha de uma tarefa será alocada no momento da sua ativação, através da execução da rotina "aloca pilha", que devolverá o seu endereço. Quando uma tarefa é terminada, a execução da rotina "desaloca - pilha(T)" colocará a disposição do sistema a área da pilha, especificada pelo apontador T. Será suposto que a alocação de mais espaço, caso seja necessário, ficará a cargo do sistema. Maiores detalhes a respeito de como implementar estas rotinas também estão fora do objetivo deste trabalho.

Execução das Rotinas Auxiliares

De modo geral, as rotinas auxiliares poderão ser invocadas pelas tarefas ou pelo sistema operacional. Para que não ocorram conflitos de acessso a objetos compartilhados, deverá ser

elaborado um sistema de controle de acesso através de semáfaros. Estes semáfaros poderão controlar o acesso às rotinas ou aos objetos. No entanto, uma vez iniciada a execução de uma destas rotinas ela deverá ser completada sem interrupções, para que o objeto não permaneça inacessível por muito tempo. Uma maneira de resolver este problema é inibir determinados tipos de interrupções do processador corrente.

3. Comandos Concorrentes

Nesta secção será descrito como poderão ser implementados os comandos concorrentes da linguagem, através do sistema de execução proposto. Os comandos concorrentes são também algumas vezes chamados de comandos de sincronização, desde que de alguma forma, eles envolvem sincronismo entre tarefas. Os pontos de sincronização são aqueles em que uma tarefa necessita atualizar o seu estado ou o estado de uma outra tarefa. Os pontos de sincronização são: o fim da ativação de uma tarefa, o ponto onde uma tarefa causa a ativação de outra tarefa, uma chamada de entrada, o início ou o fim de um comando *accept*, um comando *select*, um comando *delay*, um tratador de exceções, ou um comando *abort* . Nos pontos de sincronização as tarefas mudam de estado. Uma tarefa que estiver no estado "anormal" não poderá mudar para outro estado, exceto para o estado "terminado". Então, sempre que uma tarefa tiver que mudar de estado, deverá ser verificado se o seu estado é "anormal". Se for, o fluxo de execução do programa deverá ser desviado para o código que desativa a tarefa. Para maiores detalhes, vide o comando *abort*.

3.1. Ativação e Término de Tarefa

Uma tarefa consiste da parte de especificação e do corpo. Uma especificação de tarefa que inicia com as palavras reservadas *task type* declara um tipo tarefa. Uma especificação de tarefa que inicia sem a palavra reservada *type* define uma simples tarefa. Por razões de exposição, os dois tipos serão tratados genericamente, da mesma forma.

Suponhamos que o trecho de programa fonte a seguir faça parte da região declarativa de uma unidade U:

```
.
.
.
task type T is
.
.
.
end T;
.
.
.
task body T is
.
.
.
end T;
.
.
.
```

O código que deverá ser gerado para este trecho de programa, deverá ter a seguinte forma:

```
.
.   } código para a parte de especificação e declarativa de T
.
```

. } código para os comandos residentes no corpo de T
.

. } código para o término de T
.

O código que deverá ser gerado para a região de especificação e declarativa de T, deverá ter a seguinte forma:

```
R1:  DCTR RTE / cria o descritor e o RA para um objeto do tipo T
       .
       .  } código para a parte de especificação de T
       .
     INDP     / insere um objeto do tipo T na fila de tarefas
              / dependentes
       .
       .  } código para a parte declarativa de T
       .
     ATDP  R0 / ativa os dependentes do objeto do tipo T
     ATMS  R0 / ativa o mestre do objeto do tipo T
```

A instrução DCTR suporá que no topo da pilha da unidade corrente conterá o endereço onde deverá ser criado o descritor da tarefa. Esta instrução criará o descritor e alocará a pilha para uma tarefa. O rótulo R1 será usado para ter acesso ao código para fazer a declaração de um objeto deste tipo. O parâmetro RTE especificará o endereço do tratador de exceções do tipo.

A instrução INDP suporá que no topo da pilha da unidade corrente conterá o endereço onde foi criado o descritor da tarefa (pela instrução DCTR) e o endereço da próxima instrução a ser executada. Esta instrução fará a inserção da tarefa na fila de tarefas dependentes da unidade corrente. Note que as exceções

que forem levantadas até este ponto, referentes a elaboração da parte declarativa deste tipo, serão tratados no tratador de exceções da unidade corrente.

A instrução ATDP desviará o controle para o endereço especificado pelo parâmetro R0, caso a tarefa corrente tenha sido abortada (a tarefa corrente é uma tarefa do tipo T e não da qual a unidade U faz parte); caso contrário, esta instrução ativará todos os dependentes (se houver) da tarefa corrente. Esta instrução especifica o ponto de sincronização "o ponto onde uma tarefa causa a ativação de outra tarefa".

A instrução ATMS desviará o controle para o endereço especificado pelo parâmetro R0, caso a tarefa corrente tenha sido abortada; caso contrário, atualizará o estado desta tarefa para "normal" e verificará a possibilidade de ativar a unidade mestre. Note que se uma exceção for levantada na elaboração da parte declarativa de uma tarefa, deverá ser levantada a exceção "TASKING ERROR" na respectiva unidade mestre. Esta instrução especifica o ponto de sincronização "a fim da ativação de uma tarefa".

O código que deverá ser gerado para fazer a declaração e ativação de uma tarefa, dependerá da forma que o objeto foi declarado.

Suponhamos que o trecho de programa fonte a seguir, que faz a declaração de uma simples tarefa, faça parte da região declarativa de U:

```
.
.
.
task T is
.
.
.
end T;
.
.
.
```

O código que deverá ser gerado para fazer a declaração da tarefa T, deverá ter a seguinte forma:

```
.
.
.
      CDTC  R3    /declaração de tarefa simples
      .
      .   } código para o tipo T
      .
      .
      .
R3:   .   } próxima instrução de U
      .
      .
```

A instrução CDTC reservará espaço no topo da pilha para o descritor de T e, carregará no topo da pilha o endereço da próxima instrução a ser executada após a ativação de T, especificado pelo parâmetro R3, e o endereço onde deverá ser criado o descritor de T.

Suponhamos o trecho de programa fonte a seguir:

```
.
.
.
task type T is
.
.
.
end T;
.
.
.
S:T;
.
.
.
```

O código que deverá ser gerado para fazer a declaração de S, deverá ter a seguinte forma:

```
.
.
.
.
.   } código para o tipo T
.
.
.
.
.
CDTV  R3    /declaração de variável do tipo tarefa
.
.
.
```

A instrução CDTV reservará espaço no topo da pilha para o descritor de S e carregará no topo da pilha o endereço da próxima instrução a ser executada após a ativação de S, especificado pelo CP, e o endereço onde deverá ser criado o descritor de S. O parâmetro R3 especificará o endereço do código do tipo T.

Suponhamos o trecho de programa fonte a seguir:

```
.
.
.
task type T is
.
.
.
end T;
.
.
.
S:array (1 .. 5 ) of T;
.
.
.
```

O código que deverá ser gerado para fazer a declaração da família de tarefas S, deverá ter a seguinte forma:

```
.
.
.
.  ⎫
.  ⎬ código para o tipo T
.  ⎭
.
.
.
.  ⎫ código para calcular o limite inferior do compri-
.  ⎬
.  ⎭ mento do vetor
.  ⎫ código para calcular o limite superior do compri-
.  ⎬
.  ⎭ mento do vetor
CDFT  DT   /cria o descritor do vetor
.  ⎫ código para declarar os membros da família de ta-
.  ⎬
.  ⎭ refas de S
```

A instrução CDFT criará no endereço especificado pelo parâmetro DT o descritor da família de tarefas de S.

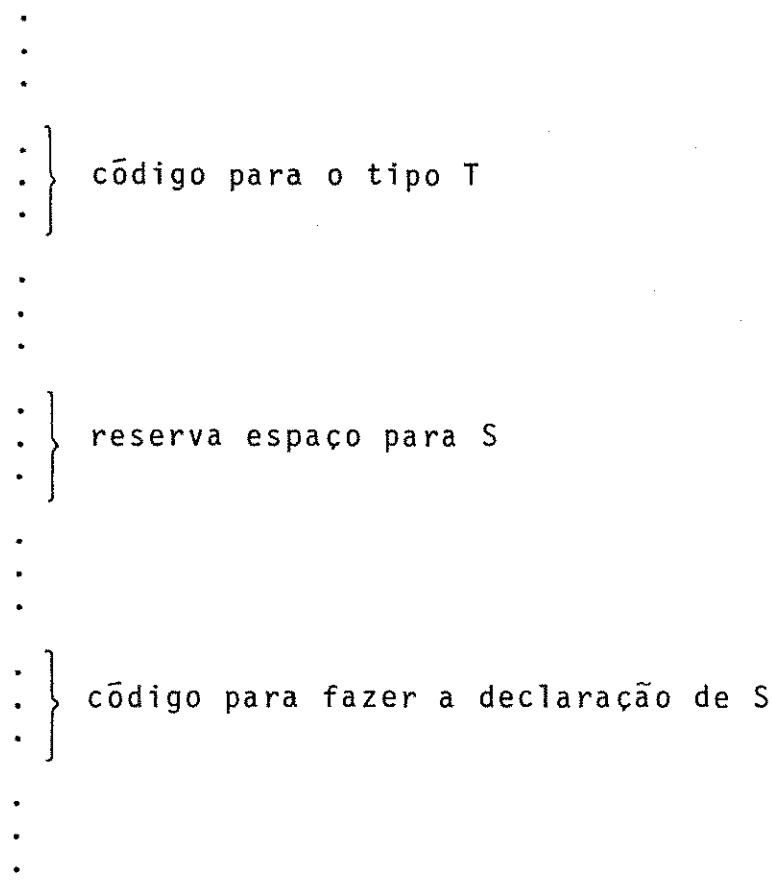
O código que deverá ser gerado para declarar os mem-

bros da família de tarefas de S será, executar a instrução DCTV o número de vezes indicado pelo comprimento do vetor.

Suponhamos o trecho de programa fonte a seguir:

```
.
.
.
task type T is
.
.
.
end T;
.
.
.
type APT is access T;
.
.
.
S:APT
.
.
.
S: = new T;
.
.
.
```

O código que deverá ser gerado para fazer a declaração da variável S do tipo de acesso APT, deverá ter a seguinte forma:



O código que deverá ser gerado para reservar espaço para S, será simplesmente uma instrução que reserva uma posição na pilha. O código para fazer a declaração de S será semelhante a instrução CDTV, exceto que neste caso deverá ser alocado espaço na memória de variáveis dinâmicas para o descritor de S e, esse endereço será armazenado ao topo da pilha (que será usado para indicar o endereço onde deverá ser criado o descritor de S) e, no espaço reservado para S (para posteriormente ter acesso ao descritor de S).

O código que deverá ser gerado para elaborar a parte de especificação de T, deverá ter a seguinte forma:

. } código para carregar descritores de entradas
.
.

. } código para carregar a prioridade de T
.
.

. } código para elaborar os pragmas
.
.

Suponhamos o trecho de programa fonte a seguir:

task T *is*

.
.
.

entry E ("lista de parâmetros formais");

.
.
.

O código que deverá ser gerado para declarar a entrada E, deverá ter a seguinte forma:

.
.
.

CDEN /carrega descritor de entrada

.
.
.

A instrução CDEN criará o descritor da entrada E no topo da pilha de T, inicializando devidamente cada campo do descritor.

Suponhamos o trecho de programa fonte a seguir:

```
task T is
.
.
.
entry E (1 .. 5) ("lista de parâmetros formais");
.
.
.
```

O trecho de programa exemplificado corresponde a declaração de uma família de entradas E. O código para carregar o descritor de família de entradas, deverá ter a seguinte forma:

```
.
.
.
.  }  código para calcular o limite inferior
.  }
.  }  código para calcular o limite superior
.  }
.  }
   CDFE  ED    /carrega descritores de entradas
.
.
.
```

A instrução CDFE criará o descritor da família de entradas E, no endereço especificado pelo parâmetro ED **e,** criará também o descritor para cada membro de E no topo de T, inicializando devidamente cada campo.

Se o programador especificar explicitamente o pragma PRIORITY, o compilador deverá gerar a instrução CRPR P, onde o parâmetro P indicará a prioridade estática da tarefa, caso contrário, a prioridade da tarefa será a "prioridade indefinida", já atribuída à tarefa, pela instrução DCTR.

A seguir está apresentado o código que deverá ser gerado para implementar o término de T:

```
CMTR   N  /completa T
VTUC      /verifica a possibilidade de término de T
TRTR      /término de T
```

A instrução CMTR atualizará o estado da tarefa para "completado" e levantará a exceção "TASKING ERROR" em todas as tarefas que estiverem nas filas das entradas, esperando para realizar rendezvous com T. O parâmetro N indicará o número de entradas da tarefa[i].

A instrução VTUC verificará, através dos descendentes da unidade, a possibilidade de término. Se for possível, isto é, se a unidade não possuir descendentes, ou se possuir, se todos eles estiverem num dos estados "esperando término" ou "esperando rendezvous ou esperando término", então será executado a próxima instrução. Caso contrário, a tarefa será suspensa e o contador de programa será atualizado para reexecutar esta instrução, quando a tarefa for reativada. Esta tarefa será reativada sempre que um dos seus descendentes passar para um dos estados: "terminado","esperando término" ou "esperando rendezvous ou esperando término".

A instrução TRTR atualizará o estado da tarefa para "terminado" e desalocará a pilha da tarefa.

3.2. Chamada de Entrada

Suponhamos o trecho de programa a seguir:

---

[i]: Para que N possa ser calculado em tempo de compilação, para efeito de contagem (e também usado na instrução CMTR), não será feito distinção entre descritores de entradas simples e de famílias de entradas.

```
task S is

.
.
.

entry  E ("lista de parâmetros formais");

.
.
.

task T is

.
.
.

task body T is

.
.
.

begin

.
.
.

S.E ("lista de parâmetros efetivos");

.
.
.

end T;
```

O código que deverá ser gerado para implementar a chamada da entrada E, deverá ter a seguinte forma:

```
    .
    .   } código para calcular o endereço de S
    .
    .
    .   } código para calcular o endereço de E
    .
    .
    .   } código para calcular os parâmetros efetivos
    .
CHEN  R0 /chamada de entrada
AEST  R0 /atualiza  o estado da tarefa T
    .
    .   } código para copiar de volta os parâmetros passa-
    .   } dos de modo out e inout
```

O código para calcular o endereço de uma tarefa que é membro de uma família de tarefas ou, de uma entrada que é membro de uma família de entradas, serão semelhantes ao cálculo do endereço de um elemento de um vetor.

Os parâmetros serão avaliados e carregados da mesma forma de subprogramas,exceto que os parâmetros, neste caso, não estarão armazenados no RA da unidade chamada, mas residirão no RA da unidade chamadora.

Verificou-se duas maneiras para resolver este problema. Uma delas seria copiar os parâmetros passados de modo *in* e *inout* para o RA da unidade chamada no início do rendezvous e, após o término do rendezvous copiar de volta, para o RA da unidade chamadora, os parâmetros passados de modo *out* e *inout*. A outra forma seria usar um registrador geral para indicar o endereço onde estariam armazenados os parâmetros, durante a realização do rendezvous. Embora com a desvantagem de duplicação de cópias de alguns parâmetros, neste trabalho adotou-se a primeira solução.

Na instrução CHEN é suposto que o endereço da entrada bem como da tarefa chamada estão armazenados no topo da pilha. A

instrução CHEN desviará o controle para o endereço especificado pelo parâmetro R0, caso a tarefa T tenha sido abortada; caso contrário, será verificado se a tarefa S está num dos estados que indicam término; se estiver, será levantada a exceção "TASKING ERROR" na tarefa T; caso contrário, finalmente será verificada a possibilidade de realizar a rendezvous imediatamente; se for possível, iniciar-se-á o rendezvous; caso contrário, a tarefa T será inserida na fila de tarefas esperando rendezvous, da entrada E.

A instrução AEST desviará o controle para o endereço especificado pelo parâmetro R0, caso a tarefa T tenha sido abordada; caso contrário, atualizará o estado da tarefa T, através do campo ESTC do RA da unidade corrente.

As instruções CHEN e AEST especificam o ponto de sincronização "chamada de uma entrada".

Deverá ser gerado código para copiar de volta, os parâmetros armazenados no topo de T, passados de modo *out* e *inout*, para as respectivas variáveis.

3.3. Comando *accept*

Suponhamos o trecho de programa fonte a seguir:

```
.
.
.
task T is
.
.
.
entry E ("lista de parâmetros formais");
.
.
.
task body T is
```

```
        .
        .
        .
        begin
        .
        .
        .
        accept E ("lista de parâmetros formais");
        .
        .
        .
        end E;
        .
        .
        .
        end T;
```

O código que deverá ser gerado para implementar o comando *accept* para a entrada E, deverá ter a seguinte forma:

```
        .
        .
        .
        .  ⎫
        .  ⎬  código para armazenar o endereço da entrada
        .  ⎭
      VREC  R0,R1/verifica se existe chamada para a entrada E
      ACCP  R0    /inicia o rendezvous
        .  ⎫
        .  ⎬  código para copiar os parâmetros passados de modo
        .  ⎭  in e inout
      FACC  R0    /termina o rendezvous
      DSVS  R2    /desvia sempre
R1:   PREX  R0,R3/propaga exceção
        .
        .
        .
```

O código para armazenar o endereço da entrada já foi visto em chamada de entrada.

A instrução VREC desviará o controle para o endereço especificado pelo parâmetro R0, caso a tarefa T tenha sido abortada; caso contrário, será carregado no campo ETEX do RA da entrada o valor do parâmetro R1, que especifica o endereço da instrução PREX (que será discutida posteriormente). Em seguida, esta instrução verificará se existem tarefas esperando para realizar rendezvous através da entrada E; se existir pelo menos uma, iniciar-se-á o rendezvous; caso contrário, a entrada será colocada no estado "aberta", a tarefa T passará para o estado "esperando rendezvous - servidor" e será suspensa a sua execução.

A instrução ACCP desviará o controle para o endereço especificado pelo parâmetro R0, caso a tarefa T tenha sido abortada; caso contrário, montará o RA referente a chamada da entrada E, no topo de T.

As instruções VREC e ACCP especificam o ponto de sincronização "início de um comando *accept*".

Após a instrução ACCP deverá ser gerado código para copiar os parâmetros passados de modo *in* e *inout* que, estarão armazenados no RA da unidade chamadora.

Após o código gerado para os comandos do corpo do comando *accept*, deverá ser gerado código para copiar os parâmetros passados de modo *out* e *inout* armazenados no RA da tarefa chamada, para a área de parâmetros da tarefa chamadora.

A instrução FACC desviará o controle para o endereço especificado pelo parâmetro R0, caso a tarefa T tenha sido abortada durante o rendezvous; caso contrário, esta instrução terminará o comando *accept*.

O rótulo R2, da instrução DSVS, especifica o endereço da primeira instrução, após o comando *accept*.

A instrução PREX será executada somente se for levan-

tada uma exceção no corpo do comando *accept* e ela não foi tratada internamente. Esta instrução desviará o controle para endereço especificado pelo parâmetro R0, caso a tarefa T tenha sido abortada; caso contrário, esta instrução propagará a exceção levantada para a tarefa com a qual se comunicava e desviará o controle para o endereço especificado pelo parâmetro R3, que indicará o endereço do tratador de exceções da unidade corrente.

As instruções FACC e PREX especificam o ponto de sincronização "o fim de um comando *accept*".

3.4. Comando *delay*

Suponhamos o trecho de programa fonte a seguir:

```
.
.
.
delay  "expressão de duração"
.
.
.
```

O código que deverá ser gerado para implementar o comando *delay* deverá ter a seguinte forma:

```
.  ⎫
.  ⎬  código para calcular a expresão de duração
.  ⎭
VRDL  R0   /executa delay
.
.
.
```

A instrução VRDL desviará o controle para o endereço especificado pelo parâmetro R0 caso a tarefa T tenha sido abortada; caso contrário, verificará o valor da expressão de duração. Se for maior do que zero atualizará o estado da tarefa para

"dormente", inserirá a tarefa na lista de tarefas dormentes e suspenderá a execução da tarefa. Caso contrário, será executada a próxima instrução.

A instrução VRDL especifica o ponto de sincronização "um comando *delay*".

3.5. Comando *select*

A seguir apresentaremos como implementar as três formas do comando *select*, que são:

- espera seletiva
- chamada condicional de entrada
- chamada com espera de entrada

Espera Seletiva

Esta forma de comando *select* permite uma combinação de espera e seleção de uma ou mais alternativas. Esta forma de comando *select* deve conter pelo menos uma alternativa *accept*. Além disso, pode conter uma alternativa *terminate* (somente uma), uma ou mais alternativas *delay* ou uma parte *else*. Estas três possibilidades são mutuamente exclusivas.

Suponhamos o trecho de programa fonte a seguir:

select
"sequência de alternativas do corpo do comando *select*"
end select;

O código que deverá ser gerado para implementar o comando *select* da forma espera seletiva, deverá ter a seguinte forma:

```
        ISEL        /inicia seleção
         .
         .  } código para implementar o corpo do comando select
         .
RM:     FSEL    R0/realiza seleção
        APAC        /ajusta a pilha
        PREX  R0,R3/propaga exceção
```

A instrução ISEL inicializará o contador de alternativas *accept* aberta e o indicador de alternativa *delay* aberta ou, alternativa *terminate* aberta ou, guarda *else*. Neste trabalho será usada a pilha da tarefa para armazenar essas informações.

O código que implementa o corpo do comando *select* armazenará no topo da pilha o endereço do descritor da entrada e o endereço (na memória de programas) das alternativas *accept* que estiverem aberta. O endereço (na memória de programas), da alternativa *delay* aberta ou, *terminate* aberta ou, da parte *else*, será armazenado no campo PCDL da tarefa corrente. No caso de alternativa *delay* aberta, será armazenado no campo IHDS da tarefa corrente, o valor da expressão de duração.

A instrução FSEL desviará o controle para o endereço especificado pelo parâmetro R0 caso a tarefa corrente tenha sido abortada. Caso contrário, esta instrução, através das informações armazenadas durante a execução das condições das alternativas, em primeiro lugar considerará as alternativas *accept* aberta. O rendezvous será realizado imediatamente se existir pelo menos uma chamada de entrada correspondente a uma das alternativas *accept* aberta. Se for possível realizar o rendezvous com mais de uma alternativa, uma delas será selecionada ao acaso. Uma alternativa *delay* será selecionada se não houver possibilidade de selecionar uma alternativa *accept* imediatamente e se o tempo de espera for menor ou igual a zero ou ainda, se nenhuma alternativa *accept*

for selecionada antes de expirar o tempo de espera. Uma guarda *else* será selecionada se não houver possibilidade de selecionar uma alternativa *accept* imediatamente, em particular, se todas as alternativas *accept* estiverem fechadas. Uma alternativa *terminate* será selecionada, caso nenhuma alternativa *accept* possa ser selecionada imediatamente. Se no comando *select* da forma espera seletiva não existir pelo menos uma alternativa *accept* aberta e se não for selecionada outra alternativa, será levantada a exceção "PROGRAM ERROR" na tarefa corrente. O parâmetro RM indica o endereço para o qual deverá ser desviado o controle, após a execução do código que implementa o corpo do comando *select*. A instrução FSEL especifica o ponto de sincronização "um comando *select*"

A instrução APAC ajustará a pilha da tarefa corrente para executar a alternativa *accept* selecionada, caso a tarefa tenha sido reativada devido à chamada de uma entrada aberta. Não será necessário gerar esta instrução se o comando *select* possuir guarda *else*.

A instrução PREX será executada somente se ocorrer uma exceção durante a execução do corpo de um comando *accept* e ela não foi tratada internamente. Os parâmetros R0 e R3 especificam o endereço do código que desativa a tarefa e o endereço do tratador de exceções da tarefa corrente, respectivamente.

O código que deverá ser gerado para implementar o corpo do comando *select* dependerá das alternativas que forem usadas pelo programador.

Suponhamos o trecho de programa fonte a seguir:

```
task T is
.
.
.
begin
.
.
.
select
.
.
.
when C ⇒accept E ("lista de parâmetros formais") do
        "sequência de comandos"
.
.
.
end select;
.
.
.
end T;
```

O código que deverá ser gerado para implementar um comando *accept* ,residente no corpo de um comando *select*, deverá ter a seguinte forma:

```
.
.
.
.
.
.   } código para calcular a expressão da condição C
.
DSVF  R1    /desvia se falso
.
.   } código para calcular e armazenar o endereço da
.   } entrada
CECA  R2    /carrega o endereço do corpo do comando accept
DSVS  R1    /desvia sempre
R2:   ACCP  R0    /inicia o rendezvous
```

```
   .
   .  } código para o corpo do comando accept
   .
FACC  R0   /finaliza o rendezvous
   .
   .  } código para os comandos opcionais
   .
DSVS  R3   /desvia sempre
   .
   .
   .
```

A expressão da condição C, se houver, será uma expressão do tipo booleana; se não houver deverá ser carregado o valor booleano verdadeiro.

O parâmetro R1 especificará o endereço onde será calculado o valor da próxima condição ou o endereço da instrução FSEL (rótulo RM), se esta for a última alternativa do comando *select*.

O código para calcular o endereço da entrada E já foi discutido na geração de código para a chamada de entrada.

A instrução CECA carregará o endereço da alternativa *accept*, que será especificado pelo parâmetro R2, e atualizará o indicador de alternativa *accept* aberta.

O código que deverá ser gerado para o comando *accept* já foi discutido anteriormente (vide comando *accept*).

O parâmetro R3, da instrução DSVS, especificará o endereço da próxima instrução que deverá ser executada após ter completado a execução do comando *select*.

Suponhamos o trecho do programa fonte a seguir:

```
.
.
.
select

.
.
.
when   C ⇒ delay "expressão de duração";
           "sequência de comandos"
.
.
.
end select;

.
.
.
```

O código que deverá ser gerado para implementar um comando *delay*, residente no corpo de um comando *select*, deverá ter a seguinte forma:

```
.
.
.
.
.   } código para a condição C
.
DSVF  R1    /desvia se falso
.
.   } código para calcular o valor da expressão de duração
.
CEAD  R2    /carrega o endereço da alternativa delay
DSVS  R1    /desvia sempre
R2:  APDL   /ajusta a pilha
.
.   } código para os comando opcionais
.
DSVS  R3    /desvia sempre

.
.
.
```

A instrução CEAD atualizará o indicador de alternativa *delay* aberta e, verificará se o valor da expressão de duração é menor do que o valor armazenado no campo IHDS do RA desta tarefa. Se isto for verdade, então o valor da expressão duração será armazenado em IHDS e, o parâmetro R2 em PCDL para indicar o endereço da alternativa *delay*. Se várias alternativas *delay* possuirem a mesma duração, uma delas será selecionada ao acaso.

O parâmetro R1 especificará o endereço da próxima condição ou o endereço da instrução FSEL (o rótulo RM) e o parâmetro R3, da instrução DSVS, especificará o endereço da próxima instrução, após o comando *select*.

A instrução APDL retirará as informações que foram armazenadas durante a execução das condições das alternativas.

Suponhamos o trecho de programa fonte a seguir:

```
Select
.
.
.
When C⇒terminate;
.
.
.
end select;
```

O código que deverá ser gerado para implementar a alternativa *terminate* deverá ter a seguinte forma:

```
.
.   código para calcular a expressão da condição
.
DSVF R1    /desvia se falso
CEAT R2    /carrega o endereço da alternativa terminate
           /aberta
TERM       /verifica término
```

O parâmetro R1 especificará o endereço da próxima instrução, após a instrução TERM.

A instrução CEAT atualizará o indicador de alternativa *terminate* aberta, armazenará uma referência para a instrução TERM e desviará o controle para o endereço especificado pelo parâmetro R2, que poderá ser o endereço de uma alternativa *accept* ou da instrução FSEL (rótulo RM).

A instrução TERM verificará se existe uma tarefa no estado "completado" ou uma unidade no estado "unidade completada" e, da qual esta tarefa é descendente e também, se todas as tarefas que descendem da unidade estão num dos estado "esperando término" ou "esperando rendezvous ou esperando término" ou "terminado"; caso isto seja verdadeiro, todas as tarefas serão terminadas simultaneamente; caso contrário, esta tarefa será suspensa.

Suponhamos o trecho de programa fonte a seguir:

```
select

.
.
.

else    "sequência de comandos"
end select;
```

O código que deverá ser gerado para implementar uma guarda *else* deverá ter a seguinte forma:

```
        CRGE  R2   /carrega guarda else
        DSVS  RM   /desvia sempre
R2:     .
        .   } código para os comandos especificados na guarda else
        .
        DSVS  R1   /desvia sempre

        .
        .
        .
```

A instrução CRGE atualizará o indicador de guarda *else* e armazenará uma referência para a região de código de comandos

da guarda *else*, especificado pelo parâmetro R2.

Os parâmetros RM e R1 especificarão o endereço da instrução FSEL e da próxima instrução após o comando *select*, respectivamente.

Chamada Condicional de Entrada

Suponhamos o trecho de programa fonte a seguir:

```
task T is
.
.
.
end T;
.
.
.
task body T is
.
.
.
begin
.
.
.
select
S.E ("lista de parâmetros efetivos");
"sequência de comandos"
else
"sequência  de comandos"
end select;
.
.
.
end T;
```

O código que deverá ser gerado para implementar esta forma do comando *select*, deverá ser:

```
        .
        .   código para calcular o endereço de S
        .
        .
        .   código para calcular o endereço de E
        .
        .
        .   código para avaliar os parâmetros
        .
    CHCE  R0,R1/chamada condicional de entrada
    AEST  R0   /atualiza o estado da tarefa
        .
        .   código para copiar de volta os parâmetros passa-
        .   dos de modo out e inout
        .
        .
        .   código para os comandos opcionais
        .
    DSVS  R2   /desvia sempre
R1:     .
        .   código para os comandos da guarda else
        .
```

O código que deverá ser gerado para calcular o endereço de S, o endereço de E e para avaliar os parâmetros já foi discutido na instrução CHEN.

Na instrução CHCE (como foi feito na instrução CHEN), é suposto que o endereço de E bem como o endereço de S, estarão armazenados no topo da pilha de T. A instrução CHCE desviará o controle para o endereço especificado pelo parâmetro R0, caso a tarefa T tenha sido abortada; caso contrário, será verificado se a tarefa S está num dos estados que indicam término; se estiver, será levantada a exeção "TASKING ERROR" na tarefa T; caso contrário, será verificada a possibilidade de realizar o rendezvous imediatamente; se for possível, a instrução preparará o contexto pa-

ra realizar o rendezvous; caso contrário, iniciará a execução dos comandos da guarda *else*, cujo endereço será especificado pelo parâmetro R1. Se o rendezvous terminar com sucesso, a tarefa T reiniciará a execução a partir dos comandos opcionais, se houver, localizados após a instrução CHCE.

As instruções CHCE e AEST especificam o ponto de sincronização "chamada de uma entrada".

O parâmetro R2, da instrução DSVS, especificará o endereço da próxima instrução, após o comando *select*.

Chamada com Espera de Entrada

Suponhamos o trecho de programa fonte a seguir:

```
task T is
.
.
.
end T;
.
.
.
task body T is
.
.
.
begin
.
.
.
select
S.E ("lista de parâmetros efetivos");
"sequência de comandos"
or
delay "expressão de duração";
"sequência de comandos"
```

end select;

.
.
.

end T;

O código que deverá ser gerado para implementar a chamada com espera da entrada E, deverá ter a seguinte forma:

```
.
.   } código para calcular o endereço de S
.

.
.   } código para calcular o endereço de E
.

.
.   } código para avaliar os parâmetros efetivos
.

.
.   } código para avaliar a expressão de duração
.

      CHEE  R0,R1/chamada com espera de entrada
      AEST  R0   /atualiza o estado da tarefa
.
.   } código para copiar de volta os parâmetros passa-
.   } dos de modo out e inout
      DSVS  R2   /desvia sempre
R1:   APDL       /ajusta a pilha para executar a alternativa
                 /delay
.
.   } código para os comandos opcionais após a alterna-
.   } tiva delay
```

O código que deverá ser gerado para calcular o endereço de S, o endereço de E e para avaliar os parâmetros já foi discutido na instrução CHEN.

O código para calcular a expressão de duração já foi discutido no comando *delay*.

Na instrução CHEE (como foi feito nas instruções de

chamada de entrada, descritas anteriormente), é suposto que o endereço de E bem como o endereço de S, estarão armazenados no topo da pilha de T. A instrução CHEE desviará o controle para o endereço especificado pelo parâmetro R0, caso a tarefa T tenha sido abortada; caso contrário, será verificado se a tarefa S está num dos estados que indicam têrmino; se estiver, será levantada a exceção "TASKING ERROR" na tarefa T; caso contrário, será verificada a possibilidade de realizar o rendezvous imediatamente. Se for possível esta instrução preparará o contexto para realizar o rendezvous; caso contrário verificará se o valor da expressão de duração é menor ou igual a zero; se for, iniciará imediatamente a execução dos comandos opcionais após a alternativa *delay*; caso contrário a tarefa T será inserida na lista de tarefas dormentes com o respectivo tempo de espera, e será também inserida na fila de tarefas esperando rendezvous, da entrada chamada. O rendezvous será realizado somente se ocorrer a execução de um comando *accept* para a entrada chamada e se esta tarefa for a primeira tarefa da fila de tarefas esperando rendezvous, da entrada, antes de expirar o tempo de espera. O parâmetro R1 especificará o endereço onde deverá reiniciar a execução da tarefa, caso ela seja reativada devido a temporização.

As instruções CHEE e AEST especificam o ponto de sincronização "chamada de uma entrada".

O parâmetro R2, da instrução DSVS, especificará o endereço da próxima instrução após o comando *select*.

3.6. Comando *abort*

Suponhamos o trecho de programa fonte a seguir:

```
        .
        .
        .
        abort T1, T2, ..., Tn
        .
        .
        .
```

O código que deverá ser gerado para implementar um comando *abort*, deverá ter a seguinte forma:

```
        .
        .
        .
        .  }  código para calcular o endereço de T1
        .
        .  }  código para calcular o endereço de T2
        .
        .
        .
        .
        .  }  código para calcular o endereço de Tn
        .
      ABRT   N,R0 /marca as tarefas como anormal
        .
        .
        .
        .
   R0:  .  }  código para desativar a tarefa
        .
```

A instrução ABRT suporá que os endereços das N tarefas, para serem abortadas, estarão armazenados no topo da pilha da tarefa corrente. Esta instrução primeiramente, verificará o estado da tarefa corrente; se for "anormal", desviará o controle para o endereço especificado pelo parâmetro R0; caso contrário, marcará como "anormal" as tarefas indicadas na pilha. Esta instrução especifica o ponto de sincronização "execução de um coman-

do *abort*".

O código que deverá ser gerado para desativar uma tarefa, deverá ter a seguinte forma:

```
R0:   DSRE          /desaloca RA de entradas
      VRDP          /verifica a existência de dependentes
      VRTD  R0,R1/verifica término de desativação
```

A instrução DSRE removerá todos os RA's de entradas do topo da pilha da unidade corrente, levantando a exceção "TASKING ERROR" na tarefa com a qual se comunicava. O rótulo R0 especifica o endereço do código que dasativa a tarefa, caso seja abortada.

A instrução VRDP verificará se a unidade corrente possui dependentes. Se possuir, a tarefa será suspensa até que todos os seus dependentes sejam terminados.

A instrução VRTD verificará se a RA do topo da pilha corresponde ao RA da tarefa. Se for, o controle será desviado para o código que termina a tarefa, especificado pelo parâmetro R1; caso contrário, deverá ser executada novamente a instrução DSRE, para remover, caso houver, RA de entradas do topo da pilha.

Suponhamos que uma tarefa que é marcada como "anormal" está na lista de tarefas dormentes (no estado "dormente" ou "esperando rendezvous - ator dormente") e que seja reativada por temporização. Como pode ser observado na instrução APDL, a própria rotina que "acorda" a tarefa verificará o estado da tarefa e se for o caso, atualizará para "normal". Como nessa rotina que "acorda " a tarefa não é conhecido o endereço da instrução que desativa a tarefa ("acordada") então, essa tarefa será desativada somente quando o estado "anormal" for detectado por alguma instrução da tarefa ou por alguma unidade que esteja encaixada na tarefa, num

ponto de sincronização.

4. Exceções

Nesta secção serão descritas as atividades do compilador e do ligador/carregador para fazer a declaração das exceções e o código que deverá ser gerado para levantar e fazer o tratamento das exceções.

4.1. Declaração de Exceções

Para fazer o tratamento das exceções será usado um número único que identifica a exceção (índice da exceção).

Durante a compilação de uma unidade de programa deverá ser montada uma tabela com as exceções declaradas na unidade.

No momento da carga/ligação deverá ser montada a tabela global de exceções, que será composta das exceções declaradas nas unidades de programas e das exceções pré-definidas pela linguagem.

A posição da exceção, na tabela de exceções, será usada como o índice da exceção no programa.

Durante a execução, o índice da exceção será tratado como uma constante.

4.2. Tratamento de Exceções

Nesta secção descreveremos como o sistema de execução deverá se comportar para fazer o tratamento das exceções.

Para que uma exceção possa ser tratada no contexto apropriado, durante a elaboração da parte declarativa de subpro-

gramas, blocos e pacotes o campo ETEX do RA da unidade estará atualizado com o endereço do tratador de exceções da unidade antecessora dinâmica. A elaboração da parte de especificação de tarefas será executada no contexto da unidade mestre.

Após a elaboração da parte declarativa de subprogramas, blocos, pacotes e da elaboração da parte de especificação de tarefas o campo ETEX do RA dessas unidades será atualizado com o endereço do seu tratador de exceções. As exceções levantadas após a parte declarativa de unidades de programas, serão tratadas nos seus respectivos tratadores de exceções. Uma exceção levantada na parte declarativa de uma tarefa implica que a exceção "TASKING ERROR" seja levantada na unidade mestre.

4.3. Propagação de Exceções

De modo geral, uma exceção deverá ser propagada para a unidade antecessora estática (no caso de pacotes e blocos) ou dinâmica (no caso de subprogramas) nos seguintes casos:

(a) Se a exceção for relevantada no tratador de exceções, através do comando *raise* ou,

(b) Se a exceção não for resolvida no tratador de exceções da unidade.

O campo IELV será inicializado com o valor verdadeiro no início do tratador de exceções (pela instrução LIEL) da unidade corrente. Após o tratamento da exceção o campo IELV será atualizado com o valor falso (pela instrução DIEL). As instruções que completam e terminam os diversos tratadores de exceções verificam o valor do campo IELV. Se for verdadeiro a exceção é propagada para a unidade antecessora (estática ou dinâmica, dependendo da unidade), caso contrário, a unidade é terminada normalmente. Com

isto, é resolvido o caso (b) e também simplificada a solução do caso (a), como pode ser visto no comando *raise*.

Uma exceção também pode ser propagada para outra tarefa. Como já mencionado, quando uma exceção é levantada dentro de um comando *accept*, mas não é tratada em uma unidade mais interna, ela será propagada para a tarefa chamadora (vide a instrução PREX, no comando *accept*).

O campo IELV foi criado para facilitar e tornar mais rápida a verificação se uma exceção é para ser propagada ou não. Ele poderia ser substituído pelo campo estado da tarefa (pelo estado, por exemplo,"em exceção") porém, verificar e atualizar o estado de uma tarefa gera instruções mais complicadas do que simplesmente testar e atualizar o valor do campo IELV.

4.4. Tratadores de Exceções

Quando ocorre uma exceção em uma unidade que não possui código para fazer o tratamento da exceção, a exceção levantada deve ser normalmente, propagada. Consequentemente, mesmo que a unidade não possui um tratador de exceções, deverá ser gerado código para que a exceção possa ser propagada. O código gerado para fazer o "tratamento de exceções" em unidade que não possui tratador de exceções, declarado explicitamente pelo programador, chamaremos de pseudo-tratador de exceções. O código que deverá ser gerado para implementar um pseudo-tratador de exceções de uma unidade, e o código para verificar o contexto bem como, para terminar o tratamento de uma exceção (de um tratador de exceções explícito), dependerá do tipo da unidade. No entanto, o código para implementar a sequência de escolhas é geral para qualquer tipo de unidade, portanto, será discutido genericamente, no final desta

secção. Finalmente, deve ser observado que um tratador de exceções especifica um ponto de sincronização.

Tratadores de Exceções de Pacotes

O código que deverá ser gerado para implementar pseudo-tratadores de exceções de pacotes deverá ter a seguinte forma:

```
TRPE   R0   /termina o pacote  e propaga exceção
```

A instrução TRPE desviará o controle para o endereço especificado pelo parâmetro R0, caso a tarefa corrente tenha sido abortada; caso contrário, esta instrução ajustará a altura da pilha e propagará a exceção indicada pelo campo IEXC do RA do pacote para a unidade antecessora estática, indicada pelo campo ARAE do RA do pacote.

O código que deverá ser gerado para implementar um tratador de exceções (explícito) de um pacote, deverá ter a seguinte forma:

```
VREA   R0   /verifica estado anormal
  .
  .   código para implementar a sequência de escolhas
  .
TRPC   R0   /termina o pacote
```

A instrução VREA desviará o controle para o endereço especificado pelo parâmetro R0, caso a tarefa corrente tenha sido abortada.

A instrução TRPC desviará o controle para o endereço especificado pelo parâmetro R0, caso a tarefa corrente tenha sido abortada; caso contrário, esta instrução verificará se a exceção levantada foi tratada. Se isto for verdade, esta instrução simplesmente ajustará a altura da pilha, caso contrário, propagará a

exceção levantada para a unidade antecessora estática.

Código para Tratadores de Exceções de Blocos e Subprogramas

O código que deverá ser gerado para pseudo-tratadores de exceções de subprogramas deverá ter a seguinte forma[i]:

```
CMUN  R0    /completa unidade
VTUC        /verifica término de unidade completada
AETR  R0    /atualiza o estado da tarefa
TUNE        /termina a unidade
```

A instrução CMUN desviará o controle para o endereço especificado pelo parâmetro R0, caso a tarefa corrente tenha sido abortada; caso contrário, esta instrução deverá ajustar a altura da pilha e atualizar o estado da tarefa para "unidade completada".

A instrução AETR desviará o controle para o endereço especificado pelo parâmetro R0, caso a tarefa corrente tenha sido abortada; caso contrário, atualizará o estado da tarefa através do estado armazenado no RA do contexto antecessor.

A instrução TUNE ajustará a altura da pilha desalocando o RA da unidade corrente e propagará a exceção indicada pelo campo IEXC desta unidade, para a unidade antecessora dinâmica, indicado pelo campo AUAN.

O código que deverá ser gerado para implementar um tratador de exceções (explícito) de um subprograma, deverá ter a seguinte forma:

```
VREA  R0    /verifica estado anormal
 .
 .   } código para implementar a sequência de escolhas
 .
```

[i]: Blocos serão tratados como subprogramas sem parâmetros.

```
CMUN  R0    /completa unidades
VTUC        /verifica término de unidade completada
  .
  .  } copia parâmetros
  .
AETR  R0    /atualiza o estado da tarefa
  .
  .  } código para terminar a unidade
  .
```

Se o subprograma possuir parâmetros, deverá ser gerado código para copiar os parâmetros passados de modo *out* e *inout* para as respectivas variáveis representadas pelos parâmetros efetivos.

O código que deverá ser gerado para terminar a unidade dependerá da unidade. Se a unidade for um bloco, deverá ser gerada a instrução TRBL, caso contrário, isto é, se a unidade for um subprograma, deverá ser gerada a instrução TRSP. Estas instruções deverão ajustar a altura da pilha para retornar para o contexto anterior; em seguida verificarão se a exceção foi tratada ou não. Se a exceção não foi tratada, a mesma deverá ser propagada para a unidade antecessora dinâmica; caso contrário, se a exceção foi tratada então, no caso de blocos, deverá ser executada a próxima instrução, que pertencerá à unidade antecessora. No entanto, no caso de um subprograma, o controle deverá ser desviado para a instrução seguinte à chamada do subprograma, cujo endereço é indicado pelo campo ENRT do RA do subprograma.

Tratadores de Exceções de Tarefas

O código que deverá ser gerado para implementar um pseudo-tratador de exceções de uma tarefa, deverá ter a seguinte forma:

```
        CMTE  N  /completa a tarefa
        VTUC     /verifica término de unidade completada
        TRTR     /termina tarefa
```

A instrução CMTE levantará a exceção "TASKING ERROR" na unidade mestre caso o estado da tarefa corrente seja "em elaboração". Esta tarefa deverá indicar para a unidade mestre o fim de sua ativação. Deverá ainda, levantar a exceção "TASKING ERROR" em todas as tarefas que estiverem em alguma de suas entradas, esperando para realizar rendezvous. O parâmetro N indicará o número de entradas na tarefa. Este parâmetro tem o mesmo significado que o da instrução CMTR. Em seguida, deverá atualizar o estado da tarefa para "completado" e ajustar a altura da pilha.

O código que deverá ser gerado para implementar um tratador de exceções (explícito) de uma tarefa, deverá ter a seguinte forma:

```
        VRET  R0, R3/verifica o estado da tarefa
        .  |
        .  } código para verificar as exceções e implementar
        .  |  os comandos das escolhas
    R3: CMTR  N      /completa tarefas
        VTUC         /verifica término de unidade completada
        TRTR         /termina tarefas
```

A instrução VRET desviará o controle para o endereço especificado pelo parâmetro R0, caso a tarefa T tenha sido abortada, caso contrário, se o estado de T for "em elaboração" será levantada a exceção "TASKING ERROR" na undade mestre e desviará o controle para o endereço especificado pelo parâmetro R3 que indicará o endereço das instruções que completam e terminam a tarefa.

Código para Implementar a Sequência de Escolhas

Suponhamos o trecho de programa fonte, de uma unidade U, a seguir:

```
.
.
.
begin
.
.
.
exception
when E1 ⇒ "sequência de comandos"
.
.
.
when En ⇒ "sequência de comandos"
others ⇒ "sequência de comandos"
end U;
.
.
.
```

O código que deverá ser gerado para implementar o tratador de execeções exposto acima, deverá ter a seguinte forma:

```
.  |
.  |  cógigo para verificar o contexto para fazer o tra-
.  |  tamento da exceção
LIEL        /liga o indicador de exceção levantada
.  |
.  |  código para verificar se a exceção levantada é i-
.  |  gual a E1
.
.
.
.  |
.  |  código para verificar se a exceção levantada é i-
.  |  gual a En
```

```
        DSVS  R1    /desvia sempre
              .
              .     código para a sequência de comandos da escolha E1
              .
        DSVS  R2    /desvia sempre
              .
              .
              .
REN:          .
              .     código para a sequência de comandos da escolha En
              .
        DSVS  R2    /desvia sempre
R1:           .
              .     código para a sequência de comandos  da  escolha
              .     others
R2:     DIEL        /desliza o indicador de exceção levantada
R3:           .
              .     código para terminar a unidade
              .
```

O código para verificar o contexto para fazer o tratamento da exceção já foi discutido anteriormente.

A instrução DIEL atualizará o campo IELV para indicar que existe uma exceção levantada.

O código para testar os nomes de exceções terá a forma de "ninhos de if's". Deverá ser gerado um conjunto de instruções para cada nome de exceção especificado nas escolhas. Este conjunto de instruções verificará se o nome da exceção levantada foi especificado em alguma escolha. Os rótulos RE1 ... REN, especificarão os endereços das escolhas (código para a sequência de comandos da exceção).

Uma outra maneira de localizar as escolhas seria através de uma tabela de indireções. O problema desta solução é que a montagem da tabela de endereços deve ser feita em tempo de execução.

Na solução adotada, para tratar as exceções (implícitas) da escolha *others*, necessitaremos simplesmente desviar o controle para a sequência de comandos da escolha *others*. Se não existir a escolha *others*, deverá ser gerada uma instrução que desvia o controle para o endereço especificado pelo rótulo R3, para que a exceção possa ser propagada ou levantada uma nova exceção, no caso de tarefas.

O rótulo R2 especificará o endereço da instrução DIEL, que deslizará o indicador de exceção levantada, indicado pelo campo IELV do RA da unidade.

4.5. Comando *raise*

Suponhamos o trecho de programa fonte a seguir:

```
.
.
.
ERROR : exception;
.
.
.
begin
.
.
.
raise ERROR;
.
.
.
exception
.
.
.
when ERROR ⇒ ...
```

```
.
.
.
end;
.
.
.
```

O código que deverá ser gerado para implementar o comando *raise* deverá ter a seguinte forma:

```
.
.
.
LVEX  I,R1 /levanta exceção
.
.
.
```

A instrução LVEX carregará o campo IEXC com o parâmetro I, que indicará o índice da exceção e desviará o controle para o tratador de exceções da unidade, cujo endereço será indicado pelo parâmetro R1.

Suponhamos agora, o trecho de programa fonte a seguir, onde o comando *raise* levanta novamente a exceção:

```
.
.
.
begin
.
.
.
exception
.
.
.
when E1⇒ ...; raise;
.
.
.
```

end;

.
.
.

O código que deverá ser gerado para implementar o comando *raise*, residente no corpo do tratador de exceções deverá ter a seguinte forma:

.
.
.

DSVS R1 /relevanta a exceção

.
.
.

O parâmetro R1 indicará o endereço das instruções que completam e terminam a unidade.

4.6. Exceções Pré-definidas pela Linguagem

O sistema operacional da máquina deverá possuir rotinas que prepararão o contexto e desviarão o controle para o tratador de exceções da unidade onde ocorreu a exceção, para resolver as exceções levantadas que são pré-definidas pela linguagem. Para preparar o contexto a rotina deverá simplesmente carregar o campo IEXC do RA da unidade com o índice da exceção levantada. O endereço para onde deverá ser desviado o controle estará especificado no campo ETEX do RA da unidade, onde a exceção foi levantada.

5. Ocorrências de Bloqueio

Bloqueio ("deadlock") é o estado no qual dois ou mais

processos esperarão indefinidamente por condições que nunca serão alcançados.

Para resolver o problema de exclusão mútua na secção 2.3., definimos as rotinas "entra na RC (A)" para obtermos o recurso A e "sai da RA (A)" para liberarmos o recurso.

Suponhamos que T1 e T2 sejam tarefas executando paralelamente e, DT1 e DT2 são recursos. Consideramos o seguinte trecho de programa:

T1	T2
entra na RC(DT1)	entra na RC(DT2)
.	.
.	.
.	.
entra na RC(DT2)	entra na RC(DT1)
.	.
.	.
.	.
sai da RC(DT2)	sai da RC(DT1)
.	.
.	.
.	.
sai da RC(DT1)	sai da RC(DT2)

Obviamente T1 e T2 estarão bloqueados.

Algumas instruções da máquina - Ada necessitarão de dois recursos simultaneamente para atender os requisitos impostos pela semântica da linguagem Ada.

As instruções ACCP, ATMS, CMTE, CMTR, FACC, FSEL e VRET não provocarão bloqueio porque, de modo geral, sempre que a tarefa T1 consequir obter o recurso DT1 a tarefa T2 estará suspensa, e consequentemente, o recurso DT2 não estará ocupado por essa tarefa.

As instruções CHCE, CHDE e CHEN poderão provocar blo-

queio nas seguintes condições:

(1) Se a entrada chamada estiver declarada na própria tarefa

(2) Se uma tarefa T1 chamar uma entrada da tarefa T2 que por sua vez, chama uma entrada da tarefa T1, "simultaneamente".

(3) Se uma tarefa T1 chamar uma entrada da tarefa T2 que por sua vez, aborta a tarefa T1, "simultaneamente".

Estas condições somente poderiam ocorrer no caso de um erro de programação. Estas condições são impossíveis de serem detectadas pelo compilador. Por exemplo, no caso (1), pode não ser possível detectar em tempo de compilação se a chamada é de uma entrada pertencente à parte de especificação da própria tarefa, se a tarefa que possui a entrada chamada e que chama a entrada pertencem à mesma família.

A instrução ABRT poderá provocar bloqueio através das condições expostas anteriormente em (3) ou, se uma tarefa - T1 abortar a tarefa T2 que por sua vez, aborta T1, "simultaneamente". Como nas instruções de chamada de entrada, esta situação somente poderá ocorrer no caso de erro de programação e, que também é impossível de ser detectada em tempo de compilação.

Obviamente, outros bloqueios devido a erros de programação, poderão ocorrer.

IV - CONCLUSÕES

A vantagem de usar a técnica de geração de código intemediário, baseado em uma máquima virtual, é a portabilidade de compiladores e a simplicidade de geração de código que, separa em fases distintas a análise sintática e semântica da linguagem.

Na elaboração do sistema de execução proposto foram tomados como base os fatores de legalidade, no que se refere a conformidade com a definição da linguagem e legibilidade. Com isso, foi até certo ponto penalizada a eficiência, isto devido às características da linguagem.

Numa implementação real, o programa que usar de maneira não controlada os comandos *abort* e *select* com espera seletiva, poderá tornar-se vagaroso. Um outro aspecto que certamente prejudicará a eficiência do programa será executar várias tarefas paralelamente, com comunicação intensa.

Para implementar o compilador Ada, faz-se necessário a criação de um sistema operacional especializado, devido principalmente à combinação de características tais como: comunicação de processos, tratamento de erros em tempo de execução e compilação em separado e consequentemente, tornando a implementação do compilador bastante complexa.

Uma maneira de reduzir este problema seria simplificar a definição da linguagem, removendo determinadas particularidades, que não prejudicasse o propósito do compilador.

Naturalmente, o sistema de execução proposto está incompleto, entretanto, a implementação das construções da parte sequencial são bem mais simples, pois assemelham-se com as construções das linguagens algorítmicas, tais como Pascal e Algol. As dificuldades de implementação das demais fases do compilador são basicamente as mesmas das linguagens citadas. Certamente, a maior dificuldade de implementação do compilador Ada, como já mencionado anteriormente, apresenta-se na implementação de um sistema operacional especializado.

BIBLIOGRAFIA

[01] DoD, "Military Standard - Ada Programming Language", ANSI/ MIL - STD - 1815A, 1983.

[02] MELLO FQ, R.D.B.P., "Proposta de um Sistema de Execução para a Linguagem de Programação ADA", Tese M.Sc., UNICAMP, 1980.

[03] IBSEN, L., "A Portable Virtual Machine for Ada", Software-Pratice and Experience, Vol. 14, 1984.

[04] GROVES, L.J. e ROGERES, W.J., "The Design of a Virtual Machine for Ada", ACM SIGPLAN, Symposium on the Ada Programming Language, 1982.

[05] ICBIAH, J.D. et al., "Preliminary ADA Reference Manual", ACM SIGPLAN Notices 14, 6, julho, 1979, Part A.

[06] ICBIAH, J.D. et al., "Preliminary ADA Reference Manual", ACM SIGPLAN Notices 14, 6, julho, 1979, Part B.

[07] KOWALTOWSKI, T., "Implementação de Linguagens de Programa-

ção", Guanabara Dois, 1983.

[08] AHO, A.V. e ULLMAN J.D., "Plinciples of Compiler Design", Addison-Weslley, 1977.

[09] DIJKSTRA, E.W., "Hierarchical Ordering of Sequencial Processes", Acta Inf 1, 2, 1971.

[10] HABERMANN, A.N., "Prevention of System Deadlocks", CACM, Vol 12, Num 7, julho, 1969.

[11] GUIMARÃES, C.C., "Princípios de Sistemas Operacionais", Campus, 1980.

[12] BRINCH HANSEN, P., "The Architecture of Concurrent Programs", Prentice-Hall, 1977.

APÊNDICE I

```
instrução DCTR  RTE /declaração de tarefa.
início
   X0 := AP de ARTC;
   X1 := aloca-pilha;
   ARAT de X0 := X1;
   SEMF de X0 := "fechado";
   ESTR de X0 := "em elaboração";
   TPRA de X1 := "TR";
   ESTC de X1 := "em elaboração";
   ARTR de X1 := X1;
   ETEX de X1 := RTE;
   NTNA de X1 := 0;
   NTNT de X1 := 0;
   AFTD de X1 := X1;
   ADST de X1 := X0;
   NVCR de X1 := 0;
   DISP[0] de X1 := X1;
   ARAC de X1 := X1;
   TOPO de X1 := "endereço de PDES de X1";
   AMST de X1 := ARUC;
   PRES de X1 := "prioridade indefinida";
   PRDN de X1 := "prioridade indefinida";
fim.
```

```
instrução INDP /insere tarefa na fila de tarefas dependentes.
início
   X0 := AP de ARTC;
   X1 := ARAT de X0;
   CPTR de X1 := CP;
   sai da RC(X0);
   insere-AFTD(X1);
   NTNA de ARUC := NTNA de ARUC + 1;
   X0 := (AP - 1) de ARTC;
   AP := AP - 2;
   CP := X0
fim.
```

```
instrução ATDP R0 /ativa dependentes.
início
   X0 := ADST de ARTC;
   entra na RC(X0);
   se ESTR de X0 = "anormal"
      então início
               sai da RC(X0);
               CP := R0
            fim;
   X1 := AFTD de ARTC;
   enquanto X1 ≠ ARTC  faça
      início
         insere-AFTP(X1);
         X1 := AFTI de X1
      fim;
   se NTNA de ARUC ≠ 0
      então início
               sai da RC(X0);
               escalonador
            fim;
   sai da RC(X0)
fim.
```

```
instrução ATMS R0 /ativa o mestre.
início
   X1 := AMST de ARTC;
   X2 := ARTR de X1;
   X3 := ADST de X2;
   entra na RC(X3);
   NTNA de X1 := NTNA de X1 - 1 ;
   X0 := ADST de ARTC;
   entra na RC(X0);
   se ESTR de X0 ≠ "anormal"
      então NTNT de X1 := NTNT de X1 + 1;
   se NTNA de X1 = 0
      então início
               sai da RC(X3);
               insere-AFTP(X2)
            fim
      senão sai da RC(X3);
   se ESTR de X0 = "anormal"
      então início
               sai da RC(X0);
               CP := R0
            fim;
   ESTR de X0 := "normal";
   ESTC de ARTC := "normal";
   sai da RC(X0)
fim.
```

```
instrução CDTC R1 /declaração de tarefa simples.
início
   (AP + 4) de ARTC := R1;
   (AP + 5) de ARTC := AP + 1;
   AP := AP + 5
fim.
```

```
instrução CDTV R1 /declaração de variável do tipo tarefa.
início
    (AP + 4) de ARTC := CP;
    (AP + 5) de ARTC := Ap + 1;
    AP := AP + 5;
    CP := R1
fim.
```

```
instrução CDFT  DT /cria o descritor do vetor.
início
   X0 := ED;
   ARDS de X0 := AP - 1;
   LINF de X0 := (AP - 1) de ARTC;
   LSUP de X0 := AP de ARTC;
   (AP - 1) de ARTC := AP de ARTC - (AP - 1) de ARTC + 1;
   AP := AP - 1
fim.
```

```
instrução CDEN /carrega descritor de entrada.
início
   (AP + 1) de ARTC := "DE";
   (AP + 2) de ARTC := "fechada";
   (AP + 3) de ARTC := "vazio";
   (AP + 4) de ARTC := 0;
   AP := AP + 4
fim.
```

```
instrução CDFE ED / carrega descritor de família de entradas.
início
   X0 := ED;
   TDEN de X0 := "DFE";
   LINF de X0 := (AP - 1) de ARTC;
   LSUP de X0 := AP de ARTC;
   ARDS de X0 := AP - 1;
   X1 := |(AP de ARTC - (AP - 1) de ARTC)| + 1;
   AP := AP - 2;
   para X0 := 1, X1 faça
   início
      AP de ARTC := "fechada";
      (AP + 1) de ARTC := "vazio";
      (AP + 2) de ARTC := 0;
      AP := AP + 3
   fim
fim.
```

```
instrução CRPR P /carrega prioridade.
início
   PRES de ARTC := P;
   PRDN de ARTC := P
fim.
```

```
instrução CMTR N /completa tarefa.
início
   X0 := ADST de ARTC;
   entra na RC(X0);
   ESTR de X0 := "completado";
   X1 := "deslocamento de PDES de ARTC";
   para X2 := 1, N  faça
   início
      se TDEN de X1 = "DE"
         então lexter(X1)
         senão para X3 := 1, |LSUP de X1 - LINF de X1| faça
                  lexter(ARDS de X1 + (X3 - 1) * 3);
      X1 := X1 + 4
   fim;
   sai da RC(X0).
fim.
```

```
proc lexter(X0);
início
   para X3 := 1, NTER de X0  faça
   início
      X1 := AFTR de X0;
      X2 := ADST de X1;
      entra na RC(X2);
      se ESTR de X2 = "esperando rendezvous ator-dormente"
         então retira-ALTD(X1);
      IEXC de X1 := "índice de exceção TASKING-ERROR";
      CPTR de X1 := ETEX de X1;
      sai da RC(X2);
      insere-AFTP(X1);
      retira-AFTR(X0,X1)
   fim
fim;
```

```
instrução VTUC    / verifica a possibilidade de término de unida-
                  / de completada.

início
   X1 := ADST de ARTC;
   entra na RC(X1);
   se NTNT de ARUC = 0
      então sai da RC(X1)
      senão se busca(ARTC, ARUC, AFTD de ARUC)
               então início
                        lib-esp(ARUC, AFTD de ARUC);
                        sai da RC(X1)
                     fim
               senão início
                        sai da RC(X1);
                        CP := CP - 1;
                        escalonador
                     fim
fim.
```

```
instrução TRTR /termina tarefa.
início
   X0 := AMST de ARTC;
   X1 := ADST de ARTC;
   entra na RC(X1);
   ESTR de X1 := "terminado";
   sai da RC(X1);
   desaloca-pilha(ARTC);
   X2 := ARTR de X0;
   X3 := ADST de X2;
   entra na RC(X3);
   retira-AFTD(ARTC);
   NTNT de X0 := NTNT de X0 - 1;
   se ESTR de X3 = "unidade completada" ou
      ESTR de X3 = "completado" ou
      ESTR de X3 = "esperando término" ou
      ESTR de X3 = "espernado rendezvous ou esperando término"
      então início
               sai da RC(X3);
               insere-AFTP(X2)
            fim
      senão sai da RC(X3);
   escalonador
fim.
```

```
instrução CHEN R0 /chamada de entrada.
início
   X0 := ADST de ARTC;
   entra na RC(X0);
   se ESTR de X0 = "anormal"
      então início
               sai da RC(X0);
               CP := R0
            fim;
   X1 := AP de ARTC;
   AP := AP - 1;
   X2 := ADST de X1;
   entra na RC(X2);
   se ESTR de X2 = "completado" ou
      ESTR de X2 = "esperando término" ou
      ESTR de X2 = "anormal" ou
      ESTR de X2 = "terminado"
      então início
               sai da RC(X0);
               sai da RC(X2);
               IEXC de ARTC := "índice da exceção TASKING-ERROR";
               CP := ETEX de ARTC
            fim;
   X3 := AP de ARTC;
   AP := AP - 1;
   insere-AFTR(X3,ARTC);
   NTER de X3 := NTER de X3 + 1;
   se (ESEN de X3 = "aberta") e
      (ESTR de X2 = "esperando rendezvous-servidor" ou
      ESTR de X2 = "esperando rendezvous-servidor dormente" ou
```

```
      ESTR de X2 = "esperando rendezvous ou esperando término")
      então início
               ESEN de X3 := "fechada";
               se ESTR de X2 = "esperando rendezvous-servidor
                                dormente"
                  então retira ALTD(X1);
               ESTR de X2 := "em rendezvous-ator";
               sai da RC(X2);
               sai da RC(X0);
               insere-AFTP(X1)
            fim
      senão início
               ESTR de X0 := "esperando rendezvous-ator";
               sai da RC(X2);
               sai da RC(X0)
            fim;
   escalonador
fim.
```

```
instrução AEST R0 /atualiza o estado da tarefa.
início
   X0 := ADST de ARTC;
   se ESTR de X0 = "anormal"
      então início
               sai da RC(X0);
               CP := R0
            fim;
   ESTR de X0 := ESTC de ARUC;
   sai da RC(X0)
fim.
```

```
instrução VREC R0,R1 /verifica se existe chamada para a entrada.
início
   X0 := ADST de ARTC;
   entra na RC(X0);
   se ESTR de X0 = "anormal"
      então início
               sai de RC(X0);
               CP := R0
            fim;
   X2 := AP de ARTC;
   X1 := AP;
   ETEX de X1 := R1;
   AP := AP + 7;
   TPRA de X1 := "EN"
   ARAE de X1 := ARUC;
   AERD de X1 := X2;
   PRDA de X1 := PRDN de ARTC;
   X5 := "falso";
   repita
      se AFTR de X2 = "vazio"
         então início
                  ESTR de X0 := "esperando rendezvous-servidor";
                  ESEN de X2 := "aberta";
                  sai da RC(X0);
                  escalonador
               fim;
      X3 := AFTR de X2;
      X4 := ADST de X3;
      entra na RC(X4);
      se ESTR de X4 = "anormal"
```

```
        então início
                sai da RC(X4);
                insere-AFTP(X3)
            fim
        senão início
                se ESTR de X4 = "esperando rendezvous-ator
                                  dormente"
                   então retira-ALTD(X3);
                ESTR de X4 := "em rendezvous-ator";
                sai da RC(X0);
                sai da RC(X4);
                X5 := "verdadeiro"
            fim
    até que X5
fim.
```

```
instrução ACCP R0 /inicia rendezvous.
início
   ARUC := AP - 8;
   X0 := ADST de ARTC;
   entra na RC(X0);
   se ESTR de X0 = "anormal"
      então início
               sai da RC(X0);
               CP := R0
            fim;
   X1 := AERD de ARUC;
   X2 := AFTR de X1;
   retira-AFTR(X1,X2);
   NTER de X1 := NTER de X1 - 1;
   ATRD de ARUC := X2;
   ESTC de ARUC := "em rendezvous-servidor";
   ESTR de X0 := "em rendezvous-servidor";
   sai da RC(X0);
   se PRES de ARTC < PRES de X2
      então PRDN de ARTC := PRES de X2
      senão PRDN de ARTC := PRES de ARTC
fim.
```

```
instrução FACC R0 /termina rendezvous.
início
   X0 := ATRD de ARUC;
   TOPO de X0 := TOPO de X0 - 2;
   insere-AFTP(X0);
   X1 := ADST de ARTC;
   se ESTR de X1 = "anormal"
      então início
                sai da RC(X1);
                CP := R0
            fim;
   PRDN de ARTC := PRDA de ARUC;
   AP := ARUC - 1
   ARUC := ARAE de ARUC;
   ESTR de X1 := ESTC de ARUC;
   sai da RC(X1).
fim.
```

```
instrução PREX R0,R3 /propaga exceção.
início
   X0 := ADST de ARTC;
   entra na RC(X0);
   se ESTR de X0 = "anormal"
      então início
               sai da RC(X0);
               CP := R0
            fim;
   X1 := ATRD de ARUC;
   IEXC de X1 := IEXC de ARUC;
   TOPO de X1 := TOPO de X1 - 2;
   CPTR de X1 := ETEX de X1;
   insere-AFTP(X1);
   PRDN de ARTC := PRDA de ARUC;
   X2 := ARUC;
   ARUC := ARAE de ARUC;
   IEXC de ARUC := IEXC de X2;
   ESTR de X0 := ESTR de X2;
   sai da RC(X0);
   AP := X2 - 1;
   CP := R3
fim.
```

```
instrução VRDL R0 /verifica a expressão de duração.
início
   X0 := ADST de ARTC;
   entra na RC(X0);
   se ESTR de X0 = "anormal"
      então início
               sai da RC(X0);
               CP := R0
            fim;
   X1 := AP de ARTC;
   AP := AP - 1;
   se X1 > 0
      então início
               IHDS de ARTC := X1;
               CPDL de ARTC := CP;
               ESTR de X0 := "dormente";
               sai da RC(X0);
               insere-ALTD(ARTC);
               escalonador
            fim;
   sai da RC(X0)
fim.
```

```
instrução ISEL /inicia comando select.
início
    IHDS de ARTC := "+ ∞";
    (AP + 1) de ARTC := "nada";
    (AP + 2) de ARTC := 0;
    AP := AP + 2
fim.
```

```
instrução FSEL R0 /realiza seleção.
início
   X0 := ADST de ARTC;
   entra na RC(X0);
   se ESTR de X0 = "anormal"
      então início
               sai da RC(X0);
               CP := R0
            fim;
   se AP de ARTC = 0
      então caso (AP - 1) de ARTC
               "nada" : início
                           sai da RC(X0);
                           IEXC de ARTC := "índice da exceção
                                           PROGRAM ERROR";
                           CP := ETEX de ARTC
                        fim;
              "delay": início
                           se IHDS de ARTC > 0
                              então início
                                       ESTR de X0 := "dormente";
                                       sai da RC(X0);
                                       insere-ALTD(ARTC);
                                       escalonador
                                    fim;
                           sai da RC(X0)
                           CP := CPDL de ARTC
                        fim;
        "terminante" : início
                           ESTR de X0 := "esperando término";
```

```
                    sai da RC(X0);
                    CP := CPDL de ARTC;
                fim;
        "else" : início
                    sai da RC(X0);
                    AP := AP - 2;
                    CP := CPDL de ARTC
                fim;
X1 := "vazio";
para X2 := 1, AP de ARTC faça
   início
      X3 := (AP - (2 * X2 + 1)) de ARTC;
      X4 := "vazio";
      X5 := "falso";
      repita
         se AFTR de X3 ≠ "vazio"
            então início
                    X6 := AFTR de X3;
                    X7 := ADST de X6;
                    entra na RC(X7);
                    se ESTR de X7 = "anormal"
                       então início
                                sai da RC(X7);
                                retira-AFTR(X3,X6);
                                NTER de X3 := NTER de X3 - 1;
                                insere-AFTP(X6)
                             fim
                       senão início
                                X4 := AP - (2 * X2 + 1);
                                X5 := "verdadeiro"
                             fim
```

```
                    fim
                senão X5 := "verdadeiro"
        até que X5;
        se X1 = "vazio"
           então X1 := X4;
           senão se X4 ≠ "vazio"
                    então início
                              X5 := X1;
                              X1 :≟ X4;
                              se X1 = X5
                                 então X6 := AFTR de X5;
                              X7 := ADST de X6;
                              sai da RC(X7)
                           fim
    fim;
se X1 ≠ "vazio"
    então início
             X3 := X1 de ARTC;
             X6 := AFTR de X3;
             se ESTR de X7 = "esperando rendezvous-ator
                             dormente"
                then retira-ALTD(X6);
             retira-AFTR(X3,X6);
             NTER de X3 := NTER de X3 - 1;
             AP := AP - (2 * (AP de ARTC) + 2);
             AERD de AP := X3;
             ATRD de AP := X6;
             sai da RC(X0);
             sai da RC(X7);
```

```
            CP := (X1 + 1) de ARTC
        fim;
se ((AP - 1) de ARTC = "delay") e
   (IHDS de ARTC ⩽ 0)
   então início
            sai da RC(X0);
            CP := CPDL de ARTC
         fim;
se (AP - 1) de ARTC = "else"
   então início
            sai da RC(X0);
            AP := AP - (2 * (AP de ARTC) + 2);
            CP := CPDL de ARTC
         fim;
para X2 := 1, AP de ARTC  faça
   início
      X3 := (AP - (2 * X2 + 1)) de ARTC;
      ESEN de X3 := "aberta"
   fim;
se (AP - 1) de ARTC = "delay"
   então início
            ESTR de X0 := "esperando rendezvous-servidor
                          dormente";
            sai da RC(X0);
            insere-ALTD(ARTC)
         fim
   senão se (AP - 1) de ARTC = "terminate"
            então início
                     ESTR de X0 := "esperando rendezvous ou
                                   esperando término";
```

```
                    sai da RC(X0);
                    CP := CPDL de ARTC
                fim
            senão início
                    ESTR de X0 := "esperando rendezvous
                                  servidor";
                    sai da RC(X0)
                fim;
    escalonador
fim.
```

```
instrução APAC /ajusta a pilha para executar comando accept.
início
   X0 := "vazio";
   X1 := "vazio";
   X2 := ADST de ARTC;
   entra na RC(X2);
   para X3 := 1, AP de ARTC  faça
      início
         X0 := AP - 2 * X3;
         X4 := X0 de ARTC;
         se ESEN de X4 = "fechada"
            então se X1 = "vazio"
                     então X1 := X0
                     senão X1 :=? X0
      fim;
      para X3 := 1, AP de ARTC  faça
         início
            X4 := (AP - 2 * X3) de ARTC;
            ESEN de X4 := "fechada"
         fim;
      sai da RC(X2);
      X4 := X1 de ARTC;
      X5 := (X1 + 1) de ARTC;
      AP := AP - 2 * (AP de ARTC) + 7;
      X6 := AP - 8;
      ETEX de X6 := CP;
      ARAE de X6 := ARUC;
      AERD de X6 := X4;
      PRDA de X6 := PRDN de ARTC;
      CP := X5
fim.
```

```
instrução CECA R2 /carrega o endereço do comando accept.
início
   X0 := AP de ARTC;
   (AP + 1) de ARTC := (AP - 1) de ARTC + 1;
   AP de ARTC := (AP - 2) de ARTC;
   (AP - 1) de ARTC := R2;
   (AP - 2) de ARTC := X0;
   AP := AP + 1
fim.
```

```
instrução CEAD R2 /carrega o endereço da alternativa delay.
início
   se AP de ARTC < IHDS de ARTC ou AP de ARTC ?= IHDS de ARTC
      então início
                IHDS de ARTC := AP de ARTC;
                CPDL de ARTC := R2;
                (AP - 1) de ARTC := "delay"
            fim;
   AP := AP - 1
fim.
```

```
instrução APDL  / ajusta a pilha para executar os comandos opcio-
                / nais da alternativa delay.
início
   X0 := ADST de ARTC;
   entra na RC(X0);
   para X1 := 1, AP de ARTC faça
      ESEN de ((AP - 2 * X1) de ARTC) := "fechada";
   sai da RC(X0);
   AP := AP - (2 * (AP de ARTC) + 2)
fim.
```

```
instrução CEAT R2 /carrega o endereço da alternativa terminate
                  /aberta.
início
   (AP - 1) de ARTC := "terminate";
   CPDL de ARTC := CP;
   CP := R2
fim.
```

```
instrução TERM /verifica término de tarefa.
início
   X0 := AMST de ARTC;
   execute
      X1 := ARTR de X0;
      X2 := ADST de X1;
      entra na RC(X2)
   termina se (ESTR de X2 ≠ "esperando término") e
              (ESTR de X2 ≠ "esperando rendezvous ou esperando
                             término")
      sai da RC(X2);
      X0 := AMST de X0
   fim-execute;
   se (ESTR de X2 = "completado" ou
      ESTR de X2 = "unidade completada") e
      busca(ATRC, X0, AFTD de X0)
      então início
               lib-esp(X0, AFTD de X0);
               NTNT de X0 := 0;
               sai da RC(X2);
               insere-AFTP(X1)
            fim
      senão sai da RC(X2)
fim.
```

```
func busca(X0, X1, X2) : booleano;
início
   se X1 = X2
      então busca := "verdadeiro"
      senão se X0 = X2
            então busca := busca (X0, X1, FTIR de X2) e
                           busca (X0, X2, AFTD de X2)
            senão início
                     X3 := ADST de X2;
                     entra na RC(X3);
                     se ESTR de X3 = "esperando término" ou
                        ESTR de X3 = "esperando rendezvous ou
                                      esperando término"
                        então busca := busca(X0,X1,FTIR de X2)
                                       e busca(X0,X1,AFTD de X2)
                        senão busca := "falso";
                     sai da RC(X3)
                  fim
fim.
```

```
proc lib-esp(X0,X1);
início
   se X0 ≠ X1
      então início
               lib-esp(X0, FTIR de X1);
               lib-esp(X1, AFTD de X1);
               desaloca-pilha(X1)
            fim
fim.
```

```
instrução CRGE R2 /carrega o endereço da guarda else.
início
   (AP - 1) de ARTC := "else";
   CPDL de ARTC := R2
fim.
```

```
instrução CHCE R0, R1 /chamada condicional de entrada.
início
   X0 := ADST de ARTC;
   entra na RC(X0);
   se ESTR de X0 = "anormal"
      então início
               sai da RC(X0);
               CP := R0
            fim;
   X1 := AP de ADST;
   AP := AP - 1;
   X2 := ADST de X1;
   entra na RC(X2);
   se ESTR de X2 = "anormal" ou
      ESTR de X2 = "completado" ou
      ESTR de X2 = "esperando término" ou
      ESTR de X2 = "terminado"
      então início
               sai da RC(X0);
               sai da RC(X2);
               IEXC de ARTC := "índice da exceção TASKING ERROR";
               CP := ETEX de ARTC
            fim;
   X3 := AP de ARTC;
   AP := AP - 1;
   se ESEN de X3 = "aberta" e
      (ESTR de X2 = "esperando rendezvous-servidor" ou
      ESTR de X2 = "esperando rendezvous-servidor dormente" ou
      ESTR de X2 = "esperando rendezvous ou esperando término")
      então início
```

```
                ESEN de X3 := "fechada";
                se ESTR de X2 = "esperando rendezvous-servidor
                                 dormente"
                   então retira-ALTD(X1);
                ESTR de X0 := "em rendezvous-ator";
                insere-AFTR(X3,ARTC);
                NTER de X3 := NTER de X3 + 1;
                sai da RC(X0);
                sai da RC(X2);
                insere-AFTP(X1);
                escalonador
            fim;
    sai da RC(X2);
    sai da RC(X0);
    CP := R1
fim.
```

```
instrução CHEE R0, R1 /chamada com espera de entrada.
início
   X0 := ADST de ARTC;
   entra na RC(X0);
   se ESTR de X0 = "anormal"
      então início
               sai da RC(X0);
               CP := R0
            fim;
   X1 := AP de ARTC;
   AP := AP - 1;
   X2 := ADST de X1;
   entra na RC(X2);
   se ESTR de X2 = "anormal" ou
      ESTR de X2 = "completado" ou
      ESTR de X2 = "esperando término" ou
      ESTR de X2 = "terminado"
      então início
               sai da RC(X2);
               sai da RC(X0);
               IEXC de ARTC := "índice da exceção TASKING ERROR";
               CP := ETEX de ARTC
            fim;
   X3 := AP de ARTC;
   AP := AP - 1;
   se ESEN de X3 = "aberta"  e
      (ESTR de X2 = "esperando rendezvous-servidor" ou
      ESTR de X2 = "esperando rendezvous-servidor dormente" ou
      ESTR de X2 = "esperando rendezvous ou esperando término")
      então início
```

```
            ESEN de X3 := "fechada";
            se ESTR de X2 = "esperando rendezvous-servidor
                             dormente"
               então retira-ALTD(X1);
            ESTR de X0 := "em rendezvous-ator"
            insere-AFTR(X3,ARTC);
            NTER de X3 := NTER de X3 - 1;
            sai da RC(X2);
            sai da RC(X0);
            insere-AFTP(X1)
         fim
senão início
            X4 := AP de ARTC;
            AP := AP - 1;
            se X4 ⩽ 0
              -então início
                        sai da RC(X2);
                        sai da RC(X0);
                        CP := R1;
                     fim;
               ESTR de X0 := "esperando rendezvous-ator
                             dormente";
               CPDL de ARTC := X4;
               insere-AFTR(X3,ARTC);
               NTER de X3 := NTER de X3 + 1;
               CPTR de ARTC := CP;
               sai da RC(X0);
               insere-ALTD(ARTC);
               sai da RC(X2)
            fim;
```

```
    escalonador
fim.
```

```
instrução ABRT N, R0 /marca o estado das tarefas como anormal.
início
   X0 := ADST de ARTC;
   entra na RC(X0);
   se ESTR de X0 = "anormal"
      então início
               sai da RC(X0);
               CP := R0
            fim;
   para X1 := 1, N  faça
      início
         X2 := (AP - (X1 - 1)) de ARTC;
         m-anormal(ARTC, X2)
      fim;
   sai da RC(X0)
fim.
```

```
proc m-anormal(X1, X2);
início
   X0 := ADST de X1;
   se X1 = X2
      então X3 := X0
      senão início
               X3 := ADST de X2;
               entra na RC(X3)
            fim;
   se ESTR de X3 = "anormal" ou
      ESTR de X3 = "completado" ou
      ESTR de X3 = "terminado"
      então sai da RC(X3)
      senão início
               se ESTR do X3 = "esperando término" ou
                  ESTR de X3 = "esperando rendezvous ou esperando
                                término"
                  então início
                           CPTR de X2 := CPDL de X2;
                           insere-AFTP(X2)
                        fim;
               ESTR de X3 := "anormal";
               se X1 ≠ X2
                  então sai da RC(X3);
               abort(X1, X2, ARAC de X2);
               abdes(X1, X2, AFTD de X2)
            fim
fim;
```

```
proc abort(X0, X1, X2);
início
   se TPRA de X2 ≠ "EN" e TPRA de X2 ≠ "PC"
      então início
               abdes(X0, X2, AFTD de X2);
               se X1 ≠ X2
                  então abort(X0, X1, AUAN de X2)
            fim
      senão abort(X0, X1, ARAE de X2)
fim;
```

```
proc abdes(X0, X1, X2);
início
   se X1 ≠ X2
      então início
               m-anormal(X0, X2);
               abdes(X0, X1, FTIR de X2)
            fim
fim;
```

```
instrução DSRE /desaloca RA de entradas.
início
   enquanto TPRA de ARUC = "EN"  faça
      início
         X1 := ATRD de ARUC;
         TOPO de X1 := (TOPO de X1) - 2;
         IEXC de X1 := "índice da exceção TASKING ERROR"
         CPTR de X1 := ETEX de X1;
         insere-AFTP(X1);
         ARUC := ARAE de ARUC
      fim
fim.
```

```
instrução VRDP /verifica a existência de dependentes.
início
   X0 := ADST de ARTC;
   entra na RC(X0);
   se NTNT de ARUC ≠ 0
      então início
               sai da RC(X0);
               escalonador
            fim;
   sai da RC(X0)
fim.
```

```
instrução VRTD R0, R1 /verifica término de desativação de tarefas.
início
   se ARUC ≠ ARTC
      então início
               ARUC := AUAN de ARUC;
               CP := R0
            fim;
   CP := R1
fim.
```

```
instrução TRPE R0 /termina pacote e propaga exceção.
início
   X0 := ARAE de ARUC;
   X1 := ADST de ARTC;
   entra na RC(X1);
   se ESTR de X1 = "anormal"
      então início
                sai da RC(X1);
                CP := R0
            fim;
   ESTR de X1 := ESTC de ARUC;
   sai da RC(X1);
   IEXC de X0 := IEXC de ARUC;
   AP := ARUC - 1;
   ARUC := X0;
   CP := ETEX de ARUC
fim.
```

```
instrução VREA R0 /verifica estado anormal.
início
   X0 := ADST de ARTC;
   entra na RC(X0);
   se ESTR de X0 = "anormal"
      então início
               sai da RC(X0);
               CP := R0
            fim;
   sai da RC(X0)
fim.
```

```
instrução TRPC  R0 / termina pacote.
início
   X0 := ARAE de ARUC;
   X1 := ADST de ARTC;
   entra na RC(X1);
   se ESTR de X1 = "anormal"
      então início
               sai da RC(X1);
               CP := R0
            fim;
   ESTR de X1 := ESTR de X0;
   sai da RC(X1);
   se IELV de ARUC
      então início
               IEXC de X0 := IEXC de ARUC;
               AP := ARUC - 1;
               ARUC := X0;
               CP := ETEX de X0
            fim;
   AP :=ARUC - 1;
   ARUC := X0
fim.
```

```
instrução CMUN R0 /completa unidade.
início
   X0 := ADST de ARTC;
   entra na RC(X0);
   se ESTR de X0 = "anormal"
      então início
               sai da RC(X0);
               CP := R0
            fim;
   ESTR de X0 := "unidade completa";
   ESTR de ARUC := "unidade completada";
   sai da RC(X0)
fim.
```

```
instrução AETR R0 /atualiza o estado da tarefa.
início
   X0 := ADST de ARTC;
   entra na RC(X0);
   se ESTR de X0 = "anormal"
      então início
               sai da RC(X0);
               CP := R0
            fim;
   ESTR de X0 := ESTC de (AUAN de ARUC);
   sai da RC(X0)
fim.
```

```
instrução TUNE /termina unidade e propaga exceção.
início
   X0 := AUAN de ARUC;
   IEXC de X0 := IEXC de ARUC;
   AP := ARUC - 1;
   ARUC := X0;
   CP := ETEX de X0        .
fim.
```

```
instrução TRBL /termina bloco.
início
   X0 := AUAN de ARUC;
   X1 := ARUC;
   AP := ARUC - 1;
   ARUC := X0;
   se IELV de X1
      então início
               IEXC de X0 := IEXC de X1;
               CP := ETEX de X0
            fim
fim.
```

```
instrução TRSP /termina subprograma.
início
   X0 := AUAN de ARUC;
   se IELV de ARUC
      então início
               IEXC de X0 := IEXC de ARUC;
               AP := ARUC - 1;
               ARUC := X0;
               CP := ETEX de X0
            fim;
   X1 := ENRT de ARUC;
   AP := ARUC - 1;
   ARUC := X0;
   CP := X1
fim.
```

```
instrução CMTE N / completa a tarefa e propaga exceção.
início
   X0 := ADST de ARTC;
   entra na RC(X0);
   se ESTR de X0 = "em elaboração"
      então início
               X1 := AMST de ARTC;
               IEXC de X1 := "índice da exceção TASKING ERROR";
               PCTR de X1 := ETEX de X1;
               X2 := ARTR de X1;
               X3 := ADST de X2;
               entra na RC(X3);
               NTNA de X1 := NTNA de X1 - 1;
               se NTNA de X1 = 0
                  então início
                           sai da RC(X3); insere-AFTP(X2)
                        fim
                  senão sai da RC(X3);
            fim;
   ESTR de X0 := "completado";
   X1 := "endereço de PDES de ARTC";
   para X2 := 1, N faça
      início se TDEN de X1 = "DE"
               então lexter(X1)
               senão para X3 := 1,|LSUP de X1 - LINF de X1| + 1
                        faça lexter(ARDS de X1 + (X3 - 1) * 3);
            X1 := X1 + 4
      fim ;
   sai da RC(X0)
fim.
```

```
instrução VRET R0, R3 / verifica exceção relevantada em tarefa.
início
   X0 := ADST de ARTC;
   entra na RC(X0);
   se ESTR de X0 = "anormal"
      então início
               sai da RC(X0);
               CP := R0;
            fim;
   se ESTR de X0 = "em elaboração"
      então início
               sai da RC(X0);
               X1 := AMST de ARTC;
               IEXC de X1 := "índice da exceção TASKING ERROR";
               PCTR de X1 := ETEX de X1;
               X2 := ARTR de X1;
               X3 := ADST de X2;
               entra na RC(X3);
               NTNA de X1 := NTNA de X1 - 1;
               se NTNA de X1 = 0
                  então início
                           sai da RC(X3);
                           insere-AFTP(X2)
                        fim
                  senão sai da RC(X3);
               CP := R3
            fim;
   sai da RC(X0)
fim.
```

```
instrução LIEL /liga indicador de exceção levantada.
início
   IELV de ARUC := "verdadeiro"
fim.
```

```
instrução DIEL /desliga o indicador de exceção levantada.
início
   IELV de ARUC := "falso"
fim.
```

```
instrução LVEX I, R1 /levanta exceção.
início
    IEXC de ARUC := I;
    CP := R1
fim.
```

APÊNDICE II