UNIVERSIDADE ESTADUAL DE CAMPINAS
FACULDADE DE ENGENHARIA ELÉTRICA
DEPARTAMENTO DE ENGENHARIA DA COMPUTAÇÃO E AUTOMAÇÃO INDUSTRIAL

UMA CONTRIBUIÇÃO À PROGRAMAÇÃO E ESCALONAMENTO DE SISTEMAS DISTRIBUÍDOS DE TEMPO-REAL

por: Juan Manuel Adán Coello
orientador: Prof. Dr. Mauricio Ferreira Magalhães
F. (Mauricio Ferreira

Tese apresentada à Faculdade de Engenharia Elétrica, da Universidade Estadual de Campinas, como parte dos requisitos exigidos para a obtenção do título de DOUTOR EM ENGENHARIA.

Campinas, SP - 1993

A

Clarinda

Felipe

Pedro

Dario

AGRADECIMENTOS

Ao Mauricio, pela orientação, apoio e incentivo sem os quais esta tese não teria sido escrita.

À Clarinda e à Cleonice, pelo empenho e eficência com que sempre participaram da busca de material bibliográfico.

Ao Alencar, ao Adilson e ao Sibelius pelo companheirismo e pelas enriquecedoras discussões técnicas.

À Márcia e ao Wagner, por estarem sempre dispostos a colaborar.

Ao Fábio, que propiciou um contexto favorável à realização deste trabalho por acreditar na necessidade e importância de que se pesquise.

Aos colegas do Instituto de Automação, por criarem um ambiente agradável de trabalho que tornou menos árdua esta empreitada.

Ao "Jack" e ao Krithi, pela valiosa oportunidade de participar das atividades do seu grupo de pesquisa.

Ao Krithi Ramamritham, por consentir na utilização do seu algoritmo de escalonamento como ponto de partida para a implementação do escalonador aqui proposto.

Ao CNPq, por financiar o nosso estágio na Universidade de Massachusetts, onde parte deste trabalho se desenvolveu.

Ao diretor do Instituro de Automação, Ronaldo Cardoso Lemos, da Fundação Centro Tecnológico para Informática, pelo suporte institucional que possibilitou a realização deste trabalho.

"O tempo chega sempre;
mas há casos em que não chega a tempo"

Camilo Castelo Branco

RESUMO

Um sistema de tempo-real deve atender às restrições temporais das aplicações a que está associado, e ser facilmente reconfigurável para adaptar-se às mudanças que deverão ocorrer ao longo da sua vida útil. Isto é, um sistema de tempo-real deve ser previsível e flexível.

Esta tese procura contribuir para a concepção de ambientes de desenvolvimento de sistemas distribuídos de tempo-real, mostrando que a produção de sistemas flexíveis e previsíveis é viável. Para isso, é proposto um modelo de programação suportado por duas linguages de programação (a LPM-RC e a LCM-RC) integradas a uma estratégia de escalonamento.

No modelo de programação proposto, aplicações são construídas combinando módulos de "software" reusáveis, o que lhes confere um alto grau de reconfigurabilidade. As restrições temporais de um módulo (periodicidade e prazo de término) não são parte do módulo propriamente dito, e sim do contexto onde é usado (das aplicações). Módulos podem compartilhar recursos usando os serviços oferecidos por servidores. Estes últimos, diferenciam-se dos demais módulos (clientes) por não terem restrições temporais próprias; eles herdam as restrições temporais dos clientes que estão atendendo. Uma interação cliente-servidor pode incluir tanto a execução de um único serviço como a execução de uma seqüência de serviços em exclusão mútua (relativamente aos demais clientes do servidor).

Para mostrar a viabilidade do modelo e das linguagens propostas, do ponto de vista do cumprimento das restrições temporais de aplicações, é proposta uma estratégia de escalonamento, integrada às linguagens, em três níveis: 1) escalonamento "off-line" para os módulos periódicos, 2) escalonamento dinâmico dirigido por tempo para os módulos aperiódicos e 3) escalonamento dinâmico de módulos sem restrições de tempo, usando os períodos em que não há módulos com restrições de tempo para executar.

Como parte da validação da estratégia de escalonamento, mostra-se como extrair de um programa as informações relevantes para fins de escalonamento e propõe-se um algoritmo para o escalonamento de módulos com restrições temporais críticas (periódicos). Este algoritmo, baseado em outro disponível na literatura, além de buscar um escalonamento que atenda às restrições de tempo, recursos e localização dos módulos periódicos, procura facilitar o escalonamento dinâmico de módulos aperiódicos. Isto é feito balanceando a carga do sistema e determinando "janelas de escalonamento" para os módulos periódicos, em lugar de instantes rígidos de início.

O algoritmo "off-line" foi avaliado através de simulações. Procurou-se, nas simulações, determinar a eficácia do algoritmo em encontrar escalonamentos factíveis usando diversas heurísticas. Em cada caso, foram medidas as taxas de sucesso do algoritmo e o custo computacional associado. Experimentos específicos foram conduzidos para avaliar a eficácia e o custo das estratégias de balanceamento de carga, assim como o seu impacto nas taxas de sucesso do algoritmo.

ABSTRACT

A real-time system should meet the timing requirements imposed by the application it implements, i.e., it should be predictable. A real-time system should also be easy to reconfigure in order to adapt to the changes that are likely to occur during its life time, i.e, it should be flexible.

The main objective of this thesis is to contribute to the design of distributed real-time systems development environments. Particularly, it is intended to show that the production of flexible and yet predictable systems is feasible. To reach this objective it is proposed a programming model, supported by two programming languages (LPM-RC and LCM-RC) integrated to a scheduling strategy.

In the proposed model, applications are constructed combining reusable modules of software, leading to highly reconfigurable systems. Modules' temporal restrictions (periodicity and deadline) are not attributes of the modules themselves, but instead depend on the context in which they are used (the applications). Modules can share resources with the aid of servers. Servers do not have their own timing requirements, but inherit their clients' timing constraints. A client-server interaction may involve the execution of a single service or the execution of a sequence of services in mutual exclusion from other clients.

The objective of the scheduling strategy integrated to the languages is to try to meet applications' timing constraints. The scheduling strategy is structured into three levels: 1) off-line scheduling of periodic modules, 2) time driven dynamic scheduling of aperiodic modules, and 3) dynamic scheduling of modules without timing constraints, when there are no periodic or aperiodic modules running.

As part of the validation of the scheduling strategy, it is shown how to extract from a program the information needed for its scheduling, and proposed an algorithm to schedule periodic modules. This algorithm is based on an off-line heuristic algorithm available in the literature, extended to support the peculiarities of the proposed model. The extended algorithm, searchs for a schedule that meets the timing, resource and placement constraints of periodic modules. Besides that, the algorithm also tries to balance the system load, in the time and space domains, in order to improve its fault tolerance and, particularly, the schedulability of aperiodic modules.

The off-line scheduling algorithm was evaluated applying to it sets of periodic modules with a wide range of time related characteristics. The conducted experiments aimed at determine the success rates and computational costs of the algorithm, using different heuristics. Specific experiments were also conducted to investigate the effectiveness and cost of the load balance strategies, as well as its influence on the success rate of the algorithm in finding feasible schedules.

SUMÁRIO

1. INTRODUÇÃO

O atraso no lançamento do Ônibus Espacial ("Space Shuttle") em seu primeiro vôo, a um custo significativamente elevado, é atribuído a um erro de temporização. O erro foi resultante de uma falha de sincronização entre processadores, ocorrida devido a uma sobrecarga transitória, verificada durante a iniciação do sistema [Stankovic (1988)]. Em 1981, um erro de "software" levou um robô estacionário a mover-se subitamente, a grande velocidade, para os limites de sua área operacional, esmagando um operário [Burns e Wellings (1990)].

Os dois casos acima são exemplos das graves consequências que podem advir do funcionamento inadequado de sistemas de tempo-real. Mesmo assim, os sistemas de tempo-real são virtualmente indispensáveis em um número cada vez maior de aplicações, nas mais diversas áreas.

Apesar de sua crescente importância, os fundamentos para o que se poderia chamar de uma ciência de sistemas de tempo-real ainda se encontram num estágio embrionário, sendo, inclusive, muito comum o emprego de uma série de conceitos equivocados com respeito à sua natureza [Stankovic (1988b) e Le Lann (1990)]. Por esse motivo, o objetivo inicial deste capítulo será caracterizar mais precisamente o que se entende por um sistema de tempo-real. A partir dessa caracterização, feita na seção 1.2, na seção 1.3 serão determinados os principais requisitos a serem atendidos por ambientes de desenvolvimento para esse tipo de sistemas. Tendo sido caracterizados os sistemas de tempo-real e determinados os requisitos de ambientes para o seu desenvolvimento, estará definido o contexto em que este trabalho se insere, permitindo que na seção 1.4 sejam traçados os seus objetivos.

1.1 O que é um sistema de tempo-real ?

A essência do conceito de tempo em sistemas de computação ainda é fruto de uma série de especulações, muitas vezes bastante subjetivas [Motus (1992)]). O Dicionário de Computação Oxford, citado por Burns e Wellings (1990), dá a seguinte definição para um sistema de tempo-real:

> "Qualquer sistema no qual o instante em que uma saída é produzida é significativo. Isso ocorre usualmente porque a entrada corresponde a algum movimento no mundo físico, devendo a saída relacionar-se ao mesmo movimento. O retardo entre o instante da entrada e o instante da saída deve ser suficientemente pequeno para um comportamento temporalmente aceitável".

Outra definição clássica de sistemas de tempo-real é dada por Martin (1967):

> " ... um sistema que controla um ambiente recebendo dados, processando-os e tomando ações, ou retornando resultados, suficientemente rápido para afetar o funcionamento do ambiente naquele instante".

Ambas definições cobrem uma vasta gama de sistemas computadorizados, incluindo um sistema operacional de tempo compartilhado, como por exemplo o UNIX. Pelas definições acima, esse tipo de sistema pode ser considerado de tempo-real, pois quando um usuário requer a execução de um comando ele fica esperando a sua conclusão, o que deve ocorrer "suficientemente rápido" a fim de satisfazê-lo. Da mesma forma, as definições acima incluem na categoria de sistemas de tempo-real um sistema de controle de vôo de uma aeronave. Pode-se afirmar que ambos os sistemas têm restrições temporais, porém, uma distinção clara entre eles pode ser feita analisando as consequências do não cumprimento do desempenho temporal esperado de cada um. Enquanto a não execução do comando requisitado por um usuário em alguns segundos pode deixar o usuário irritado ou frustado, o não cumprimento das restrições de tempo associadas ao sistema de controle de vôo pode ocasionar uma catástrofe. Neste último tipo de aplicação, uma resposta atrasada pode ser considerada tão inadequada como uma resposta errada.

Para uma caracterização mais precisa do que se entenderá neste trabalho por sistema de tempo-real deve-se considerar, portanto, que:

> "A correção de um sistema de tempo-real depende não somente dos resultados lógicos dos cômputos, como também dos instantes em que os resultados são produzidos." [Stankovic (1988b)].

1.2 Características de Sistemas de tempo-real

Em sistemas de tempo-real, o computador é usualmente interfaceado diretamente a algum equipamento físico cuja operação monitora ou controla. Nesses casos, os sistemas de tempo-real fazem parte de um sistema maior, motivo pelo qual são também denominados sistemas embutidos ("embedded systems"), ou dedicados.

O uso de sistemas de tempo-real como parte de sistemas maiores teve início no controle de processos industriais durante a década de 60. Atualmente, sistemas dedicados de tempo-real podem ser encontrados numa grande variedade de áreas de aplicação que incluem automação da manufatura, controle de motores de carros, sistemas de robótica e visão, indústria aeroespacial, sistemas multimídia e aplicações, militares e civis, de comando e controle. A complexidade dos sistemas de tempo-real também é muito variada, incluindo desde aplicações relativamente simples, como o controle de experimentos em laboratórios, até projetos de grande complexidade, como uma estação espacial.

A próxima geração de sistemas de tempo-real deverá ser usada em áreas de aplicação semelhantes àquelas em que são empregados os sistemas da atual geração. Eles, no entanto, serão cada vez mais complexos pois, de um modo geral, deverão ser distribuídos e capazes de exibir um comportamento inteligente, adaptativo e muito dinâmico [Stankovic (1988)]. No desenvolvimento e manutenção dessa nova geração de sistemas de tempo-real deverão evidenciar-se ainda mais as dificuldades atualmente encontradas, motivadas, em grande parte, pela escassez de técnicas e ferramentas adequadas ao seu desenvolvimento.

Um sistema de tempo-real pode ser visto como consistindo de um sistema de controle e de um sistema sendo controlado, ou ambiente. O sistema de controle interage com o seu ambiente baseado nas informações que dispõe a seu respeito, provenientes, em geral, de sensores a ele conectados. É fundamental que o estado do ambiente, tal como percebido pelo sistema de controle, seja consistente com o estado efetivo do ambiente, pois, caso contrário, a atuação do sistema de controle sobre o ambiente pode ser desastrosa.

Na maior parte das aplicações das quais os sistemas de tempo-real fazem parte, atividades (ou tarefas) que devem ser executadas em tempo-real coexistem com atividades sem restrições de tempo. Um exemplo deste último tipo de atividade é a realização de um planejamento de longo prazo numa fábrica automatizada. Idealmente, um sistema de tempo-real deve procurar assegurar que serão atendidos os requisitos temporais de todas as tarefas com restrições de tempo-real e minimizar o tempo médio de resposta das demais tarefas. Conforme será possível constatar nas discussões que se seguem, o problema de atender a requisitos temporais individuais, em lugar de minimizar tempos médios de resposta, costuma ser de difícil solução. Esse problema estabelece uma diferença fundamental entre os sistemas capazes de efetivamente suportar aplicações de tempo-real e os sistemas destinados a oferecer níveis médios de satisfação a usuários, ou altas taxas de ocupação de recursos.

Dependendo da maneira como chegam ao sistema (como são habilitadas), as tarefas de tempo-real podem ser classificadas como periódicas ou aperiódicas. Uma tarefa periódica chega ao sistema a intervalos regulares de duração conhecida, enquanto uma tarefa aperiódica chega ao sistema a intervalos aleatórios. Após a sua chegada, uma tarefa deve ser executada antes de um dado prazo, podendo a tarefa estar pronta para execução ao chegar ou somente após decorrido um intervalo de tempo especificado, isto é, o seu instante de partida ("start time") pode, ou não, coincidir com o instante de sua chegada.

Restrições temporais de tarefas são definidas a partir da qualidade do serviço que se espera do sistema e das características físicas do ambiente. A maior parte do processamento de informações obtidas de sensores é periódica por natureza. Por exemplo, a temperatura do núcleo de um reator nuclear deve ser lida periodicamente para detectar rapidamente quaisquer mudanças significativas. De forma semelhante, o acompanhamento da rota de um avião pressupõe a leitura dos dados produzidos a intervalos fixos por um radar. Tarefas aperiódicas são normalmente ativadas em função da ocorrência dinâmica de eventos, como por exemplo, um objeto caindo em frente a um robô móvel ou uma intervenção de um operador no console de um equipamento qualquer.

Além de restrições temporais, tarefas de tempo-real podem ter ainda diversos outros requisitos, muitos deles compartilhados com tarefas sem restrições de tempo. Tarefas podem querer acessar concorrentemente recursos compartilhados, ou ter que comunicar-se para cooperar. Além do processador, a execução de uma tarefa pode demandar outros recursos, tais como, dispositivos de entrada e saída, estruturas de dados e arquivos. Tarefas podem ter relações de precedência, resultantes, por exemplo, da decomposição de uma tarefa complexa em tarefas mais simples. Em sistemas distribuídos, tarefas podem ter restrições de alocação devido à necessidade de usar dispositivos de entrada e saída disponíveis em estações específicas, ou em virtude da aplicação de estratégias de tolerância a falhas.

Sistemas de tempo-real, especialmente aqueles de grande porte, são projetados com uma longa expectativa de vida, ao longo da qual devem evoluir para acompanhar as mudanças das necessidades dos seus usuários, e do próprio ambiente. A simples introdução de um sistema de computação tende a ser um estímulo a mudanças, devendo os serviços oferecidos pelo sistema evoluir com essas mudanças.

1.3 Requisitos de Sistemas de tempo-real

A partir da caracterização feita nos itens anteriores, pode-se dizer que um sistema de tempo-real tem como principais requisitos ser previsível e flexível.

Stankovic e Ramamritham (1990) definem previsibilidade como "ser possível mostrar, demonstrar ou provar que requisitos são satisfeitos, sujeitos a quaisquer suposições feitas, por exemplo, em relação a falhas ou às cargas de trabalho submetidas ao sistema.". Esta definição pode tanto ser aplicada a sistemas de tempo-real como a sistemas sem restrições de tempo, na medida em que ambos podem compartilhar uma série de requisitos. No entanto, os sistemas de tempo-real têm como requisito diferenciador restrições temporais rígidas, sendo a busca do atendimento a estas restrições, isto é, a previsibilidade temporal, o denominador comum a uma grande parte dos trabalhos atuais de pesquisa na área.

Flexibilidade, por sua vez, é a chave para que um sistema de tempo-real possa adaptar-se às mudanças evolutivas e operacionais que deverão ocorrer ao longo da sua vida operacional.

1.4 Objetivos do trabalho

O objetivo genérico deste trabalho é contribuir para a concepção e implementação de um ambiente de desenvolvimento de sistemas distribuídos de tempo-real. Os programas desenvolvidos nesse ambiente devem ter como principais características a flexibilidade e a previsibilidade temporal.

A construção de sistemas de tempo-real flexíveis pressupõe a integração de esforços em diversas áreas. Stankovic (1988, 1988b) e Le Lann (1990) destacam as seguintes: técnicas formais de especificação e verificação, metodologias de desenvolvimento, linguagens de programação, algoritmos de escalonamento, sistemas operacionais, comunicações em tempo-real, inteligência artificial, tolerância a falhas e arquitetura de sistemas. Nessas áreas, inúmeros problemas ainda aguardam soluções definitivas.

Embora em algum momento as áreas acima enumeradas devam convergir para dar origem aos sistemas de desenvolvimento do futuro, a consideração de todas seria um objetivo extremamente ambicioso, que demandaria um esforço que em muito transcende ao escopo deste trabalho. Por esse motivo, este trabalho restringe-se a buscar uma abordagem integrada às questões da programação e escalonamento de sistemas de tempo-real. O trabalho procurará contribuir com a solução dessas questões perseguindo os seguintes objetivos específicos:

1) Propor um modelo de programação flexível que permita a programação de um conjunto abrangente de classes de aplicações de tempo-real. Para tanto, o modelo deverá suportar a construção de programas distribuídos formados por tarefas com restrições de tempo, precedência, recursos e localização.

2) Propor linguagens de programação que reflitam e, portanto, suportem o modelo proposto;

3) Mostrar a validade e a viabilidade das linguagens propostas como ferramentas destinadas à produção de programas com comportamento temporal previsível. Para isto, as linguagens serão integradas a uma estratégia de escalonamento que procurará assegurar o cumprimento das restrições temporais, de precedência, de recursos e de localização das tarefas componentes de um programa.

1.5 Estrutura do trabalho

Este trabalho está organizado em seis capítulos, incluindo esta introdução, e um apêndice. A partir da caracterização e levantamento de requisitos feita neste capítulo, no capítulo 2 são determinados os requisitos de linguagens de programação e de algoritmos de escalonamento para sistemas de tempo-real, assim como, revistas algumas linguagens e algoritmos disponíveis, ou propostos, para o desenvolvimento desses sistemas. A partir das discussões dos dois primeiros capítulos, no capítulo 3 são propostos um modelo de programação para sistemas de tempo-real e linguagens de programação para suportá-lo. O capítulo 4 trata de um aspecto central à implementação do modelo e das linguagens: a estratégia de escalonamento responsável por assegurar que o comportamento temporal especificado nas linguagens será cumprido em tempo de execução. Nesse capítulo, além da proposta de uma estratégia geral de escalonamento que considera tarefas periódicas, aperiódicas e tarefas sem restrições de tempo, é proposto um algoritmo para o escalonamento de tarefas periódicas com restrições críticas de tempo . No capítulo 5, são analisados os resultados de simulações conduzidas para avaliar um escalonador de tarefas periódicas, desenvolvido a partir do algoritmo proposto no capítulo 4. No capítulo 6, são repassados os objetivos propostos para este trabalho e os resultados nele alcançados. Feito isso, são sugeridas algumas linhas de ação para lhe dar continuidade. Finalmente, no apêndice A, apresenta-se a especificação da sintaxe das linguagens propostas no capítulo 3, usando uma notação BNF estendida.

2. PROGRAMAÇÃO E ESCALONAMENTO DE SISTEMAS DE TEMPO-REAL

No capítulo 1 determinou-se que as principais características que um sistema de tempo-real deve apresentar são flexibilidade e previsibilidade. Neste capítulo, serão estabelecidos os requisitos básicos a serem atendidos por linguagens de programação e algoritmos de escalonamento destinadas à produção de sistemas de tempo-real com essas características. Além disso, serão revistas algumas linguagens, comerciais e experimentais, e alguns algoritmos de escalonamento, a fim de verificar em que medida eles atendem aos requisitos estabelecidos.

O capítulo está dividido em três seções. Na seção 2.1 serão determinados os requisitos de linguagens de programação de sistemas de tempo-real e examinado em que medida eles são atendidos por algumas linguagens atuais, disponíveis comercialmente e experimentais. Na seção 2.2 será feita a caracterização de algoritmos de escalonamento de programas, bem como brevemente analisados alguns dos principais algoritmos atualmente conhecidos. Finalmente, na seção 2.3, serão feitos alguns comentários finais e apresentado um resumo do capítulo.

2.1 Modelos e linguagens de programação de sistemas de tempo-real

2.1.1 Requisitos de modelos e de linguagens de programação de sistemas de tempo-real

Para permitir o desenvolvimento de programas flexíveis com comportamento temporal previsível, Berry e colegas (1983), Stankovic e Ramanritham (1988), Halang e Stoyenko (1990) e Burns e Wellings (1990a) destacam os seguintes requisitos a serem atendidos por modelos e linguagens de programação:

a) Permitir a especificação das restrições de tempo das aplicações. A possibilidade de especificar os requisitos de tempo de uma aplicação permite que se verifique durante a execução de um programa se esses requisitos foram ou não cumpridos e, em caso negativo, tomar ações corretivas.

b) Ser analisáveis, em tempo de compilação. Para que se possa garantir que as restrições de tempo de um programa serão cumpridas, é preciso analisar o programa em tempo de compilação ou configuração. Para isso, as linguagens de programação devem possibilitar, entre outras coisas, a determinação dos tempos máximos de execução dos programas nelas escritos.

c) Programação segura. Uma das principais características que se espera de um sistema de tempo-real é a sua confiabilidade. Modelos e linguagens de programação podem dar uma grande contribuição para produzir programas com essa característica se propiciarem uma programação segura. Ou seja, se induzirem a um estilo de programação que reduza o número de erros de programação e permita tratar situações excepcionais. Para tanto, é conveniente que as linguagens sejam modulares, façam uma verificação rígida de tipos de dados ("strong type checking"), ofereçam construções bem estruturadas e suportem, em tempo de execução, o tratamento de exceções.

d) Suportar a programação "in-the-large". A produção de módulos de software reusáveis, propiciando a programação "in-the-large", é muito importante, particularmente em sistemas de grande porte, pois isso permite uma significativa redução da complexidade, e, conseqüentemente, do custo de produção de software, além de contribuir para o aumento da sua qualidade.

e) Facilidade de manutenção. Sabe-se que boa parte dos custos associados ao ciclo de vida do "software" de sistemas com longos períodos de vida concentra-se na fase de manutenção. A manutenção do "software" pode ser muito simplificada, e os seus custos reduzidos, se forem empregadas linguagens de programação de fácil leitura, bem estruturadas e não muito complexas.

f) Suportar o tratamento flexível e seguro de interrupções. Programas de tempo-real usualmente interagem com uma grande diversidade de dispositivos.

Para simplificar o interfaceamento com esses dispositivos é importante dispor de um mecanismo flexível e seguro de tratamento de interrupções.

g) Suporte à distribuição de programas. Os sistemas a controlar em tempo-real são, em um grande número de casos, fisicamente distribuídos. Arquiteturas de "hardware" e de "software" que reflitam essa estrutura são consideradas as alternativas mais naturais para aplicaçõoes desse tipo [Steusloff, 1979]. Um adequado suporte ao desenvolvimento de sistemas distribuídos de tempo-real pressupõe o uso de abstrações e mecanismos apropriados, tanto ao nível do modelo de programação como das linguagens que o suportam.

h) Suporte ao conceito de tarefa. Sistemas de tempo-real normalmente envolvem a monitoração e controle de múltiplas atividades concorrentes e paralelas. Para que a programação desses sistemas não seja muito complexa, as linguagens de programação de tempo-real devem suportar o conceito de tarefa (ou processo) e oferecer mecanismos para que elas possam comunicar-se e sincronizar-se.

i) Permitir o uso compartilhado de recursos. Em sistemas com múltiplas tarefas devem ser oferecidos mecanismos que possibilitem que recursos sejam compartilhados, sem que se comprometa a sua consistência ou as restrições temporais das tarefas que os usam.

j) Implementação simples e eficiente. Alguns sistemas de tempo-real têm granularidade muito pequena, ou seja , caracterizam-se por atividades curtas com tempos de resposta muito pequenos. Nesses casos, a eficiencia do código gerado é importante. Linguagens simples e bem definidas são normalmente de fácil implementação e permitem a produção de código eficiente.

l) Suporte à implementação de mecanismos de tolerância a falhas. O uso de mecanismos, ativos ou passivos, de tolerância a falhas pode dar uma grande contribuição para o aumento da robustez e confiabilidade dos sistemas, permitindo, por exemplo, que falhas localizadas levem a uma degradação suave do sistema e não à sua total indisponibilidade.

A maior parte dos requisitos acima enumerados é comum a sistemas sem restrições de tempo. No entanto, eles assumem contornos próprios quando usados em sistemas de tempo-real. Neles, o suporte a cada um dos requisitos acima deve levar em consideração o cumprimento dos requisistos de tempo das aplicações [Mok, 84].

A rigor, todos os requisitos acima enumerados são importantes, entretanto, o suporte à especificação de restrições temporais e a possibilidade de analisar programas, visando oferecer garantias de que esses requisitos serão cumpridos em tempo de execução, destacam-se dos demais por serem requisitos exclusivos, e de importância fundamental, às aplicações de tempo-real.

2.1.2 Exemplos de linguagens de programação para sistemas de tempo-real

Durante muito tempo, a programação da maioria dos sistemas de tempo-real foi feita combinando linguagens de montagem com extensões de linguagens de alto nível de primeira geração, especialmente FORTRAN [IBM (1954)]. Tentando mudar esse cenário, foram projetadas algumas linguagens específicas para o desenvolvimento de sistemas de tempo-real. Entre elas, pode-se destacar CORAL 66 [IECCA (1970)] e RTL/2 [Barnes (1976)] na Inglaterra, PEARL [DIM (1979)] na Alemanha, e Ada [DoD (1980)] nos EUA. Além dessas linguagens que tiveram repercussão comercial, com diferentes níveis de sucesso, diversas outras foram propostas por pesquisadores de universidades e centros de pesquisa de vários países. Embora seja difícil afirmar que alguma delas chegará de fato a ser largamente empregada, é interessante estudá-las pois elas apresentam alternativas para superar inúmeras deficiências das linguagens atualmente disponíveis, devendo, por isso, influenciar a sua evolução, bem como, o desenvolvimento de novas linguagens comerciais.

Nesta seção serão revistas algumas das linguagens usadas na programação de sistemas de tempo-real, dando-se especial atenção ao grau de atendimento aos requisitos previamente estabelecidos, a fim de procurar avaliar a sua adequação ao desenvolvimento de sistemas distribuídos flexíveis, com comportamento temporal previsível. Entre as linguagens comercialmente disponíveis serão consideradas as linguagens de montagem, FORTRAN, PEARL e ADA, por serem as mais largamente empregadas e mais representativas dessa categoria. Em relação às linguagens experimentais, será dada uma idéia geral das principais linhas de trabalho na área, discutindo as linguagens CONIC [Magee et. al. (1989)], Euclid de tempo-real [Klingerman and Stoyenko, 1986] , Flex [Kenny and Lin (1991)], C Concorrente e Tempo-Real [Gehani e Ramamritham (1991)], linguagens orientadas a objetos e linguagens síncronas.

Linguagens de montagem

O uso de linguagens de montagem costuma ser associado à possibilidade de escrever programas eficientes em termos do tamanho e do tempo de execução do código produzido, bem como à simplicidade para acessar diretamente o "hardware". Contudo, a complexidade inerente ao desenvolvimento e manutenção de programas, especialmente quando de grande porte, é significativo. Além disso, os programas escritos em linguagens de montagem são extremamente dependentes de um determinado "hardware". No caso específico de sua utilização em sistemas de tempo-real, elas não oferecem nenhuma abstração para representar o conceito de processo, nem muito menos mecanismos que permitam a especificação de restrições temporais de programas.

FORTRAN

A linguagem FORTRAN [IBM (1954)], que originalmente não era muito mais do que uma linguagem de montagem independente do "hardware", recebeu uma série de extensões que a tornou bem mais estruturada.

A exemplo do que ocorre com outras linguagens seqüenciais de propósito geral, mecanismos específicos para desenvolvimento de programas concorrentes são oferecidos através de bibliotecas, limitando substancialmente a possibilidade de detecção prematura de inconsistências de programação durante a compilação dos programas.

Em FORTRAN, além de não serem oferecidos os meios para especificar as restrições temporais dos programas, não é possível determinar os seus tempos máximos de execução, não sendo praticável, portanto, a construção de programas com comportamento previsível.

Ada

A linguagem Ada [DoD (1980)] surgiu de uma iniciativa do Departamento de Defesa dos EUA, preocupado com os altos custos associados ao uso de uma infinidade de linguagens em seus projetos.

Ada é uma linguagem muito bem estruturada, fácil de ler, modular e dotada de um mecanismo de tipificação forte de dados. Ela permite o acesso direto ao "hardware" e a compilação separada de componentes.

Programas em Ada são estruturados em pacotes ("packages") e podem expressar concorrência usando tarefas que interagem entre si através de um mecanismo denominado "rendezvous". Um pacote agrupa uma coleção de tipos de dados, procedimentos, funções, tarefas e outros pacotes. Pacotes consistem de uma especificação e de uma implementação, tornando possível a compilação separada de cada componente. Apesar disso, a reutilização de pacotes é limitado pelo fato de um pacote poder endereçar diretamente entidades internas a outros pacotes.

Ada é freqüentemente criticada por ser muito grande e complexa. Ela inclui praticamente todas as características concebíveis numa linguagem moderna. Em consequência, é de dificil aprendizado e uso, além de requerer muito suporte em tempo de execução, o que dificulta a produção de código eficiente.

Ada, com um ambiente de suporte adequado, teve sucesso no tratamento da maioria das questões da engenharia de "software" de sistemas de tempo-real sem restrições críticas de tempo, ela falhou, no entanto, no atendimento da maior parte dos problemas associados a estes últimos [Burns e Wellings (1990)]. Embora concebida para aplicações de tempo-real, não há nada na linguagem que permita adequadamente expressar as restrições temporais de uma aplicação. Por outro lado, embora elegante e poderoso, o mecanismo de "rendezvous" permite que programas apresentem seqüências não determinísticas de execução [Giering III e Baker (1989)] e propicia que uma tarefa de alta prioridade possa ser obrigada a esperar por uma tarefa de baixa prioridade por um

período arbitrário (inversão de prioridade entre tarefas) [Sha and Goodenough (1990)], tornando impraticável determinar o tempo máximo de execução de um programa em tempo de compilação.

Consciente das limitações da versão atual da linguagem, o governo dos EUA iniciou um processo de revisão cujos resultados deverão tornar-se efetivos no final dos anos 90. Enquanto isso não ocorre, diversos trabalhos procuram contornar os problemas inerentes à linguagem, através do suporte à sua execução. Entre esses trabalhos, destacam-se os desenvolvidos por Sha, Rajkumar, e Lehoczky (1987), visando resolver o problema da inversão de prioridade entre tarefas e limitar o tempo durante o qual uma tarefa pode ser bloqueada por outra de menor prioridade.

PEARL

A linguagem PEARL ("Process and Experiment Automation Real-Time Language") foi projetada no início dos anos setenta por pesquisadores de diversos centros de pesquisa e empresas privadas da Alemanha, com suporte do Ministério da Pesquisa e Tecnologia daquele país. Atualmente, PEARL é bastante empregada na Alemanha, não tendo, no entanto, obtido a mesma aceitação em outros países.

PEARL é bem estruturada e faz uma verificação rígida de tipos de dados. Embora concebida vários anos antes que a linguagem Ada, PEARL tem um maior poder de expressão para o desenvolvimento de aplicações de tempo-real e controle de processos. Essa característica talvez possa ser explicada por ter a linguagem sido definida por engenheiros de controle, engenheiros elétricos e engenheiros químicos, a partir das sua experiências práticas em problemas de automação industrial, razão pela qual foi dada especial atenção aos aspectos temporais das aplicações [Halang e Stoyenko, 1991].

A fim de suportar a construção modular de sistemas complexos, programas em PEARL são compostos de módulos compilados separadamente. Um módulo pode conter uma divisão de sistema ("system division") e diversas divisões de problema ("problem division"). A divisão de sistema encapsula e isola o "hardware" e demais aspectos específicos do ambiente onde o programa será executado, dos algoritmos empregados na solução do problema específico. A divisão de sistema contém todas as informações necessárias à execução do programa, não sendo necessário recorrer a linguagens de comandos ou a qualquer outro mecanismo alheio ao PEARL para descrever o ambiente de execução. Dispositivos e endereços são associados a identificadores definidos pelo usuário na divisão de sistema. Esses identificadores serão referenciados nas divisões de problema em lugar dos dispositivos propriamente ditos. Assim, as divisões de problema podem dedicar-se exclusivamente à expressão algorítmica da solução de um dado problema de modo independente do ambiente.

Em 1989, foi padronizada na Alemanha uma extensão da linguagem visando suportar a programação de aplicações distribuídas, com a possibilidade de efetuar a sua reconfiguração dinâmicamente.

PEARL oferece algumas construções para expressar restrições temporais. O comportamento de programas pode ser associado a eventos temporais ou à ocorrência de interrupções:

<escalonamento por evento simples>::=
AT <expressão usando o relógio >|
AFTER <expressão de duração>|
WHEN <nome de interrupção>

ou a ciclos de execução:

<escalonamento>::=
AT <expressão usando o relógio> ALL <expressão de duração>
UNTIL <expressão usando o relógio> |
AFTER <expressão de duração> ALL <expressão de duração>
DURING <expressão de duração>|
WHEN <nome de interrupção> ALL <expressão de duração>
{UNTIL <expressão usando o relógio>| DURING <expressão de duração>}

Cinco comandos são oferecidos para permitir controlar a transição entre os estados das tarefas. Uma tarefa pode ser transferida do estado de adormecida para o estado de pronta:

[<escalonamento>] ACTIVATE <nome da tarefa> [PRIORITY <inteiro positivo>];

do estado de pronta para o estado de adormecida:

TERMINATE <nome da tarefa>;

do estado de pronta ou rodando para o estado de suspensa:

SUSPEND <nome da tarefa>

e vice-versa:

[<escalonamento por evento simples>] CONTINUE <nome da tarefa>

Quando aplicadas à tarefa requisitante, as duas últimas operações podem ser combinadas:

<escalonamento por evento simples> RESUME;

Apesar de todos os seus aspectos positivos, PEARL tem algumas limitações importantes. Entre elas pode-se destacar: 1) A sincronização entre tarefas é feita usando um esquema não estruturado e não temporizado baseado em semáforos; 2) Ausência de um mecanismo estruturado para tratamento de exceções; 3) Não é possível determinar o estado de uma tarefa ou recurso; e 4) Uma tarefa pode levar um tempo arbitrário para executar. Ou seja, a linguagem não permite a caracterização exata do comportamento temporal de programas, não sendo, portanto, possível analisá-los sob essa perspectiva [Halang e Henn, 1988; Halang e Stoyenko, 1990].

CONIC

O ambiente de programação CONIC [Magee et. al. (1989)] destina-se à construção de sistemas distribuídos de tempo-real de larga escala [Kramer et al. (1990)]. O principal objetivo do modelo de programação do CONIC é oferecer flexibilidade para permitir que programas possam acompanhar as mudanças evolutivas e operacionais que se verificam durante a (longa) vida desses sistemas. A flexibilidade é obtida estabelecendo uma clara separação entre a estrutura do sistema e a descrição funcional do comportamento de cada componente. Essa separação oferece as bases para a implementação de mecanismos de reconfiguração dinâmica [Kramer e Magee (1988)], isto é, para permitir a mudança da estrutura, ou de componentes de uma dada estrutura, sem que seja nescessário proceder a uma parada total do sistema. O desenvolvimento de aplicações seguindo esse modelo de programação é feito usando as linguagens CONIC/P e CONIC/C.

A linguagem CONIC/P, uma extensão da linguagem Pascal [Jensen e Wirth (1974)], é usada para programar os componentes básicos do sistema, denominados módulos. A independência entre a programação de um módulo e a estrutura do sistema da qual fará parte é conseguida através de uma interface constituída de portas de entrada e de portas de saída tipificadas. As variáveis internas a um módulo podem ser acessadas apenas pelos procedimentos internos ao módulo. A comunicação entre módulos dá-se exclusivamente através da troca de mensagens. Para efetuar uma troca de mensagens, os módulos endereçam as suas portas locais de comunicação. O uso de uma porta local de saída como uma referência indireta a uma porta remota de entrada, e vice-versa, é fundamental para dar aos módulos independência de interconexão e, portanto, da arquitetura do sistema no qual será usado.

CONIC/C é uma linguagem declarativa através da qual um sistema pode ser descrito como um conjunto de instâncias interligadas de módulos. Instâncias de módulos são interligadas conectando portas de saída a portas de entrada do mesmo tipo.

Apesar de conseguir com sucesso oferecer um modelo de programação flexível para o desenvolvimento de sistemas distribuídos, o CONIC tem algumas limitações importantes para o desenvolvimento de sistemas de tempo-real previsíveis. Como na maioria das abordagens convencionais à programação de sistemas de tempo-real, o CONIC não oferece nenhum mecanismo para especificar as restrições temporais das aplicações, sendo o escalonamento de instâncias de módulos feito usando preempção por prioridade. As construções da linguagem CONIC/P têm tempos de execução não determinísticos (ela oferece, inclusive, um comando de recepção seletiva, semelhante ao comando SELECT da linguagem ADA), tornando impraticável determinar os tempos máximos de execução de um programa.

Euclid de Tempo-Real ("Real-Time Euclid")

A linguagem Euclid de tempo-real [Klingerman and Stoyenko, 1986] foi projetada tendo como principal objetivo que os programas nela escritos fossem analisáveis com respeito à sua escalonabilidade. Ou seja, de modo a que se pudesse verificar, antes do início da execução de um programa, se os seus requisitos temporais podem ser

cumpridos. A definição da linguagem força todas as construções a serem limitadas no tempo e no espaço, eliminando muitos dos aspectos dinâmicos encontrados em linguagens de programação tradicionais, os quais levam a programas com comportamento imprevisível e, conseqüentemente, não analisáveis do ponto de vista de sua escalonabilidade. A limitação no tempo e no espaço das construções da linguagem traduz-se nos seguintes aspectos: 1) devem ser especificados limites de tempo para a execução de comandos iterativos, comandos de espera por eventos e em requisições de recursos; 2) a recursividade e a alocação dinâmica de espaço não são permitidos; e 3) devem ser especificadas as restrições de tempo dos processos (períodos e prazos).

O cumprimento dos requisitos temporais de um programa escrito em Euclid de tempo-real pode ser avaliado usando um verificador de escalonabilidade [Stoyenko, et al. (1991)] baseado na técnica de análise do pior caso de Leinbaugh e Yamini (1986).

Apesas de atender a diversos dos requisitos para linguagens de tempo-real enunciados no início deste capítulo, Euclid de tempo-real apresenta algumas limitações. Entre elas pode-se destacar o fato da linguagem ter sido projetada tendo como alvo sistemas estáticos com restrições críticas de tempo, operando em arquiteturas centralizadas. A linguagem não oferece, portanto, suporte adequado a eventos aperiódicos nem à distribuição de programas.

Flex

Em lugar de considerar que tarefas, ou partes de tarefas, têm tempos máximos constantes de execução, como na Euclid de tempo-real, na linguagem Flex [Kenny and Lin (1991)] tarefas são vistas como conjuntos de cômputos que podem ter um tempo de execução variável. Em função do tempo usado são produzidos resultados com distintos graus de precisão e consumidas diferentes quantidades de recursos. Esta abordagem é materializada sob a forma de modelos de programação baseados em computações imprecisas e polimorfismos de desempenho. No modelo de computações imprecisas, programas são vistos como processos iterativos que produzem resultados mais refinados à medida que dispõem de mais tempo para executar, ou ainda implementados segundo estratégias do tipo "dividir para conquistar", que fornecem resultados parciais à medida que perseguem a solução definitiva. No modelo de polimorfismos de desempenho, diversos níveis de desempenho são obtidos mediante a especificação de múltiplas versões do código que implementa uma dada função. Em tempo de execução, o sistema operacional poderá então escolher a versão mais adequada em função do tempo e recursos disponíveis.

Em lugar de exigir que o tempo e espaço requeridos para a execução das construções da linguagem sejam conhecidos em tempo de compilação, como feito em Euclid de Tempo-Real, para que um teste de escalonabilidade fosse possível, a proposta da Flex é construir modelos de desempenho dependentes dos dados a fim de poder determinar a confiabilidade estatística de atingir os requisitos temporais do programa.

Modelos de programação baseados em computações imprecisas representam soluções atraentes para diversas classes de aplicações. Quando, no entanto, for fundamental

garantir um nível mínimo de funcionalidade, eles devem ser associados a estratégias de escalonamento que garantam a execução do número de iterações ou da versão mínima que assegure essa funcionalidade. Isso envolve necessariamente a combinação das estratégias eminentemente dinâmicas para escolha de versões e repasse de resultados parciais, com estratégias estáticas de escalonamento. No caso do Flex, os resultados presentemente disponíveis ainda não permitem avaliar em que medida essa combinação foi bem sucedida, pois apenas versões preliminares (de baixo desempenho) do ambiente de suporte à execução da linguagem foram implementadas sob o sistema operacional Unix e, além disso, ainda não foi concluído o desenvolvimento de um analisador do comportamento temporal de programas [Kenny and Lin, 1991].

C Concorrente de Tempo-Real ("Real-Time Concurrent C - RTCC")

As linguagens experimentais até aqui discutidas são, com distintos graus de parentesco, descendentes das linguagens fortemente tipificadas Algol e Pascal. No entanto, a linguagem C, muito difundida atualmente e freqüentemente criticada pelos defensores da tipificação forte de dados, também está influenciando os projetistas de linguagens de tempo-real. A linguagem "Real-Time Concurrent C (RTCC)" [Gehani e Ramamritham (1991)], é um exemplo dessa influência. RTCC é uma extensão do C Concorrente [Gehani and Roome (1986)], desenvolvido nos laboratórios da AT&T, que por sua vez é uma extensão da linguagem C [Kernighan e Ritchie (1988)].

A RTCC é orientado a sistemas de tempo-real dinâmicos, ou seja, a sistemas onde atividades (tarefas) podem ser criadas a qualquer instante. Algumas dessas atividades podem ter prazos cujo cumprimento deve ser garantido, outras podem ter prazos que devem ser perseguidos sem ser fundamental o seu cumprimento, e, finalmente, algumas atividades podem ser instanciadas sem prazos. A linguagem permite também a definição de tratadores de exceções que serão executados quando não for possível garantir a execução de uma atividade ou quando o prazo de uma atividade não garantida for ultrapassado.

Na medida em que o escalonamento de tarefas no RTCC é eminentemente dinâmico, não é possível assegurar que atividades com prazos para conclusão rígidos poderão ser escalonadas a tempo de cumpri-los. Portanto, o uso da presente versão do RTCC restringe-se, a exemplo da linguagem Flex, a sistemas sem restrições críticas de tempo.

Linguagens orientadas a objetos

O sucesso das abordagens orientadas a objetos na simplificação do projeto, implementação e manutenção de software para aplicações sem restrições de tempo-real [Peterson, 1987] tem levado diversas tentativas de introduzir esse enfoque também no domínio dos sistemas de tempo-real [Bihari et. Al., 1989]. Linguagens como RTC++ [Ishikawa et. Al., 1990] e Mentat tempo-real [Grimshaw et. al., 1989], são exemplos de trabalhos nessa linha. Ambas estendem a linguagem C++, visando permitir a especificação das restrições temporais das aplicações. Esses trabalhos indicam que a orientação a objetos pode ser de grande valia no desenvolvimento de sistemas de

tempo-real, havendo no entanto ainda vários problemas sem solução adequada. Entre eles incluem-se aqueles subjacentes à alocação e escalonamento de recursos para suportar a criação e destruição dinâmica de objetos, criação de subclasses e herança na presença de restrições temporais.

Linguagens síncronas

Grande parte das dificuldades associadas ao uso da maioria das linguagens ditas convencionais na programação de sistemas de tempo-real decorre da sua natureza assíncrona e não determinística. Como alternativa a essas linguagens, foram propostas nos últimos anos, especialmente na França, linguagens denominadas síncronas. As principais representantes dessa categoria são as linguagens Signal [Le Guernic e Gautier (1990)], Lustre [Caspi et al. (1987)] e Esterel [Berry e Cosserat (1985)].

As linguagens síncronas oferecem primitivas de tempo-real "ideais" em processadores "infinitamente rápidos". Nestas linguagens as operações são executadas instantaneamente (consomem um tempo zero). A saída associada a uma operação (agente) é síncrona (simultânea) à entrada. Embora esta suposição seja bastante conveniente para raciocinar em termos abstratos sobre as propriedades temporais de programas, ela ignora uma série de problemas práticos difíceis de contornar como, por exemplo, a contenção por recursos e eventuais sobrecargas do sistema.

Deve ser destacado ainda que a utilização das linguagens síncronas é substancialmente limitada pela sua orientação a soluções centralizadas.

2.2 Algoritmos de escalonamento de tarefas

Duas abordagens principais podem ser adotadas para garantir o cumprimento do comportamento temporal de um programa: 1) usar técnicas formais de especificação que permitam que as restrições temporais do sistema sejam representadas e, a partir dessa especificação, analisadas; 2) procurar satisfazer os requisitos de tempo do programa usando algoritmos de escalonamento de recursos.

De uma maneira geral, técnicas de especificação formal requerem que todos os detalhes de um programa sejam precisamente definidos, enquanto estudos na área de escalonamento baseiam-se em características globais dos programas. Apesar da aparente falta de precisão da segunda abordagem, ela propicia uma modelagem dos recursos do sistema mais exata que a normalmente obtida com técnicas de especificação formal [Joseph e Goswani (1989)].

As técnicas de especificação formal são o objeto de interesse de destacados grupos de pesquisadores, devendo a sua importância crescer cada vez mais, acompanhando os resultados dos trabalhos em andamento. Atualmente, porém, elas são consideradas imaturas para uso na garantia do cumprimento das restrições temporais de sistemas de tempo-real grandes e complexos [Burns and Wellings (1990)]. Para poder usar uma técnica de especificação formal em aplicações reais, faz-se necessário, normalmente, empregar uma série de simplificações que muito reduz a sua utilidade. Apesar disso, técnicas de especificação e validação formais, no seu estágio de evolução atual, podem ter um papel importante na verificação da integridade do sistema com respeito às suas especificações.

2.2.1 Caracterização de algoritmos de escalonamento

Um algoritmo de escalonamento tem como funções básicas verificar se um dado conjunto de tarefas é escalonável e, em caso afirmativo, determinar um escalonamento factível. Um escalonamento factível estabelece a ordem e os instantes em que os recursos do sistema devem ser alocados às tarefas, de sorte que sejam respeitadas as suas restrições de tempo, precedência, localização e recursos (processadores, meio de comunicação, dispositivos de entrada e saída, etc.).

A determinação da escalonabilidade na maioria dos problemas de interesse para sistemas de tempo-real práticos é um problema NP-completo [Blazewicz, Lenstra and Kan (1983); French (1982); Garey and Johnson (1979)]; por isso, normalmente, a busca de escalonamentos para esses sistemas é feita por algoritmos heurísticos sub-ótimos

No caso dos sistemas distribuídos, os algoritmos de escalonamento devem considerar que um sistema de tempo-real consiste de um ou mais nós conectados por uma rede de comunicação. Cada nó sendo formado por um ou mais processadores, juntamente com o conjunto de recursos que podem ser requisitados pelas tarefas de aplicação. Em alguns sistemas, assume-se que os recursos requeridos pelas tarefas estão sempre disponíveis no instante em que a tarefa é invocada. Nesses sistemas, os processadores formam os recursos primários a serem alocados pelo escalonador.

Tarefas constituem normalmente as unidades de escalonamento de um sistema. Elas se traduzem em módulos de software que são invocados para realizar uma determinada função. Nos sistemas estáticos, o conjunto de tarefas a ser executado é previamente especificado. Nos sistemas dinâmicos, tarefas podem chegar (ser invocadas, ou habilitadas) a qualquer instante. Para os algoritmos de escalonamento, tarefas são caracterizadas pelas suas restrições temporais, de precedência, de localização e de recursos.

Tarefas que são ativadas a intervalos regulares são denominadas periódicas. Uma tarefa periódica com período P, será invocada uma vez a cada P unidades de tempo. Por exemplo, uma tarefa com período P igual a 10 unidades de tempo, será invocada nos instantes 0, 10, 20, etc. O prazo para o término de uma instância de execução de uma tarefa periódica é normalmente menor ou igual ao período da tarefa.

Tarefas que são invocados a intervalos não regulares e, quando invocadas, são executada uma única vez, são denominadas aperiódicas.

Além dos instantes de chegada, as restrições temporais de uma tarefa podem ser especificadas em termos de um instante posterior à sua chegada antes do qual a tarefa não deverá ser executada (instante de tarefa pronta, ou de partida), tempo de execução da tarefa (normalmente determinado no pior caso) e prazo para término da tarefa.

2.2.2 Escalonamento preemptivo por prioridade

O algoritmo de escalonamento mais simples usado em sistemas de tempo-real é a preempção por prioridade. Quando esse algoritmo é usado, as tarefas recebem prioridades de acordo com a sua importância para o sistema. Quanto maior a importância de uma tarefa, maior a sua prioridade. Dinamicamente, o escalonador irá sempre executar a tarefa pronta de maior prioridade.

A inadequação da preempção por prioridade para garantir restrições temporais de tarefas pode ser ilustrada pelo seguinte exemplo. Deve-se executar duas tarefas, T1 e T2, com as seguintes características: T1 tem alta prioridade, tempo de execução igual a 10 unidades de tempo e período de execução igual a 100 unidades de tempo; T2 tem baixa prioridade, tempo de execução igual a uma unidade de tempo e período igual a 10 unidades de tempo. Se a preempção por prioridade for usada, ao iniciar-se a operação do sistema, T1 será executada durante as primeiras dez unidades de tempo e, em conseqüência, T2 perderá o seu prazo, quando teria sido possível atender aos prazos das duas tarefas se tivesse sido executada primeiro T2 e em seguida T1.

O problema verificado acima decorre de a importância de uma tarefa e da sua urgência (prazo) não poderem ser diretamente mapeados em uma prioridade estática.

Apesar da sua inadequação, a maior parte dos sistemas convencionais de tempo-real usa o esquema acima para a determinação das prioridades das tarefas e para a sua posterior execução. O comportamento do sistema é então validado através de testes exaustivos, durante os quais as prioridades das tarefas são reajustadas até que o sistema aparente funcionar. Esse esquema funciona razoavelmente bem para sistemas pequenos.

Entretanto, à medida que o número de tarefas cresce, e a complexidade do sistema aumenta, faz-se muito difícil determinar um bom conjunto de prioridades para as tarefas. Por mais exaustivamente que o sistema seja testado é, no caso geral, impossível assegurar que todas os estados que o sistema poderá atingir durante a sua operação foram testados. Adicionalmente, tendo sido determinadas as prioridades para as tarefas, a sua alteração pode ser extremamente complexa, dificultando e encarecento a manutenção do sistema.

Liu e Layland (1973) foram os primeiros a estudar o escalonamento de tarefas periódicas independentes com prazos iguais aos períodos em sistemas com um único processador. Em seu artigo, hoje um clássico, propõem dois algoritmos para a solução desse problema: o algoritmo da taxa monotônica e o algoritmo do menor prazo.

2.2.3 O algoritmo da taxa monotônica ("rate-monotonic scheduling")

O algoritmo da taxa monotônica atribui prioridades fixas às tarefas na ordem inversa dos seus períodos, isto é, quanto menor for o período de uma tarefa maior será a sua prioridade.

Liu e Layland mostraram que o algoritmo da taxa monotônica é ótimo, ou seja, se num sistema com um único processador, um conjunto de tarefas independentes com prazos iguais aos períodos, puder ser escalonado por um algoritmo qualquer que use prioridades fixas, ele também poderá ser escalonado pelo algoritmo da taxa monotônica.

Uma das características mais interessantes da taxa monotônica é a possibilidade de verificar a escalonabilidade de um conjunto de tarefas através de um teste simples. Liu e Layland provaram que a condição suficiente para que um conjunto de n tarefas periódicas e independentes seja escalonável num sistema monoprocessador é que a soma das suas taxas de ocupação seja menor que

$$n * (2^{1/n} - 1)$$

A expressão acima corresponde a taxas de ocupação que decrescem monotonicamente de 0,83, quando n=2, a 0,693, quando n tende a infinito.

A taxa de ocupação de uma tarefa corresponde à fração do tempo do processador usado pela tarefa. Ela é calculada dividindo o tempo de execução da tarefa pelo seu período de ativação. Assim, uma tarefa com período de 100 unidades de tempo e 50 unidades de tempo de execução, leva a uma taxa de ocupação do processador de 0,5 (50%).

Lehoczky et. al. (1989) fizeram uma caracterização exata do algoritmo de escalonamento pela taxa monotônica. Eles determinaram testes mais complexos que o proposto por Liu e Layland para verificar as condições necessárias e suficientes para o escalonamento de um conjunto de tarefas. Esses testes permitem determinar a escalonabilidade de conjuntos de tarefas com taxas de utilização superiores aos limites estabelecidos por Liu e Layland.

A simplicidade na determinação da escalonabilidade de um conjunto de tarefas, e a simplicidade e baixo custo associados à escolha da próxima tarefa a rodar em tempo de execução, tornam o algoritmo da taxa monotônica bastante atraente. No entanto, as restrições por ele impostas ao modelo de tarefas são bastante severas. As tarefas devem ser periódicas, independentes, ter prazos iguais aos períodos e destinarem-se a um sistema monoprocessador.

Diversas restrições do algoritmo da taxa monotônica foram sucessivamente contornadas. Sha, Rajkumar e Lehoczky (1987) desenvolveram o protocolo de herança de prioridade para contornar o problema da inversão de prioridades entre tarefas que se sincronizam usando semáforos num sistema monoprocessador. Alem disso, derivaram um conjunto de condições suficientes para verificar a escalonabilidade das tarefas quando esse protocolo é usado. Posteriormente, Rajkumar et. al. (1988), estenderam o protocolo de herança de prioridade para sistemas multiprocessados. Sprunk, Sha e Lehoczky (1989) propuseram o algoritmo do Servidor Esporádico que permite combinar tarefas periódicas com tarefas aperiódicas portadoras ou não de restrições críticas de tempo.

É importante observar que, em muitos casos, os tempos máximos de execução das tarefas periódicas podem ser substancialmente maiores que os seus respectivos tempos médios de execução, podendo não ser possível garantir os recursos necessários à execução de todas as tarefas no pior caso (considerando os tempos máximos de execução de todas as tarefas). Nessas circustâncias, uma alternativa seria verificar se é possível assegurar recursos para que as tarefas críticas sempre cumpram os seus prazos (considerando os seus tempos máximos de execução) e para que as tarefas não críticas cumpram os seus prazos na maioria das vezes (considerando os seus tempos médios de execução). Usando esquemas de escalonamento baseados na atribuição de prioridades fixas (como na taxa monotônica), isso equivaleria a atribuir prioridades altas a tarefas importantes (críticas) e prioridades baixas a tarefas menos importantes (não críticas), de modo que numa situação de sobrecarga (a soma das demandas das tarefas não críticas excedendo a soma dos seus tempos médios de execução), perdas de prazos ocorressem na ordem inversa das prioridades das tarefas.

Embora interessante, a abordagem acima não pode ser implementada usando a proposta original da taxa monotônica, pois em tempo de execução, tarefas são despachadas para execução de acordo com as suas prioridades, e, na taxa monotônica, prioridades são atribuídas a tarefas na razão inversa de seus períodos, independentemente de serem críticas ou não. Logo, tarefas pouco importantes com períodos pequenos recebem prioridades maiores que tarefas muito importantes com períodos maiores. Visando contornar esse problema, Sha et. al. (1986) propuseram um método para a transformação dos períodos das tarefas, de modo que as tarefas mais importantes passem a receber as maiores prioridades. Usando esse método, os testes de escalonabilidade podem ser feitos usando tempos máximos de execução para as tarefas críticas e tempos médios para as demais. Assegurando-se assim que em situações de carga normal todas as

tarefas cumprirão os seus prazos e que, quando o sistema estiver sobrecarregado, as tarefas perderão seus prazos na ordem inversa da sua importância.

A necessidade de que o prazo das tarefas seja igual ao seu período, outra restrição da taxa monotônica, foi removida por Audsley et. al. (1991). A sua abordagem, denominada prazo-monotônico, propõe um teste complexo destinado a verificar as condições necessárias e suficientes para que conjuntos de tarefas com prazos menores que seus períodos sejam escalonáveis. A possibilidade de tratar tarefas com prazos menores que os períodos, atende aos requisitos de sistemas onde esse é o comportamento natural das tarefas e, adicionalmente, permite expressar relações de precedência entre tarefas em sistemas monoprocessadores, usando procedimentos de revisão de prazos [Mok (1984)]. Apesar de retirar uma restrição não contornada adequadamente nos trabalhos discutidos anteriormente, o prazo-monotônico mantem ainda o modelo básico de tarefas independentes executando em sistemas monoprocessadores.

Os trabalhos de extensão do algoritmo básico da taxa monotônica permitem a utilização dessa abordagem num conjunto de aplicações bem mais abrangente e realista do que era possível com o algoritmo original. Mesmo assim, problemas importantes associados a sistemas distribuídos de tempo-real não foram resolvidos. Entre eles podemos citar a questão da alocação das tarefas aos nós do sistema e o suporte a tarefas com restrições de precedência. A questão da alocação de tarefas é bastante relevante, pois o problema de poder ou não cumprir os prazos de um conjunto de tarefas está intimamente ligado à distribuição das tarefas entre os nós (ou estações) do sistema. Por sua vez, a observância das relações de precedência entre tarefas alocadas a nós distintos de um sistema distribuído não pode ser assegurada por uma mera revisão de prazos, conforme feito para sistemas centralizados.

Conforme esperado, além de aumentar a generalidade do algoritmo original da taxa monotônica, as extensões discutidas conduzem a um aumento considerável da complexidade dos testes de verificaçao de escalonabilidade e do suporte à execução das tarefas.

2.2.4 Escalonamento da tarefa com o menor prazo primeiro

Liu e Layland (1973) provaram também que, em sistemas monoprocessadores, o algoritmo de escalonamento dinâmico da tarefa com o "menor prazo primeiro" ("earliest deadline first" - EDF) é ótimo para conjuntos de tarefas periódicas independentes. Segundo o EDF, o escalonador deve dinamicamente atribuir a maior prioridade à tarefa com o prazo mais próximo. Quando o "menor prazo primeiro" é usado, um conjunto de tarefas é escalonável se a soma das suas taxas de ocupação for menor que 1. Ou seja, este algoritmo garante o cumprimento dos prazos das tarefas até o comprometimento total do processador. À primeira vista, este aspecto torna o algoritmo do menor prazo uma alternativa superior ao algoritmo da taxa monotônica. Isto é verdade somente para sistemas formados apenas por tarefas periódicos independentes que devem ser executadas em "hardware" monoprocessado. À medida que o modelo de tarefas se torna

mais complexo, não se dispõe ainda de extensões comparáveis às disponíveis para a taxa monotônica, que permitam uma aplicação mais generalizada da EDF.

2.2.5 O algoritmo de Leinbaugh

Leinbaugh (1980) desenvolveu um algoritmo de análise de programas que determina um limitante superior para o tempo de execução de cada uma das suas tarefas, a partir dos seus requisitos de recursos. O algoritmo prevê a execução de tarefas num sistema monoprocessado, considerando a contenção de tarefas no acesso a recursos compartilhados e, diferentemente da maioria dos trabalhos correlatos, incluindo na sua análise parte do "overhead" do sistema operacional necessário à execução da aplicação.

O algoritmo de Leinbaugh destaca-se por considerar um modelo de tarefas bastante realista. No entanto, as previsões por ele feitas para os tempos de execução de tarefas são bastante pessimistas, reduzindo, em consequência, a taxa de ocupação de recursos do sistema. O algoritmo básico de Leinbaugh foi estendido por Leinbaugh e Yamini (1986) para suportar a análise de programas a executar em sistemas distribuídos. Esta última versão, apesar de produzir estimativas mais precisas ainda pode ser melhorada, conforme mostrado por Stoyenko et. al. (1991).

2.2.6 O projeto SPRING

Uma das principais preocupações do projeto Spring [Stankovic e Ramamritham (1991)] tem sido o estudo de algoritmos para os sistemas de tempo-real grandes, complexos, distribuídos e dinâmicos.

A versão básica do algoritmo de escalonamento do Spring, proposta por Ramamritham e Stankovic (1984), procura garantir dinamicamente o escalonamento de tarefas independentes não preemptíveis, com requisitos de tempo e recursos. Uma versão desse algoritmo para tarefas preemptíveis foi posteriormente desenvolvido por Zhao, Stankovic e Ramamritham (1987).

Os algoritmos de escalonamento desenvolvidos no projeto SPRING são dinâmicos e descentralizados, da mesma forma que os seus sistemas alvo. Quando uma tarefa chega a um nó do sistema, o escalonador local verifica se a tarefa poderá ser executada nesse nó antes de vencido o seu prazo. Caso isso não possa ser garantido, os componentes do escalonador dinâmico localizados em cada nó do sistema cooperam para tentar determinar se algum deles tem recursos para a execução da tarefa.

O problema básico que cada escalonador local deve resolver é encontrar um escalonamento para um conjunto de tarefas num sistema com diversos recursos, podendo cada recurso ser usado em modo exclusivo ou compartilhado. Este problema é NP-árduo e, conseqüentemente, a sua solução demanda uma abordagem heurística. Zhao e Ramamritham (1987) verificaram experimentalmente que heurísticas simples não têm um bom desempenho devido à complexidade do problema. Entretanto, algoritmos que combinam essas heurísticas apresentam resultados bastante satisfatórios.

Quando um escalonador local não consegue garantir uma tarefa, ele a envia a outro nó para uma nova tentativa. A escolha do nó ao qual a tarefa será enviada é feita usando as técnicas do endereçamento dirigido ("focused addressing") e da requisição de ofertas ("bidding") [Zhao e Ramamritham (1985)]. Quando o endereçamento dirigido é usado, uma tarefa que não puder ser garantida no nó ao qual chegou, será enviada para um nó cuja disponibilidade de recursos estima-se suficiente para executá-la a tempo de cumprir o seu prazo. Na técnica da requisição de ofertas, o nó que não consegue garantir localmente uma tarefa envia uma requisição de ofertas para um conjunto de outros nós. Aqueles que tiverem recursos disponíveis responderão ao nó requisitante, fazendo uma oferta de recursos. O nó requisitante escolherá então a melhor oferta e encaminhará a tarefa a escalonar ao nó que a fez.

Os algoritmos da requisição de ofertas e do endereçamento dirigido foram comparados entre si, com um algoritmo flexível que combina as duas técnicas e com um algoritmo que decide aleatoriamente para que nó enviar uma tarefa que não pode ser garantida localmente [Stankovic, Ramanritham e Cheng (1985) e Ramanritham, Stankovic e Zhao (1989)]. Foi observado que o algoritmo flexível tem melhor desempenho que os demais isoladamente. Esse melhor desempenho é, no entanto, desprezível se comparado ao do algoritmo de requisição de ofertas quando os retardos de comunicação entre os nós são pequenos. O mesmo pode ser dito a respeito do desempenho dos algoritmos flexível e de endereçamento dirigido, quando os retardos de comunicação forem grandes. Notou-se ainda que o algoritmo aleatório tem um desempenho muito bom, se comparado ao algoritmo flexível, quando a carga computacional do sistema é baixa e, também, quando a carga é alta e desbalanceada.

Ainda dentro do projeto Spring, Cheng, Stankovic e Ramamritham (1986) desenvolveram um algoritmo sofisticado para dinamicamente escalonar grupos de tarefas com relações de precedência e prazos de execução. Decidir dinamicamente como distribuir um grupo de tarefas com relações de precedência num sistema distribuído é um problema complexo. A maioria dos algoritmos estáticos que consideram relações de precedência requerem um conhecimento completo de todas as tarefas do sistema e são computacionalmente onerosos; sendo, por isso, impraticável usá-los dinamicamente. No algoritmo de Cheng, Stankovic e Ramamritham, quando um grupo de tarefas chega a um nó, é feita uma tentativa de escalonamento local. Quando isso não é possível, o grupo é particionado em subgrupos que são distribuídos entre outros nós para uma tentativa de escalonamento paralelo. A distribuição de subgrupos entre os nós é feita usando um algoritmo que combina o endereçamento dirigido e a requisição de ofertas. A complexidade da maior parte desse algoritmo é quase linear ao tamanho do problema. Apesar disso, o seu custo de execução é significativo e, portanto, tem bom desempenho apenas quando as folgas dos grupos de tarefas são grandes (diferença entre o prazo das tarefas e o tempo requerido para sua execução).

Apesar de simulações terem mostrado que os algoritmos dinâmicos distribuídos mencionados até aqui apresentam um bom desempenho, o seu uso restringe-se a sistemas onde as tarefas não tem restrições críticas de tempo, pois não é possível assegurar que

toda tarefa chegando ao sistema será escalonada a tempo de cumprir o seu prazo. O escalonamento de tarefas com restrições críticas de tempo passa necessariamente pelo uso de algoritmos estáticos. Para suprir essa lacuna, Ramamritham (1990) desenvolveu um algoritmo estático para a alocação e escalonamento de tarefas periódicas com restrições de precedência e tolerância a falhas. A tolerância a falhas traduz-se na necessidade de alocar réplicas de algumas tarefas a nós distintos do sistema distribuído. Simulações mostram que a busca heurística efetuada pelo algoritmo é bastante efetiva. Os resultados das simulações sugerem que, se existe uma alocação e escalonamento factíveis para um dado conjunto de tarefas, o algoritmo consegue na maioria das vezes encontrá-los sem necessidade de efetuar "backtrackings".

O algoritmo estático de Ramamritham visa garantir "off-line" que as tarefas periódicas com restrições críticas de tempo cumprirão seus prazos, deixando a cargo de um algoritmo "on-line" (dinâmico) a busca de escalonamentos factíveis para as tarefas com chegadas dinâmicas. O escalonador dinâmico deve, naturalmente, procurar escalonar as tarefas chegando ao sistema sem comprometer as garantias dadas estaticamente às tarefas com restrições críticas. Nesse sentido, o algoritmo de Ramamritham apresenta um inconveniente não desprezível. A alocação e escalonamento geradas pelo algoritmo são, em geral, muito desbalanceadas no tempo e no espaço. O desbalanceamento no tempo ocorre porque o algoritmo procura escalonar as tarefas periódicas o mais cedo possível, concentrando os períodos de ociosidade dos processadores no final do intervalo de escalonamento (o mínimo múltiplo comum dos períodos das tarefas). Desse modo, as tarefas que dinamicamente chegarem no início do intervalo de escalonamento terão reduzidas possibilidades de serem executadas a tempo de cumprir os seus prazos. O desbalanceamento no espaço traduz-se na concentração de tarefas em alguns nós do sistema, deixando outros ociosos, o que pode também comprometer o escalonamento de tarefas chegando a estações sobrecarregadas.

Ramamritham e Adán (1990) incorporaram ao algoritmo descrito acima estratégias heurísticas destinadas a gerar alocações e escalonamentos melhor balanceados, procurando com isso aumentar a escalonabilidade de tarefas aperiódicas e melhorar a tolerância a falhas do sistema. Estudos experimentais mostraram que essas estratégias efetivamente propiciam alocações e escalonamentos sensivelmente melhor balanceados, sem redução perceptível das taxas de sucesso do algoritmo.

Os diversos algoritmos desenvolvidos no projeto Spring tratam com diversos modelos de tarefas. Somados eles abordam um amplo domínio de problemas de escalonamento encontrados em sistemas de tempo-real. No entanto, esses algoritmos ainda não foram integrados e, por conseguinte, a sua aplicação se restringe às classes específicas de aplicações que se mapeiam nos diversos modelos de tarefas suportados.

2.3 Resumo e comentários finais

Linguagens para a produção de sistemas distribuídos de tempo-real previsíveis e flexíveis devem atender a três requisitos básicos: 1) Suportar a especificação dos requisitos temporais das aplicações; 2) Ser analisáveis com respeito ao seu comportamento temporal; e 3) Produzir componentes reusáveis, tanto do ponto de vista funcional como temporal e localizacional.

As abordagens tradicionais para a programação de sistemas de tempo-real, baseadas no uso de linguagens de montagem e extensões de linguagens seqüenciais, não atendem aos dois primeiros requisitos e costumam não atender ao terceiro, ou atendê-lo de forma restrita.

As principais linguagens disponíveis comercialmente e projetadas especificamente para a programação de sistemas embutidos de tempo-real, Pearl e Ada, também não atendem aos três requisitos. Ada não oferece mecanismos adequados para a especificação de restrições temporais, nem é possível determinar os tempos máximos de execução dos programas nela escritos. Embora seja uma linguagem muito bem estruturada, os seus componentes (pacotes), tem o seu reuso limitado por endereçarem diretamente entidades internas a outros pacotes. Pearl oferece uma série de construções para a especificação de retrições temporais, é bem estruturada e suporta sistemas distribuídos. Apesar disso, programas escritos em Pearl podem ter tempos de execução arbitrários, o que também impede que se possa verificar se as suas restrições temporais serão observadas.

Algums ambientes e linguagens de programação experimentais oferecem atendimento, em distintos níveis, aos requisitos acima. Por exemplo, o ambiente CONIC [Magee et. al. (1989)] está baseado na produção de modulos reusáveis, suportanto um modelo de programação flexível que oferece elementos para a implementação de reconfiguração dinâmica de aplicações. CONIC, no entanto, não permite a especificação de restrições temporais, e seus programas podem também ter tempos arbitrários de execução. Euclid de Tempo Real permite a especificação de restrições temporais e a determinação de tempos máximos para a execução de programas. A linguagem, no entanto, não oferece suporte adequado a eventos aperiódicos, nem à distribuição de programas. Outras linguagens, como Flex [Kenny and Lin (1991)], C Concorrente de tempo-real [Gehani e Ramamritham (1991)], RTC++ [Ishikawa et. Al., 1990] e RT Mentat [Grimshaw et. al., 1989], oferecem suporte a sistemas dinâmicos cujas tarefas têm restrições temporais. As restrições temporais das tarefas devem ser garantidas dinamicamente, não sendo possível, portanto, assegurar que os prazos de tarefas críticas serão sempre cumpridos.

Escalonar um programa de modo que os seus requisitos de tempo sejam cumpridos é um problema relativamente simples quando o modelo de programação empregado também o é. Por exemplo, quando um programa é composto apenas por tarefas independentes e periódicas, os algoritmos "taxa monotônica" e "menor prazo primeiro" podem ser usados com sucesso. À medida que o modelo de programação torna-se mais elaborado, e mais adequado às necessidades de aplicações mais sofisticadas, o escalonamento dos programas resultantes torna-se mais complexo, sendo em geral

necessário combinar algoritmos simples, como o "menor prazo primeiro", com estratégias heurísticas mais elaboradas.

O Spring [Stankovic e Ramamritham (1991)] é o projeto mais destacado na aplicação desta última abordagem. Nele foram desenvolvidos diversos algoritmos heurísticos que cobrem uma boa parte dos problemas de escalonamento encontradas no desenvolvimento de sistemas distribuídos de tempo-real. Esses algoritmos, no entanto, ainda não foram integrados, de modo que a aplicação de cada um deles restringe-se a uma classe específica de sistemas.

3. PROGRAMAÇÃO DE SISTEMAS DE TEMPO-REAL NO HSTER

Embora seja freqüente tentar pôr em prática modelos de programação específicos usando linguagens de programação não desenvolvidos para o modelo - por exemplo, programação estruturada em linguagem de montagem, ou programação orientada a objetos em C ou Pascal - essa abordagem costuma oferecer resultados modestos. O seu sucesso depende intimamente de que os projetistas e implementadores de sistemas sigam rigorosamente disciplinas específicas de projeto e programação, propósito dificilmente alcançável sem o auxílio de ferramentas computadorizadas. Além disso, em muitos casos, a falta de mecanismos nas linguagens de programação que efetivamente suportem o modelo, virtualmente inviabiliza a sua implementação. Este é o caso dos sistemas de tempo real. Conforme enfatizado no capítulo dois, não é possível garantir o cumprimento das restrições temporais de um programa se não houver meios para especificá-los e, mesmo quando isso pode ser feito, se não for possível determinar o tempo máximo de execução do programa.

Neste capítulo serão propostos um modelo para a programação de sistemas distribuídos com comportamento temporal previsível e linguagens de programação para suportar esse modelo. Os principais objetivos do modelo e das linguagens são permitir a produção de programas previsíveis e flexíveis. Ou seja, de programas que além de cumprir o comportamento temporal especificado durante a sua programação, possam ser facilmente reconfigurados de modo a acompanhar, com um pequeno esforço, as mudanças dos requisitos das aplicações.

A proposta apresentada neste capítulo constitui uma evolução do ambiente STER [Adán-Coello, Lopes e Magalhães (1987)]. O STER, baseado no ambiente CONIC, oferece um modelo de programação centrado na construção de aplicações a partir de módulos de "software" reusáveis.

A implementação do primeiro protótipo do STER é descrita em [Adán Coello (1986)] e em [Lopes (1986)]. Esta primeira versão, suportava o desenvolvimento e execução de aplicações centralizadas, tendo sido posteriormente estendida para suportar aplicações distribuídas [Guimarães, Lopes, Adán-Coello e Magalhães (1989)]. Estas versões do STER foram utilizadas nos Laboratórios de Engenharia de Software oferecidos pela Escola Brasileiro-Argentina de Informática (EBAI) nos anos de 1989, em Termas de Rio Hondo (Santiago del Estero, Argentina), e 1991, em Nova Friburgo (Rio de Janeiro, Brasil). Cópias do STER foram também distribuidas a algumas universidades que o utilizaram em cursos práticos e como suporte para o desenvolvimento de pesquisas nas áreas de sistemas concorrentes e de sistemas de tempo-real [Lopes (1991)].

O uso experimental do STER mostrou a flexibilidade do modelo de programação que emprega. No entanto, a exemplo do CONIC, o STER não oferece mecanismos para especificar as restrições temporais das aplicações, nem é possível determinar o tempo máximo de execução de um programa, condições essenciais para que possam ser produzidos programas com comportamento temporal previsível.

A proposta aqui apresentada procura conservar a flexibilidade do STER e introduzir os mecanismos necessários para suportar a especificação e o cumprimento das restrições temporais das aplicações.

O modelo e as linguagens propostas neste capítulo, juntamente com a política de escalonamento a ser discutida no capítulo 4, podem ser vistos como os elementos básicos necessários ao desenvolvimento de um ambiente de programação de sistemas distribuídos de tempo-real, denominado, neste trabalho, HSTER ("Hard real-time STER").

Este capítulo tem a seguinte organização: na seção 3.1 serão discutidos os principais elementos do modelo de programação proposto e, nas seções 3.2 e 3.3, serão apresentadas, respectivamente, as linguagens LPM-RC e LCM-RC para suportar o modelo de programação.

3.1 O modelo de programação do HSTER

O modelo de programação do HSTER, a exemplo do seu predecessor o STER, tem como preocupação central possibilitar a produção de componentes de "software" reusáveis, visando suportar o conceito de programação "in the large". No HSTER, a reusabilidade de componentes é funcional, como em modelos de programação não orientados a tempo-real, e temporal, para atender aos requisitos específicos desses sistemas. A reusabilidade temporal é conseguida desacoplando a especificação funcional dos componentes básicos usados na composição de programas, denominados módulos, da especificação dos seus requisitos temporais. Um módulo deve ser projetado para desempenhar uma tarefa genérica cujas restrições temporais serão determinados quando da sua instanciação numa aplicação específica.

3.1.1 Etapas no desenvolvimento de uma aplicação

Embora não seja o propósito deste trabalho elaborar uma metodologia de desenvolvimento de software de tempo-real, é interessante observar que o modelo proposto sugere que o desenvolvimento de uma aplicação seja guiado pelos seguintes passos (com as devidas realimentações e refinamentos sucessivos):

- Decomposição do sistema em módulos - Esta atividade tem grande importância para todo o ciclo de vida do software. Módulos cuidadosamente projetados podem ser reutilizados em um grande número de aplicações. A decomposição do sistema pode ser guiada pelas abordagens funcional [DeMarco (1981)] ou orientada a objetos [Meyer (1987)], combinadas com outros critérios genéricos [Parnas (1972)] ou específicos para sistemas de tempo-real [Mok (1984b)].

- Definição das interfaces dos módulos - Tem como objetivo principal permitir o refinamento, e posterior codificação, de cada módulo independentemente dos demais. A interface de um módulo define o tipo de dados que ele troca com o exterior (os demais módulos) e os sentidos em que as trocas ocorrem.

- Configuração lógica da aplicação - Tem por propósito definir a estrutura de um programa em termos dos módulos que o compõem e da maneira como estes se interconectam. Não se especifica, nessa etapa, a localização física dos módulos, nem as suas restrições temporais. Esses aspectos podem aparecer apenas em termos lógicos, estabelecendo, por exemplo, que um módulo será periódico, sem que a duração do período seja especificado.

- Definição das restrições temporais dos módulos - As restrições temporais de módulos estão associadas às aplicações às quais eles se destinam, a periodicidade e os prazos dos módulos que formam um programa devem ser determinados a partir dos requisitos temporais do processo monitorado ou controlado.

- Implementação dos módulos - Nesta etapa, cada um dos módulos pode ser codificado independente dos demais.

Deve-se notar que o desenvolvimento de uma aplicação nem sempre pressupõe a estrita observância dos cinco passos acima. Por exemplo, nos dois passos iniciais deve-se

levar em consideração os módulos já disponíveis e, quando pertinente, reutilizá-los. Quando um módulo é reutilizado, o quinto passo não é, evidentemente, necessário para esse módulo.

3.1.2 O módulo

Módulos são as entidades básicas a partir das quais são construídos programas distribuídos. Um módulo encapsula o conjunto de dados e procedimentos sequenciais necessários à implementação de uma dada tarefa. Os dados internos a um módulo somente podem ser diretamente manipulados pelos procedimentos do próprio módulo.

Embora autônomos para realizar uma atividade específica, módulos devem cooperar entre si para atingir os objetivos comuns das aplicações que compõem. Esta cooperação será conseguida através do mecanismo de troca de mensagens.

Para permitir que módulos possam ser reutilizados com um mínimo de esforço, na troca de mensagens são sempre endereçadas portas locais aos módulos. Posteriormente, na etapa de configuração de uma aplicação, são definidos canais lógicos de comunicação conectando portas de saída a portas de entrada dos módulos que compõem a aplicação.

3.1.3 Restrições temporais de módulos

Módulos podem ter padrões periódiocos ou aperiódicos (esporádicos) de ativação. Uma vez ativado, um módulo pode, ou não, ter um prazo para concluir a sua execução. Um módulo que não tem associado um prazo para concluir a sua execução é dito um módulo sem restrições de tempo ou, simplesmente, um módulo NRT ("Non Real-Time Module"). Um módulo com prazo crítico, isto é, com um prazo que deve ser respeitado em qualquer circunstância, é denominado um módulo com restrições críticas de tempo, módulo crítico, ou simplesmente, módulo HRT ("Hard Real-Time Module"). Um módulo com prazo não crítico, isto é, com um prazo cuja perda eventual pode ser tolerada, será denominado módulo sem restrições críticas de tempo, ou simplesmente, módulo SRT ("Soft Real-Time Module").

Os padrões de ativação e os tipos de requisitos temporais (críticos ou não críticos) de módulos são normalmente função do contexto em que os módulos são usados (da aplicação) e não dos módulos propriamente ditos. Por exemplo, em algumas situações, a tarefa desempenhada por um módulo pode ter que ser realizada periódicamente a intervalos conhecidos, enquanto que em outras, de forma esporádica para tratar eventos asssíncronos, gerados pelo sistema que está sendo controlado, ou por outros módulos.

Para que seja possível buscar um escalonamento que permita atender aos requisitos de tempo de uma aplicação, é fundamental que os tempos máximos de execução dos módulos que formam a aplicação sejam conhecidos. Este reqüisito tem grande influência na definição das linguagens de programação associadas ao modelo. Construções com tempos de execução tipicamente não determinísticos, como comandos iterativos, alocação dinâmica de memória e chamadas recursivas, não devem ser oferecidas, ou

oferecidas com algumas restrições que permitam determinar o tempo máximo necessário para sua execução.

Um dos principais objetivos de um modelo de programação é ofececer as abstrações mais adequadas à solução dos problemas inerentes a um certo domínio de aplicação. Limitações de ordem teórica e prática devem, no entanto, ser observadas. No caso específico dos sistemas de tempo-real, é fundamental ter sempre presente que a determinação da escalonablidade de um conjunto de tarefas é, no caso geral, um problema computacionalmente intratável.

Em sistemas de tempo-real é possível identificar diversas atividades com ocorrência aperiódica e requisitos críticos de tempo (tratamento de sinais de emergência, por exemplo). Como, porém, não é viável assegurar dinâmicamente que esses módulos sempre cumprirão os seus prazos, dada a complexidade inerente ao problema, no modelo de programação proposto, e nas linguagens que o suportam, não se permite que módulos aperiódicos tenham restrições críticas de tempo, ou seja, eles serão sempre tratados como módulos SRT. Para contornar essa restrição, os módulos aperiódicos que, sob qualquer circustância, devem necessariamente cumprir os seus prazos, devem ser convertidos em módulos periódicos. A transformação de um módulo aperiódico em periódico é feita estimando o intervalo mínimo entre suas chegadas (período). A transformação de um módulo aperiódico em periódico permite que o escalonador reserve recursos para a sua execução, recursos que o sistema operacional deve procurar reaproveitar sempre que o módulo não for ativados nos períodos previstos.

3.1.4 Servidores

Módulos podem ter a necessidade de compartilhar recursos (arquivos, estruturas de dados, dispositivos periféricos, etc.). No HSTER, o compartilhamento de recursos é controlado por módulos especiais denominados servidores. O esquema cliente-servidor é bastante flexível, sendo totalmente compatível com o requisito de reusabilidade de módulos, central ao modelo HSTER.

Módulos servidores (ou simplemente servidores) são os agentes responsáveis por oferecer serviços relacionados a um recurso compartilhado. Servidores devem poder atender às requisições de múltiplos clientes sem comprometer a integridade do recurso compartilhado.

Módulos clientes (ou simplesmente módulos) e servidores têm estrutura análoga, ambos são compostos de dados e procedimentos. Ambos interagem com os demais componentes de uma aplicação através da troca de mensagens endereçadas a uma interface local formada de portas de entrada e saída. No entanto, servidores, ao contrário de módulos, não têm requisitos de tempo próprios. Os instantes de ativação e prazos para execução de um servidor são sempre função dos instantes em que os serviços são requisitados e dos prazos dos módulos requisitantes (clientes).

Um relacionmento cliente-servidor é estabelecido conectando portas dos módulos clientes a portas dos módulos servidores durante a construção de uma aplicação. Desse

modo, clientes e servidores podem ser reutilizados em diversas aplicações sem que seja necessário reprogramá-los, ou mesmo recompilá-los.

Um módulo cliente pode relacionar-se com um servidor usando portas de saída unidirecionais ou bidirecionais. Portas unidirecionais são usadas quando o serviço requisitado não gera nenhum resultado que o servidor deva retornar ao cliente. Portas bidirecionais são empregadas quando o cliente deve esperar por uma resposta decorrente da requisição feita.

3.1.5 Operações multiserviço

Muitas vezes, um único acesso a um servidor não é suficiente para realizar uma determinada operação. Um exemplo típico é a atualização de um item de uma base de dados compartilhada entre vários módulos por intermédio de um servidor. A operação de atualização envolve os seguintes passos: a) o módulo cliente envia uma mensagem ao servidor requisitando a leitura do item desejado; b) o servidor é ativado pela chegada da mensagem de requisição de leitura, acessa o item especificado e o repassa ao cliente numa mensagem de resposta; c) o cliente modifica o item e o reenvia ao servidor numa mensagem de requisição de escrita (ou atualização); d) o servidor recebe a mensagem de escrita e atualiza o item na base de dados. Claramente, a sequência de operações acima pode conduzir a base de dados a um estado inconsistente se for realizada concomitantemente por vários clientes referenciando o mesmo item.

Em sistemas de programação concorrente usando o conceito cliente-servidor, a manutenção da consistência de um recurso compartilhado, em situações como a ilustrada acima, é frequentemente garantida através de serviços específicos. Esse enfoque poderia ser aplicado ao exemplo acima incluindo na mensagem de requisição de leitura um campo para indicar que o recurso deve ficar à disposição exclusiva do módulo requisitante, até que o módulo explicitamente libere o recurso (usando uma mensagem específica, ou um campo de uma mensagem). A implementação desse acesso exclusivo poderia então ser feita simplesmente enfileirando as requisições que, após o recurso ter sido protegido, chegassem de outros clientes, até que o recurso fosse liberado.

Embora o enfileiramento de requisições pendentes até a liberação do recurso possa ser considerado adequado em sistemas sem restrições de tempo, o mesmo não acontece em sistemas de tempo-real. O não determinismo decorrente do enfileiramente não permite a determinação dos tempos máximos de execução dos módulos clientes, possibilitando, por exemplo, que um módulo crítico perca o seu prazo por não poder acessar um servidor alocado a um módulo não crítico.

O não determinismo no acesso a um recurso compartilhado pode ser eliminado exigindo que quando um cliente solicitar um serviço ele defina um tempo máximo de espera pela sua conclusão (um "time-out"). Decorrido esse tempo, o cliente seria notificado pelo sistema operacional e retomaria a sua execução a fim de tomar as providências necessárias à manutenção da consistência do sistema (o que não costuma ser trivial!).

No esquema acima, o "time-out" pode ser usado para a determinação do tempo máximo de execução do cliente . Essa estratégia tem alguns inconvenientes. Um deles é a dificuldade de determinar tempos de espera razoáveis que não levem, quando subdimensionados, a que um grande número de requisições não possa ser atendida ou, quando superdimensionados, a que sejam estimados tempos máximos de execução para os clientes muito pessimistas. Outro inconveniente, associado ao esquema de chamadas temporizadas, é que muitas vezes não se pode tolerar que um serviço importante não seja efetivamente realizado. Não se deve desprezar também o fato da manutenção da consistência do sistema não ser trivial quando um serviço é abandonado antes da sua conclusão, ou quando o serviço é concluído, porém, o cliente não é notificado (levando-o a comportar-se como se o serviço não tivesse sido concluído).

Para que uma operação envolvendo a requisição de múltiplos serviços, como a operação de atualização ilustrada no exemplo acima, não leve o servidor a um estado inconsistente ou ao comprometimento dos requisitos temporais do cliente, é necessário que todos as requisições sejam tratadas como integrantes de uma única operação atômica, e o servidor execute uma operação atômica por vez. Ou seja, após o servidor iniciar o atendimento do primeiro serviço de uma dada operaçao, ele não deve aceitar requisições de nenhuma outra operação até que a primeira operação seja concluída. A inclusão de parâmetros nas mensagens de início e conclusão de uma operação (ou acesso) multiserviço para delimitá-la, a fim de que o servidor se encarregue de processar apenas as requisições associadas à operação, não constituem uma solução adequada. Quando isso é feito, o escalonador não tem meios de automaticamente determinar se uma mensagem está sendo usada para uma operação monoserviço ou se ela inicia um acesso multiserviço. Para que o escalonador possa tratar adequadamente uma operação multiserviço é necessário o uso de construções específicas que delimitem claramente a região crítica definida pela operação.

Conforme será visto na seção 3.2, o HSTER propõe uma construção específica, denominada LOCK, para permitir que um módulo cliente delimite as requisições de serviços que definem uma dada operação atômica.

3.1.6 Tratamento de interrupções

Conforme destacado no capítulo 2, sistemas de tempo-real têm como uma de suas principais características a necessidade de interagir com um grande variedade de dispositivos periféricos, cada qual com características peculiares. A interação entre esses dispositivos e os sistemas de controle ocorre normalmente por intermédio de interrupções. As rotinas responsáveis pelo tratamento de interrupções foram durante muito tempo programadas em linguagem de montagem. Apenas recentemente linguagens de programação têm procurado oferecer mecanismos que permitem tratar interrupções num nível mais elevado.

No HSTER, o tratamento de interrupções é feito por módulos aperiódicos, habilitados pela chegada de mensagens que representam a ocorrência das interrupções que devem tratar. Para isso, o suporte à execução de programas gera, quando da ocorrência de uma

interrupção, uma mensagem e a deposita numa porta do módulo tratador, associada à interrupção durante a configuração da aplicação.

3.1.7 Tratamento de exceções relacionadas às restrições temporais de módulos

O cumprimento das restrições temporais dos módulos de uma aplicação é da máxima importância nos sistemas de tempo-real. Se durante a execução de uma aplicação for constatado que uma restrição temporal foi violada ou que pode ser violada, uma providência adequada deve ser tomada.

No modelo de programação proposto, três situações excepcionais relacionadas ao cumprimento das restrições temporais de uma aplicação podem requerer um tratamento específico durante a execução de um programa: 1) a não chegada das mensagens de habilitação de um módulo periódico; 2) a impossibilidade de garantir que o prazo de um módulo aperiódico será cumprido e 3) a perda do prazo de um módulo.

As mensagens de habilitação de módulos periódicos são usadas para estabelecer relações de precedência. A ausência das mensagens de habilitação é detectada no instante em que o módulo deveria iniciar a sua execução, conforme determinado pelo escalonador. Esta exceção pode ser devida a uma falha do módulo emissor da mensagem ou, no caso de módulos remotos, à falha da estação que contém o módulo origem, ou à perda da mensagem durante a sua transmissão.

Conforme discutido no capítulo 2, quando da chegada (habilitação) de um módulo aperiódico, dois tipos de algoritmos podem ser usados para procurar escaloná-lo: 1) algoritmos que não garantem que os prazos das tarefas serão cumpridos; e 2) algoritmos que fazem testes de aceitação, a fim de assegurar que a execução de uma tarefa somente será iniciada se for possível assegurar o cumprimento do seu prazo. A geração de uma exceção associada à perda do prazo de um módulo ocorre quando for usado um algoritmo do primeiro tipo, enquanto que a geração de uma exceção associada à impossibilidade de garantir que o prazo de um módulo será cumprido ocorre quando for usado um algoritmo do segundo tipo.

No HSTER é proposto um mecanismo que propicia o tratamento de exceções ao nível dos módulos e ao nível do sistema. Ao nível dos módulos, permite-se que sejam definidas rotinas tratadoras para cada um dos três tipos de exceções temporais descritos. Ao nível do sistema, define-se a política a adotar quando uma dada exceção for sinalizada. Pode-se nesse nível: a) comandar a execução do tratador definido pelo módulo; b) ignorar a presente instância de execução do módulo; e c) suspender permanentemente o módulo.

3.1.8 Níveis de abstração de um programa

O uso de módulos como os elementos básicos para a composição de um programa caracteriza um modelo de programação com dois níveis de abstração: o módulo e o programa (ou aplicação). Entretanto, no desenvolvimento de programas de larga escala,

o uso de apenas dois níveis de abstração pode conduzir a programas de estrutura muito complexa.

A fim de contribuir com a redução da complexidade da estrutura de programas, particularmente quando de grande porte e distribuídos, definem-se as entidades grupo e estação para representar níveis intermediários de abstração entre a entidade módulo e a entidade aplicação.

Um grupo é formado por módulos e grupos com forte coesão funcional, traduzida em relações de precedência e uma restrição temporal única. Durante a construção de uma aplicação, os componentes de um grupo são tratados como uma única entidade, podendo além de uma restrição temporal única, ter associada uma restrição de alocação comum.

Uma estação lógica é formada por módulos e grupos com restrições temporais próprias e não necessariamente possuidores de relações de precedência. Normalmente, os componentes de uma estação lógica apresentam coesão funcional, porém num grau menor que a verificada entre membros de grupos. Uma estação lógica pode, ou não, ser alocada a uma única estação física (ou nó). Estações lógicas podem ainda ter que ser alocadas a nós específicos do sistema em função dos dispositivos de entrada e saída que requerem, ou devido à implementação de estratégias de tolerância a falhas baseadas em redundância.

3.2 Linguagem de programação de módulos com restrições críticas de tempo (LPM-RC)

Nesta seção a LPM-RC será informalmente apresentada, utilizando uma extensão da notação BNF (vide apêndice A) e fragmentos de código. Nos fragmentos de código apresentados, reticências (...) serão usadas para indicar trechos do código facilmente subentendidos ou que não são relevantes para o entendimento do assunto em discussão.

A LPM-RC é uma extensão da linguagem Pascal [Jensen e Wirth (1974)], razão pela qual a apresentação a seguir se concentrará nos aspectos da linguagem específicos para a programação de sistemas de tempo-real. A descrição completa da sintaxe das extensões que a LPM-RC faz ao Pascal encontra-se no apêndice A.

Um programa em LPM-RC define um tipo de módulo, a partir do qual poderão ser criadas instâncias (processos) durante a configuração de uma aplicação. Um tipo de módulo é, portanto, um conceito estático, enquanto uma instância é um conceito dinâmico. O termo módulo será usado neste trabalho significando ora tipo de módulo ora instância de módulo, devendo o contexto ser suficiente para distinguir entre os dois significados. Apenas ocasionalmente, quando o termo módulo resultar ambíguo, será feita uma distinção explícita entre as duas acepções. Um módulo tem a seguinte estrutura:

```
[UNBOUNDED] [NONPREEMPTABLE]
    MODULE <identificador> [(<parâmetros formais>)];
        [<definição de contexto>]
        [<declaração de portas de interface>]
        [<definição de condição de habilitação>]
        [<definição de rótulos>]
        [<definição de constantes>]
        [<definição de tipos>]
        [<declaração de variáveis>]
        [<declaração de funções e procedimentos>]
        [<declaração do iniciador>]
        [<declaração de tratador de exceção temporal>]*
    BEGIN
        <comandos>
    END.
```

3.2.1 Cabeçalho de um módulo

Módulos podem, ou não, estar associados à execução de tarefas com restrições de tempo, ou seja, módulos podem, ou não, ter prazos a cumprir. A alocação de recursos (processador, memória, etc.) aos módulos, a fim de que eles possam cumprir os seus prazos, é responsabilidade do escalonador de módulos, conforme discutido no capítulo 2. Para que o escalonador possa alocar recursos aos módulos, é necessário que ele conheça as suas demandas por recursos, em especial, é necessário que ele conheça o tempo

máximo de execução dos módulos. Para que isso seja possível, a LPM-RC estabelece algumas restrições para a codificação de módulos que poderão ser instanciados com restrições temporais (por "default", todos os módulos). A esses módulos, qualificados como limitados no tempo ou "bounded", não é permitido fazer alocação dinâmica de memória, recursão e nem usar comandos iterativos (FOR, WHILE e REPEAT) cujo tempo máximo de execução não possa ser antecipado.

Para executar um módulo sem restrições de tempo não é necessário conhecer o seu tempo máximo de execução, de modo que não é necessário limitar o tipo de construções que ele pode usar. Módulos não limitados no tempo devem ser explicitamente definidos como "unbounded".

Deve-se observar que, em aplicações onde módulos com restrições de tempo convivem com módulos sem restrições de tempo, o sistema operacional deve ser capaz de suspender a execução de um módulo sem restrições de tempo sempre que isto seja necessário para cumprir o prazo de um módulo com restrições de tempo. Isto é, módulos sem restrições de tempo devem ser sempre preemptíveis. Por outro lado, módulos com restrições de tempo são preemptíveis por "default". Quando, em virtude da tarefa que implementam, módulos não forem preemptíveis, na sua definição deve ser usado o qualificador "nonpreemptable".

Os parâmetros formais de um módulo declaram constantes locais ao módulo que deverão ser definidas durante a instanciação do módulo. Essas constantes devem ser de um dos tipos básicos do Pascal (BOOLEAN, CHAR, INTEGER e REAL). Por exemplo, abaixo,

```
MODULE m(Ref:INTEGER);
...
VAR Int:INTEGER;
...
BEGIN
    ...
    Int:= Ref;
    ...
END.
```

define-se o tipo de módulo m que declara como parâmetro formal a constante Ref do tipo inteiro. O valor de Ref será definido em tempo de configuração da aplicação (quando for instanciado o módulo m).

3.2.2 Definição de contexto

Módulos são entidades compiladas separadamente. A fim de evitar a redundância de definições comuns a diversos módulos, elas devem ser reunidas em unidades de definição, de onde poderão ser importadas pelos módulos que as compartilham. Além de permitir que definições sejam compartilhadas, as unidades de definição podem ser muito úteis para a implementação de ferramentas de controle automático de configurações,

permitindo, por exemplo, que toda vez que uma definição for alterada sejam recompilados automáticamente todos os módulos que a utilizam.

Unidades de definição contêm, portanto, definições de tipos e constantes e declarações de procedimentos e funções que podem ser importadas por um módulo através da cláusula "use". Por exemplo, a unidade de definição UDefA:

```
DEFINE UDefA:TipMem,i;
    USE TamFam:UDefA0;
    TYPE
        TipMem =
            RECORD
            ...
            END;
        i = 1..TamFam;
END.
```

define os tipos TipMem e i. Na definição do intervalo i foi usada a definição da constante TamFam, importada da unidade de definição UDefA0. Esta, por sua vez, poderia ter a seguinte estrutura:

```
DEFINE UDefA0:TamFam,...;
    CONST
        TamFam = ...;
        ...
END.
```

A definição de TipMem, feita na unidade de definição UDefA, poderia ser importada pelo módulo m, conforme ilustrado a seguir:

```
MODULE m(Int:INTEGER);
    USE TipMem: UDefA;
    ...
BEGIN
    ...
END.
```

Via de regra, os símbolos especificados numa definição de contexto são usados para a especificação da interface dos módulos pois, para que dois ou mais módulos possam ser conectados, é necessário que eles possuam portas complementares (saída/entrada) de mesmo tipo.

3.2.3 Declaração de portas de interface

A interface de um módulo é definida pelas suas portas de comunicação. Quando um módulo deseja enviar uma mensagem ao exterior (outro módulo) ele referencia uma porta

local de saída. Do modo análogo, quando um módulo quer receber uma mensagem do exterior, ele endereça uma porta local de entrada.

Portas definem os tipos das mensagens trocadas entre os módulos e o sentido em que as trocas ocorrem. Portas unidirecionais permitem o fluxo de mensagens em apenas um sentido, saída ou entrada, enquanto portas bidirecionais suportam o trânsito de mensagens em ambos os sentidos.

Portas unidirecionais

Uma porta unidirecional define um fluxo de mensagens de sentido único. Se o fluxo for de entrada a porta será denominada de entrada e, em caso contrário, de saída.

Durante a configuração de uma aplicação, portas de saída devem ser conectadas a portas de entrada de mesmo tipo. Por exemplo, o código abaixo,

```
MODULE ma;
    USE TipMem: UDefA;
    EXITPORT ps1,ps2:INTEGER;
             ps3:TipMen;
    ...
BEGIN
    ...
END.
```

declara duas portas de saída de tipo inteiro, ps1 e ps2, e uma porta de saída do tipo TipMen. Durante a configuração de uma aplicação, ps1 e ps2 devem ser conectadas a portas de entrada do tipo inteiro e ps3 a uma porta de entrada de tipo TipMen, como seria, por exemplo, o caso das portas pe1,pe2 e pe3 definidas da seguinte forma:

```
...
ENTRYPORT
    pe1,pe2:INTEGER;
    pe3:TipMen;
...
```

Portas bidirecionais

Portas bidirecionais permitem o fluxo de mensagens entre um módulo e o exterior em ambas as direções. Quando o fluxo deve iniciar-se com o envio de uma mensagem, a porta é dita de saída e, em caso contrário, de entrada. Por exemplo,

```
...
EXITPORT psb1:INTEGER  REPLY SIGNAL;
...
```

declara uma porta bidirecional de saída de nome psb1, através da qual o módulo que a declara poderá enviar mensagens de tipo inteiro e receber mensagens do tipo predefinido SIGNAL. A porta psb1 é dita bidirecional de saída, pois uma mensagem somente poderá entrar por essa porta em resposta a uma mensagem a ela enviada. De modo análogo,

```
...
ENTRYPORT peb1:INTEGER REPLY SIGNAL;
...
```

define uma porta de entrada bidirecional, compatível com psb1 (do mesmo tipo de psb1). Por peb1 serão recebidas mensagens de tipo inteiro e, em resposta, serão enviadas mensagens do tipo SIGNAL.

O tipo SIGNAL, empregado nas definições acima, está associado a mensagens sem conteúdo definido, usadas para indicar a ocorrência de um determinado evento.

Famílias de portas

Portas de mesmo tipo e semanticamente equivalentes podem ser declaradas usando o conceito de família de portas. Uma família de portas, de entrada ou saída, é definida e referenciada de forma análoga a um vetor Pascal. Por exemplo,

```
...
ENTRYPORT
      fpe1[1..10]:INTEGER;
      Alarme[1..NumUnidades]:INTEGER REPLY SIGNAL;
...
```

declara duas famílias de portas de entrada, fpe1 e Alarme. A família fpe1 é composta de dez portas de entrada unidirecionais de tipo inteiro, endereçadas da seguinte maneira: fpe1[1], fpe1[2], ... e fpe1[10]. A família Alarme é composta de NumUnidades portas de entrada bidirecionais, por onde serão recebidas mensagens do tipo INTEGER e, em resposta, enviadas mensagens do tipo SIGNAL. A especificação,

```
MODULE ExFam(TamFamS2:INTEGER);
      USE i:UDefA;
      EXITPORT fps1[i]:CHAR REPLY SIGNAL;
            fps2[1..TamFamS2] : INTEGER;
      ...
END.
```

declara as famílias de portas fps1 e fps2. A família fps1 é declarada com o auxílio do identificador de tipo i, definido como sendo o intervalo 1..TamFam na unidade de definição UDefA (v. item 3.2.2). A família fps1 é, portanto, composta de TamFam portas de saída. Por cada porta dessa família deverão ser enviadas mensagens do tipo CHAR e recebidas, em resposta, mensagens do tipo SIGNAL. A família fps2 é declarada com o auxílio do parâmetro formal TamFamS2, a ser definido durante a criação de instâncias de ExFam.

3.2.4 Portas bloqueantes X portas não bloqueantes

O envio de mensagens através de portas de saída unidirecionais é sempre não bloqueante, ou seja, o módulo que envia uma mensagem nunca será bloqueado devido à execução da primitiva de envio da mensagem. A recepção de mensagens de portas de entrada unidirecionais pode ser bloqueante ou não. A recepção será bloqueante quando

uma porta de entrada for usada para receber uma das mensagens de habilitação do módulo, sendo nos demais casos não bloqueante.

Como o envio de mensagens através de portas unidirecionais é não bloqueante, ou assíncrono, num canal de comunicação definido pela conexão de uma porta de saída unidirecional a uma porta de entrada unidirecional, é possível que o módulo emissor gere mensagens a uma velocidade superior à capacidade de consumo do módulo receptor (o módulo que contém a porta de entrada destinatária). Quando isso ocorrer, apenas a mensagen mais recente será armazenada na porta de entrada. Se for desejado que mais de uma mensagem seja armazenada, uma fila de espera deve ser associada à porta. Por exemplo,

```
...
ENTRYPORT peb:TipMen QUEUE 5;
...
```

define uma fila capaz de armazenar as cinco últimas mensagens que chegaram à porta peb e que ainda não foram lidas.

Portas bidirecionais estão sempre associdas a primitivas bloqueantes, empregadas em relacionamentos cliente-servidor. Através de portas de saída bidirecionais um módulo cliente pode enviar uma requisição de serviço a um servidor e esperar uma resposta a esse pedido. O servidor correspondente, por sua vez, será habilitado pela chegada da mensagem de requisição a uma porta de entrada bidirecional, através da qual ele enviará a resposta sendo aguardada pelo cliente.

3.2.5 Definição da condição de habilitação de um módulo

Independentemente das suas restrições temporais, módulos podem ter condições de habilitação. Condições de habilitação são expressas em termos da chegada de mensagens a determinadas portas de entrada de um módulo.

Um módulo pode ser habilitado pela chegada de pelo menos uma mensagem a cada membro de um dado conjunto de portas ou, opcionalmente, pela chegada de uma mensagem a qualquer porta desse conjunto. No primeiro caso, a condição de habilitação do módulo será especificada usando a cláusula "enabled by all ..." e, no segundo, a cláusula "enabled by any ...". Por exemplo, o módulo mb, abaixo,

```
MODULE mb;
    USE TipMem: UDefA;
    ENTRYPORT pe1,pe2:integer; pe3:TipMen;
    ENABLED_BY ALL pe1,pe3;
    ...
BEGIN
    ...
END.
```

estará habilitado para execução quando pe1 e pe3 tiverem recebido uma mensagem. Por outro lado, o módulo mc, abaixo,

```
MODULE mc;
    USE TipMem: UDefA;
    ENTRYPORT
        pe1,pe2:integer;
        pe3:TipMen;
    ENABLED_BY ANY pe1,pe2,pe3;
    ...
BEGIN
    ...
END.
```

estará habilitado quando pe1, ou pe2, ou pe3 tiver recebido uma mensagem.

A satisfação da condição de habilitação de um módulo é necessária mas nem sempre suficiente para que ele seja considerado pronto para executar. A chegada das mensagens de habilitação é condição suficiente para que possa ser iniciada a execução de um módulo aperiódico. No caso de um módulo periódico, além da chegada das mensagens de habilitação, é necessário também que um novo período de execução tenha-se iniciado.

As condições de habilitação do tipo ANY introduzem não determinismo na habilitação de um módulo e, em consequência, no fluxo de execução do programa que usa o módulo. Por esse motivo, condições de habilitação do tipo ANY não podem ser utilizadas em módulos periódicos, pois isso tornaria impraticável garantir a sua execução mediante o uso de técnicas de escalonamento "off-line".

3.2.6 Declaração de mensagens

Mensagens são entidades que muito se assemelham às variáveis locais de um módulo. A diferença entre uma mensagem e uma variável de mesmo tipo reside no fato de que uma mensagem poderá ser enviada, ou recebida, através de uma porta de comunicação, enquanto que, uma variável não. O código a seguir ilustra a declaração de mensagens:

```
MODULE mc;
    USE TipMem: UDefA;
    ENTRYPORT
        pe1,pe2:INTEGER;
        pe3:TipMen;
    ENABLED_BY ANY pe1,pe2,pe3;
    MESSAGE
        m1,m2: INTEGER;
        m3: TipMen;
    ...
BEGIN
    m1:=1;
```

```
    ...
END.
```

O módulo mc, acima esboçado, declara duas mensagens de tipo inteiro (INTEGER), m1 e m2, e uma mensagem de tipo TipMen, m3. A declaração explícita de mensagens permite que se estabeleça uma clara separação entre as entidades primariamente associadas à interface do módulo (mensagens), e as entidades responsáveis pelo armazenamento do estado do módulo (variáveis).

3.2.7 Declaração do iniciador de um módulo

Antes que o corpo de um módulo seja executado pela primeira vez, o seu estado inicial pode ser definido usando um iniciador. A execução de um iniciador deve ter efeito local, motivo pelo qual não se permite que iniciadores façam uso das primitivas de troca de mensagens. O módulo md, mostrado a seguir, ilustra a declaração de um iniciador:

```
MODULE md(TempRef:INTEGER);
    USE NumDeUnidades: ...;
    ...
    VAR
        TemperaturaDaUnidade:ARRAY[1..NumDeUnidades] OF INTEGER;

INITIATE;
    VAR i:INTEGER;
BEGIN
    FOR i:=1 TO NumDeUnidades DO TemperaturaDaUnidade[i]:=TempRef;
END;

...
BEGIN
    (* corpo do módulo *)
    ...
END.
```

O iniciador definido pelo módulo md providenciará para que antes do corpo desse módulo, delimitado por BEGIN e END, ser executado pela primeira vez, os elementos do vetor TemperaturaDaUnidade sejam iniciados com o valor TempRef, passado como argumento a md durante a configuração da aplicação. As variáveis declaradas no iniciador, do mesmo modo que as variáveis declaradas no corpo de procedimentos, são dinâmicas e, como tais, têm escopo local ao iniciador.

3.2.8 Declaração de tratadores de exceções temporais

Tratadores de exceções temporais permitem especificar qual deve ser o comportamento de um módulo em três tipos de situações: 1) durante a execução do módulo, vence o seu prazo; 2) o escalonador dinâmico não é capaz de assegurar que poderá cumprir o prazo de um módulo que acaba de ser ativado; e 3) no instante

determinado pelo escalonador para o início da execução de um módulo, constata-se que as suas mensagens de habilitação ainda não chegaram.

Tratadores de exceções definem as ações que podem ser tomadas no âmbito do módulo. A efetiva execução de um tratador, quando o sistema operacional sinalizar a ocorrência de uma exceção, dependerá da política especificada em nível de aplicação.

Um tratador de exceção tem uma estrutura semelhante à de um iniciador, com a diferença de que em seu corpo poderá ser usado qualquer comando LPM-RC, pois um tratador representa, de fato, uma nova versão do módulo.

Tratador da perda do prazo de um módulo

O módulo mostrado a seguir ilustra a estrutura típica de um tratador de perda de prazo:

```
MODULE CalcTemperaturaDaUnidade(TempRef:INTEGER);
    USE NumDeUnidades: ...;
    ENTRYPORT
        PRecalcTemperaturaDaUnidade:SIGNAL;
    EXITPORT
        PTemperaturaDaUnidade[1..NumDeUnidades]:INTEGER;
    ENABLED_BY PRecalcTemperaturaDaUnidade;
    MESSAGE
        MSignal:SIGNAL;
        MTemperaturaDaUnidade:INTEGER;
    VAR
        TemperaturaDaUnidade:ARRAY[1..NumDeUnidades] OF INTEGER;
        i:INTEGER;
    FUNCTION RecalcTemperaturaDaUnidade(Unidade:INTEGER):INTEGER;
    ...

    HANDLER MISSED_DEADLINE;
    BEGIN(* handler *)
        (* reconduz as variáveis internas a um estado consistente *)
        ...
        MTemperaturaDaUnidade:=TempRef;
        FOR i:= 1 TO NumDeUnidades DO
            SEND MTemperaturaDaUnidade TO PTemperaturaDaUnidade[i];
    END;(*handler*)

BEGIN(* module*)
    RECEIVE MSignal FROM PRecalcTemperaturaDaUnidade;
    FOR i:= 1 TO NumDeUnidades DO
        BEGIN
            MTemperaturaDaUnidade:=RecalcTemperaturaDaUnidade(i);
            SEND MTemperaturaDaUnidade TO PTemperaturaDaUnidade[i];
        END;
END. (*module*)
```

No exemplo acima, no instante em que vencer o prazo do módulo CalcTemperaturaDaUnidade, será sinalizada uma exceção pelo sistema operacional e, caso especificado na configuração da aplicação que a exceção deve ser tratada, a execução do corpo do módulo será suspensa, passando-se o controle para o tratador de prazo perdido. O tratador zelará pela manutenção da consistência das variáveis internas ao módulo (ele pode ter sido interrompido dentro da função RecalcTemperaturaDaUnidade) e no encaminhamento imediato de valores precalculados à família de portas PTemperaturaDaUnidade.

Tratadores para módulo não garantido e para falha de habilitação

Usualmente, um tratador associado a um prazo perdido deve preocupar-se com a manutenção do estado interno do módulo. O mesmo não ocorre com os tratadores de módulo não garantido e ausência das mensagens de habilitação. Esses tratadores serão executados no lugar do corpo do módulo. O código abaixo exemplifica a declaração de tratadores para esses dois tipos de exceções:

```
MODULE CalcResC(Ref:INTEGER);
    USE
        TOcorrênciaAnormal, CodModNãoGarantido, CodFalhaDeHab: UDFx;
    ENTRYPORT
        PResA,PResB:INTEGER;
    EXITPORT
        PResC:INTEGER;
        POcorrênciaAnormal:TOcorrênciaAnormal;
    ENABLED_BY ALL PResA, PResB;
    MESSAGE
        MResA,MResB,MResC:INTEGER;
        MOcorrênciaAnormal:TOcorrênciaAnormal;
    FUNCTION RecalculeRes(ResA,ResB:INTEGER):INTEGER;
    ...

    INITIATE;
    BEGIN
        MResA:=Ref;
        MResB:=Ref;
    END;

    HANDLER NON_GUARANTEED;
    BEGIN
        WITH MOcorrênciaAnormal DO
        BEGIN
            IdInstância:=MODULE_ID;
            CodOcorrência:=CodModNãoGarantido;
        END;
        SEND MOcorrênciaAnormal TO POcorrênciaAnormal;
    END;
```

```
        HANDLER ENABLING_FAULT;
        BEGIN
            WITH MOcorrênciaAnormal DO
            BEGIN
                IdInstância:=MODULE_ID;
                CodOcorrência:=CodFalhaDeHab;
            END;
            SEND MOcorrênciaAnormal TO POcorrênciaAnormal;
            RECEIVE MResA FROM PResA
                FAIL => ;
            RECEIVE MResB FROM PresB
                FAIL => ;
            MResC:=RecalculeRes(MResA,MResB);
            SEND MResC TO PResC;
        END;

    BEGIN
        RECEIVE MResA FROM PResA ;
        RECEIVE MResB FROM PresB ;
        MResC:=RecalculeRes(MResA,MResB);
        SEND MResC TO PResC;
    END.
```

Onde, a unidade de definição UDFx poderia ter sido definida do seguinte modo:

```
    DEFINE UDFx;
        CONST
            CodModNãoGarantido = ...;
            CodFalhaDeHab = ...;
        TYPE
            TOcorrênciaAnormal =
                RECORD
                    IdInstância:INTEGER;
                    CodOcorrência:INTEGER;
                END;

        ...
    END.
```

Neste exemplo, quando o módulo CalcResC não puder ter sua execução garantida, o tratador correspondente enviará uma mensagem contendo o identificador da instância associada à exceção e um código indicativo da ocorrência. A identificação da instância sendo executada é obtida usando a função predefinida MODULE_ID. Caso alguma mensagem necessária à habilitação do módulo não estiver disponível no instante em que ele deveria iniciar a sua execução (conforme determinado por um escalonamento previamente calculado) o tratamento será análogo ao caso anterior e, adicionalmente,

será computado MResC usando os mais recentes valores de MResA e MResB (os conteúdos das últimas mensagens de habilitação recebidas).

Tanto o corpo do módulo CalcResC como o seu tratador de ausência das mensagens de habilitação fazem a leitura das mensagens de habilitação usando o comando RECEIVE, a ser discutido na próxima seção. Se não houver uma mensagem disponível na porta endereçada pelo comando RECEIVE, a sua cláusula FAIL será imediatamente executada.

Deve ser observado que para uma dada instância do módulo CalcResC, apenas um dos dois tipos de tratador poderá ser executado. Isto ocorre porque a exceção NON_GUARANTEED somente será gerada quando o módulo for aperiódico e, conseqüentemente, tiver que ser escalonado dinâmicamente, enquanto que a exceção ENABLING_FAULT, poderá ocorrer apenas se o módulo for instanciado como periódico, caso em que a sua execução será garantida "off-line".

3.2.9 O comando SEND

O comando SEND pode ser usado tanto para envios assíncronos como para envios síncronos de mensagens. O envio assíncrono é não bloqueante, realizando-se através de uma porta de saída unidirecional. Por exemplo, no código,

```
...
EXITPORT psa1:CHAR;
...
MESSAGE MCar:CHAR;
...
MCar:='a';
SEND MCar TO psa1;
...
```

após o envio da mensagem MCar à porta psa1, o módulo prosseguirá a sua execução, sem esperar por nenhum tipo de confirmação relativo à entrega de MCar. Após a execução do comando SEND, MCar poderá imediatamente ser modificada, pois ela já foi transmitida ou copiada para algum depósito ("buffer") do sistema operacional.

O envio síncrono é bloqueante, realizando-se através de uma porta de saída bidirecional. Após um envio síncrono, o módulo emissor será bloqueado até a chegada de uma mensagem de resposta, transmitida pelo módulo que recebeu a mensagem que iniciou a comunicação síncrona. Por exemplo, no trecho de código abaixo,

```
...
EXITPORT pss1:CHAR REPLY SIGNAL;
...
MESSAGE
    MCar:CHAR;
    MSignal:SIGNAL;
...
MCar:='x';
```

```
SEND MCar TO pss1 WAIT MSignal;
...
```

após o envio da mensagem MCar através da porta pss1, o módulo será bloqueado até a chegada da mensagem de resposta MSignal a pss1. Se o envio síncrono de uma mensagem for efetuado por um módulo com tempo de execução determinístico (módulo "bounded"), ele deverá incluir uma cláusula de tratamento de falha para evitar que o módulo fique indefinidamente à espera por uma mensagem que poderá não chegar nunca, ou que poderá chegar atrasada, impedindo que o módulo cumpra o seu prazo. Em módulos sem tempo de execução determinístico (módulos "unbounded"), a cláusula de tratamento de falha é opcional.

Falhas num envio síncrono podem ser decorrentes da tentativa de enviar uma mensagem a uma porta de saída que durante a configuração da aplicação não foi conectada, da falha do módulo destinatário (encarregado de gerar a mensagem de resposta), ou, caso o módulo destinatário seja remoto (localizar-se em outra estação física), da falhas da estação destinatária ou do meio de comunicação.

O exemplo abaixo ilustra o uso do comando SEND com a opção FAIL:

```
MODULE x;
    ...
    EXITPORT pss1:CHAR REPLY SIGNAL;
    ...
    MESSAGE
        MCar:CHAR;
        MSignal:SIGNAL;
        ...
BEGIN
    ...
    MCar:='x';
    SEND MCar TO pss1 WAIT MSignal
        => c1s;
           ...
           cns;
        FAIL
        =>
           CASE REASON OF
              ELINK:c1f;
              ETIMEOUT:c1f;
              ...
           END;
    ...
END.
```

O código acima especifica que deve ser enviada a mensagem MCar através da porta pss1 e, através dessa mesma porta, recebida a mensagem MSIGNAL. Se a mensagem de resposta MSIGNAL for recebida, devem ser executados os comandos c1s a cns. Alternativamente, caso uma falha seja detectada, deve ser executado o comando "CASE

REASON OF ...". Na cláusula de tratamento da falha, a razão da falha poderá ser determinada usando a função predefinida REASON. Esta função poderá retornar os códigos predefinidos "ELINK", indicando que a porta de saída pss1 não está conectada, ou "ETIMEOUT" indicando que a mensagem de resposta não foi recebida após decorrido um intervalo de tempo previsto para que isso ocorresse. As mensagens de resposta que eventualmente cheguem após a sinalização de um "timeout" serão automaticamente descartadas pelo sistema operacional.

O intervalo de tempo após o qual a comunicação é considerada falha é determinado pelo escalonador visando o cumprimento dos prazos dos módulos emissor e receptor. Módulos sem restrições de tempo (módulos "unbounded"), podem especificar na cláusula FAIL o tempo máximo que desejam esperar por uma mensagem de resposta. Se no exemplo acima fosse usada a seguinte cláusula FAIL,

```
...
FAIL 10ms
=>      c1f;
        ...
        cnf;
...
```

o módulo deveria esperar pela mensagem de resposta durante 10 milisegundos, após o que os comandos c1f a cnf seriam executados. Um módulo de duração limitada ("bounded") pode também especificar na sua cláusula FAIL o tempo máximo de espera pela resposta. Esse tempo, no entanto, irá vigorar apenas quando o módulo for instanciado sem restrições temporais. Nos demais casos, prevalecerá o "timeout" determinado pelo escalonador.

Uma comunicação síncrona falhará também (i.e., a sua cláusula FAIL será executada) quando o módulo destinatário usar o procedimento ABORT, em lugar do comando de resposta REPLY, visando sinalizar uma situação anormal. O comando ABORT pode retornar um código indicativo da razão do término abrupto da comunicação síncrona. Código que poderá ser obtido pelo módulo iniciador da comunicação usando a função REASON.

3.2.10 O comando RECEIVE

O comando RECEIVE é complementar ao comando SEND. Ele endereça portas de entrada (unidirecionais ou bidirecionais) a fim de receber (ler) mensagens de habilitação ou mensagens assíncronas. A recepção de uma mensagem em uma comunicação unidirecional é ilustrada pelo seguinte exemplo:

```
...
ENTRYPORT  pea1:CHAR;
...
MESSAGE MCar:CHAR;
...
BEGIN
```

```
    ...
    RECEIVE MCar FROM psa1
    => cs1;
       ...
       csm;
    FAIL
    => cf1;
       ...
       cfn;
    END;
    ...
END.
```

Neste exemplo, se MCar for recebida, serão executados os comandos cs1 a csm e, em caso contrário, os comandos cf1 a cfn. A cláusula FAIL do comando RECEIVE tem semântica semelhante à cláusula FAIL do comando SEND. No comando RECEIVE, porém, a cláusula FAIL não terá como possível razão de execução um ABORT remoto. Além disso, o "timeout" associado à cláusula FAIL de módulos de tempo-real será sempre zero. Ou seja, se no momento da execução do comando RECEIVE, não houver uma mensagem disponível na porta de entrada endereçada, a cláusula FAIL será imediatamente executada. Da mesma forma que na cláusula FAIL do comando SEND, um "timeout" eventualmente especificado somente será considerado se o módulo for instanciado sem restrições de tempo.

Quando um comando RECEIVE estiver sendo usado para receber as mensagens de habilitação de um módulo, somente tem sentido o uso de uma cláusula FAIL se a condição de habilitação do módulo for do tipo ANY, pois com condições do tipo ALL todas as portas de habilitação devem ter pelo menos uma mensagem disponível para que o módulo seja executado.

Por exemplo, no corpo do módulo CalcResC, mostrado a seguir:

```
MODULE CalcResC(Ref:INTEGER);
    ENTRYPORT
        PResA,PResB:INTEGER;
    ENABLED_BY ALL PResA,PResB;
    MESSAGE MResA,MResB,MResC:INTEGER;

    HANDLER ENABLING_FAULT;
    BEGIN
        RECEIVE MResA FROM PResA
            FAIL => ;
        END;
        RECEIVE MResB FROM PresB
            FAIL => ;
        END;
        ...
```

```
        END;

    BEGIN
        RECEIVE MResA FROM PResA;
        RECEIVE MResB FROM PResB;
        ...
    END.
```

é feita a recepção das mensagens de habilitação usando um RECEIVE unidirecional sem a cláusula FAIL. Esta não é necessária, pois, no instante determinado pelo escalonador "off-line" para início do módulo, as suas mensagens de habilitação deverão estar disponíveis nas portas PResA e PResB (caso não estejam, será gerada uma exceção do tipo "enabling-fault"). O tratador da exceção gerada pela falta das mensagens de habilitação, por sua parte, faz uso da cláusula FAIL. Assim procedendo, caso uma das duas mensagens de habilitação tiver chegado, o módulo poderá recebe-la.

Outro exemplo do uso do comando RECEIVE na recepção de mensagens de habilitação é mostrado a seguir:

```
MODULE mc;
    USE TipMem: UDefA;
    ENTRYPORT
        pe1,pe2:INTEGER;
        pe3:TipMen;
    ENABLED_BY ANY pe1,pe2,pe3;
    MESSAGE
        m1,m2: INTEGER;
        m3: TipMen;
    ...
BEGIN
    RECEIVE m1 FROM pe1
        FAIL => ...;
    END;
    RECEIVE m2 FROM pe2
        FAIL => ...;
    END;
    RECEIVE m3 FROM pe3
        FAIL => ...;
    END;
    ...
END.
```

No exemplo acima, o módulo mc tem uma condição de habilitação não determinística, por isso, pode começar a executar após a chegada de uma mensagem a qualquer das suas portas de habilitação. O uso da cláusula FAIL, conforme mostrado no exemplo, permite ao módulo a leitura de uma mensagem de cada uma das portas que efetivamente recebeu uma mensagem.

As portas de entrada bidirecionais são usadas em transações síncronas. Elas podem ser portas de habilitação de servidores, ou portas de entrada genéricas de módulos "unbounded".

Quando uma mensagem é recebida através de uma porta de entrada bidirecional, uma resposta deve necessariamente ser enviada a essa porta, antes que ela seja novamente endereçada para recepção. Caso, no instante de executar um RECEIVE, o conteúdo da mensagem de resposta for conhecido, pode-se efetuar uma resposta imediata usando uma cláusula REPLY conforme ilustrado a seguir:

```
SERVER ServArq;
    USE TReg: UDFx;
    ENTRYPORT GravaReg:TReg REPLY MSignal;
    ENABLED_BY GravaReg;
    MESSAGE
        Reg:TReg;
        MSignal:SIGNAL;
BEGIN
    RECEIVE Reg FROM GravaReg REPLY MSignal;
    ...
END.
```

Neste exemplo, tão logo a mensagem de habilitação Reg for recebida através da porta GravaReg, uma mensagem do tipo SIGNAL será imediatamente enviada em resposta através dessa mesma porta.

3.2.11 O comando REPLY

Se após a recepção de uma mensagem, através de uma porta de entrada bidirecional, não for desejado fazer uma resposta imediata (usando a cláusula REPLY do comando RECEIVE), pode-se, posteriormente, usar um comando REPLY para enviar a resposta, conforme ilustra o exemplo a seguir:

```
SERVER ServArq;
    USE TReg: ...;
    ENTRYPORT PLeReg:INTEGER REPLY TReg;
    ENABLED_BY PLeReg;
    MESSAGE
        MNumReg:INTEGER;
        MReg:TReg;
    PROCEDURE LêRegistro(NumReg:INTEGER; VAR Reg:TReg);
        ...
BEGIN
    RECEIVE MNumReg FROM PLeReg;
    LêRegistro(MNumReg,MReg);
    REPLY MReg TO PLeReg;
END.
```

Neste exemplo, a mensagem de resposta será construída a partir da mensagem de requisição, devendo ser encaminhada por um comando REPLY endereçado à porta por onde foi recebida a mensagem que lhe deu origem. O endereçamento da porta à qual é encaminhada a resposta deve ser explícito, pois o módulo pode ter recebido diversas mensagens através de outras portas bidirecionais, sem que as devidas respostas tenham sido encaminhadas.

3.2.12 O comando SELECT

O comando SELECT permite efetuar uma recepção seletiva. Numa recepção seletiva são especificadas diversas portas das quais se deseja receber uma mensagem. Caso mais de uma porta tiver mensagens disponíveis, no momento em que o comando for executado, uma das portas será escolhida para que seja lida uma mensagem.

O comando SELECT pode ser usado em diversas circustâncias. O exemplo a seguir ilustra o seu uso num servidor para receber mensagens de requisição:

```
SERVER ServArq;
    USE TReg: ...;
    ENTRYPORT
        GravaReg: TReg  REPLY MSignal;
        LeReg: INTEGER  REPLY TReg;
    ENABLED BY ANY GravaReg, LeReg;
    MESSAGE
        Reg:TReg; NumReg:INTEGER;
        MSignal:SIGNAL;
BEGIN
    RSELECT
        RECEIVE Reg FROM GravaReg REPLY MSignal;
        => cg1;
           ...
           cgm;
    OR
        RECEIVE NumReg FROM LeReg
        => cl1;
           ...
           cln;
           REPLY Reg TO LeReg;
    END;
END.
```

No exemplo, o comando RSELECT é usado para selecionar uma mensagem de requisição de serviço. RSELECT escolhe apenas uma das portas especificadas (GravaReg ou LeReg) e executa os comandos associados. Se a porta GravaReg for escolhida, a mensagem Reg será lida, a mensagem MSIGNAL enviada em resposta e os comandos cg1 a cgm executados. Se a porta LeReg for escolhida, a mensagem NumReg será lida, os comandos cl1 a cln executados e a mensagem de resposta Reg enviada.

Caso mais de uma porta tiver recebido alguma mensagem no instante de execução de uma recepção seletiva, a escolha da cláusula de recepção a ser executada será feita aleatoriamente, se for usado um RSELECT, ou varrendo as portas na ordem em que foram especificadas, caso seja usado um PSELECT. A política de escolha do comando de recepção seletiva é dita fraca [Burns and Wellings. (1990)], pois, em módulos com restrições temporais, pode ser mascarada pelas decisões do escalonador. Isto é, a escolha será feita de forma aleatória (RSELECT) ou priorizada (PSELECT), desde que o escalonador não tenha decidido em que ordem as requisições devem ser atendidas visando cumprir os requisitos temporais dos módulos clientes.

A interferência do escalonador na escolha da cláusula de seleção a executar não ocorrerá quando o comando de seleção (RSELECT ou PSELECT) for usado em módulos não servidores sem restrições de tempo, ou quando nenhum dos clientes pendentes do servidor tiver restrições temporais. Nesses casos, a escolha será efetivamente aleatória ou priorizada conforme se use RSELECT ou PSELECT, respectivamente.

Além das cláusulas de recepção, um comando de seleção pode também incluir uma cláusula ELSE. Esta cláusula será escolhida se nenhuma das portas referenciadas tiver uma mensagem disponível no momento da seleção. O comando de seleção com uma cláusula ELSE tem a seguinte estrutura:

```
...
RSELECT      (* ou PSELECT *)
          RECEIVE M1 FROM Pe1 => ...
     OR
          RECEIVE M2 FROM Pe2 => ...
     ELSE ce1;
END;
...
```

Neste exemplo, se no momento da execução do RSELECT (ou do PSELECT) não houver uma mensagem disponível na porta Pe1 nem na porta Pe2, o comando ce1 será imediatamente executado. Em vez de uma cláusula ELSE (que equivale a um tempo de espera zero pelas mensagens M1 e M2), o módulo poderia ter determinado um tempo de espera pelas mensagens M1 e M2, usando a cláusula TIMEOUT:

```
MODULE m(TEspReq:INTEGER);
...
     RSELECT
               RECEIVE  M1 FROM Pe1 => ...
          OR
               RECEIVE  M2 FROM Pe2 => ...
          OR
               TIMEOUT  TEspReq ms
               => ce1;
     END;
```

```
...
END.
```

Neste caso, se as portas Pe1 e Pe2 não tiverem uma mensagem disponível, o módulo ficará aguardando durante TEspReq milisegundos pela chegada de uma mensagem a qualquer dessas portas, decorrido esse prazo sem que alguma mensagem tenha chegado, ce1 será executado. A cláusula TIMEOUT, evidentemente, só tem efeito quando empregada em módulos sem restrições de tempo.

Cláusulas de seleção podem ser precedidas por uma expressão booleana denominada guarda. Nesse caso, a cláusula será considerada para seleção unicamente quando, no momento de efetuar uma seleção, a guarda estiver aberta (for avaliada verdadeira). Por exemplo, no código abaixo,

```
...
VAR HáLugar, NãoVazio: BOOLEAN;
...
RSELECT
    WHEN HáLugar
        RECEIVE Item FROM InserirItem  REPLY MSignal;
        =>  ...
            NãoVazio:=TRUE;
OR
    WHEN NãoVazio
        RECEIVE MSignal FROM LerItem
        =>  ...
            Item:=...;
            HáLugar:=TRUE;
            REPLY Item TO LerItem;
END;
...
```

especifica-se que a leitura de uma mensagem recebida pela porta InserirItem somente poderá ser considerada se a variável booleana HáLugar for TRUE. De modo análogo, a recepção por LerItem somente será considerada se a variável NãoVazio for TRUE.

3.2.13 comando LOCK

O comando LOCK permite a especificação de uma operação multiserviço, o seu uso é ilustrado pelo código a seguir:

```
MODULE AtualizaArq;
    USE TReg: ...;
    EXITPORT
        LeReg:  INTEGER  REPLY  TReg;
        GravaReg: TReg  REPLY MSignal;
    MESSAGE
        Reg:TReg; NumReg:INTEGER;
```

```
        MSignal:SIGNAL;
BEGIN
    ...
    NumReg:=...;
    LOCK
        SEND NumReg TO LeReg WAIT Reg;
        Reg:=...;
        SEND Reg TO GravaReg WAIT MSignal;
    END;
    ...
END.
```

No exemplo acima, define-se um acesso multiserviço que envolve as portas LeReg e GravaReg do módulo AtualizaArq. Durante a configuração de uma aplicação, essas portas devem ser conectadas a um mesmo servidor. Se esse servidor, em tempo de execução da aplicação, aceitar uma mensagem de requisição NumReg de uma instânca do módulo AtualizaArq, ele não poderá aceitar mensagens de requisição de qualquer outro módulo até ter recebido a mensagem Reg, enviada pela instância do módulo AtualizaArq que iniciou a operação multiservico, e ter enviado a devida mensagem de resposta.

O exemplo acima ilustra um acesso multiserviço composto de dois serviços. Entretanto, acessos multiserviços podem envolver um número arbitrário de serviços. Independentemente do número de serviços envolvido, após o recebimento da primeira requisição de um dado acesso multiserviço, o servidor ficará alocado a esse acesso até que a última mensagem do acesso seja recebida e processada.

3.2.14 Os Comandos iterativos limitados LOOP, WHILE, REPEAT e FOR

A fim de que o tempo máximo de execução de um módulo possa ser determinado, é necessário conhecer, em tempo de compilação, o número máximo de vezes que o corpo de um comando iterativo será executado. Com exceção do comando LOOP, não oferecido pelo Pascal, todos os demais comandos iterativos disponíveis na LPM-RC têm estrutura análoga aos comandos equivalentes em Pascal, com a diferença de incluirem uma cláusula AFTER. A cláusula AFTER especifica um limite para o número de iterações que um comando pode realizar e, opcionalmente, uma sequência de comandos a executar quando for feita uma tentativa de ultrapassar o limite especificado.

O comando LOOP limitado

O comando LOOP limitado define um laço de execução que será interrompido quando for executado um comando EXIT ou quando for realizado o múmero máximo de iterações especificado na sua cláusula AFTER. Por exemplo, o comando LOOP abaixo,

```
...
LOOP
    c1;
```

```
        IF <expressão> THEN EXIT;
        c2;
END   AFTER 10 TIMES
        =>    c3;
              c4;
        END;
c5;
...
```

delimita um conjunto de comandos (c1, IF... e c2) a ser executado no máximo 10 vezes. Se o comando EXIT for executado na décima iteração, ou antes, o LOOP será encerrado e a execução do módulo retomada no comando que sucede ao LOOP, no exemplo, o comando c5. Se após dez iterações o comando EXIT não for executado, os comandos c3 e c4 serão executados, o LOOP encerrado e a execução do módulo retomada a partir do comando c5.

O comando WHILE limitado

A estrutura do comando WHILE limitado é ilustrada pelo segmento de código abaixo:

```
MODULE m(NumIter:INTEGER);
...
WHILE (<expressão>) DO
        c1;
AFTER NumIter TIMES
        =>   c2;
        END;
c3;
...
```

Neste exemplo, especifica-se que o comando c1 deverá ser executado no máximo o número de vezes correspondente à constante NumIter. Após decorridas NumIter iterações sem que <expressão> tenha sido avaliada FALSE, o comando c2 será executado e o WHILE encerrado.

O comando REPEAT limitado

O comando REPEAT limitado tem a estrutura ilustrada pelo seguinte exemplo:

```
...
REPEAT
        c1;
UNTIL <expressão> AFTER 10 TIMES
        =>   c2;
        END;
c3;
...
```

O código acima especifica que o comando c1 deverá ser executado no máximo 10 vezes. Se na décima iteração, ou antes, <expressão> não for avaliada TRUE, o comando c2 será executado e o REPEAT encerrado.

O comando FOR limitado

A estrutura do comando FOR limitado é ilustrada pelo exemplo abaixo:

```
...
FOR i:=1 TO n DO
      c1
AFTER 10 TIMES
      =>    c2;
      END;
c3;
...
```

Neste exemplo, o comando c1 deverá ser executado no máximo 10 vezes. Se na décima iteração, ou antes, a variável de controle i não tiver atingido o valor indicado pelo conteúdo de n, o comando c2 será executado e o FOR encerrado.

3.3 A linguagem de configuração de módulos com restrições críticas de tempo (LCM-RC).

A LCM-RC permite que sejam definidas as entidades grupo, estação e sistema, a partir de um dado conjunto de módulos. A entidade sistema representa um programa de aplicação, constituindo o nível mais alto de um programa de configuração. As entidades grupo e estação correspondem a níveis de abstração intermediários (entre o módulo e o sistema) e, do mesmo modo que módulos, podem ser usadas para construir distintos programas de aplicação (sistemas).

3.3.1 Definição de um grupo

Grupos são constituídos de um conjunto de módulos aos quais poderá ser associada uma única restrição temporal e uma única restrição de alocação. Um grupo tem a seguinte estrutura:

```
[UNBOUNDED] [NONPREEMPTABLE]
    GROUP <identificador> [(<parâmetros formais>)];
        [<definição de contexto>]
        [<declaração de portas de interface>]
        [<declaração de índices de famílias>]
        [<definição dos componentes do grupo>]
        [<associação de portas à interface>]
        [<conexão de portas internas>]
    END.
```

O cabelhaço de um grupo é semelhante em estrutura e semântica ao cabeçalho de um módulo. Quando um grupo for limitado ("bounded"), entende-se que todos os seus componentes também o são. Os parametros formais do grupo, do mesmo modo que os parâmetros de um módulo, declaram constantes internas ao grupo que deverão ser definidas durante a sua instanciação.

A definição do contexto e a declaração das portas de interface de um grupo têm a mesma sintaxe e a mesma semântica das correspondentes seções de um módulo (v. seção 3.2).

Do mesmo modo que portas afins podem ser agrupadas em famílias de portas, módulos afins podem também ser agrupados em famílias de módulos.

A figura 3.1 mostra, a título de exemplo, o diagrama de configuração de um grupo identificado por ga, construído a partir dos módulos ma e mb, cuja interface é mostrada a seguir:

```
MODULE ma;
    ENTRYPORT pe1:SIGNAL;
    EXITPORT  ps1:INTEGER; ps2:SIGNAL;
    ENABLED_BY pe1;
END.
```

```
MODULE mb;
    USE FamX:UDFx;
    ENTRYPORT pe[FamX]:SIGNAL;
    EXITPORT  ps1:SIGNAL;
    ENABLED_BY ALL pe[FamX];
END.
```

Onde, FamX foi definido da seguinte maneira.

```
DEFINE UDFx:FamX, TamFamX, ...;
    CONST
        TamFamX = ...;
    TYPE
        FamX = 1..TamFamX;
    ...
END.
```

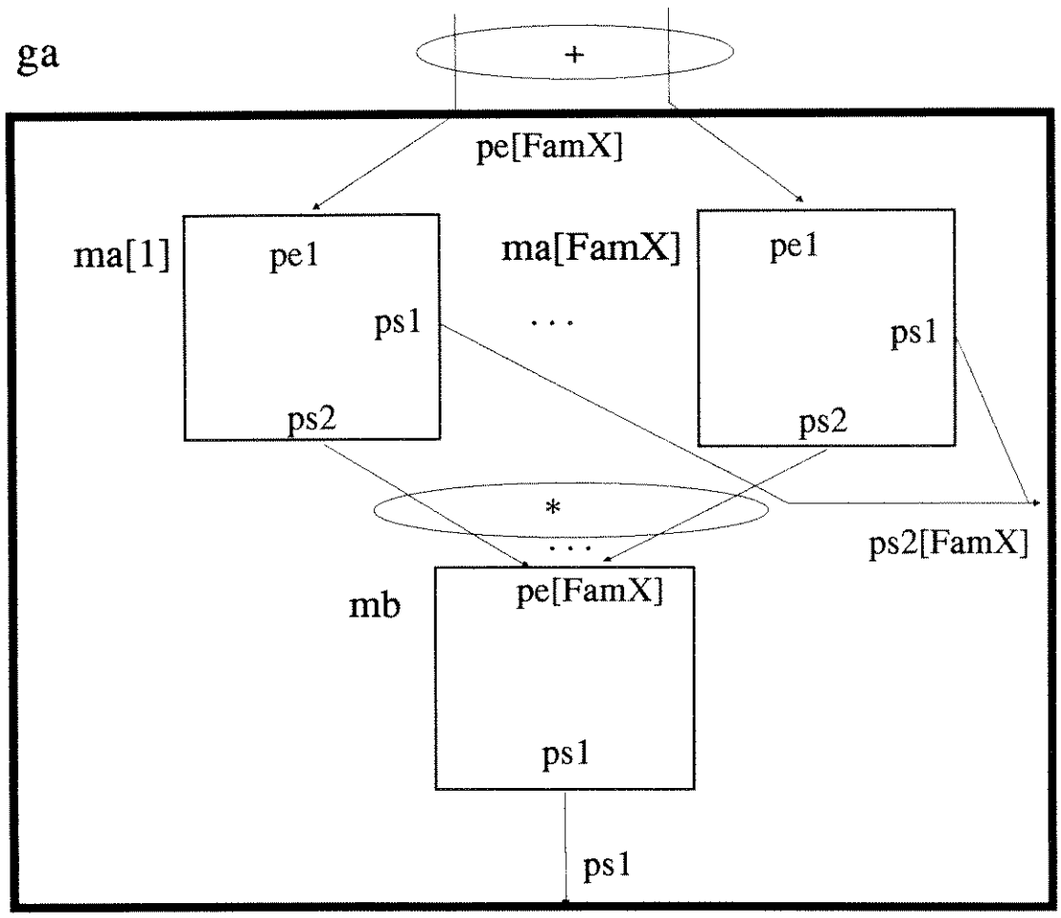


Figura 3.1: Diagrama de configuração do grupo ga

O grupo ga (v. figura 3.1) poderia ser especificado pelo seguinte programa de configuração em LCM-RC:

```
GROUP ga;
    USE FamX,TamFamX:UDFx;
    ENTRYPORT pe[TamFamX] : SIGNAL;
    EXITPORT
        ps1 : SIGNAL;
        ps2[TamFamX] : INTEGER;
    INSTANCE
        fma[TamFamX] : ma;
        mb1 : mb;
    FAMILY fx:FamX;
    ASSOCIATE
        FOR fx:= 1 TO TamFamFx DO
        BEGIN
            ma[fx].pe1  WITH pe[fx];
            ma[fx].ps1  WITH ps2[fx];
            mb.ps1 WITH ps1;
        END;
    LINK
        FOR fx:= 1 TO TamFamFx DO
            ma[fx].ps2 TO mb.pe[fx];
END.
```

A interface do grupo ga é definida pela família de portas de entrada pe, pela porta de saída ps1 e pela família de portas de saída ps2. Isto é, apenas essas portas são visíveis externamente a ga. Na cláusula INSTANCE são declarados os componentes do grupo: a família de módulos fma, do tipo ma, e a instância mb1, do tipo mb.

As portas de interface de um grupo são entidades puramente lógicas, elas devem ser mapeadas em portas dos seus módulos componentes usando uma cláusula ASSOCIATE. Por outro lado, a ligação das portas internas ao grupo, isto é, das portas de módulos que compõem o grupo e não fazem parte da sua interface, é feita na cláusula LINK.

Componentes de grupos são entidades que podem ser executadas concorrentemente. Por esse motivo, um grupo tem como condição implícita de habilitação o conjunto formado pela união das condições de habilitação dos módulos componentes. No exemplo acima, o grupo ga será habilitado pela chegada de uma mensagem a qualquer membro da família pe (pe[1],pe[2],...,pe[TamFamX]), pois cada uma dessas portas habilita um membro da família de módulos ma (ma[1],ma[2],...,ma[TamFamX]).

Grupos podem também ser formados de uma combinação de módulos e de outros grupos. Por exemplo, usando o grupo ga e os módulos mc e md, cuja interface é mostrada abaixo, poderia ser definido o grupo gb ilustrado na figura 3.2.

```
MODULE mc;
        EXITPORT ps:SIGNAL;
END.

MODULE md(TamFamY);
     ENTRYPORT pe[1..TamFamY]:SIGNAL;
     ENABLED_BY ALL pe[1..TamFamY];
END.
```

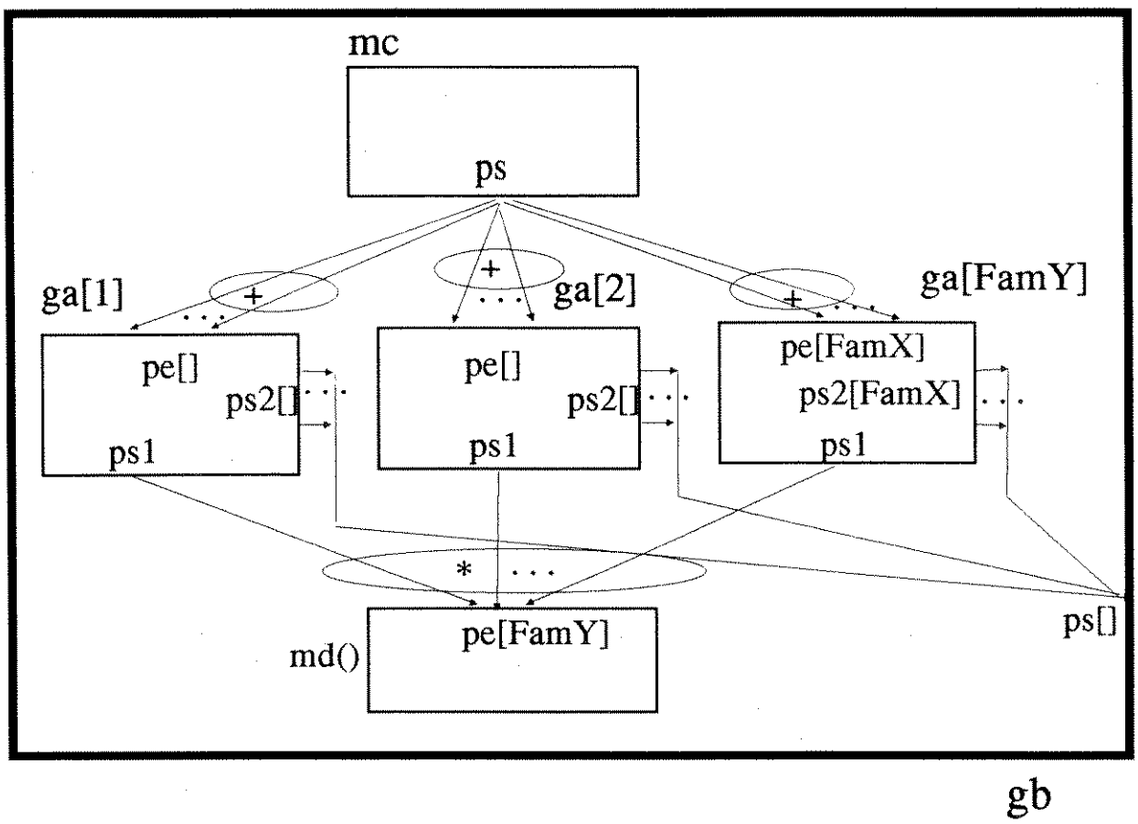


Figura 3.2: Diagrama de configuração do grupo gb

O grupo gb (v. figura 3.2) é especificado pelo seguinte código em LCM-RC:

```
GROUP gb(TamFamY:INTEGER);
     USE TamFamX,FamX : UDFx;
     EXITPORT
          ps[FamX] : INTEGER;
     INSTANCE
          ga[1..TamFamY] : ga;
          mc1 : mc;
```

```
        md1(TamFamY) : md;
    FAMILY
        fx:FamX;
        fy:1..TamFamY;
    LINK
        FOR fy:=1 TO TamFamY DO
            FOR fx:=1 TO TamFamX DO
            BEGIN
                mc1.ps1 TO ga[FY].pe[fx];
                ga[fy].ps2[fx]  TO ps[fx];
                ga[fy].ps1 TO md1.pe[fy];
            END;
END.
```

3.3.2 Definição de uma estação lógica

Conforme pode ser observado a seguir, estações lógicas têm estrutura semelhante a grupos:

```
STATION <identificador> [(<parâmetros formais>)];
    [<definição de contexto>]
    [<declaração de portas de interface>]
    [<declaração de índices de famílias>]
    [<definição de instâncias componentes da estação>]
    [<associação de portas à interface>]
    [<conexão das portas internas>]
    [<definição de restrições temporais>]
END;
```

Estações lógicas também são construídas a partir de módulos e de grupos. A principal diferença entre uma estação e um grupo é que a uma estação não se pode associar uma única restrição temporal. As restrições temporais dos componentes de uma estação devem ser individualmente especificadas na declaração da estação.

3.3.2.1 Especificação das restrições temporais dos componentes de uma estação lógica

A especificação das restrições temporais dos componentes de uma estação (módulos e grupos) é feita utilizando os comandos EVERY, SPORADICALLY e GUARANTEE. Esses comandos podem incluir uma cláusula ELSE para definir a política de tratamento de exceções a adotar quando as restrições temporais especificadas não puderem ser cumpridas.

Uma cláusula ELSE pode usar os comandos HANDLE, SKIP e STOP. O comando HANDLE especifica que, quando uma exceção ocorrer, o respectivo tratador, se definido pelo módulo, deve ser executado. O comando SKIP determina que, quando uma exceção

ocorrer, a presente instância de execução do componente deve ser saltada, isto é, ela deve ser simplesmente ignorada se o componente ainda não começou a rodar, ou prematuramente terminada, caso a execução do componente tenha sido iniciada. Finalmente, o comando STOP determina a suspensão definitiva do componente associado à exceção sinalizada.

Grupos são entidades lógicas. Quando é feita uma referência à execução de um grupo, de fato está sendo feita uma referência à execução dos módulos que compõem o grupo. Logo, quando um grupo está em execução e ocorre uma exceção, o seu tratamento (quando especificado) deve ser feito pelo tratador definido pelo módulo associado à exceção sinalizada. Por outro lado, como um grupo está associado a um único comando de escalonamento, quando se empregam as opções SKIP e STOP, todos os módulos componentes do grupo são afetados: a execução de todos (do grupo) é saltada ou suspensa.

O comando EVERY

O comando EVERY define os períodos de execução dos componentes de uma estação. Por exemplo, os comandos,

```
EVERY 50ms DO ca;
EVERY 100ms DO cb;
```

especificam, respectivamente, que o componente ca deverá ser executado a cada 50 milisegundos e que o componente cb deverá ser executado a cada 100ms. A execução dos componentes ca e cb poderá ser iniciada em qualquer instante após o início de cada período de execução, devendo ser terminada antes do início do próximo período. Isto é, o prazo de execução de cada componente é igual ao seu período. No exemplo acima, a primeira execução (ou instância de execução) do componente ca deverá ser concluída até o instante 50ms, a segunda execução deverá ser concluída até o instante 100ms, e assim por diante.

Módulos periódicos são executados na ordem e nos instantes determinados por um escalonamento gerado "off-line". Apesar disso, uma falha no meio de comunicação ou num processador remoto, pode impedir que as mensagens de habilitação de um módulo cheguem a tempo de iniciar a sua execução no instante determinado pelo escalonador. Quando isso ocorrer, será gerada uma exceção do tipo "enabling_fault". A política a adotar nessas situações pode ser definida numa cláusula ELSE. Por exemplo, no comando:

```
EVERY 50ms DO ca ELSE SKIP;
```

especifica-se que a instância de execução corrente do módulo (ou grupo) ca deve ser saltada (ignorada) se as suas mensagens de habilitação não chegarem antes do instante determinado para seu início.

Deve ser observado que o tipo de exceção à qual a cláusula ELSE faz referência é implícito. No comando acima, a exceção referênciada é do tipo "enabling_fault", por ser essa a única exceção temporal que pode ser gerada para um módulo periódico.

Por "default", a execução de um componente pode ser iniciada a qualquer instante após o início de um período, devendo ser concluída antes do término desse período. Alternativamente, o instante mais cedo de início e o instante mais tarde de término (prazo) de um componente, dentro de cada período de execução, podem ser especificados usando as cláusulas AFTER e WITHIN, do comando EVERY. Por exemplo, o comando,

EVERY 50ms DO ca AFTER 10ms WITHIN 40ms;

especifica que: 1) a cada 50ms inicia-se um período de execução de ca; 2) iniciado um período de execução, deve-se aguardar pelo menos 10ms antes de começar a execução de ca; e 3) a execução de ca deve ser terminada, no mais tardar, 40ms após o início de cada período de execução.

O comando SPORADICALLY

O comando SPORADICALLY, é usado para especificar que um módulo será ativado (chegará) esporadicamente. Por exemplo,

```
SPORADICALLY
    DO ce WITHIN 20ms;
```

especifica que quando ce for ativado (quando chegar) ele terá 20 ms para concluir a sua execução.

Caso se deseje especificar que quando o prazo de ce for perdido o tratador para essa exceção seja executado, deve ser usada a cláusula ELSE:

```
SPORADICALLY
    DO ce WITHIN 20ms
    ELSE HANDLE;
```

O comando GUARANTEE

O comando GUARANTEE pode ser usado para especificar que a execução de um módulo aperiódico deve ser iniciada apenas se o escalonador dinâmico puder garantir que o módulo irá cumprir o seu prazo. Por exemplo, o comando,

```
GUARANTEE
    DO ce WITHIN 20ms;
```

especifica que cada vez que o módulo (ou grupo) ce chegar ao sistema (for habilitado), o escalonador dinâmico deve procurar alocar os recursos necessários para que ce possa ser executado no prazo de 20ms, contado a partir de sua chegada. Se não for possível garantir que ce cumprirá o prazo estipulado, pode-se especificar que o tratador apropriado ("non_guaranteed"), se definido pelo módulo, seja executado em lugar de ce:

```
GUARANTEE
    DO ce WITHIN 20ms
    ELSE HANDLE;
```

Neste caso, porém, o tratador será executado sem o emprego do conceito de garantia, ou seja, ele disputará os recursos do sistema com os demais módulos aperiódicos não garantidos. No processo de escalonamento do tratador, será usado o prazo originalmente definido para o módulo ce. Se a execução do tratador também tiver que ser condicionada ao cumprimento de um prazo, pode-se utilizar a seguinte alternativa:

```
GUARANTEE
        DO ce WITHIN 20ms
    OR
        DO HANDLER WITHIN 10ms
END;
```

Pode-se ainda oferecer ao escalonador dinâmico prazos alternativos para a execução de um módulo, por exemplo:

```
GUARANTEE
        DO ce WITHIN 20ms;
    OR
        DO ce WITHIN 30ms;
    OR
        DO HANDLER WITHIN 5ms
END;
```

Neste exemplo, pede-se ao escalonador dinâmico que tente escalonar ce com prazo de 20ms. Caso isto não seja possível oferece-se como alternativa o prazo 30ms. Se nenhum dos dois prazos puder ser garantido, o escalonador deve tentar assegurar que o tratador da excecão "módulo não garantido" será executado num prazo de 5ms. Se nenhuma das alternativas puder ser satisfeita, nem ce nem seu tratador serão executados na presente ativação de ce. É interessante notar que esta construção, além de suportar a definição de prazos alternativos para o módulo ce, suporta ainda a definição de duas versões de ce: o tratador e o corpo de ce.

3.3.2.2 Exemplo de estação

Neste item será ilustrado o uso da construção STATION através de um exemplo simples. No exemplo serão usados o grupo gb (v. fig. 3.2), os módulos me e mf, e o servidor sa, cuja interface é descrita a seguir:

```
GROUP gb(TamFamY:INTEGER);
    USE TamFamX,FamX : UDFx;
    EXITPORT ps[FamX] : INTEGER;
        FAMILY fp:[1..TamFamPort];
        EXITPORT ps[fp] : INTEGER;
    END;

    MODULE me;
        ENTRYPORT pe: INTEGER;
```

```
    EXITPORT  LeTotal : SIGNAL REPLY INTEGER;
END;

MODULE mf;
    ENTRYPORT evento : SIGNAL;
    EXITPORT  ps    : INTEGER;
    ENABLED_BY evento;
END;

SERVER sa;
    ENTRYPORT
        ResParcial : INTEGER;
        LeTotal    : SIGNAL REPLY INTEGER;
    ENABLED_BY ANY ResParcial, LeTotal;
 END;
```

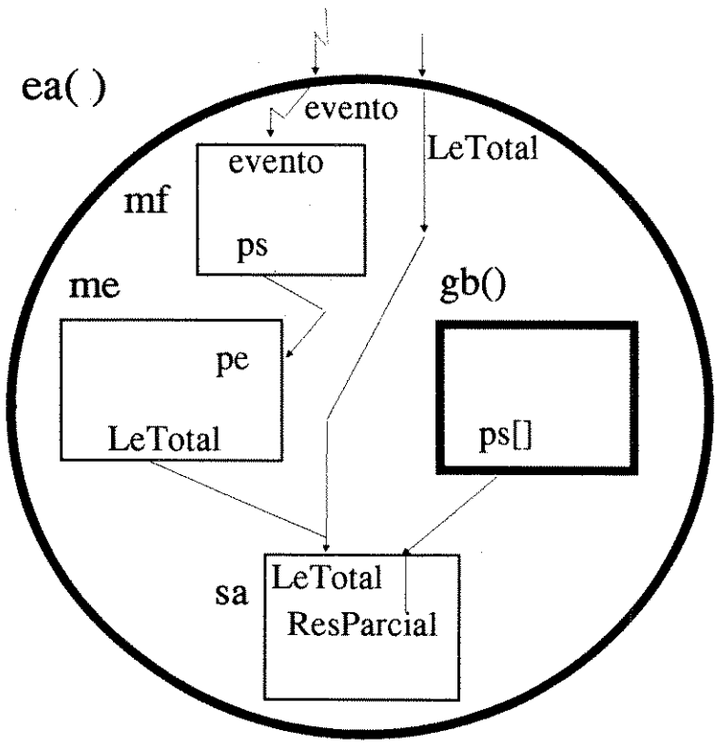


Figura 3.3: Diagrama de configuração da estação ea

Usando os componentes gb,me,mf e sa, o código a seguir especifica a estação ea, cujo diagrama de configuração é mostrado na figura 3.3:

```
STATION ea(TamFamY:INTEGER);
    USE FamX,TamFamX:UDFx;
    ENTRYPORT
        LeTotal : SIGNAL REPLY INTEGER;
        evento : SIGNAL;
    INSTANCE
        gb1(TamFamY) : gb;
        me1 : me;
        mf1 : mf;
        sa1 : sa;
    FAMILY fx:FamX;
    ASSOCIATE
        sa1.LeTotal     WITH  LeTotal;
        mf1.evento      WITH evento;
    LINK
        FOR fx:=1 TO TamFamX DO
            gb1.ps[fx]  TO sa1.ResParcial;
        me1.LeTotal  TO  sa1.LeTotal;
        mf1.ps  TO  me1.pe;
    SCHEDULE
        EVERY 30ms gb1;
        EVERY 50ms me1;
        SPORADICALLY
            DO mf1  WITHIN  5ms;
END.
```

A estação ea é composta por instâncias do grupo ga, dos módulos me e mf, e do servidor sa. Externamente, apenas a porta LeTotal, do servidor sa1 e a porta evento, do módulo mf, são visíveis. Para o grupo gb1 e para o módulo me1 foram especificados períodos de 30 e 50 milisegundos, respectivamente. Tanto gb1 como me1 podem utilizar-se dos serviços do servidor sa1. Este, não tem uma restrição temporal própria, ele deve ser escalonado como parte do esforço necessário ao cumprimento dos prazos dos seus clientes, gb1 e me1. A chegada de uma mensagem à porta evento da estação habilita o módulo mf1, que deve ser executado no prazo de 5ms. O módulo mf1 pode comunicar-se com o módulo me1 enviando mensagens assíncronas através do canal definido pelas portas mf1.ps e me1.pe.

3.3.3 Definição de um sistema

Sistemas são compostos de módulos, grupos e estações. A definição da entidade sistema corresponde à etapa final da configuração de um programa distribuído. Nesta etapa, além da especificação das restrições temporais dos módulos e dos grupos não incluídos em estações, podem ser especificadas restrições de alocação para os componentes do sistema (módulos, grupos e estações). A definição de um sistema tem a seguinte estrutura:

```
SYSTEM <identificador>;
    [<definição de contexto>]
    [<definição da arquitetura física do sistema distribuído>]
    [<declaração de índices de famílias>]
    [<definição de instâncias componentes do sistema>]
    [<definição de restrições de alocação de componentes>]
    [<conexão de portas de componentes do sistema>]
    [<definição de restrições temporais>]
END.
```

3.3.3.1 Definição da arquitetura física do sistema distribuído

A definição da arquitetura física do sistema distribuído consiste no mapeamenteo dos nomes simbólicos dos nós, ou estações físicas, que compõem o sistema distribuído para seus endereços físicos. O mapeamento é dependente da arquitetura do sistema empregado, pressupondo-se que é possível efetuar a troca de mensagens entre qualquer par de nós do sistema distribuído. Em adição, para fins de escalonamento, deve ser possível determinar o tempo máximo necessário para transferir uma mensagem (de tamanho conhecido) entre um par arbitrário de nós.

A definição da arquitetura física do sistema distribuído é feita numa cláusula NETWORK, conforme ilustrado a seguir:

```
NETWORK (* define nós da rede e seus endefeços físicos *)
    NoA = ... ;
    NoB= ... ;
    NoC = ... ;
```

3.3.3.2 Definição de restrições de alocação para componentes

As restrições de alocação de componentes podem ser definidas especificando diretamente o endereço do nó onde um componente deve ser executado, ou sob a forma de restrições de exclusão ou inclusão. Uma restrição de exclusão é usada para especificar que um dado conjunto de componentes deve ser alocado a estações diferentes, não importando quais sejam elas. Este tipo de restrição pode ser usada, por exemplo, como parte de uma estratégia de tolerância a falhas, assentada na criação de instâncias redundantes de um mesmo tipo de módulo. Uma restrição de inclusão especifica que um dado conjunto de componentes do sistema deve ser alocado a um mesmo nó do sistema distribuído, sem precisar a qual nó.

A definição das restrições de alocação dos componentes de um sistema pode ser ilustrada pelo seguinte exemplo:

```
ALLOCATE
    ea1, mf1 AT NóA;
    ea1,me1,me2  AT DIFFERENT;
```

O código acima especifica que os componentes ea1 e mf1 devem ser alocados ao nó identificado por NóA e que os componentes ea1, me1 e me3 podem ser alocados a qualquer nó do sistema, desde que cada um seja alocado a um nó diferente dos demais. Se considerarmos que a especificação acima diz respeito ao sistema cuja arquitetura física foi especificada em 3.3.3.1, os componentes ea1 e mf1 deverão ser alocados ao NoA e os componentes me1 e me2 aos nós NoB e NoC, respectivamente, ou vice-versa.

3.3.3.3 Exemplo de sistema

A figura 3.4 mostra o diagrama de configuração do sistema Sa, construído usando os módulos me e mf e a estaçao lógica ea, apresentados em 3.3.2.2.

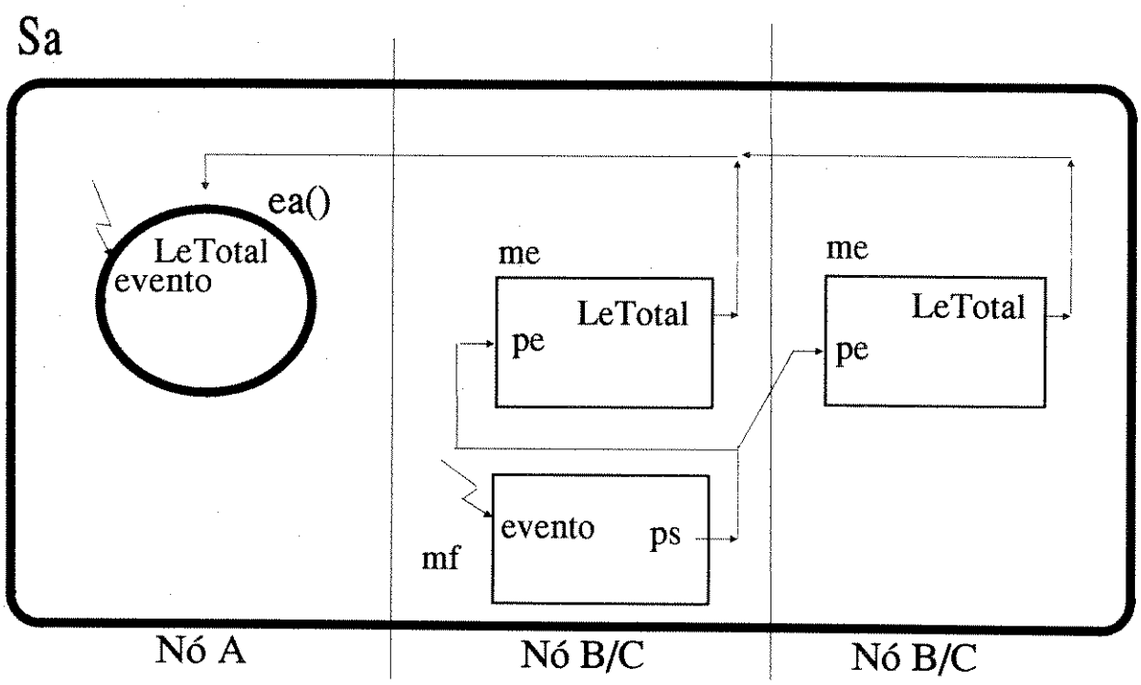


Figura 3.4: Diagrama de configuração do sistema Sa

Para a especificação de Sa será também utilizada a unidade de definição DefSa, mostrada a seguir:

```
DEFINE DefSa;
    CONST
        EndNóA= ...;
        EndNóB= ...;
```

```
        EndNóC = ...;
        IntEventoNóA=...;
        IntEventoNóB=...;
        TamFamY=...;
END.
```

A especificação do sistema Sa em LCM-RC é feita pelo seguinte programa:

```
SYSTEM Sa;
    USE
        IntEventoNóA,IntEventoNóB, EndNóA, EndNóB,
        EndNóC,TamFamY : DefSa;
    NETWORK (* define nós da rede e seus endereços físicos *)
        NoA = EndNóA ;
        NoB = EndNóB ;
        NoC = EndNóC ;
    INSTANCE
        me1,me2 : me;
        mf1 : mf;
        ea1(TamFamY) : ea;
    ALLOCATE
        ea1 AT NoA;
        mf1 AT NóB;
        ea1,me1,me2 AT DIFFERENT;
    LINK
        INTERRUPT.IntEventoNóA  TO ea1.evento;
        INTERRUPT.IntEventoNóB  TO mf1.evento;
        me1.LeTotal TO ea1.LeTotal;
        me2.LeTotal TO ea1.LeTotal;
        mf1.ps TO me1.pe;
        mf1.ps TO me2.pe;
    SCHEDULE
        EVERY 50ms me1;
        EVERY 30ms me2;
        SPORADICALLY
            DO mf1  WITHIN  10ms;
END.
```

Na configuração do sistema Sa, as portas evento, da estação lógica ea1 e do módulo mf1, foram associadas às interrupções IntEventoNóA e IntEventoNóB, respectivamente. Os detalhes envolvidos na implementação dessa associação são dependentes da arquitetura do "hardware" empregado, devendo ser considerados, caso a caso, durante o transporte das linguagens e do suporte à sua execução para cada arquitetura.

3.4 Sumário e comentários finais

O modelo de programação e as linguagens propostas neste capítulo têm como preocupação central possibilitar que aplicações sejam construídas a partir de módulos de "software" reusáveis. A construção de aplicações a partir de módulos reusáveis já foi explorada em trabalhos anteriores [Adán-Coello, Lopes e Magalhães (1987)] que levaram ao desenvolvimento do protótipo de um ambiente de desenvolvimento de sistemas distribuídos concorrentes denominado STER. O uso experimental do STER mostrou a validade desse enfoque, validade reforçada pelo crescente interesse despertado por abordagens orientadas a objetos, com as quais compartilha inúmeros conceitos.

Neste trabalho, a reusabilidade de módulos além de funcional, como no STER, é também temporal. Isso é conseguido desacoplando a implementação da funcionalidade associada a um módulo, feita durante a sua programação, da especificação do seu comportamento temporal -próprio do contexto em que é usado-, feita durante a configuração de uma aplicação.

Para a programação de módulos foi proposta a linguagem LPM-RC (Linguagem de Programação de Módulos com Restrições Críticas de tempo) baseada na linguagem LPM do STER [Lopes (1986)]. A LPM-RC procura conservar os elementos da LPM que propiciam a reusabilidade de módulos e acrescentar os elementos necessários à produção de aplicações com comportamento temporal previsível. Ambas procuram facilitar a reusabilidade de módulos suportando a sua interação através da troca de mensagens endereçadas a portas de comunicação locais aos módulos.

Na LPM-RC, apenas os módulos sem restrições temporais (módulos NRT) podem usar os mecanismos e comandos que podem requerer tempos arbitráriamente longos de execução: recursividade, alocação dinâmica de memória e iterações não limitadas. Além disso, a LPM-RC permite a definição de módulos especiais, denominados servidores, para controlar o acesso a recursos compartilhados por diversos módulos, denominados módulos clientes. A LPM-RC permite ainda que módulos clientes especifiquem operações atômicas multiserviço, viabilizando que algoritmos de escalonamento possam garantir o cumprimento das restrições temporais dos clientes e assegurar a manutenção da consistência dos recursos compartilhados por intermédio dos servidores.

A LPM-RC também permite a definição de tratadores para exceções temporais, sinalizadas quando for constatado que as restrições temporais de um módulo foram violadas, ou há a possibiliade de que sejam violadas. A efetiva execução de um tratador, quando uma exceção for sinalizada, está condicionada à política de tratamento de exceções da aplicação onde o módulo for usado, definida durante a configuração da aplicação.

Para a configuração de uma aplicação foi proposta a LCM-RC (Linguagem de Configuração de Módulos com Restrições Críticas de tempo), a partir da LCM [Lopes (1986); Guimarães, Lopes, Adán e Magalhães (1989)] do STER.

Um programa de configuração tem como propósito definir quais módulos compõem uma aplicação, como esses módulos se inter-relacionam, quais são suas restrições temporais e a que estações do sistema devem ser alocados. A fim de simplificar a estrutura de aplicações de maior porte, uma aplicação pode ser construída em quatro níveis de abstração, representados pelas entidades módulo, grupo, estação e sistema.

4. ESTRATÉGIA DE ESCALONAMENTO DE PROGRAMAS HSTER

A implementação das linguagens propostas no capítulo 3 envolve três etapas principais: 1) o desenvolvimento de compiladores (num sentido amplo) para a LPM-RC e para a LCM-RC; 2) o desenvolvimento de um escalonador que garanta "off-line" que as restrições de tempo dos módulos periódicos serão cumpridas, e 3) a implementação do suporte à execução de programas.

Os compiladores para a LPM-RC e para a LCM-RC têm como objetivo básico a geração de código executável, da mesma forma que quaisquer outros compiladores. No entanto, no código gerado pela LPM-RC e pela LCM-RC devem ser distinguidos três tipos de entidades lógicas: módulos com restrições críticas de tempo (módulos periódicos, ou HRT), módulos com restrições não críticas tempo (módulos aperiódicos, ou SRT) e módulos sem restrições de tempo (módulos NRT). Estas entidades, por sua vez, podem estar subdivididas em entidades de menor granularidade, os blocos (ou segmentos) de escalonamento, que constituem as menores porções de código sequencial escalonáveis para execução. Portanto, além do código executável propriamente dito, o código objeto gerado pelos compiladores deve prover as informações necessárias ao escalonamento do programa visando cumprir os seus requisitos de tempo. Isto é, deve identificar os blocos de escalonamento, especificando seus respectivos tempos de execução, restrições temporais e relações de precedência com outros blocos.

O escalonador "off-line", resultado da segunda etapa, deve, a partir das informações que os compiladores extrairam de um programa, procurar uma alocação e escalonamento para os blocos de escalonamento dos módulos periódicos que garanta que, em tempo de execução, as suas restrições de tempo sempre serão atendidas. Além disso, o escalonador "off-line" deve procurar gerar escalonamentos que ofereçam ao sistema operacional flexibilidade para tratar as chegadas dinâmicas de módulos aperiódicos.

A terceira etapa, o suporte à execução de programas, compreende atividades que envolvem tanto o sistema operacional como os protocolos necessários à comunicação de módulos remotos (localizados em distintos nós, ou estações, de um sistema distribuído). As principais funções do sistema operacional, sob a perspectiva do cumprimento das restrições temporais de um programa, são executar os módulos com restrições de tempo, de acordo com o escalonamento gerado "off-line", e escalonar dinâmicamente os módulos aperiódicos e os módulos sem restrições temporais. Os protocolos de comunicação devem assegurar que mensagens serão entregues dentro dos prazos estipulados, isto é eles devem oferecer tempos de comunicação determinísticos.

O objetivo deste capítulo é discutir a estratégia de escalonamento de programas escritos usando a LPM-RC e a LCM-RC, elemento integrador das três etapas descritas acima, sob a ótica do cumprimento das restrições de tempo de um programa. Para tanto , o capítulo foi dividido em sete seções. Na seção 4.1 são discutidos os principais aspectos associados à extração, em tempo de compilação, das informações necessárias ao escalonamento de programas. Na seção 4.2 é proposto um algoritmo para o escalonamento de módulos periódicos. Na seção 4.3 são propostos alguns mecanismos simples que podem ser incorporados ao algoritmo discutido na seção 4.2, visando gerar escalonamentos "off-line" que ofereçam ao escalonador dinâmico flexibilidade na tarefa de procurar cumprir os requisitos de tempo de módulos aperiódicos. Na seção 4.4 são discutidas algumas questões relacionadas ao escalonamento dinâmico de módulos aperiódicos, de módulos sem restrições de tempo e de tratadores de exceções. Na seção 4.5 são feitas algumas considerações relacionadas a uma implementação do algoritmo de escalonamento "off-line" de módulos aperiódicos, realizada em linguagem C++ sob ambiente UNIX. Finalmente, na seção 4.6, é feito um resumo do capítulo, assim como algumas considerações finais.

Deve-se observar que a unidade lógica de escalonamento no HSTER é o módulo, e a unidade física o segmento (ou bloco) de escalonamento, conforme será discutido nas próximas seções. Entretanto, neste capítulo serão usados frequentemente os termos tarefa e subtarefa, em lugar de módulo e segmento de escalonamento, respectivamente, por serem os dois primeiros termos os normalmente empregados na literatura relacionada ao escalonamento de programas.

4.1 Geração de um Grafo de Escalonamento

A maior parte da informação presente no código de um programa é irrelevante no momento de seu escalonamento. Para fins de escalonamento, um programa resume-se a um conjunto de blocos (ou segmentos) de código seqüencial com restrições de alocação, precedência e tempo. A representação de um programa que contém apenas essas informações será, neste texto, denominada grafo de escalonamento.

Um grafo de escalonamento consiste de um grafo acíclico dirigido, onde os nós representam os segmentos de código seqüencial de um programa distribuído e os arcos as relações de precedência entre os segmentos. Os segmentos de código representados pelos nós do grafo constituem as menores partes de um programa para fins de escalonamento, sendo, por esse motivo, também chamados segmentos de escalonamento. Nós e arcos têm custos associados. O custo de um nó corresponde ao tempo de computação requerido para executar o segmento de escalonamento que ele representa, e o custo de um arco representa o tempo necessário para transferir uma mensagem entre os nós por ele conectados. Ambos os custos dependem do "hardware", "software" e meio de comunicação que serão utilizados para executar a aplicação. Os custos dos arcos e dos nós serão determinados durante a compilação e configuração dos módulos de um programa. Não é propósito deste trabalho discutir esse assunto. Ao leitor interessado no tema, recomenda-se o artigo de Puschner e Koza (1989).

O mapeamento de um programa em grafos de escalonamento é feito em dois passos. O primeiro, durante a compilação de um módulo e o segundo durante a configuração de um programa.

4.1.1 Mapeamento de um Módulo

Para mapear um módulo deve ser considerado, inicialmente, se o módulo está, ou não, envolvido num relacionamento cliente-servidor. Caso esteja, deve ser considerado o papel assumido pelo módulo (cliente ou servidor).

Um módulo que não acessa servidores é mapeado em um único segmento de escalonamento -e, portanto, em um único nó do grafo de escalonamento-, com um arco de entrada para cada porta de habilitação do módulo, conforme é ilustrado na figura 4.1. Nessa figura, os números que aparecem no canto superior direito dos nós indicam o tempo de execução dos módulos correspondentes (arbitrariamente escolhidos a título de exemplo) e os números ao lado dos arcos, o tempo necessário para transmitir uma mensagem através do canal lógico representado pelo arco.

A figura 4.2 ilustra o mapeamento de um módulo que acessa um servidor. Conforme pode ser observado na figura, a requisição de um serviço, feita usando um comando SEND, determina o fim de um bloco de escalonamento, e a espera pela resposta, feita pela cláusula WAIT do comando SEND, o início do bloco seguinte. Para representar um bloco de escalonamento, derivado da requisição de um serviço, é criado um nó no grafo de escalonamento, com um arco de saída representando a mensagem de requisição. Por outro lado, ao nó criado para representar um bloco que se inicia com a espera pela

resposta de um servidor é associado um arco de entrada (ou arco de habilitação), representando a mensagem de resposta. Os arcos assinalados com "lock" e "unlock" indicam, respectivamente, o envio das mensagens de requisição dos serviços que iniciam e concluem uma operação multiserviço.

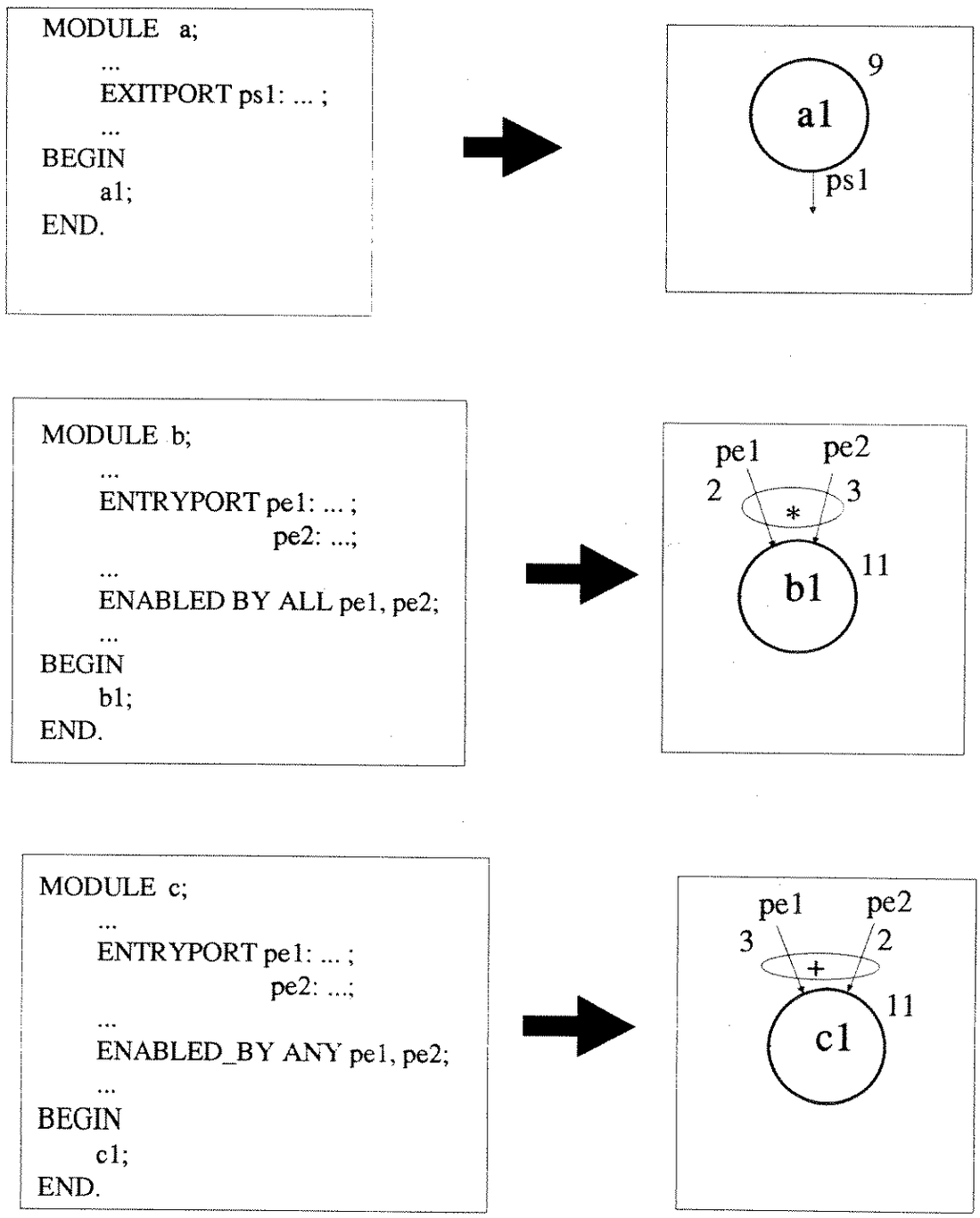


Figura 4.1: Exemplos de mapeamentos de módulos

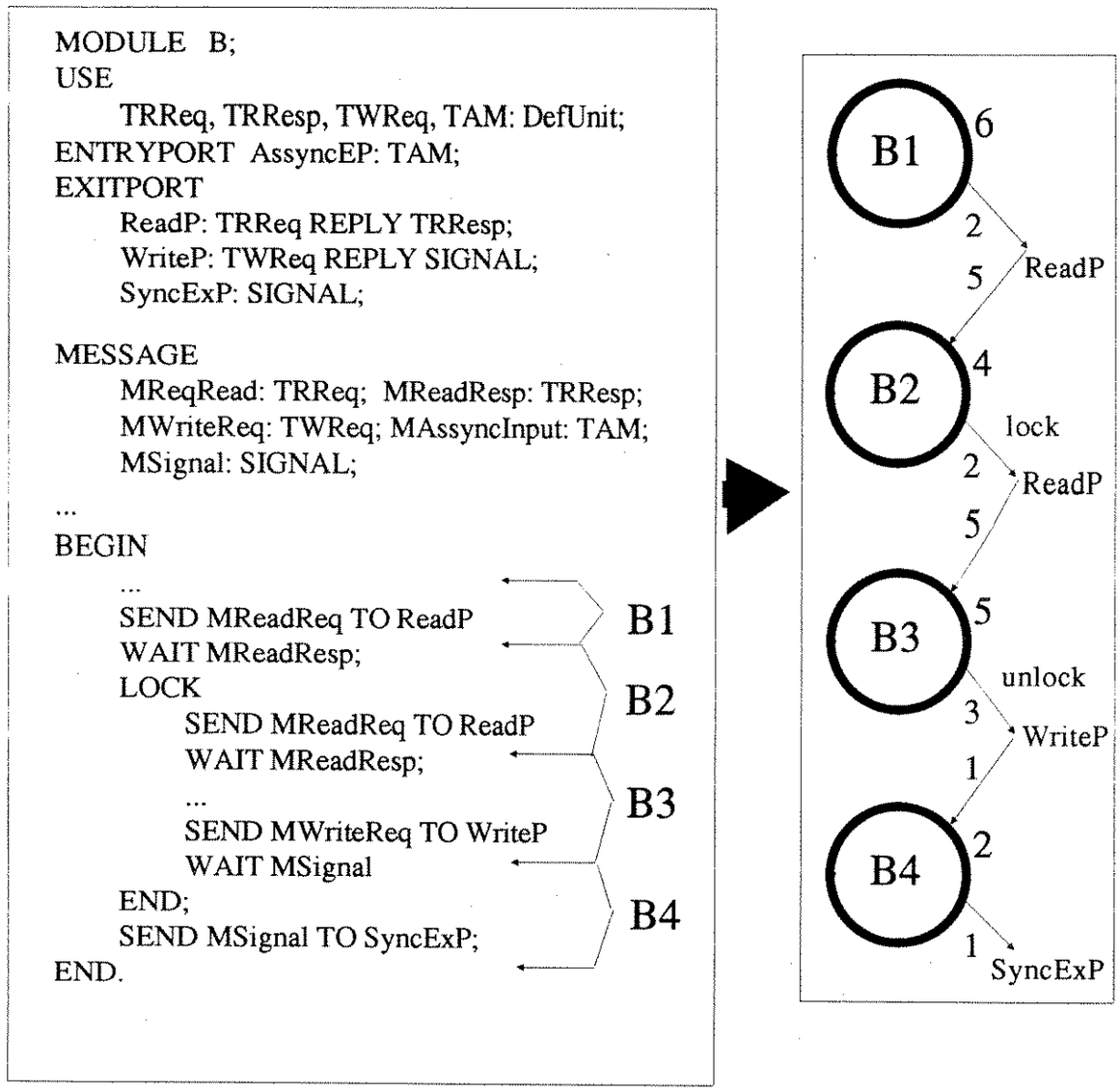


Figura 4.2:Mapeamento de um módulo que acessa um servidor

Serviços podem não gerar respostas, nesse caso, será associado um arco de entrada ao nó que sucede a requisição para representar a mensagem indicativa da conclusão do serviço. Essa mensagem define uma relação implícita de precedência entre servidor e cliente, indicando que embora o cliente não receba explicitamente uma resposta do servidor, a sua execução somente será retomada quando o serviço requisitado tiver sido concluído.

A figura 4.3 ilustra o mapeamento de um servidor. Conforme pode ser observado na figura, o código que implementa cada serviço oferecido é associado a um nó com um arco de entrada e um arco de saída. O arco de entrada representa a mensagem de requisição do

serviço e o arco de saída a mensagem de resposta do serviço, enviada explicitamente pelo servidor ou implicitamente pelo suporte à sua execução.

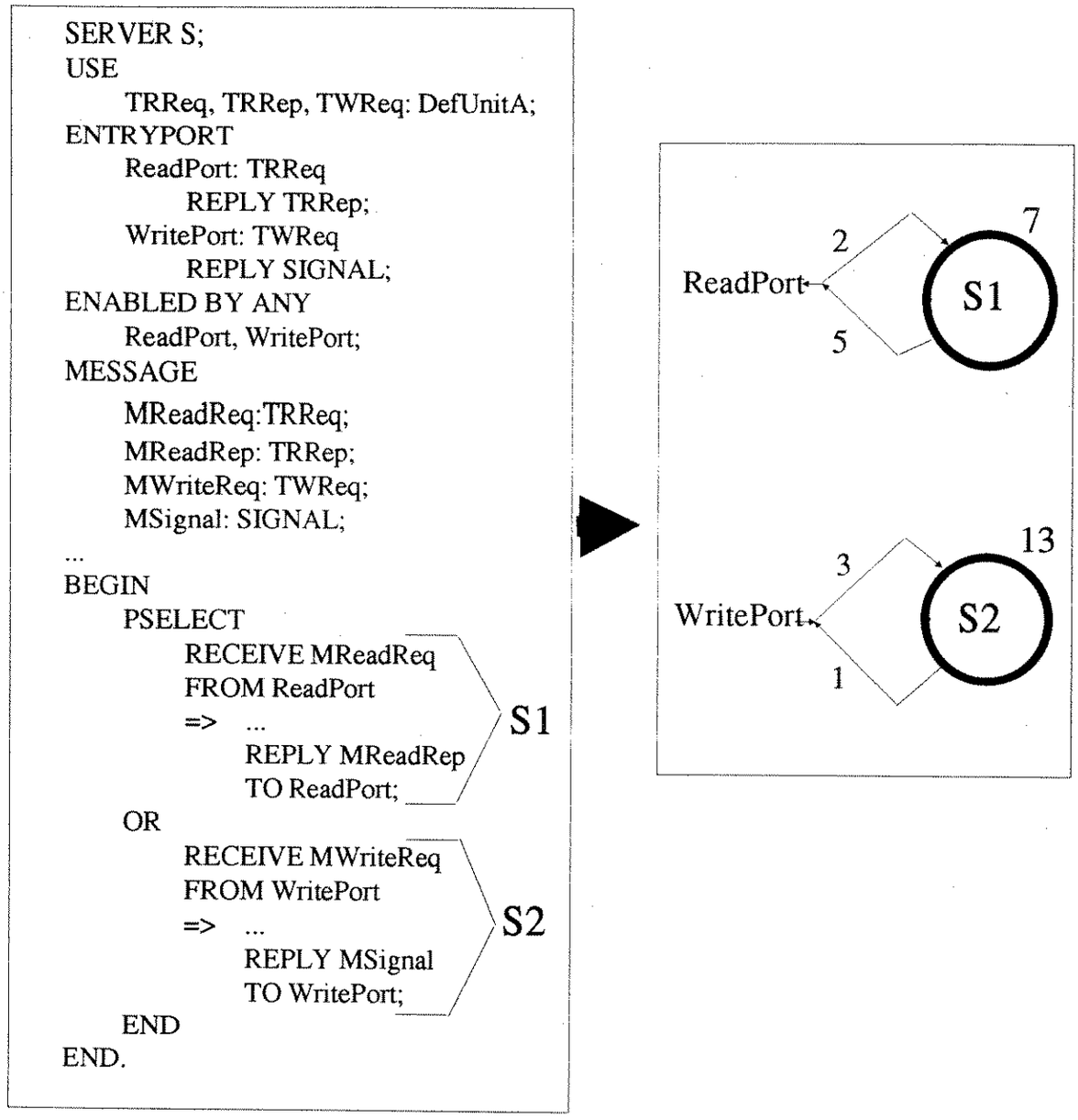


figura 4.3: Mapeamento de um servidor.

4.1.2 Mapeamento de uma Configuração

Durante a configuração de uma aplicação, os grafos parciais dos módulos que a formam são combinados, resultando um grafo que representa todos os módulos periódicos e um grafo para cada módulo aperiódico independente.

No grafo de escalonamento que corresponde aos módulos periódicos são representadas todas as ocorrências desses módulos no intervalo de escalonamento do grafo, definido como o mínimo múltiplo comum dos períodos dos módulos. Gerado o grafo para o intervalo de escalonamento, o problema de garantir o cumprimento das restrições temporais dos módulos periódicos resume-se a encontrar um escalonamento para esse intervalo. Se um escalonamento for encontrado, basta repetí-lo sucessivamente para assegurar que os módulos periódicos cumprirão os seus prazos.

Os grafos de escalonamento gerados para os módulos aperiódicos contêm apenas uma ocorrência de cada módulo, indicando que esses módulos devem ser escalonados dinamicamente a partir do instante de sua chegada.

Após ter sido feita a combinação dos grafos parciais que representam os módulos que formam uma aplicação, são associados aos nós dos grafos resultantes as restrições de localização e tempo especificadas no programa de configuração.

Para ilustrar o mapeamento de uma configuração, consideremos o diagrama de configuração mostrado na figura 4.4.

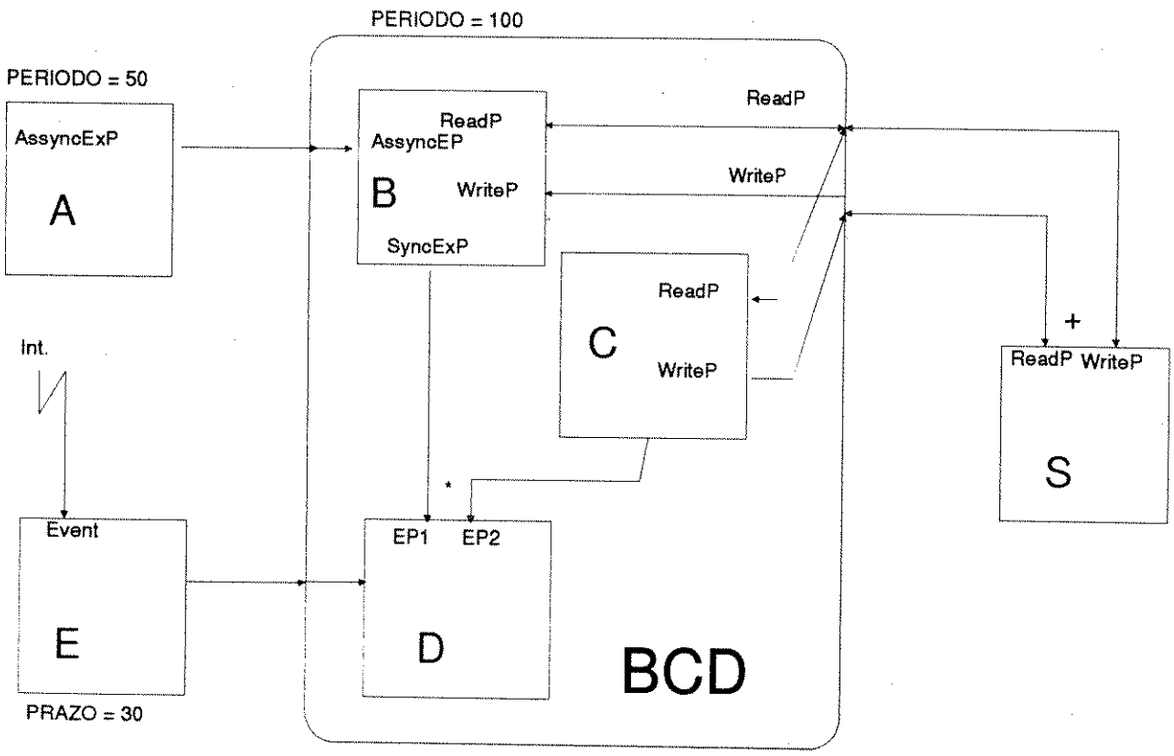


Figura 4.4: Diagrama de configuração de uma aplicação exemplo

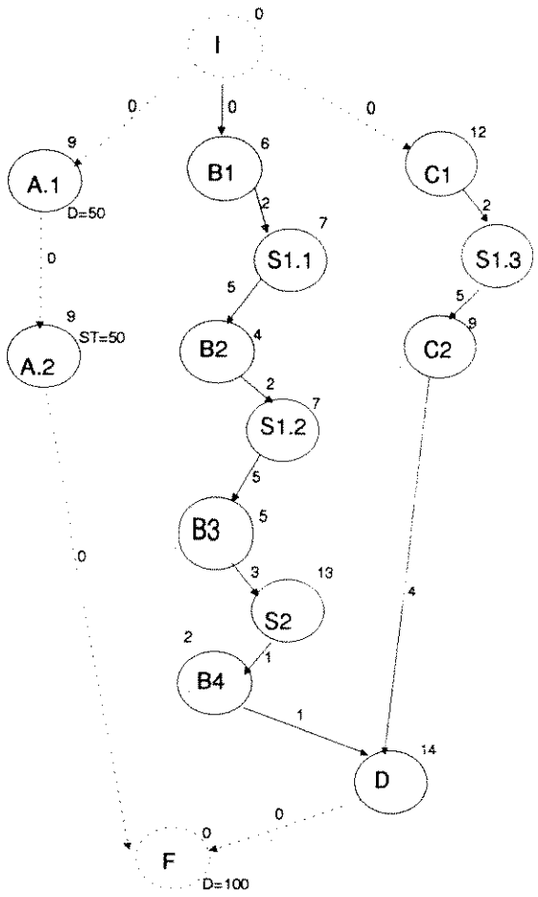
O diagrama apresentado na figura 4.4 descreve uma aplicação composta pelos módulos A e E; pelo grupo BCD e pelo servidor S (cujo código fonte e grafo parcial de escalonamento são mostrados na figura 4.3). O módulo A é independente dos demais (não tem relações de precedência) e periódico, devendo ser habilitado a cada 50 unidades de tempo. O diagrama indica que durante a sua execução ele pode enviar mensagens assíncronas ao grupo BCD. O módulo E é aperiódico, sendo habilitado por uma interrupção associada à sua porta Event. Uma vez habilitado, ele terá um prazo de 30 unidades de tempo para concluir a sua execução, durante a qual poderá enviar mensagens assíncronas ao grupo BCD. Este grupo -formado pelos módulos B (v. figura 4.2), C e D, onde a execução de B e C deve preceder a execução de D-, tem um período de execução de 100 unidades de tempo, e acessa os serviços oferecidos pelo servidor S.

A configuração ilustrada na figura 4.4 deve ser transcrita para um programa LCM-RC. Durante o processamento desse programa, os grafos parciais dos módulos componentes da aplicação descrita serão combinados, gerando-se os dois grafos de escalonamento mostrados na figura 4.5. O primeiro representando as relações de precedência existentes entre os módulos periódicos e o relacionamento destes com o servidor S, e o segundo representando o módulo aperiódico E.

O intervalo de escalonamento do grafo representando os módulos periódicos é de 100 unidades de tempo (MMC de 50 e 100, períodos do módulo A e do grupo BCD, respectivamente). Nesse intervalo, ocorrem duas instâncias de execução do módulo A, uma do módulo B, uma do módulo C e quatro do servidor S (uma para cada serviço requisitado).

Na construção do grafo de escalonamento dos módulos periódicos, foram gerados dois nós terminais: I associado ao início do intervalo de escalonamento e F ao seu fim. Os arcos pontilhados representam relações implícitas de precedência, sem nenhuma relação com a conexão das portas dos módulos. A expressão D=<inteiro_positivo> especifica o prazo ("deadline") de um nó e, consequentemente, dos nós que o precedem. A expressão ST=<inteiro_positivo>, especifica o instante de partida ("start time") de um nó.

O grafo gerado para o módulo aperiódico E contém um único nó, que ao ser habilitado, pela ocorrência da interrupção event, terá um prazo para execução de 30 unidades de tempo.



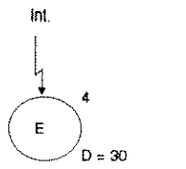


Figura 4.5: Grafos de escalonamento da configuração mostrada na figura 4.4.

4.2 Escalonamento de Módulos Periódicos

A literatura discute inúmeros algoritmos de escalonamento de tarefas periódicas. Não se tem conhecimento, no entanto, de um que se aplique diretamente ao modelo de tarefas resultante do modelo de programação do HSTER. Decidiu-se, por esse motivo, estender um algoritmo desenvolvido por Ramamritham (1990), que suporta parcialmente o modelo de tarefas do HSTER, a fim de que ele seja capaz de tratar os elementos específicos do HSTER.

O algoritmo original de Ramamritham procura estaticamente alocar e escalonar tarefas periódicas em sistemas distribuídos interconectados por protocolos de comunicação determinísticos (que permitem prever os retardos de transmissão). O algoritmo de Ramaritham lida com tarefas periódicas formadas de subtarefas não preemptíveis. Essas subtarefas podem ter restrições de precedência, alocação, recursos e redundância (para implementar mecanismos de tolerância a falhas). As extensões ao algoritmo visam suportar os conceitos de módulo, servidor e acesso multiserviço a um servidor, não considerados por Ramamritham, porém fundamentais para o modelo de programação do HSTER.

Dado, portanto, o grafo de escalonamento que representa um determinado programa, o algoritmo descrito nesta seção tentará alocar os módulos do programa a estações físicas do sistema alvo e escalonar os segmentos dos módulos, respeitando as restrições de localização, precedência e tempo especificadas.

O algoritmo de escalonamento procede à busca de uma alocação e de um escalonamento factíveis, a partir de um grafo de escalonamento, em dois passos: 1) construção de um grafo de comunicação e 2) alocação e escalonamento propriamente ditos dos módulos. O pseudocódigo do algoritmo é o seguinte:

```
PROCEDIMENTO EscalonarMódulosPeriódicos;
INÍCIO
    FC:=0;
    REPITA
        Construir um grafo de comunicação;
        BuscarEscalonamento;
        FC:=FC + IncFC;
    ATÉ (Encontrado um escalonamento) OU (FC > FCMax)
FIM.
```

4.2.1 Construção do Grafo de Comunicação (GC)

O objetivo desta atividade é determinar, para cada par de nós do grafo de escalonamento conectados por um arco, se eles devem ou não ser alocados a uma mesma estação física. Quando dois segmentos que se comunicam são alocados a estações distintas há um aumento potencial de paralelismo na execução dos nós do grafo. Por

outro lado, quando dois segmentos são alocados à mesma estação, a sua conclusão será antecipada, pois não será necessário que a transferência de mensagens entre eles se dê através do meio de comunicação.

Dois nós, i e j, não envolvidos em um relacionamento cliente-servidor (C-S), com custos de execução C_i e C_j unidades de tempo, respectivamente, e tempo de comunicação igual a Com_{ij} unidades de tempo, serão alocados a uma mesma estação se

$$\frac{(C_i+C_j)}{Comm_{ij}} < FC$$

Onde, FC é um parâmetro ajustável denominado fator de comunicação.

Para um determinado valor do FC, este esquema tende a alocar pares de nós com custos de comunicação elevados à mesma estação. Se o algoritmo não conseguir encontrar um escalonamento factível para um dado valor de FC, uma nova tentativa será feita com um novo valor de FC. O maior valor de FC que necessita ser testado é denominado FCmax, onde

$$FC_{max} = \max \frac{C_i+C_j}{Com_{ij}} + 1, \text{ para qualquer i, j.}$$

Como o custo de comunicação entre dois segmentos comunicantes é sempre maior que zero (no mínimo um sinal de habilitação deve ser enviado), um FC igual a FC_{max} irá forçar a que todos os pares de segmentos comunicantes sejam alocados à mesma estação. À medida que FC decresce, mais e mais pares de segmentos comunicantes serão alocados a estações distintas. Quando FC for igual a zero, todos os pares de segmentos comunicantes serão alocados a estações distintas.

Módulos e servidores devem ser tratados como unidades de escalonamento, ou seja, todos os seus segmentos de escalonamento devem ser alocados a uma mesma estação. Logo, no caso de relacionamentos cliente-servidor, é necessário tomar decisões de alocação para pares cliente-servidor, e não individualmente para pares de segmentos de escalonamento (nós no grafo de escalonamento). Isto requer que, para uma dada interação cliente-servidor, seja considerada a relação existente entre a soma total dos custos de computação e a soma dos custos de comunicação dos segmentos de escalonamento das duas entidades envolvidos na interação e não, como na interação entre clientes, simplesmente a relação existente entre pares de segmentos comunicantes.

Portanto, clientes e servidores que se comunicam serão alocados à mesma estação se a soma dos custos de todos os seus segmentos de escalonamento envolvidos na interação, dividida pela soma dos seus respectivos custos de comunicação for menor que FC. A heurística por trás deste esquema é a mesma usada para módulos monosegmentados: a melhor decisão de alocação consiste, geralmente, em manter numa mesma estação módulos com uma demanda de comunicação elevada, relativamente aos seus custos de execução.

Quando for determinado que dois nós devem ser alocados a estações distintas, será inserido entre eles um novo nó para representar o custo de comunicação envolvido. Esse nó será chamado nó de comunicação ou subtarefa de comunicação.

Um grafo de comunicação conterá, portanto, nós que representam subtarefas de processamento (os segmentos de escalonamento dos módulos) e nós que representam subtarefas de comunicação. Os arcos ligando os nós do grafo de comunicações representam apenas as relações de precedência existentes entre eles. O custo de comunicação, que no grafo de escalonamento é representado pelo custo dos arcos, no grafo de comunicação corresponde ao custo dos nós de comunicação.

Uma vez construído um grafo de comunicação, o algoritmo de alocação e escalonamento deve, portanto, tentar encontrar um escalonamento no qual pares de nós de processamento, não conectados por nós de comunicação, sejam alocados a uma mesma estação, e pares de nós de processamento, conectados por um nó de comunicação, sejam alocados a estações distintas.

4.2.2 Alocação e Escalonamento de Subtarefas

Um grafo de comunicação indica para cada par de segmentos de escalonamento (subtarefas) comunicantes se eles devem, ou não, ser alocados a uma mesma estação, sem determinar, em caso afirmativo, a estação.

A partir de um grafo de comunicação, decisões coordenadas serão tomadas a fim de alocar e escalonar os seus nós (segmentos de escalonamento de módulos e subtarefas de comunicação). Para isso, será utilizado um algoritmo heurístico de busca que leva em consideração as relações temporais, de precedência e de alocação dos segmentos. O pseudocódigo do algoritmo é mostrado a seguir:

```
PROCEDIMENTO BuscarEscalonamento;
    INÍCIO
        determinar os instantes mais tarde de partida das subtarefas;
        criar o primeiro ponto de busca;
        ENQUANTO (lista de subtarefas prontas não vazia) E (EscalonamentoFactível)
        FAÇA
            SE  (um escalonamento factível é possivel a partir deste ponto)
                E ( ainda há mapeamentos válidos neste ponto) ENTÃO
                INÍCIO
                    gerar o próximo mapeamento válido;
                    escalonar as subtarefas da lista de pronto de acordo com o
                        mapeamento gerado;
                    gerar o próximo ponto de busca;
                FIM
            SENÃO
                INÍCIO
```

```
                descartar o ponto corrente;
                SE (existe um ponto de busca anterior) E (retornos permitidos)
                    ENTÃO retornar ao ponto anterior
                SENÃO
                    EscalonamentoFactível:=FALSO;
            FIM;
FIM;
```

O algoritmo assume que as estações físicas do sistema alvo (onde será executada o programa) têm um único processador e que as subtarefas (segmentos de escalonamento), são todas não preemptíveis. Logo, subtarefas prontas podem ser alocadas e escalonadas apenas a estações desocupadas, ou seja, a estações que não estão executando nenhuma subtarefa. O meio de comunicação também é considerado um recurso a ser alocado às subtarefas de comunicação.

Determinação dos prazos para partida das subtarefas

A ordem em que uma uma lista de subtarefas prontas for percorrida, irá condicionar o sucesso ou fracasso da busca por uma alocação e escalonamento factíveis. Por exemplo, suponhamos que no instante 0 duas tarefas, t_1 e t_2, ficaram prontas, t_1 com prazo 1 e t_2 com prazo 2, e ambas com tempo de execução 1. Se for decidido escalonar primeiro t_2 e depois t_1, apenas t2 cumprirá seu prazo. Alternativamente, se for escalada t_1 e depois t_2, ambas tarefas cumprirão os seus prazos. Neste exemplo, t_1 deve partir (iniciar a sua execução) no instante 0, a fim de que possa cumprir o seu prazo e t_2 deve partir até o instante 1 pelo mesmo motivo. Diz-se, então, que o prazo para partida, ou instante mais tarde de partida, de t_1 é o instante zero de t_2 é o instante um.

De uma maneira geral, o retardo na execução das subtarefas com os menores prazos para partida irá retardar o tempo de término de todo o conjunto de subtarefas que delas dependem. Por esse motivo, a fila de subtarefas prontas associada a cada ponto de busca será organizada em ordem crescente de prazos de partida e, quando houver empate, em ordem decrescente de número de sucessores. Esta organização da fila de pronto corresponde à implementação de uma heurística que pode ser denominada IMTP/NMSI (Instante Mais Tarde de Partida/Número Máximo de Sucessores Imediatos).

No cálculo dos instantes mais tarde de partida das subtarefas é usado o conceito de custo do caminho entre duas subtarefas, t_i e t_j, $CC(t_i,t_j)$, definido como a soma dos custos de todas as subtarefas (de processamento ou comunicação) que estão no caminho que liga t_i a t_j no grafo de comunicação, incluindo t_i e t_j.

Se t_f for a última subtarefa do grafo de comunicação e $P(t_f)$ o prazo para a sua conclusão, o IMTP de uma subtarefa qualquer do grafo de comunicação, t_i, será igual à diferença entre $P(t_f)$ e o custo do maior caminho entre t_i e t_f, ou seja,

$$IMTP(t_i) = P(t_f) - \max CC(t_i,t_f).$$

Além de instantes mais tarde de partida, subtarefas têm também associados instantes mais cedo de partida (IMCP). O IMCP de uma subtarefa é definidos como sendo igual ao IMCP da tarefa da qual é parte. Este, por sua vez, é definido como sendo igual ao instante de início de cada período de execução da tarefa somado ao seu instante de partida ("start time"), quando definido. Por exemplo, uma tarefa com período 20 e "start time" 3 terá como instantes mais cedo de partida 3,23,43, etc.

Geração dos pontos de busca

Um ponto de busca armazena as decisões tomadas até um determinado instante de busca. Ele descreve, portanto, uma alocação e um escalonamento parciais, e a fila de nós prontos para execução (lista de pronto) no instante de busca (instante corrente). O pseudocódigo a seguir mostra como o algoritmo cria o primeiro ponto de busca.

```
PROCEDIMENTO criar o primeiro ponto de busca;
    BEGIN
        (* iniciação do primeiro ponto de busca *)
        relógio:=0;
        mover o nó inicial do grafo de comunicação para a lista de pronto do
            ponto de busca corrente;
    END;
```

A criação do primeiro ponto de busca consiste, essencialmente, na iniciação da variável relógio (indica o instante de busca), e da fila de pronto. O relógio será iniciado com zero, indicando o instante inicial do intervalo de escalonamento das tarefas (módulos) representadas no GC. A fila de subtarefas prontas será iniciada com o nó inicial do GC, representando o disparo do conjunto de tarefas periódicas. A geração dos pontos de busca subsequentes é feita de acordo com o pseudocódigo mostrado a seguir:

```
PROCEDIMENTO gerar o próximo ponto de busca;
    INÍCIO
        relógio:= min( min(instante de partida mais cedo das subtarefas habilitadas),
                       min(instante de término das subtarefas ocupando recursos));
        mover para a nova fila de pronto as subtarefas da fila de pronto do ponto
            anterior que não foram escalonadas;
        mover para a fila de pronto as subtarefas que ficaram prontas neste instante;
    FIM;
```

A criação de pontos de busca está associada à definição de novos instantes de busca e da atualizacão da fila de pronto. A geração de um novo ponto de busca representa mudanças no estado das subtarefas, decorrentes do término da execução de subtarefas que estavam ocupando recursos ou da chegada do instante mais cedo de partida de subtarefas habilitadas (subtarefas cujos predecessores já terminaram de executar).

Geração e validação de mapeamentos

Um mapeamento define uma alocação e um escalonamento de subtarefas da fila de pronto a recursos livres. Se, num dado instante de busca, houver p subtarefas na fila de

pronto, $t_1,t_2,\ldots, t_p$, e n estações livres, um mapeamento $(t_{i1},t_{i2},\ldots, t_{in})$ representa a atribuição da iésima subtarefa da fila de pronto (i=1..p) à enésima estação livre. Por exemplo, o mapeamento (t_2,t_4) representa a atribuição da segunda e da quarta tarefas na fila de pronto à primeira e à segunda estações livres do sistema, respectivamente.

Um mapeamento pode determinar que alguns recursos permaneçam ociosos porque nenhuma das subtarefas prontas os necessita, ou porque se deseja manter o recurso livre à espera de alguma subtarefa que ainda não está pronta e tem um IMTP menor que o de uma subtarefa pronta desejando usar o recurso. Para tratar de maneira uniforme a ambos os casos, durante o mapeamento de subtarefas a recursos, são definidas subtarefas ociosas. Dessa forma, um mapeamento sempre atribui uma subtarefa a um recurso, se a subtarefa atribuída for uma subtarefa ociosa, o recurso permanecerá desocupado. Se num dado ponto de busca houver n recursos livres, serão adicionadas, para fins de mapeamento, n subtarefas ociosas, $o_1,o_2,\ldots,o_n$, à fila de pronto.

O processo de mapeamento consiste, basicamente, na geração de uma sequência lexicografica ordenada, de modo que um novo mapeamento pode ser gerado a partir do estado da fila de pronto e do último mapeamento produzido. Um mapeamento com efeito idêntico ao de outro anteriormente gerado não precisa ser considerado. Isto ocorre quando a única diferença entre um mapeamento e um anterior corresponde à atribuição de uma tarefa ociosa a um recurso ao qual já havia sido mapeada anteriormente outra tarefa ociosa. O efeito de ambos os mapeamentos é idêntico: o recurso em questão ficará livre. Embora recursos possam ficar ociosos, havendo na fila de pronto tarefas não ociosas, mapeamentos que não ocupem pelo menos um desses recursos são descartados.

Considere-se, a título de exemplo, que num dado ponto de busca a fila de subtarefas prontas contém duas subtarefas, t_1 e t_2, ordenadas nessa ordem, e que dois recursos estão livres. Nessa situação, podem ser gerados os seguintes mapeamentos:

(t_1,t_2); (t1,o1); (t1,o2);
(t2,t1); (t2,o1); (t2,o2);
(o1,t1); (o1,t2); (o1,o2);
(o2,t1); (o2,t2); (o2,o1).

Desses mapeamentos, (t1,o1) e (t1,o2); (t2,o1) e (t2,o2); (o1,t1) e (o2,t1); (o1,t2) e (o2,t2); (o1,o2) e (o2,o1) têm o mesmo efeito, podendo, em cada caso, um deles ser suprimido. Os mapeamentos (o1,o2) e (o2,o1) também serão suprimidos pois deixam todos os recursos ociosos. Dos 12 possíveis mapeamentos que podem se gerados usando o critério lexicográfico, serão considerados apenas os seguintes:

(t1,t2); (t1,o); (t2,t1); (t2,o); (o,t1); e (o,t2).

Uma vez gerado um mapeamento, a sua validade deve ser verificada. Para que um mapeamento seja válido, ele deve atender às seguintes condições:

- Se entre as subtarefas que ainda não foram escalonadas a de menor IMTP estiver pronta e for escalonável (o recurso que necessita estiver livre), ela deve ser escalonada. Ou seja, o recurso em questão não pode permanecer ocioso ou ser alocado a outra subtarefa;

- As subtarefas devem ter sido mapeadas, ou não, à mesma estação de seus predecessores imediatos, de acordo com o determinado pelo grafo de comunicação;
- Subtarefas de processamento que tenham uma mesma subtarefa de processamento como sucessora devem ser alocadas à mesma estação;
- Segmentos de escalonamento de um mesmo módulo devem ser alocados à mesma estação;
- Se foram mapeados segmentos que acessam servidores, e acessos multiserviços envolvendo esses servidores estiverem em curso, os segmentos recém mapeados devem fazer parte dos respectivos clientes que iniciaram os acessos multiserviços em andamento.

Verificando a factibilidade de um escalonamento parcial

A cada ponto de busca, o algoritmo procura determinar se as decisões tomadas até então ainda podem conduzir a um escalonamento factível. Isto é feito verificado as seguintes condições:

- Os instantes de partida mais tarde das subtarefas na fila de pronto devem ser maiores ou iguais ao instante corrente;
- O tempo disponível num dado recurso, entre o instante atual e o fim do intervalo de escalonamento, deve ser suficiente para executar as tarefas que deverão requerer esse recurso;
- Se um conjunto de subtarefas, $tp_1 .. tp_m$, na fila de pronto tiver um sucessor comum, ts, e o grafo de comunicação indicar que elas devem ser alocadas à mesma estação, a soma do instante atual com os tempos de execução dessas subtarefas deve ser menor ou igual que o instante de partida mais tarde da subtarefa sucessora (ts). Se $C(t_i)$ indicar o custo da subtarefa t_i e IMTP(ts) o instante mais tarde de partida da subtarefa ts, esta condição pode ser resumida pela seguinte expressão:

$$\text{instante corrente} + \sum_{i=1}^{m} C(tp_i) \leq IMTP(ts)$$

- Se uma subtarefa na fila de pronto, tp, tiver como sucessoras as tarefas $ts_1 .. ts_m$ com as quais deve cohabitar numa estação, e $ts_1..ts_m$ tiverem um sucessor comum, tss, com o qual elas devam também cohabitar, então:

$$\text{instante corrente} + C(tp) + \sum_{i=1}^{m} C(ts_i) \leq IMTP(tss)$$

As condições acima permitem fazer uma avaliação prévia do futuro próximo ("look ahead"). Se nesse horizonte o escalonamento parcial disponível se mostrar não factível, com certeza o ponto atual não poderá conduzir a um escalonamento total factível, devendo, portanto, ser abandonado.

Exemplo de Escalonamento

Dado o grafo de escalonamento da figura 4.5, o algoritmo descrito nesta seção produzirá o escalonamento ilustrado pela figura 4.6.

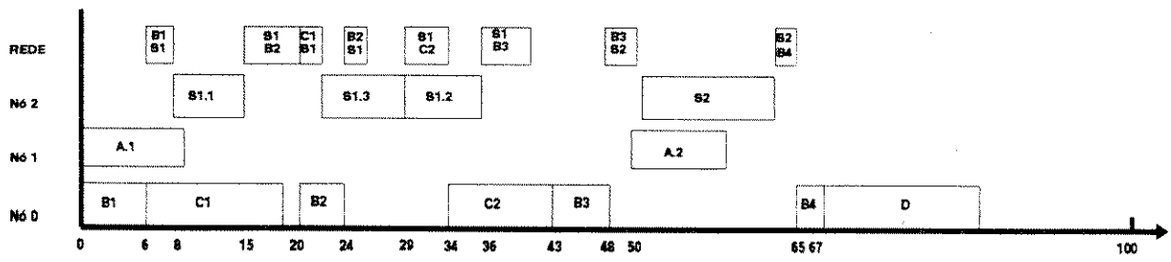


Figura 4.6: Escalonamento estático gerado para o grafo da figura 4.6.

4.3 Preparando um escalonamento estático para tratar chegadas dinâmicas de módulos aperiódicos

Os escalonamentos produzidos por algoritmos estáticos de escalonamento de tarefas periódicas, incluindo o algoritmo proposto na seção 4.2, costumam apresentar três características que afetam negativamente o escalonamento dinâmico de tarefas aperiódicas:

1) A distribuição da carga de processamento entre as estações tende a não ser bem balanceada;

2) Intervalos livres tendem a agrupar-se no final do intervalo de escalonamento; e

3) Os instantes de início de cada subtarefa são rigidamente definidos.

Quando a carga de processamento alocada às estações não é balanceada, a probabilidade de que se consiga encontrar um escalonamento para tarefas aperiódicas chegando a estações sobrecarregadas tende a ser pequena. De maneira semelhante, a probabilidade de escalonar uma tarefa aperiódica pode ser muito reduzida se ela chegar durante o intervalo de tempo que concentra a execução das tarefas periódicas. Finalmente, a definição de instantes rígidos para o início da execução das tarefas periódicas reduz a margem de manobra do escalonador dinâmico. Em diversas situações, provocar pequenos atrasos nas partidas das tarefas periódicas permite atender a tarefas aperiódicas com folga reduzida, sem prejuízo do cumprimento dos prazos das tarefas periódicas.

Nesta seção, serão discutidos alguns esquemas que visam aumentar as chances de um algoritmo dinâmico ser bem sucedido na determinação de um escalonamento para tarefas aperiódicas, na presença de um escalonamento gerado "off-line" para tarefas periódicas. No item 4.3.1, serão discutidas algumas técnicas simples para melhorar o balanceamento da carga do sistema a fim de contornar as duas primeiras características de um escalonamento estático acima enumeradas, e, no item 4.3.2, serão discutidos dois esquemas para contornar a terceira característica. Estes esquemas, um com efeitos locais às estações e outro global ao sistema, procuram definir janelas de escalonamento para as tarefas, em substituição aos instantes estáticos de início e término de execução gerados pelo algoritmo discutido na seção anterior.

Os esquemas discutidos nesta seção foram originalmente propostos e analisados por Ramamritham e Adán (1990), e posteriormente por Ramamritham, Fohler e Adán (1993), juntamente com outros aspectos, não inseridos no contexto deste trabalho, relacionados à alocação e escalonamento estáticos de tarefas periódicas complexas.

4.3.1 Balanceamento da carga imposta às estações

Para tentar balancear a carga imposta às estações de um sistema distribuído, durante a alocação e escalonamento de tarefas periódicas, foram desenvolvidas duas estratégias: a Estratégia da Carga Máxima (ECM) e a Estratégia do Fator de Ociosidade (EFO).

Na ECM, as decisões de alocação do algoritmo são limitadas pela carga máxima que se permite impor a cada uma das estações do sistema. O Algoritmo tenta produzir uma alocação na qual cada estação não receba mais que um dado percentual da carga total de processamento das tarefas a escalonar. Esse percentural de carga máxima, alocável a qualquer estação do sistema, é fornecido pelo parâmetro CargaMáxima. Quando não for possível encontrar um escalonamento para um determinado valor de CargaMáxima, será feita uma nova tentativa utilizando um novo valor. O novo de CargaMáxima será obtido adicionando-lhe um incremento definido pelo parâmentro IncCargaMáxima. Esse procedimento será repetido até que um escalonamento bem sucedido seja encontrado ou até que CargaMáxima atinja o valor 1 (100%), o que equivale a permitir que toda a carga possa ser alocada a uma única estação.

Na EFO, introduzem-se intervalos de ociosidade após o término de cada subtarefa escalonada, reduzindo a quantidade de subtarefas que pode ser alocada a uma estação e forçando o algoritmo a tentar distribuir as tarefas entre outras estações do sistema, o que pode contribuir para a melhoria do balanço da carga do sistema, no espaço (entre as estações) e no tempo (ao longo do intervalo de escalonamento).

Intervalos de ociosidade são introduzidos em função do valor do parâmetro FatorDeOciosidade. Esse parâmetro indica que percentual do tempo de execução de cada subtarefa deve ser deixado livre após o seu término. Por exemplo, se FatorDeOciosidade for igual a 0,2 e uma subtarefa t requerer 50 unidades de tempo para executar, após o seu término o escalonador estático deverá deixar a estação ociosa por pelo menos 10 unidades de tempo.

Cada vez que o algoritmo falha em encontrar um escalonamento com um dado FatorDeOciosidade, este recebe um decremento determinado pelo parâmetro DecFatorDeOciosidade. Este procedimento é repetido até que o conjunto de tarefas seja escalonado com sucesso ou FatorDeOciosidade atinja o valor zero.

O pseudocódigo do algoritmo de alocação e escalonamento com a inclusão das estratégias de balanceamento de carga é o seguinte:

```
PROCEDIMENTO BuscarEscalonamentoBalanceado
            (CargaMáxima, FatorDeOciosidade, IncCargaMáxima,DecFatorDeOciosidade);
  INÍCIO
      Sucesso:=FALSO;
      ENQUANTO ((CargaMáxima <= 100) E NÃO Sucesso) FAÇA
      INÍCIO
          FatorDeOciosidadeCorrente:=FatorDeOciosidade;
          ENQUANTO ((FatorDeOciosidadeCorrente >= 0) E (NÃO Sucesso)) FAÇA
          INÍCIO
              Sucesso:= BuscarEscalonamento;
              FatorDeOciosidadeCorrente:=
                      FatorDeOciosidadeCorrente - DecFatorDeOciosidade;
          FIM
          CargaMáxima:=CargaMáxima + IncCargaMáxima;
```

FIM
FIM;

O pseudocódigo do procedimento BuscarEscalonamento, referênciado no algoritmo acima, foi apresentado no ítem 4.4.2. A estratégia ECM é aplicada quando o parâmetro CargaMáxima for iniciado com um valor inferior 100%. Do mesmo modo, a estratégia EFO é aplicada quando FatorDeOciosidade for iniciado com um valor maior que zero. O algoritmo aplica, portanto, uma das estratégias, ou ambas, dependendo dos valores iniciais de CargaMáxima e FatorDeOciosidade.

Dadas as estratétias ECM e EFO, o algoritmo de escalonamento original, identificado no algoritmo estendido mostrado acima pelo procedimento BuscarEscalonamento, deve sofrer as seguintes extensões:

- Se uma subtarefa com tempo de execução c for alocada a uma determinada estação, e escalonada para inciar sua execução no instante i, a execução de uma nova tarefa nessa estação não poderá iniciar-se antes do instante i+c+FatorDeOciosidade*c.
- Ao verificar a factibilidade de um escalonamento parcial deve-se, na projeção do futuro próximo ("look ahead"), levar em consideração o tempo ocioso que será inserido após o término das subtarefas a escalonar;
- Durante a verificação da validade de um mapeamento, deve-se assegurar que nenhuma estação recebeu uma fração da carga total do sistema superior ao indicado por CargaMáxima.

A medição do balanceamento da carga de um sistema composto das estações e_1, e_2,..., e_{ne} pode ser feita usado uma métrica que denominaremos desvio de carga, DC, definida da seguinte forma:

$$DC = \sum_{i=1}^{ne} |CargaMédia - Carga(e_i)|$$

Na expressão acima, CargaMédia é calculada dividindo a carga computacional total do conjunto de tarefas sendo considerado no intervalo de escalonamento pelo número de estações do sistema alvo, ne, e Carga(e_i) corresponde à carga efetivamente alocada à estação e_i. Ambos fatores são expressos em termos de percentuais, ou frações, da carga total. O desvio de carga é uma indicação da falta de balanceamento da carga do sistema. Conforme pode ser facilmente observado na expressão acima, quanto melhor for distribuída a carga entre as estações do sistema, menor será o correspondente desvio de carga.

Considere-se, por exemplo, que num sistema com duas estações, 40% da carga correspondente a um dado conjunto de tarefas foi alocado a umas das estações e os restantes 60% da carga foram alocados à outra estação. Neste caso, CargaMédia será 50% e o desvio de carga será dado por abs(50-40)+abs(50-60), ou seja 20%. Se 50% da carga computacional do conjunto de tarefas no intervalo de escalonamento tivesse sido alocado

a cada estação, o sistema teria uma um balanço de carga perfeito e o seu desvio de carga seria igual a 0.

Os efeitos do uso combinado das estratégias ECM e EFO no escalonamento das tarefas representadas na figura 4.5 é mostrado na figura 4.7.

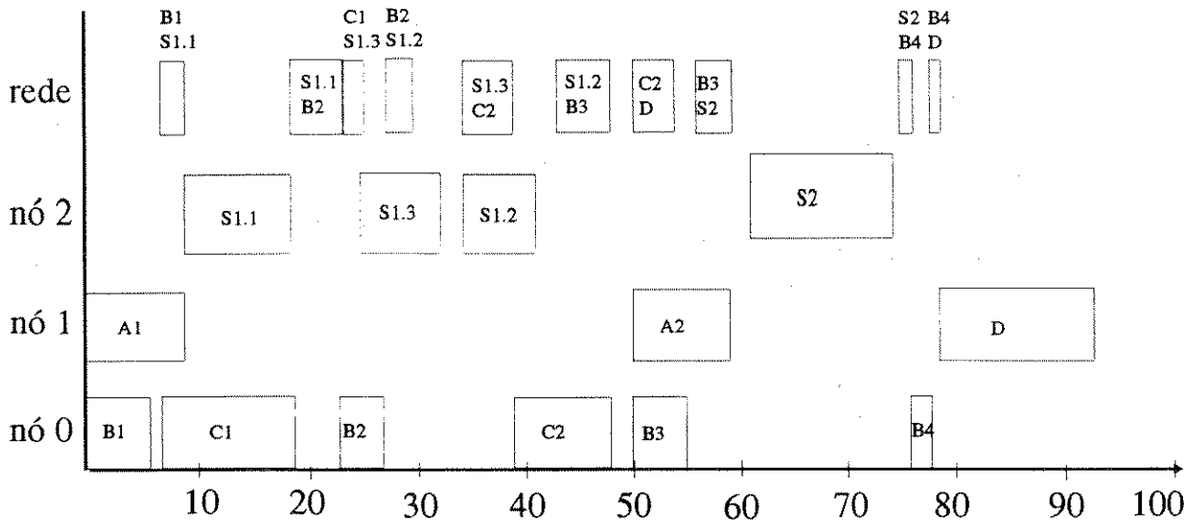


Figura 4.7: Alocação e escalonamento da aplicação representada na figura 4.6, usando a ECM combinada com a EFO

Comparando o escalonamento mostrado na figura 4.7 com aquele produzido sem a utilização das estratégias de balanceamento de carga (figura 4.6), pode-se observar que o emprego das estratégias ECM e EFO levou à alocação do módulo D à estação 1 e, consequentemente, a uma carga melhor distribuída. Adicionalmente, pequenos intervalos livres foram inseridos pela estratégia EFO entre os segmentos de escalonamento. Esses intervalos livres poderão ser usados quando do escalonamento dinâmico de tarefas aperiódicas. A melhor distribuição de carga pode ser comprovada comparando os desvios de carga associados aos dois escalonamentos: o escalonamento gerado sem o auxílio das estratégias de balanceamento tem um desvio de carga de 33%, enquanto que no escalonamento que emprega as estratégias ECM e EFO, o desvio de carga é de apenas 6%.

As estratégias ECM e EFO procuram balancear apenas a carga de processamento entre as estações, em consequência, elas podem aumentar a carga alocada ao meio de

comunicação, conforme pode ser observado na figura 4.7, podendo levar a uma redução apreciável da taxa de vazão de mensagens assíncronas.

Pelo fato das mensagens assíncronas não estarem envolvidas com a transferência de informações associada a restrições críticas de tempo, não é, a rigor, imprescindível a sua entrega. Mesmo assim, este problema deve ser melhor apreciado, pois o bom desempenho de um sistema pode estar associado a uma boa vazão média das mensagens assíncronas.

4.3.2 Determinação de janelas de escalonamento

Dado um escalonamento estático factível, os esquemas que serão apresentados neste item procuram oferecer ao escalonador dinâmino um maior grau de liberdade para o escalonamento das tarefas aperiódicas.

A idéia básica por trás dos esquemas apresentados é encontrar instantes de partida mais cedo e instantes de partida mais tarde para as tarefas periódicas estaticamente escalonadas, respeitando as suas restrições de precedência. Ou seja, em vez de requerer que as subtarefas iniciem sua execução num instante específico, será gerada uma janela de escalonamento para cada subtarefa. Desse modo, a execução das subtarefas poderá ser iniciada em qualquer ponto da janela que lhes foi associada, sem que isso viole a factibilidade de todo o conjunto de tarefas.

Dada uma janela para cada subtarefa estaticamente alocada e escalonada, o escalonador dinâmico poderá retardar o início de uma subtarefa periódica a fim de executar uma tarefa aperiódica que acaba de chegar à estação, desde que assegure que a subtarefa periódica retardada será iniciada dentro da sua respectiva janela.

Serão considerados dois esquemas para a produção de janelas de escalonamento: o esquema local para determinação dos instantes mais tarde de partida das subtarefas (IMTPL) e o esquema global para determinação dos instantes mais tarde de partida das subtarefas (IMTPG).

Tanto no IMTPL como no IMTPG, o início da janela de uma subtarefa corresponde ao instante a partir do qual ela foi estaticamente escalonada para execução, pois o algoritmo estático escalona subtarefas o mais cedo possível.

A fim de manter os esquemas propostos simples, eles não modificam a ordem de execução das subtarefas determinada estaticamente, pois, em caso contrário, seria necessário proceder a uma modificação global da alocação e escalonamento das subtarefas.

Subtarefas com relações lógicas de precedência têm a ordem de execução automaticamente assegurada por essa relação. Qualquer esquema proposto deve necessáriamente respeitar essas relações, sob pena de tornar o escalonamento não factível. O mesmo não ocorre com as demais subtarefas. Para elas, a manutenção da ordem em que foram escalonadas será assegurada através da definição de uma precedência física. Enquanto a precedência lógica é parte da especificação do sistema, e

independente das decisões de alocação e escalonamento tomadas, a precedência física corresponde à ordem particular em que as subtarefas foram escalonadas.

Esquema Local para Determinação do Instante Mais Tarde de Partida das subtarefas (IMTPL)

No esquema IMTPL, a precedência física é dada pela ordem em que as subtarefas alocadas à uma dada estação foram escalonadas para execução. Para que as decisões tomadas numa estação não tenham que ser propagadas a outras estações com as quais ocorre troca de mensagens, neste esquema o instante mais tarde de início de uma subtarefa com sucessores remotos será sempre igual ao seu instante mais cedo de início (o instante de início originalmente computado pelo algoritmo), i.e., a sua janela de escalonamento será nula.

Para calcular os IMTP das subtarefas, o escalonamento de cada estação é ordenado em ordem topológica reversa e o instante de início mais tarde de uma subtarefa t_i (sem sucessores remotos) é definido como sendo:

$$IMTP(t_i)=Min[P(t_i),mim(IMTP(sucessores(t_i)))]-(C(t_i));$$

Na expressão acima, sucessores(t_i) inclui tanto os sucessores lógicos como os sucessores físicos da subtarefa t_i.

Conhecidos os instantes mais cedo e mais tarde de início das subtarefas periódicas, o escalonador dinâmico de cada estação passa a ter maior flexibilidade para escalonar as tarefas aperiódicas quando de sua chegada à estação. A fim de atender a uma tarefa aperiódica recém chegada, o escalonador dinâmico poderá alterar o escalonamento gerado para as subtarefas periódicas, desde que elas sejam executadas na ordem determinada estaticamente (precedência física) dentro das suas respectivas janelas de escalonamento (definidas por seus instantes mais cedo e mais tarde de início).

Esquema Global de Determinação do Instante Mais Tarde de Partida (IMTPG)

O esquema global assume que o processador de comunicação de cada estação conhece a ordem em que todas as mensagens do sistema deverão ser enviadas. Neste esquema, cada mensagem terá também uma janela de transmissão, de modo que uma mensagem poderá ser transmitida quando todas as mensagens anteriores a ela, na ordem definida estaticamente, tiverem sido transmitidas e o instante corrente estiver dentro de sua janela de transmissão.

No esquema global, quando da criação de uma janela de escalonamento numa estação, mudanças podem ser propagadas para outras estações. Entretanto, a ordem em que as tarefas devem ser executadas em cada estação (determinada pelo algoritmo original) será mantida. Isto é válido também para o canal de comunicação: mensagens deverão ser transmitidas na ordem inicialmente determinada pelo escalonador estático.

A determinação de IMTP globais é semelhante à determinação de IMTP locais, com a diferença de que agora são consideradas também as tarefas de comunicação. A

precedência física entre subtarefas é estabelecida, portanto, entre todas as tarefas do sistema, sejam elas de processamento ou de comunicação.

A tabela 4.1 mostra as janelas de escalonamento produzidas pelos esquemas local e global para o conjunto de tarefas representado pelo grafo de escalonamento mostrado na figura 4.5, em três cenários: 1) quando o algoritmo de escalonamento não emprega as estratégias de balanceamento de carga (v.figura 4.6); 2) quando o algoritmo emprega a ECM; e 3) quando o algoritmo combina a utilização das estratégias ECM e EFO (v. figura 4.7).

Pode-se observar na tabela que as janelas de escalonamento, especialmente quando determinadas pelo IMTPG, eliminam as vantagens da inserção de vazios após o término de uma subtarefa (estratégia EFO) para fins de escalonamento dinâmico de tarefas aperiódicas. Na realidade, quando se dispõe de janelas de escalonamento, a estratégia EFO reduz a flexibilidade do escalonador dinâmico, pois o uso da EFO tende a reduzir a largura das janelas, podendo o sistema ficar ocioso em períodos durante os quais não há tarefas aperiódicas pendentes de execução. Nesses períodos de ociosidade, uma subtarefa periódica já poderia estar sendo executada para antecipar a sua conclusão e propagar os instantes de ociosidade para o futuro, quando poderiam ser de utilidade para a execução de tarefas aperiódicas a chegar. Ou seja, quando são geradas janelas de execução, a utilidade da EFO restringe-se à busca de um melhor balanço de carga. Caberia aqui questionar os benefícios dessa estratégia relativamente à ECM e a validade de combinar ambas estratégias. Estas questões serão retomadas no capítulo 5, onde serão apresentados os resultados obtidos de uma avaliação experimental do algoritmo estático de alocação e escalonamento.

		INSTANTE DE INÍCIO					
		LOCAL			GLOBAL		
Subtarefa	t. exec.	sem B.C.	ECM	ECM+EFO	sem B.C.	ECM	ECM+EFO
A1	9	0-41	0-41	0-41	0-41	0-41	0-41
A2	9	50-91	50	50	50-91	50-77	50-77
B1	6	0	0	0	0-19	0-18	0-18
B2	4	20	20	23	20-42	20-41	23-41
B3	5	43	43	50	43-62	43-61	50-61
B4	2	65-84	65	76	65-84	65-83	76-83
C1	12	6	6	7	6-27	6-26	7-26
C2	9	34	34	39	34-53	34-52	39-52
D	14	67-86	68	79	67-86	68-86	79-86
S1.1	7	8	8	9	8-27	8-26	9-26
S1.2	7	29	29	34	29-50	29-49	34-49
S1.3	7	22	22	25	22-41	22-40	25-40
S2	13	51	51	59	51-70	51-69	59-69
B1-S1-1	2	6	6	7	6-25	6-24	7-24
C1-S1.3	2	20	20	23	20-39	20-38	23-38
S1.1-B2	5	15	15	18	15-34	15-33	18-33
S1.3-C2	5	29	29	34	29-48	29-47	34-47
B2-S1.2	2	24	24	28	24-46	24-45	28-45
C2-D	4		43	50		43-62	50-62
S1.2-B3	5	36	36	43	36-57	36-56	43-56
B3-S2	3	48	48	56	48-67	48-66	56-66
S2-B4	1	64	64	75	64-83	64-82	75-82
B4-D	1		67	78		67-85	78-85

Tabela 4.1: Escalonamento, com janelas, para os conjuntos de tarefas correspondentes aos grafos mostrados na figura 4.5.

4.4 Escalonamento dinâmico

Além das tarefas periódicas, cujos padrões de ativação determinísticos permitem o uso de técnicas de escalonamento "off-line", a estratégia de escalonamento do HSTER deve levar em consideração também as tarefas aperiódicas e as tarefas sem restrições temporais. Estas tarefas, devido ao seu padrão não determinístico de ativação, devem, necessariamente, ser escalonadas dinamicamente.

Dado que a ênfase da estratégia de escalonamento, e deste trabalho como um todo, são os componentes críticos de uma aplicação, serão apenas sugeridas, em linhas gerais, estratégias que permitam a coexistência de tarefas aperiódicas, tarefas sem restrições de tempo e tarefas periódicas. Deve ser enfatizado que o sucesso no escalonamento dinâmico de tarefas não críticas está intimamente ligado ao escalonamento gerado "off-line" para as tarefas críticas, e que, portanto, a coexistência entre as duas categorias de tarefas deve ser considerada já durante o escalonamento das tarefas críticas. No caso do HSTER, convém recordar que a preocupação com o escalonamento dinâmico de tarefas foi uma das principais razões que motivaram a busca de estratégias de balanceamento de carga para incorporar ao algoritmo "off-line".

4.4.1 Escalonamento de tarefas aperiódicas

Há duas abordagens principais para o escalonamento dinâmico de tarefas aperiódicas, na presença de um escalonamento gerado "off-line" para as tarefas periódicas: 1) utilizar em cada estação um algoritmo que escalone as tarefas para execução levando em conta os seus prazos, porém sem a preocupação de garantir que será possível cumprir o prazo de todas as tarefas aceitas pela estação; 2) utilizar algoritmos que somente aceitem para execução as tarefas para as quais seja possível garantir os recursos que lhes permitam cumprir os prazos.

Quando a primeira abordagem é utilizada, se o prazo de uma tarefa for perdido e um tratador dessa exceção tiver sido definido, a execução da tarefa será interrompida e o tratador da exceção executado. Se um tratador não tiver sido definido, a tarefa continuará competindo por recursos até o seu término, utilizando o prazo original.

O escalonamento dinâmico de tarefas aperiódicas sem usar o conceito de garantia é de aplicação simples e, na medida em que as tarefas aperiódicas têm prazos não críticos, perfeitamente adequado à maioria das situações. Entretanto, às vezes essa pode não ser a estratégia mais indicada, ou mesmo totalmente inaceitável. Essa estratégia não é adequada, por exemplo, quando após a perda do prazo a tarefa perde a importância para o sistema. Quando isto acontece, a principal consequencia da continuação da execução da tarefas é o desperdício de recursos que poderiam ser usados para atender a tarefas ainda em condições de cumprir os seus prazos.

Por outro lado, o escalonamento dinâmico sem o uso do conceito de garantia é inaceitável quando tarefas não críticas compartilham um servidor com tarefas críticas. Neste caso, o servidor deve estar disponível para receber às requisições das tarefas críticas nos instantes determinados "off-line". Isto implica em que uma tarefa não crítica

somente poderá ser autorizada a acessar o servidor compartilhado se for possível garantir que a transação será concluída a tempo de atender às requisições das tarefas críticas previamente escalonadas.

No HSTER, o escalonamento dinâmico de tarefas, sem usar o conceito de execução garantida, é relativamente simples: basta executar as tarefas aperiódicas nos períodos livres do intervalo de escalonamento das tarefas periódicas, retardando as tarefas periódicas, se necessário, quando definidas janelas de escalonamento. A ordem em que diversas tarefas aperiódicas, habilitadas num dado instante, devem ser executadas pode ser determinada pela política associada a algum algoritmo dinâmico simples como, por exemplo, executar primeiro as tarefas com o menor prazo ("earliest deadline first"), ou executar primeiro as tarefas com a menor folga ("least laxity first").

A implementação do conceito de execução garantida de tarefas aperiódicas independentes chegando às estações de um sistema distribuído consiste, basicamente, na determinação do tempo disponível nas estações entre os instantes de chegada das tarefas e os instantes em que vencem os seus prazos [Melo, Magalhães e Adán-Coello (1992)]. Já no caso de tarefas aperiódicas que utilizam servidores, é necessário utilizar abordagens mais complexas na linha, por exemplo, do algoritmo de escalonamento distribuído do sistema Spring [Ramamritham et al. (1989)]. O emprego desse tipo de algoritmo, em combinação com o algoritmo "off-line" discutido neste capítulo, é um problema interessante que transcende, no entanto, o escopo deste trabalho.

Independentemente de qual abordagem for usada, a escalonabilidade de tarefas aperiódicas será grandemente influenciada pela alocação e escalonamento produzidos "off-line" para as tarefas periódicas. Nesse sentido, o uso das estratégias para balanceamento de carga e para a determinação de janelas de escalonamento, apresentadas neste capítulo, podem contribuir para um apreciável aumento do número de tarefas aperiódicas capazes de cumprir os seus prazos.

4.4.2 Escalonamento dinâmico de tratadores de exceções

A política a seguir quando da ocorrência de uma exceção relacionada ao cumprimento das restrições temporais de um módulo é definida estaticamente durante a configuração dos programas que usam o módulo. Por outro lado, a detecção de exceções e o seu efetivo tratamento ocorrem em tempo de execução dos programas.

O tratamento de uma exceção pode ficar exclusivamente a cargo do sistema operacional (quando o programa de configuração da aplicação sendo executada utilizar as alternativas "skip" ou "kill" como políticas de tratamento), ou pode requerer a execução do tratador definido pelo módulo envolvido com a exceção sinalizada (quando for usada a alternativa "handle"). Neste último caso, o tratador deverá ser escalonado para execução dinamicamente.

Na medida em que a execução das tarefas periódicas é garantida "off-line", a única exceção que elas devem prever é a não chegada das mensagens de habilitação nos instantes previamente determinados pelo escalonador estático. Isso pode ocorrer em

decorrêcia de erros no meio de comunicação ou em uma estação remota. Nesse caso, a execução do tratador da exceção, se especificado, deverá usar o intervalo de tempo reservado para a tarefa periódica, pois o tratador nada mais é que uma das versões da tarefa sob exceção.

No caso das tarefas aperiódicas, podem ser definidos tratadores para a perda de um prazo, ou para a impossibilidade de garantir o seu cumprimento. O escalonamento da execução destes tratadores deve ser feito dinamicamente quando da ocorrência da exceção, empregando os mesmos algoritmos usados para o escalonamento das tarefas aperiódicas. Podendo-se, portanto, utilizar, ou não, o conceito de execução garantida.

4.4.3 Escalonamento de tarefas sem restrições de tempo

As tarefas sem restrições de tempo estão normalmente associadas a atividades de médio e longo prazos realizadas em "background". Essas tarefas não devem competir por recursos com as tarefas com restrições de tempo, devendo ser executadas apenas quando não houver tarefas com restrições de tempo habilitadas. Pode-se, para isso, utilizar políticas convencionais, tais como, a alocação do processador às tarefas durante fatias de tempo ("time sharing"), ou a preempção por prioridade.

4.5 Implementação do escalonador estático de tarefas periódicas

O algoritmo de escalonamento de tarefas periódicas descrito neste capítulo, incluindo as estratégias para balanceamento de carga e geração de janelas de escalonamento, foi implementado em C++, em ambiente UNIX [Kernighan and Pike (1984)], como uma extensão a um escalonador disponível [Ramamritham (1990)].

O escalonador recebe como entrada a descrição do grafo de escalonamento de um programa e as características do sistema onde o programa será executado (número de estações, número de vias de comunicação, etc.), e gera, como saída, uma alocação e um escalonamento factíveis, quando capaz de encontrá-los.

A descrição do grafo de escalonamento de um programa, bem como as características do sistema onde o programa será executado, é feita usando uma linguagem especificamente desenvolvida para esse propósito, denominada Linguagem de Descrição de Grafos de Escalonameto (LDGE). Um programa fonte em LDGE é traduzido para o conjunto de estruturas de dados usadas pelo escalonador por um processador da LDGE, implementado usando as ferramentas Yacc e Lex do ambiente UNIX.

A figura 4.8 apresenta, a título de exemplo, o programa em LDGE usado para a descrição do grafo de escalonamento da figura 4.5. Conforme pode ser observado, além de descrever o grafo de escalonamento propriamente dito, numa especificação escrita em LDGE pode-se especificar o número máximo de "backtracks" que o algoritmo poderá realizar, características do sistema alvo (número de estações no sistema e número de vias de comunicação que interligam as estações) e valores iniciais para os parametros FatorDeOciosidade ("MAXGAPFACT") e CargaMáxima ("MAXCPULOAD").

```
DISTRIBUTED SCHEDULING 3 CPUS 1 NET
CF-STRATEGIE 100 BACKTRACKS 1 LOOP 0 MAXGAPFACT 100 MAXCPULOAD

TASK Teste OVERALL_DEADLINE=100
    ENTRY SUBTASK I
            REDUNDANCY=0 COMPUTATION TIME=0 DL=999999 ST=0
        SUCESSOR      A.1 COSTS=0
        SUCESSOR      B.1 COSTS=0
        SUCESSOR      C.1 COSTS=0
    SUBTASK A.1 MODULE A
            REDUNDANCY=0 COMPUTATION TIME=9 DL=50 ST=0
        SUCESSOR      A.2 COSTS=0
    SUBTASK B1 MODULE B
            REDUNDANCY=0 COMPUTATION TIME=6 DL=999999 ST=0
        SUCESSOR      S1.1 COSTS=2
    SUBTASK C1 MODULE C
            REDUNDANCY=0 COMPUTATION TIME=12 DL=100 ST=0
        SUCESSOR      S.1.3 COSTS=2
    SUBTASK A2   MODULE A
            REDUNDANCY=0 COMPUTATION TIME=9 DL=999999 ST=50
        SUCESSOR      F COSTS=0
    SUBTASK S1.1 SERVER S
            REDUNDANCY=0 COMPUTATION TIME=7 DL=999999 ST=0
        SUCESSOR      B2 COSTS=5
    SUBTASK S1.3 SERVER S
            REDUNDANCY=0 COMPUTATION TIME=7 DL=999999 ST=0
        SUCESSOR      C2 COSTS=5
    SUBTASK B2 MODULE B
            REDUNDANCY=0 COMPUTATION TIME=4 DL=999999 ST=0
        SUCESSOR      S1.2 COSTS=2
    SUBTASK C2 MODULE C
            REDUNDANCY=0 COMPUTATION TIME=9 DL=100 ST=0
        SUCESSOR      D COSTS=4
    SUBTASK S1.2 SERVER S LOCK 1
            REDUNDANCY=0 COMPUTATION TIME=7 DL=999999 ST=0
        SUCESSOR      B3 COSTS=5
    SUBTASK B3 MODULE B
            REDUNDANCY=0 COMPUTATION TIME=5 DL=999999 ST=0
        SUCESSOR      S2 COSTS=3
    SUBTASK S2 SERVER S UNLOCK 1
            REDUNDANCY=0 COMPUTATION TIME=13 DL=999999 ST=0
        SUCESSOR      B4 COSTS=1
    SUBTASK B4 MODULE B
            REDUNDANCY=0 COMPUTATION TIME=2 DL=999999 ST=0
        SUCESSOR      D COSTS=1
    SUBTASK D
            REDUNDANCY=0 COMPUTATION TIME=14 DL=999999 ST=0
        SUCESSOR      F COSTS=0
    EXIT SUBTASK F
            REDUNDANCY=0 COMPUTATION TIME=0 DL=100 ST=0
```

Figura 4.8: Descrição em LDGE do grafo de escalonamento da figura 4.6.

4.6 Sumário e Comentários Finais

Programas desenvolvidos usando as linguagens LPM-RC e LCM-RC, segundo o modelo de programação HSTER, podem incluir tarefas (módulos) periódicas com restrições críticas de tempo, tarefas aperiódicas com restrições não críticas de tempo e tarefas sem restrições de tempo. O escalonamento de cada uma dessas classes de tarefas apresenta particularidades que conduzem a uma estratégia de escalonamento em três níveis: 1) escalonamento "off-line" de tarefas periódicas, visando que seus prazos sempre serão cumpridos; 2) escalonamento dinâmico de tarefas aperiódicas, visando cumprir a maioria dos prazos destas tarefas; e 3) escalonamento dinâmico de tarefas sem restrições de tempo, usando os intervalos de tempo em que não haja tarefas com restrições de tempo habilitadas, visando boa utilização de recursos e equidade (ou observância de prioridades) no atendimento a estas tarefas.

Além de discutir em linhas gerais a estratégia de escalonamento das três classes de tarefas presentes num programa HSTER, este capítulo dá ênfase ao cumprimento das restrições temporais de tarefas críticas, propondo, para isso, um algoritmo específico para a alocação e escalonamento dessas tarefas em sistemas distribuídos.

O algoritmo de escalonamento de tarefas periódicas considera as restrições de tempo, precedência e localização das tarefas, assim como, a necessidade de acessar em modo exclusivo servidores compartilhados. A alocação e escalonamento de tarefas com essas restrições é um problema computacionalmente intratável, sendo, por isso, necessário tentar alocá-las e escaloná-las "off-line". Ainda assim, à medida que o número de tarefas a escalonar cresce, o número de possibilidades a avaliar durante a busca por um escalonamento tende a crescer exponencialmente, razão pela qual o algoritmo proposto emprega heurísticas que tratam de determinar, o quanto antes, se as decisões tomadas até um determinado ponto de busca podem, ou não, conduzir a um escalonamento factível. Quanto mais puder ser restrita a busca, menos tempo e recursos serão necessários para determinar um escalonamento, ou a impossibilidade de encontrá-lo. Para restringir a busca o algoritmo procura identificar o maior número possível de caminhos de busca não factíveis e descartar o maior número possível de mapeamentos inválidos, empregando testes simples. Esses testes podem ser generalizados, o que, certamente, provocará um aumento do custo da busca. Os eventuais benefícios resultantes da generalização dos testes deverão, portanto, ser avaliados face ao esperado aumento do custo de produção de escalonamentos.

O não determinísmo associado à habilitação (chegada) de tarefas aperiódicas leva à necessidade de escaloná-las para execução dinamicamente, usando os intervalos de tempo não alocados "off-line" para a execução das tarefas periódicas. Para isso, duas abordagens foram sugeridas: 1) utilizar algoritmos dinâmicos simples dirigidos por tempo, tais como o menor prazo ou a maior folga; 2) Usar algoritmos mais elaborados que aceitem para execução apenas as tarefas para as quais possa ser garantido o cumprimento dos seus prazos. O uso da primeira abordagem é, em geral, aceitável quando for possível oferecer uma taxa média de cumprimento de prazos razoável, pois

tarefas aperiódicas não têm restrições críticas de tempo, sendo, por isso, tolerável que prazos sejam eventualmente perdidos. A segunda abordagem deve, no entanto, ser empregada quando as tarefas aperiódicas compartilham servidores com as tarefas periódicas. Neste caso, é preciso garantir que uma tarefa aperiódica não estará de posse de um servidor quando uma tarefa periódica o necessite.

Independentemente da abordagem empregada pelo escalonador dinâmico, as características do escalonamento gerado para as tarefas periódicas terão grande influência na escalonabilidade das tarefas aperiódicas. Se a carga de processamento imposta estaticamente às estações for muito desbalanceada, tarefas aperiódicas chegando a estações sobrecarregadas terão pouca chance de serem atendidas a tempo de cumprir os seus prazos.

Para procurar balancear a carga de processamento imposta às estações de um sistema distribuído, foram propostas duas estratégias simples para uso durante a alocação e escalonamento de tarefas periódicas: a Estratégia da Carga Máxima (ECM) e a Estratégia do Fator de Ociosidade (EFO). A ECM limita a carga que pode ser imposta a uma estação e a EFO gera períodos livres após o término da execução das tarefas, o que, por sua vez, forçará a que elas sejam mais uniformemente distribuídas.

Além de uma carga mal balanceada, a definição de instantes de tempo rígidos para o início da execução das tarefas periódicas pode inviabilizar o cumprimento dos prazos de tarefas aperiódicas. Isto pode ocorrer mesmo em situações nas quais pequenas alterações nos instantes de início das tarefas periódicas permitiriam o cumprimento dos prazos de ambos os tipos de tarefas. Por esse motivo, foram também propostos neste capítulo dois esquemas para transformar os instantes rígidos de início de tarefas periódicas em janelas de escalonamento. Desse modo, o escalonador dinâmico passa a ter maior flexibilidade para atender às tarefas aperiódicas. Dada uma janela de escalonamento, o instante de início de uma tarefa periódica pode ser retardado quando conveniente para atender a uma tarefa aperiódica. A duração do retardo que dinâmicamente pode ser imposto a uma tarefa periódica é dada por uma janela de execução (ou escalonamento). Se a janela for respeitada, o retardo de uma tarefa periódica não comprometerá o cumprimento do seu prazo, e nem das demais tarefas periódicas.

O algoritmo de escalonamento de tarefas periódicas, discutido neste capítulo, considera que blocos de escalonamento são sempre não preemptíveis. Essa restrição nem sempre se aplica a todas as tarefas de uma dada aplicação, do ponto de vista de suas características lógicas. Além disso, ao não considerar a possibilidade de fazer preempções de blocos de escalonamento logicamente preemptíveis, reduz-se a probabilidade de encontrar um escalonamento factível para todo o conjunto de tarefas. Nesse sentido, novos estudos e experimentos devem ser conduzidos para permitir avaliar os eventuais ganhos decorrentes do tratamento de tarefas preemptíveis, bem como o impacto dessa extensão sobre a complexidade do algoritmo e o custo de produção de escalonamentos.

Os conceitos módulo, servidor e acessos multiserviço, são considerados durante a construção de grafos de comunicação e na verificação da validade de mapeamentos gerados durante o processo de alocação e escalonamento, mas não no mapeamento propriamente dito, nem para verificar a factibilidade de um escalonamento completo a partir de um dado escalonamento parcial ("look ahead"). O uso de heurísticas que levem em conta os conceitos de servidor, módulo e acesso multiserviço, também nessas etapas, deve contribuir para que um número ainda maior de pontos de busca seja descartado mais cedo e, em conseqüência, seja reduzido o tempo de busca e de recursos necessários para determinar um escalonamento, ou a impossibilidade de encontrá-lo.

O algoritmo de escalonamento de tarefas com restrições críticas de tempo, incluindo as estratégias de balanceamento de carga, foi avaliado experimentalmente. Os resultados dos experimentos conduzidos são o assunto do capítulo 5.

Os conceitos módulo, servidor e acessos multiserviço, são considerados durante a construção de grafos de comunicação e na verificação da validade de mapeamentos gerados durante o processo de alocação e escalonamento, mas não no mapeamento propriamente dito, nem para verificar a factibilidade de um escalonamento completo a partir de um dado escalonamento parcial ("look ahead"). O uso de heurísticas que levem em conta os conceitos de servidor, módulo e acesso multiserviço, também nessas etapas, deve contribuir para que um número ainda maior de pontos de busca seja descartado mais cedo e, em conseqüência, seja reduzido o tempo de busca e de recursos necessários para determinar um escalonamento, ou a impossibilidade de encontrá-lo.

O algoritmo de escalonamento de tarefas com restrições críticas de tempo, incluindo as estratégias de balanceamento de carga, foi avaliado experimentalmente. Os resultados dos experimentos conduzidos são o assunto do capítulo 5.

5. AVALIAÇÃO EXPERIMENTAL DO ALGORITMO DE ALOCAÇÃO E ESCALONAMENTO

O algoritmo de escalonamento "off-line" de tarefas periódicas proposto no capítulo 4 foi implementado na linguagem C++, sob ambiente UNIX, conforme discutido na seção 4.5. Essa implementação foi avaliada através de simulações, empregando casos de teste que procuram retratar conjuntos de tarefas (programas) com características diversificadas. O processo empregado para gerar os casos de teste e a discussão dos resultados das simulações, visando avaliar a eficácia do algoritmo de escalonamento proposto, são precisamente os objetivos deste capítulo.

Conforme discutido anteriormente, a determinação de um escalonamento factível para um conjunto de tarefas periódicas, com as características resultantes do modelo de programação do STER, é um problema computacionalmente intratável. Em consequência, quando um determinado algoritmo heurístico não consegue encontrar um escalonamento para um dado conjunto de tarefas, não significa necessariamente que as tarefas não sejam escalonáveis. Por esse motivo, a eficácia de um algoritmo heurístico de escalonamento costuma ser avaliada comparando-o com outros algoritmos. Pode-se, então, comparar os algoritmos a partir das suas respectivas taxas de sucesso em encontrar escalonamentos factíveis aliadas aos custos de produção desses escalonamentos.

Na avaliação do escalonador, serão comparados os resultados obtidos com a aplicação de três heurísticas para decidir se módulos cooperantes devem ou não ser alocados a uma mesma estação: 1) a heurística proposta no capítulo 4 para o HSTER, aqui chamada de heurística cliente-servidor, 2) a heurística empregada no algoritmo original de Ramamritham (1990), aqui chamada heurística nó-nó, e 3) heurística aleatória (decisão arbitrária). Para cada heurística serão feitos testes para avaliar a sua eficácia quando não se permite ao algoritmo a realização de "backtracks" ao chegar a um escalonamento parcial não factível, e quando, nas mesmas circunstâncias, permite-se um número limitado de "backtracks" (100).

Além de comparar o desempenho relativo das heurísticas cliente-servidor (C-S), nó-nó e aleatória, com e sem "backtracks", serão buscadas evidências de que os insucessos na busca por escalonamentos factíveis, usando a heurística C-S, devem-se à não escalonabilidade dos conjuntos de tarefas sendo considerados, e não a limitações inerentes ao algoritmo.

O desempenho do algoritmo é avaliado em termos do percentual dos casos de teste que foi possível alocar e escalonar (taxa de sucesso) versus os correspondentes custos, expressos em termos do número de pontos de busca (v. 4.2.2) criados durante o processo de alocação e escalonamento.

O capítulo tem a seguinte organização: na seção 5.1 descreve-se o processo de geração de casos de testes; na seção 5.2 são apresentados e discutidos os experimentos realizados para avaliar o algoritmo sem a utilização das estratégias de balanceamento de carga; na seção 5.3, apresenta-se e discute-se os experimentos conduzidos com o algoritmo

implementando as estratégias de balanceamento de carga; e, finalmente, na seção 5.4 faz-se um resumo do capítulo e alguns comentários finais.

5.1 Geração de casos de teste

Cada ponto dos gráficos que serão apresentados nas seções 5.2 e 5.3 foi produzido executando o algoritmo de alocação e escalonamento para 100 diferentes casos de testes. Cada caso de teste consiste de um conjunto de tarefas periódicas, representando, no modelo de programação do HSTER, módulos periódicos de um mesmo programa. Os conjuntos de tarefas foram produzidos em dois passos. No primeiro, utiliza-se um programa que faz a geração aleatória de grafos, denominado Randog [Molesky (1989)], para produzir grafos de escalonamento que representam conjuntos de tarefas que não contém servidores e, consequentemente, acessos a servidores. No segundo passo utiliza-se um filtro que introduz servidores e acessos a servidores nos grafos.

5.1.1 Geração de casos de testes sem servidores

O programa Randog pode ser usado para a geração de grafos genéricos, árvores e cadeias. Para os experimentos descritos neste capítulo foram gerados conjuntos de tarefas periódicas independentes, compostas de subtarefas com relações de precedência que podem ser representadas como grafos genéricos com as seguintes características:

- Em cada caso de teste, as subtarefas (nós do grafo de escalonamento) terão um tempo de execução uniformemente distribuído entre 50 (t_exec_min) e 100 (t_exec_max) unidades de tempo.

- Os custos de comunicação, cc, de cada par de subtarefas com relações de precedência foram calculados a partir da seguinte relação:

  (50 * taxa_de_comunicação) <= cc <= (100 * taxa_de_comunicação). (1)

  Tendo sido realizados experimentos com taxa_de_comunicação de 0,1 e 0,4.

- Com probabilidade taxa_de_redundância, subtarefas podem ter o número de cópias redundantes indicado pelo parâmetro n_réplicas, visando a implementação de mecanismos de tolerância a falhas. Nos experimentos realizados, taxa_de_redundância é igual a 0,1 e n_réplicas é igual a 1.

- Foi definido um parâmetro denominado fator_de_folga, variando de 0,9 a 1,6, a partir do qual o período, P, da tarefa de menor período de cada caso de teste é calculado usando a seguinte relação:

  P = T * tamanho_da_tarefa * (1 + (taxa_de_redundância * n_réplicas)) * fator_de_folga (2)

  onde, T é o tempo médio de execução de uma tarefa, ou seja, 75, e tamanho_da_tarefa o número de subtarefas da tarefa sendo gerada. A relação acima implica que, dados os requisitos computacionais de uma tarefa, quanto maior for o fator_de_folga, maior será o período da tarefa e, conseqüentemente, mais fácil será escaloná-la.

Os resultados apresentados neste capítulo foram obtidos para casos de testes com três tarefas periódicas. A primeira composta de uma subtarefa (tamanho_da_tarefa = 1), a segunda de duas subtarefas e a terceira de três subtarefas. O período da primeira tarefa periódica será igual a P, calculado usando a relação (2), o período da segunda tarefa será de 2P e o período da terceira de 3P. Dessa forma, para cada caso de testes, será buscado um escalonamento de tamanho igual ao mínimo múltiplo comum dos períodos das três tarefas (intervalo de escalonamento), isto é, 6P, ou seja, o grafo de escalonamento representando cada caso de testes terá 20 nós, 18 representando as subtarefas e outros dois representando o nó inicial e o nó final, respectivamente.

5.1.2 Adição de servidores aos casos de teste

A cada grafo gerado por Randog, com as características descritas no item 5.1.1, são adicionados acessos a servidores mediante a utilização de um filtro, isto é, de um programa que usa como entrada a saída de Randog e gera o conjunto de tarefas periódicas (caso de teste) que será efetivamente submetido ao algoritmo de alocação e escalonamento.

Foram feitos testes com dois tipos de acessos a servidores: Acessos simples (ASS) e acessos multiserviço (AMS). Um ASS ocorre quando uma subtarefa cliente requisita ao servidor a execução de um único serviço, e um AMS ocorre quando uma subtarefa cliente requisisita a um servidor a execução de vários serviços em exclusão mútua. Nos experimentos descritos neste capítulo, um AMS envolve sempre a requisição de dois serviços.

Adição de acessos simples (ASS)

A inclusão de um acesso simples a um servidor é feita transformando um nó do grafo gerado por Randog, tx, em três novos nós, conforme ilustrado pela figura 5.1. O primeiro dos três nós gerados, tx1, corresponde ao segmento (de escalonamento) da tarefa cliente ao final do qual é requisitado um serviço, o segundo, sx, ao segmento do servidor responsável por oferecer o serviço pedido e, finalmente, o terceiro, tx2, ao segmento da tarefa cliente que se inicia com a chegada da resposta enviada pelo servidor.

Para que o grafo com ASSs mantenha aproximadamente a mesma folga do grafo original (prazo do último nó menos a soma dos custos de todos os nós), o custo de cada um dos nós gerados para representar um ASS, c_nó_ASS, será calculado usando a seguinte relação:

$$\lceil (t\_exec\_min / 3) \leq \quad c\_nó\_ASS \leq \lceil (t\_exec\_max / 3 \quad (3)) \rceil$$

onde , $\lceil (x)$ representa o menor inteiro maior ou igual a x.

Considerando que nos experimentos descritos t_exec_min é igual a 50 unidades de tempo e que t_exec_max é igual a 100 unidades de tempo, os custos de tx1, sx e tx2 irão aleatoriamente variar entre 17 a 33 unidades de tempo.

Os arcos conectando os nós clientes ao nó servidor terão custos (representando o custo de comunicação entre os correspondentes blocos de escalonamento) c_com_cs_ASS, determinados pela seguinte relação:

$$(4)\ c\_com\_min\_cs\_ASS \leq c\_com\_cs\_ASS \leq c\_com\_max\_cs\_ASS,$$

onde,

$$c\_com\_min\_cs\_ASS = \lceil (taxa\_de\_comunicação*(c\_com\_min/3)),$$

$$c\_com\_max\_cs\_ASS = \lceil (taxa\_de\_comunicação*(c\_com\_max/3)),$$

$$c\_com\_mim = (taxa\_de\_comunicação*50), \text{ e}$$

$$c\_com\_max = (taxa\_de\_comunicação*100).$$

Ou seja, se a taxa_de_comunicação (TC) for igual a 0,1, o custo de comunicação entre nós clientes e o nó servidor, c_com_cs_ASS, será de 2 ou 3 unidades de tempo, e se TC for igual a 0,4, c_com_cs_ASS irá variar de 7 a 14 unidades de tempo.

O percentual de nós do grafo originalmente gerado por Randog a transformar em acessos simples a servidores é definido pelo parâmetro "taxa de acessos simples a servidores" (TASS). Dado TASS, gera-se o grafo com os acessos a servidores percorrendo o grafo originalmente produzido por Randog e transformando, com probabilidade TASS, nós desse grafo nos três nós que representarão um ASS no novo grafo.

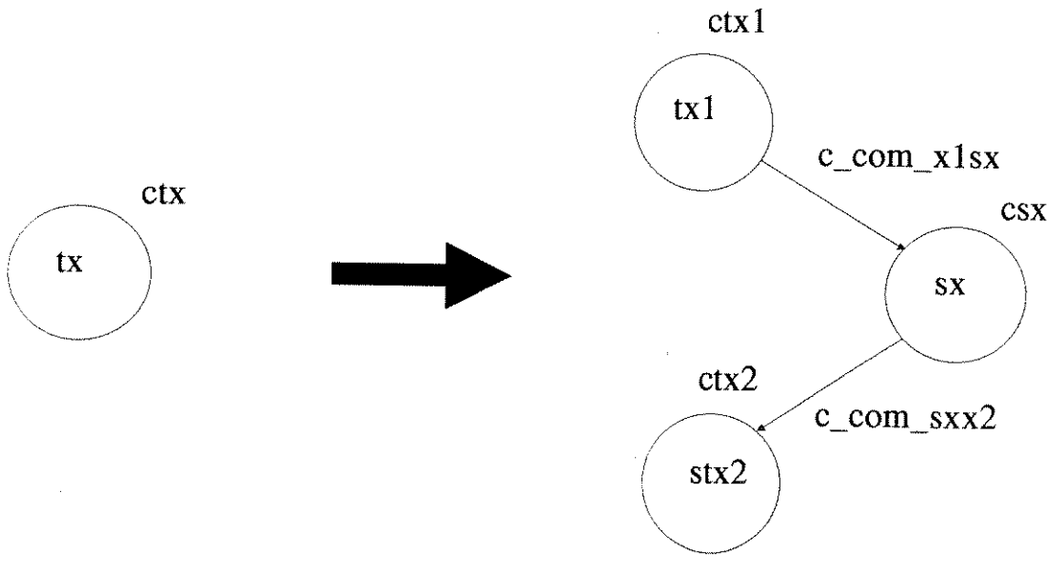


Figura 5.1: Geração de um acesso simples a um servidor

A inclusão de ASS mantém a folga dos grafos constante (o custo computacional dos nós do grafo permanece constante), relativamente aos grafos sem servidores que lhes

deram origem, ela contribui, porém, para o aumento dos custos de comunicação entre os nós (aumentado o número de arcos). Cada ASS acrescenta dois novos arcos ao grafo, cada um desses com um custo aproximado de 33% do custo de um arco do grafo original, conforme determinado pela relação (4). Como a fração de nós do grafo original convertidos em ASS é dado por TASS, a inclusão de ASSs conduz a um aumento aproximado da carga de comunicação do grafo original de TASS*0,66. Por exemplo, se TASS for igual a 0,1, o grafo com ASSs terá uma demanda de comunicação cerca de 6,6% maior do que o grafo que lhe deu origem.

O aumento do número de arcos nos grafos com ASS, com o consequente aumento dos custos de comunicação, leva a um previsível aumento da dificuldade de escalonar os novos grafos, relativamente aos grafos em ASS, por dois motivos: 1) o aumento dos custos de comunicação retarda o início dos nós que sucedem ao nó original que foi convertido num ASS e 2) os acessos a um servidor compartilhado irão aumentar a contenção entre os nós clientes.

Adição de acessos multiserviço (AMS)

Nos experimentos descritos neste capítulo, acessos multiserviço envolvem sempre a requisição de dois serviços. A inclusão de acessos multiserviço, ilustrada pela figura 5.2, é feita usando um procedimento análogo ao empregado para a inclusão de acessos simples. Conforme pode ser observado na figura 5.2, na transformação de um nó do grafo original num AMS são gerados cinco novos nós: um nó representando o segmento do cliente que antecede à requisição do primeiro serviço, tx1, um nó representando o segmento do servidor encarregado de oferecer o primeiro serviço pedido, sx, um nó representando o segmento do cliente que recebe a notificação de que o primeiro serviço foi atendido e solicita o segundo serviço, tx2, um nó representando o segmento do servidor encarregado de prestar o segundo serviço pedido, sy, e, finalmente, um nó representando o segmento do cliente executado após o atendimento do segundo serviço, tx3.

O custo associado a cada nó envolvido num AMS, c_nó_AMS, é calculado pela seguinte relação:

$$(5)\ \lceil (t\_exec\_min/5) \leq c\_nó\_AMS \leq \lceil (t\_exec\_max/5).$$

Aos arcos que conectam os nós gerados é associado um custo c_com_cs_AMS, definido pela seguinte relação:

$$(6)\ c\_com\_min\_cs\_AMS \leq c\_com\_cs\_AMS \leq c\_com\_max\_cs\_AMS,$$

onde,

$$c\_com\_min\_cs\_AMS = \lceil (taxa\_de\_comunicação*(c\_com\_min/5)) \text{ e}$$

$$c\_com\_max\_cs\_AMS = \lceil (taxa\_de\_comunicação*(c\_com\_max/5)).$$

O percentual de nós do grafo originalmente gerado por Randog a transformar acessos multiserviço é definido pelo parâmetro "taxa de acessos multiserviço" (TAMS). Dado TAMS, o grafo com os acessos a servidores é gerado percorrendo o grafo original e transformando, com probabilidade TAMS, seus nós em novos nós represntando AMS.

Da mesma forma que quando da inclusão de ASS, a inclusão de AMS a um grafo gerado por Randog mantém a folga do grafo constante e contribui para o aumento dos custos de comunicação entre os nós. Cada AMS acrescenta quatro novos arcos ao grafo, cada um com um custo aproximado de 20% do custo de um arco do grafo original, conforme determinado pela relação (6). Como a fração de nós do grafo original convertidos em ASS é dado por TASS, a inclusão de AMSs aumenta a carga de comunicação do grafo original em aproximadamente TASS*0,80. Por exemplo, se TAMS for igual a 0,1, o grafo com ASSs terá uma demanda de comunicação cerca de 8% maior que o grafo que lhe deu origem.

A inserção de AMS a um grafo gerado por Randog aumenta o número de arcos do grafo e, conseqüentemente, os custos de comunicação, sendo, por isso, previsível o aumento da dificuldade de escalonar o novo grafo, relativamente ao grafo original e a grafos de igual folga com ASSs.

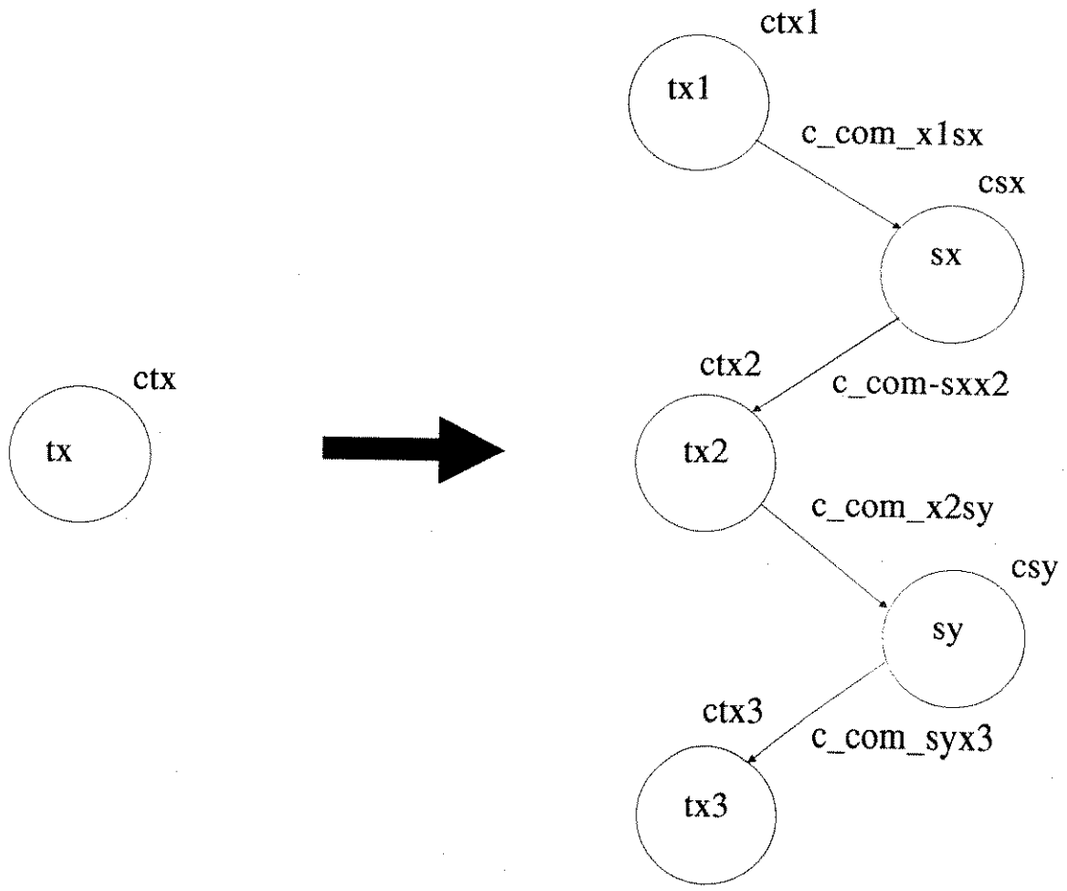


Figura 5.2: Geração de um acesso multiseviço a um servidor

5.2 Desempenho do algoritmo sem o uso das estratégias de balanceamento de carga

Nesta seção são discutidos experimentos realizados para avaliar o desempenho do algoritmo de alocação e escalonamento de tarefas periódicas sem o emprego das estratégias para balanceamento de carga. Convém relembrar que cada ponto dos gráficos apresentados é o resultado da execução do algoritmo para 100 casos de teste, produzidos conforme descrito nos itens 5.1.1 e 5.1.2.

Foram conduzidos experimentos para avaliar o comportamento do algoritmo com conjuntos de tarefas (casos de teste) com taxas de comunicação de 0,1 e 0,4, com 0, 1 e 2 servidores e TASS e TAMS de 0 e 0,1. Usando os procedimentos descritos nos itens 5.1.1 e 5.1.2, são produzidos grafos de escalonamento com 20 nós, para TASS ou TAMS iguais a zero, grafos com 24 nós, em média, para TASS de 0,1, e grafos com 28 nós para TAMS de 0,1.

Foi também conduzido um experimento para determinar a influência do número máximo de "backtracks" que se permite ao algoritmo realizar sobre a taxa de sucesso e custos de escalonamento.

5.2.1 Acessos simples a um servidor

O gráfico 5.1 mostra a taxa de sucesso na determinação de um escalonamento factível de conjuntos de testes onde 10% das subtarefas fazem acessos simples a um servidor (S=1; TASS = 0,1) e a taxa de comunicação (TC) é igual a 0,1. O gráfico apresenta resultados obtidos quando a construção do grafo de comunicação é feita considerando pares de nós comunicantes (heurística nó-nó), quando são considerados todos os nós envolvidos em interações cliente-servidor (heurística C-S) e quando as decisões de alocação de clientes e servidores são tomadas aleatoriamente.

O gráfico mostra claramente que as decisões de alocação tomadas considerando os custos de comunicação e execução das subtarefas (heurísticas nó-nó e C-S) são sensivelmente melhores do que aquelas feitas de modo arbitrário (heurística aleatória).

Quando não se faz "backtracks", pode-se notar que a heurística cliente-servidor tem, na maioria das vezes, um melhor desempenho do que a heurística nó-nó. Em alguns casos, por exemplo, para uma folga igual 1,1, a taxa de sucesso da heurística C-S excede em mais de 20% a taxa de sucesso da heurística nó-nó.

O uso de um número limitado de "backtracks", juntamente com as heurísticas nó-nó e cliente-servidor permite aumentar ainda mais a taxa de sucesso do algoritmo, em alguns casos, por exemplo, para uma folga de 1,2, o aumento é novamente superior a 20%. Por outro lado, o uso de "backtracks" torna insignificante a diferença, em termos de taxa de sucesso, das heurísticas nó-nó e C-S.

Com relação à heurística aleatória, o uso de "backtracks" leva a um aumento substancial (90 % em alguns casos) da taxa de sucesso em relação aos resultados obtidos com a mesma heurística sem o uso de "backtracks", o que evidencia o grande número de

decisões erradas tomadas usando essa heurística. Entretanto, mesmo realizando "backtracks", o desempenho da heurística aleatória é bastante modesto se comparado ao das outras duas.

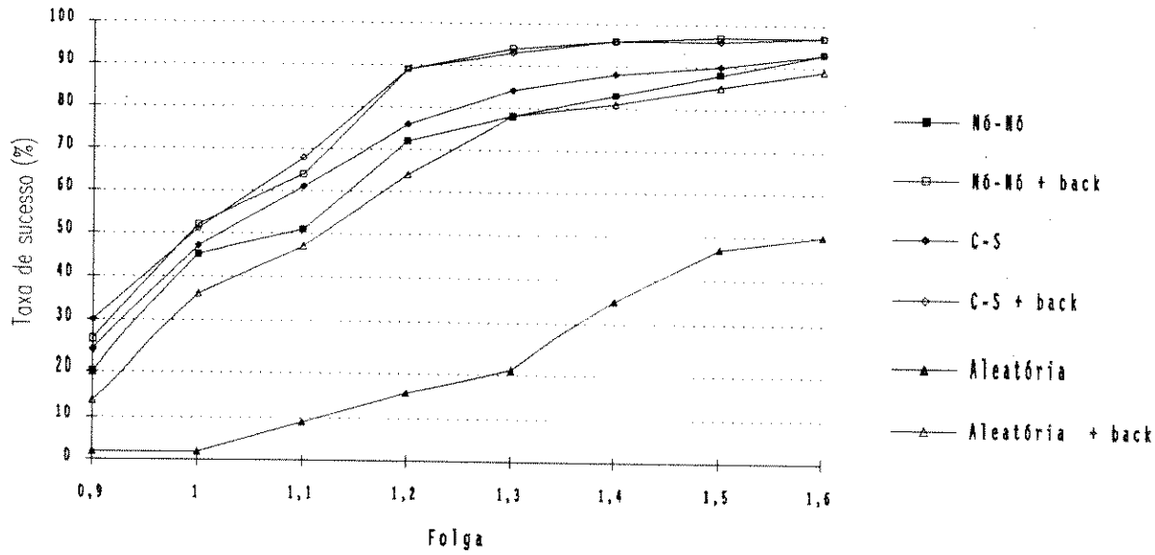


Gráfico 5.1: Taxa de sucesso para S=1; TASS=0,1 e TC=0,1.

O gráfico 5.2 mostra os custos associados à utilização de cada heurística, em número de pontos criados durante o processo de busca, nos casos de teste em que o algoritmo teve sucesso na determinação de escalonamentos factíveis. Pode-se observar que a heurística cliente-servidor tem um custo menor que a heurística nó-nó para todas as folgas. Para a maioria das folgas acontece, inclusive, de o custo da heurística cliente-servidor com "backtracks" ser equivalente ao custo da heurística nó-nó sem "backtracks. Os custos da decisão aleatória são equivalentes aos custos da heurística cliente-servidor, porém, conforme constatado no gráfico 5.1, a taxa de sucesso da heurística aleatória é sensívelmente menor do que a taxa obtida usando a heurística cliente-servidor.

O gráfico 5.3 mostra o custo das heurísticas nos casos em que o algoritmo não teve sucesso na busca por um escalonamento factível. Quando não se realizam "backtracks", a heurística cliente-servidor apresenta um custo ligeiramente inferior ao custo da heurística nó-nó, indicando que, com ela, o algoritmo consegue determinar mais cedo que não poderá encontrar um escalonamento factível. Por outro lado, quando são permitidos "backtrack" a heurística cliente-servidor visita um número maior de pontos de busca (testa mais caminhos) até concluir que não conseguirá encontrar um escalonamento factível.

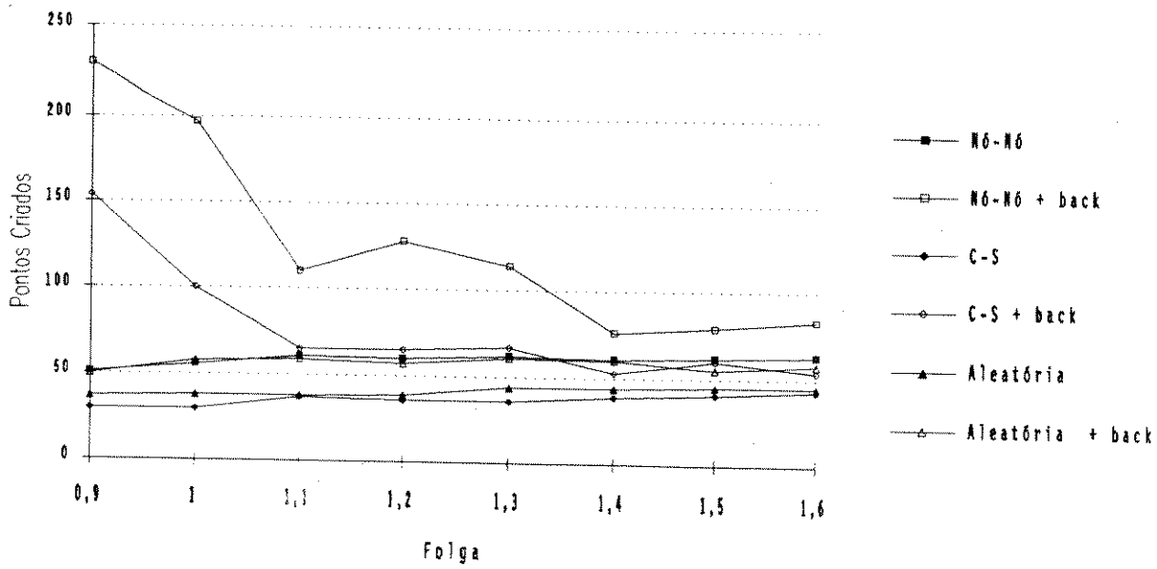


Gráfico 5.2: Custo das buscas bem sucedidas para S=1; TASS=0,1 e TC=0,1.

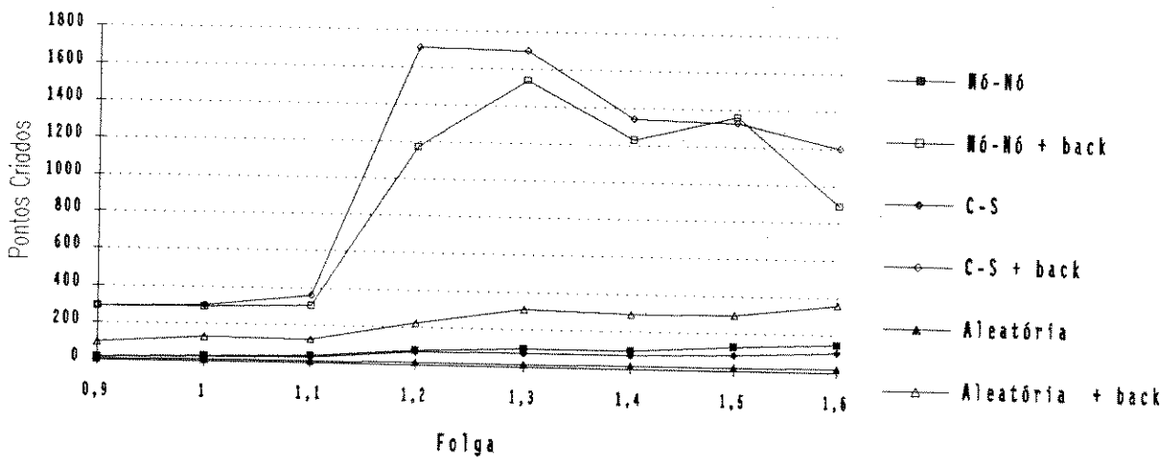


Gráfico 5.3: Custo das buscas mal sucedidas para S=1; TASS=0,1 e TC=0,1.

5.2.2 Acessos multiserviço a um servidor

Os gráficos 5.4, 5.5 e 5.6 mostram a taxa de sucesso, custo para buscas bem sucedidas e custos para buscas mal sucedidos, respectivamente, do algoritmo de alocação e escalonamento quando cada caso de teste, com TC de 0,1, inclui um servidor (S=1) e 10% das suas subtarefas faz um acesso multiserviço ao servidor (TAMS = 0,1).

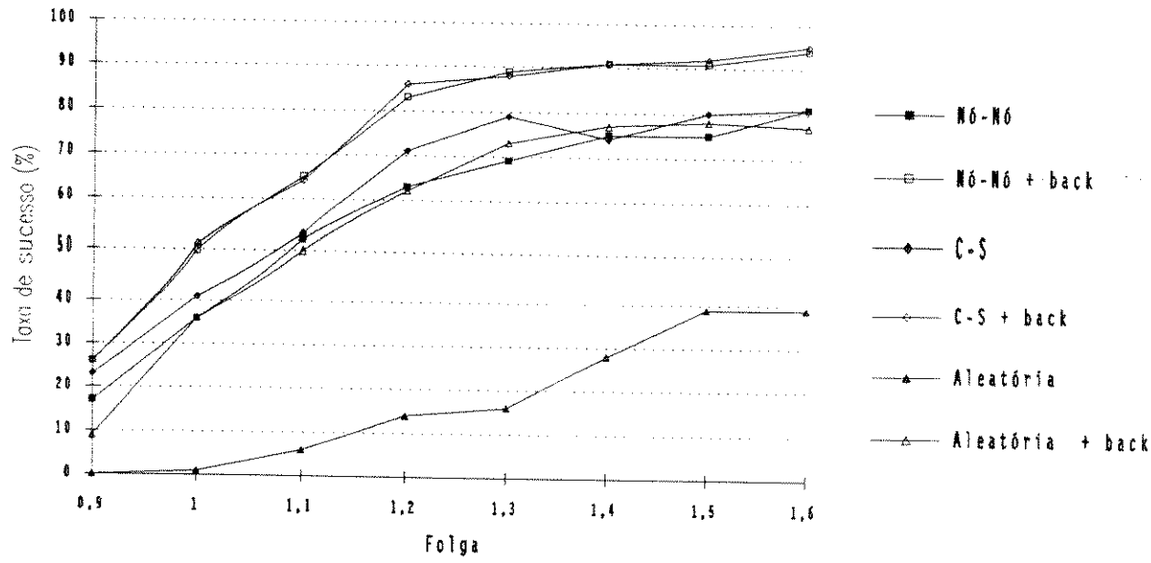


Gráfico 5.4: Taxa de sucesso para S=1; TASS=0; TAMS=0,1 e TC=0,1.

Pode-se verificar, comparando os gráficos 5.4 e 5.1, que a realização de acessos multiserviço acarreta a diminuição da taxa de sucesso, relativamente à observada com acessos simples. Embora os custos associados à execução das subtarefas sejam aproximadamente os mesmos em ambos os casos, a diminuição da taxa de sucesso pode ser explicado pelo aumento dos custos de comunicação entre os nós e pelo maior contenção experimentada em grafos com AMS, relativamente a um grafo com acessos simples, em função do processo de geração de ASS e AMS descrito em 5.1.2.

Na figura 5.5, pode-se observar que a heurística C-S usando, ou não, "backtracks", acarreta menores custos para efetuar (um maior percentual de) escalonamentos factíveis. Em especial, comparando as figuras 5.2 e 5.5, pode-se observar que com AMSs a diferença entre os custos da estratégias C-S e Nó-Nó com "backtracks" é maior ainda que a observada para ASS.

Para o custo associado às buscas mal sucedidas, a relação entre as estratégias é a mesma observada no experimento anterior. Aqui também se observa que a estratégia C-S visita, em geral, mais pontos até concluir pela impossibilidade de escalonar os conjuntos de testes.

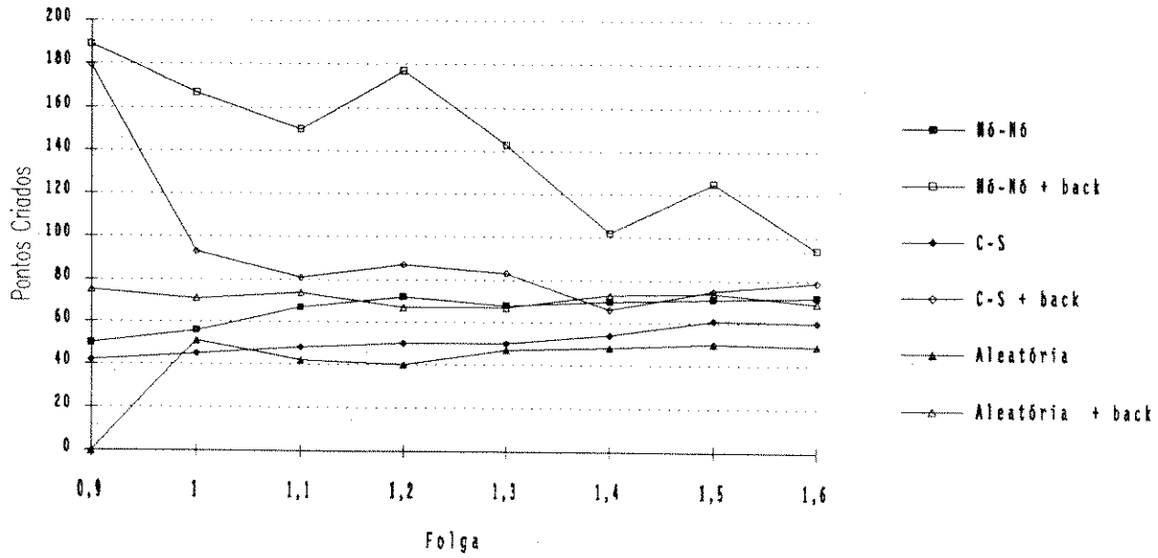


Gráfico 5.5: Custo das buscas bem sucedidas para S=1;TASS=0; TAMS=0,1 e TC=0,1.

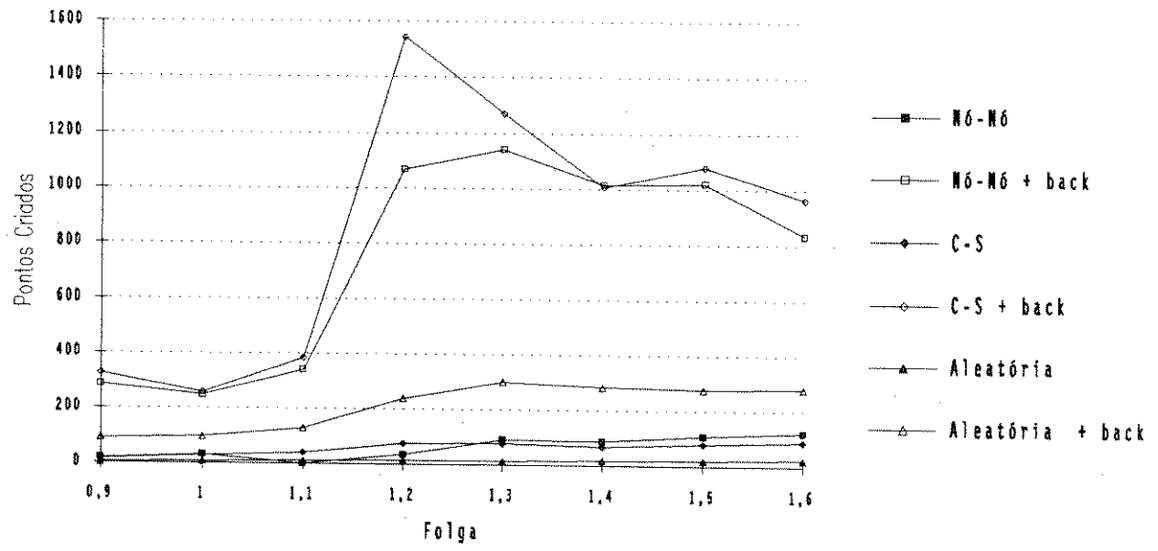


Gráfico 5.6: Custo das buscas mal sucedidas para S=1; TASS=0;TAMS=0,1 e TC=0,1.

5.2.3 Desempenho quando acessos simples são combinados com acessos multiserviço a um único servidor

Os gráficos 5.7, 5.8 e 5.9 mostram a taxa de sucesso, custo das buscas bem sucedidas e custo das buscas mal sucedidos, respectivamente, decorrentes do uso do algoritmo de alocação e escalonamento quando num mesmo grafo são combinados acessos simples e multiserviço a um único servidor. Para este experimento, 20% dos nós do grafo gerado por Randog são transformados em acessos a um servidor. Metade desses acessos corresponde a acessos simples (TASS=0,1) e metade a acessos multiserviço (TAMS=0,1).

A relação entre o custo e o benefício das heurísticas mantém-se aproximadamente a mesma verificada nos experimentos anteriores. Entretanto, agora, a taxa de sucesso diminui significativamente. Podem ser apontadas, novamente, como razões para essa redução o aumento do custo de comunicação global do grafo, e o aumento da contenção entre nós clientes para acessar o servidor. O aumento do custo de comunicação do grafo é decorrente da maior porcentagem de nós do grafo original que foram transformados em acessos a servidores (20% neste experimento e 10% nos anteriores). O aumento da contenção no acesso ao servidor também aumenta sensivelmente, por haver agora um maior número de nós clientes que compartilha o servidor. A maior contenção entre os clientes reduz, naturalmente, a possibilidade de atender a todos a tempo de que cada um possa cumprir o seu prazo.

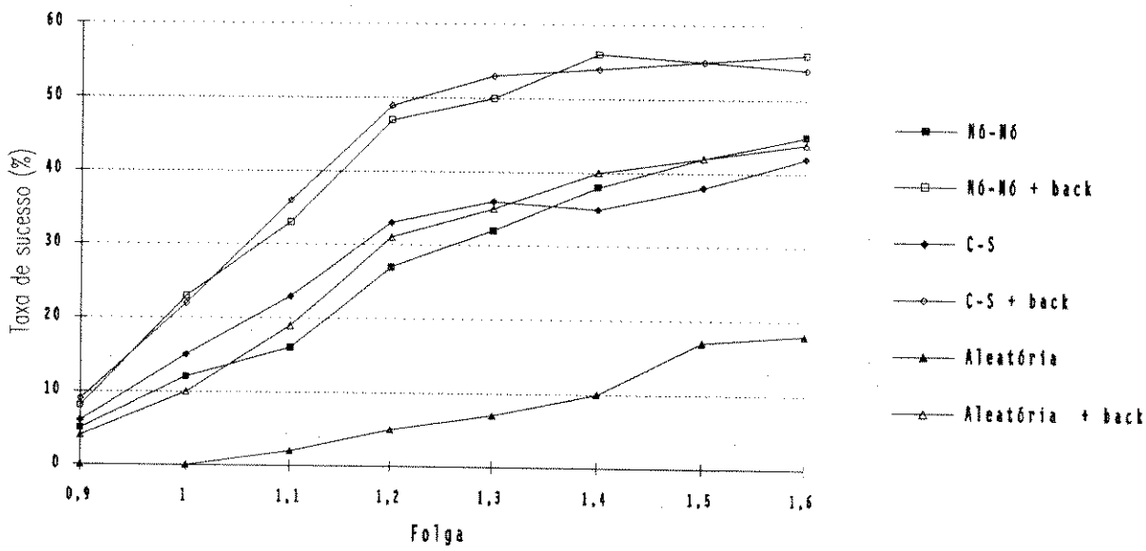


Gráfico 5.7: Taxa de sucesso para S=1; TASS=0,1; TAMS=0,1 e TC=0,1.

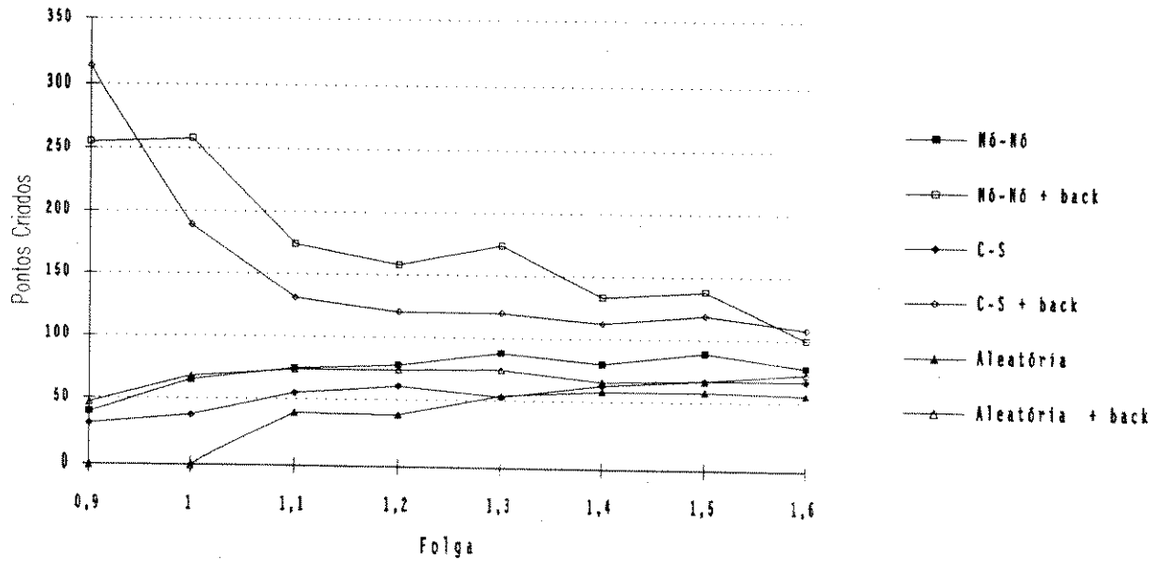


Gráfico 5.8: Custo das buscas bem sucedidas para S=1;TASS=0,1; TAMS=0,1 e TC=0,1.

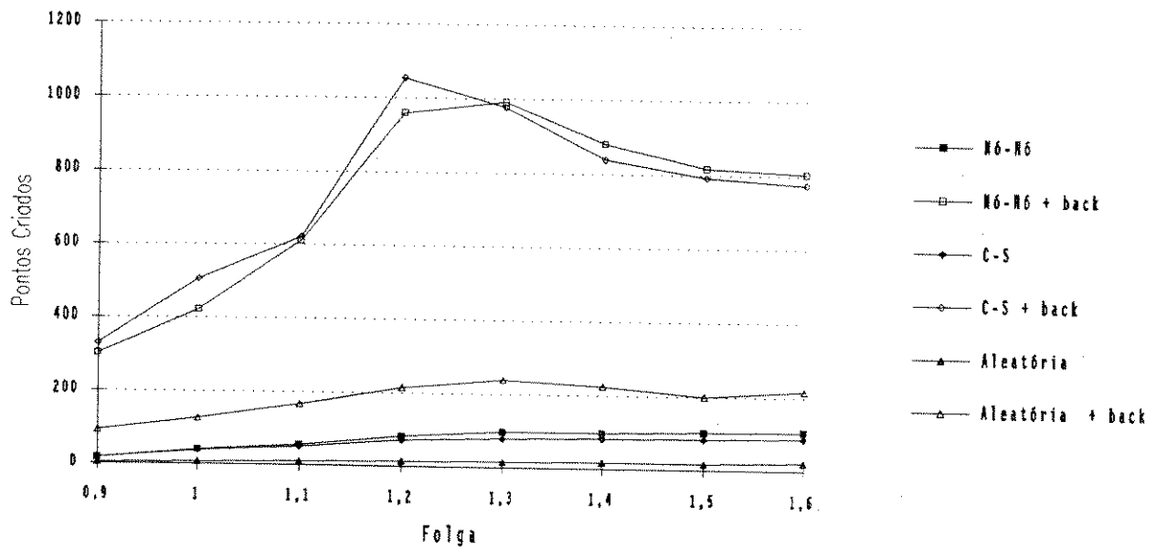


Gráfico 5.9: Custo das buscas mal sucedidas para S=1; TASS=0,1;TAMS=0,1 e TC=0,1.

5.2.4 Efeito do aumento da taxa de comunicação entre as subtarefas

Os gráficos 5.10, 5.11 e 5.12 mostram a taxa de sucesso, custo para buscas bem sucedidas e custo para buscas mal sucedidas, respectivamente, para conjuntos de tarefas com as mesmas características das tarefas usadas no experimento descrito no item anterior, salvo pela taxa de comunicação que aqui é de 40% (TC=0,4).

Nota-se aqui uma redução adicional da taxa de sucesso decorrente da maior dificuldade de escalonar subtarefas que, quando alocadas a estações diferentes, utilizam-se do meio de comunicação com mais intensidade, mantendo-o mais ocupado e retardando a entrega das mensagems de habilitação às subtarefas remotas. A esse propósito, convém lembrar que os custos de comunicação entre as subtarefas não são usadas quando do cálculo do período das tarefas. Os fatores dominantes nesse cálculo são o tempo de execução das subtarefas e o fator de folga (v. 5.1.1).

Com exceção da redução da taxa de sucesso, mantém-se as mesmas consideracões feitas para os experimentos anteriores relativamente ao desempenho das diversas heurísticas. Cabe ressaltar que aqui a superioridade da heurística cliente-servidor é particularmente acentuada, tanto em termos da taxa de sucesso como do custo. Observe-se que em alguns casos a taxa de sucesso do algoritmo usando a heurística cliente-servidor é 70% superior à taxa obtida com a heurística nó-nó, a um custo inferior.

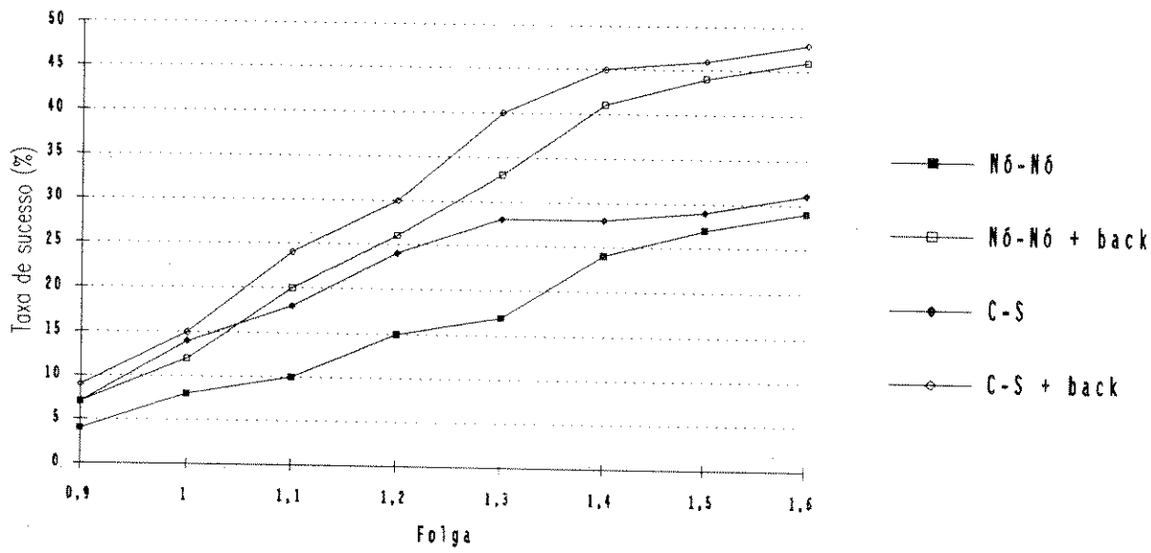


Gráfico 5.10: Taxa de sucesso para S=1; TASS=0,1; TAMS=0,1 e TC=0,4.

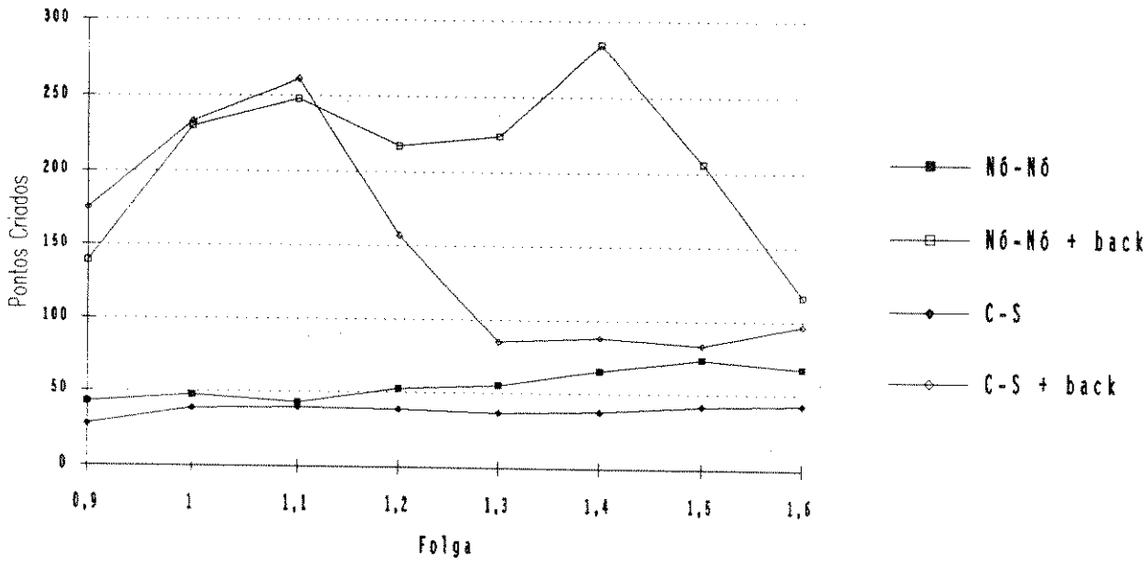


Gráfico 5.11: Custo das buscas bem sucedidas para S=1;TASS=0,1; TAMS=0,1 e TC=0,4.

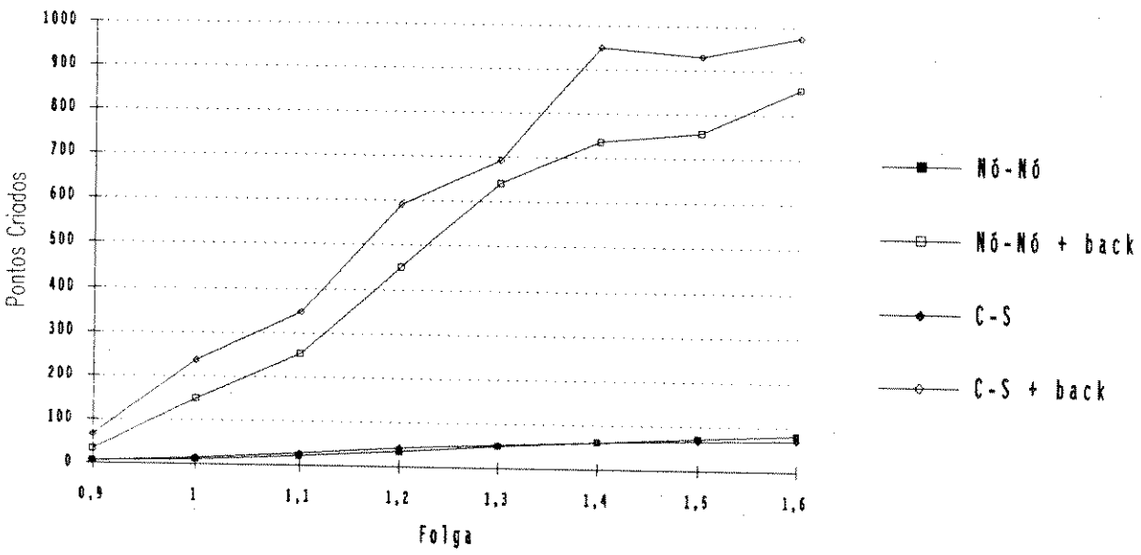


Gráfico 5.12: Custo das buscas mal sucedidas para S=1; TASS=0,1;TAMS=0,1 e TC=0,1.

5.2.5 Desempenho do algoritmo na presença de dois servidores

Neste experimento são usados conjuntos de testes com as características dos conjuntos usados no experimento anterior usando dois servidores, ao invés de um. Comparando o gráfico 5.13, que mostra a taxa de sucesso deste experimento, com o gráfico 5.10, que mostra a taxa de sucesso do experimento anterior, pode-se verificar que o principal efeito da inclusão de mais um servidor aos conjuntos de tarefas é um sensível aumento da taxa de sucesso do algoritmo. Este resultado é o esperado na medida em que a presença de um novo servidor reduz a contenção entre as tarefas clientes.

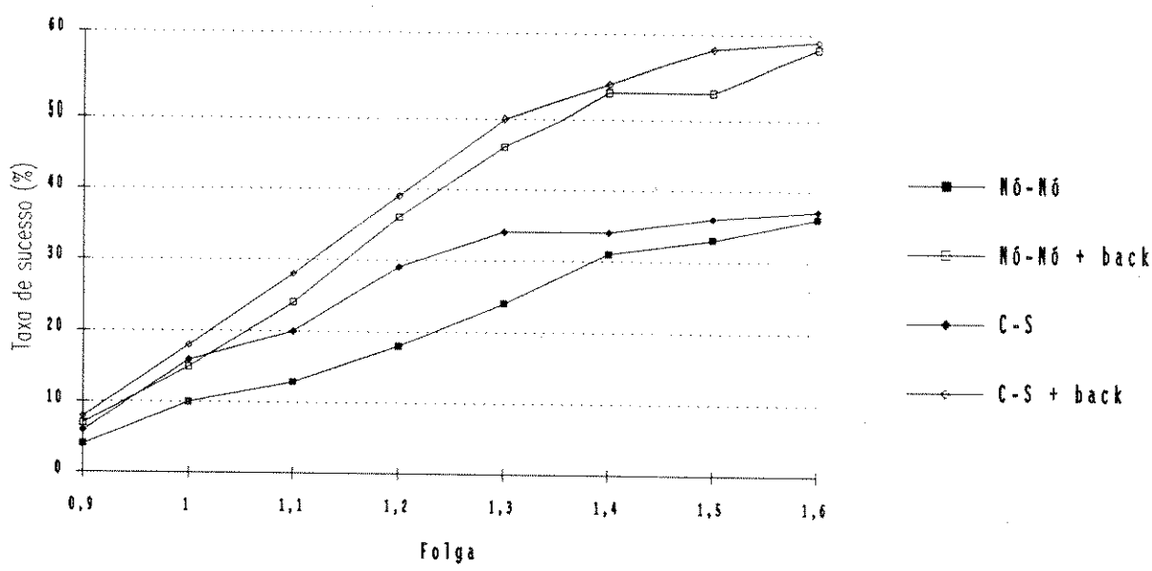


Gráfico 5.13: Taxa de sucesso para S=2; TASS=0,1; TAMS=0,1 e TC=0,4.

5.2.6 Efeito do aumento do número de "backtracks" sobre a taxa de sucesso

Nos experimentos anteriores verificou-se que o uso de um número limitado de "backtracks" (100) levou a um aumento razoável da taxa de sucesso do algoritmo. Neste ítem vamos procurar verificar os efeitos de aumentos adicionais no número de "backtracks" permitidos.

As figuras 5.14, 5.15 e 5.16 mostram a taxa de sucesso, custo das buscas bem sucedidas e custo das buscas mal sucedidas, respectivamente, para conjuntos de tarefas com um servidor, TASS de 0,1 e TC de 0,1, quando são permitidos 0, 100, 400 e 800 "backtracks".

Conforme pode ser observado no gráfico 5.14, quando a heurística C-S é usada, a realização de um pequeno número de "backtrack" possibilita um aumento sensível da taxa de sucesso, após o que, aumentos adicionais do número de "backtracks" permitidos resulta em aumentos apenas marginais da taxa de sucesso. Esses aumentos mínimos são, no entanto, obtidos a custos elevados, conforme pode ser observado nos gráficos 5.15 e 5.16.

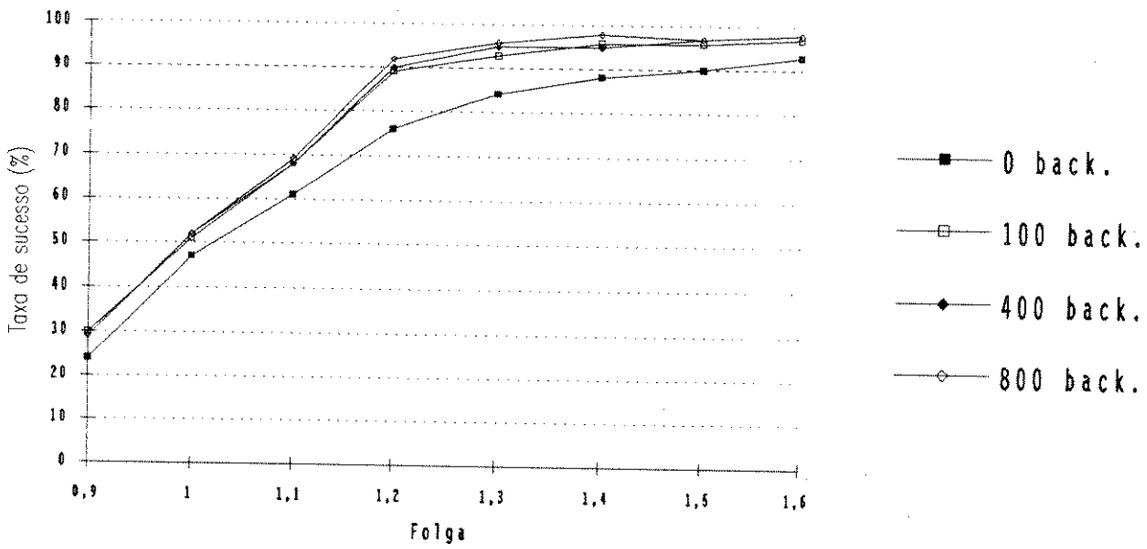


Gráfico 5.14: Taxa de sucesso para S=1; TASS=0,1; TAMS=0 e TC=0,1.

Estes resultados sugerem, por um lado, que é interessante, e possível, aprimorar as heurísticas empregadas para que o algoritmo possa atingir maiores taxas de sucesso sem o uso de "backtracks". Por outro lado, o fato de um pequeno número de "backtracks" ser suficiente para que se atinja o que parece ser o topo da taxa de sucesso, indica que na maioria das vezes as decisões tomadas pelo algoritmo, com base nas heurísticas apresentadas, são de boa qualidade.

O aumento da taxa de sucesso do algoritmo, quando são permitidos alguns "backtracks" nas etapas de mapeamento e escalonamento de nós de um dado grafo de comunicação, indica que as eventuais decisões incorretas devem estar concentradas nesse etapas e não na etapa de geração do grafo de comunicação, pois a realização de "backtracs" não leva à geração de um novo grafo de comunicação. Em outras palavras, aparentemente as decisões tomadas durante a geração do grafo de comunicação (alocar modulos e servidores a uma mesma estação ou não) foram mais precisas que as decisões tomadas durante o mapeamento e escalonameto dos nós desse grafo (quando, e a que estação, alocar segmentos dos módulos e dos servidores).

A permanência da taxa de sucesso praticamente estável à medida que aumentamos o número de "backtracks", sugere ainda que, se após um pequeno número de "backtracks" o algoritmo falhar em encontrar um escalonamento factível, o conjunto de tarefas sendo considerado provávelmente não seja escalonável.

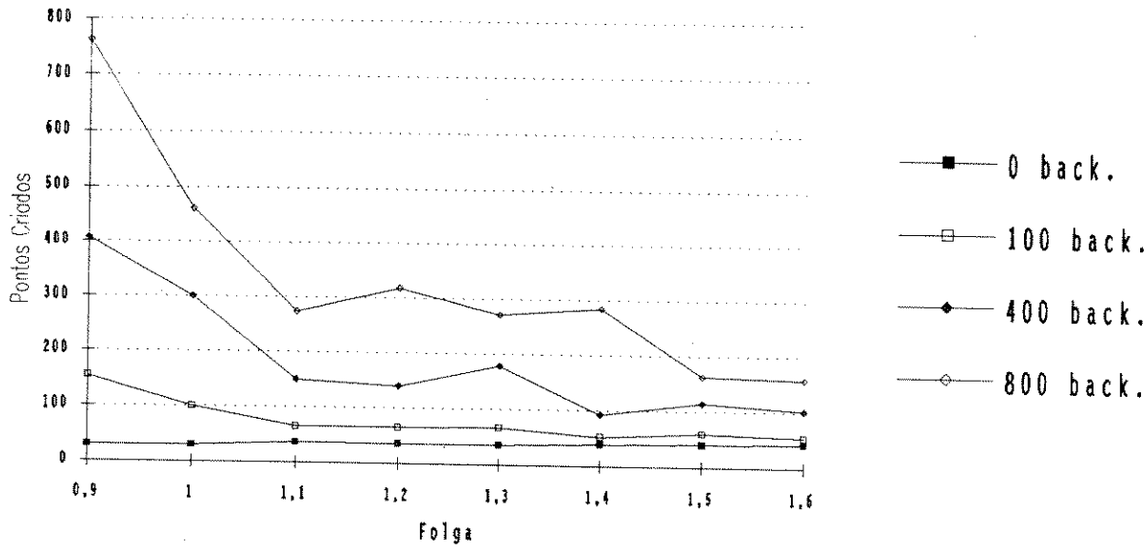


Gráfico 5.15: Custo das buscas bem sucedidas para S=1;TASS=0,1; TAMS=0 e TC=0,1.

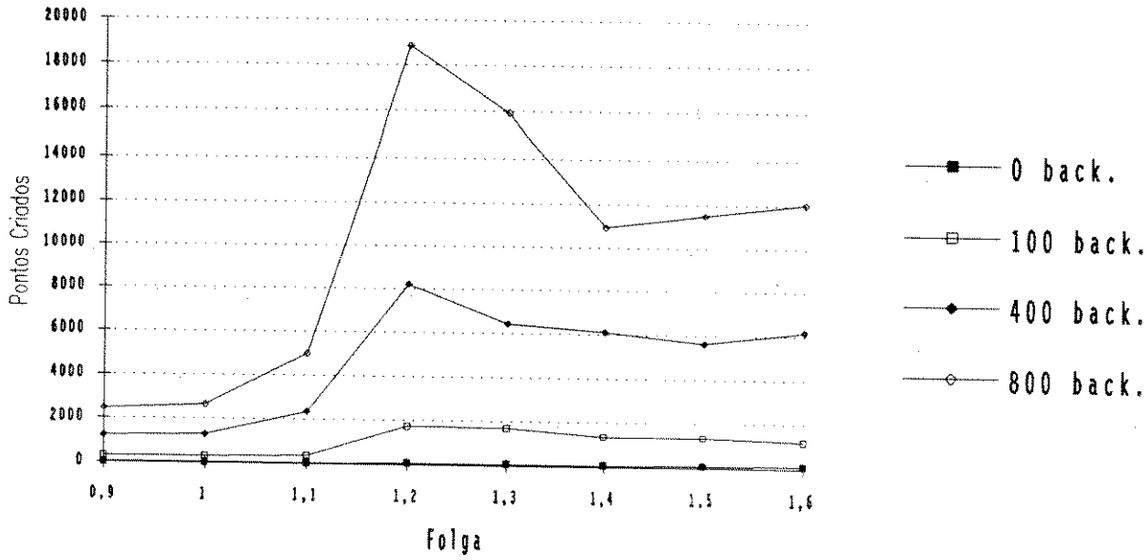


Gráfico 5.16: Custo das buscas mal sucedidas para S=1; TASS=0,1;TAMS=0 e TC=0,1.

5.3 Desempenho do algoritmo utilizando as estratégias para balanceamento de carga

Nesta seção será estudado o efeito da utilização das estratégias para balanceamento de carga ECM e EFO. Serão avaliados os efeitos do uso das estratégias sobre o balanço da carga computacional do sistema, assim como o custo associado à sua utilização. Em todos os experimentos discutidos, o algoritmo de alocação e escalonamento emprega a heurística C-S para a construção dos grafos de comunicação dos casos de testes.

Nos experimentos aqui descritos, o valor inicial do parâmetro CargaMáxima é dado por

$$\lceil (100/\text{número de estações}).$$

Como nos experimentos definiu-se que o sistema alvo tem seis estações, a relação acima indica que o algoritmo tentará, inicialmente, encontrar um escalonamento no qual cada estação não receba mais que 17% da carga total imposta ao sistema pelos casos de teste. Uma alocação obtida para esse valor de CargaMáxima seria perfeitamente balanceado. Caso não seja possível encontrar um escalonamento factível com esse valor, CargaMáxima é incrementada em 3% (i.e., IncCargaMáxima =3) a fim de que uma nova tentativa seja feita, e assim sucessivamente até que um escalonamento seja encontrado ou CargaMáxima chegue a 100%.

Na aplicação da estratégia EFO, o parâmetro FatorDeOciosidade é iniciado com 100, ou seja, após o termino de cada subtarefa, o algoritmo procura deixar o processador disponível por um intervalo de tempo de duração igual ao tempo de execução da subtarefa que acaba de ser executada. Se um escalonamento não puder ser obtido com esse valor, nova tentativa é feita usando um FatorDeOciosidade 10% menor que o anterior (i.e., DecFatorDeOciosidade = 10), e assim sucessivamente até que um escalonamento factível seja encontrado ou FatorDeOciosidade chegue a zero.

5.3.1 Desempenho das estratégias para balanceamento de carga

O gráfico 5.17 mostra o efeito das estratégias EFO e ECM sobre o balanço da carga do sistema. O balanço da carga é medido usando a métrica desvio de carga, definida em 4.3.1. Conforme pode ser observado no gráfico, o uso das estratégias ECM e EFO contribui sensivelmente para balancear a carga alocada ao sistema. No caso extremo, que ocorre para tarefas com fator de folga 1,6, o desvio de carga quando não se usa a ECM ou a EFO é cerca de 100% maior que o obtido com a estratégia ECM.

O gráfico 5.18 mostra o limite de carga médio imposto a cada estação quando o algoritmo é bem sucedido na busca por um escalonamento fáctivel e utiliza a estratégia ECM. Note-se que os valores de CargaMáxima mostrados no gráfico não correspondem à carga efetivamente alocada a cada estação, eles representam os valores médios do parâmetro CargaMáxima usados para limitar a carga alocada a cada estação, com os quais o algoritmo conseguiu alocar e escalonar os casos de testes.

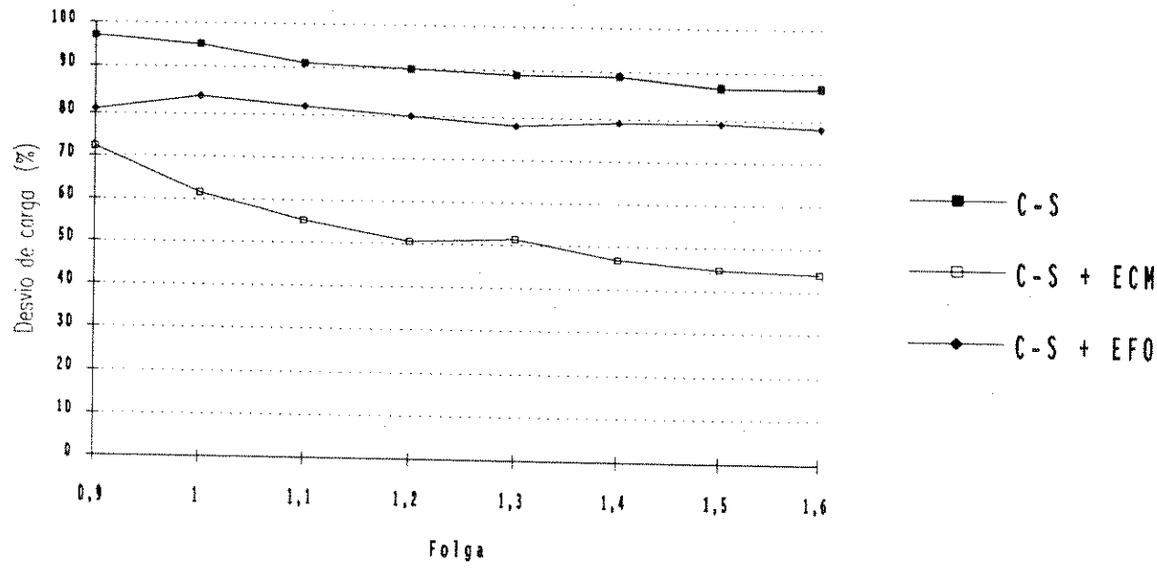


Gráfico 5.17: Efeito das estratégias ECM e EFO sobre o balanço de carga.

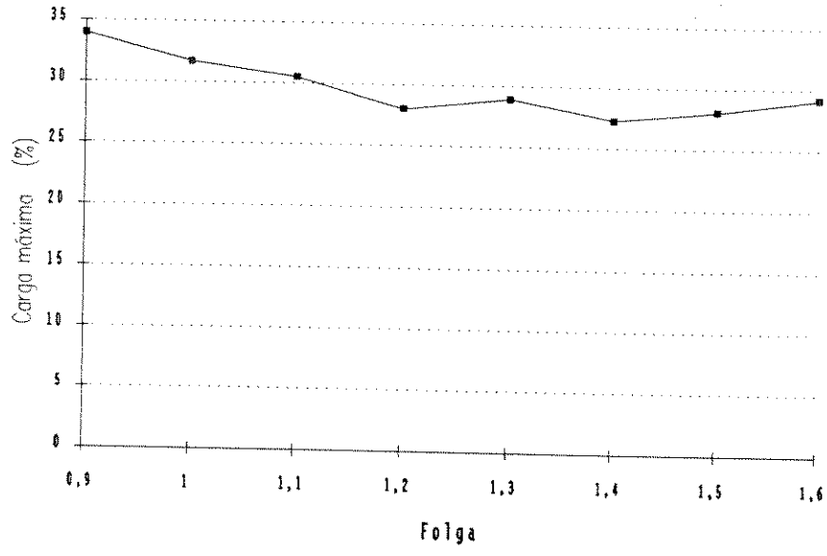


Gráfico 5.18: Limite de carga médio por estação usando a ECM.

O gráfico 5.19 mostra o fator de ociosidade médio usado pela estratégia EFO para as tarefas escalonadas com sucesso. Como seria de se esperar, à medida que o fator de folga aumenta, o fator de ociosidade aumenta também, pois, com folgas maiores, as tarefas podem ser melhor distribuídas ao longo do intervalo de escalonamento.

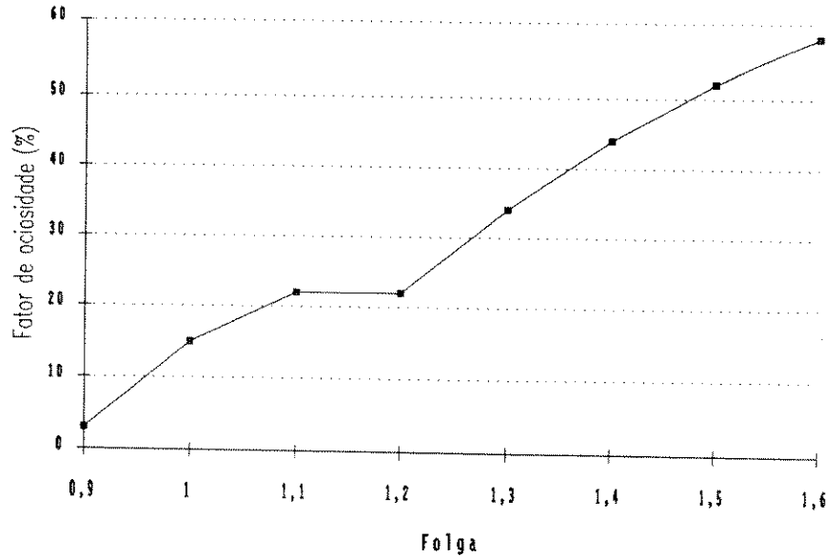


Gráfico 5.19: Fator médio de ociosidade obtido com a estratégia EFO.

Os gráficos 5.20 e 5.21 mostram o custo associado ao uso das estratégias de balanceamento de carga para buscas bem sucedidas e para buscas mal sucedidos, respectivamente. Pode-se observar que, do ponto de vista do balanceamento de carga (mostrado no gráfico 5.17), a estratégia ECM apresenta uma melhor relação custo/benefício. Isso se deve, em parte, ao fato de, independentemente da folga das tarefas, a estratégia EFO sempre tentar gerar um escalonamento factível começando a busca com um FatorDeFolga de 100%, valor evidentemente muito alto para tarefas com folgas pequenas.

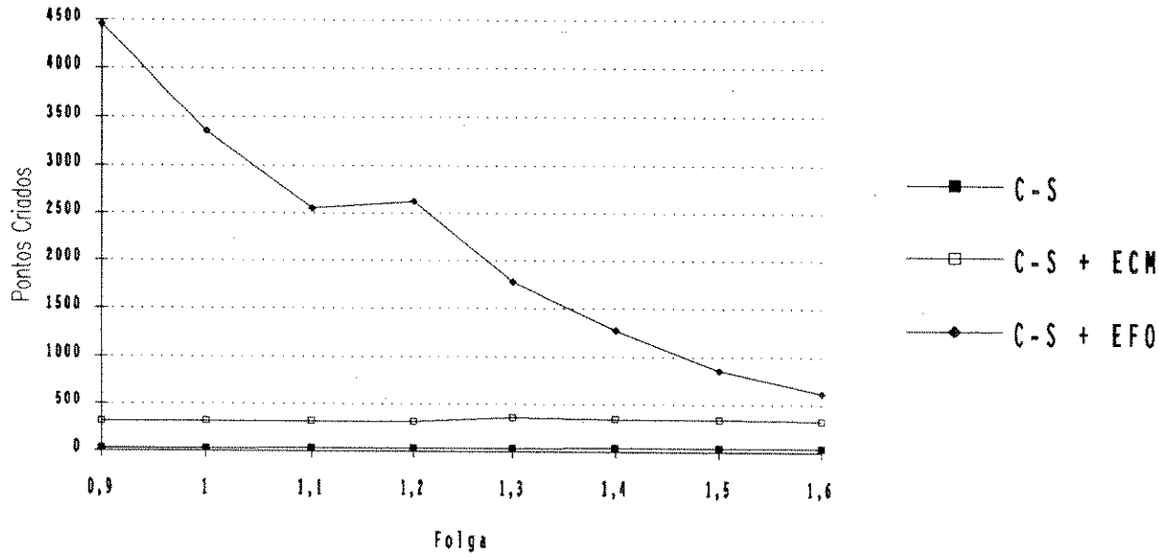


Gráfico 5.20: Custo das estratégias de balanceamento de carga para buscas bem sucedidas.

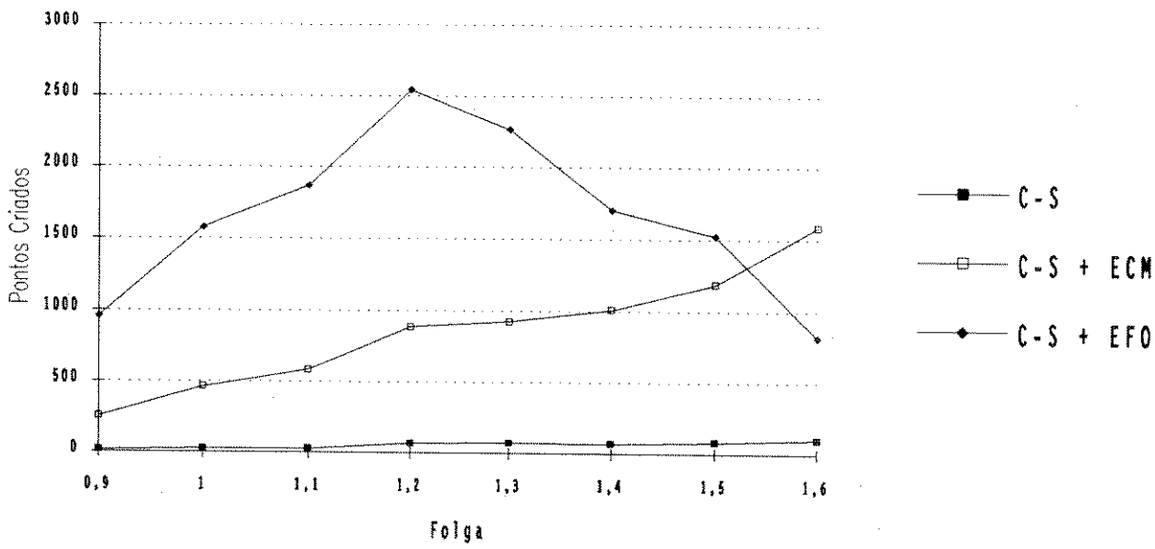


Gráfico 5.21: Custo das estratégias de balanceamento de carga para buscas mal sucedidas.

5.3.2 Efeito do Uso de Valores Adaptativos para CargaMax

Os resultados dos experimentos discutidos no item 5.3.1 sugerem claramente que os valores iniciais de FatorDeFolga e CargaMáxima têm grande influência sobre os custos das buscas por escalonamentos factíveis. O experimento descrito neste item procura avaliar de que modo o uso de valores iniciais adaptativamente determinados para CargaMáxima afeta a eficacia e o custo do algoritmo de escalonamento. A idéia aqui é redefinir os valores iniciais de CargaMáxima à medida que o algoritmo é executado repetidas vezes para conjuntos de tarefas com características similares. A similaridade entre as tarefas será associada neste experimento ao fator de folga das tarefas, embora, critérios arbitrariamente mais elaborados que possibilitassem considerar outras características das tarefas poderiam também ser usados.

Foram usados neste experimento os mesmos conjuntos de testes usados no experimento descrito no item 5.3.1. Agora, porém, o parâmetro CargaMáxima foi iniciado com valores derivados daqueles para os quais foi possível encontrar um escalonamento factível no experimento anterior (gráfico 5.18). Usando os resultados desse experimento foram definidos os seguintes valores iniciais para os pares (Fator de Folga, CargaMax): (0,9;29);(1,0;24);(1,1;22);(1,2;20); (1, 3;21);(1,4;20);(1,5;19) e (1,6;20).

Os gráficos 5.22, 5.23 e 5.24, mostram o desvio de carga, custo para escalonamentos bem sucedidos e custo para escalonamentos mal sucedidos, respectivamente, obtidos com a utilização dos valores adaptativo de CargaMáxima. Pode-se observar nos gráficos que o uso desses valores conduziu a um pequeno aumento do desvio de carga, compensado, no entanto, por um razoável decréscimo do custo de produção de escalonamentos balanceados.

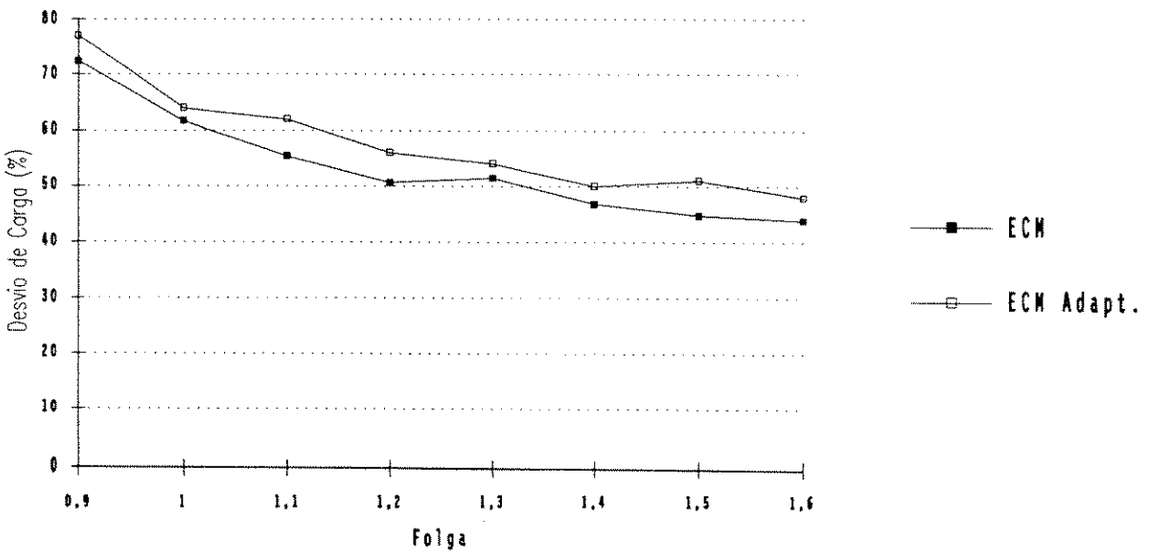


Gráfico 5.22: Balanceamento de carga com o uso de valores adaptativos e CargaMáxima.

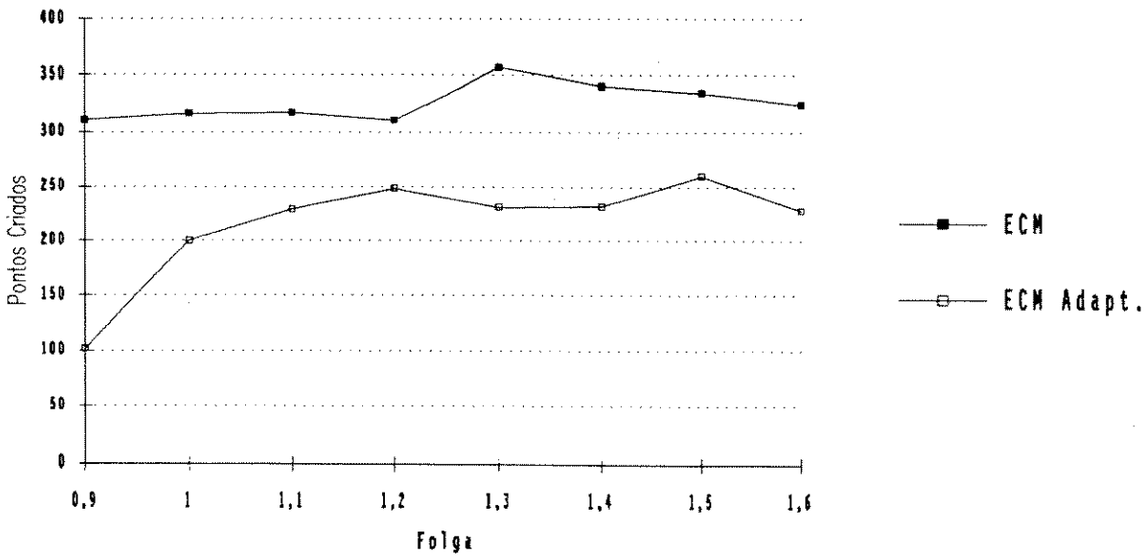


Gráfico 5.23: Custo das buscas bem sucedidas com o uso de valores adaptativos e CargaMáxima.

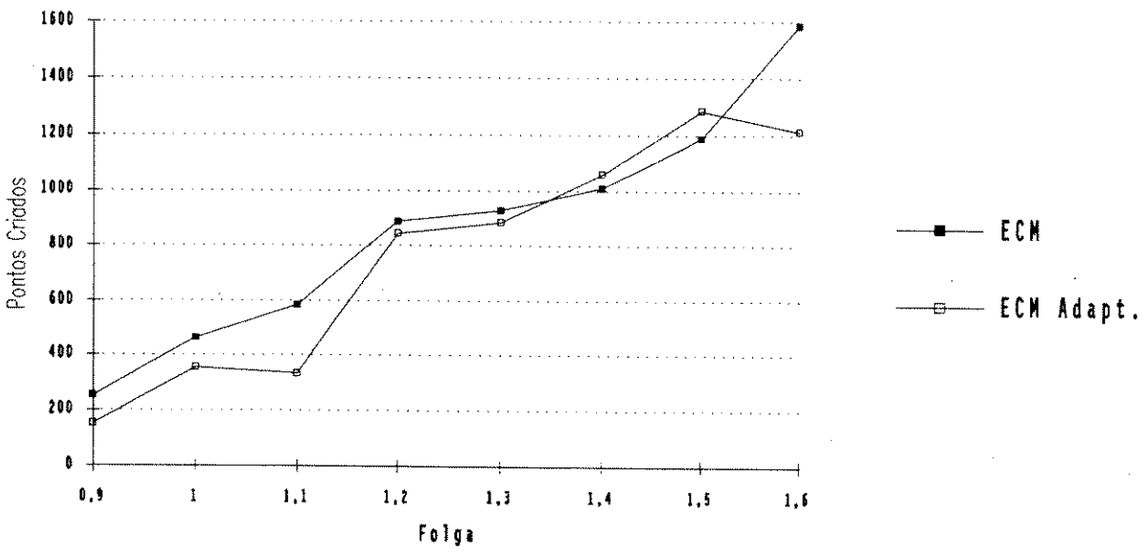


Gráfico 5.24: Custo das buscas mal sucedidas com o uso de valores adaptativos e CargaMáxima.

O experimento relatado neste item estuda apenas os efeitos resultantes da definição de valores iniciais adaptativos para CargaMáxima. Resultados semelhantes são, no entanto, esperados para igual procedimento envolvento FatorDeFolga, conforme sugerem os experimentos descritos em [Ramamritham e Adán (1990)]. Nesse trabalho, Ramamritham e Adán, discutem um experimento semelhante ao aqui apresentado, conduzido para avaliar os efeitos do uso de fatores de folga adaptativos para tarefas que não acessam servidores. Nesse experimento, foi observado que o uso de fatores de folga adaptativos não teve virtualmente nenhum efeito negativo sobre o balanço de carga ou taxa de sucesso do algoritmo, reduzindo, no entanto, de forma notável o custo para a obtenção de escalonamentos factíveis bem balanceados.

5.4 Resumo e comentários finais

Nesta seção foi analisado o comportamento do algoritmo de alocação e escalonamento para conjuntos de tarefas com relações de precedência e compartilhamento de servidores num amplo intervalo de fatores de folga. Os experimentos demonstram a importância de que as heurísticas empregadas na alocação e escalonamento considerem características específicas do modelo de programação e das tarefas a serem escalonadas. Isto foi evidenciado pelo aumento do número de buscas bem sucedidas, acompanhado de redução de custo, quando o algoritmo de escalonamento considera as entidades módulo e servidor no processo de geração do grafo de comunicação, em lugar de tomar decisões aleatórias, ou de basear as suas decisões apenas nas características de pares de nós do grafo de escalonamento.

As heurísticas utilizadas durante a busca por um escalonamento fáctivel, a partir de um grafo de comunicação, permitem que, na maioria das vezes, decisões corretas sejam tomadas. Isto é sugerido pela constatação de que a realização de um pequeno número de "backtracks" (revisão de uma decisão) permite ao algoritmo aproximar-se do que parece ser o limiar da taxa de sucesso alcançável, ou seja, de conseguir encontrar escalonamentos factíveis para a maioria das tarefas (aparentemente) escalonáveis. Acredita-se que o algoritmo atinje o limiar da taxa de sucesso alcançável porque, após a realização de um pequeno número de "backtracks", aumentos expressivos no número de "backtracks" conduzem a aumentos desprezíveis na taxa de sucesso do algoritmo.

Por outro lado, o aumento inicial da taxa de sucesso, verificado quando se permite a realização de um pequeno número de "backtracks", sugere que as decisões erradas devem estar sendo tomadas principalmente durante o mapeamento e escalonamento, e não durante a criação do grafo de comunicação. Exatamente nas etapas em que as entidades módulo e servidor não são consideradas. Esta constatação indica que ganhos significativos podem ser conseguidos com novas heurísticas que considerem, também durante o processo de alocação e mapeamento, as especificidades do modelo de programação do HSTER, em particular, as entidades módulo e servidor, e não apenas as restrições temporais dos blocos de escalonamento.

Os experimentos mostram também que mecanismos simples, como a Estratégia da Carga Máxima (ECM) e a Estratégia do Fator de Ociosidade (EFO), podem contribuir decisivamente para a produção de alocações e escalonamentos bem balanceados. Conforme esperado, o uso dessas estratégias aumenta o custo computacional do algoritmo de alocação e escalonamento, aumento justificável quando for importante produzir escalonamentos melhor balanceados, seja para aumentar a tolerância do sistema a falhas, seja para facilitar o escalonamento dinâmico de tarefas aperiódicas. Com respeito à relação custo benefício das duas estratégias, os experimentos indicam ser mais vantajoso utilizar a ECM.

Uma redução apreciável do custo de aplicação das estratégias ECM e EFO pode ser obtido pela definição adaptativa de valores iniciais para os parâmentros CargaMaxIni e

FatorDeFolgaIni. A escolha desses valores constitui em si um problema interessante, requerendo um estudo mais elaborado.

Como destacado no texto, uma das principais motivações para a utilização das estratégias de balanceamento de carga neste trabalho foi procurar aumentar a probabilidade de dinamicamente escalonar tarefas aperiódicas, a fim de que as suas restrições de tempo sejam satisfeitas. A quantificação dos benefícios obtidos durante o escalonamento dinâmico de tarefas aperiódicas, em decorrência do uso das estratégias de balanceamento de carga, requer, contudo, a realização de experimentos específicos.

6. CONCLUSÃO

Este capítulo está estruturado em três seções. Na seção 6.1 são revistos os objetivos do trabalho e os resultados alcançados. Na seção 6.2 são discutidos os resultados obtidos. Finalmente, na seção 6.3 são propostas algumas linhas de ação para dar continuidade a este trabalho.

6.1 Revisão de objetivos e resultados

Estabeleceu-se como objetivo geral para este trabalho contribuir para a concepção de ambientes de desenvolvimento de sistemas distribuídos de tempo-real flexíveis e previsíveis. Procurou-se atingir esse objetivo abordando alguns dos aspectos principais de um ambiente com essas características: o modelo de programação, as linguagens de programação e a estratégia de escalonamento para cumprimento dos requisitos temporais das aplicações. Esses três elementos foram situados no contexto de um ambiente denominado HSTER ("Hard real-time STER") com modelo e linguagens de programação baseadas em trabalhos anteriores que resultaram na implementação do protótipo do ambiente de desenvolvimento de sistemas distribuídos concorrentes STER [Adán-Coello, Lopes e Magalhães (1987)].

As principais idéias relacionadas ao modelo de programação do HSTER e às linguagens que o respaldam foram originalmente discutidas em [Adán e Magalhães (1991)] e [Adán-Coello e Magalhães (1992)]. O mapeamento de programas em grafos de escalonamento, bem como a estratégia de escalonamento, foram apresentadas em [Adán e Magalhães (1992)]. Os mecanismos para balanceamento de carga (ECM e EFO) e geração de janelas de escalonamento, incorporados ao algoritmo de escalonamento, foram apresentados incialmente em [Ramamritham e Adán (1990)] e posteriormente em [Ramamritham, Fohler e Adán (1993)], juntamente com outros aspectos relacionados ao escalonamento estático de tarefas periódicos complexas. Finalmente, uma estratégia simples para a implementação do conceito de execução garantida de tarefas aperiódicas foi discutido em [Melo, Magalhães e Adán-Coello (1992)].

O modelo de programação

O modelo de programação proposto tem como preocupação central possibilitar que aplicações sejam construídas a partir de módulos de "software" reusáveis. Nesta abordagem, o projeto de uma aplicação tem como preocupação central definir os módulos que a compõem e o seu inter-relacionamento. Feito isso, a implementação da aplicação é caracterizada por duas etapas principais: programação de módulos e configuração de aplicações a partir dos módulos disponíveis.

A construção de aplicações a partir de módulos reusáveis já foi explorada no STER, cuja utilização experimental mostrou a validade desse enfoque, reforçada pelo crescente interesse que as abordagens orientadas a objetos, com as quais guarda estreita semelhança, despertam atualmente.

No modelo de programação do HSTER, a reusabilidade de módulos além de funcional, como no STER, é também temporal. Isto é conseguido desacoplando a implementação da funcionalidade associada a um módulo, feita durante a sua programação, da especificação do seu comportamento temporal, próprio do contexto em que é usado, e definido durante a configuração de uma aplicação.

As linguagens de programação e configuração de módulos

Para a programação de módulos com restrições temporais foi proposta a linguagem LPM-RC (Linguagem de Programação de Módulos com Restrições Críticas de tempo). Esta linguagem, que no referente à programação seqüencial tem a sintaxe e a semântica da linguagem Pascal, estende a linguagem LPM [Lopes (1986)], proposta e implementada como parte do ambiente STER. A LPM-RC procura conservar os elementos da LPM que conferem flexibilidade a módulos e acrescentar os elementos necessários à produção de aplicações previsíveis.

Da mesma forma que a LPM, a LPM-RC procura assegurar a reusabilidade de um módulo oferecendo como mecanismo de interação entre módulos a troca de mensagens, realizada endereçando portas de comunicação locais aos módulos.

Na LPM-RC, os mecanismos e comandos cujos tempos de execução não podem ser previstos -como recursividade, alocação dinâmica de memória e iterações não limitadas - podem ser utilizados apenas pelos módulos sem restrições temporais (módulos NRT).

A LPM-RC permite a definição de módulos especiais, denominados servidores, para controlar o acesso a recursos compartilhados por módulos clientes. Na interação com um servidor, um módulo cliente pode especificar uma operação multiserviço, durante a qual o servidor lhe será alocado em modo exclusivo.

A fim de ser possível calcular o tempo máximo de execução dos módulos com restrições de tempo, eles devem limitar-se a utilizar mecanismos e comandos que não requerem tempos arbitrariamente longos de execução. O conhecimento dos tempos máximos de execução dos módulos, bem como da ocorrência de acessos multiserviços a servidores, é vital para que possa ser buscado um escalonamento que atenda aos prazos dos módulos (atribuídos durante a configuração de uma aplicação).

A LPM-RC permite ainda a especificação de tratadores para exceções associadas ao comportamento temporal dos módulos. Esses tratadores definem as ações a tomar ao nível do módulo para fazer face à exceção. A efetiva execução dos tratadores, quando da ocorrência de exceções, depende de uma política de mais alto nível, definida durante a configuração da aplicação.

Para efetuar a configuração de aplicações foi proposta a LCM-RC (Linguagem de configuração de módulos com restrições críticas de tempo). A LCM-RC compartilha

também diversas características com a LCM, desenvolvida para o STER, e apresenta características complementares, necessárias à especificação do comportamento temporal dos programas de aplicação.

Um programa de configuração, escrito em LCM-RC, define a estrutura lógica de uma aplicação (os componentes e o seu inter-relacionamento), assim como as restrições temporais e de alocação de seus componentes.

Visando simplificar a estruturação de aplicações complexas, ou de grande porte, além das entidades básicas de configuração, os módulos, e da aplicação propriamente dita, o sistema, um programa em LCM-RC pode definir as entidades grupo e estação lógica para representar níveis intermediários de abstração.

A partir de uma configuração, os módulos de uma aplicação podem ser classificados como NRT quando instanciados sem restrições temporais, SRT quando instanciados para execuções aperiódicas com restrições de tempo não críticas e HRT quando instanciados para execução periódica com restrições críticas de tempo.

A estratégia de alocação e escalonamento de módulos

Para garantir que as restrições de tempo de um módulo serão atendidas, de acordo com a classe à qual ele pertence (HRT, SRT ou NRT), foi proposta uma estratégia de escalonamento em três níveis: escalonamento estático para módulos HRT, escalonamento dinâmico dirigido por tempo para módulos SRT e escalonamento dinâmico não dirigido por tempo para módulos NRT. Desses três níveis, o escalonamento de módulos HRT mereceu um maior destaque neste trabalho, por lidar com os componentes cuja execução é essencial garantir.

Para mostrar a viabilidade de integrar as linguagens de programação com a estratégia de escalonamento proposta, foi desenvolvido um escalonador estático para os módulos periódicos HRT, e sugeridas maneiras de combiná-lo com o escalonamento dinâmico de módulos SRT e NRT.

Para proceder à busca de um escalonamento, programas são mapeados em grafos de escalonamento. Um grafo de escalonamento é uma abstração do programa que lhe deu origem que retém apenas a informação necessária aos algoritmos de escalonamento. Os nós do grafo de escalonamento, ou blocos de escalonamento, representam segmentos de código seqüencial dos módulos e constituem as entidades básicas de escalonamento. Os arcos do grafo de escalonamento representam as relações de precedência existentes entre os blocos de escalonamemto.

A busca de um escalonamento para grafos representando programas HSTER é um problema NP-árduo. Razão pela qual utiliza-se para esse propósito um escalonador que emprega um algoritmo de busca heurístico. O algoritmo empregado foi construído a partir de outro descrito na literatura [Ramamritham (1990)], o qual teve que ser estendido para atender às especificidades decorrentes do modelo de programamação proposto para o HSTER. Em particular, foi necessário incorporar ao algoritmo a possibilidade de escalonar as entidades módulo e servidor, bem como acessos multiserviço a servidores.

O sucesso na busca de um escalonamento factível, para um programa que deve ser executado num sistema distribuído, está intimamente ligado à alocação dos módulos às estações do sistema. Em alguns casos, a escolha da estação à qual um módulo deve ser alocado está condicionada à disponibilidade de um recurso específico requerido pelo módulo, em outros, quer-se simplesmente que o módulo cumpra a sua execução num dado prazo, sem que a estação onde ele irá residir seja importante. A determinação manual de uma alocação, para a qual um escalonamento fáctivel possa ser encontrado, tende a tornar-se um problema bastante complexo à medida que o número de módulos e de estações cresce. Por esse motivo, no algoritmo de escalonamento desenvolvido, as decisões de escalonamento (quando executar os módulos) são tomadas conjuntamente com as decisões de alocação (onde executar os módulos).

Módulos aperiódicos, pelo não determinismo associado aos instantes em que ficarão habilitados para execução, devem ser escalonados dinamicamente. O que implica que nem sempre será possível encontrar um escalonamento que lhes permita cumprir os prazos. Por essa razão, no HSTER, módulos aperiódicos são sempre considerados SRT. Quando um módulo aperiódico tiver restrições críticas de tempo, ele deverá ser convertido em periódico e escalonado como tal. A conversão de um módulo aperiódico num equivalente periódico é feita estimando um período para sua ocorrência. Em tempo de execução, o sistema operacional deve recuperar os recursos alocados a um módulo aperiódico, convertido em periódico, em todos os intervalos de escalonemento em que ele não ocorra (não seja habilitado).

Módulos aperiódicos são normalmente independentes, ou seja, não têm relações de precedência nem compartilham recursos com outros módulos. Sugere-se, por esses motivos, que o seu escalonamento seja feito usando algoritmos dinâmicos simples dirigidos por tempo, como, por exemplo, o escalonamento do módulo com o menor prazo primeiro ("earliest deadline first"), ou o escalonamento do módulo com a menor folga primeiro ("least laxity first").

Uma abordagem diferente deve ser considerada quando módulos aperiódicos não forem independentes, por exemplo, pela necessidade de compartilharem servidores com módulos periódicos. Nesse caso, é necessário usar algoritmos dinâmicos de escalonamento que utilizem o conceito de execução garantida. Esses algoritmos devem aceitar para execução apenas os módulos para os quais seja possível assegurar que a sua execução não irá comprometer o cumprimento dos prazos de outros módulos já garantidos, em particular, dos módulos periódicos.

Preparação para o escalonamento dinâmico de tarefas aperiódicas

O balanceamento da carga alocada a um sistema distribuído, durante o escalonamento estático de módulos periódicos, influi decisivamente na tolerância a falhas do sistema, assim como na taxa de tarefas aperiódicas capazes de cumprir os seus prazos.

O desbalanceamento da carga no espaço, ou seja, entre as estações do sistema, pode levar a que todo o sistema seja seriamente afetado por uma pane numa estação muito

carregada. Por outro lado, uma tarefa aperiódica ao chegar a uma estação muito carregada pode ter reduzidas possibilidades de cumprir o seu prazo.

O desbalanceamento da carga no tempo, isto é, a concentração da carga computacional em certos trechos do intervalo de escalonamento, aliada à determinação de instantes rígidos para a execução dos módulos periódicos, tende também a reduzir a taxa de tarefas aperiódicas que cumprirão seus prazos.

Visando contribuir para o aumento da tolerância a falhas do sistema e, particularmente, para uma boa taxa de tarefas aperiódicas capazes de cumprir os seus prazos, foram propostas e implementadas duas estratégias destinadas a balancear a carga do sistema durante a alocação e escalonamento dos módulos periódicos: a estratégia do fator de ociosidade (EFO) e a estratégia da carga máxima (ECM).

A fim de oferecer flexibilidade adicional ao escalonador dinâmico de tarefas aperiódicas, uma vez obtido um escalonamento estático, procura-se substituir os instantes rígidos de início e término dos segmentos de escalonamento por janelas de escalonamento. Com esse propósito, foram propostos e implementados dois mecanismos simples: o esquema local de determinação do instante mais tarde de partida (IMTPL) e o esquema global de determinação do instante mais tarde de partida (IMTPG).

Associada uma janela de escalonamento a um segmento de uma tarefa periódica, o início da execução do segmento não precisará dar-se num instante rígido, e sim, em qualquer ponto da janela, sem que isso comprometa o cumprimento do seu prazo, nem dos prazos dos demais módulos periódicos. Dadas janelas de escalonamento, o escalonador dinâmico pode retardar o início de segmentos periódicos, sempre que isso seja conveniente para atender às tarefas aperiódicas que dinamicamente chegarem ao sistema.

6.2 Discussão dos resultados obtidos

Até recentemente, a maioria dos ambientes de desenvolvimento de sistemas de tempo-real simplesmente ignorava o elemento que de fato os distingue de ambientes voltados a outros domínios, o tempo. A preocupação dos projetistas desses ambientes era, basicamente, oferecer mecanismos que permitissem suportar a execução concorrente de processos usando políticas de escalonamento baseadas em preempção por prioridade. Esse paradigma, conforme discutido no capítulo 1 deste trabalho, é inadequado à produção de sistemas de tempo-real confiáveis, especialmente à medida em que eles se tornam maiores, mais complexos e distribuídos. A partir dessa constatação, estabeleceu-se como objetivo principal deste trabalho procurar contribuir para o desenvolvimento de sistemas distribuídos flexíveis com comportamento temporal previsível.

Procurou-se oferecer flexibilidade através de um modelo de programação, suportado pelas linguagens LPM-RC e LCM-RC, centrado na construção de módulos reusáveis, funcional e temporalmente. Por outro lado, buscou-se assegurar o comportamento temporal previsível de programas colocando à disposição do usuário, nas linguagens, mecanismos que lhe permitem especificar as restrições temporais das aplicações e integrando as linguagens a uma estratégia de escalonamento que busca uma alocação e um escalonamento que satisfaçam a essas restrições.

É importante ressaltar que o modelo de programação, linguagens e estratégia de escalonamento, foram desenvolvidos não somente pensando no objetivo genérico "flexibilidade com previsibilidade", mas também procurando oferecer um modelo suficientemente geral para que possa ser aplicado em um número abragente de aplicações reais. Para isso, suporta-se aplicações que combinem tarefas periódicas e aperiódicas, tarefas com restrições de tempo e tarefas sem restrições de tempo, tarefas com restrições de alocação e tarefas sem restrições de alocação, e tarefas compartilhando, ou não, recursos.

Para o compartilhamento de recursos, permite-se que módulos clientes especifiquem acessos multiserviço a servidores. O escalonador procura garantir a realização desses acessos em modo exclusivo, atendendo às restrições temporais dos módulos clientes. Apesar de útil em inúmeras situações, esse mecanismo não é suportado adequadamente nos ambientes de programação de sistemas de tempo-real existentes.

O algoritmo de escalonamento estático foi avaliado através de simulações que sugerem que se um dado conjunto de tarefas é escalonável, o algoritmo será capaz de encontrar um escalonamento fáctivel sem realizar "backtracks" ou, eventualmente, com um pequeno número de "backtracks". Comparando a heurística empregada pelo escalonador com outras heurísticas que não consideram aspectos específicos do modelo de programação proposto (módulos, servidores e acessos multiserviço), verificou-se que as decisões tomadas considerando esses aspectos conduzem a maiores taxas de sucesso na busca por escalonamentos factíveis, a custos menores.

As simulações mostram também que as estratégias EFO e ECM contribuem substancialmente para a geração de escalonamentos melhor balanceados. Não foi avaliado, porém, em que medida escalonamentos melhor balanceados, aliados a janelas de escalonamento, efetivamente contribuem para aumentar a escalonabilidade de módulos aperiódicos, devendo este aspecto ser objeto de simulações específicas.

Deve-se ressaltar a importância de integrar modelos de programação, linguagens de programação e estratégias de escalonamento de programas. A não consideração dos resultados teóricos e práticos disponíveis na área de escalonamento de tarefas, quando da formulação de modelos e linguagens, pode facilmente conduzir a programas cujo comportamento temporal é impossível de ser garantido. Por sua vez, na formulação de estratégias de escalonamento é importante procurar suportar modelos de programação que permitam atender a um número abrangente de classes de aplicaçõaes reais.

O uso de módulos reusáveis confere grande flexibilidade a uma aplicação para adaptar-se a eventuais mudanças no sistema sendo controlado. A reconfiguração de uma aplicação, incluindo a mudança dos requisitos temporais dos seus componentes, pode frequentemente ser feita alterando apenas o seu programa de configuração, sem necessidade da recodificação ou mesmo recompilação de módulos.

Usando o modelo, linguagens e estratégia de escalonamento propostos, para que uma nova configuração de uma aplicação possa ser posta em prática, é necessário parar a execução do programa com a configuração anterior. Em alguns tipos de aplicação isso pode ser bastante complicado e oneroso, razão pela qual, sugere-se no próximo item o suporte à reconfiguração dinâmica como uma interessante maneira de dar continuidade a este trabalho.

6.3 Continuidade deste trabalho

A partir dos resultados mais específicos e concretos produzidos neste trabalho - o modelo de programação, as linguagens e o escalonador estático - e do melhor entendimento que durante o seu desenrolar foi alcançado com respeito aos problemas associados ao desenvolvimento de sistemas com restrições críticas de tempo, sugerem-se, a seguir, três possíveis linhas de ação que poderiam direcionar trabalhos futuros.

A primeira linha de ação sugerida teria como objetivo implementar um protótipo de um ambiente para o desenvolvimento e execução de aplicações distribuídas, o HSTER, formado pelas linguagens LPM-RC e LCM-RC e pelo núcleo de um sistema operacional distribuído. Uma segunda linha de ação teria por finalidade o aprofundamento em tópicos específicos considerados neste trabalho. Pode-se destacar nesta linha o aprimoramento das heurísticas usadas para o escalonamento e balanceamento de carga de tarefas periódicas. A terceira linha de trabalho sugerida teria como objetivo aumentar a flexibilidade e generalidade do ambiente proposto através de um processo evolutivo.

Implementação de um protótipo

A implementação de um protótipo do ambiente permitiria avaliar a efetividade das idéias nele incorporadas. A sua utilização experimental possibilitaria também a obtenção de um entendimento maior ainda dos problemas associados ao desenvolvimento e suporte à execução de sistemas distribuídos de tempo-real, a partir do que, novos mecanismos e conceitos poderiam ser propostos, implementados e avaliados.

A implementação de um protótipo do ambiente envolve básicamente três atividades: 1) desenvolvimento de processadores para as linguagens LPM-RC e LCM-RC, 2) desenvolvimento do núcleo de um sistema operacional que suporte a execução dos programas gerados pelas linguages e 3) integração do núcleo a protocolos de comunicação determinísticos.

O processador para a LPM-RC deve, em adição às funções genéricas de processadores para linguagens "convencionais", gerar as informações necessárias à construção de grafos de escalonamento. Para isso, deve dividir os módulos em segmentos de escalonamento e determinar os seus tempos máximos de execução. Esta última atividade é intimamente dependente da arquitetura do "hardware" das estações físicas onde os módulos serão executados. Sendo importante, para que possa ser levada a cabo, o uso de arquiteturas onde o tempo máximo de execução de instruções seja determinístico e não apresente variação apreciável, como pode ocorrer, por exemplo, em arquiteturas do tipo "pipeline" [Colnaric, (1992)].

Para a montagem do grafo de escalonamento, durante o processamento de um programa em LCM-RC, é necessário determinar os tempos máximos necessários à transferência de mensagens entre módulos localizados em estações físicas distintas. Do mesmo modo que a determinação dos tempos máximos de execução de segmentos de escalonamento está intimamente ligada à arquitetura das estações onde são executados, a determinação dos tempos máximos necessários ao envio de mensagens é dependente do

meio físico e protocolos de comunicação enpregados para conectar as estações do sistema [Stankovic, Ramamritham and Nahum, (1991)] [Arvind, Ramamritham and Stankovic, (1991)].

O núcleo do sistema operacional deverá ser construído em torno da estratégia de escalonamento proposta. Ele será responsável pela execução dos módulos periódicos e pelo uso do meio de comunicação de acordo com o escalonamento estático, e pela detecção e tratamento de exceções conforme a política definida na configuração das aplicações. Além disso, o núcleo deverá incorporar um algoritmo de escalonamento dinâmico para escalonar as tarefas aperiódicas, sem comprometer os prazos das tarefas periódicas cuja execução já foi assegurada fora de linha. A execução dos módulos periódicos, de acordo com o escalonamento "off-line", e o escalonamento dinâmico de módulos aperiódicos dependem, naturalmente, da existência de uma noção global de tempo [Lamport, (1978); Lamport, (1984); Kopetz and Oschsenreither, (1987); Volz, Sha and Wilcox (1991)].

Quando não for necessário suportar o conceito de execução garantida de módulos aperiódicos, o escalonador dinâmico pode, conforme sugerido no capítulo 4, utilizar algoritmos dinâmicos simples dirigidos por tempo para os quais já se dispõe de apreciável experiência de implementação.

O suporte ao conceito de execução garantida de módulos aperiódicos é consideravelmente mais difícil. Um trabalho nessa direção poderia partir de algoritmos disponíveis relativamente simples como, por exemplo, o proposto por Melo (1993), ou mesmo de algoritmos mais elaborados, como aqueles desenvolvidos no projeto SPRING [Stankovic e Ramamritham (1991)].

Na implementação de um protótipo do HSTER seria interessante, e relativamente simples, suportar o conceito de análise incremental do comportamento temporal de programas, incluido no contexto mais abrangente da construção incremental de programas.

A construção incremental de programas deve neste contexto ser entendida como o processo de criação de programas em várias etapas, ou estágios, onde, a transição de uma etapa para a seguinte, ocorre à medida que, progressivamente, se obtém um maior entendimento da estrutura dos programas e dos seus componentes.

A análise incremental do comportamento temporal de um programa consiste, básicamente, em analisar o programa em cada um dos seus estágios de desenvolvimento, visando verificar se as suas restrições temporais, naquele estágio, podem ser cumpridas. Na análise feita em cada estágio, devem ser utilizados os tempos de execução (no pior caso) efetivos dos módulos já disponíveis, e estimativas dos tempos de execução dos componentes em desenvolvimento (componentes que, no processo de construção da aplicação, podem ser sucessivamente decompostos em novos componentes os quais, eventualmente, redundarão em módulos).

Evolução do modelo e das linguagens

O modelo e as linguagens propostos podem evoluir em diversos graus e direções. Uma evolução interessante do modelo seria a generalização dos mecanismos de suporte ao tratamento de exceções. Isto poderia ser feito, por exemplo, permitindo a inclusão de exceções definidas pela aplicação, não necessáriamente associadas ao cumprimento de restrições temporais, e oferecendo comandos para suportar a reconfiguração do sistema durante o tratamento das exceções

Com relação à LCM, uma extensão interessante seria permitir a especificação da arquitetura física do ambiente em que o sistema será executado, a exemplo da linguagem Pearl. Desse modo, seria possível definir a localização, o tipo e a quantidade de recursos disponíveis, facilitando a implementação de estratégias automáticas para alocação de módulos, visando melhorar o balanço da carga e o aumento da tolerância a falhas do sistema.

A linha de evolução mais abrangente deste trabalho provávelmente seja o suporte à reconfiguração dinâmica de programas. Ela poderia ser oferecida em diferentes níveis de generalidade, incluindo a reconfiguração da aplicação mediante a redefinição das restrições temporais de módulos, da reconexão de portas, da eliminação de instâncias de módulos, da criação de instâncias de módulos, da inclusão de novos tipos de módulos na aplicação, ou mesmo todas essas possibilidades juntas.

Caso se decidisse suportar reconfigurações dinâmicas, elas poderiam ser especificadas, basicamente, de duas maneiras: pelo usuário durante a operação do sistema, ou programadas para serem realizadas em determinados instantes de tempo, ou quando o programa atingisse determinados estados, associados, por exemplo, à ocorrência de exceções.

A elaboração de um modelo de reconfiguração de programas, bem como a sua sua operacionalização, não é trivial. Entre outros requisitos, é necessário assegurar que, durante uma reconfiguração, as consistências lógica e temporal do programa são preservadas. Para isso, deve-se considerar questões tais como: Quando uma reconfiguração pode ser efetuada? Como garantir as restrições temporais da nova configuração? (Deve-se garantir estáticamente as restrições temporais de configurações típicas do sistema, associadas a fases de operação,ou a garantia deve ser buscada dinâmicamente?).

Dado o grande interesse que o paradigma de desenvolvimento e programação orientado a objetos desperta atualmente, com o qual o modelo de programação do HSTER guarda estreita semelhança, uma das mais promissoras linhas de evolução do modelo consiste, precisamente, num aprofundamento nessa direção. Seguindo essa linha, Cardozo, Magalhães e Adán, (1993) discutem uma proposta baseada no conceito de objetos ativos, onde é dado especial destaque ao suporte a objetos com restrições de tempo não críticas e invocação dinâmica. Essa proposta inclui diversos dos aspectos relativos à reconfiguração dinâmica de aplicações mencionados nos parágrafos anteriores.

APÊNDICE A
Sintaxe das linguagens LPM-RC e LCM-RC

Neste apêndice será apresentada a sintaxe das linguagens LPM-RC e LCM-RC, usando uma extensão da notação BNF. As principais convenções usadas na notação são as seguites:

- <identificador> : identificador é um simbolo não terminal;
- [<construção>] : <construção> é opcional;
- [<construção>]* : indica zero ou mais repetições de <construção>;
- <construção1>|<construção2> : deve-se escolher a <construção1> ou a <construção2>
- quando um metasímbolo for parte de um símbolo terminal, ele será escrito entre aspas. Por exemplo os metasímbolos <,>,[e], deverão ser escritos "<", ">", "[" e "]", quando usados como símbolos terminais.
- a palavra identificador será freqüentemente abreviada por id.

A.1 A Linguagem de Programação de Módulos com Restrições Críticas de Tempo (LPM-RC)

A LPM-RC é uma extensão da linguagem Pascal. Neste apêndice, apenas as extensões ao Pascal são tratadas. A descrição da sintaxe da linguagem Pascal pode ser encontrada em [Jensen and Wirth (1974)].

A.1.0 Unidade de definição

<unidade de definição>::=
DEFINE <identificador>:<símbolos exportados>;
[<definição de contexto >]
[<definição de constantes>]
[<definição de tipos>]
[<declaração de procedimentos e de funções>]
END.

<símbolos exportados>::= <identificador> [,<identificador>]*

<definição de contexto>::=
USE <referência a unidade de definição> [;<referência a unidade de definição>]*;

<referência a unidade de definição>::=
<identificador>[,<identificador>]*:<unidade de definição>

A.1.1 Declaração de um módulo

<declaração de um módulo> ::=
[UNBOUNDED] [NONPREEMPTABLE] <tipo de módulo> <identificador>
[(<parâmetros formais>)];
[<definição de contexto>]
[<declaração de portas de interface>]
[<definição da condição de habilitação>]
[<declaração de mensagens>]

[<declaração de rótulos>]
[<definição de constantes>]
[<definição de tipos>]
[<declaração de variáveis>]
[<declaração de funções e procedimentos>]

[<declaração do iniciador>]
[<declaração de tratador de exceção temporal>]*

BEGIN
<comandos>
END.

<tipo de módulo>::= MODULE|SERVER

<parâmetros formais>::= <grupo de constantes>[;<grupo de constantes>]*

<grupo de constantes>::=
    <id. de constante> [,<id. de constante>]:<tipo básico Pascal>

A.1.2 Definição de contexto

<definição de contexto>::=
    USE <referência a unidade de definição> [;<referência a unidade de definição>]*;

<referência a unidade de definição>::=
    <identificador>[,<identificador>]*:<unidade de definição>

A.1.4 Declaração de portas de interface

<declaração de portas de interface>::=
    ENTRYPORT <declaração de porta de entrada>
        [;<declaração de porta de entrada>]*; |
    EXITPORT <declaração de porta de saida> [;<declaração de porta de saida>];

<declaração de porta de entrada>::=
    <porta>[,<porta>]:<tipo da mensagem> [<resposta> | <fila>]

<declaração de porta de saida>::=
    <porta>[,<porta>]:<tipo da mensagem> [<resposta>]

<porta>::=<identificador>["["<família>"]"]

<família>::=<id. de tipo intervalo> | <intervalo>

<tipo da mensagem>::=<tipo básico Pascal> |
    <identificador de tipo de mensagem>

<resposta>::=REPLY <tipo da mensagem>

<fila>::=QUEUE <tamanho da fila>

<tamanho da fila>::=
    <constante inteira positiva> | <id. de constante inteira positiva>

<intervalo>::= <limite de intervalo> .. <limite de intervalo>

<limite de intervalo>::= <id. de constante>|<constante>

A.1.5 Declaração de mensagens

<declaração de mensagens>::= MESSAGE <mensagem> [;<mensagem>]*;

<mensagem>::=<identificador>[,<identificador>]:<tipo da mensagem>

A.1.6 Definição da condição de habilitação do módulo

<definição de condição de habilitação>::=
ENABLED_BY [<tipo de habilitação>]<lista de portas de entrada>

<tipo de habilitação>::= ALL | ANY

<lista de portas de entrada>::=
<id. porta de entrada> [,<id. porta de entrada>]*

A.1.7 Declaração do iniciador

<declaração do iniciador>::=INITIATE; <bloco Pascal>;

A.1.7 Declaração de tratador de exceção temporal

<declaração de tratador de exceção temporal>::=
HANDLER <tipo de exceção>;
<declarações pascal>
BEGIN
<comandos>
END;

<tipo de exceção>::=
ENABLING_FAULT | NON_GUARANTEED | MISSED_DEADLINE

A.1.8 Comandos

<comandos>::=<comado>[;<comando>]*

<comando>::=<comando Pascal>|<comando LPM-RC>

<comando LPM-RC>::=
<comando SEND>|
<comando RECEIVE>|
<comando REPLY>|
<comando SELECT>|
<comando LOCK>|
<comando FOR limitado>|
<comando WHILE limitado>|
<comando REPEAT limitado>|
<comando LOOP limitado>|
<comando EXIT>

A.1.9 Comando SEND

<comando SEND>::= SEND <id. de mensagem> TO <id. de porta de saida>
[<cláusula de resposta>]

<cláusula de resposta>::=
<WAIT bloqueante>|<WAIT temporariamente bloqueante>

<WAIT bloqueante>::= WAIT <id. de mensagem>

<WAIT temporariamente bloqueante>::=
WAIT <id. de mensagem>
[=> <comandos>]
FAIL [<especificação de tempo>] => <comandos>
END

<especificação de tempo>::=<tempo> <unidade>

<tempo>::=<constante inteira positiva> | <id. de constante inteira positiva>

<unidade>::=h|m|s|ms|us

A.1.10 Comando RECEIVE

<comando RECEIVE>::=
RECEIVE <id. de mensagem> FROM <id. de porta de entrada>
[REPLY <id. de mensagem>] [<tratamento do RECEIVE>]

<tratamento do RECEIVE>::=
<trata recepção> <trata falha do RECEIVE> END |
<trata recepção> END |
<trata falha do RECEIVE> END

<trata recepção>::= [=> <comandos>]

<trata falha do RECEIVE>::= FAIL [<especificação de tempo>] => <comandos>

A.1.11 Comando REPLY

<comando REPLY>::=
REPLY <id. de mensagem> TO <id. de porta de entrada>

A.1.12 Comando SELECT

<Comando SELECT>::=
<tipo de seleção>
<parte select>
[OR_SELECT <parte select>]*
[ELSE_SELECT <comandos>]
END

<tipo de seleção>::= PSELECT | RSELECT

<parte select>::= [<repetição>][<guarda>] <cláusula de seleção> [=><comandos>]

<repetição>::=
FOR <variável de controle>:=<valor inicial> TO <valor final> DO

<guarda>::= WHEN <expressão booleana>

<cláusula de seleção>::= <comando RECEIVE> | <temporização>

<temporização>::=TIMEOUT <especificação de tempo>

A.1.13 Comando LOCK

<comando LOCK>::=
LOCK
<comando SEND>
[<comando, excepto um LOCK>]*
<comando SEND>
END

A.1.14 Comando LOOP limitado

<comando LOOP limitado>::=
LOOP
<comandos>
END <limite de iterações>

<limite de iterações>::= AFTER <inteiro decimal> TIMES [=> <comandos> END]

A.1.15 Comando EXIT

<comando EXIT>::=EXIT

A.1.16 Comando WHILE limitado

<comando WHILE limitado>::= <comando WHILE do Pascal> <limite de iterações>

A.1.17 Comando REPEAT limitado

<comando REPEAT limitado>::= <comando REPEAT do Pascal> <limite de iterações>

A.1.18 Comando FOR limitado

<comando FOR limitado>::= <comando FOR do Pascal> <limite de iterações>

A.2 Linguagem de Configuração de Módulos com Restrições Criticas de Tempo (LCM-RC)

Nesta seção é apresentada a sintaxe das entidades grupo, estação e sistema. Quando um símbolo não terminal for referenciado e não definido posteriormente, deve-se entender que a sua sintaxe é análoga à do mesmo símbolo não terminal da LPM-RC.

A.2.1 Declaração de um Grupo

<declaração de um Grupo>::= [UNBOUNDED] [NONPREEMPTABLE]
GROUP <identificador> [(<parâmetros formais>)];
[<declaração de contexto>]
[<declaração de portas de interface>]
[<declaração dos índices de famílias>]
[<declaração das instâncias componentes do grupo>]
[<associação de portas à interface>]
[<conexão das portas internas>]
END.

A.2.1.1 Declaração dos índices de famílias

<declaração dos índices de famílias>::= FAMILY <ind_família> [;<ind_família>]*

<ind_família>::=<lista de identificadores>:"["<intervalo>"]"

<lista de identificadores>::=<identificador>[;<identificador>]*

A.2.1.2 Definição das Instâncias Componentes do Grupo

<definição das instâncias componentes do grupo>::=
INSTANCE <instâncias comp. grupo> [;<instâncias comp. grupo>]*;

<instâncias comp. grupo>::=
<lista de instâncias comp. grupo>: <id. de tipo de componente de grupo>

<lista de instâncias comp. grupo>::=
<declarador de instância comp. grupo> [,<declarador de instância comp. grupo>]*;

<declarador de instância comp. grupo>::=
<identificador> [<intervalo>] [(<parâmetros reais>)]

<id. de tipo de componente de grupo>::=<id. de módulo> | <id. de grupo>

A.2.1.3 Associação de portas à interface

<associação de portas à interface>::=
ASSOCIATE <cláusula de associação> [;<cláusula de associação>]*;

<cláusula de associação>::= <associação de família>|<associação individual>

<associação de família>::=
FOR <id. de índice de família>:=
<limite de intervalo> TO <limite de intervalo> DO <associação>

<associação>::=
<associação individual> |
BEGIN
<associação individual> [;
<associação individual>]*
END

<associação individual>::=
<porta de um componente> WITH <id. de porta da interface>[,<id. de porta da interface>

<porta de um componente>::=<porta de um módulo>|<porta de um grupo>

<porta de um módulo>::= <id. de um módulo>.<id. porta>["["<membro de família>"]"]

<porta de um grupo>::= <id. de um grupo>.<id. porta>["["<membro de família>"]"]

<membro de família>::=<constante> | <id. de constante> | <id. de índice de família>

A.2.1.4 Conexão das portas internas

<conexão das portas internas>::=
LINK <ligação de portas>[;<ligação de portas>]

<ligação de portas>::=<ligação de família>|<ligação individual>

<ligação de família>::=
FOR <id. índice de família>:=<limite de intervalo>
TO <limite de intervalo> DO <ligação>

<ligação>::= <ligação individual> |
BEGIN
<ligação individual> [;
<ligação individual>]*
END

<ligação individual>::=
<porta de saída de um componente> TO <porta de entrada de um componente>[,
<porta de entrada de um componente>]

<porta de saída de um componente>::=<porta de um módulo>|
<porta de um grupo>

<porta de entrada de um componente>::=<porta de um módulo>|
<porta de um grupo>

A.2.2 Declaração de uma estação lógica

<declaração de uma estação lógica>::=
STATION <identificador> [(<parâmetros formais>)];
[<definição de contexto>]
[<declaração de portas de interface>]
[<declaração de índices de famílias>]
[<definição de instâncias componentes da estação>]
[<associação de portas à interface>]
[<conexão das portas internas>]
[<definição de restrições temporais>]
END.

A.2.2.1 Definição de instâncias componentes da estação

<definição de instâncias componentes do estação>::=
INSTANCE <instâncias comp. estação> [;<instâncias comp. estação>]*;

<instâncias comp. estação>::=
<lista de instâncias comp. estação>:<id. de tipo de componente de estação>

<lista de instâncias comp. estação>::=
<declarador de instância comp. estação> [,<declarador de instância comp. estação>]*;

<declarador de instância comp. estação>::=
<identificador> [<intervalo>] [(<parâmetros reais>)]

<id. de tipo de componente de estação>::=<id. de módulo> | <id. de grupo>

A.2.2.2 Definição de restrições temporais

<definição de restrições temporais>::=
SCHEDULE <comando de escalonamento> [;<comando de escalonamento>]*

A.2.2.3 Comando de escalonamento

<comando de escalonamento>::=
<comando EVERY>|
<comando SPORADICALLY>|
<comando GUARANTEE>

A.2.2.4 comando EVERY

<comando EVERY>::=
EVERY <período>
<parte DO>
[ELSE <política de trat. exceção>]

<período>::=<especificação de tempo>

A.2.2.5 comando SPORADICALLY

<comando SPORADICALLY>::=
SPORADICALLY
<parte DO>
[ELSE <política de trat. exceção>]

A.2.2.6 Comando GUARANTEE

<comando GUARANTEE>::= <garantia simples> | <garantia múltipla>

<garantia simples>::=
GUARANTEE
<parte DO>
[ELSE <política de trat. exceção>]

<garantia múltipla>::=
GUARANTEE
<parte DO>
[OR <parte DO>]*
[ELSE <política de trat. exceção>]
END

A.2.2.7 Parte DO

<parte DO>::=
DO <entidade de escalonamento> [AFTER <instante de partida>]
[WITHIN <prazo>]

<entidade de escalonamento>::= <id. instância> | HANDLER

<instante de partida>::=<especificação de tempo>

<prazo>::=<especificação de tempo>

A.2.2.8 Política de tratamento de exceção

<política de trat. exceção>::= HANDLE | SKIP | STOP

A.2.3 Definição de um sistema

<definição de um sistema>::=
SYSTEM <identificador>;
[<definição de contexto>]
[<definição da arquitetura física do sistema distribuído>]
[<definição das instâncias componentes do sistema>]
[<declaração dos índices de famílias>]
[<conexão de portas de componentes do sistma>]
[<definição de restrições de alocação de componentes>]
[<definição de restrições temporais>]
END.

A.2.3.1 Definição da arquitetura física do sistema distribuído

<definição da arquitetura física do sistema distribuído>::=
NETWORK <definição de nó>[;<definição de nó>]

<definição de nó>::= <identificador> = <endereço físico do nó>

A.2.3.2 Definição das Instâncias Componentes do Sistema

<definição das instâncias componentes do sistema>::=
INSTANCE <instâncias comp. sistema> [;<instâncias comp. sistema>]*;

<instâncias comp. sistema>::=
<lista de instâncias comp. sistema>:<id. de tipo de componente de sistema>

<lista de instâncias comp. sistema>::=
<declarador de instância comp. sistema> [,<declarador de instância comp. sistema>]*;

<declarador de instância comp. sistema>::=
<identificador> [<intervalo>] [(<parâmetros reais>)]

<id. de tipo de componente de sistema>::=
<id. de módulo> | <id. de grupo> |<id. de estação>

A.2.3.3 Conexão das portas de componentes do sistema

<conexão das portas de comp. do sistema>::=
LINK <ligação de portas comp. sist.>[;<ligação de portas comp. sist.>]

<ligação de portas comp. sist.>::=
<ligação de família comp. sist.>|<ligação individual comp. sist.>

<ligação de família comp. sist.>::=
FOR <id. índice de família comp. sist. >:=<limite de intervalo>
TO <limite de intervalo>
DO <ligação comp. sist.>

<ligação comp. sist.>::=
<ligação individual comp. sist.> |
BEGIN
<ligação individual comp. sist.> [;
<ligação individual comp. sist.>]*
END

<ligação individual comp. sist.>::=
<porta de saída comp. sist.> TO <porta de entrada comp. sist.>[,
<porta de entrada comp. sist.>]

<porta de saída comp. sist.>::=
<porta de um módulo>|<porta de um grupo>|<porta de uma estação)

<porta de entrada comp. sist.>::=
<porta de um módulo>|<porta de um grupo>|<porta de uma estação>

A.2.3.4 Definição de restrições de alocação de componentes

<definição de restrições de alocação de componentes>::=
ALLOCATE <restrição de alocação>[;<restrição de alocação>]*

<restrição de alocação>::=
<id. comp. do sistema>[,<id. comp. do sistema>]* AT <localização>

<localização>::=
<identificador de nó> | SAME | DIFFERENT

REFERÊNCIAS

Adán-Coello J.M. (1986). " Suporte de Tempo-Real para um Ambiente de Programação Concorrente". Dissertação de Mestrado. Faculdade de Engenharia Elétrica. UNICAMP. Agosto. 1986.

Adán-Coello, J.M., A.B. Lopes e M.F. Magalhães. (1987). "Um Ambiente para Desenvolvimento de Software de Tempo-Real e sua Implementação". Anais do VII Congresso da Sociedade Brasileira de Computação. Salvador. Ba.

Adán, J.M. and M.F. Magalhães. (1991). "Developing Reconfigurable Distributed Hard Real-Time Control Systems in STER". Proc. IFAC workshop on Algorithms and Architectures for Real-Time Control.

Adán-Coello, J.M. e M.F. Magalhães. (1992). "Ster's Multilevel Programming Model for Distributed Hard Real-Time Systems". Microprocessing and Microprogramming, 34, North-Holland.

Adán J.M. and M.F. Magalhães. (1992). "A Scheduling Strategy for a Distributed Hard Real-Time Programming Environment". Proc. IFAC Real-Time Programming.

Arvind, K., K. Ramamritham and J.A. Stankovic. (1991). "A Local Area Network Architecture for Communication in Distributed Real-Time Systems". The Journal of Real-Time Systems, 3(2).

Audsley, N.C, A. Burns, M. F. Richardson and A. J. Wellings. (1991). "Hard Real-Time Scheduling: The Deadline-Monotonic Approach". 8th IEEE Workshop on Real-Time Operating Systems and Software, Atlanta, USA.

Audsley, N.C, A. Burns, M. F. Richardson and A. J. Wellings. (1992). "Deadline Monotonic Scheduling Theory". IFAC International Workshop on Real- Time Programming.

Baker., T. (1989). "The Use of Ada for Real-Time Systems". 26th AOC EW Technical Symposium and Convention.

Barnes, J.G.P. (1976). "RTL/2 Design and Philosophy". London: Heyden International Topics in Science.

Berry, D., S. Moisan, and J. Rigault. (1983). "Esterel: Towards a Sunchronous and Semantically Sound High-Level Language for Real-Time Applications". IEEE Real-Time Systems Symp., Dec.

Berry, G., L. Cosserat. (1985). "The ESTEREL synchronous programming language and its mathematical semantics". In Lecture Notes in Computer- Science, 197, Springer-Verlag.

Bihari, T., P. Gopinath and K. Schwan. (1989). "Object-Oriented Design of Real-Time Software". In Proc. of the IEEE Real-Time Systems symposium.

Blazewicz, J., J.K. Lenstra and A.H.G.R. Kan. (1983). "Scheduling Subject to Resource Constraints: Classification and Complexity". Discrete Applied Mathematics 5, 11-24.

Burns, A., and A. Wellings. (1990a). "Real-Time Systems and Their Programming Languages". Addisson-Wesley.

Burns, A., and A.J. Wellings. (1990). "The Notion of Priority in Real-Time Programming Languages". Computer Languages, Vol. 15, no. 3.

Cardozo, E., M.F. Magalhães and J.M. Adán. (1993). "Programming Flexible Distributed Real-Time Systems". IEEE Fourth Workshop on Future Trends of Distributed Computing Systems, September. (A ser realizado).

Caspi, P. et al. (1987). "LUSTRE: a declarative language for progrmming synchronous systems". 14th ACM symposium on Principles of Programming Languages.

Castor., V. (1989). "Letter intiating the Ada revision process". ACM Ada Letters, IX(1):12.

Cheng, S., J. A. Stankovic and K. Ramamritham. (1986). "Dynamic Scheduling of Groups of Tasks with Precedence Constraints in Distributed Hard Real-Time Systems". IEEE Real-Time Systems Symposium.

Cheng, S.-C., J. A. Stankovic and K. Ramamritham. (1987). "Scheduling Algorithms for Hard Real-Time Systems - A brief Survey". Real-Time Systems Newsletter, vol.3, no. 2. IEEE Computer Society.

Colnaric, M., W.A. Halang. (1992). "Architectural Support for Predictability in Hard Real-Time Systems". in Proc. IFAC Real-Time Programming.

DeMarco, T., "Structured Analysis and System Specification", Yourdom Press, 1981.

DIN (1979). "DIN 66253, Programming Language PEARL, Pat 1:Basic PEARL, Part 2: Full PEARL". Deutsches InstituT fur Normung (DIN).

DOD (1980). "MIL-STD-1815: Ada Programming Language". US Department of Defense, DoD Single-Stock Point, Philadelphia, Penn.

French, S. (1982). "Sequencing and Scheduling: An Introduction to the Mathematics of the Job-Shop". Ellis Horwood Ltda.

Garey M.R. and D.S. Johnson. (1975). "Complexity Results for Multiprocessor Scheduling Under Resource Constraints". SIAM J. Comput. 4(4).

Garey M.R. and D.S. Johnson. (1978). "Strong NP-Completeness Results: Motivations Examples and Implications". JACM. 25(3).

Garey M.R. and D.S. Johnson. (1979). "Computers and Intractability: a Guide to the Theory of NP-Completeness". Freemanm, San Francisco.

Grimshaw, A. S., A. Silberman and J. W.S. Liu. (1990). "Real-Time Mentant Programming Language and Arqchitecture". Seventh Workshop on Real-Time Operating Systems and Software.

Guimarães, E. G., A.B. Lopes, J.M. Adán-Coello e M.F. Magalhães. (1989). "A Distribuição do Ambiente para Desenvolvimento de Software de Tempo Real STER". Seminário Franco-Brasileiro em Sistemas Distribuídos. Florianópolis, SC.

Halang, W. A., and R. Henn. (1988). "Additional Pearl Language Structures for the Implementation of Reliable and Inherently safe Real-Time Systems". IFAC Real-Time Programming.

Halang, W. A., A. D. Stoyenko. (1990). "Comparative Evaluation of High-Level Real-time Programming Languages". The Journal of Real-Time Systems, vol. 2., pp. 365-382.

Halang, W. A., A. D. Stoyenko. (1991). "Constructing Predictable Real-Time Systems". Kluwer Academic Publishers.

Herrtwich, R.G. (1991). "Time Capsules: An Abstraction for Access to Continous-Media Data". The Journal of Real-Time Systems. 3 (4). December.

IBM Corporation (1954). "Specifications for the IBM mathematical FORmula TRANslating system FORTAN".

Ichbiah, J. et al. (1979). "Preliminary Ada reference manual and rationale for the design of the Ada programming language". ACM SIGPLAN notices, 14 (6), June.

IECCA (1970). "IECCA Official Definition of CORAL 66". London: Her Majesty's Stationery Office.

Ishikawa, Y., H. Tokuda and C. W. Mercer. (1990). "Object-Oriented Real-Time Language Design: Constructs for Timing Constraints". Proc. OOPSLA/ECOOD.

Jensen, K. and N. Wirth. (1974). "Pascal User Manual and Report". Springer- Verlag.

Joseph, M. and A. Goswami. (1989). "Formal description of realtime systems: a review". Information and Software Technology. 31(2).

Kenny, K. B., and K.-J. Lin. (1991). "Building Flexible Real-Time Systems Using the Flex Language". IEEE Computer.

Kernighan, B.W., and R. Pike. (1984). "The UNIX Programming Environment". Prentice-Hall, Inc.

Kernighan, B.W., and D.M. Ritchie. (1988). "The C Programming Language". 2nd. Ed., Englewood Cliffs, NJ. Prentice Hall.

Kopetz, H. and Oschsenreither, W. (1987). "Clock Synchronization in Distributed Real-Time Systems". IEEE Trans. on Computers 36 (8).

Kramer, J., J. Magee, A. Finkelstein. (1990). "A Construtive Approach to the Design of Distributed Systems". 10th IEEE International Conference on Distributed Systems.

Klingerman, E., and A. D. Stoyenko. (1986). "Real-Time Euclid: A Language for Reliable Real-Time Systems". IEEE Transactions on Software Engineering.

Lamport, L. (1978). "Time, Clocks, and the Ordering of Events in Distributted Systems". Comm. of the ACM 21 (6).

Lamport, L. (1984). "Using Time Instead of Timeouts for Fault-Tolerant Distributed Systems". ACM Trans. on Programming Languages and Systems, pp. 254-250.

Le Guernic, P., T. Gautier. (1990). "Data-Flow to Von Neumann: The SIGNAL Approach". Rapport de Recherche no. 1229, INRIA.

Leinbaugh, D. (1980). "Guaranteed Response Times in a Hard-Real-Time Environment". IEEE Trans. on Software Engeineering. vol. SE-6, No. 1.

Leinbaugh, D. W., and M. R. Yamini. (1986). "Guaranteed response times in a distributed hard-real-time environment". IEEE Trans. Software Eng. Dec.

Le Lann, G. (1990). "Critical Issues for the Development of Distributed Real-Time Computing Systems". Rapport de Recherche no. 1274. INRIA.

Le Lann, G. (1991). "Designing Real-Time Dependable Distributed Systems". Rapport de Recherche no. 1425. INRIA.

Leoczky, J., L. Sha, Y. Ding. (1989). "The Rate Monotonic Scheduling Algorithm: Exact Characterization And Average Case Behavior". IEEE Proc. Real-Time Systems Symposium.

Liu, C.L., J. W. Layland. (1973). "Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment". JACM, vol. 20. no.1, January.

Liu, J.W..-S, et al. (1991). "Algorithms for Scheduling Inprecise Computations". IEEE Computer, May.

Lopes, A.B. e J.M. Adán-Coello. (1985). "Um Ambiente para o Projeto e Implementação de Software para Sistemas Distribuídos de Controle em Tempo Real". Anais do Congresso Nacional de Automação Industrial.

Lopes, E. (1991). "Prototipagem pra Implementação de Especificações Lotos em um Ambiente de Tempo-Real". Dissertação de Mestrado. FEE/UNICAMP. Abril de 1991.

Lopes, A.B. (1986). "LPM e LCM: Linguagens para Programação e Configuração de Aplicações de Tempo-Real". Dissertação de Mestrado em Sistemas e Computação. Centro de Ciências e Tecnologia. Universidade Federal da Paraíba. Agosto de 1986.

Luqi (1993). "Real-Time constraints in a rapid prototyping language". Computer Languages, 18(2).

Magee, J. (1984). "Provision of Flexibility in Distributed Systems". Ph.D. Thesis. Department of Computing, Imperial College of Science & Technology. London.

Magee, J., J. Kramer and M. Sloman. (1989). "Constructing Distributed Systems in Conic". IEEE Transactions on Software Engineering, 15(6).

Martin, J. (1967). "Design of Real-Time Computer Systems. Series in Automatic Computation". Prentice-Hall.

Melo Jr, A. de, M.F. Magalhães e J.M. Adán-Coello. (1992). "Escalonamento "on-line" de Processos Esporádicos em Sistemas de Tempo Real Crítico". Anais do Congresso Brasileiro de Automática.

Melo Jr, A. de. (1993). "Uma estratégia de escalonamento de processos periódicos e esporádicos em sistemas de tempo-real crítico monoprocessados". Dissertação de mestrado. DCA/FEE, UNICAMP. Fev. 1993.

Meyer, B. (1987). "Reusability: The case for Object-Oriented Design". IEEE Software.

Molesky, L.D. (1989). "Random Graph Generation in an Unix Environment", Technical Report, University of Massachusetts, Sept.

Mok, A. K. (1983). "Fundamental Design Problems of Distributed Systems for The Hard-Real-Time Environment". PhD Thesis, Department of Electrical Engineering and Computer Science. Massachusetts Institute of Technology.

Mok, A. K. (1984). "The Design of Real-Time Programming Systems Based on Process Models". IEEE Real-Time Systems Symposium.

Mok, A.K. (1984b). "The Decomposition of Real-Time System Requirements into Process Models", IEEE Real-Time Systems Symposium.

Motus, L. (1992). "Time Concepts in Real-Time Software". IFAC International Wokshop on Real-Time Programming.

Parnas, D.L. (1972). "On The Criteria to be Used in Decomposing Systems into Modules", Comm. ACM, December.

Peterson, G. E. (1987). "Object-Oriented Computing". IEEE Computer Society Press.

Puscher, P. and C. Koza. (1989). "Calculating the Maximum Execution Time of Real-Time Programas". The Journal of Real-Time Systems, 1, pp. 159-175.

Ragunathan Rajkumar, L. Sha and J. Lehoczky. (1988). "Real-Time Synchonization Protocols for Multiprocessors". Proc. IEEE Real-Time Systems Symposium.

Ramamritham, K. and J. A. Stankovic. (1984). "Dynamic Task Scheduling in distributed hard real-time systems". IEEE Software, 1(5).

Ramamritham, K., J. A. Stankovic and W. Zhao. (1989). "Distributed Scheduling of Tasks with Deadlines and Resource Requirements". IEEE Trans. on Computers, 38(8).

Ramamritham, K. (1990). "Allocation and Scheduling of Complex Periodic Tasks". 10th IEEE International Conference on Distributed Computing Systems.

Ramamritham, K., and J. M. Adán. (1990). "Providing for Dynamic Arrivals During the Static Allocation and Scheduling of Complex Periodic Tasks". COINS Technical Report, University of Massachusetts.

Ramamritham, K., G. Fohler and J.M. Adán. (1993). "Issues in the Static Allocation and Scheduling of Complex Periodic Tasks". 10th IEEE Workshop on Real-Time Operating Systems and Software.

Rzehak, H. (1992). "Real-Time UNIX: What Performance can we Expect ?". IFAC International Wokshop on Real-Time Programming.

Sha, L., and J. B. Goodenough. (April 1990). "Real-Time Scheduling Theory and Ada". IEEE Computer.

Sha, L., J. P. Lehoczky and R. Rajkumar. (1986). "Solutions for Some Practical Problems in Prioritized Preemptive Scheduling". Proc. IEEE Real- Time Systems Symposium.

Sha, L., R. Rajkumar, and J. P. Lehoczky. (1987). "Priority Inheritance Protocols: An Approach to Real-Time Sychronization". Tech. Report CMU-CS-87- 181, Computer Science Department, Carnegie Mellon University.

Sha, L., R. Rajkumar, J. P. Lehoczky and K. Ramamritham (1987b). "Mode Change Protocols for Priority-Driven Preemptive Scheduling". The Journal of Real- Time Systems, Vol. 1, pp. 243-264.

Sprunt, B., L. Sha and J. Lehoczky. (1989). "Aperiodic Task Scheduling for Hard-Real-Time Systems". The Journal of Real-Time Systems, 1, 27-60.

Stankovic, J. A. (1988). "Real-Time Computing Systems: The Next Generation. in Hard Real-Time Systems". IEEE Computer Society Press.

Stankovic, J. A. (1988b). "Misconceptions About Real-Time Computing". IEEE Computer, October.

Stankovic, J. A. and Krithi Ramamritham. (1988). "What is a Real-Time System". in Hard Real-Time Systems. IEEE Computer Society Press.

Stankovic, J. A. and K. Ramamritham. (1990). "Editorial: What is Predictability for Real-Time Systems?". The Journal of Real-Time Systems, 2(4).

Stankovic, J. A. and K. Ramamritham. (1991). "The Spring Kernel: A New Paradigm for Real-Time Systems". IEEE Software. May.

Stankovic, J. A. K. Ramamritham and E. M. Nahum (1991). "Predictable Interprocess Communication for Hard Real-Time Systems". 10th IFAC Workshop on Distributed Computer Computer Control Systems.

Steusloff, H. (1989). "Structures of Automatic Control Systems and The Consequences for Programming Languages". Process Automation.

Stoyenko, A. D., V. C. Hamacher and R. C. Holt. (1991). "Analyzing Hard-Real- Time Programs For Guaranteed Schedulability". IEEE Trans. on Software Engineering. vol. 17. no. 8.

Stroustrup, B. (1986). "The C++ Programming Language". Reading MA: Addison- Wesley.

Stroustrup, B. (1988). "What is object-oriented programming?". IEEE Software, 5(3), 10-20.

Volz, R. A., L. Sha and D. Wilcox. (1991). "Maintaining Global Time in Futurebus+". The Journal of Real-Time Systems, 3 (1).

Wegner, P., and S. B. Zdonik (1988). "Inheritance as an Incremental Modification Mechanism or What Like Is and Isn't Like". In. Lect. Notes in Comp. Sc. 322, pp. 55-78.

Xu, J. and D. L. Parnas. (1990). "Scheduling Processes with Release Times, Deadlines, Precedence, and Exclusion Relations". IEEE Trans. on Software Engineering. 16(3).

Zhao, W. and K. Ramamritham. (1985). "Distributed Scheduling Using Bidding and Focused Addressing". IEEE Real-Time Systems Symposium.

Zhao, W. and K. Ramamritham (1987). "Simple and Integrated Heuristic Algorithms for Scheduling Tasks with Time and Resource Constraints". The Journal of Systems and Software 7, 195-205.

Zhao, W., K. Ramamritham and J. A. Stankovic. (1987). "Preemptive Scheduling Under Time and Resource Constraints". IEEE Trans. on Computers. 36(8).