

**UNIVERSIDADE FEDERAL DE SANTA CATARINA**

**PROGRAMA DE PÓS-GRADUAÇÃO EM ENGENHARIA ELÉTRICA**

**PROJETO E IMPLEMENTAÇÃO DE  
UM NÚCLEO DE TEMPO-REAL  
SEGUNDO A ABORDAGEM SINCRONA**

**DISSERTAÇÃO SUBMETIDA À UNIVERSIDADE FEDERAL DE SANTA CATARINA  
PARA A OBTENÇÃO DO GRAU DE MESTRE EM ENGENHARIA ELÉTRICA**

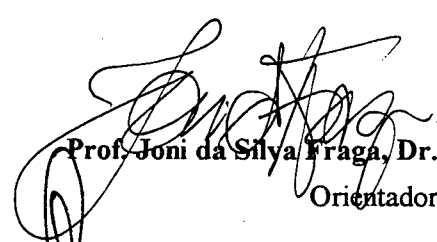
**Alberto Daniel Ferrari**

**FLORIANÓPOLIS, JULHO DE 1994**

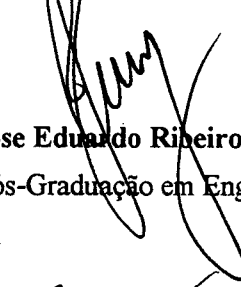
**PROJETO E IMPLEMENTAÇÃO DE UM NÚCLEO DE TEMPO-REAL  
SEGUNDO A ABORDAGEM SINCRONA**

**Alberto Daniel Ferrari**

Esta dissertação foi julgada para a obtenção do título de  
**Mestre em Engenharia**  
especialidade Engenharia Elétrica,  
área de concentração Sistemas de Controle e Automação Industrial,  
e aprovada em sua forma final pelo Curso de Pós-Graduação



Prof. Joni da Silva Fraga, Dr.  
Orientador



Prof. Jose Eduardo Ribeiro Cury, Dr.État.  
Coordenador do Curso de Pós-Graduação em Engenharia Elétrica

Banca Examinadora :



Prof. Joni da Silva Fraga, Dr. (Presidente)  
Orientador



Prof. Jean-Marie Farines, Dr.  
Co-Orientador



Prof. Julius Leite, Ph.D.



Prof. Marcelo Stemmer, Dr.

*A mi familia*

## **Agradecimentos**

*Gostaria agradecer à CAPES, pelo generoso auxílio financeiro,*

*aos meus amigos brasileiros, João, Fernando, Carla, Luiz Otávio, Idmilson,  
e ao Juan, pela alegria e amizade,*

*à TurmaRT (Keiko, Rómulo, Luiz) pelas gostosas discussões de terça à tarde,  
que sem dúvida contribuíram para elevar o nível desta dissertação,*

*aos meus orientadores, Joni e Jean-Marie, pela sua guia e suporte,*

*aos meus pais, pelo apoio permanente que sempre tive deles,*

*ao Brasil, por ter me dado a oportunidade de compartilhar  
estes dois anos inesquecíveis com seu povo,*

*finalmente, a Ana, cujo carinho e compreensão me levaram até aqui.*

# PROJETO E IMPLEMENTAÇÃO DE UM NÚCLEO DE TEMPO-REAL SEGUNDO A ABORDAGEM SINCRONA

## RESUMO

A criticalidade das aplicações de tempo real atuais exige que suas restrições temporais sejam respeitadas; a previsibilidade dos suportes de execução (entre eles, os *núcleos de tempo real* - NTRs) é indispensável para garantir tais restrições.

Este trabalho apresenta uma estrutura para NTRs do tipo reativa e sua implementação baseada na Hipótese de Sincronia [Benveniste91] sobre o modelo da linguagem Esterel. A Hipótese de Sincronia assume que o sistema executa sobre uma máquina infinitamente rápida: esta simplificação fornece vantagens interessantes em relação às abordagens convencionais, entre elas o formalismo das suas linguagens, o determinismo do resultado e o maior nível conceitual da sua descrição. Em consequência, o núcleo implementado é fundamentalmente previsível e sua análise automática fica facilitada.

A inexistência de uma solução geral para o problema de escalonamento nos leva à necessidade de termos suportes suficientemente flexíveis de modo a atender faixas diferentes de escalonamento. Neste sentido, propomos uma metodologia para a especialização de NTRs a partir de estruturas genéricas, segundo as necessidades da aplicação em mãos.

## **A SYNCHRONOUS AND CUSTOMIZABLE REAL-TIME KERNEL**

### **ABSTRACT**

Real-Time applications are recognized by the criticality of their missions, which makes the satisfaction of their temporal needs of foremost concern; predictability of the execution support is necessary in order to guarantee such restrictions.

This work introduces a reactive architecture for a real-time kernel, and its implementation under the Synchronous Hypothesis [Benveniste91] with the Esterel language model. This hypothesis amounts to assume the execution of the system on an infinitely fast machine, which brings up several advantages: the formalism of the approach, the determinism of the final product, and a higher conceptual level. These, in turn, make the resulting kernel predictable and amenable to automatic analysis.

The lack of algorithms to schedule generic task scenarios arises the need to reconfigure the kernel for almost each application. We propose here a methodology that, from a set of general structures, lets the designer customize its executive for the current application's temporal model.

# ÍNDICE

CAPÍTULO 1: INTRODUÇÃO	1
CAPÍTULO 2: O ESCALONAMENTO EM SISTEMAS DE TEMPO REAL MONOPROCESSADOR	4
2.1. O PROBLEMA DE TEMPO REAL	5
2.2. MODELO TEMPO-REAL DE TAREFAS	7
2.2.1. O ESCALONADOR	9
2.2.2. TIPOS DE ALGORITMOS DE ESCALONAMENTO	10
2.3. ALGORITMOS DE ESCALONAMENTO TRADICIONAIS	11
2.3.1. ESCALONAMENTO ESTÁTICO	12
2.3.1.1. <i>Executivo Cíclico</i>	12
2.3.1.2. <i>Taxa Monotônica</i>	13
2.3.2. ESCALONAMENTO DINÂMICO	15
2.3.2.1. <i>Earliest Deadline</i>	16
2.3.2.2. <i>Menor Folga</i>	17
2.3.2.3. <i>Bin-Packing</i>	18
2.3.3. ALGORITMOS MISTOS - MUF	19
2.4. CONSIDERAÇÕES GERAIS	20
2.5. CONCLUSÃO	23
CAPÍTULO 3: SUPORTES DE EXECUÇÃO PARA APLICAÇÕES DE TEMPO REAL	24
3.1. ABORDAGENS DE IMPLEMENTAÇÃO EVENT-TRIGGERED VS TIME-TRIGGERED	25
3.1.1. ABORDAGEM EVENT-DRIVEN: SPRING	25
3.1.1.1. <i>O Escalonamento no Spring</i>	26
3.1.2. ABORDAGEM TIME-DRIVEN: MARS	27
3.1.2.1. <i>Arquitetura do Hardware</i>	27
3.1.2.2. <i>Arquitetura do Software</i>	28
3.1.2.3. <i>Método de Projeto</i>	28

<b>3.2. ARQUITETURAS MONOLÍTICA VS. MICRONÚCLEO .....</b>	<b>29</b>
3.2.1. SISTEMAS UNIX PARA TEMPO REAL .....	29
3.2.1.1. <i>O Padrão Posix</i> .....	29
3.2.1.2. <i>Sistemas Unix Comerciais</i> .....	30
3.2.2. SO BASEADOS EM MICRONÚCLEO: CHORUS .....	30
3.2.2.1. <i>Arquitetura Geral</i> .....	31
<b>3.3. SUPORTES ORIENTADOS A OBJETOS.....</b>	<b>32</b>
3.3.1. MODELO GERAL .....	32
3.3.2. PROTOCOLO TIME-FENCE: ARTS .....	33
3.3.3. ADAPTABILIDADE: CHAOS .....	33
3.3.4. GERENCIAMENTO COOPERATIVO: R-SHELL .....	34
<b>3.4. CONSIDERAÇÕES GERAIS .....</b>	<b>34</b>
<b>3.5. CONCLUSÃO .....</b>	<b>37</b>

## CAPÍTULO 4: CONCEPÇÃO DE UM NÚCLEO DE TEMPO-REAL SEGUNDO A ABORDAGEM SÍNCRONA 39

<b>4.1. SISTEMAS REATIVOS .....</b>	<b>40</b>
<b>4.2. A ABORDAGEM SÍNCRONA DE IMPLEMENTAÇÃO.....</b>	<b>41</b>
4.2.1. LINGUAGENS SÍNCRONAS .....	41
4.2.2. A LINGUAGEM ESTEREL .....	43
4.2.3. ARQUITETURA DE UMA APLICAÇÃO ESTEREL .....	43
<b>4.3. UM NÚCLEO DE TEMPO REAL SEGUNDO A ABORDAGEM SÍNCRONA .....</b>	<b>44</b>
4.3.1. ESPECIFICAÇÃO DE UM NÚCLEO DE TEMPO REAL .....	45
4.3.2. O NÚCLEO ESPECIFICADO VISTO COMO SISTEMA REATIVO .....	47
4.3.3. MODELO SÍNCRONO DE IMPLEMENTAÇÃO DA ENTIDADE REATIVA .....	49
4.3.4. MODULARIZAÇÃO DA ENTIDADE REATIVA .....	50
<b>4.4. CONSIDERAÇÕES GERAIS .....</b>	<b>55</b>
<b>4.5. CONCLUSÃO .....</b>	<b>55</b>



## CAPÍTULO 5: ASPECTOS PRÁTICOS DO DESENVOLVIMENTO DO NÚCLEO

<b>5.1. ASPECTOS DE IMPLEMENTAÇÃO .....</b>	<b>57</b>
5.1.1. AMBIENTE DE DESENVOLVIMENTO ESTEREL .....	57
5.1.2. SUPORTE BINÁRIO .....	58
5.1.3. INTEGRAÇÃO DO ELEMENTO REATIVO .....	58
5.1.3.1. <i>Conversão do Suporte para Tempo Real</i> .....	59
5.1.3.2. <i>"Sincronização" do Suporte</i> .....	60
5.1.4. VALIDAÇÃO DA HIPÓTESE DE SINCRONIA.....	62
5.1.5. FASES DO PROJETO.....	62
<b>5.2. METODOLOGIA DE GERAÇÃO DE NÚCLEOS ESPECÍFICOS ...</b>	<b>65</b>
5.2.1. PARTICULARIZAÇÃO POR SELEÇÃO DE COMPORTAMENTOS INTERNOS .....	65
5.2.2. PARTICULARIZAÇÃO POR SELEÇÃO DA POLÍTICA DE ESCALONAMENTO.....	66
<b>5.3. CONSIDERAÇÕES GERAIS .....</b>	<b>69</b>
<b>5.4. CONCLUSÃO .....</b>	<b>70</b>

CAPÍTULO 6: RESULTADOS E CONCLUSÕES FINAIS 71

REFERÊNCIAS BIBLIOGRÁFICAS 73

ANEXO 1: CODIGO ESTEREL DO NÚCLEO

ANEXO 2: EXEMPLO DE ESCALONAMENTO

# CAPÍTULO 1

## INTRODUÇÃO

Aplicações de tempo real são caracterizadas pela imposição de restrições temporais no comportamento do *elemento computacional de controle*, por parte do *ambiente* que se deseja controlar. A criticalidade destas aplicações faz necessário garantir *a priori* o cumprimento de tais restrições, desde que sua inobservância pode trazer conseqüências graves de tipo humano, ecológico e/ou económico ao conjunto.

Para garantir tais restrições, é preciso que o sistema seja *previsível*, isto é, que seu comportamento possa ser deduzido a partir de seus atributos [Stank88]. Esta condição estende-se a cada um dos componentes do sistema, e em especial aos seus comportamentos temporais. Na literatura, esta noção de previsibilidade tem sido associada à antecipação determinista [Kopetz89] ou probabilista [Stank91] [Jensen85] do comportamento do sistema: nos dois casos existe a necessidade de um conhecimento ou estimativa previa de suas características temporais.

Dentre as abordagens mais comuns à implementação do elemento de controle<sup>1</sup> para o caso monoprocessador está a decomposição da aplicação num conjunto de tarefas sequenciais, rodando pseudo-concorrentemente sobre um suporte de execução multitarefa e interagindo através dos recursos por ele oferecidos, acessíveis pelas suas *primitivas*. Neste caso, o suporte ou *núcleo de tempo real* (NTR) deve ser previsível, de maneira a fornecer um embasamento adequado para a análise do escalonamento da CPU entre as tarefas, visando garantir o cumprimento de suas restrições temporais.

Em geral, o escalonamento nos suportes convencionais é baseado na atribuição subjetiva de "importâncias" (prioridades) às tarefas por parte do projetista: o não-determinismo do comportamento resultante complica os testes de garantia e dificulta a reprodução de cenários problemáticos para sua

---

<sup>1</sup> existem outras, vide [Benveniste91]

depuração. Esta imprevisibilidade impede também o cálculo preciso dos tempos máximos de execução de suas primitivas, fator fundamental para a aplicação dos testes de escalonabilidade.

Este trabalho apresenta uma estrutura para núcleos de tempo real do tipo *reativa* e a implementação de um protótipo de NTR baseado na Hipótese de Sincronia [Benveniste91], usando o modelo da linguagem Esterel [Berry88] [Boussinot91]. A Hipótese de Sincronia assume que o sistema executa sobre uma máquina infinitamente rápida, simplificação que fornece vantagens interessantes em relação às abordagens convencionais: o formalismo das suas linguagens, o determinismo do resultado e o maior nível conceitual da sua descrição. Em consequência, o núcleo implementado é fundamentalmente previsível e sua análise automática fica facilitada.

Por outro lado, devido à complexidade NP-completa do problema de escalonamento em circunstâncias realistas (tarefas com restrições de recursos, relações de precedência, etc) [Mok84], não existem na literatura [Audsley90] [Cheng88] políticas de escalonamento para tempo real suficientemente gerais que consigam escalonar todas as situações possíveis com um custo de execução razoável. Isto faz com que as soluções viáveis fiquem restritas a aplicações de modelos temporais relativamente simples, decorrentes das premissas usadas para a derivação dos testes de escalonabilidade das políticas envolvidas.

Por exemplo, o algoritmo *taxa monotônica* (*Monotonic Rate*) [Liu73] no sistema ARTS [Tokuda89] é usado para escalonar conjuntos de tarefas independentes e periódicas, embora estendido para o tratamento de restrições de recursos com o *Protocolo Limite de Prioridade* (*Priority Ceiling Protocol*) [Sha88]. Já os sistemas Mars [Kopetz89] e Spring [Stank91] utilizam técnicas heurísticas com enumeração implícita de soluções para o escalonamento de tarefas críticas, técnicas que embora abrangem modelos mais gerais, têm tempos de execução pseudo-exponenciais que não as fazem viáveis ou então devem ser aplicadas apenas de forma sub-ótima.

O mesmo acontece com os mecanismos internos ao NTR: existe uma relação de compromisso entre o nível das primitivas e sua eficiência na implementação, que deve ser resolvida segundo a aplicação. Aplicações como controle direto de robôs precisam de primitivas simples porém rápidas, pelo dinamismo do ambiente em que o sistema é embutido. Já sistemas de maior nível hierárquico assumem folgas maiores para trabalhar, e portanto podem oferecer primitivas mais sofisticadas e escalonamentos mais apurados para o conjunto. Mars, no entanto, resolve estes problemas escalonando tudo antes da execução (ainda a comunicação e sincronização entre tarefas, e alocação de recursos): isto fornece uma grande eficiência na execução, mas sobre uma estrutura notoriamente menos flexível que as anteriores (pelo menos dinamicamente).

Portanto, a inexistência de algoritmos de escalonamento gerais e de estruturas internas

suficientemente flexíveis ao mesmo tempo que eficientes nos leva à necessidade de termos suportes configuráveis para atender diferentes faixas de escalonamento. Neste sentido, propomos uma metodologia para a especialização de NTRs a partir de estruturas genéricas, segundo as necessidades da aplicação em mãos. Esta metodologia, baseada na composição de autômatos segundo o modelo da linguagem Esterel, permite escolher o escalonador do NTR baseado na análise do modelo temporal da aplicação, e integrá-lo ao núcleo final. Da mesma forma, pode-se escolher entre diversos comportamentos para a mesma funcionalidade, ou ainda implementações distintas do mesmo comportamento, que privilegiam a eficiência por sobre a memória ocupada, etc. Assim, o usuário confrontado à tarefa de programar várias aplicações com modelos temporais distintos não precisaria mudar de NTR, mas poderia conformar *núcleos específicos* para o modelo alvo, sem sacrificar as características principais do NTR que já possui.

Esta dissertação é organizada como segue. No capítulo 2 é discutido o problema de tempo real em geral, junto com um modelo de tarefas tempo-real que servirá para analisar, a continuação, algumas das políticas de escalonamento mais usadas na prática: taxa monotônica, earliest deadline, menor folga, bin-packing e máxima urgência. O capítulo conclui fazendo uma comparação dos algoritmos cobertos.

O capítulo 3 trata suportes de execução e sistemas operacionais com características modernas, descrevendo as principais abordagens de implementação (*event-driven vs time-driven*), critérios de estrutura (*monolítica vs micronúcleo*) e finalmente analisando um modelo de objetos para tempo real e as qualidades salientes de algumas de suas técnicas.

O capítulo 4 apresenta o conceito de Sistema Reativo e descreve a abordagem síncrona para sua implementação. Depois é apresentado o projeto de um NTR simples usando a abordagem síncrona, junto com nossa proposta de implementação e descrição dos seus módulos internos.

O capítulo 5 descreve aspectos relacionados com a implementação de um protótipo do núcleo anterior concretizada no LCMI. Descreve-se o ambiente da linguagem Esterel utilizado, e como um suporte convencional foi adaptado para este fim. O capítulo termina propondo uma metodologia de particularização de NTRs para aplicações específicas.

Finalmente, o capítulo 6 apresenta as conclusões e resultados finais do trabalho, e algumas propostas de extensão para o núcleo implementado.

# CAPÍTULO 2

## O ESCALONAMENTO EM SISTEMAS DE TEMPO REAL MONOPROCESSADOR

Sistemas de Tempo Real ditos "Hard" caracterizam-se pela ocorrência de conseqüências graves ao conjunto, caso seus componentes "críticos" não cumpram determinadas restrições lógicas e temporais, impostas pelo meio externo. Aplicações para seu controle rodam sobre suportes computacionais especiais.

O escalonador é o modulo, dentro destes suportes, encarregado de distribuir no tempo os recursos do sistema entre as tarefas que pertencem à aplicação, de modo às "críticas" cumprirem suas restrições temporais. No caso de suportes monoprocessadores, a única CPU deve ser compartilhada entre todos os processos.

Neste capítulo analisa-se o conceito de tempo real e as características desejáveis neste tipo de sistema, junto à introdução de um modelo temporal de tarefas que definirá termos a serem usados ao longo desta dissertação. A seguir, é analisado o conceito de escalonador e os algoritmos de escalonamento monoprocessador mais significativos, junto a uma avaliação comparativa e considerações finais.

**-The righth answer late is wrong**

## 2.1. O PROBLEMA DE TEMPO REAL

Robótica, controle de processos, supervisão de plantas químicas e nucleares, aviónica, sistemas embarcados em satélites, sistemas de ignição eletrônica e suspensão ativa para automóveis, controle de tráfego aéreo. O rastreamento de uma nave espacial, na sua órbita, por uma antena. Células autônomas de manufatura. Estas aplicações são alguns exemplos de *Sistemas de Tempo Real*, sistemas caracterizados por exigirem correção temporal às ações do seu elemento de controle, além da necessária correção lógica. Seu comportamento satisfatório depende não apenas do controle aplicado, mas também do tempo em que é fornecido, esta dependência temporal sendo determinada pelo ambiente a controlar em cada caso. O não cumprimento das restrições temporais pode acarretar conseqüências severas de tipo humanas, ambientais e/ou econômicas.

Um Sistema de Tempo Real pode ser visto como composto de duas partes: uma *parte física* (um metrô, uma planta química, ou uma rede de comutação telefônica) e um *computador de controle*, fortemente acoplados (Figura 2.1). Por precisar de correção temporal, ele pode ser comparado a um casal de dançarinos, onde cada deve reagir aos movimentos do outro, e ambos devem levar em conta uma agenda temporal (no caso, o ritmo da música) [Lauber89].

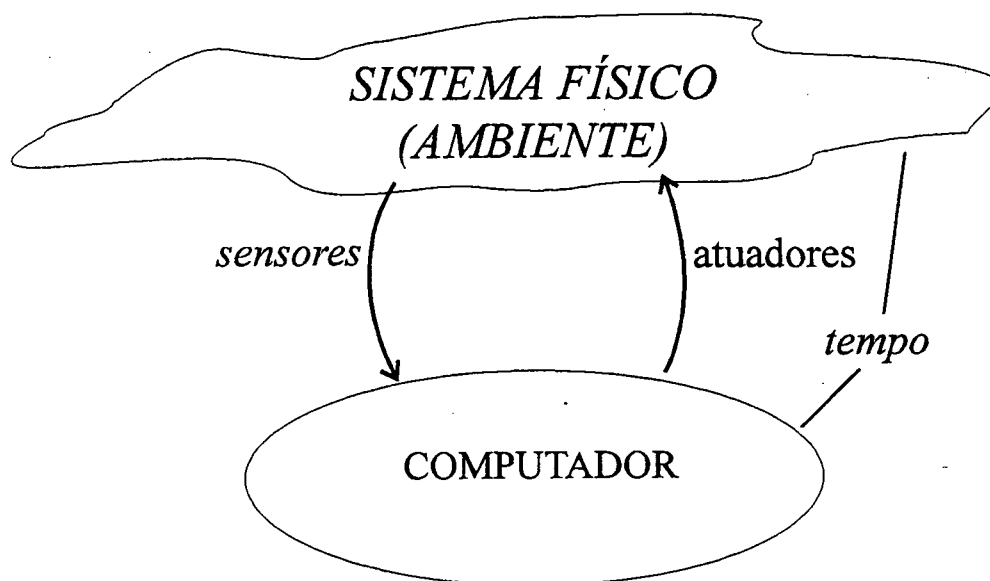


Figura 2.1: Modelo de Sistema de Tempo-Real

Geralmente, a lógica de comportamento do computador de controle é decomposta num conjunto de tarefas que executam pseudo-concorrentemente sobre um suporte adequado (*núcleo de tempo real*). Estas tarefas podem apresentar restrições temporais na sua execução, tipicamente na forma de "deadlines". No contexto deste trabalho, assumiremos o computador de controle como sendo monoprocessador: tem apenas uma CPU.

O cumprimento do deadline de uma tarefa **hard** é considerado fundamental para a operação do sistema, e portanto só deve executar se for garantidas de antemão. Se for apenas desejável e um eventual descumprimento não trazer conseqüências graves, a tarefa é dita **soft**.

Segundo [Stan90], as tarefas **hard** são aquelas que não fornecem benefício algum ao sistema quando completadas depois do seu "deadline". São chamadas **críticas** se, além disso, puderem acontecer conseqüências graves ao sistema caso não forem cumpridas suas restrições temporais. A importância das tarefas críticas obriga a alocação temprana de todos os recursos necessários para sua execução. Na visão de [Stan90], muitos sistemas tratam as tarefas **hard** como críticas quando, de fato, só uma pequena parte delas é verdadeiramente crítica: isto faz com que os sistemas fiquem superdimensionados.

As características desejáveis de um sistema de tempo real são [Sha90] [Sha93]:

- **previsibilidade**: garantia do cumprimento dos deadlines das tarefas críticas, e tempos de resposta mínimos para o resto.
- alto grau de **escalabilidade** para tarefas **hard**. A escalabilidade é o máximo nível de utilização de recursos para o qual o cumprimento dos "deadlines" propostos é garantido.
- **estabilidade** frente a sobrecargas de cômputo transitórias: quando o sistema é sobrecarregado por eventos imprevistos, nem todas as tarefas continuarão garantidas. No entanto, determinadas tarefas críticas devem continuar a cumprir seus "deadlines".

A mais importante de todas é a **previsibilidade temporal**: deve ser possível demonstrar que os requerimentos temporais das tarefas críticas serão cumpridos, sujeitos às premissas feitas (por exemplo, em relação a falhas e sobrecargas). Esta previsibilidade baseia-se na avaliação das propriedades temporais das tarefas individuais e do sistema, e determina implicações em todos os níveis do conjunto: características arquitetônicas previsíveis facilitam a construção de software operacional previsível, que por sua vez leva à construção de aplicações previsíveis.

A arquitetura do sistema deve ser capaz de *fornecer uma previsão provável*<sup>1</sup> de sua capacidade de cumprir as restrições temporais impostas. O requerimento é que a previsibilidade possa ser deduzida a partir das características estruturais do sistema, mas não é preciso que uma prova seja feita. Então, a *provabilidade* implica que as técnicas de projeto devem ser derivadas de modelos de execução formais, de maneira a produzir resultados *prováveis* dentro das restrições do modelo [Locke92].

O protocolo limite de prioridade (*priority ceiling protocol*) [Sha90] é um bom exemplo de um protocolo previsível, mas não determinístico: embora não precise o tempo que uma tarefa de alta prioridade vai ficar bloqueada por uma de menor prioridade, garante um patamar máximo para a duração desse bloqueio. Outro exemplo é o algoritmo de escalonamento *bin-packing*: nele não sabemos a priori quando será executada uma tarefa, mas temos certeza que cumprirá seu "deadline".

## 2.2. MODELO TEMPO-REAL DE TAREFAS

Tarefas de tempo real são caracterizadas, então, pela existência de restrições temporais para sua execução, além das restrições de correção lógica e requerimentos de recursos e precedência. Os *requerimentos de recursos* são a descrição, a cada momento da execução do programa, dos recursos necessários para seu desenvolvimento. Os de *precedência* descrevem as relações [antecessor, sucessor] que devem ser respeitadas na ordem de execução das tarefas. Finalmente, as *restrições temporais* são descritas por uma tupla  $T(a_t, r_t, c_t, d_t, p_t, w)$ <sup>2</sup>, definida pelo projetista para cada tarefa (Figura 2.2).

O **tempo de chegada**  $a_t$  (*arrival time*) é o instante em que a tarefa se incorpora ao conjunto de tarefas corrente do sistema, onde o **peso**  $w$  mede sua importância relativa. O **tempo de pronto**  $r_t$  (*ready time*) é o tempo a partir do qual a tarefa já pode ser executada. O **tempo de computação**  $c_t$  (*execution time* ou *computation time*) é o tempo de CPU que a tarefa precisa para executar seu código; quando uma tarefa executa por um tempo maior que o tempo de cômputo atribuído, diz-se que *sobrecarrega* o sistema. O **deadline**  $d_t$  é o limite de tempo antes do qual a tarefa deve completar sua execução; quando respeitado, diz-se que a tarefa *cumpriu* seu deadline. O deadline pode ser relativo ao tempo de chegada, que é absoluto [Cheng88] [Ausdley90].

<sup>1</sup> No contexto, *provável* é a capacidade de ser provado logicamente. Este conceito é distinto de *determinismo*, qualidade que indica que a seqüência e tempos de execução das tarefas são completamente predeterminados e fixos para toda a execução do sistema. Embora o determinismo é suficiente para o sistema ser previsível, não é necessário.

<sup>2</sup> Os nomes dos seus componentes e suas interpretações podem variar dependendo do autor e do modelo específico de tarefa sendo tratado [Schwan92] [Xu90].



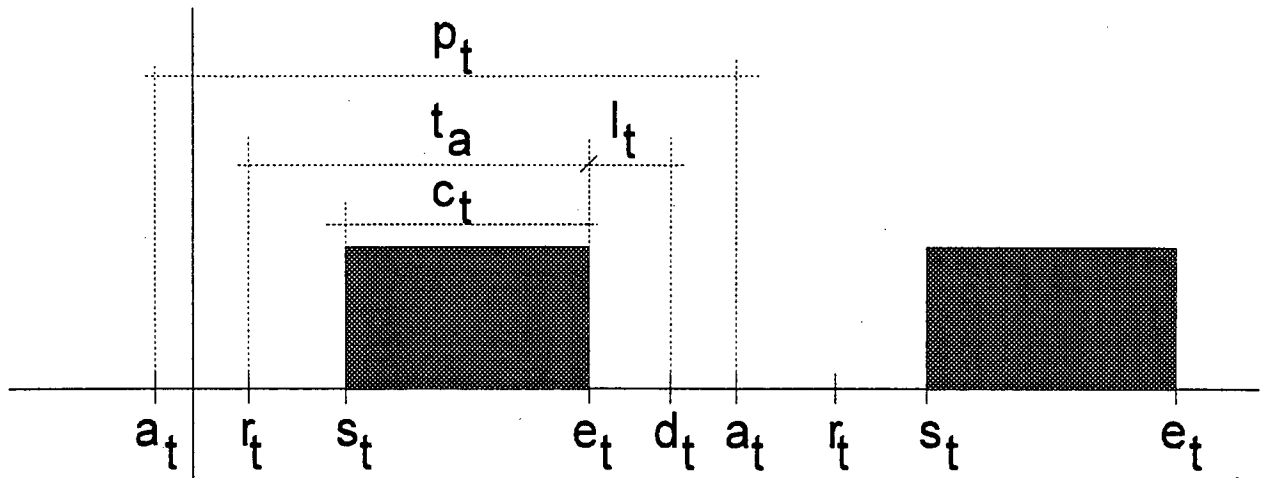


Figura 2.2: Parâmetros temporais de uma tarefa

Outras figuras temporais são determinadas no escalonamento do conjunto de tarefas e não pelo usuário. Define-se **tempo de começo**  $s_t$  (*start time*) de uma tarefa como o instante em que obtém a CPU pela primeira vez; o **tempo de término**  $e_t$  (*completion time* ou *end time*) é o tempo no qual a tarefa termina sua execução. O **tempo de resposta**  $t_a$  é a diferença entre o tempo de término e o tempo de pronto [Leinb80]. O **atraso de uma tarefa**  $l_t$  (*lateness*) é a diferença entre o tempo de término e seu "deadline" absoluto. Definiremos o **atraso de um escalonamento** como o máximo atraso dentre as tarefas da agenda [Xu90].

Quando o tempo  $p_t$  entre invocações sucessivas de uma tarefa é regular, ela é dita **periódica** de período  $p_t$ . Em geral, as *instâncias* de tarefas periódicas apresentam "deadlines" hard que coincidem com o final do período corrente. Tarefas com períodos múltiplos ou submúltiplos entre si são ditas **harmônicas**. Sua fase  $b_i$  define-se como o tempo de pronto da sua primeira requisição; diz-se que um conjunto de tarefas está *em fase* se todas têm o mesmo  $b_i$  [Shih91].

$$\begin{aligned}
 c_{tp} &= c_{ta} \\
 c_{ta} &\leq d_{tp} \leq d_{ta} \\
 p_{tp} &\leq \min(d_{ta} - d_{tp}, a_{min})
 \end{aligned}$$

Figura 2.3: Regras para a transformação de uma tarefa aperiódica numa periódica equivalente

Tarefas que acontecem apenas uma vez no sistema são ditas **one-shot**. Tempo de chegada irregular é característico de tarefas **aperiódicas** (com deadlines soft) e **esporádicas** (com deadlines

hard). Para o processamento das tarefas esporádicas é necessário modelá-las como respeitando um intervalo de tempo mínimo entre chegadas  $a_{\min}$ . Tarefas aperiódicas ( $c_{ta}, d_{ta}, a_{\min}$ ) podem ser transformadas em *periódicas equivalentes* ( $c_{tp}, d_{tp}, p_{tp}$ ) satisfazendo as condições da Figura 2.3 [Mok84].

Tarefas que não precisam sincronizar com outras para evoluir são ditas **independentes**; tarefas **interdependentes** interagem de muitas maneiras, incluindo relações de exclusão e precedência.

### 2.2.1. O ESCALONADOR

Sistemas de Tempo Real precisam de suportes computacionais como *núcleos multitarefa* para executar, que deverão levar em conta as particularidades mencionadas acima. Uma parte fundamental desses suportes é o **escalonador**, elemento que tem a função de coordenar o acesso das tarefas aos recursos do sistema. O escalonamento para Tempo-Real está referido, em geral, ao seqüenciamento das tarefas no acesso a qualquer recurso (por exemplo, a CPU) cujo uso implique a satisfação de restrições temporais.

O escalonador é ativado em instantes chamados **pontos de escalonamento** para executar um algoritmo ou política que ordene o acesso aos recursos. O **despachante** é o módulo que, após cada ponto de escalonamento, coloca a rodar a tarefa escolhida pelo escalonador. Para um dado **conjunto de tarefas** o escalonador produz um **escalonamento, plano, ou agenda** que é, basicamente, um mapeamento de tarefas em segmentos de tempo de CPU. Se existir um escalonamento que cumpre as precondições dadas, o conjunto de tarefas é dito **escalonável, viável, factível** ou **possível**. O algoritmo de escalonamento deve, então, fornecer critérios para a análise da viabilidade do conjunto -existe uma agenda?- e se afirmativo, o método para achá-la.

Os algoritmos de escalonamento baseiam explicitamente suas decisões nas informações temporais das tarefas e nas restrições do sistema, e estão caracterizados pelo perfil temporal de tarefas que suportam. Se o algoritmo faz com que uma tarefa cumpra suas restrições, diz-se que *garante* a tarefa. A *eficácia*<sup>3</sup> de um algoritmo de escalonamento é medida com referencia à **taxa de garantia**, percentagem dos conjuntos possíveis gerados aleatoriamente que consegue garantir, para um modelo determinado do sistema a controlar. O principal objetivo no projeto de um algoritmo é maximizar sua taxa de garantia para tarefas críticas. A *eficiência* do algoritmo está relacionada com medidas de desempenho do mesmo, como sobrecarga por escalonamento, sua complexidade computacional, etc.

---

<sup>3</sup> qualidade das decisões, segundo a acepção de Ciências da Computação

Deve ficar claro que as heurísticas usadas pelos algoritmos para o escalonamento das tarefas (p.ex., executar primeiro as de "deadlines" mais próximos) são diferentes dos seus testes de aceitabilidade. Embora possam ser aplicadas em quase a totalidade dos modelos, as heurísticas (e portanto os algoritmos) apresentam testes de garantia apenas para modelos temporais específicos. A pesquisa corrente é em direção a expandir esses modelos para que testes de escalonamento possam ser aplicados em ambientes mais realistas, isto é, com menos restrições.

## 2.2.2. TIPOS DE ALGORITMOS DE ESCALONAMENTO

Os algoritmos de escalonamento podem ser estáticos ou dinâmicos. Na abordagem **estática**, **pré-run-time** ou **off-line**, as decisões são tomadas antes da execução, gerando uma **tabela de expedição** para o despachante em tempo de compilação ou de configuração. A tabela contém todas as informações que o despachante precisa para ativar, a cada ponto da base de tempo discreta, a próxima tarefa a executar. A abordagem **dinâmica**, **run-time** ou **on-line**, toma as decisões em tempo de execução, baseando-se nas requisições correntes para serviço. Existem ainda abordagens **mistas**, que tentam capturar as vantagens das duas anteriores<sup>4</sup>.

Os algoritmos **estáticos** precisam de conhecimento completo e *a priori* do conjunto de tarefas a escalonar. Isto justifica-se pelo fato de que, na maioria das aplicações de tempo real, a parte principal da computação é por conta de tarefas periódicas, onde o sequenciamento e características temporais das tarefas são conhecidas de antemão [Xu93]. Abordagens estáticas também não conseguem dar conta de *grupos de tarefas dependentes*<sup>5</sup> [Stan91], e em geral de qualquer cenário do qual não se disponha da totalidade das informações.

A abordagem **dinâmica** tem custo maior em tempo de execução para calcular o escalonamento, que é progressivo. Considera apenas as informações relativas à(s) tarefa(s) requisitada(s), podendo ser aplicada estáticamente como um caso especial. Nestes caso, uma tarefa é considerada **escalonável** se pode ser escalonada para cumprir seu "deadline" de modo que as outras tarefas previamente aceitas permaneçam garantidas. A abordagem dinâmica é mais flexível para se adaptar a um cenário que evolui ou na presença de faltas, podendo tomar decisões mais inteligentes ao contar com informação atualizada sobre o estado do sistema e dos seus componentes.

---

<sup>4</sup> No caso de um escalonador baseado em prioridades (onde em todo momento executa-se a tarefa pronta de maior prioridade), a qualidade de estático ou dinâmico refere-se ao fato das prioridades serem fixas, ou mudarem durante a sua execução.

<sup>5</sup> onde os requerimentos de tempo e recursos de um subconjunto das tarefas só podem ser conhecidos depois que todas as tarefas precedentes foram executadas

Um algoritmo estático é **ótimo** se, para qualquer conjunto de tarefas, sempre produz uma agenda possível quando outro algoritmo que suporta o mesmo modelo temporal consegue fazê-lo<sup>6</sup>. De forma semelhante, um algoritmo dinâmico é **ótimo** se uma tarefa determinada como não-escalonável por ele também é rejeitada por qualquer outro algoritmo [Schwan92]. O problema geral do escalonamento para modelos temporais complexos é difícil e computacionalmente intratável, pelo que na prática são utilizados algoritmos sub-ótimos.

Algoritmos de escalonamento **estáveis** tem comportamento previsível frente a sobrecargas de processamento e conseguem garantir um subconjunto de suas tarefas críticas para um nível determinado de sobrecarga [Sha90]: tarefas fora do **conjunto crítico** podem perder seu "deadline" em caso de sobrecarga.

Finalmente, o algoritmo pode ser **preemptivo** ou não. No primeiro caso a tarefa corrente pode ser interrompida se outra mais urgente requerer serviço; a preempção só pode acontecer quando não forem violadas determinadas condições (por exemplo, restrições de exclusão mútua). No escalonamento **não preemptivo** ou **cooperativo**, a tarefa corrente não pode ser interrompida, sendo ela mesma quem decide quando liberar seus recursos associados -normalmente depois de completar a execução-. A não-preempção é razoável num cenário onde muitas tarefas pequenas (de tempo de execução curto, em relação ao tempo que leva uma mudança de contexto) devem ser executadas.

## 2.3. ALGORITMOS DE ESCALONAMENTO TRADICIONAIS

Na continuação serão analisados os algoritmos de escalonamento monoprocessador mais tradicionais, no sentido de serem os primeiros a formar parte do corpo da teoria de sistemas de tempo real, e provavelmente também os mais estudados e aplicados na prática. Para sua apresentação os classificaremos em estáticos, dinâmicos e mistos, segundo a abordagem utilizada no seu funcionamento.

Muitas destas abordagens tem a desejável propriedade de separar a correção lógica do programa de sua correção temporal: enquanto as condições de escalonabilidade sejam satisfeitas, todas as tarefas do conjunto cumprirão seus "deadlines" sem que o programador saiba exatamente quando vai executar cada uma [Sha90].

---

<sup>6</sup> Esta definição não pode ser utilizada em sistemas dinâmicos, onde as tarefas chegam aleatoriamente e portanto nem todas as suas características são conhecidas de antemão.

## 2.3.1. ESCALONAMENTO ESTÁTICO

A abordagem estática tem uma série interessante de vantagens, entre elas a eficiência (precisa de poucos recursos em tempo de execução). A seguir são descritos seus maiores expoentes: o executivo cíclico e o algoritmo de taxa monotônica. Este último é o principal algoritmo de escalonamento estático para tarefas independentes e periódicas, apresentando extensões para o tratamento de recursos e tarefas não periódicas.

### 2.3.1.1. EXECUTIVO CÍCLICO

O **executivo cíclico** é uma estrutura de controle para a alternância de execução de várias tarefas sobre uma única CPU; esta alternância é determinista, de modo que seu comportamento temporal (*timing*) é previsível [Baker89]. Aplica-se principalmente para tarefas periódicas, embora possa-se estender a tarefas aperiódicas com servidores adequados.

O executivo cíclico divide cada tarefa periódica numa sequência de códigos não preemptíveis. O *timeline* ou *agenda cíclica* especifica a ordem (fixa ao longo da execução de um programa) em que estes segmentos serão executados, sendo repetido indefinidamente. O tempo para desenvolver a agenda cíclica é chamado *ciclo maior*, e é igual ao mínimo múltiplo comum dos períodos das tarefas envolvidas. Cada agenda cíclica é composta de *ciclos menores* ou *quadros*, de duração fixa; existem vários critérios para a escolha do tamanho dos quadros [Baker91], devendo sempre ser maior que a duração da menor tarefa do sistema.

A Condição 2.1 mostra a **condição suficiente de escalonabilidade**, que define um patamar máximo para a utilização da CPU ( $U$ ), por um número  $n$  de tarefas do sistema. Os pontos de escalonamento são os finais de cada ciclo menor, e portanto facilmente controláveis com um temporizador de hardware. Se no final do último ciclo menor de uma tarefa o sistema não estiver sobrecarregado, a CPU estará *idle* ou executando uma tarefa em background. Caso contrário, existe uma situação de sobrecarga de processamento cuja recuperação é difícil: se continuar executando a tarefa corrente tem-se o risco de provocar uma falha temporal num outro ponto (imprevisível) do sistema; se for abortada, pode-se criar um estado inconsistente nos dados persistentes da tarefa corrente e que compartilha com outras.

O executivo cíclico precisa que todas as tarefas sejam harmônicas para ter agendas cíclicas de cumprimento razoável. Se não for o caso, pode-se aumentar a frequência de determinadas tarefas, modificar o tamanho do ciclo menor, ou ambas. O problema é que o desempenho global é prejudicado

pelo incremento artificial na utilização do processador imposto por esta solução, sem um aumento na funcionalidade original. Em todo caso, a estrutura do programa é sacrificada para acomodar o código nos *slots* de tempo certos e portanto, a aplicação resultante é *frágil*: qualquer mudança no sistema obrigará a uma reestruturação general do conjunto, frente à possibilidade de efeitos temporais laterais nas outras tarefas pelas modificações introduzidas.

Este método apresenta um *jitter*<sup>7</sup> muito baixo, útil para determinadas aplicações: laços de realimentação em controle de processos ou aquelas que dependem de dispositivos especializados cujas entradas e saídas devem apresentar relações temporais precisas.

Em resumo, o executivo cíclico é um método manual que apresenta poucas vantagens. É recomendável apenas para aplicações com requerimentos de *jitter* estreitos e tarefas harmônicas, e onde as soluções do esquema de prioridade fixa não sejam satisfatórias [Locke92]. Sua aplicação aumenta os custos a nível de ciclo de vida do software final (sobretudo de manutenção e teste).

### 2.3.1.2. TAXA MONOTÔNICA

O clássico **taxa monotônica** (TM) tem sido utilizado como base para desenvolver uma teoria de escalonamento para tempo real (*Generalized Rate-Monotonic Analysis*) que soluciona um amplo espectro de problemas práticos [Klein93]. Taxa Monotônica é um algoritmo preemptivo, que atribue prioridades fixas (maior quanto menor o período) para um conjunto de tarefas periódicas e independentes: em todo momento o sistema executa a tarefa pronta de maior prioridade. No seu modelo temporal, o período de cada tarefa é constante e seu "deadline" é o final do período corrente, isto é: precisa completar antes da chegada da próxima instância. A tarefa é assumida pronta ao chegar e o tempo de troca de contexto é desprezado; as tarefas não se suspendem voluntariamente [LiuC73]. Fica claro que este modelo temporal é bem restritivo.

$$U \leq n(2^{1/n} - 1) \quad [1]$$

onde  $U$  é a taxa de utilização total da CPU

$$U = \sum_i U_i = \sum_{i=1}^n \frac{C_{ti}}{P_{ti}} \quad [2]$$

e  $C_{ti}$  e  $P_{ti}$  são os tempos de execução e período da tarefa  $i$

Condição 2.1: Condição suficiente de escalonabilidade para Taxa Monotônica

<sup>7</sup> Jitter é a variação no tempo, de ciclo em ciclo, com que um resultado é apresentado ao ambiente externo.

A Condição 2.1 mostra a **condição suficiente de escalonabilidade**, que define um patamar máximo para a utilização da CPU ( $U$ ), por um número  $n$  de tarefas do sistema. Este patamar converge para 69% quando  $n$  tende a infinito: portanto, se o conjunto das tarefas apresentar uma utilização conjunta do processador menor que 69% tem o escalonamento garantido por este algoritmo. Este é um valor pessimista, calculado para o pior conjunto de tarefas possível: para um conjunto escolhido aleatoriamente, o limite provável é de 88%. Em muitos casos as tarefas são harmônicas ou quase, o que leva o patamar perto de 100% [Sha90]. Ainda se uma tarefa excede sua taxa de utilização original  $U_i$ , o conjunto continuará escalonável desde que a condição [1] seja cumprida [Locke92].

Um conjunto de tarefas com taxa de utilização conjunta  $U$  maior que o limite dado por [1] pode ser escalonável com taxa monotônica se cumpre o **teste necessário e suficiente de escalonabilidade**, que é baseado no Teorema 2.1.

**Teorema 2.1:**

Para um conjunto de tarefas independentes e periódicas, se cada tarefa cumpre seu primeiro "deadline" quando todas começam ao mesmo tempo, então os "deadlines" sempre serão cumpridos, para qualquer combinação de tempos de começo (*task phasing*).

Este teste aplica-se às tarefas de menor prioridade do conjunto, cujo cômputo faz o sistema superar o patamar seguro [Sha90], e consiste em verificar que em algum ponto de escalonamento do primeiro período da tarefa (o seu primeiro "deadline", e os fins de período das tarefas de maior prioridade, anteriores ao seu primeiro "deadline"), ela e todas as tarefas de maior prioridade conseguem completar o seu número de execuções correspondente nesse período. Se assim for, tanto a tarefa analisada como as de maior prioridade cumprirão seus "deadlines", mesmo tendo uma taxa de utilização maior que o patamar definido em [1].

O algoritmo taxa monotônica é estável frente a sobrecargas, e seu *conjunto crítico* é formado pelas  $k$  tarefas de maior prioridade, cuja soma das taxas de utilização não supere o patamar seguro [1]. Como a prioridade reflete apenas o período da tarefa e não a sua importância, existem situações em que tarefas importantes ficam fora do conjunto crítico por terem períodos maiores. A técnica de **transformação de período** permite segurar uma tarefa determinada dentro do conjunto crítico: escalona-se a tarefa como se seu período fosse metade do original ( $=P_i/2$ ), e suspende-se sua execução após consumir metade do seu pior tempo de cômputo. A suspensão pode ser voluntária, originada pela própria tarefa.

O modelo temporal suportado pelo taxa monotônica não permite o tratamento de tarefas não periódicas; no entanto, elas ainda podem ser tratadas por extensões adequadas. Os *servidores aperiódicos* [Sprunt89] utilizam-se do tempo de computação de tarefas periódicas especiais para o processamento das requisições aperiódicas; uma modificação dos mesmos permite tratar as tarefas esporádicas. Taxa Monotônica também foi estendido para sincronizar o acesso de tarefas a recursos compartilhados: neste caso, o *Protocolo Limite de Prioridade (Priority Ceiling Protocol - PCP)* [Sha90] [Chen90] [Baker91] associa a cada semáforo um limite de prioridade, que impede o acesso ao recurso às tarefas com prioridade menor que este limite.

[Lehoczky90] apresenta uma extensão do algoritmo base para o caso em que o "deadline" de cada instância é postergado além do tempo da próxima invocação. [Shih93] generaliza o caso anterior para "deadlines" arbitrários (mas anteriores ao tempo da próxima invocação). Em ambos os casos, os resultados são matematicamente complexos e válidos apenas quando todas as tarefas tem seus "deadlines" modificados da mesma forma, o que restringe sua aplicação. [Sha93] ainda apresenta uma extensão para o tratamento de sistemas distribuídos. Todas estas extensões tem suporte analítico rigoroso, apresentando testes de escalonabilidade que são uma extensão do original para Taxa Monotônica.

Finalmente, Taxa Monotônica é um algoritmo ótimo entre os algoritmos de prioridade fixa, no sentido que nenhum outro algoritmo de prioridade fixa pode escalonar um conjunto de tarefas que o TM não consiga [LiuC73]. Sua implementação é simples (as prioridades são fixas), apresentando baixa sobrecarga de escalonamento e desempenho aceitável. No entanto, sua utilização não é conveniente quando o número de tarefas esporádicas é grande, ou na presença de tarefas periódicas chegando dinamicamente ao sistema. [Obenza93] e [Klein94] oferecem uma visão atualizada da posição do algoritmo na comunidade de Tempo Real.

### 2.3.2. ESCALONAMENTO DINÂMICO

O escalonamento dinâmico incrementa a adaptabilidade e reconfigurabilidade do sistema frente a falhas ou eventos externos, e permite a ativação dinâmica de manipuladores de erro [Blake91]. As decisões são mais adequadas a um ambiente cambiante, por estarem baseadas em informação atualizada sobre o estado do sistema.

Apresentamos aqui os principais algoritmos dinâmicos de escalonamento. Destaca-se entre eles o *earliest deadline*, por atingir o máximo nível teórico de utilização de processador e pela sua flexibilidade, decorrente das extensões que suporta para modelos temporais além do original.



2.3.2.1. EARLIEST DEADLINE

O algoritmo *earliest deadline* (ED) aplica-se a tarefas com o modelo temporal válido para taxa monotônica. Em todo momento é executada a tarefa pronta de "deadline" mais próximo. Assume-se que o "deadline" de uma instância de tarefa é o tempo de chegada da próxima instância da mesma. A **condição necessária e suficiente de escalonabilidade** para ED (Condição 2.2) mostra que, enquanto a utilização total da CPU for inferior a 100%, todos os "deadlines" do sistema escalonado com ED serão cumpridos.

$$U = \sum_i U_i = \sum_{i=1}^n \frac{c_{ti}}{P_{ti}} \leq 1 \quad [3]$$

**Condição 2.2:** Condição Necessária e Suficiente de Escalonabilidade para Earliest Deadline

Existem algumas variações úteis do ED: o **EDL** (Earliest Deadline as Late as possible) [Chetto89] escalona as tarefas o mais perto possível do "deadline": desde o "deadline" para trás, em direção ao tempo atual. EDL é oposto ao ED tradicional, que poderia ser chamado de **EDS** (Earliest Deadline as Soon as Possible) por escalonar a tarefa o mais perto possível do começo do período. O **escalonador inverso** (EI) [Shih92] é uma heurística intermediária entre o EDS e o EDL, que escalona as tarefas em ordem decrescente de tempo de prontas, e o mais perto possível do "deadline". A Figura 2.4 mostra um conjunto de tarefas escalonado por estes algoritmos.

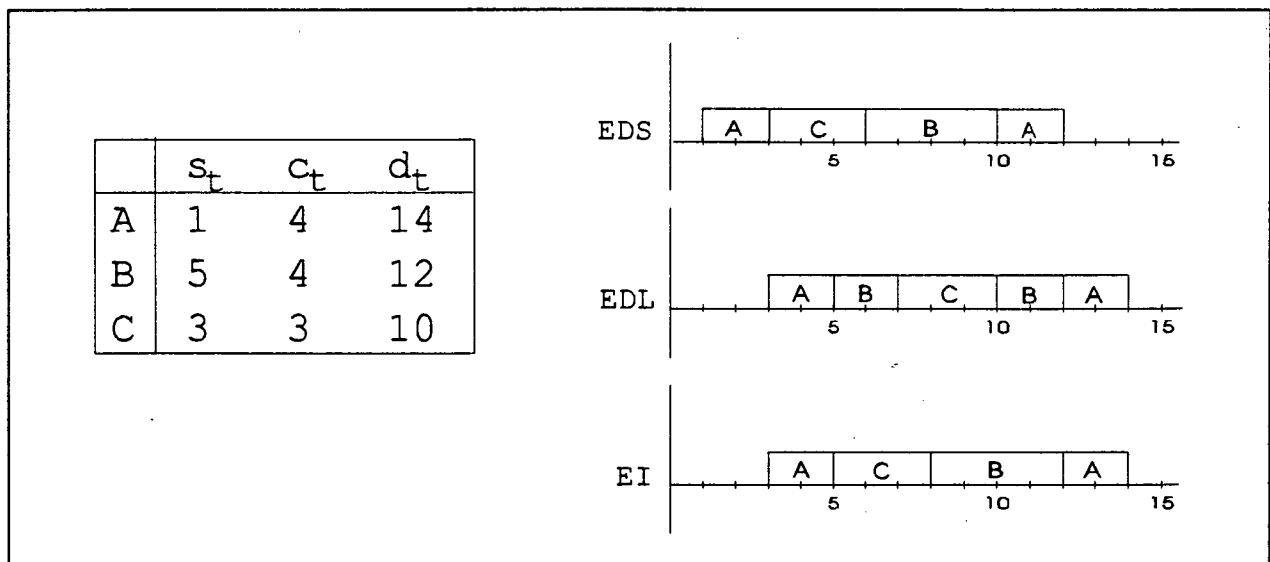


Figura 2.4: Conjunto de tarefas escalonado por Earliest deadline (EDS), EDL e Escalonamento Inverso

[Chetto89] demonstra que a aplicação de EDL maximiza o tempo de execução disponível para atender requisições aperiódicas, apresenta métodos para o cálculo deste tempo e demonstra que EDS minimiza esse tempo aperiódico. No mesmo artigo é apresentado um teste de aceitação on-line para tarefas esporádicas.

O algoritmo ED pode ainda ser utilizado para escalonar tarefas não periódicas. Neste caso, o escalonamento é chamado *deadline monotônico* e sua **condição necessária e suficiente de escalonabilidade**, para um conjunto de tarefas ordenado por tempo de resposta crescente, é mostrada na Condição 2.3 [Henn89].

$$\forall i \quad 1 \leq i \leq n \quad g_i(t) \geq \sum_{j=1}^i m_j(t), \quad 0 \leq t \leq \infty \quad [4]$$

onde  $g_i(t) = d_i - t$  : tempo para o "deadline"

$d_i$  : "deadline" da tarefa  $i$

$t$  : tempo presente

$m_j(t)$  : tempo de execução residual para completar a tarefa  $j$

### Condição 2.3: Condição necessária e suficiente de escalonabilidade para *deadline monotônico*

Pelo restrito modelo temporal suportado, o earliest deadline original não suporta sincronização das tarefas no acesso aos recursos compartilhados do sistema. No entanto, ele tem sido estendido para isso através do *Kernelized Monitor* [Kopetz92]. [Mok84], o *Protocolo Limite Dinâmico de Prioridade (Dynamic Priority Ceiling Protocol)* [Chen90] e a *Política de Pilha de Recursos (Stack Resource Policy - SRP)* [Baker91]. Os servidores aperiódicos do taxa monotônica podem ser adaptados para melhorar os tempos médios de resposta do ED a requisições aperiódicas [Baker91]. [Homa94].

O earliest deadline é um algoritmo globalmente ótimo [LiuC73], e ainda é ótimo para conjuntos de tarefas arbitrários, não necessariamente periódicos; sua prova da otimalidade pode ser achada em [Dert89]. Sua principal desvantagem é não ser estável, é dizer, não garante quais as tarefas que cumprirão seus "deadlines" em condições de sobrecarga.

#### 2.3.2.2. MENOR FOLGA

Menor Folga (*Least Laxity - LL*) é um algoritmo preemptivo para tarefas do modelo do taxa monotônica, que sempre escolhe para executar a tarefa de menor folga [Stew91]. *Folga* é o tempo

máximo que o escalonador pode atrasar a execução da tarefa antes dela perder seu "deadline" "corrente", sendo uma medida da flexibilidade disponível para escalonar uma tarefa num dado instante.

$$\text{folga}(t) = \max\{ 0, \text{deadline} - t - \text{tempo\_execução} \}$$

Menor Folga apresenta a mesma condição de escalonabilidade que o earliest dealine, e também é ótimo quando o tempo de troca de contexto é desprezado [Audsley90]. Como o earliest dealine, é instável frente a sobrecargas de processamento. A diferença principal com o ED é que o LL leva em conta o tempo de execução remanescente das tarefas no seu cálculo.

O fenômeno de *trashing* acontece quando duas ou mais tarefas têm folgas semelhantes. Quando uma delas executa, as outras diminuem sua folga paulatinamente até ficar por baixo do valor de folga da tarefa corrente<sup>8</sup>. Ao chegar neste ponto, a tarefa corrente perde a CPU para alguma das tarefas com folga inferior, e o ciclo se repete; como resultado, as tarefas ficam se revezando no uso da CPU. Se o tamanho do slot de execução não for notoriamente maior que o tempo de troca de contexto, a maior parte do tempo será gasto em trocas de contexto ao invés do conjunto avançar efetivamente na sua execução.

### 2.3.2.3. BIN-PACKING

Este algoritmo preemptivo apresenta um teste de escalonabilidade para tarefas esporádicas baseado no EDL e executa as tarefas segundo EDS [Blake91] [Berry92] [Schwan92]. O tempo do processador é dividido em *bins*, que representam intervalos disjuntos de tempo arranjados cronologicamente. Cada "bin" começa no tempo de chegada ou "deadline" de uma tarefa, e vai até o tempo de chegada ou "deadline" mais próximo.

Quando uma tarefa chega, o escalonador calcula o tempo livre entre os bins com tempos finais maiores que o tempo de pronta e menores que o "deadline" da tarefa, para conferir se tem suficiente para escaloná-la. Se tiver, o escalonador cria novos bins, compatíveis com o tempo de chegada da tarefa e o mais perto possível do seu "deadline", e os incorpora à lista, ajustando finalmente as quantidades de tempo livre nos bins afetados e novos. Caso contrário, a nova tarefa é rejeitada. A fila de prontos do despachante é ordenada segundo EDS, pelo que sempre é executada a tarefa pronta com tempo de começo passado e "deadline" mais próximo.

---

<sup>8</sup> da definição matemática surge que a folga da tarefa corrente permanece constante enquanto executa

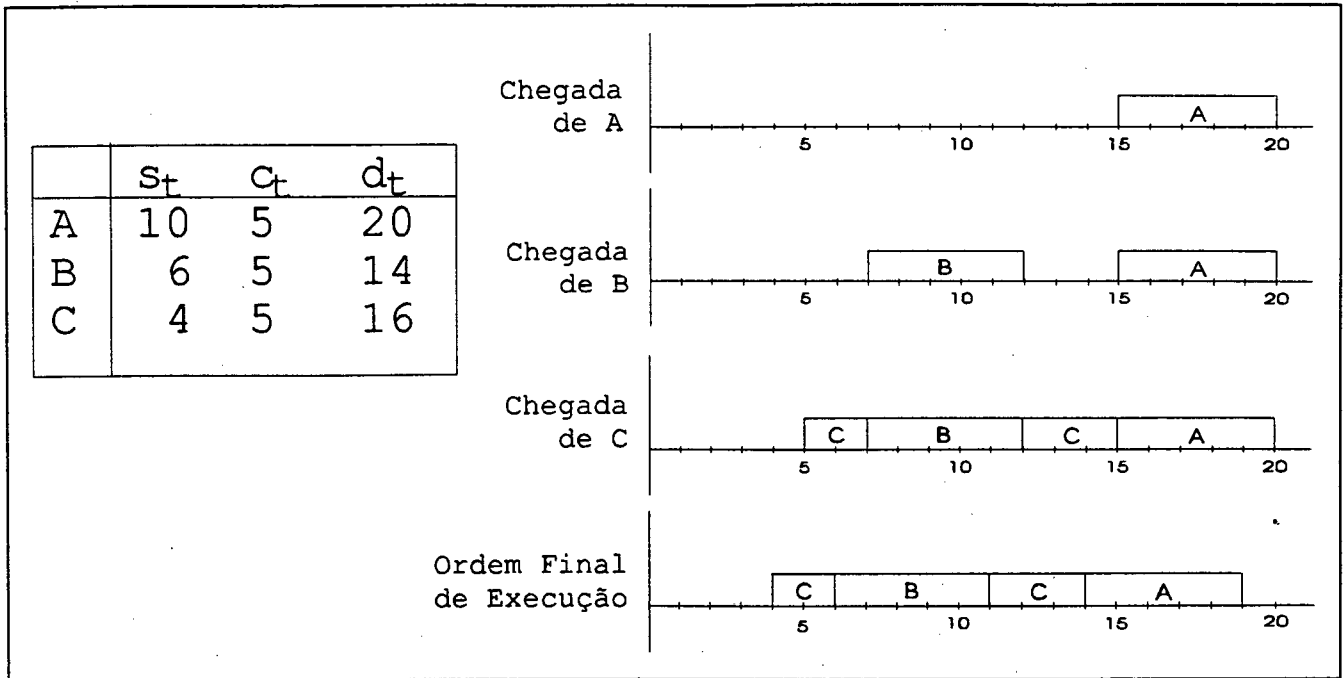


Figura 2.5: Exemplo de Escalonamento com algoritmo Bin-Packing

O algoritmo pode ser estendido para respeitar restrições sobre pedidos de recursos, recursos que podem ser *exclusivos* (memória) ou *escalonáveis* (displays, drives de disco). A cada recurso escalonável associa-se uma lista de bins semelhante à da CPU, que representa o tempo do recurso disponível: a análise de um pedido envolve o processamento simultâneo de várias listas diferentes de bins [Blake91]. [Schwan92] estende o algoritmo para suportar tarefas periódicas e requisições com relações de precedência (igualando o tempo de começo da tarefa seguinte com o "deadline" da precedente, ou ainda melhor, definindo os tempo de início e "deadline" da tarefa conseqüente como posteriores aos da antecedente [Liu91]), junto com um teste de aceitabilidade em tempo polinomial.

### 2.3.3. ALGORITMOS MISTOS - MUF

Embora flexível e com alta taxa de utilização da CPU, o uso exclusivo de escalonamento dinâmico não é apropriado por não oferecer garantias em caso de sobrecarga da CPU. Estratégias mistas combinam as melhores propriedades das abordagens estáticas e dinâmicas. Uma delas consiste em escalonar as tarefas mais críticas e de alta frequência (como o controle de baixo nível) com algoritmos estáticos garantidos tipo taxa monotônica, e no tempo restante servir as tarefas menos críticas e freqüentes (controle médio e alto) segundo uma abordagem dinâmica de baixo custo como earliest deadline [Blake91] [LiuC73].

Em [Stew91] é apresentado o algoritmo Máxima Urgência Primeiro (**Maximum-Urgency-First - MUF**), que é uma mistura do taxa monotônica, earliest deadline e menor folga. A cada tarefa é atribuída uma *urgência*, combinação de uma componente dinâmica (inversamente proporcional à folga da tarefa) e duas fixas: a *criticalidade* (alta se a tarefa pertence ao conjunto crítico, senão baixa), de maior precedência que a componente dinâmica; e a *prioridade do usuário*, de menor precedência.

Primeiro define-se as prioridades estáticas para cada tarefa, e analisa-se a escalonabilidade do conjunto crítico segundo taxa monotônica. Para determinar a próxima tarefa a executar:

1. Escolhe-se a tarefa de maior criticalidade.
2. Se existe mais do que uma tarefa com criticalidade alta, escolhe-se a de menor folga.
3. Caso exista mais do que uma tarefa com igual folga, escolhe-se segundo a prioridade do usuário.
4. Frente a tarefas com iguais atributos de criticalidade, folga e prioridade de usuário, escolhe-se segundo a ordem de chegada.

O conjunto completo é reescalonado sempre que uma tarefa é acrescentada ao sistema. Os "deadlines" das tarefas pertencentes ao conjunto crítico são garantidos (como no TM), e ainda o sistema tem limite máximo de escalonabilidade de 100% para elas (como no ED). O escalonamento é previsível, desde que é possível saber quais as tarefas que cumprirão seus "deadlines" para um nível determinado de sobrecarga.

## 2.4. CONSIDERAÇÕES GERAIS

Sistemas de Tempo Real já formam parte do nosso cotidiano. Sua criticalidade faz necessário garantir seu funcionamento correto até em casos extremos. Suas arquiteturas e suportes de execução (executivos multitarefas) devem ser previsíveis para poder formalizar essas garantias; neste sentido, os escalonadores são os grandes responsáveis pela previsibilidade do conjunto, ao distribuírem no tempo os recursos do sistema para atingir seus objetivos.

Existe na literatura uma extensa lista de políticas de tempo real, que baseiam explicitamente suas decisões nos parâmetros temporais das tarefas alvo. As analisadas apresentam graus distintos de complexidade e eficiência, requerendo distintas qualidade e quantidade de informação temporal do sistema; a sensibilidade do seu comportamento às mesmas é variável. As mais completas (e que

portanto oferecem maior grau real de escalonabilidade) fornecem mecanismos para alocação integrada de recursos (como é o caso dos heurísticos e do bin-packing).

No Quadro II-1 é apresentada uma tabela comparativa dos principais algoritmos abordados. Ao confrontá-los é importante avaliar qual é a taxa de utilização que permitem ainda fornecendo garantias: algoritmos com taxa de utilização pobre podem rejeitar conjuntos de tarefas que heurísticas mais inteligentes conseguiriam escalonar. Isto leva à sub-utilização dos recursos disponíveis e, em consequência, o sistema deve ser sobredimensionado para cumprir seu objetivo.

TIPO DE ALGORITMO	NOME	HEURÍSTICA	ESTÁVEL?	ÓTIMO?	EXTENSÕES	COMENTÁRIOS
ESTÁTICOS	EXECUTIVO CÍCLICO	(subjativa)			X	. previsível . frágil . manual
	*TAXA MONOTÔNICA	menor período	X	X	X	. estável . restrito
DINÂMICOS	*EARLIEST DEADLINE	deadline mais próximo		X	X	. alta escalonabilidade
	*MENOR FOLGA	menor folga		X		. trashing
	BIN-PACKING	earliest deadline			X	
MISTOS	*MÁXIMA URGÊNCIA	maior urgência	X	X		. 100% escalonab. E estável

Quadro II-1: Algoritmos de escalonamento monoprocessador tradicionais. Os marcados com \* suportam o mesmo modelo temporal de tarefa (o do taxa monotônica)

O algoritmo Taxa Monotônica e suas extensões apresentam-se como um dos mais sólidos teoricamente, suportando modelos de tarefas razoavelmente genéricos a um custo de execução aceitável. Ele oferece uma transição "suave" às técnicas formais desde as abordagens mais convencionais ao problema (prioridades definidas arbitrariamente pelo usuário), porém sua extensão para multiprocessadores ou sistemas distribuídos não é totalmente elegante. Embora com embasamento formal, a análise com taxa monotônica é complicada e propensa a erro quando o sistema a escalonar é complexo; não sendo intuitiva, a experiência do projetista conta muito para obter um resultado

satisfatório<sup>9</sup>. Esta análise é ainda frágil, no sentido que uma modificação parcial na estrutura do sistema força o recálculo de grande parte dos números seguintes. Seu comportamento deixa de ser simples de entender quando estendido para tratar situações complexas (por exemplo, com limite de prioridades), extensões que diminuem ainda mais a escalonabilidade do sistema. Mesmo assim, o algoritmo continua previsível, que é sua principal vantagem.

Em caso de sobrecarga, algoritmos alternativos ao taxa monotônica (como earliest deadline ou menor folga) não mais oferecem previsibilidade, embora sua escalonabilidade teórica seja maior. Neste caso, outras abordagens como algoritmos baseados no modelo de escalonamento *impreciso* apresentam-se como mecanismos promissores para a tolerância a falhas temporais, a partir da aceitação de resultados com qualidade menor que a normal.

Existem ainda algoritmos que suportam abordagens alternativas ao problema de escalonamento em tempo real. A abordagem de Benefício Acumulativo assume que o término de uma tarefa fornece um benefício ao sistema que depende do seu tempo de término. Assim, o escalonador ordena a execução das tarefas de maneira a maximizar sua contribuição ao benefício coletivo, através da abordagem *melhor esforço* (*best effort*). Finalmente, modelos temporais complexos podem ser tratados com algoritmos de procura em árvore, que caracterizam-se por utilizar heurísticas para encontrar agendas que satisfaçam simultaneamente um conjunto complexo de restrições temporais, de precedência e de recursos. O leitor interessado é referido a [Ferra94].

Como consequência da complexidade NP-completa do problema geral de escalonamento envolvendo recursos [Mok84], cada algoritmo é fortemente amarrado a um modelo temporal de tarefa específico, de pouca generalidade. Não existe um algoritmo único que reúna as vantagens principais (flexibilidade e previsibilidade) para tarefas suficientemente genéricas rodando sobre monoprocessador, com um custo de execução razoável. Portanto, não é realista tentar resolver aplicações diferentes com um único algoritmo de escalonamento.

A aplicação em mãos determina o modelo tempo-real de tarefa dominante, que pela sua vez condiciona a escolha dos algoritmos de escalonamento aplicáveis. Deve-se escolher o algoritmo que mais se adapte ao modelo de tarefas da aplicação: este deveria surgir de uma análise cuidadosa de suas características temporais e da disponibilidade de recursos no sistema. Por exemplo, para robótica (tarefas curtas e muito frequentes) são necessárias políticas eficientes, de baixo custo de execução. No caso da supervisão de uma célula flexível, com tarefas esporádicas formando grupos, sem comunicação entre si e com restrições de precedência, um algoritmo como o misto MUF ou o bin-packing seria apropriado.

---

<sup>9</sup> [Obenza93] e [Sha93] mostram exemplos de análise onde isto é percebido imediatamente.

Isto nos leva à necessidade de termos núcleos com suficiente flexibilidade para suportar a escolha da política quando for necessário. Assim, o usuário confrontado à tarefa de programar aplicações com modelos temporais distintos não precisaria mudar de núcleo, mas poderia reconfigurá-lo para o modelo alvo, sem sacrificar suas características principais.

## 2.5. CONCLUSÃO

Neste capítulo abordaram-se conceitos referentes ao problema da execução em Tempo Real em geral, e a políticas de escalonamento em arquiteturas monoprocessador para ambientes críticos em particular.

A partir da análise das características dos Sistemas de Tempo Real, apresentou-se um modelo temporal para tarefas cuja terminologia sustentou a discussão posterior. Uma parte fundamental dos núcleos de tempo real, o escalonador, foi analisada como elemento chave na procura de previsibilidade no conjunto.

Os algoritmos de escalonamento clássicos estáticos (executivo cíclico e taxa monotônica), dinâmicos (earliest deadline, menor folga e bin-packing) e mistos (máxima urgência) foram apresentados, junto a algumas condições matemáticas de escalonabilidade e mecanismos de extensão para outros modelos temporais. O funcionamento do algoritmo bin-packing e das distintas variantes de heurísticas do earliest deadline (EDS, EDL, EI) foram mostradas através de exemplos.



# CAPÍTULO 3

## SUPORTES DE EXECUÇÃO PARA APLICAÇÕES DE TEMPO REAL

Neste capítulo apresentam-se algumas das abordagens mais significativas ao problema de Tempo Real e descrevem-se alguns suportes de execução e sistemas operacionais de tempo real, tanto acadêmicos como comerciais que as implementam.

A profusão de problemas que surgem ao tentar garantir os deadlines das atividades críticas de uma aplicação motiva o uso de técnicas variadas para sua solução, que vão desde as complementares (como o caso de tolerância a falhas e gerenciamento cooperativo de recursos), passando por estruturais (UNIX para tempo-real ou micronúcleos) e até divergentes (como as abordagens *event-triggered* e *time-triggered*).

Cada item deste capítulo desenvolve uma abordagem específica ao problema, sob a descrição de uma implementação representativa que a suporte. No final do capítulo, apresenta-se uma comparação qualitativa das mesmas salientando suas diferenças principais e conclusões que influíram na especificação do núcleo de tempo real a ser implementado.

**"Don't force it, son.  
You can always get a bigger hammer."**

### 3.1. ABORDAGENS DE IMPLEMENTAÇÃO EVENT-TRIGGERED VS TIME-TRIGGERED

Segundo o mecanismo que dispara as atividades, sistemas operacionais (SO) de tempo real podem ser classificados em Event-Triggered ou Time-Triggered [Kopetz91] [Kopetz92] [Kopetz94]. Os sistemas **Event-Triggered** iniciam suas atividades com a ocorrência de eventos significativos, tanto externos (como interrupções de dispositivos) como internos ao sistema (chegada de tarefas, alterações nos estados de outras tarefas, interrupção do temporizador, etc), tomando on-line todas as decisões de escalonamento.

As atividades de processamento e comunicação nos sistemas **Time-Triggered** são disparadas pela progressão do tempo, quando este atinge valores predefinidos; todas as tarefas e comunicações são periódicas. Embora a ocorrência de eventos no ambiente não possa ser controlada, os pontos no tempo em que o computador os reconhece são pré-determinados e fixos. Isto dispensa seu tratamento por interrupções (as variáveis de estado externas e periféricos são amostrados em tempos específicos), reconhecendo apenas a do temporizador: quando acontece, o despachante procura numa tabela a tarefa a ser ativada nesse momento.

Na abordagem Time-Triggered, o escalonamento é estático, a partir da análise detalhada do ambiente e levando em conta as premissas consideradas válidas sobre seu comportamento. Esta análise precisa de informação completa e *a priori* sobre o sistema, gerando implementações que em grande parte são válidas por construção. O sistema é dimensionado para ser previsível mesmo no pior caso possível de carga (que em geral é muito difícil de acontecer), o que leva a uma grande subutilização dos recursos disponíveis. Já no caso Event-Triggered, sua flexibilidade favorece a otimização do uso dos recursos, porém sua previsibilidade não é total, abarcando apenas um subconjunto das atividades do sistema. Frente a uma falha, a atitude destes sistemas é a de *melhor esforço*, mostrando uma degradação paulatina nas suas funcionalidades.

#### 3.1.1. ABORDAGEM EVENT-DRIVEN: SPRING

O **Spring** [Mole90] [Stan87] [Stan91] é um sistema operacional multiprocessador distribuído, desenvolvido para servir à próxima geração de aplicações de tempo real: grandes, complexos,

distribuídos e adaptativos. Seu principal objetivo é conseguir uma computação **previsível** a nível de aplicação, e também flexível. Apresenta uma sofisticada estrutura de escalonamento, suportando alocação *integrada* de recursos (tanto da CPU como dos outros) com alto grau de escalonabilidade.

### 3.1.1.1. O ESCALONAMENTO NO SPRING

Dependendo da importância e dos requerimentos temporais, as tarefas são classificadas em *críticas*, *essenciais* e *não-essenciais*. O não cumprimento das restrições temporais das tarefas críticas (em pequeno número em relação às outras) pode ocasionar uma catástrofe, e portanto são garantidas estáticamente. As essenciais são garantidas em tempo de execução e o não cumprimento de suas restrições degrada o desempenho geral mas não tem conseqüências maiores. As não-essenciais, por último, não apresentam restrições temporais.

O Spring separa o mecanismo da política de escalonamento, e é composto de quatro níveis. No menor existem os *despachantes*, e depois os escalonadores *locais*, responsáveis pela garantia dinâmica das tarefas essenciais. O terceiro nível é o de escalonamento distribuído, que tenta achar um nó para executar qualquer tarefa que não possa ser garantida localmente. O quarto nível (*metaescalonador*) chaveia algoritmos de escalonamento quando percebe mudanças significantes no ambiente.

Para o escalonamento, o compilador segmenta os programas em entidades escalonáveis não-preemptivas (tarefas), segundo tempo e recursos. Quando uma tarefa *essencial* chega ao sistema, o núcleo calcula suas necessidades de recursos e restrições de tempo; o escalonador tenta então garanti-la dinamicamente, de maneira que a nova tarefa e o conjunto de tarefas previamente garantido cumpram seus deadlines. Se consegue, o núcleo aloca todos os recursos requeridos quando a tarefa começa sua execução; senão, tarefas de importância menor à nova são retiradas do sistema se com isso consegue-se garantir a nova. Como resultado, a tarefa nova ou um conjunto das tarefas menos importantes são sujeitas à semântica de falha apropriada.

Conhecendo o tempo máximo que o escalonador leva para garantir uma tarefa<sup>1</sup>, este custo é somado ao tempo atual para determinar a *linha de corte*: todas as tarefas com tempo de começo anterior à linha são reservadas para os despachantes, e não podem ser reescaloadas. Apenas o escalonamento das tarefas que começam após a linha de corte pode ser modificado para acomodar uma nova tarefa chegando ao sistema.

O algoritmo de escalonamento *local* é de procura em árvore subótimo [Ramam90] [Stan91], sendo suas heurísticas funções simples das características temporais das tarefas. O conjunto de

---

<sup>1</sup> este tempo máximo depende do algoritmo específico e do número de tarefas já garantidas pelo sistema.

algoritmos de escalonamento disponíveis sobre tarefas preemptivas ou não, periódicas ou não, com restrições temporais, requerimentos de recursos e relações de precedência. Spring suporta ainda *grupos de tarefas* (varias tarefas com tempos de começo e deadline comuns e relações de precedência).

Para obter previsibilidade, tanto o dispatcher como o escalonador e todas as primitivas do sistema têm custo de execução limitados. Embora a complexidade do algoritmo usado na versão 1 seja linear com o número de tarefas do conjunto ( $O(N)$ ), seu tempo de execução de pior caso é limitado pelo número máximo de avaliações da função heurística permitido para cada invocação. A partir da proposta de tempo máximo para achar a agenda e conhecendo quanto tempo leva processar uma tarefa, pode-se calcular o número máximo de tarefas que é possível tratar numa execução. Mesmo assim, o algoritmo de escalonamento tem uma carga de processamento considerável, e portanto o Spring aplica-se a processos com folgas médias-grandes (maiores de 100 mSec).

### 3.1.2. ABORDAGEM TIME-DRIVEN: MARS

MARS (Maintainable Real-Time System) [Kopetz89] [Kopetz92] [Kopetz94] [Pospis92] é um sistema operacional distribuído com suporte de tolerância a falhas, apto para aplicações de controle de processos. Seu objetivo é garantir a execução de aplicações de tempo real em condições pré-antecipadas de carga extrema<sup>2</sup> e falhas.

#### 3.1.2.1. ARQUITETURA DO HARDWARE

Um sistema MARS é formado por um ou mais *clusters*: sistemas distribuídos compostos de *estações* conectadas por um barramento síncrono de tempo real, com uma cópia do MARS rodando em cada uma. Todos as estações seguem uma base de tempo global, que lhes permite sincronizar suas atividades e utilizar um protocolo de acesso ao barramento tipo TDMA -determinista e livre de colisões-. O *clustering* facilita a extensibilidade e manutenção: novos componentes podem ser acrescentados sem modificação dos atuais, até atingir o limite de configuração. A partir dali, os componentes existentes podem expandir-se em novos *clusters*.

A previsibilidade do MARS é consequência do determinismo obtido através da sua arquitetura time-triggered e da metodologia desenvolvida para o projeto de suas aplicações. A arquitetura toma conta dos serviços de redundância ativa de recursos de processamento e comunicação, assim como detecção e tratamento de erros, de forma transparente ao programador [Kopetz92]. Desta maneira consegue-se separar os aspectos lógicos dos temporais no projeto da aplicação, pelo que o

---

<sup>2</sup> Carga extrema significa que todos os eventos possíveis acontecem na sua máxima (embora especificada) taxa de ocorrência.

programador precisa se preocupar apenas com um programa sequencial para o qual deve cumprir um determinado *orçamento temporal*.

A tolerância a falhas está baseada na redundância *ativa* dos componentes e em múltiplas transmissões de cada mensagem no barramento. Assume-se que os componentes possuem propriedades de auto-teste e de *falha silenciosa*<sup>3</sup>: assim, sempre que houver um componente funcionando, seu tipo de serviço será mantido no sistema.

### 3.1.2.2. ARQUITETURA DO SOFTWARE

O Mars é temporalmente *rigido*, e portanto totalmente determinista. Executa um conjunto fixo de tarefas periódicas, que comunicam entre si e o ambiente apenas por mensagens. As tarefas recebem todas as mensagens necessárias antes de começar, realizam os cálculos necessários e terminam enviando suas mensagens de saída, não sendo permitida a comunicação durante o cômputo.

Processadores comunicam-se através de *mensagens de estado*, semânticamente equivalentes a variáveis globais. Elas carregam informações sobre o estado do ambiente num instante de tempo e são válidas durante um certo intervalo, após o qual o SO as descarta. A lista de mensagens do núcleo é atualizada somente em cada *tick* do relógio, e portanto as mudanças de estado provocadas pelas mensagens acontecem simultaneamente em todas as estações.

### 3.1.2.3. MÉTODO DE PROJETO

Para descrever as atividades do sistema, MARS baseia-se no conceito de *transações de tempo real*: a seqüência de passos de processamento e comunicação que se segue a partir de um estímulo do ambiente até sua resposta correspondente, dentro de um intervalo de tempo dado. Durante o desenvolvimento de uma aplicação, o usuário descreve suas transações e as refina sucessivamente até chegar a uma seqüência de tarefas e trocas de mensagens, onde as tarefas são relacionadas por um *grafo de precedência*.

Os tempos de execução das tarefas, junto com as mensagens, formam a entrada do escalonador estático. O algoritmo do escalonador é de procura em árvore subótimo, semelhante ao do Spring, e produz uma agenda de tarefas e atribuição de mensagens aos *slots* de tempo do barramento que cumpre todos os requerimentos temporais, de sincronização e comunicação impostos pela especificação. Se não conseguir achar uma agenda adequada, então os parâmetros temporais das tarefas devem ser modificados, ou os refinamentos rejeitados, em forma conveniente.

Para o projeto MARS foi desenvolvido um sofisticado conjunto de ferramentas de apoio. O

---

<sup>3</sup> eles operam como devido ou não produzem nenhum resultado

*compilador*, além de gerar o código executável, gera a *árvore temporal* a partir da análise estática do código fonte. Uma ferramenta de *edição temporal* calcula o tempo máximo de execução de cada tarefa, e permite ao usuário a observação imediata dos efeitos no sistema de mudanças planejadas (através da edição tanto do texto fonte quanto dos seus tempos projetados), que assim podem ser avaliadas antes de serem implementadas. Foi desenvolvida também uma ferramenta de análise de dependência, capaz de estimar e analisar falhas e seus impactos no conjunto.

## 3.2. ARQUITETURAS MONOLÍTICA VS. MICRONÚCLEO

A seguir são analisadas as duas principais abordagens para a implementação de sistemas operacionais: a clássica *monolítica*, onde todo o código do suporte é fechado num único programa sem possibilidade de configuração externa pelo usuário, e a mais recente estrutura de *micronúcleo*, que tenta separar suas distintas funcionalidades em módulos que podem ser configurados até dinamicamente.

### 3.2.1. SISTEMAS UNIX PARA TEMPO REAL

#### 3.2.1.1. O PADRÃO POSIX

O padrão POSIX [Corvin90] [Singh90a] vem sendo tomado como referência pelos fabricantes de sistemas operacionais comerciais, que utilizam a interface e semântica de suportes padrões de propósitos gerais tipo UNIX (LynxOS, QNX) ou DOS (RTKernel) como base para desenvolver produtos compatíveis de alto desempenho. POSIX é um padrão que especifica um conjunto de interfaces e funcionalidades para as partes de um sistema operacional independente UNIX-like. Desde 1987, o Grupo P1003.4 do Comitê POSIX da IEEE vêm desenvolvendo extensões de tempo real à especificação POSIX original. O documento gerado (P1003.4) define a sintaxe e semântica de uma série de facilidades frequentemente requeridas para a implementação eficiente de aplicações de tempo real; tais extensões versam sobre semáforos binários, "locks" de processos em memória, escalonamento por prioridades, controle de temporizadores, gerenciamento de *threads*, etc.

A norma P1003.4 não determina quais mecanismos de implementação ou níveis de desempenho devem ser providos, embora especifica suas *métricas* de desempenho: os vendedores de sistemas conformes devem fornecer os valores numéricos específicos para seus produtos. A familiaridade dos potenciais usuários com a interface convencional do sistema UNIX, junto à disponibilidade de ferramentas e ambientes de desenvolvimento no sistema nativo, são fatores importantes no presente sucesso desta abordagem.

### 3.2.1.2. SISTEMAS UNIX COMERCIAIS

LinxOS [Singh90b] e QNX são típicos UNIX de tempo real comerciais que apresentam um desempenho relativo melhorado em relação aos tradicionais. Estes sistemas continuam a ser multitarefa e com escalonador dirigido por prioridades, embora o projetista agora possa definir a prioridade *absoluta* de cada tarefa. A execução dentro do núcleo (monolítico no caso do LynxOS, micronúcleo no caso do QNX) pode ser interrompida, visando obter uma maior responsividade a eventos externos: a maior parte das estruturas de dados são projetadas para serem acessadas durante períodos muito curtos, única ocasião onde as mudanças de tarefas são desabilitadas.

Os parâmetros de comparação costumam ser os tempos de resposta às interrupções e de troca de contexto. Entre outras coisas, estes suportes apresentam sistema de arquivo mais eficientes, conectividade em rede e facilidades para projetos embutidos (são ROM-ables). Ao invés de tratar os dispositivos de E/S como arquivos (abordagem UNIX tradicional), proveem uma interface comum para vários deles (como servocontroladores, sensores, etc), o que melhora o desempenho no seu acesso.

### 3.2.2. SO BASEADOS EM MICRONÚCLEO: CHORUS

Micronúcleo é uma tecnologia que consiste em separar as funções básicas do SO da sua interface de programação, o que permite suportar múltiplas interfaces e facilita a programação orientada a objetos e distribuída [Farrow93]. Esta abordagem é oposta à tradicional, de núcleo/SO *monolíticos*<sup>4</sup>.

A interface que o conjunto oferece ao usuário é denominada *personalidade*, e compreende também seu sistema de arquivos e interface de programação. A idéia básica é suportar no micronúcleo apenas um conjunto muito básico de serviços genéricos, de maneira que qualquer sistema operacional (*personalidade*) possa ser implementado por cima; tipicamente, estes serviços são gerenciamento de memória, *dispatching*, passagem de mensagens e um executivo multitarefa básico sobre o hardware.

As funções do micronúcleo podem ser integradas em *servidores* de sistema externos, de maior nível, para gerenciar outros recursos físicos e lógicos (arquivos, dispositivos e serviços de comunicação de alto nível). O fato de decompor um SO monolítico em servidores independentes se comunicando por mensagens implica em que estes servidores não mais precisam estar presentes no mesmo computador local. Isto fornece uma abstração uniforme para a distribuição de serviços pela rede, aumentando a tolerância a falhas ao nível do software através da reconfiguração dinâmica de módulos.

<sup>4</sup> curiosamente, a maior parte dos SO baseados em micronúcleo tem tamanho maior que os monolíticos [Farrow93], e também podem perder um pouco de eficiência pela maior sobrecarga de comunicação entre os servidores.

CHORUS [Bricker91] [CDes90] [Chorus88] é um exemplo de micronúcleo para a implementação de sistemas operacionais abertos e distribuídos, que permite particularizar um componente para uma combinação aplicação/ambiente específica. Seu modelo de processamento é de atores com múltiplas atividades (*threads*). Um *ator* é simplesmente um recipiente de recursos que oferece um espaço de endereçamento virtual no qual múltiplas atividades podem se executar. Uma atividade é caracterizada por um contexto (estado do processador: registros, PC, stack, etc) e está ligada a um único ator, embora um ator possa ter várias atividades; uma troca de contexto entre atividades do mesmo ator é bem menos custosa que uma entre atores distintos, já que os recursos do ator não precisam ser salvos.

Atividades comunicam-se por troca de mensagens não tipadas através de portos. *Portos* são ligados aos atores e podem migrar de um ator para outro. Portos também podem ser agrupados em *grupos de portos*, o que permite a difusão (*broadcasting*) de mensagens e endereçamento funcional. O grupo de portos provê um conjunto flexível de semânticas para o mapeamento cliente/servidor, incluindo reconfiguração dinâmica dos servidores.

### 3.2.2.1. ARQUITETURA GERAL

O micronúcleo do Chorus oferece serviços genéricos de escalonamento, gerenciamento de memória e comunicação, e tratamento de eventos em tempo real; gerenciadores de dispositivos são manipulados por servidores separados e protocolos.

Chorus/MIX [Armand89] é um exemplo de uma personalidade compatível-binária com UNIX System V, implementada por cima de Chorus. De estrutura totalmente distribuída, seu desempenho compara-se favoravelmente à dos UNIX monolíticos equivalentes; os serviços oferecidos em UNIX e sua semântica tradicional foram estendidos para suportar computações distribuídas, paralelas e de tempo real.

As atividades são escalonadas como entidades independentes segundo prioridades definidas pelo usuário. O escalonamento é preemptivo: a qualquer momento roda a tarefa pronta de maior prioridade. As decisões de escalonamento são baseadas em prioridades absolutas, calculadas como a soma da prioridade do ator dono da atividade e a prioridade *relativa* da atividade dentro do ator.

O suporte de comunicação do Chorus prevê dois tipos básicos de primitivas de comunicação. O RPC (Remote Procedure Call), síncrono, é otimizado segundo a distribuição física particular dos processos intervenientes; o IPC (InterProcess Communication) é de semântica muito simples: comunicação unidirecional com transparência de locação, sem detecção de erro ou controle de fluxo (serviços sem conexão), e permite aos servidores chamar-se entre si e trocar dados independentemente



do lugar da rede onde são executados.

### 3.3. SUPORTES ORIENTADOS A OBJETOS

A seguir descreveremos algumas características do modelo orientado a objetos para tempo real, reconhecido como vantajoso em relação ao de decomposição funcional em distintos pontos: adaptabilidade e maior reusabilidade do código, e melhor entendimento da aplicação em mãos.

#### 3.3.1. MODELO GERAL

No modelo orientado a objetos [Booch91] [Booch86], o universo é composto de objetos e mensagens. Objetos encapsulam dados e fornecem *métodos* para seu acesso, que são ativados por *mensagens* dirigidas ao objeto e que são a única maneira de manipular seus dados internos. Uma *classe* representa todos os objetos com algum conjunto de características similares; os *objetos* criados a partir dela são ditos *instâncias* da classe. Classes podem ter *subclasses*, compreendendo os objetos da superclasse com características comuns adicionais. Uma classe de objetos *herda* as características das suas superclasses, em particular todos seus métodos: isto facilita a criação de novas classes a partir das velhas [Bihari92].

Segundo o número de atividades (*threads*) associadas para a execução dos seus métodos, objetos podem ser **ativos** (pelo menos uma) ou **passivos**. Objetos ativos podem iniciar atividades assincronamente, sem necessidade de estímulo externo; objetos com uma única atividade só podem processar suas requisições em sequência, e portanto são *não-preemptivos*. Os objetos passivos respondem apenas a invocações externas e, sem terem atividades próprias, não realizam ações por si só.

No caso de aplicações de tempo real, o modelo de objetos descrito deve ser estendido para suportar *encapsulamento temporal*, isto é, os parâmetros temporais dos objetos devem ser acessíveis externamente para viabilizar análises globais de escalabilidade, de maneira a evitar a propagação de uma falha temporal num objeto (por exemplo, a perda do deadline de um método) a outros objetos do sistema.

O conceito de hierarquia de classes facilita a *particularização* de sistemas operacionais. Tal é o caso de **Choices** [Campb87], onde para cada problema específico (aplicação ou configuração particular de hardware) é construída uma versão especial do suporte, feita *sob medida*. Assim pretende-se chegar ao menor sistema operacional capaz de suportar a aplicação, para obter maior desempenho. A particularização é guiada e assistida pela estrutura induzida sobre o sistema pela

hierarquia: instâncias particulares das classes da hierarquia são escolhidas e combinadas para produzir o sistema operacional específico. Isto fornece também uma melhor visão conceitual de como as distintas partes estão relacionadas entre si.

### 3.3.2. PROTOCOLO TIME-FENCE: ARTS

O ARTS [Ishi92] [Mercier92] [Toku89] é um sistema operacional distribuído e orientado a objetos que suporta *encapsulamento temporal* através do protocolo "cerca" de tempo (*time fence*). Este protocolo permite o teste do cumprimento das restrições temporais de um objeto em tempo de execução.

A cada método num objeto de tempo real é associada uma "cerca" de tempo: seu tempo de execução do pior caso. Quando invocado por uma atividade de tempo real, o método será executado caso exista suficiente tempo disponível em relação à sua cerca especificada; de outra maneira, produz-se um *erro de cerca*. Ao começar sua execução, um temporizador é ativado para gerar uma exceção se sua computação não for completada no limite previsto de pior caso. Desta maneira, pode-se monitorar o consumo de tempo de uma atividade e rapidamente isolar uma violação temporal.

### 3.3.3. ADAPTABILIDADE: CHAOS

CHAOS (Concurrent Hierarchical Adaptable Object System) [Bihari92] [Schwan87] [Schwan90a] [Schwan90b] [Schwan91] [Schwan92] é uma linguagem e sistema de programação e execução baseado<sup>5</sup> em objetos, projetado para aplicações dinâmicas de tempo real sobre multiprocessadores.

Chaos suporta criação dinâmica de objetos mantendo um *pool* de instâncias pré-criadas: ao criar uma classe, o sistema cria um *pool* de instâncias e o associa a ela. Frente a um pedido de criação de objeto, uma instância pré-criada é retirada do *pool*, os parâmetros da chamada copiados nela, e seu nome registrado num servidor de nomes; para destruí-lo, simplesmente retorna-se sua instância ao *pool* original.

As construções suportadas por Chaos permitem adaptabilidade às aplicações, que pode ser *preventiva* (antecipando mudanças no ambiente operacional) ou *reativa* (em resposta a acontecimentos externos inesperados como falhas, sobrecargas temporais, etc). As atividades preventivas tentam

---

<sup>5</sup> diz-se *baseado* em objetos porque o modelo não suporta totalmente o paradigma: neste caso não tem previsão para herança.

garantir níveis determinados de funcionalidade ou desempenho; os mecanismos reativos implicam mudanças dinâmicas no software, como chaveamento de implementações distintas de um método, mudança no número de atividades internas, ou até do valor das prioridades relativas dos métodos [Bihari92]. Se uma falha acontecer num módulo crítico, então um módulo menos importante é reconfigurado para tomar conta das tarefas do módulo falho. Chaos ainda permite que a aplicação substitua as políticas de escalonamento do sistema [Nata92].

### 3.3.4. GERENCIAMENTO COOPERATIVO: R-SHELL

Uma abordagem promissora para a construção de sistemas de tempo real é contruir o software como um conjunto de objetos espertos ou agentes: estes têm conhecimento sobre si mesmos e se auto-gerenciam, negociando com os outros seus serviços [Bihari92].

R-Shell [Nata92] suporta *negociação da garantia* de um objeto. O gerenciamento dos recursos do sistema é realizado conjuntamente entre a aplicação e o sistema operacional. Cada objeto da aplicação é gerenciado pelo seu próprio *agente escalonador*, AE. O sistema operacional compreende um conjunto de recursos, os pertencentes ao mesmo tipo sendo controlados pelo *gerente de recurso* específico GR.

Um AE interage com os GRs do sistema através de mensagens, para obter garantias de que os requerimentos de recursos do seu objeto serão satisfeitos e seus deadlines cumpridos. Os GRs interagem entre si para fornecer os recursos requisitados por um AE. Mensagens de exceção são enviadas à aplicação se sua garantia não pode ser satisfeita. Nesse caso, a aplicação pode ativar manipuladores de exceção internos, computação imprecisa ou múltiplas versões, ou ainda *substituição de recursos*: os GRs liberam recursos alternativos ou quantidades maiores de recursos de outro tipo para compensar a falta dos recursos requisitados.

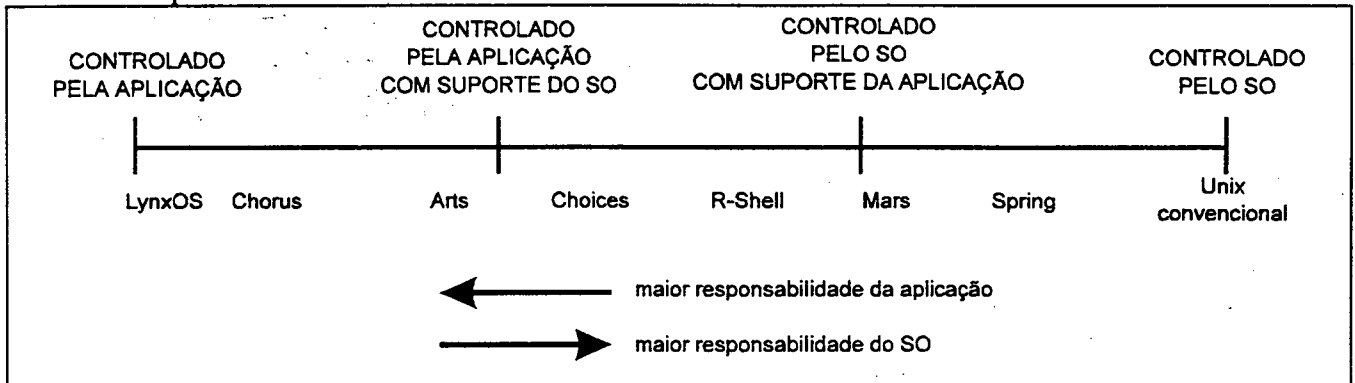
## 3.4. CONSIDERAÇÕES GERAIS

O estudo efetuado sobre suportes para tempo real evidenciou a existência de muitas e variadas abordagens para a solução do problema. Os sistemas operacionais estudados oferecem grau distinto de suporte às aplicações, e suas interfaces também apresentam graus diferentes de abstração; no entanto, ainda é possível compará-los segundo alguns pontos de vista.

Dependendo da divisão de responsabilidade no gerenciamento dos recursos do sistema (CPU e outros) entre a aplicação e o suporte, podemos distinguir as seguintes abordagens [Nata92]

(Figura 3.1):

- . *recursos controlados pelo SO*: O SO toma a responsabilidade total pelo escalonamento, ignorando nas suas decisões as características temporais das aplicações. É a abordagem default nos SO convencionais a nível de usuário.
- . *recursos controlados pelo SO utilizando semântica das aplicações*: o SO continua controlando os recursos, mas aplica algum tipo de informação temporal sobre as aplicações (como deadlines, requerimentos de recursos, criticalidade, e função de valor) nas suas decisões.
- . *recursos controlados pela aplicação com suporte do SO*: a aplicação escalona os recursos do sistema, baseada no conhecimento das suas próprias características e com informação de estado fornecida pelo SO (disponibilidade de recursos, nível de carga, etc).
- . *recursos controlados pela aplicação*: o SO não oferece nenhuma informação sobre o sistema, pelo que a aplicação deve incorporar todas as técnicas necessárias para o gerenciamento de recursos. É a abordagem comum nos SOs de tempo real comerciais.



**Figura 3.1:** Classificação de Suportes de Execução segundo a distribuição de responsabilidades no gerenciamento dos recursos do sistema

A abordagem a utilizar vai ser definida pela filosofia de projeto e do método utilizado para construir aplicações, mas não pelo SO ou a linguagem de implementação.

O Quadro III-1 mostra uma comparação dos suportes estudados. A escolha do perfil de sistema operacional deve ser determinada pelas características do ambiente que se deseja controlar. Aplicações críticas em ambientes controlados (como chão de fábrica) e razoavelmente estáticas são fortes candidatos à abordagem *time-driven*, que garante sua previsibilidade. Por outro lado, ambientes com dinâmica forte ou com carga dinâmica desconhecida ou imprevisível devem ser tratados segundo uma abordagem tipo *event-driven*, onde a filosofia *best-effort* é até aceitável.

Dos suportes estudados, os únicos a fornecer previsibilidade a nível de aplicação foram o Spring e o MARS. Desses, o Spring caracteriza-se pela sua flexibilidade durante a execução e o MARS

QUADRO III-1  
QUADRO COMPARATIVO DOS SUPORTES ESTUDADOS

SUPORTES DE EXECUÇÃO	TIPO DE ESCALONAMENTO	TRATAMENTO DE ERRO	CARACTERÍSTICAS PRINCIPAIS
Arts	<ul style="list-style-type: none"> <li>. garantias por taxa monotônica + protocolo limite de prioridade</li> </ul>	<ul style="list-style-type: none"> <li>. por tratadores de exceção.</li> </ul>	<ul style="list-style-type: none"> <li>. encapsulamento temporal por protocolo <i>time-fence</i></li> </ul>
Chaos	<ul style="list-style-type: none"> <li>. earliest deadline garantido</li> </ul>	<ul style="list-style-type: none"> <li>. por recuperação em avanço e em retrocesso</li> </ul>	<ul style="list-style-type: none"> <li>. adaptabilidade</li> <li>. sofisticação no modelo</li> </ul>
Choices	<ul style="list-style-type: none"> <li>. controlado pela aplicação</li> </ul>	<ul style="list-style-type: none"> <li>. traps e manipuladores de eventos</li> </ul>	<ul style="list-style-type: none"> <li>. particularizável</li> <li>. arquitetura aberta</li> </ul>
Chorus	<ul style="list-style-type: none"> <li>. por prioridades em faixas</li> </ul>	<ul style="list-style-type: none"> <li>. por tratadores de exceção</li> </ul>	<ul style="list-style-type: none"> <li>. tecnologia de micronúcleo</li> <li>. configurável</li> <li>. interface particularizável</li> </ul>
LynxOS	<ul style="list-style-type: none"> <li>. por prioridades</li> </ul>	<ul style="list-style-type: none"> <li>. por tratadores de exceção</li> </ul>	<ul style="list-style-type: none"> <li>. UNIX-like</li> <li>. desempenho melhorado</li> <li>. núcleo monolítico</li> </ul>
Mars	<ul style="list-style-type: none"> <li>. estático, heurístico</li> </ul>	<ul style="list-style-type: none"> <li>. mascaramento de erro por redundância temporal e física</li> </ul>	<ul style="list-style-type: none"> <li>. abordagem <i>time-triggered</i> (determinista)</li> </ul>
Spring	<ul style="list-style-type: none"> <li>. heurístico, estático para tarefas <i>hard</i>, dinâmico para <i>soft</i></li> </ul>		<ul style="list-style-type: none"> <li>. abordagem event-triggered</li> <li>. flexibilidade</li> <li>. amplo modelo temporal de tarefas</li> </ul>

pelo contrário. Ambos estão suportados por ferramentas sofisticadas de assistência ao projeto, e sua previsibilidade os fazem aplicáveis em sistemas caracterizados pelo seu altíssimo custo e criticalidade de missão (como o ônibus espacial). No resto dos casos, a disponibilidade de ferramentas maduras para o projeto e desenvolvimento das aplicações continua a ser o principal obstáculo para a adoção das técnicas avançadas propostas nas pesquisas de tempo real nas arquiteturas de hardware mais convencionais. Esta é a chave do sucesso dos SO UNIX-like, estado da arte atual de grande parte das aplicações de tempo real de pequena e média escala.

As vantagens de um projeto orientado a objetos justificam sua adoção permanente, independentemente das outras características. Arquiteturas baseadas em micronúcleo oferecem grande facilidade para a reconfiguração estática e dinâmica, conformando núcleos de características e tamanho mais adaptados à cada situação. Em muitos suportes percebe-se um esforço importante em direção a configurabilidade dos seus comportamentos, principalmente através da escolha dos mecanismos mais apropriados para a aplicação em mãos (como o caso das políticas de escalonamento em Chaos). Isto é devido à inexistência de recursos suficientemente gerais para suportar todos os casos possíveis na prática com igual eficiência (p.ex., chamada a procedimentos vs. IPC vs. RPC).

Este estudo não foi exaustivo nem todos os casos foram tratados com o mesmo detalhe (principalmente pelo diferente nível técnico da bibliografia achada), pelo que a comparação final pode não ser totalmente justa. Mesmo assim, o objetivo foi dar uma idéia das abordagens mais importantes, e quais suas principais características estruturais.

### 3.5. CONCLUSÃO

Neste capítulo foram apresentadas algumas abordagens ao problema de tempo real, junto a suportes de execução específicos que as suportam. Nenhuma delas é suficiente para o tratamento integral das necessidades que podem advir de um projeto concreto; algumas, como Chaos, apresentam características que as colocariam em várias categorias simultaneamente, embora apenas uma foi salientada.

As duas abordagens principais de processamento também foram discutidas: o núcleo Spring foi apresentado como representante de sistemas *event-triggered*, caracterizando-se pela flexibilidade que oferece e sua sofisticada estrutura de escalonamento. Descreveu-se depois o Mars como expoente da abordagem *time-triggered*, destacando-se pelo determinismo, suporte de tolerância a falhas e sua metodologia de projeto, amplamente suportada por um completo ambiente de desenvolvimento.

Arquiteturas monolíticas para tempo real (LynxOS) e de micronúcleo (Chorus) foram

analisadas. Um modelo genérico de objetos de tempo real foi apresentado, junto com sistemas operacionais que implementam características interessantes dentro desse paradigma (Choices -particularização de núcleos, Arts -protocolo de cerca de tempo, Chaos -adaptabilidade e R-Shell -gerenciamento cooperativo de garantias).

# CAPÍTULO 4

## CONCEPÇÃO DE UM NÚCLEO DE TEMPO-REAL SEGUNDO A ABORDAGEM SÍNCRONA

Suportes de execução como os vistos no capítulo anterior são exemplos de *sistemas reativos*, caracterizados pela interação forte com seu ambiente em tempo real.

A abordagem síncrona é uma técnica relativamente recente para a implementação deste tipo de sistemas, oferecendo uma série de vantagens em relação às abordagens assíncronas convencionais. Estas vão desde a sustentação formal das linguagens que a suportam até a melhora na qualidade do resultado, segundo os parâmetros da Engenharia de Software.

Este capítulo desenvolve os conceitos anteriores e discute a linguagem síncrona de implementação Esterel. Apresenta depois um projeto de núcleo de tempo real simples segundo a abordagem síncrona, visando obter as vantagens mencionadas acima para este caso específico.



## 4.1. SISTEMAS REATIVOS

O computador de controle mostrado na Figura 2.1 pode ser visto como um *Sistema Reativo* [Berry88] [Benveniste91] [Halbwachs91] [Boussinot91], caracterizado pela interação forte e contínua com seu ambiente. Os dois elementos influenciam-se de forma recíproca: o sistema reativo reage a estímulos ou mudanças sentidas no seu ambiente, eventualmente agindo sobre ele (Figura 4.1). Neste contexto, *Sistemas de Tempo Real* seriam sistemas reativos cujas interações devem ainda cumprir restrições temporais, impostas pelo meio que devem controlar.

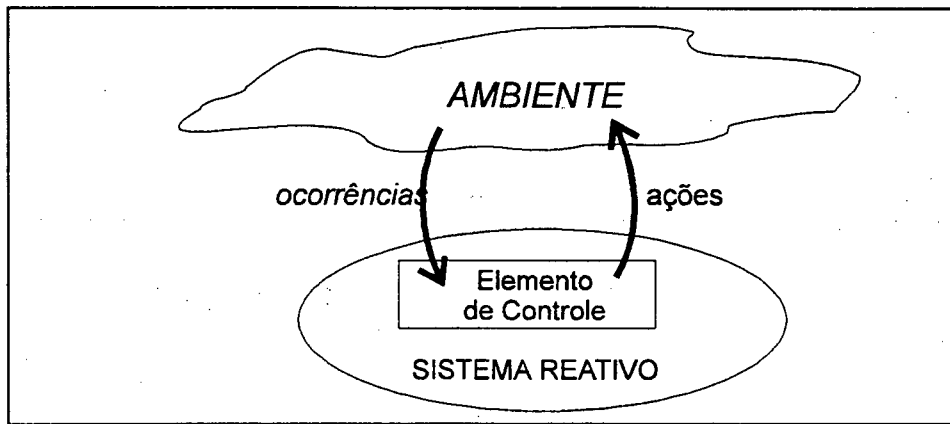


Figura 4.1: Arquitetura Reativa e interações com o ambiente

Dentro do elemento reativo, a arquitetura reativa distingue a *componente de controle* -que coordena o sistema e decide as ações a serem executadas frente a estímulos externos- do resto -que oferece à componente de controle suporte a nível de interface com o meio, recolhendo as informações externas e executando suas ordens. Ao separar o elemento de política do mecanismo de ação, este modelo geral aporta flexibilidade, permitindo mudar qualquer um sem alterar o outro. A lógica da política pode ser trocada por outra mais eficiente, e o mecanismo pode ser melhorado e refinado de forma independente. Qualquer otimização na componente de controle beneficiará simultaneamente a todas as aplicações onde esta seja reutilizada [Wood92].

As arquiteturas reativas são aplicadas principalmente em automação industrial e tratamento de sinais, mas não estão apenas limitadas a aplicações em tempo real: interfaces homem-máquina, protocolos de comunicação, etc, são apontados como sistemas reativos [Benveniste91]. Em [Wood92], por exemplo, o modelo de arquitetura reativa é usado num suporte para o gerenciamento de aplicações distribuídas sobre uma rede local.

## 4.2. A ABORDAGEM SÍNCRONA DE IMPLEMENTAÇÃO

A *abordagem assíncrona* de execução, explicitada nos modelos de programação de CSP, OCCAM, DP e ADA [Andrews83], decompõe o sistema (reativo) em tarefas, unidades de concorrência que evoluem fracamente acopladas. A comunicação entre tarefas é, portanto, assíncrona devido aos tempos arbitrários que se passam entre a ativação de uma primitiva de comunicação e sua conclusão.

Uma técnica recente para a implementação de sistemas reativos é a *abordagem síncrona*, onde as atividades concorrentes evoluem fortemente acopladas [Berry93]. Esta abordagem é baseada na *hipótese de sincronismo* [Benveniste91]:

- as saídas (reações) são síncronas às entradas, isto é, são produzidas simultaneamente a estas;
- as ações e processamentos internos são instantâneos, isto é, não levam tempo observável para se desenvolver;
- as comunicações se dão por difusões instantâneas (*instantaneous broadcasting*).

Em resumo, podemos caracterizar um sistema síncrono como se executando sobre uma máquina infinitamente rápida. A principal consequência desta idealização é uma simplificação conceitual que facilita o modelamento e análise formais do sistema reativo.

A abordagem síncrona pode ser aplicada a qualquer sistema reativo: basta demonstrar que o sincronismo é uma abstração válida para a sua implementação. Na prática, deve-se conferir que o controle completa sempre sua reação a um acontecimento externo antes da chegada do próximo evento, para um ambiente dado [Halbwachs91].

### 4.2.1. LINGUAGENS SÍNCRONAS

Na abordagem síncrona, o elemento de controle do sistema é identificado e depois programado numa *linguagem síncrona*. Este tipo de linguagem não é de propósito geral, como C ou Ada, mas serve apenas para a programação do elemento de controle de sistemas reativos.

Existem várias linguagens síncronas. As *imperativas*, como CSML [Clarke91] e Esterel [Boussinot91], utilizam um formalismo baseado em *estados* e portanto são mais aptas para a programação de aplicações onde o fluxo de controle predomina sobre o de dados. Linguagens síncronas *equacionais*, como LUSTRE [Halbwachs91] e SIGNAL [LeGuernic91], modelam a

aplicação como um sistema de equações recorrentes e têm aplicabilidade onde o fluxo de dados é o que prevalece, que é o caso em processamento de sinais. A Figura 4.2 mostra exemplos de código dos dois tipos de linguagens.

a) Código Lustre para um filtro de segunda ordem [Halbwachs91]

```

const a,b,c,d,e: real.

node SECOND_ORDER(x: real) returns (y: real);
var u,v: real;
let
  y = a*x + (0.->pre(u));
  u = b*x - d*y + (0.->pre(v));
  v = c*x - e*y;
tel.

```

b) Código CSMIL para um monitor de acesso a memória [Clarke91]

```

loop
  switch
    case IQ_st == EMPTY:
      compress read(PC-MAB, MDB-IQ) endcompress;
      break
    case TS_st == INVALID & !push-req:
      compress lower(push-rdy); read(SP-MAB, MDB-TS) endcompress;
      break
    case TS_st == MODIFIED & !pop-req:
      compress lower(pop-rdy); write(SP-MAB, TS-MDB) endcompress;
      break
    default:
      skip;
  endswitch
endloop

```

Figura 4.2: Exemplos de código síncrono.

A semântica das construções destas linguagens é descrita matematicamente, o que possibilita a realização de provas de correção e validação do código gerado. O formalismo da semântica permite que seus compiladores utilizem algoritmos especiais para a detecção e rejeição de programas não deterministas, pelo que o processo de compilação pode ser visto como um procedimento formal de validação da hipótese de sincronismo [Benveniste91].

A previsibilidade é uma característica distintiva da abordagem síncrona em relação ao modelo assíncrono. O código gerado é determinista, no sentido que uma sequência determinada de eventos de entrada produz sempre a mesma sequência de eventos de saída<sup>1</sup> [Benveniste91]. Este determinismo

<sup>1</sup> No caso das linguagens síncronas imperativas, o determinismo implica que para qualquer estado do sistema e qualquer evento de entrada, exista apenas um único evento de saída [Berry93].

diminui grandemente o número de comportamentos possíveis do sistema, facilitando sua análise automatizada.

## 4.2.2. A LINGUAGEM ESTEREL

Esterel [Berry88] [Berry93] é uma linguagem síncrona imperativa e paralela com capacidade de modularização hierárquica, que serve para descrever os módulos de um sistema reativo, suas interfaces e conexões com outros módulos e com o ambiente externo.

Módulos Esterel podem ser montados a partir da composição síncrona de outros módulos. Módulos comunicam-se entre si e com o ambiente por meio de *sinais*, que podem visar a simples sincronização ou ainda transferência de dados, e *sensores*, canais de informação assíncronos. Um módulo é ativado por um *evento de entrada*, formado pelos sinais da sua interface presentes nesse instante; o módulo reage executando seu corpo e eventualmente emitindo sinais de saída. Por ser uma linguagem síncrona, assume-se que a execução da reação é instantânea; instruções Esterel só consomem tempo se sua semântica assim o determina [Berry88].

Esterel não é uma linguagem de propósito geral, não oferecendo suporte para tratamento de dados. Quando necessário, este é resolvido por chamadas a funções e procedimentos externos que, diferentemente dos sinais, não possuem poder sincronizante.

A compilação Esterel gera uma máquina de estados finita ou autômato, onde o paralelismo, sincronismo e comunicação presentes no programa fonte são serializados pelo compilador. O código gerado é garantido ser determinista pelo compilador [Berry88], que também garante sua equivalência semântica com o programa fonte.

O autômato resultante pode ser validado, simulado e ainda executado (WYPIWYE - What You Prove Is What You Execute [Berry89]): por isso, o programa fonte é considerado uma *especificação executável*. A *verificação lógica* é feita confrontando o autômato com uma especificação na forma de lógica temporal ou de autômato [Benveniste91]. Em aplicações de tempo real é preciso uma *verificação temporal*: constroi-se um grafo a partir da estrutura do programa e anotam-se seus ramos com os tempos de execução, sobre a máquina alvo, das instruções Esterel correspondentes [André93b]; como o código gerado não é recursivo, o tempo máximo para cada reação é finito e pode ser determinado com precisão analisando o grafo resultante.

## 4.2.3. ARQUITETURA DE UMA APLICAÇÃO ESTEREL

O autômato final é gerado pelo compilador Esterel e fornecido sob a forma de um programa

numa linguagem de programação convencional (como C ou Forth), acessível através de um conjunto de chamadas [CISI88Intrfc].

Para funcionar, o autômato precisa ser complementado com uma *máquina de execução*, encarregada da interface com o exterior assíncrono, e de uma *subcamada de dados*, elemento passivo que contém as implementações dos dados, funções e procedimentos referenciados pelo autômato [André93a]. Ambos elementos são providos pelo usuário e específicos para cada aplicação (Figura 4.3).

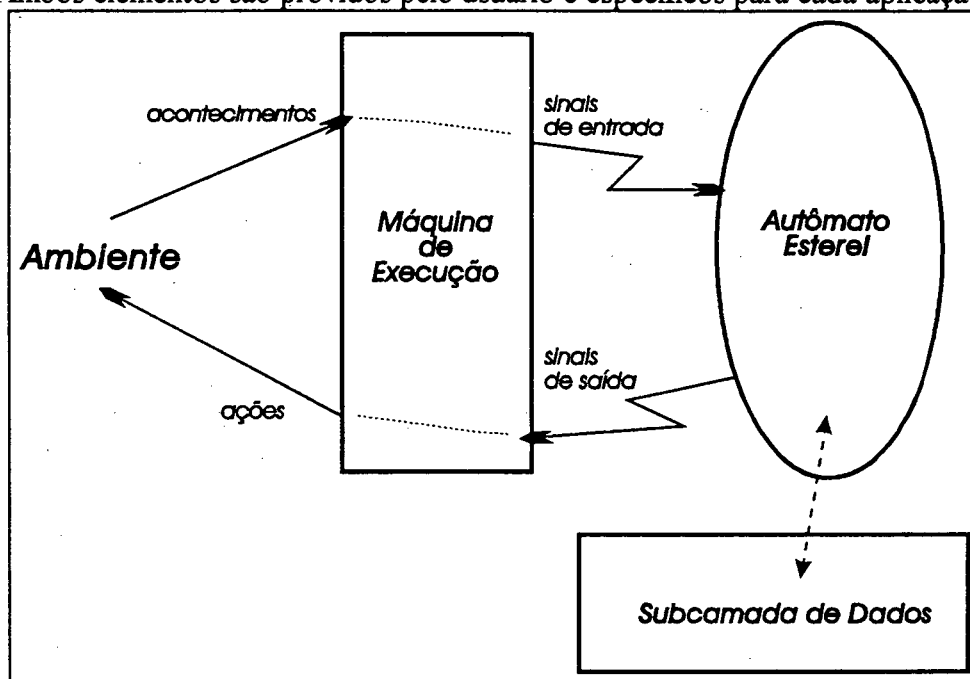


Figura 4.3: Arquitetura de uma Aplicação Esterel

Frente a um acontecimento externo, o autômato toma as decisões, mas não as executa. A máquina de execução o isola do seu ambiente: quando um fato relevante acontece, ela gera os sinais de entrada correspondentes e ativa o autômato. Os sinais de saída decorrentes da sua evolução são interpretados pela máquina de execução e traduzidos em ações sobre o ambiente, conforme sua semântica [André93a]. A máquina de execução deve ainda garantir as relações de exclusão entre sinais de entrada definidas no programa fonte. Na prática, o retardo introduzido pela máquina de execução entre um acontecimento físico e sua conversão em sinais equivalentes produz um certo assincronismo como ambiente cujo alcance pode ser medido e simulado [André93b] [André93c].

### 4.3. UM NÚCLEO DE TEMPO REAL SEGUNDO A ABORDAGEM SÍNCRONA

Na continuação apresentamos a especificação de núcleo de tempo real visando-o como um sistema reativo, para depois descrever o projeto do seu elemento de controle segundo a abordagem

síncrona. Nesta aplicação, o controle do núcleo é o módulo encarregado principalmente de definir as ações a executar frente a determinadas requisições de serviço pelas aplicações rodando por cima. Finalmente, propõe-se uma decomposição modular do elemento de controle.

Note-se que um núcleo de tempo real *não é* uma aplicação de tempo real: ele não deve cumprir restrições temporais de classe alguma. Sua grande obrigação é oferecer seus serviços de forma previsível (isto é, respeitando um tempo máximo para sua execução). Com este embasamento e o conhecimento das características temporais do suporte, das tarefas da aplicação e do meio externo, o algoritmo de escalonamento deve garantir o cumprimento das restrições temporais impostas pelo próprio meio através do escalonamento adequado dos recursos do sistema (que incluem a CPU).

### 4.3.1. ESPECIFICAÇÃO DE UM NÚCLEO DE TEMPO REAL

O núcleo a especificar será multitarefa e monousuário, com uma interface de aplicação e semântica de primitivas semelhante a dos núcleos convencionais. Suas principais primitivas são mostradas no Quadro IV-I, e escolhidas por serem serviços simples, ortogonais e representativos dos encontrados nos núcleos clássicos.

GERENCIAMENTO DE TAREFAS:	
* <i>task_id_t</i>	<i>spawn(task_descriptor *task)</i>
. <i>task_id_t</i>	<i>getid(void)</i>
* <i>void</i>	<i>kill(task_id_t task)</i>
SERVIÇOS TEMPORAIS:	
* <i>int</i>	<i>delay(No_ticks)</i>
* <i>void</i>	<i>wakeup(task_id_t task)</i>
COMUNICAÇÃO E SINCRONIZAÇÃO:	
* <i>void</i>	<i>send(mssg_t *mssg, mbox_t *mbox)</i>
* <i>mssg_t *</i>	<i>receive(mbox_t *mbox, time_t timeout)</i>
* <i>int</i>	<i>P(semaphore_t *sema, time_t timeout)</i>
* <i>void</i>	<i>V(semaphore_t *sema)</i>

Quadro IV-I: Principais primitivas suportadas pelo NTR.

O modelo de programação está baseado na decomposição da aplicação num conjunto de tarefas executando sobre espaços de endereçamento individuais, e se comunicando e sincronizando através das primitivas (*send()*, *receive()*) e (*P()* e *V()*) do núcleo. As decisões do núcleo estarão baseadas em informação temporal explícita sobre as tarefas da aplicação.

Das primitivas suportadas distinguem-se as *reativas* (marcadas com estrelas \*), como aquelas cuja execução pode vir a modificar o estado de alguma tarefa dentro do sistema. Note-se que, para as

chamadas potencialmente bloqueantes (como *receive()* ou *P()*), a tarefa chamadora deve especificar um *timeout*, máximo tempo que deseja ficar bloqueada: se após esse tempo a chamada não for satisfeita, ela é abortada e a tarefa volta ao estado de pronta. Esta última característica é desejável pois aumenta a previsibilidade do sistema. A Figura 4.4 mostra os estados possíveis de uma tarefa e suas transições quando uma primitiva reativa é executada<sup>2</sup>.

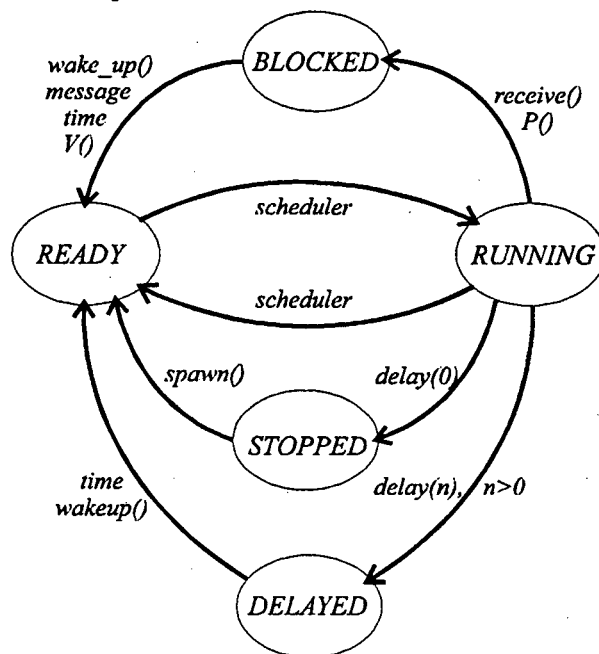


Figura 4.4: Estados das tarefas e transições entre eles.

As necessidades de recursos para a execução das primitiva do núcleo devem ser conhecidas e limitadas; estes recursos incluem tempo de CPU (tempo máximo de execução) e tamanho do *stack*. Quando se dispõe desta informação e da descrição da seqüência de chamadas ao sistema para cada tarefa da aplicação, testes de escalonamento mais sofisticados podem ser aplicados pelas políticas. Estas informações serão lidas de arquivos de configuração durante a iniciação do núcleo.

O escalonador do núcleo será configurável pelo usuário, que poderá escolher a política de escalonamento entre earliest deadline, menor folga, taxa monotônica e urgência máxima; também oferecerá suporte para as políticas definidas pelo projetista, mas não para mudança dinâmica de escalonamento.

A alocação de memória será estática, e provida na hora de carregar o núcleo. Não será fornecido sistema de arquivos nem mecanismos de gerenciamento de memória virtual a nível de interface. O primeiro foge dos objetivos do trabalho; no segundo caso, não se conhecem mecanismos de paginação que atuem em tempo real.

<sup>2</sup> a requisição de uma chamada *kill()* (não mostrado na figura) leva a tarefa de qualquer estado ao estado KILLED

O núcleo deve fornecer mecanismos para a detecção e tratamento de falhas temporais, elemento inexistente nos núcleos não-tempo real. Heurísticas serão aplicadas para a detecção precoce destas falhas, de maneira a ter mais chances de recuperação pela ativação de procedimentos de exceção ou notificação às camadas superiores da aplicação.

O protótipo do núcleo de tempo real a implementar privilegiará os aspectos de estrutura sobre a eficiência. A plataforma alvo escolhida foi o PC, pelo completo domínio que permite sobre seu conjunto; o processador rodará em modo real e portanto sem proteção de hardware. Outras alternativas como plataformas UNIX foram descartadas pela falta de controle que se tem sobre o hardware.

### **4.3.2. O NÚCLEO ESPECIFICADO VISTO COMO SISTEMA REATIVO**

A estrutura reativa é aplicada a um núcleo de tempo real identificando seu hardware e as tarefas da aplicação com o "ambiente", e o núcleo como a "estrutura de controle".

O núcleo interage com seu ambiente de várias e complexas maneiras. As tarefas da aplicação têm acesso aos serviços do NTR pela sua interface, composta de chamadas a primitivas. A ativação de serviços faz com que o núcleo reaja executando ações internas (atualização das suas estruturas de dados) ou externas (mudança da tarefa corrente, reprogramação do hardware ou acionamento de dispositivos externos). Elementos do hardware (relógio, operações de E/S, exceções, interrupções) ativam no núcleo códigos tratadores ou ainda tarefas da aplicação. Estas interações são praticamente permanentes e caracterizadas pela sua reciprocidade; portanto, com a escolha do ambiente/estrutura de controle feita acima, pode-se encarar um NTR como um sistema reativo.



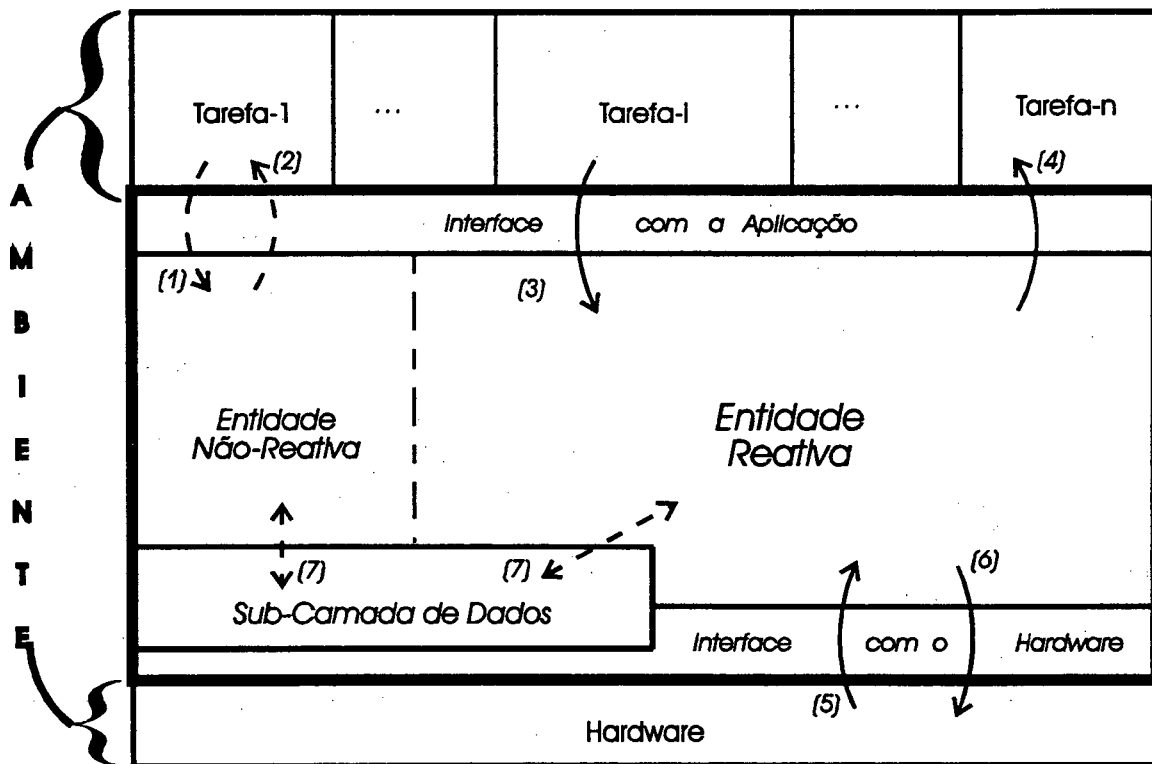


Figura 4.5: Arquitetura do NTR e Interações com seu Ambiente

- requisições reativas
- - - -> requisições não-reativas
- ← - -> chamadas a funções/procedimentos

A Figura 4.5 mostra o nosso NTR segundo a concepção reativa apresentada acima, junto às principais interações com seu ambiente. Podemos distinguir:

- \* o *elemento reativo*, implementando a lógica das primitivas reativas,
- \* o elemento *não-reativo*, encarregado de implementar a lógica do resto das primitivas,
- \* a *subcamada de dados*, que oferece suporte de dados aos elementos anteriores,
- \* o *ambiente* do núcleo, composto pelas tarefas da aplicação e o hardware da máquina alvo, e separados do núcleo pelas respectivas interfaces.

O elemento reativo implementa apenas a tomada de decisões importantes frente a acontecimentos externos como invocações às chamadas reativas por parte das tarefas. As tarefas requisitam os serviços do núcleo através da Interface com a Aplicação, e a Interface com o Hardware permite o acesso ao hardware e a sinalização de eventos acontecendo nele.

A seta (1) sinaliza chamadas a primitivas não reativas e os valores de retorno (2). Elas são resolvidas pela entidade não reativa no estilo convencional, sem necessidade de interação com a entidade reativa pela falta de "reatividade" da primitiva invocada. Exemplos típicos são as chamadas

para iniciação e recuperação de informação (como *get\_id()*).

A seta marcada com (3) simboliza chamadas a serviços reativos (como *send()*), e portanto são resolvidas pela entidade reativa: a interface da aplicação ativa a entidade reativa, após a conversão da chamada e seus parâmetros numa forma adequada para o seu processamento. Como consequência da chamada, o elemento reativo pode vir modificar o estado de uma tarefa da aplicação ((4)-p.ex., um eventual desbloqueamento de uma tarefa que aguardava essa mensagem).

A seta identificada por (5) representa estímulos gerados pelo suporte (como no caso da interrupção de relógio), e a (6) ações sobre o hardware (como sua reprogramação) provocadas pela reação da entidade reativa aos primeiros. Finalmente, as setas sinalizadas com (7) são manipulações de dados invocadas pelas entidades reativa e não reativa do núcleo e servidas pela subcamada de dados comum (chamada a funções e procedimentos internos, e retorno dos resultados).

Deve-se salientar que as características de sincronismo do comportamento do elemento reativo do núcleo são internas ao núcleo e transparentes às tarefas da aplicação, que enxergam apenas uma interface oferecendo serviços com semântica equivalentes às de um núcleo convencional. O sincronismo refere-se à maneira de processar a reação a eventos externos, mas não ao que deveria existir entre entidades concorrentes tentando acessar recursos compartilhados.

### 4.3.3. MODELO SÍNCRONO DE IMPLEMENTAÇÃO DA ENTIDADE REATIVA

A Figura 4.6 mostra nosso modelo de implementação síncrono para a entidade reativa do núcleo proposto, identificando nela um autômato, uma máquina de execução (ME) e a subcamada de dados descrita acima como componentes do modelo.

O autômato é o produto da compilação do programa fonte Esterel descrevendo a lógica de controle das primitivas reativas. Podemos ver que o autômato interage com seu ambiente (as tarefas da aplicação e o hardware) somente através da máquina de estados, que o encapsula e com quem se comunica exclusivamente por meio de sinais e sensores. Ele ainda invoca diretamente a subcamada de dados, para resolver manipulações de dados que não são possíveis em Esterel ou são ineficientes.

Quando uma tarefa requisita uma primitiva reativa do núcleo, a ME translada o chamado num conjunto de sinais de entrada e seus valores, formando um *evento*. Este evento é passado ao autômato, que depois é ativado. O processamento do evento pelo autômato pode determinar a emissão de sinais de saída e a chamada a funções e procedimentos do suporte de dados, num "instante de duração nula" (segundo a hipótese síncrona [Berry88]).

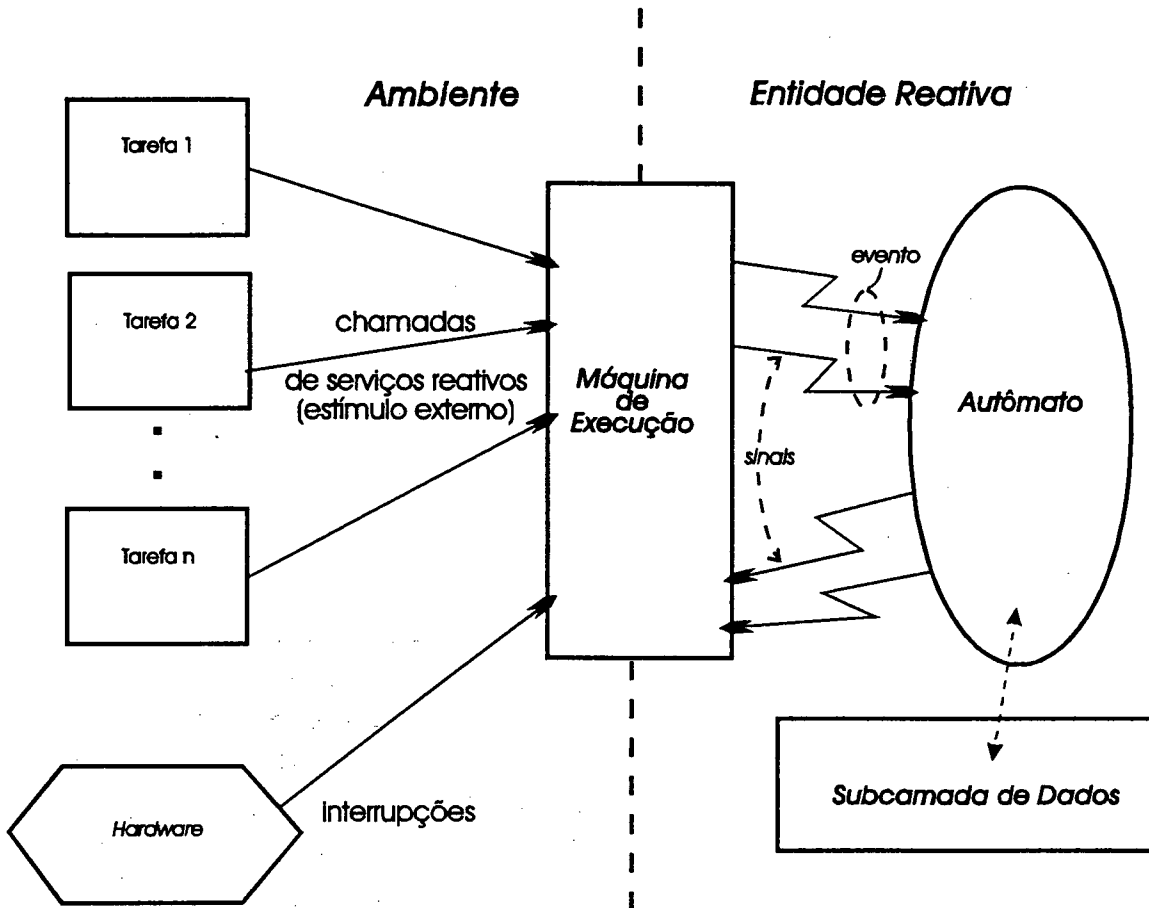


Figura 4.6: Modelo Síncrono de Implementação proposto

A ME traduz os sinais de saída emitidos pelo autômato nas ações correspondentes sobre as tarefas (típicamente mudanças no estado delas). Com o autômato novamente em repouso, a ME entrega o fluxo de controle a tarefa escolhida pelo despachante para executar e fica a espera da próxima chamada para repetir este comportamento. O processo é idêntico caso o estímulo provenha do próprio hardware ao invés de ser uma chamada reativa.

No caso dos NTRs, a ME é parte da estrutura que se mantém fixa frente a mudanças na lógica de controle; já o suporte de dados pode variar em função de mudanças nas políticas e mecanismos internos usados no núcleo.

#### 4.3.4. MODULARIZAÇÃO DA ENTIDADE REATIVA

Para o projeto síncrono do núcleo é preciso sua decomposição em módulos interconectados

(Figura 4.7). O critério nela usado foi atribuir, para cada *tipo* de primitiva mostrado no quadro IV-1, um módulo de tratamento no maior nível de abstração.

Podemos ver que o módulo de maior hierarquia, *KERNEL*, é formado a partir da composição síncrona dos módulos:

- \* **coordenador** (*COORDINATOR*): sua principal função é a iniciação, configuração, coordenação e remoção do núcleo e dos códigos da aplicação.
- \* **gerenciador de tarefas** (*TASK\_MANAGER*): suporta os serviços externos de criação e destruição dinâmica de tarefas, além do gerenciamento das filas de tarefas prontas e bloqueadas e escalonamento.
- \* **gerenciador de comunicações e sincronização** (*COURIER*): implementa o tratamento de mensagens e primitivas de sincronização.
- \* **servidor de alarmes** (*AWAKER*): implementa as chamadas relacionadas com alarmes temporais.
- \* **tratador de exceções temporais** (*TIME\_SENTINEL*): sua função é detectar, sinalizar e eventualmente tratar exceções provocadas por falhas temporais nas tarefas da aplicação, verificando que executem dentro de suas restrições temporais.

O módulo principal é disparado quando uma tarefa da aplicação requisita um serviço reativo do núcleo. *KERNEL* comunica-se com seu exterior (a máquina de estados) através de sinais<sup>3</sup> que identificam a primitiva invocada (sob a forma <nomechamada\_call> e de sensores para leituras de dados relativamente estáticos. Além deles, *KERNEL* recebe sinais do hardware (*TICK*) e emite sinais de erro (*INTERNAL\_ERROR*, *SLACK\_FAILURE*, *WRKTIME\_FAILURE*, *ED\_FAILURE*), que ativam os tratadores especializados, tarefas do usuário de alta prioridade que ficam permanentemente bloqueadas até a ocorrência de algum destes sinais, quando passam a executar as políticas de recuperação de erro definidas para cada tarefa. Informações estáticas durante a execução de uma reação são lidas através de sensores (neste caso, *mssg* e *timeout*). O Anexo 1 detalha os módulos e os códigos Esterel correspondentes.

O módulo *TASK\_MANAGER* implementa as chamadas reativas relacionadas com a criação e remoção de tarefas (*spawn()* e *kill()*), e parte dos sinais para a mudança de estados (*SPAWN*, *KILL*, *BLOCK*, *UNBLOCK*). Em determinadas situações invoca o escalonador para determinar a próxima tarefa a rodar; o escalonador é formado pela política (encarregada de definir a próxima tarefa) e o despachante externo (que faz a troca de contexto necessária). Se a lista de prontas for modificada em

---

<sup>3</sup> a figura não mostra os sinais envolvidos na iniciação do sistema.

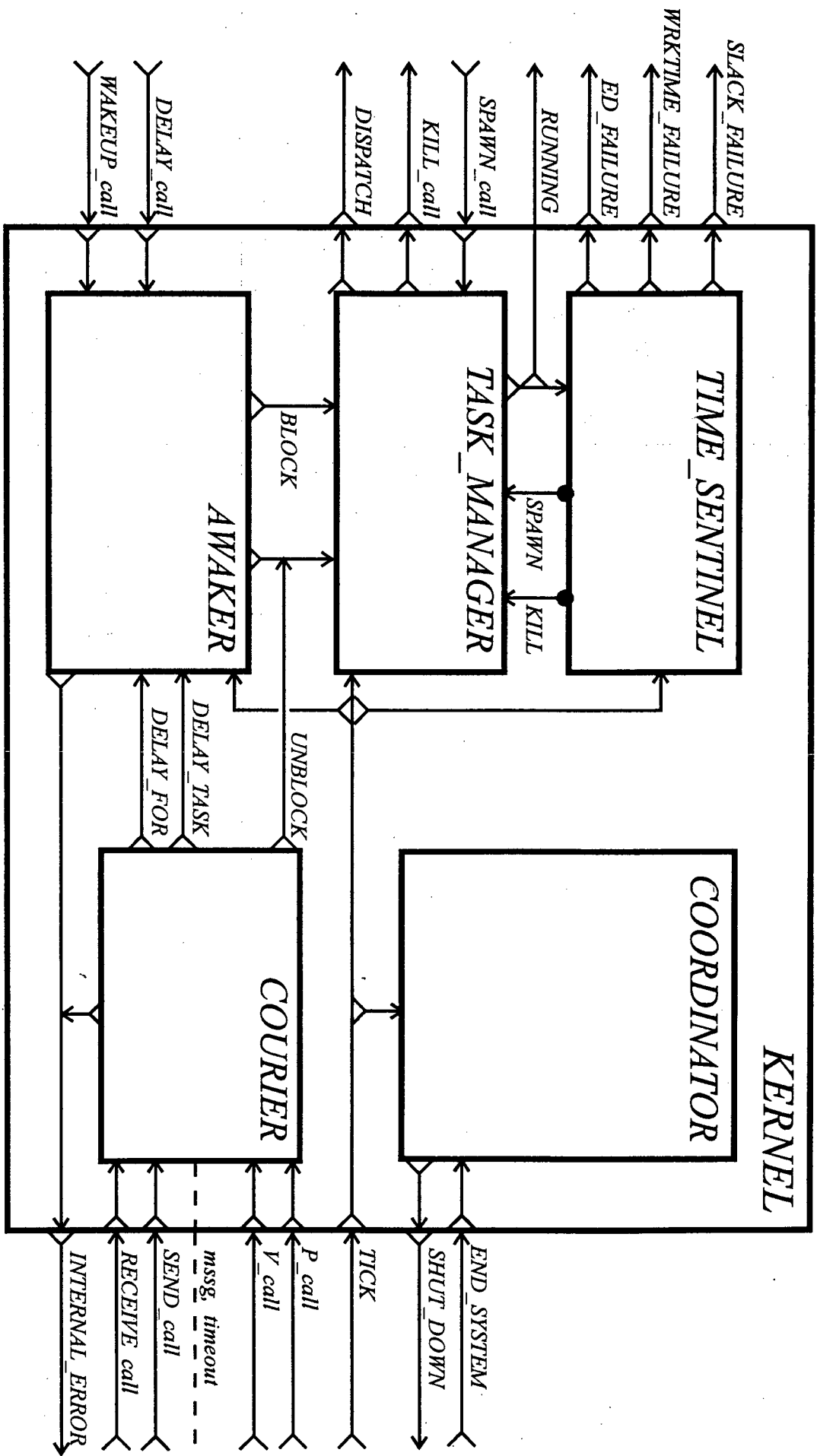


Figura 4.7: Estrutura modular do NTR e principais sinais envolvidos

decorrência da execução (isto é, uma tarefa foi acrescentada a ou retirada da lista de prontas), o *TASK\_MANAGER* ativa a política de escalonamento para mantê-la ordenada. Após a execução do autômato, o despachante é ativado para rodar a primeira tarefa da lista de prontas.

O módulo *AWAKER* gerencia a lista de alarmes do sistema, formada pelas tarefas que requisitam a primitiva *delay()*. Na requisição, a tarefa corrente é bloqueada e incorporada à lista de alarmes; após o tempo desejado, a tarefa é colocada novamente na lista de prontas. Este comportamento é o mesmo que nos núcleos convencionais, e não é determinista: o argumento da chamada especifica apenas o tempo *mínimo* que a tarefa deve estar bloqueada, podendo vir a recuperar o processador muito depois da chamada. Uma tarefa pode ser acordada antes da hora por uma chamada *wakeup()* executada por outra tarefa do sistema.

O módulo *TIME\_SENTINEL* tenta aplicar critérios especiais de maneira a detectar os erros temporais antes que se transformem em perdas de deadlines, se possível. Entre os critérios aplicados para a deteção se encontra, por exemplo, não ativar uma tarefa se carece de suficiente folga para se executar. Quando detecta uma exceção, o módulo aborta a tarefa que a provocou e ativa o procedimento *default* ou o tratador definido para a tarefa falha. Também mantém estatísticas sobre seu funcionamento.

Embora a comunicação e sincronização seja suportada a nível da aplicação por mensagens e semáforos, dentro do código síncrono do autômato são substituídas por sinais equivalentes, eventualmente valorizados. O módulo *COURIER* implementa a lógica das chamadas correspondentes, sendo formado pela composição síncrona dos módulos (*RECEIVE\_handler*, *SEND\_handler*, *P\_handler* e *V\_handler*). O tratador da comunicações implementa as chamadas *receive()* e *send()* através dos módulos *RECEIVE\_handler* e *SEND\_handler*. No primeiro caso, se houver uma mensagem na caixa de correio argumento, então o módulo escolhe a mensagem mais adequada da fila associada e entrega para a tarefa requisidora; caso contrário, a tarefa corrente é enfileirada na fila associada à caixa de correio argumento e bloqueada pelo tempo máximo especificado na chamada. No segundo caso, se houver alguma tarefa aguardando a mensagem (e portanto bloqueada), desbloqueia-a e lhe passa mensagem; senão, guarda a mensagem na caixa de correio argumento. Para sincronização, os módulos (*P\_handler* e *V\_handler*) implementam a semântica convencional para essas chamadas.

As estruturas de dados associadas a determinados objetos do núcleo (semáforos, caixas de correio, etc) são criadas e iniciadas junto com a criação da estrutura correspondente. No caso da caixa de correio, por exemplo, tem associada duas listas: uma lista de mensagens que aguardam ser lidas (de tipo *MBOX\_mssg\_queue*), e uma lista de tarefas bloqueadas por uma mensagem dessa caixa (de tipo *MBOX\_task\_queue*). Só uma delas é ativa ao mesmo tempo: não podem existir tarefas bloqueadas por mensagem quando a caixa ainda tem mensagens sem ler.

O núcleo tem uma concepção hierarquizada: cada módulo é formado pela composição síncrona de outros mais primitivos. A Figura 4.8 ilustra o exemplo do manipulador de exceções temporais, formado pelos módulos *verificadores de deadline* (*ED\_CHECKER*), *de tempo máximo de execução* (*WRKTIME\_CHECKER*) e *de folga* (*SLACK\_CHECKER*).

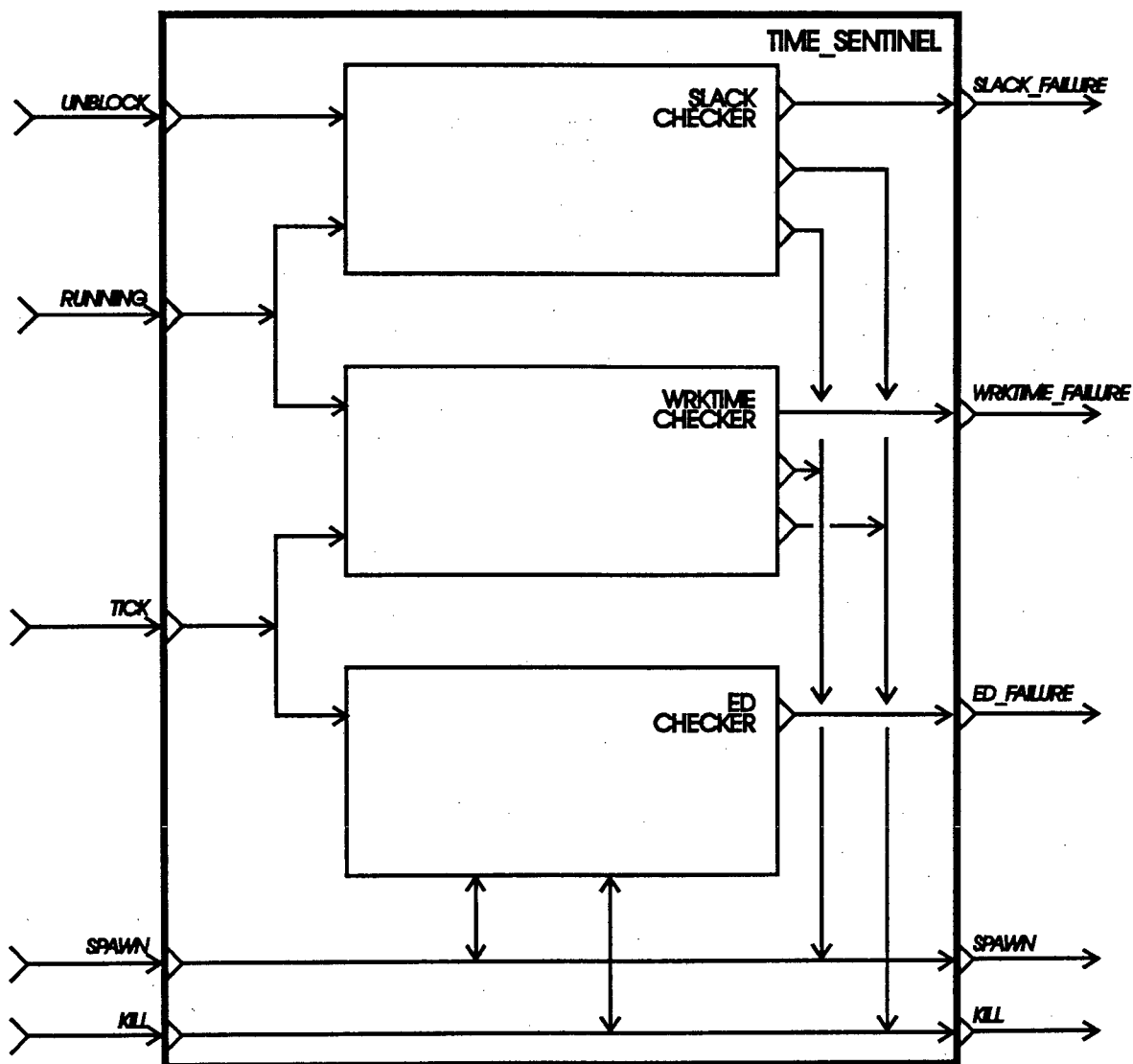


Figura 4.8: Decomposição modular do Manipulador de Exceções Temporais

O escalonador trata apenas as tarefas da aplicação, mas não o funcionamento dos módulos internos do NTR. Salvo o gerenciador de comunicações e sincronização (*COURIER*), o resto auto-escalona-se, isto é, cada módulo define o momento da sua próxima invocação e ativa-se sozinho, sem intervenção de outros módulos. Na realidade isto é uma ilusão implementada através dos operadores paralelos de Esterel. Como a execução do autômato é sequencial, não é preciso sincronizar seu acesso a estruturas de dados compartilhadas.

## 4.4. CONSIDERAÇÕES GERAIS

Um requisito necessário para ter um comportamento previsível numa aplicação é rodá-la em um suporte previsível. Num núcleo de tempo real, a previsibilidade implica que suas primitivas e reações apresentem comportamentos definidos e limites precisos para seus tempos de execução. Só assim os testes de escalabilidade, baseados nestas informações e nas características da aplicação, podem fornecer garantias realistas sobre o cumprimento das restrições temporais do sistema.

Acreditamos que a aplicação da abordagem síncrona na implementação de um núcleo de tempo real traz benefícios significativos às aplicações por ele controladas: além do determinismo, o código gerado apresenta bem menos erros (por causa das provas formais de correção que podem ser feitas) e é ainda mais estruturado (sua camada de controle é totalmente separada do resto); ferramentas automáticas podem ser usadas para determinar o tempo de execução máximo para cada primitiva ou função do núcleo. Por outro lado, a modularidade das linguagens síncronas permite que a concepção e implementação do núcleo sejam feitas por refinamentos sucessivos a partir da especificação inicial; programá-lo usando uma linguagem com operadores paralelos como Esterel possibilita usar estruturas concorrentes onde antes não existia um suporte para isso: no código fonte do próprio núcleo.

Os passos seguidos no projeto síncrono do núcleo (especificação, mapeamento numa estrutura reativa, modelagem síncrona da entidade reativa, e finalmente sua modularização) podem ser tomados como indicativos no desenvolvimento de qualquer aplicação reativa segundo a abordagem síncrona.

## 4.5. CONCLUSÃO

Neste capítulo foi apresentado o conceito de sistema reativo e as características das interações com seu ambiente, junto com as vantagens encontradas na concepção de uma aplicação segundo a visão reativa. Sistemas Reativos podem ser implementados segundo a abordagem síncrona, que pela Hipótese de Sincronismo consegue vantagens estruturais que facilitam o tratamento formal do resultado. As características das linguagens de programação síncronas em geral, e de Esterel em particular, foram analisadas. O resultado de uma compilação Esterel (um autômato) precisa de uma estrutura de apoio complementar para executar, que foi detalhada.

Especificou-se um núcleo de tempo real convencional e pequeno, visando seu projeto segundo a abordagem síncrona. Após a identificação do seu elemento reativo, aplicou-se a arquitetura descrita acima para chegar a um modelo de implementação síncrono. Finalmente, a entidade reativa do núcleo foi modularizada e descreveram-se cada um dos seus componentes.



# **CAPÍTULO 5**

## **ASPECTOS PRÁTICOS DO DESENVOLVIMENTO DO NÚCLEO**

A implementação de um núcleo de tempo real segundo a abordagem síncrona é um processo complexo que motiva sua abordagem em fases incrementais com sua correspondente validação. Por outro lado, a inexistência de algoritmos de escalonamento e de estruturas de tempo real suficientemente genéricas com custo de execução razoável faz necessária a configurabilidade do núcleo nestes aspectos.

Neste capítulo serão detalhados os aspectos práticos mais significativos relacionados com a implementação do nosso núcleo, junto a uma descrição das diferentes fases que se seguiram no seu desenvolvimento. Apresenta-se também uma metodologia para a criação de núcleos particularizados para aplicações específicas, segundo a escolha pelo usuário do algoritmo de escalonamento e dos comportamentos internos do núcleo a partir do estudo do modelo temporal da aplicação a desenvolver.

## 5.1. ASPECTOS DE IMPLEMENTAÇÃO

A seguir serão descritos os principais aspectos da implementação do núcleo de tempo real proposto no capítulo anterior segundo a abordagem síncrona, a partir da modificação de um suporte de execução pré-existente para suportar a Hipótese de Sincronismo e algoritmos de escalonamento para tempo real. Descreve-se o ambiente de desenvolvimento envolvido, as modificações arquiteturais no suporte de execução, e finalmente a validação da hipótese síncrona sobre o resultado, junto à relação das fases seguidas para o desenvolvimento.

### 5.1.1. AMBIENTE DE DESENVOLVIMENTO ESTEREL

A linguagem escolhida para a programação síncrona do núcleo foi Esterel<sup>1</sup>. Seu ambiente [Boussinot91] é formado pelo seu compilador e um conjunto de ferramentas. O compilador Esterel translada o código fonte síncrono num autômato expresso como um procedimento escrito numa linguagem procedural (como C, Ada ou Forth). Os estágios intermediários de compilação [CISI88Doc] utilizam formatos padronizados, o que permite fazer combinações com códigos síncronos gerados por outras linguagens (por exemplo, os produzidos por Lustre [Halbwachs91]).

Para a depuração, conta-se com um simulador básico de execução (*csimul*) e sua versão gráfica (*xsimul*). Eles são apenas máquinas de execução especiais para interfaceamento com o projetista, que permitem conferindo o estado, reações habilitadas, valor das variáveis, etc, do autômato a cada momento. O simulador não trabalha a nível do código fonte, mas toma o código gerado pelo compilador Esterel e acrescenta-o de funções de entrada/saída que permitem seu controle pela consola do projetista, ao invés do ambiente alvo. A simulação não precisa de outra informação do usuário além da definição dos tipos de dados envolvidos e as funções e procedimentos da subcamada de dados que são referenciados. Pelo determinismo da abordagem, sequencias específicas de sinais podem ser definidas e reproduzidas a qualquer momento e ainda de maneira automática, facilitando assim a depuração de código pseudo-paralelo; esta facilidade não existe na contrapartida assíncrona.

Programas Esterel são validados confrontando os autômatos gerados com um conjunto de especificações. Utiliza-se a ferramenta *Auto*, que a partir da abstração de comportamentos relevantes é capaz de conferir relações de equivalência de distintos tipos entre o autômato original e um outro autômato, este último descrevendo a especificação a respeitar [Boussinot91]. O autômato reduzido pode ser visualizado com *Autograph*, possibilitando conferir visualmente seu comportamento pela análise de sua estrutura.

---

<sup>1</sup> A versão de Esterel utilizada no projeto foi a 3.23.

[André93b] e [André93c] descrevem simuladores da máquina de execução que permitem, a partir da descrição temporal das suas características, determinar e simular as distorções temporais (em relação à hipótese síncrona) introduzidas no sistema pelo retardo entre um acontecimento físico e sua conversão em sinais equivalentes.

### 5.1.2. SUPORTE BINÁRIO

Desde o começo definiu-se que a implementação do núcleo seria um protótipo, o que permitiria concentrarmo-nos nos aspectos estruturais da aplicação da abordagem síncrona, ao invés de na eficiência da sua execução ou na abrangência de suas primitivas. Pela mesma causa foi escolhido implementar o núcleo síncrono reutilizando um *Suporte Binário* pré-existente: CTask [Wagner90], um executivo multitarefa de domínio público que roda por cima do PC-DOS. Esta escolha foi baseada na qualidade da informação e disponibilidade do código fonte completo do CTask.

No seu modelo de programação, uma aplicação CTask é um conjunto de tarefas que executam pseudo-concorrentemente e que acessam as primitivas CTask através de chamadas a procedimentos. Cada tarefa tem uma função C associada que contém seu código. CTask é fornecido como uma biblioteca C, com a qual são ligados os módulos objeto das tarefas da aplicação, funções. A função *main()* ou equivalente deve ser provida pelo usuário, que logo no começo deve instalar e iniciar CTask. Após isso, cada tarefa é criada a partir da sua função, junto a outras informações como sua prioridade, e começa sua execução.

O escalonamento em CTask é preemptivo e baseado em prioridades. CTask carece de escalonador: a cada ponto de escalonamento, o despachante CTask toma a tarefa pronta de maior prioridade e a coloca para executar. As prioridades são definidas pelo próprio usuário *off-line*, e podem ser modificadas dinamicamente. Os próprios servidores de interrupção são tarefas de alta prioridade. Quando acontece uma interrupção, o hardware executa o código de tratamento associado: este, após coletar os dados relevantes sobre o evento, desbloqueia a tarefa servidora da interrupção e retorna o controle ao despachante. A seguir executará o servidor, se tiver a maior prioridade entre as tarefas habilitadas do sistema.

### 5.1.3. INTEGRAÇÃO DO ELEMENTO REATIVO

Para nosso projeto, CTask devia ser convertido num núcleo de tempo real verdadeiro e depois "sincronizado", isto é, seu elemento de controle implementado segundo a abordagem síncrona. O primeiro passo impõe a aplicação de uma política de escalonamento de tempo real, suficientemente abstrata como para poder mudá-la com facilidade, por cima do despachante de CTask. Desta maneira,

obriga-se o comportamento final do sistema a seguir as decisões de um algoritmo de escalonamento de tempo real. O segundo ponto implica a inserção do CTask no modelo síncrono de implementação, tentando modificar o menos possível sua estrutura original.

### 5.1.3.1. CONVERSÃO DO SUPORTE PARA TEMPO REAL

Como o CTask é movido por prioridades, nossa abordagem foi modificar sua estrutura através de um mapeamento das decisões do escalonador num esquema de prioridades. As prioridades de todas as tarefas da aplicação são definidas num valor default (*prioridade de prontas*), e definiu-se o nível *escolhida* como a prioridade (maior que o nível de *prontas*) da tarefa selecionada para executar pelo algoritmo de escalonamento. A tarefa escolhida para executar termina ganhando uma prioridade maior do que as outras e portanto obtém efetivamente o processador.

As ativações do despachante redirigiram-se ao algoritmo de escalonamento de tempo real (já integrado ao código de CTask): assim, toda vez que o despachante é invocado, a política executa antes e decide se a tarefa corrente deve continuar rodando ou não. Se não, a política fornece a próxima tarefa a executar, a tarefa corrente vê sua prioridade reduzida ao nível de *prontas*, e a prioridade da tarefa-resultado elevada ao nível de *escolhida*. A seguir é invocado o despachante, que rodará a tarefa pronta de maior prioridade: esta não necessariamente será a determinada pelo algoritmo, podendo ser uma tarefa CTask servidora (de maior prioridade) que a escolhida pelo escalonador. A própria política de escalonamento executa num nível alto de prioridade, apenas superados pelos servidores do sistema. Deve-se salientar que o escalonador tem controle apenas sobre a execução das tarefas da aplicação, mas não sobre as do sistema.

A previsibilidade temporal do sistema não é prejudicada pela presença das tarefas servidoras de CTask, cujo número é extremamente reduzido (duas ou três) e seu tempo de execução pequeno (CTask foi projetado para ser eficiente). Os comportamentos dos servidores oferecem a funcionalidade mínima necessária, sem maiores pretensões. Portanto, podemos ignorar sua presença com segurança, desde que não vão introduzir erros grandes nos cálculos de escalonabilidade. Em todo caso, seu efeito poderia ser levado em conta a partir da sua modelagem matemática e incorporação no algoritmo de escalonamento utilizado. Desprezamos, também, o tempo necessário para troca de contexto, desde que não aparece nos modelos dos algoritmos implementados.

A política de tempo real é obedecida, então, através da modificação dinâmicas das prioridades das tarefas da aplicação em função das decisões do algoritmo de escalonamento. Portanto fica claro que o núcleo resultante desta abordagem será tão previsível quanto sejam a execução das primitivas e o comportamento do suporte CTask. A previsibilidade a nível de núcleo exige o comportamento previsível das suas primitivas, e tempos máximos de execução para elas, de maneira a possibilitar testes de escalonabilidade precisos. O tempo de execução das primitivas de CTask é garantidamente pequeno

e portanto é desprezado, e o comportamento de suas primitivas é assumido previsível.

### 5.1.3.2. "SINCRONIZAÇÃO" DO SUPORTE

Nossa abordagem para "sincronizar" o núcleo foi, a partir do estudo das primitivas de CTask e do seu diagrama de estados de tarefas, escolher um subconjunto de primitivas, incorporá-lo à especificação do núcleo e substituir sua implementação por uma síncrona equivalente. As primitivas escolhidas foram *reativas*, segundo a definição introduzida no capítulo anterior. Como os serviços do CTask são oferecidos através de chamadas a procedimentos e não por *traps* ou interrupções, decidiu-se rodar o autômato (e a máquina de execução) como mais uma tarefa no sistema (da mesma maneira que os servidores do núcleo), se comunicando com o resto através dos mecanismos suportados pelo CTask (especificamente, mensagens).

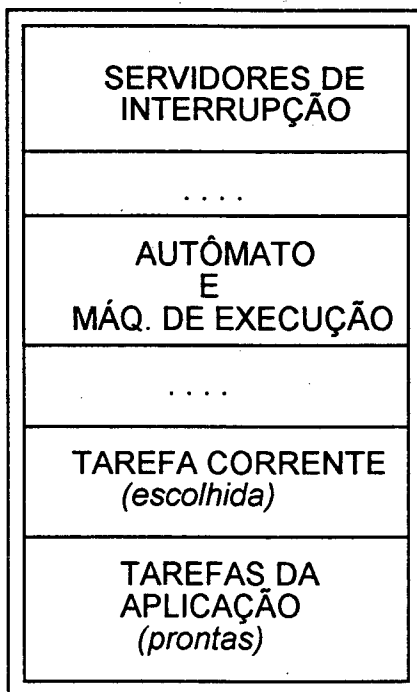


Figura 5.1: Mapa de prioridades dentro do Suporte Binário

A exemplo de [André93a], nosso autômato roda num nível de prioridade maior que qualquer outra tarefa da aplicação (ainda do nível *escolhida*) é menor que os outros tratadores de interrupção (Figura 5.1). O primeiro critério garante que vai executar e sua alta prioridade é justificada pelo autômato pertencer ao núcleo; já rodar sob os tratadores de interrupção implica numa melhor resposta do sistema aos acontecimentos do ambiente, que continuam sendo reconhecidos mesmo durante a execução de código reativo.

O autômato interage com o ambiente apenas através da máquina de execução, e em resposta a requisições de primitivas reativas e interrupções de hardware (somente do relógio nesta implementação). A máquina de execução foi implementada numa estrutura produtor/consumidor,

bloqueando-se à espera da chegada de eventos. Como consequência, o autômato ativa-se para cada requisição (os eventos são *serializados*), embora seja possível combinar várias e processá-las numa única ativação.

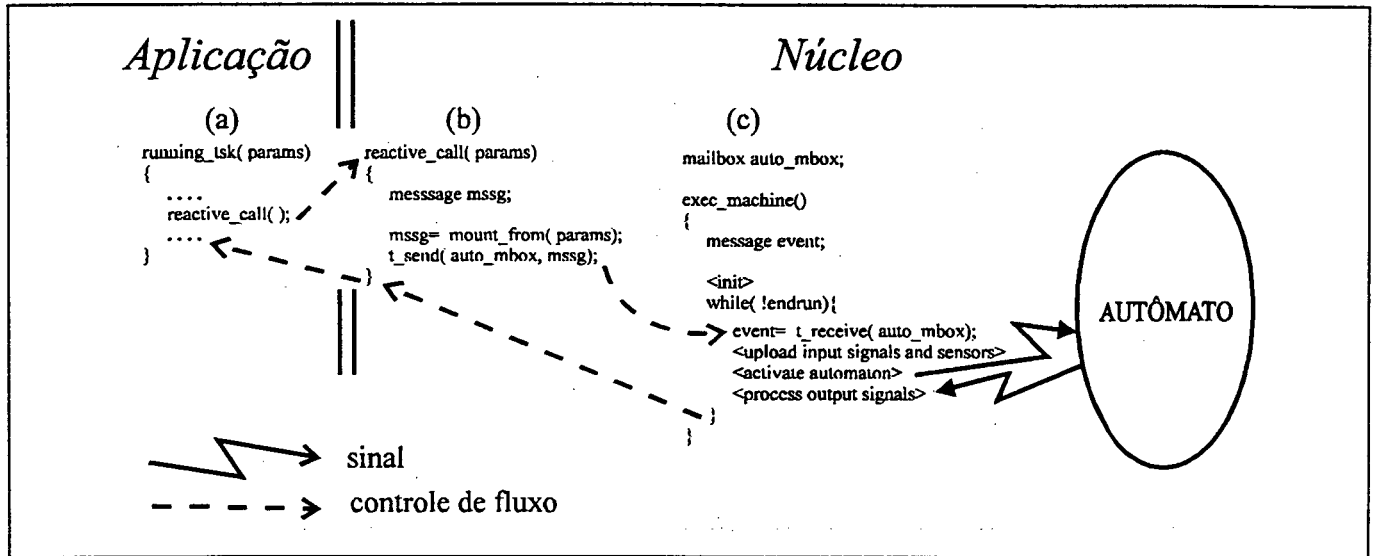


Figura 5.2: Funcionamento do núcleo frente a uma requisição reativa

A Figura 5.2 mostra como é processada uma requisição reativa genérica, após a qual não há troca de contexto. Quando a tarefa corrente invoca uma primitiva reativa (Figura 5.2-a), a função da Interface com a Aplicação associada monta uma mensagem (Figura 5.2-b) com os parâmetros da chamada e a envia à máquina de execução, que estava bloqueada aguardando, e desta forma é liberada para executar. A máquina de execução ativa os sinais e sensores correspondentes e chama o autômato (Figura 5.2-c), que evolui e muda de estado, eventualmente emitindo sinais de saída. Ao recuperar o controle, a máquina de execução traduz estes sinais nas ações correspondentes e bloqueia-se novamente à espera do próximo evento, retornando o controle ao CTask. O CTask invoca o algoritmo de escalonamento e depois seu despachante coloca a rodar a tarefa pronta de máxima prioridade (que continua a ser a tarefa chamadora, pois o autômato não ordenou troca de contexto). No caso do tratamento da interrupção do relógio, a diferença é que o tratador envia a mensagem correspondente diretamente à máquina de execução.

As funções e procedimentos chamados pelo núcleo não precisam sincronizar seu acesso a dados compartilhados. Quando se tem uma sequência de emissões de sinais de saída e/ou de chamadas a procedimentos ou funções, a metáfora síncrona especifica que são executadas simultaneamente e ainda de forma "instantânea"; esta execução, porém, concorre pela CPU com o resto dos componentes de controle paralelos do programa Esterel. Na prática, a execução é sequencial pela natureza do código gerado: as execuções das chamadas não se sobrepõem e portanto não é preciso sincronizar o acesso a dados comuns.

O núcleo final viria então na forma de uma biblioteca C contendo o autômato, a máquina de

execução e o suporte de baixo nível CTask. Esta biblioteca seria ligada à aplicação para formar um único programa executável na arquitetura alvo.

#### 5.1.4. VALIDAÇÃO DA HIPÓTESE DE SINCRONIA

Na nossa aplicação (um núcleo de tempo real), o sincronismo da abordagem refere-se ao existente entre a entidade reativa do núcleo e os estímulos externos, e não entre as entidades concorrentes (tarefas da aplicação) tentando acessar recursos compartilhados comuns. Por isso, elas devem continuar a sincronizar seu acesso aos recursos através das primitivas usuais (como semáforos) que o CTask oferece.

Anteriormente foi dito que a hipótese síncrona devia ser validada para cada implementação específica. Isto implicava em demonstrar que o autômato conseguia reagir totalmente a um evento antes da chegada do próximo, para um determinado ambiente<sup>2</sup>. Como a tarefa que fez a chamada não pode recuperar o processador enquanto a reação corrente não terminar (Figura 5.2), e as outras não conseguem interrompê-la por terem prioridade menor que a máquina de execução, é impossível a ativação de uma outra requisição reativa enquanto uma reação estiver em andamento, pelo menos por causa das tarefas. Se acontecer uma interrupção de relógio no meio de uma reação, é reconhecida mas não executada até o final da reação corrente (eventos são serializados). O período relativamente grande da interrupção (18 mSec) garante que o próximo tick chegará bem depois do término da reação da anterior.

Como o código do autômato não é reentrante, deve-se garantir a ausência de sobreposição de reações a eventos distintos. Nesta implementação, a serialização imposta pela estrutura produtor/consumidor adotada, faz com que as mensagens de ativação sejam reconhecidas num único ponto do código (Figura 5.2-a), e portanto impede que o autômato seja reativado durante uma reação; isto é válido tanto no caso das tarefas quanto das interrupções. Conseqüentemente, não é preciso desativar as interrupções durante a execução do autômato, que será sempre atômica.

Assume-se também que o código da entidade reativa, como as outras primitivas, executa em tempo nulo; caso contrário, os testes de escalonamento deveriam levar em conta no seu cálculo o tempo de execução próprio e o de cada uma das primitivas invocadas pela tarefa analisada, complicando desnecessariamente a implementação do protótipo.

#### 5.1.5. FASES DO PROJETO

Para o desenvolvimento do código em Esterel foi seguido o ciclo codificação-compilação-

---

<sup>2</sup> implícito no conceito de "ambiente" está a definição de uma taxa máxima de ocorrências de eventos.

simulação típico para esta classe de sistemas usando as ferramentas Esterel. Já no caso do desenvolvimento dos algoritmos de escalonamento e sua integração ao suporte binário, os algoritmos escolhidos (taxa monotônica, earliest deadline, menor folga e máxima urgência) foram codificados em C segundo a teoria. Note-se que todos eles compartilham o mesmo modelo temporal de tarefas: independentes e periódicas. O resultado foi integrado a um *simulador de escalonamento*, ferramenta desenvolvida especialmente, que toma a descrição temporal de um conjunto de tarefas *virtuais* (lidas de um arquivo de configuração), e a partir da escolha de uma política de escalonamento, gera os "timelines" correspondentes à sua execução. O simulador trabalha sobre o comportamento temporal do núcleo sendo escalonado mas não sobre seu processamento real: por isso as tarefas virtuais não têm código associado.

a) Vetor de Teste típico

```
CONFIGURATION FILE: testvct2.tst
Task Set: Same as Stewart's, but without task 'D'
Number of tasks in the set: 3

Task Set Description:
-----
Name           Criticality  Period  ExecTime  Task Load
Task A         high         6       2         33.3%
Task B         high        10       4         40.0%
Task C         high        12       3         25.0%
```

b) Resultados do seu escalonamento segundo taxa monotônica

```
Selected Scheduling Algorithm: Rate Monotonic (RM),
which has a schedulability bound of 78.0% for 3 tasks.
Critical set is composed of
    Task A,
    Task B,
which accounts for a critical load of 73.3%, over a total system load of 98.3%
WARNING: the whole task set MAY NOT be schedulable under RM

At 12: task c ("Task C"), instance 1, Deadline Failure
At 24: task c ("Task C"), instance 2, Deadline Failure

TIMELINE: [Rate Monotonic (RM)]

a b  a c b a b c a b  a c  Za
0123456789012345678901234567890
0      1      2      3

13 context switches
Cross-reference Names:
a      Task A
b      Task B
c      Task C
. . . .
```

Figura 5.3: Vetor de Teste típico e resultado do seu escalonamento por taxa monotônica



A correção da implementação dos algoritmos de teste foi conferida pelo contraste dos resultados do escalonamento de um conjunto de cenários com os resultados esperados pela teoria (Figura 5.3). Estes *vetores de teste* (Figura 5.3-a) foram escolhidos de maneira a exercitar aspectos críticos dos algoritmos implementados. Os resultados de suas simulações foi salvo num *arquivo de resultados padrões*, após conferir que eram consistentes com os resultados previstos na teoria. A título de exemplo, a Figura 5.3-b mostra os resultados do escalonamento do vetor de teste anterior pelo algoritmo taxa monotônica (o Anexo 2 apresenta o resultado do seu escalonamento pelos outros algoritmos disponíveis).

De posse do produto da compilação Esterel do núcleo, foi codificada sua máquina de execução. A integração dos algoritmos de escalonamento ao elemento reativo resultante deu lugar a um novo simulador, desta vez com código síncrono no seu interior. Rodaram-se novamente todos os vetores de teste, e conferiu-se que o resultado de suas simulações gerasse exatamente as mesmas sequências de teste que o programa anterior. O sucesso da comparação garantiu a integração certa entre as políticas e o autômato.

Para a conferência, utilizou-se uma abordagem automática, que através de *scripts* e o utilitário *diff* de UNIX escalonava, de forma sistemática, todos os vetores de teste por todos os algoritmos disponíveis, e compararam-se os resultados com o arquivo de resultados padrões. Com esta abordagem tem-se a vantagem que novos algoritmos ou implementações mais eficientes dos atuais podem ser testados com mínimo esforço, apenas conferindo se o resultado foge do esperado. Também serve para determinar qual algoritmo adapta-se melhor a um conjunto de tarefas determinado.

A codificação dos algoritmos e o desenvolvimento da parte reativa do núcleo foram feitos fundamentalmente num sistema UNIX-SUN. Já na arquitetura alvo (PC-DOS), o CTask foi modificado para aceitar os algoritmos de escalonamento: as tarefas tinham agora código verdadeiro associado e eram executadas, não mais simuladas. CTask foi modificado para que cada chamada ao despachante seja interceptada e desviada ao algoritmo de escalonamento escolhido. Um identificador da nova tarefa corrente era armazenado sequencialmente num buffer após cada ponto de escalonamento; no final da execução, o buffer era interpretado e gerava-se um arquivo no formato do arquivo de resultados padrão. Rodaram-se novamente os vetores de teste, conferindo o funcionamento correto do conjunto através do procedimento sistemático anterior. Neste ponto o CTask estava suportando algoritmos de tempo real.

Finalmente, o núcleo síncrono foi "implantado" no CTask ao substituir seu processamento das chamadas reativas selecionadas pelo autômato. Os vetores de teste foram rodados novamente, fornecendo o resultado do escalonamento correto. Salienta-se que o código reativo é, na visão do CTask, mais uma tarefa de aplicação. Isto implica em apenas uma modificação mínima no seu código para sua implementação.

## 5.2. METODOLOGIA DE GERAÇÃO DE NÚCLEOS ESPECÍFICOS

Propomos neste item uma metodologia para a geração de NTRs específicos, particularizáveis pelo próprio usuário, baseada na escolha de seus comportamentos internos e/ou do algoritmo de escalonamento a partir das características temporais da aplicação. A idéia básica é combinar o item selecionado com as funções do núcleo que se mantêm não modificadas frente a qualquer variação do escalonador ou de comportamento interno (*autômato base*).

### 5.2.1. PARTICULARIZAÇÃO POR SELEÇÃO DE COMPORTAMENTOS INTERNOS

A motivação principal para a particularização do núcleo por comportamentos internos é desempenho: implementando apenas as funcionalidades realmente usadas é possível obter núcleos mais rápidos e "leves" (em termos de tamanho de código), o que é interessante para aplicações de alta performance ou embutidas, tudo isto sem abrir mão da validação formal sobre o autômato resultante. A partir do estudo das tarefas da aplicação seria possível definir a presença ou não de uma funcionalidade no núcleo, e em caso de existir, qual seu comportamento específico. Exemplos de módulos selecionáveis seriam:

- \* presença de tratadores de exceção temporal e seus comportamentos: abortar a tarefa, reiniciá-la, envio de mensagens às camadas superiores
- \* presença de serviços temporais (por exemplo, *delay*)
- \* modelo de sincronização e comunicação: desnecessários para tarefas independentes, poderia-se escolher entre semáforos, mensagens, etc. Ainda a comunicação poderia contemplar a existência de uma rede e suas abstrações (p.ex., semáforos remotos)
- \* escalonamento preemptivo ou não: no segundo caso, ao invés de aguardar o término de um lapso de tempo, o núcleo simplesmente esperaria que a própria tarefa entregasse o processador.
- \* tarefas das filas de tarefas do núcleo poderiam ser retiradas segundo uma política de escalonamento específica para cada fila ou situação. Por exemplo, na chegada de uma mensagem, uma tarefa da lista de bloqueadas por comunicação poderia ser escolhida segundo sua prioridade, ou FIFO.

Ainda para o módulo ou funcionalidade, poderia ser escolhida uma implementação que

privilegiasse a velocidade de execução por sobre a memória ocupada, ou vice-versa. As vantagens desta abordagem viriam do fato de ter um núcleo ajustado à aplicação em mãos. Note-se que não estamos falando de *configurar* o núcleo (entendido como a ativação/desativação de elementos internos) mas de criar núcleos totalmente novos para cada aplicação, a partir da composição síncrona de uma estrutura básica com o(s) comportamento(s) desejado(s). Estes, junto ao núcleo base, estariam sob a forma de código intermediário Esterel disponíveis para a composição síncrona final.

### 5.2.2. PARTICULARIZAÇÃO POR SELEÇÃO DA POLÍTICA DE ESCALONAMENTO

Para a seleção do algoritmo de escalonamento propõe-se a seguinte metodologia, cujos passos principais são ilustrados na Figura 5.4:

- 1- A partir da análise cuidadosa das tarefas da aplicação, define-se o modelo de tarefas com que vai-se trabalhar, e escolhe-se a política de escalonamento mais apropriada segundo os critérios apresentados no Capítulo 2.
- 2- O algoritmo escolhido é retirado da *biblioteca de algoritmos* (repositório que contém um conjunto de algoritmos de escalonamento pré-codificados em Esterel, na forma de código intermediário) e composto sincronamente com o autômato base, gerando após fases sucessivas um arquivo C contendo o código do *autômato final*. O autômato base contém a lógica das primitivas reativas do núcleo que não são afetadas pela política de escalonamento. A composição síncrona é o resultado da ligação, por parte do compilador, do autômato base à política escolhida: sua composição satisfatória garante o determinismo do resultado, pelas técnicas usadas pelo compilador para rejeitar programas não deterministas.
- 3- De posse deste arquivo e dos fontes C descrevendo a subcamada de dados e a máquina de execução, os três são combinados num código objeto único.
- 4- O código resultante é simulado e suas propriedades conferidas usando as ferramentas do ambiente Esterel
- 5- O resultado validado é levado ao sistema alvo e integrado com o *suporte binário*, que implementa as rotinas de mais baixo nível e as funções dependentes da arquitetura alvo. Esta integração é feita pela recompilação do código do autômato e sua posterior ligação ao suporte binário numa biblioteca.
- 6- Finalmente, a biblioteca do núcleo é ligada aos códigos objeto da aplicação específica formando um único código executável para o sistema alvo (PC).

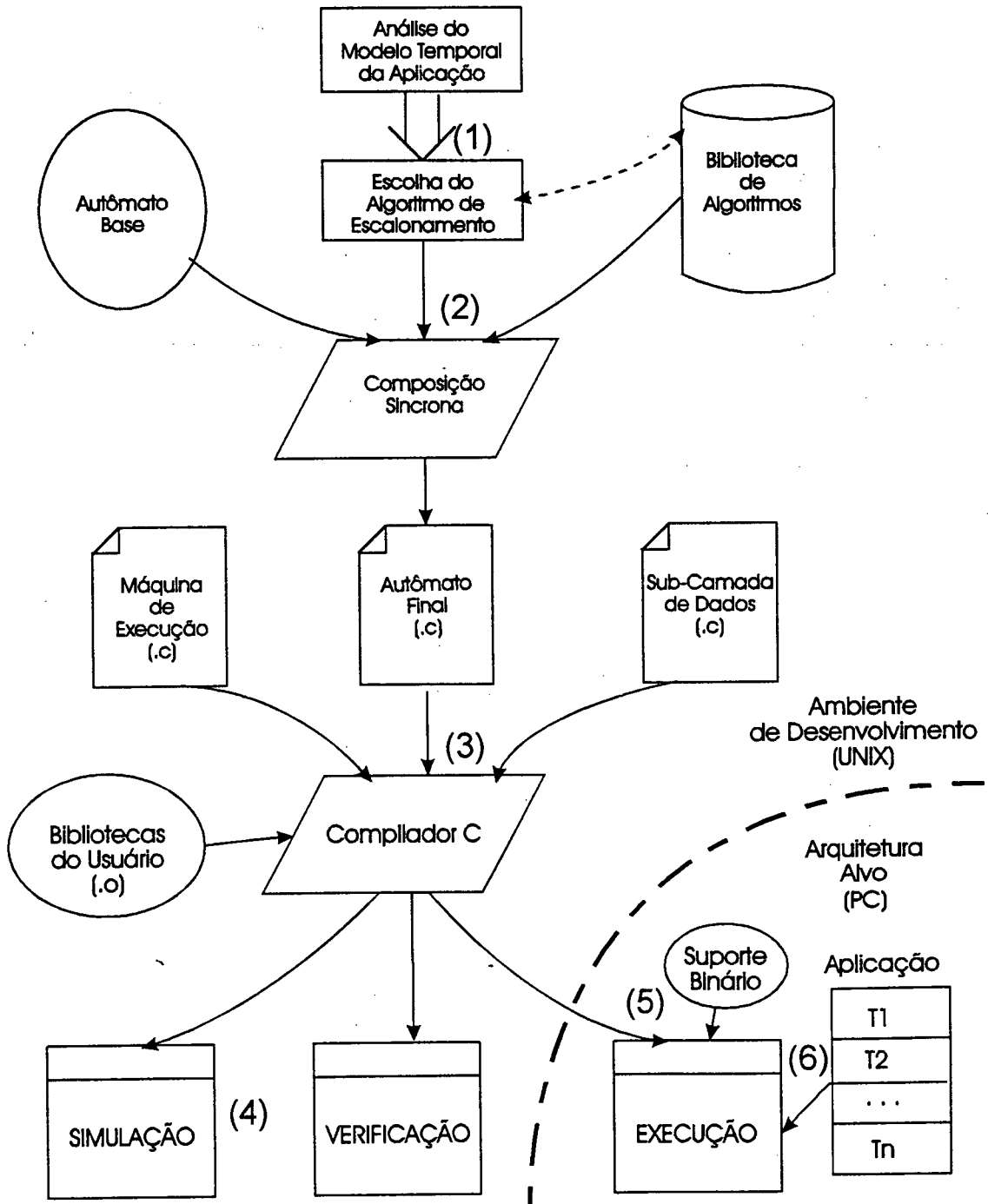


Figura 5.4: Metodologia de particularização de Núcleos

O núcleo é visto pela aplicação na forma de uma biblioteca C, contendo o autômato, seu interfaceamento e suporte de baixo nível. A aplicação do usuário, escrita na forma de um conjunto de tarefas acessando o NTR, é compilada e ligada à biblioteca do NTR, formando um código único que executará no ambiente alvo real.

A Biblioteca de Algoritmos pode ser acrescida com outras políticas, até eventualmente as criadas pelo próprio usuário. O procedimento difere se a política é estática ou dinâmica. No caso das políticas estáticas, baseia-se em descrever na linguagem Esterel as seqüências de ativações das tarefas segundo uma agenda pré-definida. O algoritmo estático gera esta agenda em *pré-runtime*, que será executada pelo sistema (repetidamente, no caso periódico).

Exemplos deste tipo de políticas são a taxa monotônica e o executivo cíclico. A agenda gerada é descrita pelas tuplas (tarefa, duração) que um eventual gerador de programas Esterel utilizaria para a definição da tarefa a rodar a cada instante de escalonamento. Por exemplo, considerando a agenda:

(a,t1), (b,t2), (c,t3), (d,t4);

se o processamento começar no instante  $t$ , o código 'a' executaria de  $t$  até  $(t+t_1)$ , 'b' entre  $(t+t_1)$  e  $(t+t_2)$ , e assim por diante, gerando um código semelhante ao da Figura 5.5. Este módulo é ativado pelo sinal SCHEDULE e retorna o sinal RESPONSE, indicando a tarefa selecionada. O código da figura é para uma agenda periódica.

```
1  loop
2      await SCHEDULE
3      time= ?REQUEST % SCHEDULE_LENGTH;
4      if time < t1 then
5          emit RESPONSE(a)
6      else if time < t2 then
7          emit RESPONSE(b)
8      else if time < t3 then
9          emit RESPONSE(c)
10     else
11         emit RESPONSE(d)
12     endif
13 end
```

Figura 5.5: Código Esterel expressando a agenda exemplo do texto

Ao contrário das estáticas, as políticas dinâmicas precisam ser codificadas em Esterel. Estas codificações são dificultadas pelas limitações da linguagem no manuseio de dados, o que implica na necessidade de uso mais frequente de procedimentos e funções codificadas a nível da subcamada de dados. Este problema é tratado com a programação em Esterel das estruturas de controle internas às políticas seguido pela sua complementação com os procedimentos e funções da subcamada de dados.

### 5.3. CONSIDERAÇÕES GERAIS

Esterel permite escrever especificações orientadas a eventos, de alto nível. Não sendo uma linguagem de propósitos gerais, deve ser usada apenas para a programação de entidades reativas. Ainda assim, é conveniente a programação das estruturas de controle simples com linguagens procedurais fora do autômato, ao invés de fazê-lo em Esterel: o projetista deve ser cuidadoso em colocar as funcionalidades onde são implementadas de maneira mais natural [Berry91]. A verificação que permite o formalismo de Esterel aumenta substancialmente a robustez e confiabilidade do código gerado, ao melhorar sua qualidade em relação ao número de erros lógicos presentes. A abordagem de validação é semelhante à do LOTOS: uma lógica para raciocinar com sequências de eventos, mas não com os valores temporais específicos deles.

Na implementação dos escalonadores, assim como o resto das estruturas do núcleo, percebeu-se que existe um compromisso entre a generalidade e a eficiência que pode ser conseguida. Isto é, a programação de interfaces "dedicadas" a determinado tipo de algoritmos pode explorar suas características particulares para conseguir ganhos de eficiência que não são possíveis numa interface genérica e portanto mais reusável, suportando um modelo temporal mais geral de tarefas.

A particularização de núcleos por seleção de componentes internos pode não ser de grande utilidade no caso de aplicações grandes, com modelos temporais complexos e que usam praticamente todas as funcionalidades disponíveis no núcleo para sua execução. Embora o usuário possa escolher para sua aplicação um algoritmo entre os disponíveis, o eventual acréscimo de outros não é possível sem recompilar o conjunto. Uma alternativa seria implementar os códigos dos algoritmos como bibliotecas de ligado dinâmico (*dynamic link libraries*), de maneira a poder variar o conjunto de políticas dinamicamente. Este conceito também serviria para outros componentes do núcleo que são configuráveis em tempo de compilação.

Ao desprezar o tempo de execução das primitivas de CTask e do autômato, perde-se parte das vantagens da abordagem síncrona neste caso (a predição determinista do tempo máximo de execução de cada primitiva reativa a partir do seu código sequencial). Além disso, não se dispõe das ferramentas necessárias para a geração e análise estática da árvore de execução dos códigos envolvidos, ficando como única alternativa a medição empírica dos tempos de execução.

Nossa abordagem para a implantação do núcleo síncrono em CTask acabou sendo mais simples que as apresentadas em [André93a], pela disponibilidade de mensagens a nível do suporte binário (*t\_send()* e *t\_receive()*). O código do autômato gerado é bastante eficiente, embora sofra de

explosão de estados<sup>3</sup> em determinadas circunstâncias. Isto pode ser evitado ou minimizado pelo uso de ferramentas e técnicas apropriadas [Berry93]. [André93b] e [André93c] propõem uma implementação *parcialmente assíncrona*, onde os módulos da aplicação são programados em autômatos individuais, que executam numa ordem fixa com cada ativação.

Afirmamos que nosso núcleo não pode ser comparado de forma justa com outros. Primeiro, porque é um protótipo, sem preocupação nenhuma com as questões de eficiência que costumam ser o parâmetro de comparação; por outro lado, as técnicas de *benchmarking* ainda são imaturas, não existindo um consenso para o caso específico de Tempo Real. A única comparação válida seria com o CTask, núcleo que serviu de base para implementar o nosso: aí deveríamos conferir que não foi perdido muito desempenho pela introdução do autômato, pois apenas foi acrescentado código sequencial a algumas das suas primitivas. Essa diferença de performance deve ser ainda menor se nossa estrutura for otimizada, eliminando, por exemplo, a camada de Interface com a Aplicação.

## 5.4. CONCLUSÃO

A partir da especificação e projeto do núcleo segundo a abordagem síncrona apresentada no capítulo anterior, este capítulo explorou os aspectos relacionados com a nossa implementação da arquitetura proposta. Após a descrição do ambiente Esterel utilizado e do papel de cada ferramenta no desenvolvimento de uma aplicação síncrona genérica, expôs-se o suporte binário sobre o qual o núcleo foi montado e as modificações que este suporte sofreu para acomodar o elemento reativo e as políticas de escalonamento para tempo real. Fez-se uma relação das fases do desenvolvimento, salientando o grau (elevado) de automatização na validação dos sucessivos estágios do projeto, e finalmente validou-se o resultado como suportando corretamente a hipótese síncrona.

Na segunda parte do capítulo descreveu-se uma metodologia de geração de núcleos para aplicações específicas, salientando as possibilidades de configuração a nível de comportamento internos e algoritmos de escalonamento, e sua escolha segundo o modelo temporal da aplicação sendo desenvolvida.

---

<sup>3</sup> A partir da versão 4.41 de Esterel não mais aparece este problema, gerando diretamente código linear.

# CAPITULO 6

## RESULTADOS E CONCLUSÕES FINAIS

*Aqui nada é urgente,  
porque urgente é tudo aquilo que não foi providenciado  
a tempo*

Do estudo das políticas de escalonamento pode-se concluir que existe uma grande variedade de algoritmos, se diferenciando principalmente pelo modelo temporal de tarefas que abrangem e pela sua complexidade computacional. Porém, foi provado que não existem algoritmos que possam escalonar um conjunto de tarefas de modelos temporais genêricos com um custo de execução razoável; portanto, para cada aplicação em particular deve ser escolhido o algoritmo mais apropriado após a análise do modelo temporal de suas tarefas.

Do mesmo modo, existe uma variedade de abordagens práticas ao problema de tempo real (e de suportes e sistemas operacionais que as implementam), mas nenhuma consegue dar conta de todos os tipos de aplicações. Portanto, a escolha da estrutura de suporte adequada deve ser feita analisando as características específicas da aplicação em mãos.

Dos resultados alcançados, dois consideramos significativos. O primeiro é a definição de uma arquitetura reativa implementada num modelo síncrono de processamento para núcleos de tempo real. A aplicação da abordagem síncrona neste caso específico trouxe várias vantagens: além das estruturais



(maior nível conceitual do código, formalismo da abordagem), o determinismo do resultado permitiu a reprodução de cenários específicos com precisão, o que melhorou grandemente sua capacidade de validação por teste. O código gerado para as transições do autômato, ao ser sequencial, pode ser analisado por meios computacionais para determinar com precisão o tempo máximo de execução. A separação mais nítida do elemento de controle do de dados é desejável porque fornece uma melhor estruturação e simplificações importantes na análise de correção do núcleo.

Uma das dificuldades de programar em Esterel foi a mudança na concepção do problema, que agora deve ser tratado em forma síncrona. Isto faz com que o programador deva passar um período de transição até dominar o novo paradigma e atingir uma produtividade razoável na linguagem. Por outro lado, a carência de suporte para sinais multivalorizadas faz com que determinadas situações devam ser resolvidas de forma pouco elegante (isto é, decompondo um sinal multivalorizado em vários, cada um acarretando um componente de valor). Também fica difícil escolher entre utilizar sinais de saída e chamadas a procedimentos, quando a principal diferença conceitual (o "poder sincronizante" das primeiras) não é preciso.

O suporte à modularidade provisto por Esterel e outras linguagens síncronas facilita a extensão e modificação do programa fonte, embora os critérios arbitrários empregados para a decomposição modular do problema fazem que nem sempre as soluções atingidas sejam ótimas. Este defeito tende a diminuir com a abordagem a objetos, porém as linguagens síncronas atuais não oferecem suporte nenhum nesse sentido.

O segundo resultado significativo é a definição de uma metodologia para a geração de NTRs específicos, no contexto da arquitetura reativa proposta. Esta metodologia faz uso da modularidade da linguagem Esterel para a configuração do escalonador do NTR em questão, e a reconfigurabilidade é dada no escalonamento do sistema e na escolha de comportamentos internos adequados à aplicação em mãos.

As versões de núcleos implementadas em nosso laboratório (com taxa monotônica, *earliest deadline* e executivo cíclico) segundo o modelo de arquitetura reativo proposto e usando a metodologia definida, funcionaram a contento. Os *timelines* produzidos se mostraram coerentes com a teoria e atenderam as necessidades temporais da aplicação.

A interface do núcleo proposto poderá ser gradativamente ampliada no futuro para suportar outros serviços e comportamentos, por exemplo: serviços de comunicação mais sofisticados (envio síncrono e garantido de mensagens). Como uma demonstração da qualidade da arquitetura (a atingir), sua estrutura não deveria mudar muito com o acréscimo dessa novas funcionalidades.

# REFERÊNCIAS BIBLIOGRÁFICAS

## CAPÍTULO 1

- [Audsley90] N. Audsley and A. Burns. *Real-Time System Scheduling*. University of York, Predictably Dependable Computing Systems (PDCS) -ESPRIT, May 1990.
- [Benveniste91] A. Benveniste and G. Berry. *The Synchronous Approach to Reactive and Real-Time Systems*. Proceedings of the IEEE: *Special Issue on Real-Time Systems*, Vol. 79, N° 9, September 1991.
- [Berry88] G. Berry and G. Gonthier. *The ESTEREL synchronous programming language: Design, Semantic, Implementation*. Rapport de Recherche 842, INRIA, 1988.
- [Boussinot91] F. Boussinot and R. de Simone. *The ESTEREL Language*. Rapports de Recherche 1487, INRIA, July 1993. Also in Proceedings of the IEEE: *Special Issue on Real-Time Systems*, Vol. 79, N° 9, September 1991.
- [Cheng88] S.-C. Cheng and J. Stankovic. *Scheduling Algorithms for Hard Real-Time Systems - Brief Survey*. Hard Real-Time Systems: Tutorial, IEEE, July 1988.
- [Jensen85] E. D. Jensen, C. D. Locke and H. Tokuda. *A Time-driven Scheduling Model for Real-Time Operating Systems*. Proceedings IEEE Real-Time Systems Symposium, December 1985.
- [Kopetz89] H. Kopetz et al. *Distributed Fault-Tolerant Real-Time Systems: The MARS Approach*. IEEE Micro, February 1989.
- [Liu73] C. L. Liu and J. Leyland. *Scheduling Algorithms for Multiprogramming in Hard-Real Time Environment*. JACM 20(1), 1973.
- [Mok84] A. Mok. *The Design of RealTime Programming Systems based on Process Models*. Proceedings of the IEEE Real-Time Systems Simposium, 1984.
- [Sha88] L. Sha, R. Rajkumar and J. Lehoczky. *Priority Inheritance Protocol: An Approach to Real-Time Synchronization*. Dept. of Computer Science, Carnegie-Mellon University, May 1988.
- [Stank88] J. Stankovic. *Misconceptions about Real-Time Computing: A Serious Problem for Next Generation Systems*. IEEE Computer, October 1988.
- [Stank91] J. Stankovic. *The Spring Kernel: A New Paradigm for Hard-Deadline Real-Time Systems*. IEEE Software, Vol. 8, N° 3, May 1991.
- [Tokuda89] H. Tokuda and C. Mercer. *ARTS: A Distributed Real-Time Kernel*. ACM Operating Systems Review - Special Issue, 1989.

## CAPÍTULO 2

- [Audsley90] N. Audsley and A. Burns. *Real-Time System Scheduling*. University of York, Predictably Dependable Computing Systems (PDCS) -ESPRIT, May 1990.
- [Baker89] T. Baker and A. Shaw. *The Cyclic Executive Model and Ada*. *Real-Time Systems*, 1, 7-25 (1989).
- [Baker91] T. Baker. *Stack-Based Scheduling of Realtime Systems*. *Real-Time Systems*, 3, 67-99 (1991).
- [Berry92] S. Berryman and I. Sommerville. *Modeling and Evaluating the Feasibility of Timing Constraints Under Different Real-Time Scheduling Algorithms*. *Real-Time Systems Journal*, 4:287-306, 1992.
- [Blake91] B. Blake, and K. Schwan. *Experimental Evaluation of a Real-Time Scheduler for a Multiprocessor System*. *IEEE Transactions on Software Engineering*, Vol.17, No.1, January 1991.
- [Chen90] M.-I. Chen and K.-J. Lin. *Dynamic Priority Ceilings: A Concurrency Control Protocol for Real-Time Systems*. *Real-Time Systems*, 2, 325-346 (1990).
- [Cheng88] S.-C. Cheng and J. Stankovic. *Scheduling Algorithms for Hard Real-Time Systems - Brief Survey*. *Hard Real-Time Systems: Tutorial*, IEEE, July 1988.
- [Chetto89] H. Chetto and M. Chetto. *Some Results of the Earliest Deadline Scheduling Algorithm*. *IEEE Transactions on Software Engineering*, Vol.15, No.10, October 1989.
- [Dert89] M. Dertouzos, and A. Mok. *Multiprocessor On-Line Scheduling of Hard-Real-Time Tasks*. *IEEE Transactions on Software Engineering*, Vol.15, No.12, December 1989.
- [Ferra94] A. Ferrari. *Algoritmos de escalonamento para Tempo Real*. Nota interna LCMI, UFSC, 1994.
- [Henn89] R. Henn. *Feasible Processor Allocation in a Hard-Real-Time Environment*. *Real-Time Systems*, 1, 77-93 (1989).
- [Homa94] N. Homayoun and P. Ramanathan. *Dynamic Priority Scheduling of Aperiodic Tasks in Hard Real-Time Systems*. *Real-Time Systems*, 6, 207-232 (1994).
- [Klein93] M. Klein et al. *A Practitioner's Handbook for Real-Time Analysis: Guide to Rate-Monotonic Analysis for Real-Time Systems*. Kluwer Academic Publishers, Boston, July 1993.
- [Klein94] M. Klein, J. Lehoczky and R. Rajkumar. *Rate Monotonic Analysis for Real-Time Industrial Computing*. *IEEE Computer*, January 1994.
- [Kopetz92] H. Kopetz. *Scheduling*. An Advanced Course on Distributed Systems. Estoril, Portugal, 1992.
- [Lauber89] R. Lauber. *Forecasting Real-Time Behavior During Software Design using a CASE Environment*. *Real-Time Systems*, 1 (1989).
- [Lehoczky90] J. Lehoczky. *Fixed Priority Scheduling of Periodic Task Sets with Arbitrary Deadlines*. *Proceedings of the IEEE Real-Time Systems Symposium*, 1990.
- [Lein80] D. Leinbaugh. *Guaranteed Response Times in a Hard Real-Time System*. *IEEE Transactions on Software Engineering*, January 1980.

- [LiuC73] C. Liu and J. Layland. *Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment*. Journal of the ACM, Vol.20, No.1, January 1973.
- [LiuJ91] J. Liu et al. *Algorithms for Scheduling Imprecise Computations*. IEEE Computer, May 1991.
- [Locke92] D. Locke. *Software Architecture for Hard-Real-Time Applications: Cyclic Executives vs. Fixed Priority Executives*. Real-Time Systems, 4, 37-53 (1992).
- [Mok84] A. Mok. *The Design of RealTime Programming Systems based on Process Models*. Proceedings of the IEEE Real-Time Systems Symposium, 1984.
- [Obenza93] R. Obenza. *Rate Monotonic Analysis for Real-Time Systems*. IEEE Computer, March 1993.
- [Schwan92] K. Schwan and H. Zhou. *Dynamic Scheduling of Hard Real-Time Tasks and Real-Time Threads*. IEEE Transactions on Software Engineering, Vol.18, No.8, August 1992.
- [Sha90] L. Sha and J. Goodenough. *Real-Time Scheduling Theory and Ada*. IEEE Computer, April 1990.
- [Sha93] L. Sha et al. *A Systematic Approach to Designing Distributed Real-Time Systems*. IEEE Computer, September 1993.
- [Shi91] W. K. Shih, J. Liu and C. Liu. *Modified Rate Monotonic Algorithm for Scheduling Periodic Jobs with Deferred Deadlines*. Real-Time Systems Newsletter, Vol. 7, No 1/2, Winter/Spring 1991.
- [Shi92] W. K. Shih, J. Lin. *On-line Scheduling of Imprecise Computations to Minimize Error*. IEEE Proceedings of the Real-Time Systems Symposium, December 1992.
- [Sprunt89] B. Sprunt, L. Sha and J. Lehoczky. *Aperiodic Task Scheduling for a Hard-Real-Time System*. Real-Time Systems, 1, 27-60 (1989).
- [Stan90] J. Stankovic and K. Ramamritham. *Editorial - What is Predictability for Real-Time Systems?*. Real-Time Systems, 2, 247-253 (1990).
- [Stan91] J. Stankovic and K. Ramamritham. *The SPRING Kernel: A New Paradigm for Real-Time Systems*. IEEE Software, May91.
- [Stew91] D. Stewart and P. Koshla. *Real-Time Scheduling of Sensor-Based Control Systems*. Proceedings of Eighth IEEE Workshop on Real-Time Operating Systems and Software, pp. 144-150, May 1991.
- [Xu90] J. Xu and D. Parnas. *Scheduling Processes with Release Times, Deadlines, Precedence, and Exclusion Relations*. IEEE Transactions on Software Engineering, Vol. 16, No 3, March 1990.
- [Xu93] J. Xu and D. Parnas. *On Satisfying Timing Constraints in Hard Real-Time Systems*. IEEE Transactions on Software Engineering, Vol. 19, No 1, January 1993.

## CAPÍTULO 3

- [Armand89] F. Armand et al. *Multi-Threaded Processes in CHORUS/MiX*. Chorus Systèmes, CS/TR-89-37.3, May 1990.

- [Bihari92] T. Bihari and P. Gopinath. *Object-Oriented Real-Time Systems: Concepts and Examples*. IEEE Computer, December 1992.
- [Booch86] G. Booch. *Object-Oriented Development*. IEEE Transactions on Software Engineering, February 1986.
- [Booch91] G. Booch. *Object-Oriented Design with Applications*. Benjamin/Cummings, Redwood City, Calif 1991.
- [Bricker91] A. Bricker et al. *Architectural Issues in Microkernel-Based Operating Systems: the CHORUS Experience*. Computer Communications, Vol. 14, No. 6, July 1991.
- [Campb87] R. Campbell, G. Johnston and V. Russo. *CHOICES (Class Hierarchical Open Interface for Custom Embedded Systems)*. ACM Operating Systems Review, July 1987.
- [CDes90] *Real-Time OS Struggle with Multiple Tasks*. Computer Design, October 1, 1990.
- [Chorus88] *CHORUS Distributed Operating Systems - Technical Report*. Chorus Systèmes, CS/TR-88-7.6, November 1988.
- [Corvin90] W. Corwin, C. Locke and K. Gordon. *Overview of the IEEE POSIX P1003.4 Real-Time Extension to POSIX*. Real-Time Systems NewsLetter, Winter 1990.
- [Farrow93] R. Farrow. *Microkernels: The Soul of a New OS*. UnixWorld, November 1993.
- [Ishi92] Y. Ishikawa, H. Tokuda and C. Mercer. *An Object-Oriented Real-Time Programming Language*. IEEE Computer, October 1992.
- [Kopetz89] H. Kopetz et al. *Distributed Fault-Tolerant Real-Time Systems: The MARS Approach*. IEEE Micro, February 1989.
- [Kopetz91] H. Kopetz. *Event-Triggered versus Time-Triggered Real-Time Systems*. Technical University of Vienna, Vienna, Austria, 1991.
- [Kopetz92] H. Kopetz et al. *The Programmer's View of MARS*. Proceedings of the IEEE Real-Time Systems Symposium, December 1992.
- [Kopetz94] H. Kopetz and G. Grunsteidl. *TTP - A Protocol for Fault Tolerant Real-Time Systems*. IEEE Computer, January 1994.
- [Mercier92] C. Mercier and H. Tokuda. *The ARTS Real-Time Object Model*. Proceedings of the Real-Time Systems Symposium, December 1992.
- [Mole90] L. D. Molesky et al. *Implementing a Predictable Real-Time Multiprocessor Kernel - The Spring Kernel*. Real-Time Systems Newsletter, Vol. 6, Nº. 2, Spring 1990.
- [Nata92] S. Natarajan and W. Zhao. *Issues in Building Dynamic Real-Time Systems*. IEEE Software, September 1992.
- [Pospis92] G. Pospischil et al. *Developing Real-Time Tasks with Predictable Timing*. IEEE Software, September 1992.
- [Ramam90] K. Ramamritham, J. Stankovic and P. Shiah. *Efficient Scheduling Algorithms for Real-Time Multiprocessor Systems*. IEEE Transactions on Parallel and Distributed Systems, Vol. 1, No. 2, April 1990.

- [Schwan87] K. Schwan et al. *High-Performance Operating System Primitives for Robotics and Real-Time Control Systems*. ACM Transactions on Computer Systems, Vol. 5, No.3, August 1987.
- [Schwan90a] K. Schwan, A. Geith, and H. Zhou. *From CHAOS<sup>base</sup> to CHAOS<sup>arc</sup>: A Family of Real-Time Kernels*. Proceedings of the IEEE Real-Time Systems Symposium, December 1990.
- [Schwan90b] K. Schwan, and A. Geith. *CHAOS<sup>arc</sup>: A Kernel for Predictable Programs in Dynamic Real-Time Systems*. Real-Time Systems Newsletter, Vol. 6, No. 2, Spring 1990.
- [Schwan91] K. Schwan, H. Zhou, and A. Geith. *Real-Time Threads*. ACM Operating Systems Review, Vol 25, October 1991.
- [Schwan92] K. Schwan and H. Zhou. *Dynamic Scheduling of Hard Real-Time Tasks and Real-Time Threads*. IEEE Transactions on Software Engineering, Vol.18, No.8, August 1992.
- [Singh90a] I. Singh. *Real-Time Extensions Need Standardization*. UnixWorld, March 1990.
- [Singh90b] I. Singh and M. Bunnell. *LynxOS: UNIX Rewritten for Real-Time*. Real-Time Systems Newsletter, Vol. 6, No. 2, Spring 1990.
- [Stan87] J. A. Stankovic, and K. Ramamritham. *The Design of the Spring Kernel*. Proceedings of the Real-Time Systems Symposium, 146-57, December 1987.
- [Stan91] J. A. Stankovic, and K. Ramamritham. *The Spring Kernel: A New Paradigm for Real-Time Systems*. IEEE Software, May 1991.
- [Toku89] H. Tokuda and C. Mercer. *ARTS: A Distributed Real-Time Kernel*. ACM Operating Systems Review 23(3), July 1989.

## CAPÍTULO 4

- [André93a] C. André and M.-A. Péraldi. *Effective Implementation of ESTEREL programs*. 5<sup>th</sup> Euromicro Workshop on Real-Time Systems, Oulu (Finland), June 1993.
- [André93b] C. André and M.-A. Péraldi. *Programmation Synchrone d'Applications Temps-Réel sur Microcontrôleurs*. Rapport de Recherche N<sup>o</sup>92-64. Salon des Solutions Informatiques Temps-Réel, Paris, Janvier 1993.
- [Andrews83] G. Andrews. *Concepts and Notations for Concurrent Programming*. ACM Computing Surveys, N<sup>o</sup> 15-1, March 1983.
- [Benveniste91] A. Benveniste and G. Berry. *The Synchronous Approach to Reactive and Real-Time Systems*. Proceedings of the IEEE: *Special Issue on Real-Time Systems*, Vol. 79, N<sup>o</sup> 9, September 1991.
- [Berry88] G. Berry and G. Gonthier. *The ESTEREL synchronous programming language: Design, Semantic, Implementation*. Rapport de Recherche 842, INRIA, 1988.
- [Berry89] G. Berry. *Real-Time Programming: Special Purpose or General Purpose Languages*. Proc. IFIP 89 World Computer Congress, San Francisco, 1989.
- [Berry93] G. Berry. *Communicating Reactive Processes*. POPL'93, Charleston, South Carolina, USA.

- [Boussinot91] F. Boussinot and R. de Simone. *The ESTEREL Language*.  
Rapports de Recherche 1487, INRIA, July 1993. Also in Proceedings of the IEEE: *Special Issue on Real-Time Systems*, Vol. 79, N<sup>o</sup> 9, September 1991.
- [CISI88Intrfc] Cisi Engenharia. *Esterel V3 - C Interface Manual*.  
CISI ENGENHARIA, INRIA, Ecole Des Mines, Armines, 1988.
- [Clarke91] E. Clarke et al. *A Language for Compositional Specification and Verification of Finite State Hardware Controllers*.  
Proceedings of the IEEE: *Special Issue on Real-Time Systems*, Vol. 79, N<sup>o</sup> 9, September 1991.
- [Halbwachs91] N. Halbwachs. *The Synchronous Data Flow Programming Language LUSTRE*.  
Proceedings of the IEEE: *Special Issue on Real-Time Systems*, Vol. 79, N<sup>o</sup> 9, September 1991.
- [LeGuernic91] P. Le Guernic et al. *Programming Real-Time Applications with SIGNAL*.  
Proceedings of the IEEE: *Special Issue on Real-Time Systems*, Vol. 79, N<sup>o</sup> 9, September 1991.
- [Wood92] M. Wood. *Fault-Tolerant Management of Distributed Applications using the Reactive System Architecture*. Ph. D. Thesis, Cornell University, 1992.

## CAPÍTULO 5

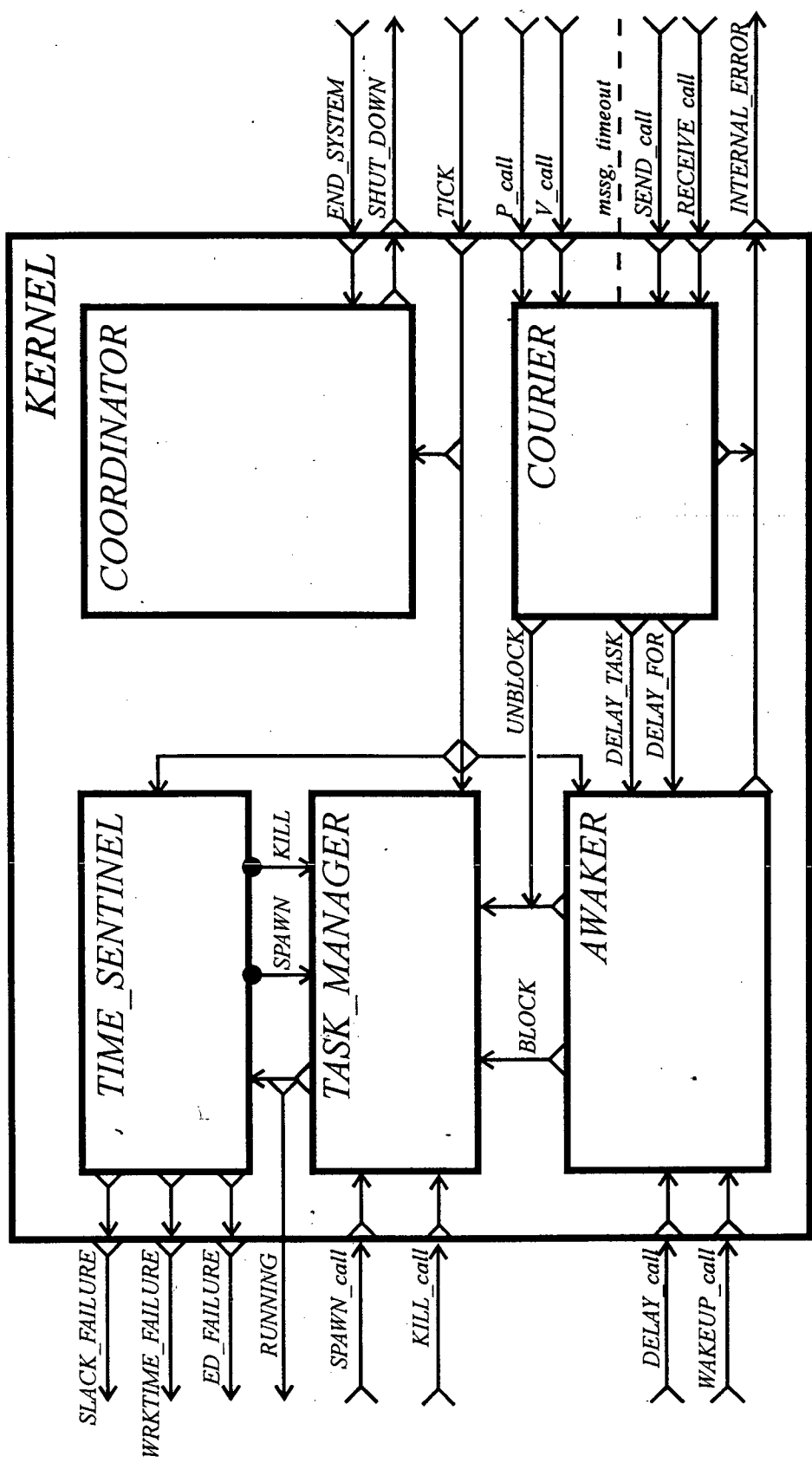
- [André93a] C. André and M.-A. Péraldi. *Effective Implementation of ESTEREL programs*.  
5<sup>th</sup> Euromicro Workshop on Real-Time Systems, Oulu (Finland), June 1993.
- [André93b] C. André. *Input/Output Simulator for the Esterel Execution Machine*.  
TR93-09, I3S Lab, Sophia Antipolis (France), February 1993.
- [André93c] C. André and M.-A. Péraldi. *Using the Synchronous Language "Esterel" in Simulation*.  
MIM-S2 '93, Brussels, April 1993, pp83-88.
- [Berry91] G. Berry. *Incremental Development of an HDLC entity using Esterel*.  
Computer Networks and ISDN Systems, 22 (1991) 35-49.
- [Berry93] G. Berry. *Communicating Reactive Processes*. POPL'93, Charleston, South Carolina, USA.
- [Boussinot91] F. Boussinot and R. de Simone. *The ESTEREL Language*.  
Rapports de Recherche 1487, INRIA, July 1993. Also in Proceedings of the IEEE: *Special Issue on Real-Time Systems*, Vol. 79, N<sup>o</sup> 9, September 1991.
- [CISI88Doc] Cisi Engenharia. *Esterel V3 - Documentation*.  
CISI ENGENHARIA, INRIA, Ecole Des Mines, Armines, 1988.
- [Halbwachs91] N. Halbwachs. *The Synchronous Data Flow Programming Language LUSTRE*.  
Proceedings of the IEEE: *Special Issue on Real-Time Systems*, Vol. 79, N<sup>o</sup> 9, September 1991.
- [Wagner90] T. Wagner. *CTask: A Multitasking Kernel for C - Version 2.2*. Ferrari Electronic GmbH, 1990.

## **ANEXO 1: CODIGO ESTEREL DO NÚCLEO**

A seguir apresenta-se o código esterel que implementa a Entidade Reativa do nosso núcleo de tempo real. Cada um dos módulos componentes é apresentado num formato gráfico tipo *blueprint*, e depois é mostrado seu código.



L.CMI EEL	Nome do Projeto:		Etapas:		Leflor	Data
	Autor(es):		Proposta	Publicação		
Data:		Versão:		Aprovação		



Nº:	Referência
Título:	



```

input TICK;

input END_SYSTEM; % it can also be emitted from outside (from CLOCK)
procedure KERNEL_SHUTDOWN(); %only executed in simulation mode

%----- instruction part -----
input NEW_APP_TIME: TIME_TYPE;

loop
  var App_feeder_time: TIME_TYPE
  in
    do
      do
        nothing
        watching App_feeder_time TICK
        timeout
          emit UNBLOCK( APP_FEEDER())
        end
        watching NEW_APP_TIME
        App_feeder_time := ?NEW_APP_TIME;
      end
    end
end

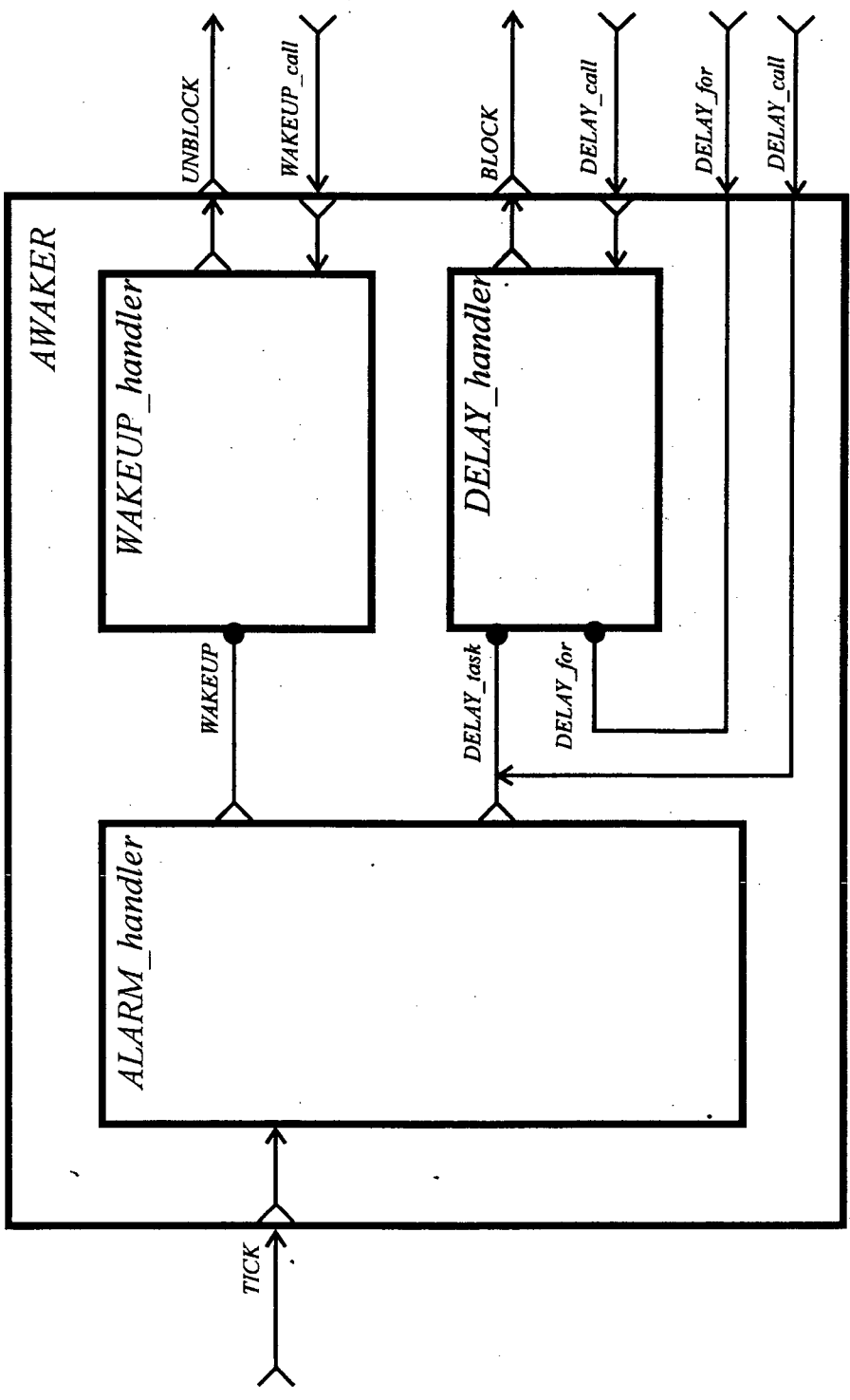
end module %COORDINATOR
%-----

% TASK_MANAGER
%-----%-----%-----%-----%-----%-----%-----%-----%-----
module TASK_MANAGER

%----- interface part -----
type TASK_TYPE, TASK_QUEUE;
%----- declaration part -----
input BLOCK( TASK_TYPE),
  UNBLOCK( TASK_TYPE),
  SPAWN_task( TASK_TYPE),
  KILL_task( TASK_TYPE);
output RUNNING( TASK_TYPE);
procedure TASK_SCHEDULE( TASK_QUEUE);
%----- instruction part -----
every [BLOCK or UNBLOCK]
  emit RUNNING( SCHEDULE( READY_QUEUE) );
end
||
every SPAWN_call
  emit SPAWN( ?SPAWN_call);
  call SPAWN_task( ?SPAWN_call);
end
||
every KILL_call
  emit KILL( ?KILL_call);
  call KILL_task( ?KILL_call);
end
end module %TASK_MANAGER
%-----

```

LCMI EEL	Nome do Projeto:		Etapas:		Leitor	Data
	Autor(es):		Proposta			
Data:		Versão:		Aprovação		
				Publicação		



Nó:	Referência
Título:	

```

% AWAKER
%-----%-----%-----%-----%-----%-----%-----%-----%-----%
module AWAKER: #temporal services provider

%----- interface part -----
type TASK_TYPE, TIME_TYPE, ALARM_TYPE;

%----- declaration part -----
input DELAY_call( TASK_TYPE),
      WAKEUP_call( TASK_TYPE),
      TICK;
relation DELAY_call # WAKEUP_call # TICK;

inputoutput DELAY_task( TASK_TYPE), DELAY_for( TIME_TYPE);

output BLOCK( TASK_TYPE);

function SET_ALARM( TIME_TYPE): ALARM_TYPE,
      OFF_ALARM( TIME_TYPE):ALARM_TYPE,
      SCHEDULE( QUEUE_TYPE):TASK_TYPE,
      FIRST_DELAYED_TASK():TASK_TYPE,
      ALARM_queue( TIME_TYPE): QUEUE_TYPE;
procedure ENQUEUE( TASK_TYPE, QUEUE_TYPE)();

%----- instruction part -----
do
  var No_ticks, Next_alarm:TIME_TYPE, task:TASK_TYPE
  in
    copymodule ALARM_handler
    ||
    copymodule DELAY_handler
    ||
    copymodule WAKEUP_handler
  end
end module %AWAKER
%-----%-----%-----%-----%-----%-----%-----%-----%-----%

% DELAY_handler
%-----%-----%-----%-----%-----%-----%-----%-----%-----%
module DELAY_handler:

%----- interface part -----
%----- declaration part -----
%----- instruction part -----
every DELAY_call do
  emit DELAY_task( ?RUNNING) # front end
  # DELAY_TASK can only be
  # called by the current task
  emit DELAY_for( ?DELAY_call)
end
||
var Alarm: TIME_TYPE in
every DELAY_task do
  Alarm:= SET_ALARM( ?DELAY_for);

```

```

    call ENQUEUE( ?DELAY_task, ALARM_queue( Alarm) ) ( ) ;
    emit BLOCK( ?DELAY_task);
end
end var

end module %DELAY_handler
%-----

% WAKEUP_handler
%-----%-----%-----%-----%-----%-----
module WAKEUP_handler:    # wakeup(task) service request

%----- interface part -----
%----- declaration part -----

%----- instruction part -----
every WAKEUP_call do    # it's argument is the task to be awakened
    emit WAKEUP( ?WAKEUP_call);
    call OFF_ALARM( ?WAKEUP_call); # it also updates
        # NEXT_ALARM value
end
||
every WAKEUP do
    emit UNBLOCK( ?WAKEUP);
    call UNQUEUE( ?WAKEUP);
end

end module %WAKEUP_handler
%-----

% ALARM_handler
%-----%-----%-----%-----%-----%-----
module ALARM_handler:    # update alarm list on TICK

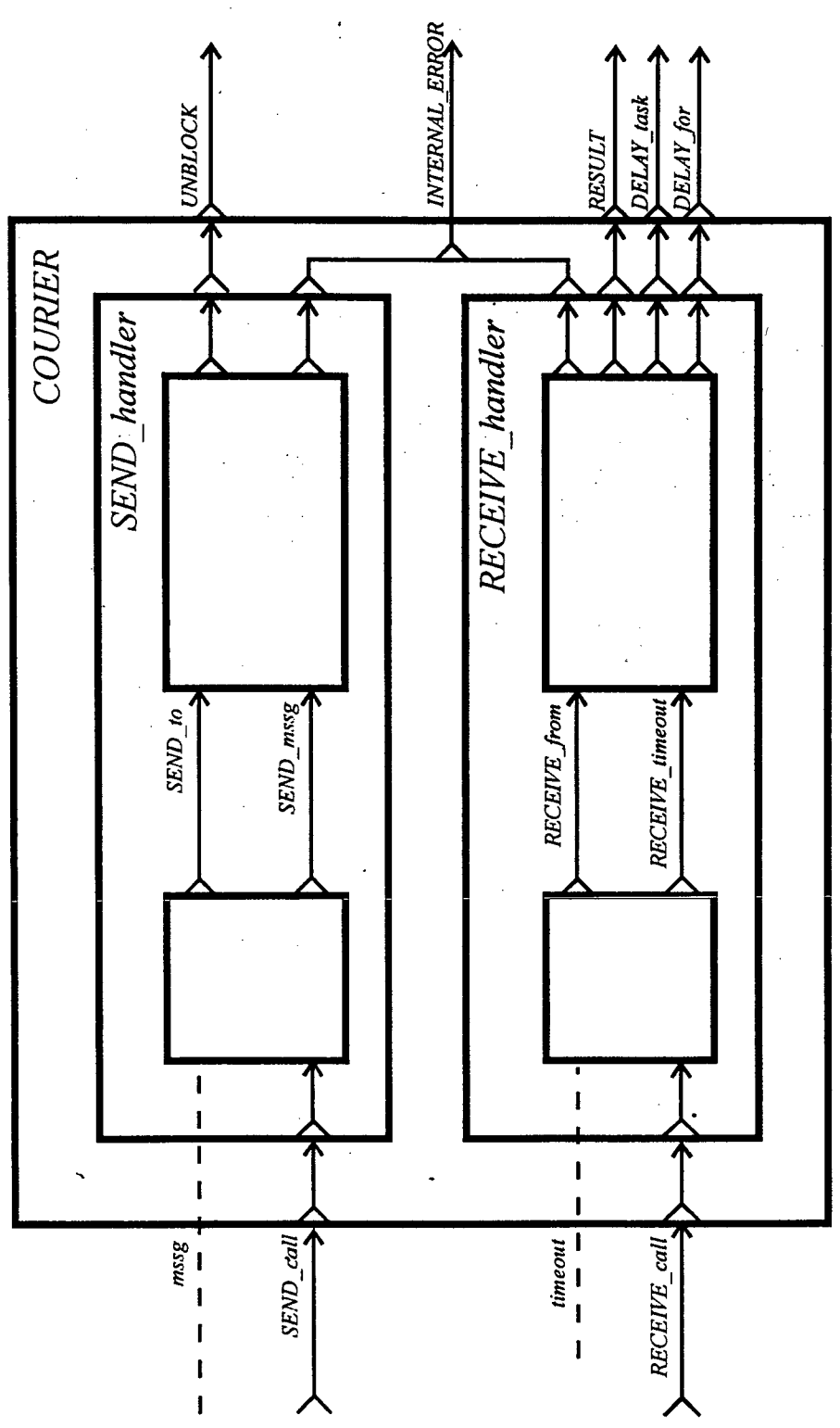
%----- interface part -----
%----- declaration part -----

%----- instruction part -----
loop
    do
        nothing
        watching Next_alarm TICK
        timeout
            emit WAKEUP( FIRST_DELAYED_TASK());
        watching [DELAY_TASK or WAKEUP]
    end
end

end module %ALARM_handler
%-----

```

Nome do Projeto: Autor(es): Data:	Versão:			Lector	Data
	Etapas: Proposta Aprovação Publicação				
L.CMI EEL					



Nó:	Título:	Referência
-----	---------	------------

```

% COURIER
%-----%-----%-----%-----%-----%-----
module COURIER:

%----- interface part -----
type MBOX_TYPE, TASK_TYPE, QUEUE_TYPE, MSSG_TYPE, TIME_TYPE;

%----- declaration part -----
input SEND_call( MBOX_TYPE),
      RECEIVE_call( MBOX_TYPE);

output BLOCK( TASK_TYPE),
      UNBLOCK( TASK_TYPE),
      MESSAGE( MSSG_TYPE);
output DELAY_task( TASK_TYPE),
      DELAY_for( TIME_TYPE);

ouput INTERNAL_ERROR( string);
sensor mssg;
sensor timeout: TIME_TYPE;

function MBOX_TASK_queue( MBOX_TYPE): QUEUE_TYPE;
function EMPTY( QUEUE_TYPE): boolean;
procedure SAVE_MAIL( QUEUE_TYPE, MSSG_TYPE);
procedure ENQUEUE( TASK_TYPE, QUEUE_TYPE);

%----- instruction part -----
  copymodule SEND_handler
  ||
  copymodule RECEIVE_handler
  ||

end module %COURIER
%-----

% SEND_handler
%-----%-----%-----%-----%-----%-----
module SEND_handler:

%----- interface part -----
%----- declaration part -----
%----- instruction part -----
signal SEND_to: MBOX_TYPE, SEND_mssg: MSSG_TYPE
in
every SEND_call do
  emit SEND_to( ?SEND_call)
  emit SEND_mssg( ?mssg)
end
  ||
every SEND_to do
  present SEND_mssg then # verify it's emitted simultaneously
    # with SENDINFO
    if EMPTY( MBOX_TASK_queue( ?SEND_to) ) then

```



```

        call SAVE_MAIL( MBOX_MSSG_queue( ?SEND_to),END_mssg())
    else
        emit UNBLOCK( MBOX_SCHEDULE( MBOX_MSSG_queue( SEND_TO)))
        emit MESSAGE( ?SEND_mssg)
    end if
else
    emit INTERNAL_ERROR( "COURIER/SEND_handler");
end present
end every

```

```
end signal
```

```
end module %SEND_handler
```

```
%-----
```

```
% RECEIVE_handler
```

```
%-----%-----%-----%-----%-----
```

```
module RECEIVE_handler:
```

```
%----- interface part -----
```

```
%----- declaration part -----
```

```
inputoutput RECEIVE_from: MBOX_TYPE,
    RECEIVE_timeout: TIME_TYPE;
```

```
%----- instruction part -----
```

```
every RECEIVE_call do
```

```
    emit RECEIVE_from( ?RECEIVE_call)
```

```
    emit RECEIVE_timeout( ?timeout);
```

```
end
```

```
||
```

```
every RECEIVE_from do
```

```
    present RECEIVE_timeout then # verify it's emitted
```

```
        # simultaneously with RECEIVE_timeout
```

```
        if EMPTY( MBOX_MSSG_queue( ?RECEIVE_from) then
```

```
            call ENQUEUE( ?RUNNING, MBOX_MSSG_queue( ?RECEIVE_FROM))
```

```
            emit DELAY_task( ?RUNNING)
```

```
            emit DELAY_for( ?RECEIVE_timeout)
```

```
        else
```

```
            emit MESSAGE( MBOX_SCHEDULE( MBOX_MSSG_queue( ?RECEIVE_from)))
```

```
        end if
```

```
    else
```

```
        emit INTERNAL_ERROR( "COURIER/RECEIVE_handler");
```

```
    end present
```

```
end
```

```
end module %RECEIVE_handler
```

```
%-----
```

```
%-----
```

```
% SEMAPHORE
```

```
%-----%-----%-----%-----%-----
```

```
module SEMAPHORE:
```

```

%----- interface part -----
type TASK_TYPE, SEMAPHORE_TYPE;

%----- declaration part -----
inputoutput DELAY_task( TASK_TYPE)
        DELAY_for( TIME_TYPE);

input P_call( SEMAPHORE_TYPE),
        V_call( SEMAPHORE_TYPE);
relation P_call # V_call

output BLOCK: TASK_TYPE,
        UNBLOCK: TASK_TYPE,
        RUNNING: TASK_TYPE;
sensor timeout( TIME_TYPE);

procedure DECREMENT( SEMAPHORE_TYPE)(),
        INCREMENT( SEMAPHORE_TYPE)(),
        ENQUEUE( TASK_TYPE, QUEUE_TYPE)();
function VALUE( SEMAPHORE_TYPE): integer;
function SEMA_SCHEDULE( QUEUE_TYPE): SEMAPHORE_TYPE;

%----- instruction part -----
    copymodule P
||
    copymodule V

end module %SEMAPHORE
%-----

% P_handler
%-----%-----%-----%-----%-----%-----
module P_handler:

%----- interface part -----
%----- declaration part -----
%----- instruction part -----
every P_call do
    call DECREMENT( ?P_call)()
    if VALUE( ?P_call) < 0
        emit DELAY_TASK( ?RUNNING); # only the current task can
            # emit this signal
        emit DELAY_FOR( ?timeout);
        call ENQUEUE( ?RUNNING, SEMA_task_queue( ?P_call) )
    end
end

end module %P_handler
%-----

% V_handler
%-----%-----%-----%-----%-----
module V_handler:

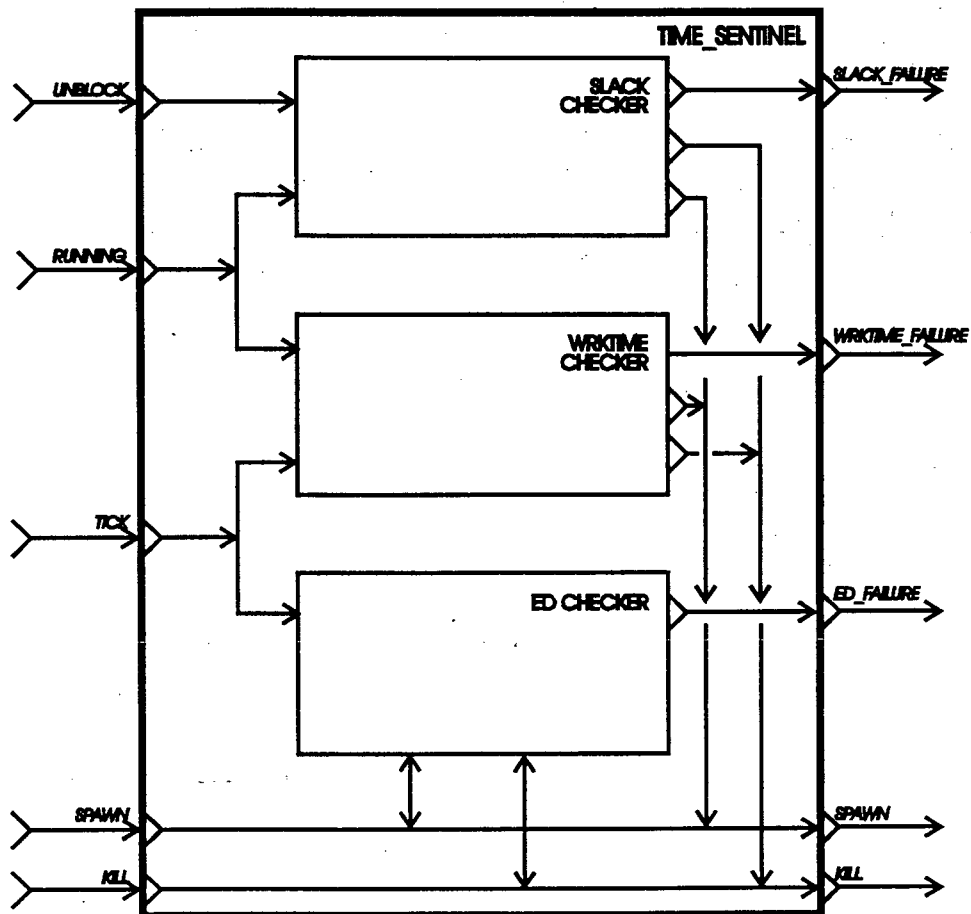
```

```
%----- interface part -----
%----- declaration part -----
procedure INCREMENT( semaphore_type);
function VALUE( semaphore_type): integer;
function SCHEDULE( task_list_type): task_type;
function TASK_QUEUE(?? ): task_queue_type;

%----- instruction part -----
every V_call do
  call INCREMENT( ?V_call)()
  if VALUE( ?V_call) < 1
    emit UNBLOCK( SEMA_SCHEDULE( SEMA_task_queue( ?V_call)() ))
  end
end

end module %V_handler
%-----
%-----
```

Nome do Projeto: Autor(es): Data:	Etapas:			Leitor	Data
	Proposta	Aprovação	Publicação		
Nome do Projeto: Autor(es): Data:	Versão:				



Nó:	Título:	Referência
-----	---------	------------





## ANEXO 2: EXEMPLO DE ESCALONAMENTO

A seguir é mostrado o resultado do escalonamento do conjunto de tarefas do exemplo 5.6 pelos distintos algoritmos disponíveis.

### VETOR DE TESTE:

CONFIGURATION FILE: testvct2.tst

Task Set: Same as Stewart's, but without task 'D'

Number of tasks in the set: 3

Task Set Description:

Name	Criticality	Period	ExecTime	Task Load
Task A	high	6	2	33.3%
Task B	high	10	4	40.0%
Task C	high	12	3	25.0%

### RESULTADOS DO SEU ESCALONAMENTO:

Selected Scheduling Algorithm: Rate Monotonic (RM),  
which has a schedulability bound of 78.0% for 3 tasks.

Critical set is composed of

Task A,

Task B,

which accounts for a critical load of 73.3%, over a total system load of 98.3%

WARNING: the whole task set MAY NOT be schedulable under RM

At 12: task c ("Task C"), instance 1, Deadline Failure

At 24: task c ("Task C"), instance 2, Deadline Failure

TIMELINE: [Rate Monotonic (RM)]

a b a c b a b c a b a c Za

0123456789012345678901234567890

0 . 1 2 3

13 context switches

Cross-reference Names:

a Task A

b Task B

c Task C

. . . . .

Selected Scheduling Algorithm: Earliest-Deadline-First (EDF),

which has a schedulability bound of 100%

Total system task load = 98.3%

So, the whole task set IS schedulable under EDF

TIMELINE: [Earliest-Deadline-First (EDF)]

a b a c b a b c a b a c  
0123456789012345678901234567890  
0 1 2 3

12 context switches

Cross-reference Names:

a Task A  
b Task B  
c Task C

.....

Selected Scheduling Algorithm: Least-Laxity-First (LLF),  
which has a schedulability bound of 100%  
Total system task load = 98.3%  
So, the whole task set IS schedulable under LLF

TIMELINE: [Least-Laxity-First (LLF)]

a b c a c b a b c a c b a b c a  
0123456789012345678901234567890  
0 1 2 3

16 context switches

Cross-reference Names:

a Task A  
b Task B  
c Task C

.....

Selected Scheduling Algorithm: Maximum-Urgency-First (MUF),  
which has a schedulability bound of 100%  
Critical set is composed of

Task A,  
Task B,  
Task C,

which accounts for a critical load of 98.3%, over a total system load of 98.3%  
So, the whole task set MAY BE schedulable under MUF

TIMELINE: [Maximum-Urgency-First (MUF)]

a b c a c b a b c a c b a b c a  
0123456789012345678901234567890  
0 1 2 3

16 context switches

Cross-reference Names:

a Task A  
b Task B  
c Task C



Vemos que neste caso, o vetor de teste não consegue ser escalonado pelo algoritmo taxa monotônica: como o vetor não cumpre com a condição suficiente de escalonabilidade teórica, o simulador adverte ao usuário sobre o risco potencial. Já os algoritmos earliest deadline e menor folga conseguem escaloná-lo, este último de maneira menos eficiente pelo maior número de trocas de contexto. Finalmente, o algoritmo de máxima urgência também adverte o perigo decorrente do não cumprimento da condição de escalonabilidade do taxa monotônica (no qual está baseado), mas acaba escalonando-o satisfatoriamente.