UNIVERSIDADE REGIONAL DE BLUMENAU

CENTRO DE CIÊNCIAS EXATAS E NATURAIS

CURSO DE CIÊNCIAS DA COMPUTAÇÃO

(Bacharelado)

APLICAÇÃO EM TEMPO REAL UTILIZANDO A LINGUAGEM DE PROGRAMAÇÃO ERLANG

TRABALHO DE CONCLUSÃO DE CURSO SUBMETIDO À UNIVERSIDADE REGIONAL DE BLUMENAU PARA A OBTENÇÃO DOS CRÉDITOS NA DISCIPLINA COM NOME EQUIVALENTE NO CURSO DE CIÊNCIAS DA COMPUTAÇÃO — BACHARELADO

AMILTON CESAR SCHMIDT

BLUMENAU, MARÇO/2001

2000/2-5

APLICAÇÃO EM TEMPO REAL UTILIZANDO A LINGUAGEM DE PROGRAMAÇÃO ERLANG

AMILTON CESAR SCHMIDT

ESTE TRABALHO DE CONCLUSÃO DE CURSO, FOI JULGADO ADEQUADO PARA OBTENÇÃO DOS CRÉDITOS NA DISCIPLINA DE TRABALHO DE CONCLUSÃO DE CURSO OBRIGATÓRIA PARA OBTENÇÃO DO TÍTULO DE:

BACHAREL EM CIÊNCIAS DA COMPUTAÇÃO

_____

Prof. José Roque Voltolini da Silva — Orientador na FURB

_____

Prof. José Roque Voltolini da Silva — Coordenador do TCC

BANCA EXAMINADORA

_____

Prof. José Roque Voltolini da Silva

_____

Prof. Antonio Carlos Tavares

_____

Prof. Dalton Solano dos Reis

Dedico este trabalho à minha esposa, à minha família, aos professores,

pelo apoio recebido não apenas neste momento, mas

durante toda minha formação acadêmica.

AGRADECIMENTOS

Primeiramente, gostaria de agradecer a Deus, pela oportunidade de viver, de estudar, de cursar uma faculdade. Ao mesmo tempo, pedir forças para, na vida, continuar seguindo o caminho que Ele me aponta.

Gostaria de agradecer ao meu orientador, Prof. José Roque Voltolini da Silva, que me deu todo o apoio e incentivo na confecção deste trabalho.

A meus pais, Paulo Schmidt e Cellecina Marquez Schmidt, meus irmãos, enfim, todos os meus familiares, pela oportunidade de estudar e cursar uma faculdade e também pelo incentivo que me deram ao longo de minha vida.

A minha esposa, Kátia Regina Vargas Schmidt, por vivenciar todos os passos deste trabalho, oferecendo-me, através de sua presença, apoio e incentivo, em todos os momentos.

Agradeço também, minha professora de primário (1ª à 4ª série), Maria Terezinha Fernandes Reitz, que ensinou-me os primeiros passos no caminho de minha formação, e aos frades franciscanos da Província Imaculada Conceição do Brasil, dos quais recebi minha formação no ensino médio (1º e 2º grau).

A todos os colegas e amigos do Curso de Ciências da Computação, conquistados durante minha vida acadêmica.

Aos meus amigos de trabalho da Souza Cruz, que acompanharam mais este passo importante em minha vida.

A todos aqueles que direta ou indiretamente contribuíram para a realização deste trabalho.

SUMÁRIO

AGRADECIMENTOS iv

SUMÁRIO v

LISTA DE FIGURAS ix

LISTA DE QUADROS x

LISTA DE QUADROS x

RESUMO xii

ABSTRACT xiii

1 INTRODUÇÃO 1

1.1 MOTIVAÇÃO 2

1.2 OBJETIVOS 3

1.3 ORGANIZAÇÃO DO TEXTO 3

2 PROCESSOS CONCORRENTES 4

2.1 PRINCIPAIS CONCEITOS 4

2.2 PROGRAMAÇÃO SEQUENCIAL X CONCORRENTE 6

2.3 SISTEMA OPERACIONAL 6

2.4 TAREFAS E *THREADS* 7

2.4.1 ESTADOS DE UMA TAREFA 8

2.4.2 REGIÃO CRÍTICA 10

2.4.3 SINCRONIZAÇÃO ENTRE TAREFAS 11

2.4.3.1 SINCRONIZAÇÃO DE COOPERAÇÃO 11

2.4.3.2 SINCRONIZAÇÃO DE COMPETIÇÃO 11

2.4.4 SEMÁFOROS 13

2.4.5 MONITORES 13
2.4.6 PASSAGEM DE MENSAGEM 14
2.5 VANTAGENS DA CONCORRÊNCIA 15
3 SISTEMAS DE TEMPO REAL 17
3.1 TEMPO REAL HOJE 17
3.2 CONCEITOS BÁSICOS 18
3.3 CRÍTICO X NÃO CRÍTICO 19
3.4 PREVISIBILIDADE 21
3.5 SISTEMAS OPERACIONAIS EM TEMPO REAL 22
3.6 TAREFA EM TEMPO REAL 22
3.7 ESCALONAMENTO DE TAREFAS 23
3.7.1 NÍVEIS DE ESCALONAMENTO 24
3.8 ABORDAGENS DE TEMPO REAL 24
3.8.1 ABORDAGEM ASSÍNCRONA 25
3.8.2 ABORDAGEM SÍNCRONA 25
4 A LINGUAGEM ERLANG 27
4.1 O AMBIENTE DE DESENVOLVIMENTO ERLANG 27
4.1.1 O *SHELL* DO ERLANG 27
4.2 TIPOS DE DADOS 28
4.3 PRINCIPAIS MECANISMOS 33
4.3.1 PADRÃO EMPARELHADO (*PATTERN MATCHING*) 34
4.3.1.1 PADRÃO = EXPRESSÃO 34
4.3.1.2 PATTERN MATCHING QUANDO UMA FUNÇÃO É CHAMADA 35
4.3.2 MODULE 36
4.3.3 CLÁUSULAS 37

4.3.3.1 CABEÇA DAS CLÁUSULAS 38
4.3.3.2 CLÁUSULA *GUARD* 38
4.3.3.3 CORPO DA CLÁUSULA 39
4.3.4 PRIMITIVAS *CASE* E *IF* 40
4.3.5 GARBAGE COLLECTION 41
4.4 CONCORRÊNCIA E TEMPO REAL EM ERLANG 41
4.4.1 CRIAÇÃO DE PROCESSOS 41
4.4.2 COMUNICAÇÃO ENTRE PROCESSOS 42
4.4.2.1 RECEBIMENTO DE MENSAGENS DE UM PROCESSO ESPECÍFICO 44
4.4.2.2 EXEMPLOS ILUSTRATIVOS 45
4.4.3 TIMEOUTS 48
4.4.4 REGISTRO DE PROCESSOS 51
4.4.5 ESCALONAMENTO DE PROCESSOS E TEMPO REAL 52
4.4.6 PRIORIDADES DE PROCESSOS 53
4.5 AMBIENTE GRÁFICO 54
4.5.1 MODELO GS 54
4.5.2 FUNÇÕES GS 54
4.5.3 OBJETOS GS 55
5 DESCRIÇÃO DO MODELO DE ESPECIFICAÇÃO 58
5.1 DIAGRAMA DE TRANSIÇÃO DE ESTADOS 58
5.1.1 TABELA DE TRANSIÇÃO DE ESTADO 60
5.1.2 MATRIZ DE TRANSIÇÃO DE ESTADO 61
6 ESPECIFICAÇÃO DO PROTÓTIPO 63
7 IMPLEMENTAÇÃO E APRESENTAÇÃO DO PROTÓTIPO 65
7.1 PROPRIEDADES DO PROTÓTIPO 65

7.2 DESCRIÇÃO DAS PRINCIPAIS FUNÇÕES DO PROTÓTIPO 66
7.3 APRESENTAÇÃO DO PROTÓTIPO 77
8 CONSIDERAÇÕES FINAIS 79
8.1 DIFICULDADES ENCONTRADAS 79
8.2 LIMITAÇÕES 80
8.3 EXTENSÕES 80
REFERÊNCIAS BIBLIOGRÁFICAS 81

LISTA DE FIGURAS

Figura 1: Suspensão e retomada de tarefas em um ambiente multitarefa 5

Figura 2: Possíveis transições entre os estados de uma tarefa 9

Figura 3: A necessidade de sincronização de competição 12

Figura 4: Processamento em série de entradas concorrentes 15

Figura 5: Processamento em paralelo de entradas concorrentes 16

Figura 6: Classificações dos sistemas de tempo real 20

Figura 7: Ambiente Erlang 28

Figura 8: Manipulações de *tuples* e *lists* no ambiente Erlang 32

Figura 9: Máquina de estado finito 48

Figura 10: Visualização da interface gerada pelo código do quadro 33 57

Figura 11: Diagrama de transição de estados 59

Figura 12: DTEs do protótipo 63

Figura 13: Representação da tela principal 68

Figura 14: Representação dos botões da malha ferroviária 69

Figura 15: Representação das linhas 71

Figura 16: Tela do protótipo com seu funcionamento 77

LISTA DE QUADROS

Quadro 1: Operações aritméticas 29

Quadro 2: Convenções para uso dos caracteres 30

Quadro 3: Estrutura *Reference* 31

Quadro 4: Funções das *lists* 33

Quadro 5: Estrutura do *pattern matching* 34

Quadro 6: *Pattern Matching* = Expressão 35

Quadro 7: Chamada de função do *pattern matchig* 35

Quadro 8: Estrutura de um módulo 36

Quadro 9: Modelo de chamadas nos módulos 37

Quadro 10: Chamada nos módulos usando a primitiva *import* 37

Quadro 11: Programa fatorial 38

Quadro 12: Programa fatorial invertido 38

Quadro 13: Operações *guards* 39

Quadro 14: Corpo da cláusula 39

Quadro 15: Estrutura da primitiva *case* 40

Quadro 16: Estrutura da primitiva *if* 41

Quadro 17: Função spawn 42

Quadro 18: Passagem de mensagem 42

Quadro 19: Transmissão da mensagem 43

Quadro 20: Estrutura do *receive* 44

Quadro 21: Especificando o processo 44

Quadro 22: Recebimento de mensagem de um processo específico 45

Quadro 23: Exemplo de um contador 45

Quadro 24: Exemplo de um contador melhorado 46

Quadro 25: Representação do código Erlang 48
Quadro 26: Sintaxe do *receive* com *timeout* 49
Quadro 27: Função *sleep* 49
Quadro 28: Função *flush_buffer ( )* 50
Quadro 29: Prioridade de mensagens no *timeout* 50
Quadro 30: *Timeout* independente 51
Quadro 31: Processos registrados 52
Quadro 32: Função *process_flag*( ) 53
Quadro 33: Características do ambiente gráfico *gs* 57
Quadro 34: Tabela de transição de estado 61
Quadro 35: Matriz de transição de estado 61
Quadro 36: Criação dos processos cruzamento 66
Quadro 37: Criação da tela principal 67
Quadro 38: Botões que representam as cidades 69
Quadro 39: A função linha 70
Quadro 40: A função cria_trem 71
Quadro 41: A função trem 72
Quadro 42: A função cruzamento 73
Quadro 43: A função direção 74
Quadro 44: A função tempo 75
Quadro 45: Semáforo binário 76

RESUMO

Este trabalho realiza um estudo da linguagem de programação Erlang. Uma avaliação da linguagem é feita através do desenvolvimento de uma aplicação de uma malha ferroviária. Aspectos como processos concorrentes e tempo real são considerados no desenvolvimento do trabalho.

ABSTRACT

This work realize a study Erlang programming language. A valuation is maked through aplication in the railroad control. Aspects with process concurrent and real time are considered in the work development.

1 INTRODUÇÃO

No cenário atual de desenvolvimento de softwares, existem diversas propostas de linguagens de programação. Umas já bem conhecidas, como Pascal, C, COBOL, outras mais recentes, como Object Pascal, Java, entre outras, cada uma com o seu valor e características particulares.

Segundo [GHE1991], as linguagens de programação tem o propósito de produzir softwares e são uma das ferramentas necessárias para esta tarefa. Portanto, toda linguagem tem por objetivo principal o desenvolvimento de programas ou sistemas de computação, conforme as características de seu projeto.

Quando surge no cenário de programação uma nova proposta, nasce também a necessidade de estudá-la. Conforme [SEB2000] existem alguns benefícios potenciais no estudo dos conceitos de uma linguagem, tais como:

a) aumento da capacidade de expressar idéias;

b) maior conhecimento para a escolha de uma linguagem apropriada;

c) capacidade aumentada para aprender novas linguagens;

d) entender melhor a importância da implementação;

e) aumento da capacidade de projetar novas linguagens;

f) avanço global da computação.

Entre estes benefícios, destaca-se o maior conhecimento para a escolha de uma linguagem apropriada para uma determinada aplicação, por exemplo, uma aplicação de tempo real. Portanto, o estudo de uma linguagem de programação visa o conhecimento de suas principais características que a possam destacar no aspecto desenvolvimento e complexidade em relação a outras linguagens. E por isso, escolheu-se a linguagem Erlang para estudo.

Em [ERI2000], encontram-se as principais vantagens da linguagem Erlang, as quais são:

a) redução do tempo de desenvolvimento e de correção de erros;

b) código mais simples e de fácil entendimento;

c) o código pode ser substituído ou alterado enquanto o sistema está operando;

d) possui plataforma independente, o que permite fácil comunicação com

outras linguagens como C++ ou Java;

e) soluções completas para problemas básicos;

f) possui tratamento de exceções e processos distribuídos;

g) explora conceitos de processos concorrentes e tempo real.

A linguagem de programação Erlang surgiu da necessidade Empresa Ericsson de encontrar uma linguagem com características que pudessem suportar sistemas complexos, especialmente na área de telecomunicações. Em sua busca, não foi encontrada no mercado uma linguagem que satisfizesse as necessidades da empresa. Com isto, em 1987, a própria Ericsson, na Suécia, com o apoio da *Ellemtel Computer Science Laboratories*, iniciou em seu laboratório de computação o desenvolvimento de uma nova proposta. Uma equipe encabeçada por Bjarne Däcker trabalhou durante vários anos no estudo desta idéia e na implementação deste projeto. Em 1994, a linguagem foi oficialmente lançada com o nome Erlang.

O nome Erlang foi uma homenagem ao matemático dinamarquês Agner Krarup Erlang (1878 – 1929) que desenvolveu a teoria de processos de *stochastic* no equilíbrio estatístico. Sua teoria é amplamente utilizada na área de telecomunicações ([ARM1993]).

Diante destes aspectos, este trabalho propõe o estudo desta linguagem. Suas características expressivas no ambiente atual do mundo da informática, tais como concorrência, comunicação de processos, tratamento de erros e tempo real, necessitam de uma avaliação com o intuito de verificar o seu aproveitamento no desenvolvimento de novas aplicações. A Ericsson criou esta linguagem com o objetivo de usá-la em aplicações de telecomunicações que é o foco de seu negócio, porém, conforme [ERI2000], já existem 80 universidades ao redor do mundo utilizando esta proposta como ferramenta de estudo em suas salas de aula e laboratórios, objetivando, com certeza, a ampliação das áreas de aplicação desta linguagem.

Sendo assim, este trabalho procura desenvolver uma aplicação em tempo real para ser implementada na linguagem Erlang.

1.1 MOTIVAÇÃO

Esta proposta foi motivada principalmente pelas características e vantagens da

linguagem Erlang apresentada pela Ericsson em sua página na *internet* ([ERI2000]), sendo que não se tem conhecimento da utilização desta ferramenta na região de Blumenau. Outro motivo essencial foi a utilização de conceitos de processos concorrentes e tempo real, áreas pouco exploradas e atualmente tão importantes.

1.2 OBJETIVOS

O objetivo principal deste trabalho é desenvolver um estudo da Linguagem de Programação Erlang. Para tanto, uma avaliação da linguagem é realizada através do desenvolvimento de uma aplicação de tempo real.

O trabalho tem como objetivos secundários:

a) avaliar possíveis aplicações na linguagem Erlang;

b) verificar o comportamento da aplicação em tempo real na linguagem Erlang.

1.3 ORGANIZAÇÃO DO TEXTO

No capítulo 1 estão definidos os objetivos do trabalho, apresentando algumas considerações sobre o referido trabalho.

No capítulo 2 é apresentado aspectos sobre processos concorrentes.

No capítulo 3 é apresentado aspectos sobre sistemas de tempo real.

No capítulo 4 é apresentada a linguagem de programação Erlang, com suas características e aspectos importantes.

No capítulo 5 é apresentado o modelo de especificação que foi usado no trabalho.

No capítulo 6 é mostrada a especificação do protótipo para controle de uma malha ferroviária.

No capítulo 7 descreve-se a implementação do protótipo feita através da linguagem de programação Erlang e também é feita uma breve apresentação do protótipo, a nível de usuário.

No capítulo 8 encontram-se as considerações finais do trabalho.

2 PROCESSOS CONCORRENTES

Para o entendimento e estudo da linguagem de programação Erlang, como também de tempo real, é necessário o conhecimento dos principais conceitos de processos concorrentes. Por outro lado, no desenvolvimento da aplicação proposta neste trabalho, estes conceitos são fundamentais.

2.1 PRINCIPAIS CONCEITOS

Segundo [SEB2000], a concorrência é naturalmente dividida em:

a) nível de instrução: executa duas ou mais instruções de máquina simultaneamente;
b) nível de comando: executa dois ou mais comandos simultaneamente;
c) nível de unidade: executa duas ou mais unidades de subprogramas simultaneamente;
d) nível de programa: executa dois ou mais programas simultaneamente;

Um programa concorrente especifica duas ou mais listas de comandos que são executadas concorrentemente, como processos paralelos ([AMO1988]).

Segundo [AMO1988], processamento concorrente pode ser provido segundo as seguintes formas:

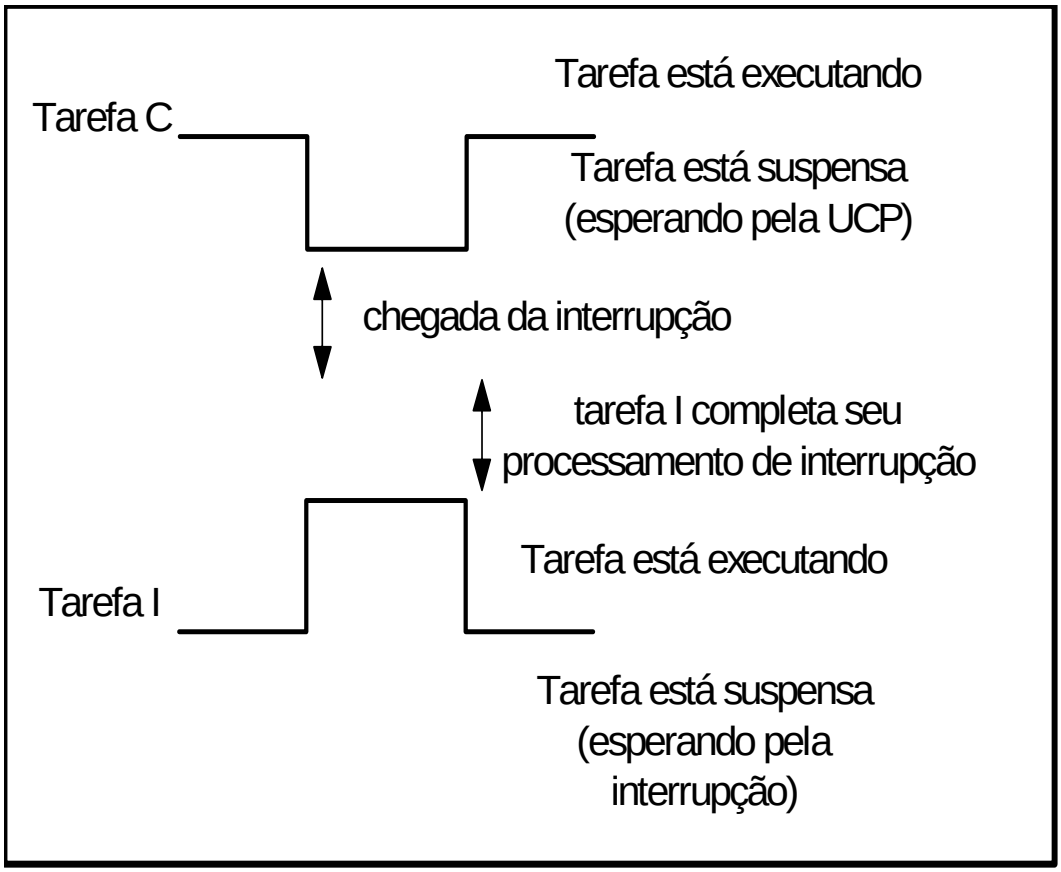
a) processamento distribuído: os processos são executados em paralelo por uma rede de processadores conectados por uma rede de comunicação;
b) multiprocessamento: os processos são executados em paralelo por uma rede de processadores fortemente conectados;
c) multiprogramação: na multiprogramação os processos compartilham pelo acesso a um processador comum. Neste caso a simultaneidade somente ocorre nos processos de E/S que podem estar sendo executados em paralelo com os outros tipos de instrução.

A execução concorrente de unidades de programa pode ocorrer fisicamente em processadores separados ou logicamente usando alguma forma de tempo fatiado em um sistema de computador de um único processador ([SEB2000]).

A necessidade de interromper uma parte de uma aplicação concorrente espontaneamente, para realizar captura de dados, forçam uma organização particular para estes tipos de programas. Aplicações concorrentes devem ser escritas como uma série de programas componentes separados, que podem ser executados concorrentemente ([RIP1993]). Esses componentes são chamados tarefas (*TASKS*) ou ainda processos e a organização é chamada multitarefa (*multitasking*).

Como cada tarefa é em si um programa viável, ela pode ser iniciada, suspensa, retomada e encerrada separadamente ([RIP1993]). Quando, por exemplo, uma interrupção é solicitada a tarefa C (figura 1), para suspender a execução em deferência da tarefa I, a tarefa C pára de executar, seu contexto é salvo, o contexto da tarefa I é instalado na máquina e se inicia a execução de I. Mais tarde a tarefa C terá seu contexto reinstalado na máquina e sua execução seguirá normalmente como se a interrupção de sua execução não tivesse ocorrido.

Figura 1: Suspensão e retomada de tarefas em um ambiente multitarefa



Fonte: [RIP1993]

2.2 PROGRAMAÇÃO SEQUENCIAL X CONCORRENTE

A característica marcante da programação seqüencial é a execução em seqüência das instruções do programa. É preciso que uma instrução seja completada para que a outra inicie sua execução. Deste modo, defini-se um único processo seqüencial, uma instrução executada após a outra, seguindo passo a passo a execução do programa, praticamente como uma receita.

Ao contrário da seqüencial, a programação concorrente controla a execução simultânea de vários processos seqüenciais. O programa não segue uma seqüência pré-definida.

A programação concorrente apresenta maiores dificuldades de implementação em relação à programação seqüencial. Neste sentido, pode-se destacar o *deadlock*, o bloqueio fatal de um programa que ocorre quando uma tarefa está esperando para que uma outra forneça uma determinada informação e esta tarefa, por algum motivo, já terminou. Este problema não aconteceria na programação seqüencial. Outro aspecto que pode ser citado, é o acesso dos processos às regiões críticas, apresentadas posteriormente neste trabalho.

Sistemas operacionais são exemplos de programas que sustentam maior concorrência, e segundo [RIP1993], são o ponto de foco natural destes sistemas, definindo o vocabulário com o qual o programa concorrente é escrito.

2.3 SISTEMA OPERACIONAL

O sistema operacional é o programa mestre. Ele decide qual tarefa executar no processador e realiza as trocas de contexto requeridas ([RIP1993]).

O sistema operacional é responsável pela organização de todo o trabalho do processador. Portanto, a execução e acesso aos recursos necessários aos processos concorrentes são controlados pelo sistema operacional.

Geralmente no trabalho concorrente, é comum haver uma quantidade razoável de tarefas executando concorrentemente, algumas sendo suspensas pela chegada de uma entrada que deve ser atendida imediatamente, outras sendo ativadas porque receberam os dados que necessitavam para sua ativação.

2.4 TAREFAS E *THREADS*

Uma tarefa é um programa completo, capaz de ser executado separadamente, contendo uma seqüência de código a ser executado, bem como sua própria pilha (*stack*) e suas próprias áreas locais de dados.

"Uma tarefa é uma unidade de um programa que pode estar em execução concorrente com outras unidades do mesmo programa" ([SEB2000]).

É interessante distinguir tarefas e subprogramas. Para isto, pode-se citar três características distintas:

a) as tarefas podem ser implicitamente iniciadas, já os subprogramas precisam ser chamados explicitamente;
b) quando uma unidade de programa chama uma tarefa, ela não precisa aguardar que esta termine sua execução antes de prosseguir por si mesma;
c) quando a execução de uma tarefa é concluída, o controle pode retornar ou não à unidade que iniciou essa execução.

Segundo [FAR2000], um programa que é executado por uma única tarefa é chamado de programa seqüencial. Neste caso, existe somente um fluxo de controle durante a execução. Um programa concorrente é executado simultaneamente por diversas tarefas que cooperam entre si, trocando informações, que significa trocar dados ou realizar algum tipo de sincronização. Portanto, é necessária a existência de interação entre tarefas para que o programa seja considerado concorrente.

Geralmente, os termos tarefa e processo são utilizados com o mesmo sentido. Tarefas ou processos são abstrações que incluem um espaço de endereçamento próprio (possivelmente compartilhado), um conjunto de arquivos abertos, um conjunto de direitos de acesso, um contexto de execução formado pelo conjunto de registradores do processador, além de vários outros atributos cujos detalhes variam de sistema para sistema. O tempo gasto para chavear o processador entre duas tarefas é definido por este conjunto de atributos, isto é, o tempo necessário para mudar o conjunto de atributos em vigor.

Conforme [FAR2000], o uso da abstração *thread* pode tornar a programação

concorrente mais eficiente. *Threads* são tarefas leves, no sentido de que os únicos atributos particulares que possuem são os que estão associados com o contexto de execução, ou seja, os registradores do processador. Os demais atributos são herdados da tarefa que a possue. Desta forma, o chaveamento entre duas *threads* de uma mesma tarefa é muito mais rápido que o chaveamento entre duas tarefas. Por exemplo, como todas as *threads* de uma mesma tarefa compartilham o mesmo espaço de endereçamento, a unidade de gerenciamento de memória não é afetada pelo chaveamento entre elas.

Assim, o termo tarefa ou processos pode ser usado para representar um conjunto de recursos como espaço de endereçamento, arquivos, *threads*, ao passo que o termo *thread* pode ser usado para denotar um fluxo de execução específico ([FAR2000]).

Em um programa, uma *thread* de controle é a ordem de pontos do programa atingidos à medida que o controle passa por ele. *Thread* é o encadeamento, isto é, a definição do caminho de execução dentro dos processos ([SEB2000]).

Uma *thread* serve para efetuar o rastreamento do fluxo de execução ao longo do programa-fonte, sendo que os programas concorrentes, podem ter várias *threads* de controle executadas pelos processadores. Quando um programa *multithread*, isto é, com duas ou mais *threads*, é executado em uma máquina com um único processador, suas *threads* são direcionadas para uma única *thread*, tornando o programa, neste caso, virtualmente em *multithread*.

2.4.1 ESTADOS DE UMA TAREFA

Conforme [MAC1994], uma tarefa, em um sistema multi-programável, não é executada durante todo o tempo pelo processador. Durante sua existência, a tarefa passa por uma série de estados. Basicamente, os estados em que uma tarefa pode se encontrar são classificados em:

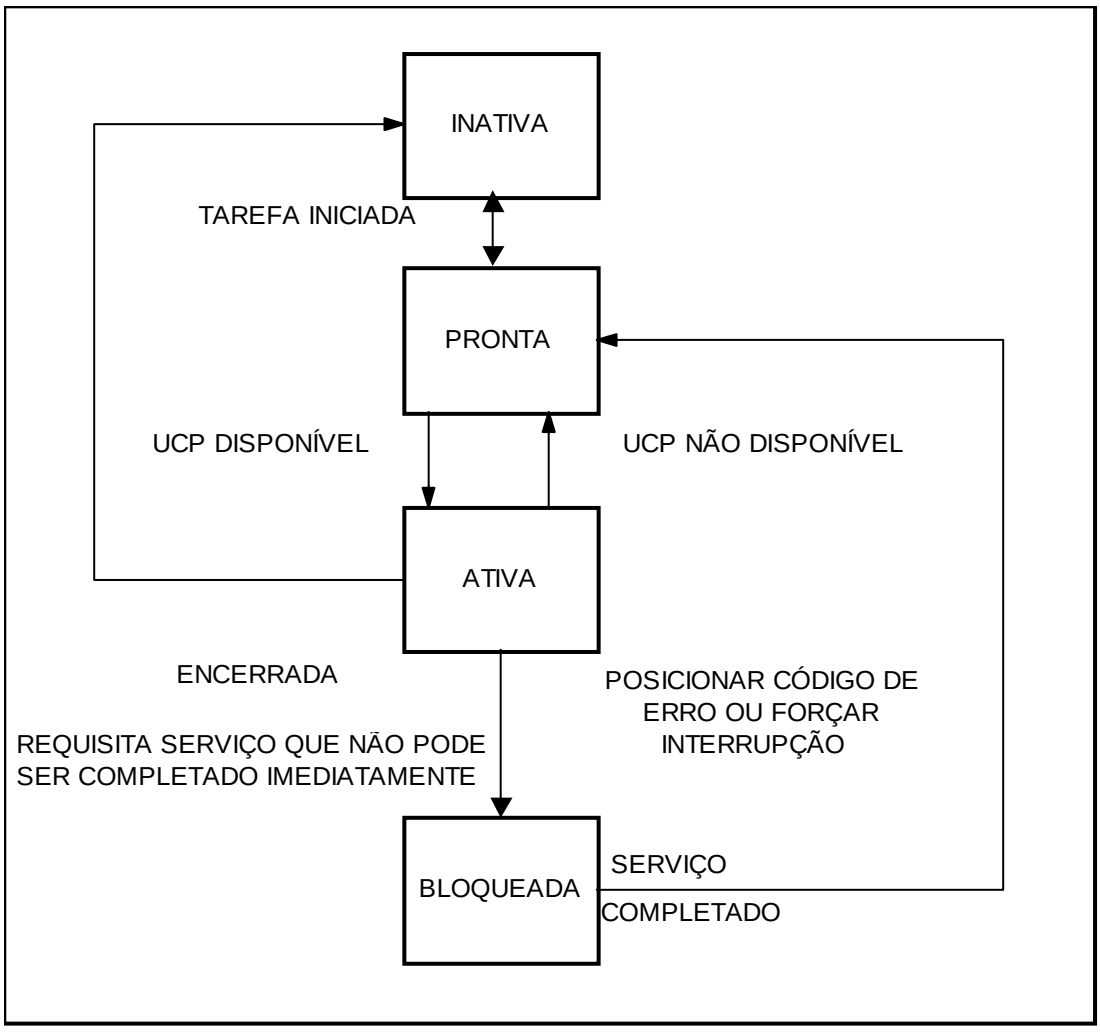
a) execução ou ativa (*running*): se ela está em execução no processador;
b) pronta (*ready*): se ela está pronta para ser executada tão logo o processador torne-se disponível. O sistema operacional é responsável por determinar a ordem pela qual as tarefas no estado de pronto devem ganhar o processador;
c) bloqueada (*blocked*): se um serviço requisitado não foi ainda atendido, ou se há

algum outro motivo que impeça sua execução, a tarefa está no estado bloqueado;

d) inativa (*dormant*): se ela terminou, ou se ainda não foi requisitada para executar.

As possíveis transições entre estes quatro estados estão representadas na figura 2.

Figura 2: Possíveis transições entre os estados de uma tarefa



Fonte: [RIP1993]

Uma tarefa pode mudar várias vezes de estado. Essas mudanças podem ser involuntárias, causadas pelo sistema operacional, ou voluntárias, causadas pela própria tarefa. Segundo [MAC1994], existem quatro mudanças de estado de uma tarefa:

a) pronto/execução: quando uma tarefa é criada, o sistema a inclui em uma lista de tarefas no estado pronto, onde espera uma chance para ser executada;

b) execução/bloqueado: uma tarefa executando passa para o estado bloqueado por

eventos gerados pela própria tarefa. Um exemplo pode ser visto quando ocorre uma requisição de entrada/saída. Neste caso, a tarefa ficará neste estado aguardando a conclusão do evento solicitado;

c) bloqueado/pronto: uma tarefa bloqueada, para a qual foi solicitada a execução, deverá passar primeiro para o estado de pronto para poder ser novamente executada;
d) execução/pronto: uma tarefa em execução entra no estado de pronto por eventos gerados pelo sistema. Um exemplo é o fim da fatia de tempo que a tarefa possui para executar. Neste caso, ela volta para o estado de pronto, onde espera uma nova fatia de tempo para poder novamente executar.

Dentro dos conceitos de sistemas operacionais, existem diferentes estratégias para a escolha da próxima tarefa a ser executada e que receberá o tempo solicitado a UCP. Isto, quando ela estiver pronta para executar. As regras mais simples são primeira-a-chegar, primeira-a-ser-servida e pedaços-de-tempo-iguais-para-todas-as-tarefas.

Embora estes métodos sejam os mais simples, nem sempre são os melhores, pois em certos momentos, determinadas tarefas são mais urgentes que outras, devendo assim receber o tempo de CPU antes. Esta situação pode ser resolvida com a atribuição de prioridades para as tarefas. Deste modo, além dos estados das tarefas deve-se observar as prioridades, obrigando que as tarefas de maior prioridade sejam executadas antes das tarefas de prioridade inferior. No caso de tarefas prontas com a mesma prioridade, a escolha pode ser feita segundo a regra primeira-a-chegar, primeira-a-ser-servida.

2.4.2 REGIÃO CRÍTICA

Quando se trata de programação concorrente (multitarefa em particular) não se pode deixar de se preocupar com o compartilhamento de recursos ([LAP1993]). Na maioria dos casos, esses recursos só podem ser usados por uma tarefa de cada vez, e, o uso do mesmo recurso não pode ser interrompido. Esses recursos são chamados *serially reusable*. Isso inclui certos periféricos, memória compartilhada e a UCP. Enquanto a UCP se protege contra o uso simultâneo por mais de um processo, outros recursos não podem fazer o mesmo ([LAP1993]). Esses recursos são chamados de região crítica. Se dois ou mais processos entram na mesma região crítica simultaneamente, certamente poderão ocorrer conflitos.

O uso simultâneo de um recurso compartilhado é chamado de colisão (*collision*). Para se evitar este problema, existem alguns mecanismos que podem ser implementados, tais como, a sincronização entre tarefas.

2.4.3 SINCRONIZAÇÃO ENTRE TAREFAS

A sincronização entre tarefas é um mecanismo que permite o controle da seqüência da execução das tarefas dentro de um programa. Existem dois tipos de sincronização necessários quando as tarefas compartilham dados:

a) sincronização de cooperação: as tarefas precisam esperar pela conclusão do processamento específico do qual sua operação correta depende;
b) sincronização de competição: as tarefas precisam esperar pela conclusão de qualquer processamento por parte delas e que ocorra atualmente em dados compartilhados específicos.

2.4.3.1 SINCRONIZAÇÃO DE COOPERAÇÃO

Segundo [SEB2000], a sincronização de cooperação entre a tarefa A e B é necessária quando a tarefa A precisa aguardar que a B conclua alguma atividade específica antes que ela prossiga sua execução.

O algoritmo produtor-consumidor é um exemplo de sincronização de cooperação, onde uma unidade do programa produz algum valor de dados ou recursos e a outra unidade o utiliza. Os dados produzidos são colocados em um *buffer* (área da memória reservada para armazenar dados enquanto estão sendo processados) de armazenamento pela unidade que produz e retirada pela unidade consumidora. A ordem ou seqüência de armazenamentos e remoções do *buffer* deve ser sincronizada. Desta maneira, se o *buffer* estiver vazio, não deve permitir que a unidade consumidora pegue dados do mesmo e caso *buffer* esteja cheio, não deve ser permitido que a unidade produtora coloque novos dados.

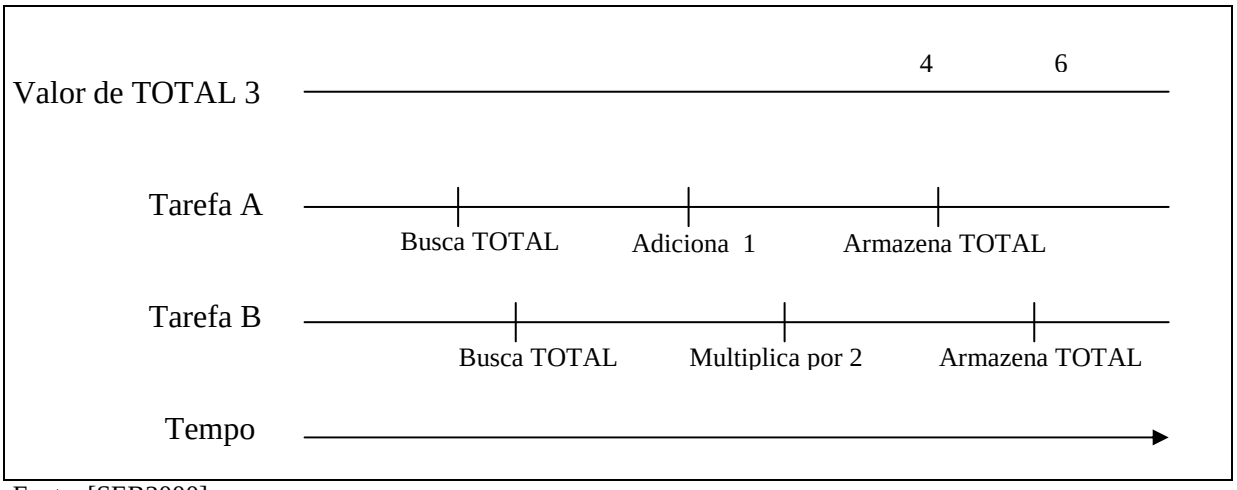
2.4.3.2 SINCRONIZAÇÃO DE COMPETIÇÃO

A sincronização de competição é necessária entre duas tarefas quando ambas requerem o uso de algum recurso que não pode ser usado simultaneamente ([SEB2000]).

A principal função da sincronização de competição é impedir que duas tarefas acessem uma estrutura de dados compartilhados exatamente ao mesmo tempo. Neste caso, é preciso garantir o acesso mutuamente exclusivo aos dados compartilhados, o que pode-se também chamar de exclusão mútua, que é a garantia de que se duas ou mais tarefas solicitarem o mesmo recurso ou dado, somente será aceita uma solicitação de cada vez.

Para um maior compreensão, segue um exemplo do uso da sincronização de competição que pode ser visto na figura 3. Neste exemplo, a tarefa A deve adicionar o valor 1 à variável compartilhada chamada TOTAL, que é inicializada com o valor 3, e a tarefa B deve multiplicar o valor total por 2. Neste caso, se A e B buscarem o valor de TOTAL antes que qualquer uma das tarefas coloque seu novo valor de volta, o resultado será incorreto. Porém se A colocar seu novo valor de volta primeiro, TOTAL será 6 e se B fizer isto, TOTAL será 4, como mostra a figura 3.

Figura 3: A necessidade de sincronização de competição



Fonte: [SEB2000]

O método geral para oferecer acesso mutuamente exclusivo a um recurso compartilhado é considerá-lo como algo que uma tarefa pode possuir e depois permitir que somente uma única tarefa o possua de cada vez. Para ganhar um recurso compartilhado, uma tarefa deve solicitá-lo. Quando uma tarefa termina o uso do recurso compartilhado que possui, ela deve liberá-lo para que o mesmo possa tornar-se disponível para outras tarefas ([SEB2000]).

Semáforos, monitores e a passagem de mensagens são exemplos de métodos que podem oferecer acesso mutuamente exclusivo a recursos compartilhados.

2.4.4 SEMÁFOROS

Uma metodologia para proteger regiões críticas foi sugerida em 1965, por Edsger Dijkstra, e envolve uma variável especial chamada semáforo e duas operações chamadas primitivas do semáforo na mesma variável ([LAP1993]). Um semáforo S, é uma região de memória que atua como uma chave para proteger as regiões críticas. Duas operações, a espera (*wait*) e o sinal (*signal*), são usados para setar ou resetar o semáforo. Tradicionalmente a operação *wait* é denotada por P(S) e a operação *signal* é denotada por V(S).

A operação *wait* serve para suspender qualquer chamada de programa até que o semáforo S seja falso. A operação *signal* serve para setar o semáforo S para falso. Processos que entram numa região crítica são apoiados pelas chamadas *wait* e *signal* ([LAP1993]). Isso serve para prevenir que mais de um processo entre na região crítica.

Segundo [SEB2000], um semáforo é uma estrutura de dados que consiste em um número inteiro e uma fila que armazena descritores de tarefas. Os descritores são constituídos de uma estrutura que armazena todas as informações relevantes sobre o estado de execução de uma tarefa.

O conceito de semáforo consiste na colocação de proteções (*guards*) no código que acessa a estrutura, a fim de oferecer o acesso limitado a um recurso compartilhado. Portanto, o semáforo é uma implementação de proteção que serve para permitir que uma tarefa acesse uma estrutura de dados compartilhados de cada vez.

2.4.5 MONITORES

De acordo com [HAN1977], Edsger Dijkstra sugeriu, em 1971, que todas as operações de sincronização em dados compartilhados fossem reunidas em uma única unidade de programa chamada de monitor.

Uma das características mais importantes dos monitores é que os dados compartilhados estão dentro deles, em vez de residirem em qualquer outra unidade do programa. Uma vez que

todos os acessos residem no monitor, a sua implementação pode ser feita de modo a garantir um acesso sincronizado, simplesmente permitindo um acesso de cada vez ([SEB2000]).

Os monitores são unidades de programa, que podem apenas ser declaradas, inicializadas e utilizadas através de suas entradas. As entradas são operações de acesso ao recurso que se quer monitorar, oferecidas pelo monitor ao uso externo. As variáveis locais do monitor podem ser acessadas apenas através de operações previstas, garantindo-se a segurança do acesso, pela implementação, que não permite, em nenhuma hipótese, mais de uma ativação simultânea de alguma entrada do monitor.

Segundo [SEB2000], a construção monitor é um método confiável e seguro para fornecer sincronização de competição para acesso a dados compartilhados em unidades concorrentes que compartilham uma única memória. Porém, em um sistema distribuído, no qual cada processador tem sua própria memória, a sincronização pode ser obtida muito naturalmente com a passagem de mensagens.

2.4.6 PASSAGEM DE MENSAGEM

A passagem de mensagem consiste em uma técnica de manipular o problema que ocorre quando vários pedidos simultâneos são feitos por outras tarefas para comunicar-se com uma em especial. Esta técnica permite que todas tarefas solicitantes recebam a mesma oportunidade de entrarem em contato com uma determinada tarefa.

A passagem de mensagens pode ser síncrona ou assíncrona ([SEB2000]). O conceito usado no modelo de passagem de mensagens síncrona é o de que as tarefas estão freqüentemente ocupadas e, assim, não podem receber mensagens de outras unidades. Se a tarefa A e B estiverem executando, e A deseja enviar uma mensagem para B, não é desejável que B pare de executar para receber a mensagem que será enviada por A. Isso significaria uma ruptura no processamento atual dela. Além disso, as mensagens normalmente causam um processamento associado no receptor, e como este ainda está incompleto, não seria a melhor opção.

A alternativa para a implementação da passagem de mensagens é fornecer um mecanismo que permita a uma tarefa estipular para outras tarefas o momento em que estará

pronta para receber mensagens.

Uma tarefa pode ser construída de modo a poder suspender sua execução em certo ponto, porque está ociosa ou porque precisa de informações de outra unidade antes que possa prosseguir. Nesta situação, se a tarefa A quiser transmitir uma mensagem para B, e se esta estiver disposta a receber, a mensagem poderá ser enviada. Esta transmissão é chamada de *rendezvous.* O *rendezvous* somente pode ocorrer se tanto o emissor como o receptor quiserem que ele aconteça. Neste caso, a informação da mensagem pode ser transmitida em qualquer ou ambas as direções.

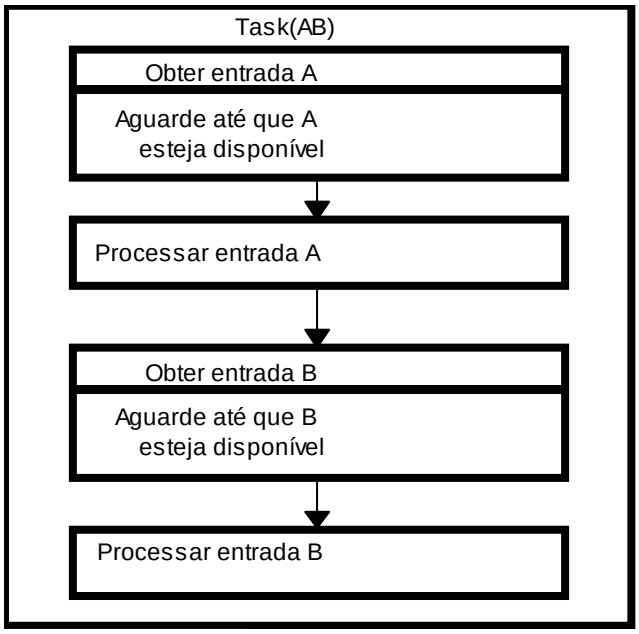
Na vida real o *rendezvous* é uma organização onde para as pessoas envolvidas, uma hora e um local predefinidos são essenciais ([KAV1992]).

O conceito de passagem de mensagens assíncrona atribui às tarefas a capacidade de enviarem mensagens a outras tarefas sem observar nenhuma seqüência. Neste caso, não há sincronismo entre o emissor e o receptor.

2.5 VANTAGENS DA CONCORRÊNCIA

Muitas vezes uma tarefa deve ficar bloqueada até que um serviço seja completado (se a tarefa necessita da entrada de um periférico, ela freqüentemente não poderá prosseguir até que a entrada esteja disponível) ([RIP1993]). A figura 4 mostra uma tarefa chamada “Task(AB)”, onde o processador não poderá obter a entrada B até que tenha encerrado a entrada A.

Figura 4: Processamento em série de entradas concorrentes



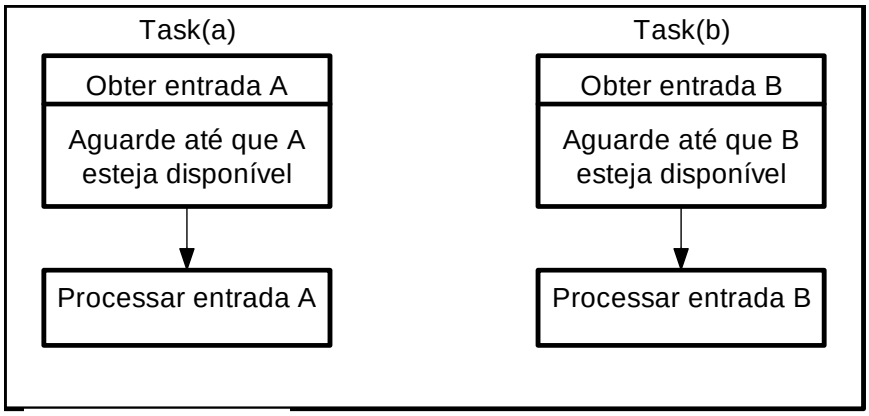
FONTE: [RIP1993]

Neste caso, não há problema se outra tarefa puder manter o processador ocupado até o término de A, mas se não houver outro trabalho a ser feito, o processador ficará ocioso.

No entanto, se o processamento de B não depende de A, pode-se observar uma organização mais produtiva como a desenvolvida na figura 5. Deste modo, fica explícita a utilização dos benefícios da concorrência para a execução das tarefas.

As vantagens da programação concorrente também podem ser obtidas através da modelagem de processos concorrentes independentes, onde os processos que não dependem de outras tarefas podem evoluir paralelamente, tendo assim um melhor desempenho ([RIP1993]).

Figura 5: Processamento em paralelo de entradas concorrentes



Fonte: [RIP1993]

3 SISTEMAS DE TEMPO REAL

A preocupação com o tempo, entre filósofos e matemáticos, vem desde a antigüidade. Uma das contribuições mais conhecida desta época foi a de Zenon cujo paradoxo sobre o tempo e o espaço levanta questões a respeito da continuidade no tempo e no espaço ([FAR2000]).

Na atualidade, pode-se dizer que tempo é dinheiro. A rapidez na tomada de decisões, nas comunicações e nas atividades em geral, tornou-se um dos paradigmas dominantes na área da informação. Utiliza-se cada vez mais a expressão “tempo real” em diversas situações, às vezes com propriedade, outras porém, apenas com objetivo comercial. De fato, o tempo está sempre presente em todas as atividades, mesmo que não seja de forma explícita. Da mesma forma, as atividades relacionadas à informática seguem esta regra.

Um número crescente de aplicações de importância no mundo atual apresenta comportamentos definidos segundo normas relacionadas ao tempo. Aplicações de controle de tráfego aéreo ou ferroviário, de telecomunicações, de robótica, de multimídia, são alguns exemplos nos quais as restrições temporais são imprescindíveis. Essas aplicações são agrupadas no que normalmente é identificado como sistemas de tempo real.

3.1 TEMPO REAL HOJE

A maior parte dos sistemas de tempo real é projetada e implementada com ferramentas convencionais de verificação e de implementação. Por exemplo, na prática corrente, são usadas linguagens de alto nível com construções não deterministas ou mesmo linguagens de baixo nível, mas sem a preocupação de tratar o tempo de uma forma mais explícita, tornando difícil a garantia da implementação das restrições temporais. Os sistemas operacionais e suportes de tempo de execução geralmente utilizados apresentam mecanismos para implementar escalonamentos dirigidos a prioridades; estas nunca refletem as restrições temporais definidas para essas aplicações. Na prática usual, a importância em termos das funcionalidades presentes nessas aplicações é determinante nas definições dessas prioridades; o que pode ser contestado, pois os possíveis graus de importância de funções em uma aplicação nem sempre se mantém na mesma ordem relativa durante todo o tempo de execução desta. Essas práticas têm permitido resolver de forma aceitável e durante muito tempo certas

classes de problemas de tempo real nas quais as exigências de garantia sobre as restrições temporais não são tão imprescindíveis ([FAR2000]).

Entretanto, as necessidades de segurança num número cada vez maior de aplicações e a ligação dessa com a correção temporal desses sistemas colocam em xeque as metodologias e ferramentas convencionais, sob pena de perdas em termos financeiros, ambiental ou humano. Essas aplicações exigem toda uma demanda de algoritmos, de suportes computacionais e de metodologias que ultrapassa as ferramentas até então utilizadas e lançam de certa forma novos desafios para os programadores desse tipo de sistemas ([FAR2000]).

Apesar da evolução nos últimos anos, em termos de conceitos e métodos, para tratar a problemática de sistemas de tempo real, a adoção no setor produtivo, desses novos algoritmos, suportes e metodologias não se dá no mesmo ritmo. Na prática, o uso de meios mais convencionais continua, mesmo no caso de aplicações críticas, o que pode ser a causa de muitas situações desastrosas.

3.2 CONCEITOS BÁSICOS

Primeiramente, se faz necessário, diferenciar Sistemas de Tempo Real (STR) de Sistemas Não de Tempo Real. STR são sistemas que exigem que as respostas sejam válidas e que estejam dentro de prazos impostos pelo ambiente. Os resultados devem estar corretos lógica e temporalmente. Já os Sistemas Não de Tempo Real são aqueles que esperam respostas válidas em prazos aceitáveis, não especificados. Portanto, STR são sistemas que fazem o trabalho usando o tempo disponível e sistemas em geral, fazem o trabalho usando o tempo necessário.

“Um Sistema de Tempo Real é um sistema que produz reações a estímulos oriundos do ambiente dentro de intervalos de tempos impostos pelo ambiente (é incluído entre estes estímulos o passar do tempo físico)” ([FAR2000]).

A grande diferença entre programas em tempo real e não em tempo real, vem da natureza das especificações do programa, isto é, do que os programas devem fazer. Algumas vezes a diferença vem justamente das escalas relativas de tempo envolvidas na solução do problema.

A necessidade de interromper uma parte de uma aplicação em tempo real espontaneamente, para realizar captura de dados ou outras funções mais urgentes, forçam uma organização particular para esses tipos de programas. Aplicações em tempo real devem ser escritas como uma série de programas de componentes separados que podem ser executados concorrentemente.

Na busca do entendimento de STR surgem algumas concepções erradas, tais como:

a) tempo real significa execução rápida;
b) computadores mais rápidos vão resolver todos os problemas;
c) STR são sistemas pequenos, escritos em *assembly*;
d) STR são sistemas formados apenas por tratadores de interrupções;
e) não existem problemas específicos da área de tempo real;
f) STR operam em ambientes estáticos.

Em [SON1995], são apresentadas algumas questões importantes e que devem ser levadas em consideração para o sucesso de um sistema de tempo real. São elas:

a) métodos formais para especificar e verificar as exigências destes sistemas;
b) gerenciamento dos recursos para assegurar que as exigências temporais sejam cumpridas;
c) linguagens de programação e ferramentas para suportar um poderoso software de desenvolvimento de processos;
d) comunicação em tempo real para suportar o tráfego de mensagens em tempo real satisfazendo as restrições individuais de tempo;
e) tolerância à falhas para assegurar uma reabilitação adequada no melhor tempo possível.

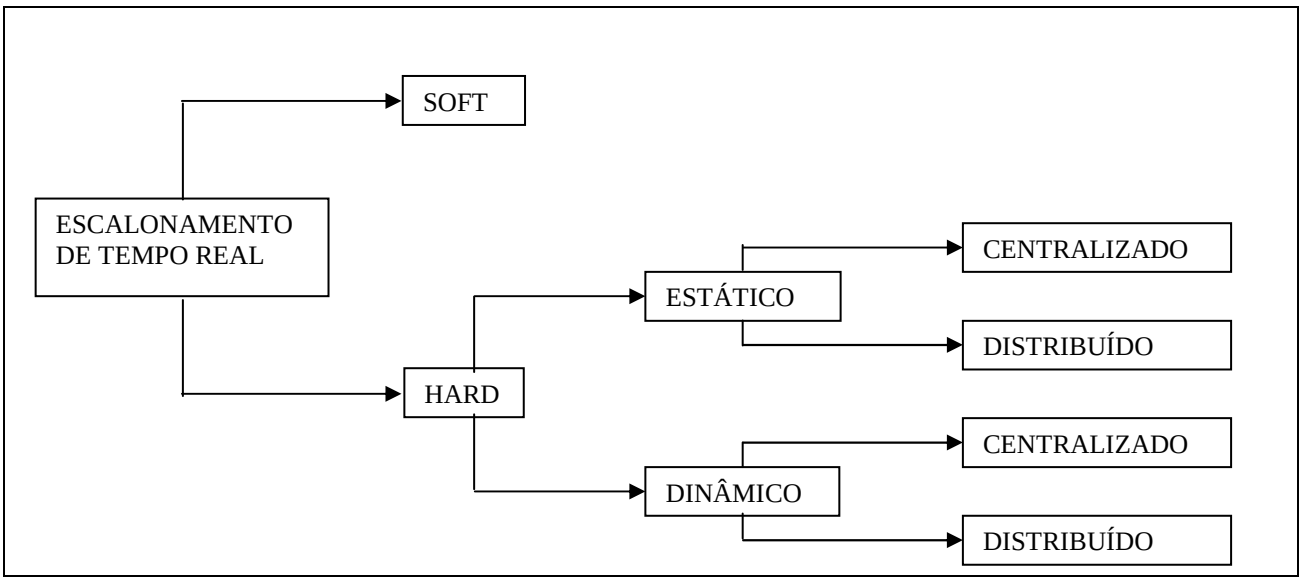
É extremamente necessário saber que os STR´s têm conceitos muito mais profundos e que precisam ser vistos para que a sua compreensão atinja um nível desejado. Conceitos como tempo real crítico e não crítico, previsibilidade, tarefas, escalonamento de tarefas serão apresentados a seguir para um maior entendimento dos STR’s.

3.3 CRÍTICO X NÃO CRÍTICO

Segundo [MAG1990] e [FAR2000], os sistemas de tempo real podem ser divididos

em duas categorias: *soft real-time* e *hard real-time*. Ainda que os sistemas *hard real-time* podem ser subdivididos em estático e dinâmico segundo seu escalonamento, como mostra a figura 6.

Figura 6: Classificações dos sistemas de tempo real



Tempo real não crítico (*soft real-time*) são representados por sistemas onde o requisito temporal descreve apenas um comportamento desejado, e não imprescindivelmente necessário. Um exemplo pode ser visto em um sistema de processamento bancário.

Por outro lado, tempo real crítico (*hard real-time*) são definidos por sistemas nos quais a falha temporal pode resultar em conseqüências catastróficas. Nestes casos, é necessário garantir os requisitos temporais no próprio projeto. Exemplos podem ser encontrados em sistemas para controle de cruzamento ferroviário, controle de mísseis, controle aéreos, etc. Nos sistemas *hard real-time*, o escalonamento pode ser:

a) estático: se as prioridades são atribuídas às tarefas uma única vez em modo *off-line* e não são mais alteradas durante a execução;
b) dinâmico: as prioridades das tarefas são determinadas em tempo de execução, podendo ser alteradas de acordo com a evolução dos sistemas.

Tanto o escalonamento estático quanto o dinâmico ainda podem ser divididos em:

a) centralizado: os processadores estão localizados em um único ponto do sistema e o

custo de comunicação é insignificante. Um exemplo é encontrado nos sistemas monoprocessadores;

b) distribuído: os processadores estão situados em diferentes pontos do sistema e o custo de comunicação entre eles deve ser considerado. Um exemplo são os computadores interligados através de redes locais.

3.4 PREVISIBILIDADE

Uma das crenças mais comuns é que o problema de tempo real é resolvido pelo aumento da velocidade computacional. A rapidez de um cálculo visa melhorar o desempenho de um sistema computacional, minimizando o tempo de resposta médio de um conjunto de tarefas, enquanto que o objetivo de um cálculo em tempo real é o atendimento dos requisitos temporais de cada uma das atividades de processamento caracterizadas nesses sistemas ([STA1988]).

A obtenção de um tempo curto de resposta não é a garantia de que os requisitos temporais de cada processamento no sistema serão atendidos. O mais importante para os sistemas de tempo real é o conceito de previsibilidade.

Conforme [FAR2000], um sistema de tempo real é dito previsível no domínio lógico e no domínio temporal quando, independentemente de variações de hardware (desvios do relógio), da carga e de falhas, o comportamento do sistema pode ser antecipado, antes mesmo de sua execução.

Para se prever a execução de um STR e garantir o cumprimento das restrições temporais, é preciso definir um conjunto de hipóteses sobre o comportamento do ambiente externo no que diz respeito à carga e às falhas:

a) hipótese de carga: esta hipótese determina o que corresponde a carga computacional máxima gerada pelo ambiente em um intervalo de tempo, entre cada reação do sistema de tempo real;

b) hipótese de falhas: esta hipótese descreve os tipos e freqüências de falhas com os quais o sistema deve interagir quando está em execução, continuando a atender os requisitos funcionais e temporais.

Além das hipóteses da carga e das falhas, a garantia da previsibilidade depende ainda de um conjunto de fatores ligados a arquitetura de hardware, ao sistema operacional e as linguagens de programação são também importantes. Ou seja, os tempos gastos, no pior caso, na execução de códigos de aplicação precisam ser conhecidos, o que, na realidade, não é uma tarefa simples.

A abordagem determinística de previsibilidade acima exposta está associada a uma antecipação determinista do comportamento temporal do sistema, onde se pode antecipar que todos os prazos colocados a partir das interações com seu ambiente serão atendidos. Porém, existe também uma abordagem probabilista, onde o conceito de previsibilidade está relacionado a uma antecipação probabilista do comportamento do sistema, baseada em estimativas ou simulações que estipulam probabilidades dos prazos a serem atendidos, o que é muito útil em sistemas onde a carga computacional não pode ser conhecida profundamente.

A previsibilidade, portanto, é um requisito necessário que deve ser incluído em sistemas de tempo real, a fim de que as restrições temporais possam ser cumpridas.

3.5 SISTEMAS OPERACIONAIS EM TEMPO REAL

Em geral, aplicações são construídas a partir dos serviços oferecidos por um sistema operacional. No caso de aplicações de tempo real, o atendimento dos requisitos temporais depende não somente do código da aplicação, mas também da colaboração do sistema operacional no sentido de permitir previsibilidade ou pelo menos um desempenho satisfatório ([FAR2000]).

Assim como aplicações convencionais, aplicações de tempo real são mais facilmente construídas se puderem utilizar os serviços de um sistema operacional. Desta forma, o programador não precisa preocupar-se com a gerência dos recursos básicos (processador, memória física), podendo usar as abstrações de mais alto nível criadas pelo sistema operacional (tarefas, segmentos).

3.6 TAREFA EM TEMPO REAL

Tarefa é o segmento de código cuja execução possui atributo temporal próprio. O instante máximo desejado para a conclusão de uma tarefa é chamado de *deadline*.

Cada tarefa é um programa completo que é capaz de ser executado de forma independente. Cada uma tem um segmento de código que ela executa, bem como sua própria pilha (*stack*) particular e suas próprias áreas locais de dados, isto é, as áreas onde a tarefa pode armazenar parâmetros de chamada de procedimentos, retornar endereços, dados temporários e variáveis que não são compartilhadas com outras tarefas. Além disso, cada tarefa tem seu próprio conjunto de valores para o contador do programa e ponteiro da pilha.

A previsibilidade está associada à capacidade de poder antecipar, em tempo de projeto, verificando se os processamentos em um STR serão executados dentro de seus prazos especificados. Caso esta previsibilidade esteja associada a uma previsão determinista, o sistema exigirá que todos os *deadlines* sejam respeitados. Porém, se ela estiver associada a uma antecipação probabilista baseada em estimativas, as probabilidades são relacionadas com os *deadlines*, definindo as possibilidades dos mesmos serem respeitados.

3.7 ESCALONAMENTO DE TAREFAS

Em sistemas onde as noções de tempo e de concorrência são tratadas explicitamente, conceitos e técnicas de escalonamento formam o ponto central na previsibilidade do comportamento de sistemas de tempo real ([FAR2000]).

O objetivo da multi-programação é ter várias tarefas executando ao mesmo tempo, para maximizar a utilização da CPU. O objetivo da divisão do tempo é que possibilite o uso da CPU pelas diversas tarefas, enquanto isto, o usuário poderá interagir com cada programa, durante sua execução. Em um sistema monoprocessamento, as tarefas somente poderão ser executadas separadamente até o final de sua execução. A divisão de tempo na CPU utilizada por cada tarefa é função de um mecanismo chamado escalonador.

O termo escalonamento (*scheduling*) identifica o procedimento de ordenar tarefas na fila do estado pronto. Uma escala de execução é então uma ordenação ou lista que indica a ordem de ocupação da CPU por um conjunto de tarefas disponíveis na fila de pronto. O escalonador é o componente do sistema operacional responsável em tempo de execução pela administração da CPU.

Segundo [FAR2000], é o escalonador que implementa uma política de escalonamento

ao ordenar para execução sobre o processador um conjunto de tarefas.

3.7.1 NÍVEIS DE ESCALONAMENTO

O escalonamento pode ser apresentado através de níveis. Para [DEI1984], existem três níveis importantes de escalonamento. São eles:

a) escalonamento de alto nível (*high-level scheduling*): este tipo de nível de escalonamento também pode ser chamado de escalonamento de tarefas (*job scheduling*). Este escalonamento determina qual tarefa foi aprovada para competir pelos recursos do sistema. Escalonamento por admissão (*admission scheduling*) é outro nome dado para este escalonamento, porque ele determina a tarefa que ganhará a permissão do sistema para ser executado;
b) escalonamento de nível intermediário (*intermediate-level scheduling*): este escalonamento determina qual tarefa deve ser liberada para competir pelo processador. Este nível é responsável pelas flutuações na carga do sistema, podendo aliviar sua operação, suspendendo, ativando ou retornando tarefas para a execução, alcançando objetivos de desempenho geral do sistema. Desta maneira, o escalonamento atua como um *buffer* entre a admissão da tarefa e a atribuição do processador para esta tarefa;
c) escalonamento de baixo nível (*low-level scheduling*): este escalonamento determina qual tarefa está pronta para ser atribuída ao processador quando o mesmo torna-se disponível.

3.8 ABORDAGENS DE TEMPO REAL

O problema tempo real consiste em especificar, verificar e implementar sistemas ou programas que, mesmo com recursos limitados, apresentam comportamentos previsíveis, atendendo as restrições temporais impostas pelo ambiente ou pelo usuário ([FAR2000]). Considerando esses aspectos de construção, tempo real pode ser visto inicialmente como um problema de programação concorrente. Então baseado no modo de tratar a concorrência surgem duas abordagens distintas: a abordagem assíncrona e a abordagem síncrona.

3.8.1 ABORDAGEM ASSÍNCRONA

A abordagem assíncrona trata a ocorrência e a percepção de eventos de maneira independente, numa ordem arbitrária, porém não simultânea.

Esta abordagem tenta descrever o sistema do modo mais exato possível. Isto inclui uma série de detalhes de hardware e de software, como modelo de tarefas, conceitos de escalonamento, suporte de sistema operacional, etc.

Para a obtenção de uma descrição detalhada do sistema, esta abordagem baseia-se na observação da ocorrência dos eventos de forma não simultânea, conhecida como entrelaçamento de eventos (*interleaving*), sendo assim considerada uma abordagem orientada à implementação. Conseqüentemente, se torna necessário observar na especificação e no decorrer do projeto, certas características de software e hardware. Por outro lado, a busca de uma descrição completa do comportamento da implementação torna complexa a análise das propriedades do sistema, devido à necessidade do tratamento de um grande número de estados.

A abordagem assíncrona é fundamentada no tratamento explícito da concorrência e do tempo de uma aplicação em tempo de execução, sendo o escalonamento a questão principal da previsibilidade dos sistemas de tempo real.

3.8.2 ABORDAGEM SÍNCRONA

A abordagem síncrona tem o seu princípio básico que os cálculos e as comunicações não levam tempo ([FAR2000]), isto é, o tempo de execução dos cálculos e das comunicações pode ser desconsiderado, pois não atrapalha a performance do sistema. Ao contrário da abordagem assíncrona, onde estes detalhes têm muita importância.

Nesta abordagem, é preciso levar em consideração dois aspectos:

a) a descrição do comportamento do sistema é menos dependente das questões de implementação, sendo que a observação dos eventos é feita de forma cronológica, permitindo uma possível simultaneidade entre eles;
b) o tempo é colocado em um nível de abstração dentro da implementação, não influenciando na execução da aplicação.

Na abordagem síncrona, a única preocupação do programador da aplicação é garantir a resposta correta aos eventos do ambiente, no sentido lógico. A correção no sentido temporal está automaticamente garantida, pois, como a velocidade de processamento é considerada infinita, o tempo de resposta será sempre zero ([FAR2000]). Portanto, a premissa básica desta abordagem é a simultaneidade dos eventos de entrada e saída, levando em conta que cálculos e comunicações não levam tempo, pois a máquina é sempre considerada muito rápida.

Considerando estes aspectos, fica claro que nesta abordagem a concorrência é resolvida sem o entrelaçamento de tarefas e o tempo é tratado de maneira implícita, tornando esta abordagem orientada para o comportamento de uma aplicação e sua verificação, facilitando a especificação e análise das propriedades dos sistemas de tempo real.

Segundo [FAR2000], o uso da abordagem síncrona é uma solução recomendada para os seguintes sistemas: controle de processos de tempo real, sistemas de manufatura, sistemas de transportes, sistemas embutidos, sistemas autônomos, sistemas de supervisão, protocolos de comunicação, entre outros. Entretanto, aplicações complexas que envolvem programas de cálculo, grande quantidade de dados a manusear e distribuição serão mais facilmente solucionados através do uso da abordagem assíncrona.

4 A LINGUAGEM ERLANG

Segundo [CAS1998], em 1984, a Ericsson iniciou uma série de experiências usando um grupo de linguagens e de técnicos de programação com o objetivo de identificar as qualidades requeridas em um ambiente de desenvolvimento de software para aplicações de telecomunicações. As experiências incluíram linguagens imperativas, funcionais e lógicas. A conclusão das experiências foi a não existência de nenhuma linguagem com todas as características necessárias para este tipo complexo de aplicação.

A linguagem Erlang foi projetada para satisfazer exigências e características de um ambiente de telecomunicações. Características como, *soft real time*, restrições temporais, suporte a programação concorrente, e recursos para atualização de código sem parar o sistema. Quando um cliente faz uma ligação telefônica, ele espera a resposta com o menor período de tempo possível. Para assegurar que este serviço seja prestado ao cliente dentro de suas expectativas, é necessário para o programador poder especificar ações para que isto aconteça dentro de um determinado tempo e programar reações se estas ações não fizerem o esperado.

4.1 O AMBIENTE DE DESENVOLVIMENTO ERLANG

O código Erlang é compilado em um código de *byte* que é interpretado. Isto permite que o código seja facilmente transportado entre sistemas. O Erlang provê um ambiente adicional que permite ao programador interagir diretamente com as funções no ambiente de desenvolvimento. Esta interação é uma característica poderosa e muito útil do Erlang.

4.1.1 O *SHELL* DO ERLANG

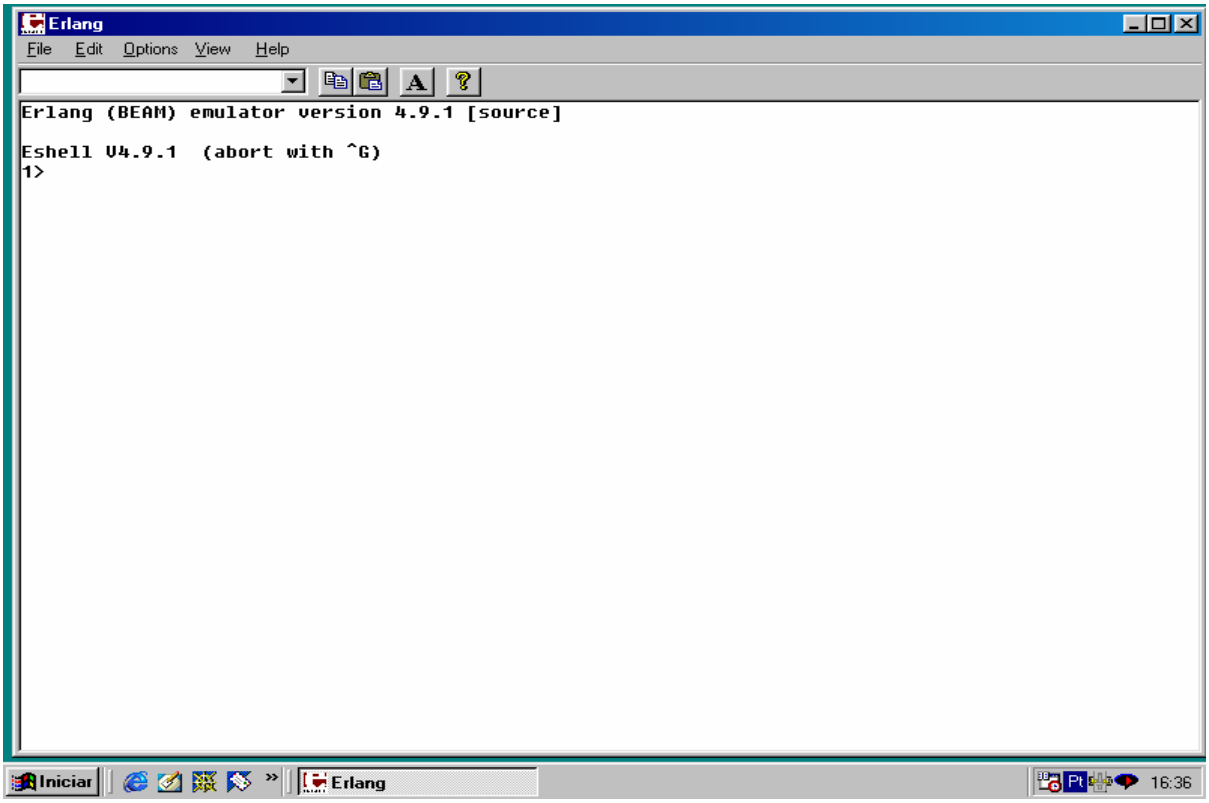
O *shell* interpreta fragmentos de código Erlang inseridos pelo usuário, como mostra a figura 7. O *shell* permite que variáveis sejam declaradas e funções sejam chamadas somente como ações inseridas em programa Erlang. A diferença entre o *shell* e o programa é que o *shell* não permite que o usuário defina suas próprias funções.

Os comandos internos só podem ser acessados digitando o nome de função com seus argumentos entre parêntesis. Os módulos podem ser acessados prefaciando o nome de função

com o nome de módulo digitado em um editor, salvo em um diretório com a extensão *erl*.

Na figura 7 pode-se observar o ambiente Erlang onde são executados e interpretados os códigos escritos nesta linguagem.

Figura 7: Ambiente Erlang



Programas em Erlang são compostos de funções que se agrupam em módulos. Estas funções criam processos que são os elementos executáveis de um sistema Erlang. Os processos se comunicam entre si através da passagem de mensagens. Um mecanismo de distribuição embutido permite aos programadores a criação de um sistema cujos processos podem ser executados concorrentemente ([ARM1993]).

4.2 TIPOS DE DADOS

Nomes, verbos e adjetivos em nossa língua são conjuntos de palavras que desempenham papéis particulares em nossa comunicação, e podem ser usados em determinados contextos. Os tipos e subtipos de palavras e arranjos de palavras em um idioma são às vezes também chamados de partes de uma linguagem. Da mesma forma, nas

linguagens de programação também existem partes, representando os seus componentes. Em particular, os dados de um programa são divididos em um número diferente de tipos.

Erlang suporta cinco tipos simples de dados:

a) *integer*: um número inteiro positivo ou negativo;
b) *float*: um número com parte fracionária;
c) *atom*: um nome constante;
d) *pid*: um identificador de processos;
e) *reference*: um único valor que pode ser copiado ou passado mas não pode ser gerado novamente.

Também suporta dois tipos de dados compostos:

a) *tuple*: um conjunto de elementos de tamanho fixo;
b) *list*: um conjunto de elementos de tamanho variável.

Um *term* é um tipo de dado que pode receber o valor de quaisquer dos tipos de dados acima citados.

Os tipos de dados *floats* e *integers* podem ser combinados em expressões aritméticas. O quadro 1 mostra as operações que podem ser usadas.

Quadro 1: Operações aritméticas

Opção	Descrição
+ X	Soma X
- X	Diminui X
X * Y	Multiplica X por Y
X / Y	X dividido por Y
X div Y	X dividido por Y (valor inteiro da divisão)
X rem Y	Resto inteiro de X dividido por Y
X band Y	Bitwise and of X and Y
X + Y	Soma X mais Y
X – Y	Diminui Y de X
X bor Y	Bitwise or of X and Y
X bxor Y	Bitwise xor of X and Y
X bsl Y	Arithmetic shift left of X by Y bits
X bxr Y	Shift right of X by Y bits

Fonte: [CAS1998]

Um *atom* é uma constante de nomes. Seu valor é o nome que lhe foi atribuído. Dois *atoms* são equivalentes quando são escritos igualmente. São exemplos de *atoms*: ‘Pedro’, ‘este é um nome’, etc. Os *atoms* também podem ser seguidos de caracteres especiais, como mostra o quadro 2.

O ambiente de programação Erlang foi projetado para executar processos concorrentes. Cada programa é executado independentemente de outros programas: parâmetros e memória não são compartilhados entre os programas. Isto faz com que cada *thread* de execução, no ambiente de desenvolvimento Erlang, seja um processo. O identificador de processos (*Pid*) é a única ligação entre os processos. *Pids* são usados para a comunicação entre processos e para identificá-los no ambiente de desenvolvimento, permitindo operações como a criação, destruição e troca da seqüência de prioridades dos processos.

Quadro 2: Convenções para uso dos caracteres

Caracter	Significado
\b	Retrocesso
\d	Delete
\e	Esc
\f	Nova folha
\n	Nova linha
\r	Enter
\t	Tab
\v	Tab vertical
\\	Retorno
\^A ... \^Z	Control A (0) até control Z (26)
\”	Aspas
\000	Caracter com valor octal 000

Fonte: [CAS1998]

Algumas operações são impossíveis de serem programadas em Erlang, ou são impossíveis de serem programadas com eficiência. Por exemplo, não existe um modo de encontrar a estrutura interna de um *atom*, ou da hora do dia, etc. Entretanto, Erlang possui um número de *built-in functios* (BIFs) que executam estas operações. A BIF *date()* retorna a data do dia em que é executada. Como esta, existe uma série de BIFs que já estão prontas dentro do ambiente Erlang.

O Erlang também provê uma função que devolve, em seu ambiente, um valor único

chamado *reference*. Deste modo, não podem ser geradas duas *reference* idênticas que irão ocasionar um erro no programa.

References são objetos únicos. A BIF *make-ref()* retorna um objeto único garantindo que este será diferente de todos os outros objetos no sistema. No quadro 3 é apresentada a estrutura de uma *reference* através da BIF *make-ref()*.

Quadro 3: Estrutura *Reference*

```
Request (Server, Req) ->
        Server ! {R = make_ref(), self(), Req},
                receive
                        {Server, R, Reply} ->
                                Reply
                end.
```

Fonte: [ARM1993]

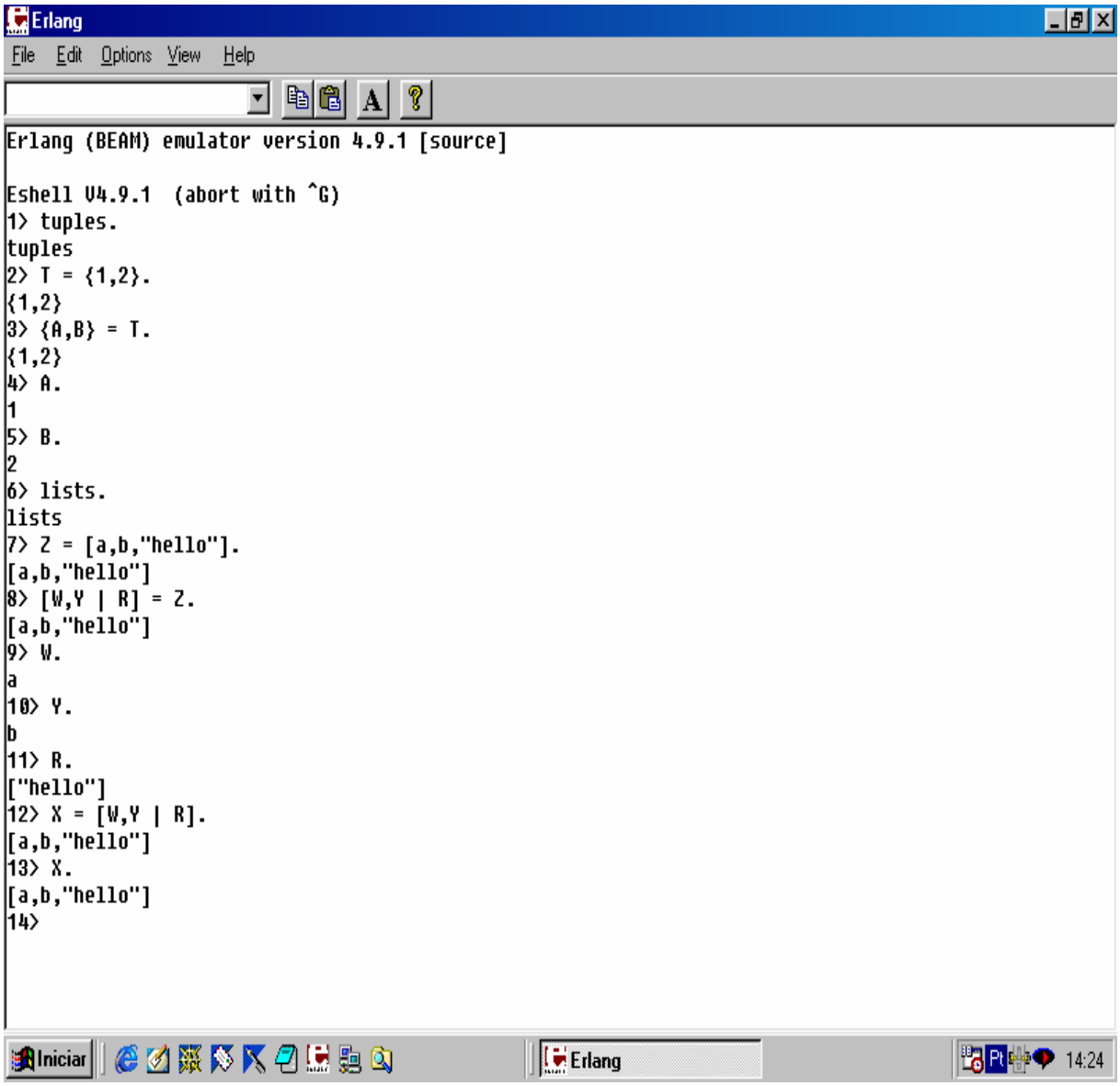
No quadro 3, *request (Server, Req)* envia uma solicitação *Req* para o servidor com o nome *Server*. O pedido contêm uma única *reference R*. A resposta do servidor é checada para assegurar a presença de uma única *reference R*. Este método de comunicação com o servidor fornece uma confirmação que a solicitação foi bem sucedida.

Tuples são estruturas de dados usadas para armazenar um número fixo de elementos separados por vírgulas dentro de chaves, como os exemplos {1, 2, 3} e {a, {1, 2, 3}}. Os elementos são identificados pela sua posição e podem ser extraídos usando padrão emparelhado (*pattern matching*), conforme a figura 8.

O tamanho de um *tuple* é equivalente ao número de elementos existentes no *tuple*.

A estrutura de dados *list* não tem um tamanho predeterminado. O Erlang define um número de operações e funções, permitindo a criação de uma nova estrutura *list* a partir de uma já existente, contendo mais ou menos elementos que a original. Os elementos, nesta estrutura, são separados por vírgulas dentro de colchetes, como mostram os exemplos: [1,2,3], [Pedro,20,3], [{a,b}, {a,b,c}], [].

Figura 8: Manipulações de *tuples* e *lists* no ambiente Erlang



Erlang tem uma notação especial para gerar facilmente listas de caracteres. Uma *string* de caracteres incluída entre aspas é convertida em um *list* de inteiros representado por caracteres, como mostra a figura 8.

O separador vertical (|) é usado na notação de uma estrutura *list* para separar a parte *head* do *tail* de uma *list*. Alguns exemplos deste modelo de notação aparecem na figura 8.

Segundo [CAS1998], a maioria das funções Erlang são escritas para manipular e retornar *lists* próprias ou bem constituídas. Estas *lists* têm um *list* vazio ([]) como seu último

elemento. Algumas funções úteis que operam em listas são encontradas no quadro 4.

Quadro 4: Funções das *lists*

Funções	Descrição
atom_to_list(X)	Retorna a lista dos caracteres ASCII que formam o atom X
float_to_list(X)	Retorna a lista dos caracteres ASCII que representam o float X
integer_to_list(X)	Retorna a lista dos caracteres ASCII que representam o valor do inteiro X
list_to_atom(X)	Retorna um atom composto pelos caracteres da lista ASCII de X
list_to_float(X)	Retorna um float composto pelos caracteres da lista ASCII de X
list_to_integer(X)	Retorna um inteiro composto pelos caracteres da lista ASCII de X
hd (L)	o head – primeiro elemento - da lista L
tl (L)	o tail – último elemento - da lista L
lenght (L)	o número de elementos da lista L

Fonte: [CAS1998]

Conforme [CAS1998], as variáveis do Erlang se comportam de maneira diferenciada de variáveis de linguagens como C, Ada e Java. Erlang têm as seguintes propriedades:

a) o escopo (região de um programa no qual uma variável pode ser acessada) de uma variável estende o seu primeiro aparecimento em uma cláusula através da cláusula final de uma função Erlang;
b) o conteúdo de uma variável Erlang persiste da atribuição até o final da cláusula;
c) as variáveis Erlang podem ser atribuídas somente uma vez;
d) ocorre um erro quando é acessada uma variável Erlang não relacionada;
e) as variáveis Erlang não são tipos. Qualquer termo pode ser relacionado a uma variável.

A propriedade de permitir a uma variável ser relacionada apenas uma vez é conhecida como atribuição simples.

4.3 PRINCIPAIS MECANISMOS

A linguagem de programação Erlang tem alguns mecanismos básicos que precisam ser entendidos com clareza. Este entendimento permitirá ao programador o desenvolvimento do

código Erlang através de suas principais caraterísticas. Mecanismos como *pattern matching*, *guards*, *module*, *garbagge collection* são apresentados a seguir.

4.3.1 PADRÃO EMPARELHADO (*PATTERN MATCHING*)

Os padrões têm a mesma estrutura dos outros tipos de dados, podendo atribuir valores às variáveis. As variáveis são iniciadas com letras maiúsculas. No quadro 5 são apresentados exemplos de padrões, onde A, B, X_1 e Var_um são variáveis.

Quadro 5: Estrutura do *pattern matching*

```
{A, a, 12, [12,34|{a}]}
{A, B, 23}
{x, {X_1}, 12, Var_um}
[]
```

Fonte: [CAS1998]

O padrão emparelhado provê um mecanismo básico que atribui valores para as variáveis. A variável cujo valor foi atribuído é dita ligada (*bound*). Caso contrário é não ligada (*unbound*). O ato de nomear um valor para uma variável é chamado ligação (*binding*) e o valor de uma variável que for ligada, não poderá ser alterado.

Quando um padrão e um termo tiverem a mesma estrutura então eles poderão ser emparelhados. Sempre que um tipo de dado é encontrado no padrão, o mesmo tipo de dado é encontrado na posição correspondente do termo. No caso onde o padrão contém uma variável não ligada, a variável é ligada ao elemento correspondente no termo.

O padrão emparelhado acontece:

a) quando é avaliada uma expressão como A = B;
b) quando uma função é chamada;
c) quando ocorre o casamento de um padrão nas primitivas *case* ou *receive*.

4.3.1.1 PADRÃO = EXPRESSÃO

Quando o padrão é igualado com uma expressão acontecerá a avaliação da expressão e

o resultará no casamento contra o padrão. Este casamento pode tanto ocorrer com sucesso como falhar. Se o casamento tiver sucesso, é executada qualquer variável que esteja ligada no padrão. Um exemplo pode ser visto no quadro 6.

Quadro 6: *Pattern Matching* = Expressão

```
[{pessoa, nome, idade, _} | T] =
      [{pessoa, Cesar, 27, masculino},
       {pessoa, Katia, 26, feminino}, ...]
```

No exemplo do quadro 6, acontecem as seguintes ligações:

a) T é ligado com [{pessoa, Katia, 26, feminino}, ...]};

b) nome é casado com César;

c) idade é ligada com 27.

No exemplo do quadro 6 também é usada uma variável anônima escrita em "_" (variáveis anônimas são usadas quando a sintaxe requer uma variável, da qual não é preciso saber o valor).

4.3.1.2 *PATTERN MATCHING* QUANDO UMA FUNÇÃO É CHAMADA

O *pattern matching* pode ocorrer também quando há a chamada de uma função.

O programa do quadro 7 define uma função chamada classifica_dias/1, que retorna "fim de semana" se a função for chamada com o argumento sábado ou domingo, ou retorna "dia de semana" quando a função for chamada com qualquer outro argumento (dia). Portanto, quando uma função é executada, os argumentos da função são emparelhados com os padrões passados na definição da função.

Quadro 7: Chamada de função do *pattern matchig*

```
-module(datas).
-export([classifica_dias/1]).

classifica_dias(sábado)   -> "fim de semana";
classifica_dias(domingo)  -> "fim de semana";
classifica_dias(_)        -> "dia de semana";
```

O casamento de um padrão nas primitivas *case* ou *receive* serão vistos posteriormente.

4.3.2 MODULE

O sistema de módulo é usado na linguagem Erlang para permitir que um programa grande seja dividido em conjunto de módulos. Cada módulo tem seu próprio espaço de nome, o que possibilita o uso livre de nomes iguais de função em diferente módulos, sem qualquer confusão ([ARM1993]).

O sistema de módulos trabalha limitando a visibilidade das funções contidas dentro de um determinado módulo. O modo pelo qual uma função pode ser chamada depende do nome do módulo, do nome da função e se o nome de função acontece em uma declaração de importação (*import*) ou de exportação (*export*) no módulo. O quadro 8 apresenta a estrutura *module*.

Quadro 8: Estrutura de um módulo

```
-module(lists1).
-export([reverse/1]).

reverse(L) ->
    reverse(L, []).

reverse([H | T], L) ->
    reverse(T, [H | L];
reverse([], L) ->
    L.
```

Fonte: [ARM1993]

O programa do quadro 8 define uma função *reverse*/1 que mostrará a ordem contrária dos elementos de uma lista. *Reverse*/1 é a única função que pode ser chamada fora do módulo. As únicas funções que podem ser chamadas de fora de um módulo devem estar contidas nas declarações de exportação para o módulo.

Outra função no módulo, *reverse*/2, está disponível para uso somente dentro do módulo. As funções *reverse*/1 e *reverse*/2 são completamente diferentes. Em Erlang, duas funções com o mesmo nome, porém com número diferente de argumento, são funções totalmente diferentes.

Há dois métodos para chamar funções em outro módulo. No quadro 9 a função *reverse*/1 foi chamada usando completamente a função qualificada com o nome *lists1:reverse(L)* na chamada. Outro método de chamar funções, é apresentado no quadro 10,

onde implicitamente o nome qualificado da função é atribuído através da declaração *import*.

Quadro 9: Modelo de chamadas nos módulos

```
-module(sort1).
-export([reverse_sort/1, sort_1]).

reverse_sort(L) ->
    lists1:reverse(sort(L)).

sort(L) ->
    lists:sort(L).
```

Fonte: [ARM1993]

Quadro 10: Chamada nos módulos usando a primitiva *import*

```
-module(sort2).
-import(list1, [reverse/1]).

-export([reverse_sort/1, sort_1]).

reverse_sort(L) ->
    reverse(sort(L)).

sort(L) ->
     lists:sort(L).
```

Fonte: [ARM1993]

Segundo [ARM1993], o uso de ambas as formas acima apresentadas é necessário para solucionar ambigüidades. Por exemplo, quando dois diferentes módulos exportam a mesma função, explicitamente qualificam as funções com o nome que deve ser usado.

Outra característica importante dos módulos em Erlang, conforme [CAS1998], é a possibilidade de alterar e substituir o código de um programa enquanto o sistema está rodando. Isto acontece quando um módulo inteiro é importado no sistema para substituir um módulo que já foi anteriormente carregado.

4.3.3 CLÁUSULAS

Cada função em Erlang é construída de várias cláusulas. As cláusulas são separadas por ponto-e-vírgula “;”. Cada cláusula individual consiste de três partes:

a) uma cabeça de cláusula;

b) um *guard* opcional;

c) um corpo.

4.3.3.1 CABEÇA DAS CLÁUSULAS

A cabeça de uma cláusula consiste em um nome de função seguido por vários argumentos separado por vírgulas. Cada argumento é um padrão válido.

Quando uma chamada de função é feita, a chamada é seqüencialmente emparelhada com o conjunto das cabeças da cláusula que define a função.

4.3.3.2 CLÁUSULA *GUARD*

As *guards* são condições que precisam ser satisfeitas antes que uma cláusula é escolhida.

Uma *guard* pode ser um teste simples ou uma sucessão de testes simples separados por vírgulas. Um teste simples é uma comparação aritmética, a comparação de um termo, ou uma chamada para um sistema pré-definido de teste de função. As *guards* podem ser vistas como uma extensão do *pattern matching* ([ARM1993]).

Para avaliar uma *guard*, todos os seus testes são avaliados. Se todos são verdadeiros, então o *guard* tem sucesso, do contrário, falha. A ordem de avaliação dos testes em uma *guard* não é definida. Se a *guard* tem sucesso, então o corpo desta cláusula é avaliado. Se o teste da *guard* não obter sucesso, a próxima cláusula será candidata a ser experimentada.

Uma vez que a cabeça foi casada e *guard* de uma cláusula foi selecionado, o sistema executa esta cláusula e avalia o corpo da cláusula. Um exemplo do uso da cláusula *guard* pode ser visto no quadro 11 que apresenta um programa para calcular o fatorial.

Quadro 11: Programa fatorial

```
factorial(N) when N == 0 -> 1;
factorial(N) when N > 0 -> N * factorial(N - 1).
```

No quadro 12, é apresentado o exemplo do quadro acima, com a ordem das cláusulas invertidas. Neste caso, a combinação dos padrões da cabeça com os testes da cláusula *guard* servem para identificar exclusivamente a cláusula correta.

Quadro 12: Programa fatorial invertido

```
factorial(N) when N > 0 -> N * factorial(N - 1);
factorial(N) when N == 0 -> 1;
```

As operações possíveis em uma cláusula *guard* são apresentadas no quadro 13 .

Quadro 13: Operações *guards*

Operações	Descrição
X > Y	X maior que Y
X < Y	X menor que Y
X =< Y	X menor ou igual a Y
X >= Y	X maior ou igual a Y
X = = Y	X igual a Y
X / = Y	X diferente de Y
X = : = Y	X exatamente igual a Y
X = / = Y	X não exatamente igual a Y

Fonte: [CAS1998]

4.3.3.3 CORPO DA CLÁUSULA

O corpo de uma cláusula consiste de uma seqüência de uma ou mais expressões que são separadas por vírgulas. Todas as expressões de uma seqüência são avaliadas consecutivamente. O valor da seqüência é definido para ser o valor da última expressão da seqüência. Isto fica ilustrado no exemplo do quadro 14, onde a segunda cláusula da função fatorial é escrita.

Quadro 14: Corpo da cláusula

```
factorial(N) when N> 0 ->
    N1 = N - 1,
    F1 = factorial(N1),
        N * F1.
```

Durante a avaliação de uma seqüência, cada expressão é avaliada e o resultado pode ser casado com um padrão ou ser descartado. Segundo [ARM1993], há várias razões para dividir o corpo de uma função em uma seqüência de chamadas:

a) assegurar execução seqüencial do código. Cada expressão, no corpo de uma função, é avaliada consecutivamente, enquanto as funções ocorridas dentro de uma chamada de função podem ser executadas em qualquer ordem;
b) aumentar o entendimento. Pode ser mais claro escrever a função como uma seqüência de expressões;
c) desempacotar valores de retorno de uma função;

d) usar novamente os resultados de uma chamada de função.

4.3.4 PRIMITIVAS *CASE* E *IF*

A linguagem Erlang possui também as primitivas *case* e *if* que podem ser usadas como condições para avaliação dentro do corpo de uma cláusula, sem a necessidade do uso de uma função adicional.

A expressão *case* permite escolha entre alternativas dentro do corpo de uma cláusula e tem a sintaxe ilustrada no quadro 15.

Quadro 15: Estrutura da primitiva *case*

```
case Expr of
     Pattern1 [when Guard1] -> Seq1;
     Pattern2 [when Guard2] -> Seq2;
     ...
     PatternN [when GuardN] -> SeqN;
     True -> Seq99
end.
```

Fonte: [ARM1993]

Primeiramente, a expressão *Expr* é avaliada, então, o valor de *Expr* é casado seqüencialmente com os padrões *Pattern1*,..., *PatternN* até uma ligação é achada. Se uma ligação e um teste de uma *guard* obter sucesso, então correspondente seqüência de chamada é avaliada. As *guards* em um *case* têm a mesma forma como as *guards* de uma função. Então, o valor da primitiva *case* será o valor da seqüência selecionada.

No *case*, pelo menos um padrão tem que ser casado, pois, do contrário, será gerado um erro *run-time*. Para evitar este erro, deve ser adicionado o padrão *true* que irá garantir que ocorra o casamento com a última ramificação da primitiva *case*.

Neste caso, as guards *Guard1*,..., *GuardN* são avaliadas consecutivamente. Se um *guard* tem sucesso então a seqüência relacionada é avaliada. O resultado desta avaliação se torna o valor do *if*. Também no *if*, as *guard* têm a mesma forma das *guards* das funções.

Como no *case* acontece um erro se nenhuma das *guards* tiver sucesso. Nesta situação, o teste de *guard true* pode ser adicionado a estrutura da primitiva *if*.

A sintaxe da primitiva *if* pode ser vista no quadro 16.

Quadro 16: Estrutura da primitiva *if*

```
if
    Guard1 ->
        Sequence1;
    Guard2 ->
        Sequence2;
    ...
    GuardN ->
        SequenceN;
    True -> Sequence99
end.
```

Fonte: [ARM1993]

4.3.5 GARBAGE COLLECTION

Linguagens como C e C++ suportam funções que explicitamente alocam e desalocam a memória (C provê funções como *alloca*, *malloc*, *calloc* e *free* e C++ possui o *new* e *delete*). Erlang não possui gerenciamento explícito de memória, isto é, o programador livremente aloca e desaloca memória. No momento em que a linguagem cria um espaço para armazenar o conteúdo de uma variável, um espaço livre, se for necessário, é alocado automaticamente. Quando fica parada ou não é mais usada ou referenciada, a memória é desalocada.

O processo de administrar livremente a alocação de memória é chamado muitas vezes de *garbage collection*, sendo este, um mecanismo que pode evitar erros associados com o gerenciamento de memória.

4.4 CONCORRÊNCIA E TEMPO REAL EM ERLANG

Processos e comunicação entre processos são conceitos fundamentais em Erlang, e toda a concorrência, criação de processos e a comunicação entre processos, dentro do ambiente Erlang, são explícitas.

4.4.1 CRIAÇÃO DE PROCESSOS

Um processo é uma unidade auto-suficiente, separada de computação que existe concorrentemente com outros processos no sistema. Não há nenhuma hierarquia presente entre processos; o programador de uma aplicação pode criar a hierarquia explicitamente.

A função existente na biblioteca Erlang chamada spawn/3 (/3 significa o número de

argumentos) cria e começa a execução de um novo processo, como mostra o quadro 17.

Quadro 17: Função spawn

```
Pid = spawn(Module, FunctionName, ArgumentList)
```

Em vez de avaliar a função, entretanto, e devolver o resultado como em outras aplicações, *spawn/3* cria um novo processo simultâneo para avaliar a função e retornar o *Pid* (identificador de processo) do processo recentemente criado. *Pids* são usados para todas as formas de comunicação entre processos. A chamada para *spawn/3* retorna imediatamente quando o processo novo foi criado e não espera por uma determinada resposta da função.

Um processo terminará automaticamente quando a avaliação da função feita na chamada da criação for completada. O valor de retorno desta função fica perdido, não existe um espaço definido para este resultado aparecer.

Um identificador de processo é um objeto ou dado válido que pode ser manipulado como qualquer outro objeto. Por exemplo, pode ser armazenado em um *list* ou *tuple*, comparados a outros identificadores, ou enviar mensagens a outros processos.

4.4.2 COMUNICAÇÃO ENTRE PROCESSOS

Em Erlang, a única forma de comunicação entre processos é feita através de passagem de mensagens. Uma mensagem é enviada a outro processo por meio do caracter "!" (*send*) , como é apresentado no quadro 18.

Pid é o identificador do processo para o qual a mensagem é enviada. Uma mensagem pode ser qualquer termo válido de Erlang e *send* é uma primitiva que avalia seus argumentos. Seu valor de retorno é a mensagem enviada.

Quadro 18: Passagem de mensagem

```
Pid ! mensagem
```

Como demonstra o quadro 19, será primeiro avaliado foo(12) para adquirir o

identificador processo e bar(baz) para a mensagem ser enviada. Como em outras funções do Erlang, a ordem de avaliação não é definida. A primitiva *send* retorna a mensagem enviada com seu valor. O envio de uma mensagem é uma operação assíncrona que a chamada *send* fará não esperando pela mensagem chegar ao destino ou ser recebida. Até mesmo se o processo para o qual a mensagem está sendo enviada já terminou, o sistema não notificará o remetente. Isto faz parte da natureza assíncrona da passagem de mensagem. Esta aplicação se deve a implementação de todas as formas de verificação. Mensagens sempre são entregues aos destinatários, e na mesma ordem em que foram enviadas aos mesmos.

Quadro 19: Transmissão da mensagem

```
foo(12) ! bar(baz)
```

A primitiva *receive* é usada para receber mensagens. No quadro 20 é apresentda a sua sintaxe.

Cada processo tem uma caixa postal e todas as mensagens que são enviadas ao processo são armazenadas na caixa postal na mesma ordem em que elas chegam. No quadro 20, Message1 e Message2 são padrões que são emparelhados com outras mensagens que estão na caixa postal do processo. Quando há o casamento de uma mensagem, e a execução da *guard* correspondente tem sucesso, a mensagem é selecionada, removida da caixa postal e então a ação correspondente é avaliada. O *receive* retorna o valor da última expressão avaliada nas ações. Como em outras formas de *pattern matching*, qualquer variável não identificada, faz com que o casamento da mensagem seja parado. Qualquer mensagem que está na caixa postal e não for selecionada pelo *receive*, permanecerá na caixa postal na mesma ordem em que foram armazenadas e será casado diante de um próximo *receive*. O processo *receive* avaliado será suspenso até uma mensagem ser casada.

Erlang tem um seletivo mecanismo no *receive*, assim nenhuma mensagem que chega inesperadamente a um processo pode bloquear outras mensagens para aquele processo. Porém, toda mensagem não casada pelo *receive* permanece na caixa postal, e não será retirada, se o código elaborado pelo programador não conter nenhum comando que execute uma tarefa constante na caixa postal.

Quadro 20: Estrutura do *receive*

```
Receive
        Message1 [when Guard1] ->
              ações1;
        Message2 [when Guard2] ->
              ações2;
        ...
end.
```

Fonte: [ARM1993]

No momento em que um *receive* for executado, ele tenta encontrar uma mensagem que está na caixa postal, para emparelhar cada padrão com a mensagem. Quando há o casamento, a mensagem que foi emparelhada é retirada da caixa postal. O *receive* não implementa nenhum método de prioridade de mensagens, sendo que o casamento é feito na ordem em que as mensagens estão armazenadas. No caso de um *receive* genérico, com uma variável do tipo *anymessage* que não está ligada, a mensagem será assumida por qualquer mensagem contida na caixa postal, porém somente a primeira será casada e removida.

A implementação de prioridade de mensagens pode ser feita através do mecanismo *timeout* que será visto posteriormente.

4.4.2.1 RECEBIMENTO DE MENSAGENS DE UM PROCESSO ESPECÍFICO

É necessário freqüentemente receber mensagens de um processo específico. Para isto, o remetente deve incluir o próprio identificador do processo na mensagem, como o exemplo do quadro 21, onde é enviado uma mensagem que contêm explicitamente o identificador do processo remetente, retornando através da função *self ( )*, o identificador do processo.

Quadro 21: Especificando o processo

```
Pid ! {self( ) ,abc}
```

Quando uma mensagem é enviada para um processo específico, o comando somente será executado se o *Pid* for relacionado com o identificador do processo remetente. O quadro

22 mostra como esta mensagem pode ser recebida.

Quadro 22: Recebimento de mensagem de um processo específico

```
Receive
        {Pid, mensagem} ->
                executa comando a
end.
```

4.4.2.2 EXEMPLOS ILUSTRATIVOS

Para ilustrar os conceitos vistos sobre o tratamento de comunicação e concorrência entre processos, serão apresentados alguns exemplos mostrados em ([ARM1993]). O primeiro exemplo mostrado no quadro 23 é um módulo que cria processos contendo contadores que podem ser incrementados.

O exemplo do quadro 23 demonstra alguns conceitos básicos:

a) um novo processo contador é iniciado em cada chamada *counter:start/0* e cada processo é executado pela chamada da função *counter:loop(0)*;
b) a função recursiva responsável pela criação de um processo é suspensa quando está esperando uma entrada. O *loop* é esta função recursiva que assegura que o contador do processo será avaliado em um espaço constante;
c) recepção de uma mensagem seletiva, que neste caso é a mensagem *increment*.

Quadro 23: Exemplo de um contador

```
-module(counter).
-export([start/0, loop/1]).

start() ->
    spawn(counter, loop, [0]).

loop(Valor) ->
    receive
        increment ->
            loop(Valor + 1)
     end.
```

Fonte: [ARM1993]

O exemplo do quadro 23 também apresenta algumas deficiências:

a) não existe nenhum modo de acessar o valor do contador em cada um dos processos. Como os dados são locais do processo, só podem ser acessados pelo próprio processo;
b) o protocolo de mensagem está explícito. Outros processos podem enviar a mensagem *increment* para cada *counter*.

Estas deficiências são resolvidas no exemplo colocado no quadro 24. Este exemplo traz módulo *counter* que permite incrementar os contadores, ter acesso aos seus valores e também possibilita que os mesmos sejam parados.

Quadro 24: Exemplo de um contador melhorado

```
-module(counter).
-export([start/0, loop/1, increment/1, value/1, stop/1]).

start() ->
    spawn(counter, loop, [0]).
increment(Counter) ->
    Counter ! increment.
value(Counter) ->
    Counter ! {self(), value},
    receive
        {Counter, Value} ->
            Value
    end.
stop(Counter) ->
    Counter ! stop.

%%LOOP DO CONTADOR
loop(Valor) ->
    receive
        increment ->
            loop(Valor + 1);
        {From, value} ->
            From ! {self(), Valor},
            loop(Valor);
        stop ->           %SEM CHAMADA RECURSIVA
            true;
        Other ->          %TODAS AS MENSAGENS
            loop(Valor)
    end.
```

Fonte: [ARM1993]

Como no exemplo anterior, um novo processo contador é iniciado com *counter:start( )* que retorna o *Pid* do novo contador. Os protocolos das mensagens das funções *increment*,

value e *stop* são escondidas nas operações dos contadores.

O processo do contador usa um seletivo mecanismo de recebimento para processar os pedidos solicitados. Também é apresentada uma solução para o problema de manipulação de mensagens desconhecidas. A última cláusula no *receive* tem uma variável não ligada chamada *Other* como sua mensagem padrão; isto casará com qualquer mensagem que não foi emparelhada por outras cláusulas. Neste caso, a mensagem é ignorada e é aguardada uma próxima. Isto é uma técnica padrão para tratar mensagens desconhecidas: o *receive* retira as mensagens da caixa postal.

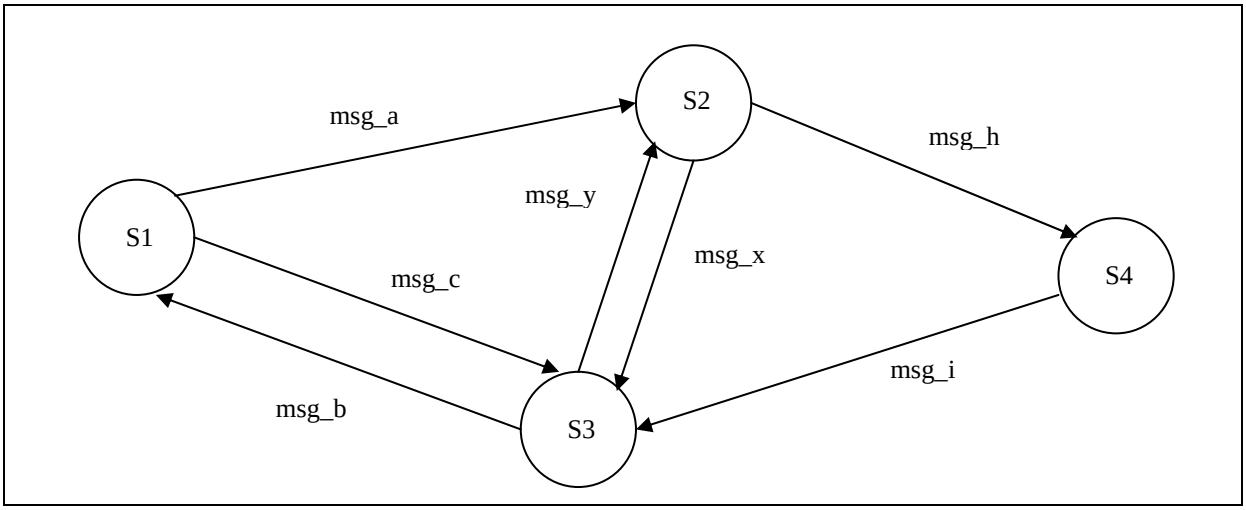
Quando o valor de um contador for acessado, é preciso enviar *Pid* como parte da mensagem para permitir que o processo de contador mande de volta uma resposta. Esta resposta também contém o identificador do processo remetente. Neste caso o contador irá habilitar a recepção de um processo específico para esperar pela mensagem que contém a resposta. É inseguro esperar por uma mensagem que contém um valor desconhecido, neste caso um número, como em outras mensagens que podem ser enviadas aos processos já emparelhados. Portanto, mensagens enviadas entre processos, normalmente possuem um modo de identificá-los, ou pelos seus conteúdos, como nas mensagens de pedido para o processo contador, ou através de um único e facilmente reconhecido identificador, como na resposta para o pedido do valor.

Em [ARM1993], também é apresentado o modelo de uma máquina de estado finito, mostrado na figura 9. O modelo tem quatro estados, possíveis transições e os eventos que causam estas transições.

Um modo fácil de implementar um programa como uma máquina de estado/evento é mostrado no quadro 25. Este código representa os estados e a administração das transições entre eles. Cada estado é representado por uma função separada e os eventos são representados através de mensagens.

No *receive*, existe a espera das funções estado por uma mensagem de evento. Quando uma mensagem é recebida, a máquina de estado finito faz uma transição para o novo estado chamando a função para aquele estado. Tendo certeza que cada chamada para um novo estado é uma última chamada, a máquina executará o processo em espaço constante.

Figura 9: Máquina de estado finito



Fonte: [ARM1993]

Quadro 25: Representação do código Erlang

`s1() -> receive msg_a -> s2(); msg_c -> s3() end.`	`s2() -> receive msg_x -> s3(); msg_h -> s4() end.`	`s3() -> receive msg_b -> s1(); msg_y -> s2() end.`	`s4() -> receive msg_i -> s3() end.`

Fonte: [ARM1993]

Os dados do estado podem ser direcionados para as funções do estado através da soma dos argumentos. Com este modelo, as ações serão executadas ao entrar em um estado que é finalizado antes do *receive* e qualquer ação que será executada ao deixar o estado é feita no *receive* depois de uma mensagem ter chegado, porém antes da chamada da função de um novo estado.

4.4.3 TIMEOUTS

A primitiva *receive* em Erlang pode ser incrementada com um opcional: o *timeout*. A sintaxe é elaborada como mostra o quadro 26.

Neste caso, *TimeOutExpr* é uma expressão que avalia um inteiro que é interpretado como um tempo cedido em milisegundos. A precisão do tempo será limitada pelo sistema

operacional ou hardware nos quais Erlang está sendo implementado. Assim, se nenhuma mensagem for selecionada neste tempo acontece o *timeout* e *ActionsT* é marcada para executar.

Quadro 26: Sintaxe do *receive* com *timeout*

```
Receive
    Message1 [When Guard1]
        Actions1;
    Message2 [When Guard2]
        Actions2;
    ...
    after
        TimeOutExpr -> ActionsT
     end.
```

Fonte: [ARM1993]

Dois valores que podem ser um argumento da expressão *timeout* têm um significado especial:

a) *Infinity*: O *atom infinity* é um *timeout* que nunca acontecerá. Isto pode ser útil se o tempo do *timeout* é calculado em tempo real. Quando é desejado executar uma expressão para calcular a duração do *timeout*, se o retorno é um valor infinito então a espera é indefinida;
b) 0 (zero): O *timeout* de 0 significa que o *timeout* acontecerá imediatamente, mas o sistema tenta todas as mensagens correntes na caixa postal.

Porém, os *timeouts* usando o *receive* ainda têm outras utilidades que podem ser imaginadas. A função *sleep(Time)* que aparece no quadro 27 suspende o processo atual durante um tempo de milisegundo:

Quadro 27: Função *sleep*

```
sleep(Time) ->
    receive
        after Time ->
            true
     end.
```

Fonte: [ARM1993]

No quadro 28 é apresentada a função *flush_buffer( )* que esvazia completamente a

caixa postal do processo corrente.

Quadro 28: Função *flush_buffer ( )*

```
flush buffer() ->
    receive
        AnyMessage ->
            flush_buffer()
        after 0 ->
            true
     end.
```

Fonte: [ARM1993]

Contando que existam mensagens na caixa postal, a primeira delas (a variável *AnyMessage* que não é ligada coincide com qualquer mensagem, isto é, a primeira mensagem) será selecionada e a função *flush_buffer* será chamada novamente, mas quando a caixa postal estiver vazia, a função retornará através da cláusula *true* do *timeout* (o valor do *timeout* igual a 0 garante isto).

A prioridade de mensagens pode ser implementada usando o valor de *timeout* igual 0, como mostra o quadro 29. A função *priority_receive* retornará a primeira mensagem da caixa postal, a não ser que a mensagem *interrupt* já tenha chegado, neste caso o *interrupt* será retornado. Primeiro é executado o *receive* para a mensagem *interrupt* com um *timeout* igual a 0, conferindo se aquela mensagem está na caixa postal, então retorna *interrupt*. Caso contrário, o *receive* é executado com o padrão *AnyMessage* que casará com a primeira mensagem da caixa postal.

Quadro 29: Prioridade de mensagens no *timeout*

```
Priority_receive() ->
    receive
        interrupt ->
            interrupt
        after 0 ->
            AnyMessage ->
                AnyMessage
        end
    end.
```

Fonte: [ARM1993]

O *timeout* também pode ser implementado fora da função *receive*, deixando de ser

puramente local. O *timeout* pode ser ativado através de uma chamada de função, criando um *timeout* independente, como demonstra o quadro 30.

Quadro 30: *Timeout* independente

```
-module(timer).
-export([timeout/2, cancel/1, timer/1]).

timeout(Time, Alarm) ->
    spawn(timer, timer, [self(), Time, Alarm]).
cancel(Timer) ->
    Timer ! {self(), cancel}.

timer(Pid, Time, Alarm) ->
    receive
        {Pid, cancel} ->
            true
    after Time ->
        Pid ! Alarm
    end.
```

Fonte: [ARM1993]

A chamada da função *timer:timeout(Time, Alarme)* acarreta a mensagem *Alarm* para ser enviada para a chamada do processo depois do tempo *Time*. Então, a função devolve um identificador para o cronômetro. Depois de completar sua tarefa, o processo pode esperar por esta mensagem. Usando o identificador do cronômetro, a chamada do processo pode cancelar o cronômetro chamando a função *timer:cancel(Timer).* Esta chamada, porém, não é a garantia de que quem está chamando, irá adquirir uma mensagem de alarme, pois a mensagem de cancelamento do cronômetro pode chegar depois da mensagem de alarme ter sido enviada.

4.4.4 REGISTRO DE PROCESSOS

Para enviar uma mensagem a um processo, é necessário saber seu identificador (*Pid*). Segundo [ARM1993], em alguns casos, isto não é prático e nem desejável. Por exemplo, em um sistema grande pode haver muitos servidores globais, ou um processo pode desejar esconder sua identidade por razões de segurança. Para admitir que um processo envie uma mensagem para outro processo sem conhecer sua identificação, o caminho é registrar os processos, atribuindo-lhes nomes.

Com o intuito de solucionar este problema, o Erlang possui um modo para registrar

processos, isto é, atribuir nomes aos processos, sendo que este deve ser uma variável do tipo atom.

Existem quatro funções básicas para manipular os nomes dos processos registrados:

a) *register(Name, Pid):* associa um nome *atom* ao processo *Pid*;
b) *unregister(Name):* remove a associação entre o nome *atom* e um processo;
c) *whereis(Name):* retorna o identificador de processo associado com o nome registrado *Name*; se nenhum processo foi associado com este nome, devolve um *atom* indefinido;
d) *registered( ):* retorna uma lista com todos os nomes de processos atualmente registrados.

A mensagem que envia a primitiva "!" também permite o uso do nome de um processo registrado como destino, como mostra o quadro 31.

Quadro 31: Processos registrados

```
number_analyser! {ego (), {analyse,[1,2,3,4]}}
```

No quadro 31, os processos enviam a mensagem {*Pid*, {*analyse*, [1, 2, 3, 4]}} para o processo registrado como *number_analyser*. *Pid* é o identificador de processos do processo que avalia a primitiva *send* "!".

4.4.5 ESCALONAMENTO DE PROCESSOS E TEMPO REAL

Na linguagem Erlang alguns critérios são observados para a implementação do escalonamento dos processos. São eles:

a) o algoritmo de escalonamento deve ser justo, isto é, qualquer processo que pode ser executado será executado, se possível na mesma ordem em que eles tornaram-se *runnable*;
b) a nenhum processo será permitido bloquear a máquina por muito tempo. É permitido a um processo por um período pequeno de tempo, chamado fatia de tempo, antes que seja reorganizado, para permitir outro processo no estado *runnable*

seja executado.

Segundo [ARM1993], são fixadas fatias de tempo para permitir que o processo que atualmente está executando termine em aproximadamente 500 reduções (uma redução é equivalente a uma chamada de função), antes de ser replanejado.

Um das exigências da linguagem Erlang era que seu uso fosse satisfatório para aplicações de *soft real-time*, onde tempos de resposta estivessem na ordem de milisegundos. Neste sentido, é necessário o conhecimento de um algoritmo de escalonamento com os critérios acima citados, para que uma aplicação com estas características possa ser implementada em Erlang.

Outra característica importante para sistemas de Erlang que serão usados para aplicações de tempo real é a administração de memória. Erlang esconde toda a administração de memória do programador ([ARM1993]). A memória é alocada automaticamente quando se torna necessário o uso de uma nova estrutura de dados e desalocada mais tarde, quando esta estrutura de dados fica sem uso durante um período mais longo. A alocação e o desalocamento da memória devem ser feitos de tal maneira que não bloqueie o sistema por qualquer duração de tempo, preferencialmente durante um tempo menor que a fatia de tempo de um processo. Desta forma, a natureza do tempo real de uma implementação não será afetada.

4.4.6 PRIORIDADES DE PROCESSOS

Atualmente, todos processos criados são executados com a mesma prioridade. Porém, algumas vezes, seria desejável que alguns processos executassem mais freqüentemente ou menos freqüentemente que outros. Por exemplo, um processo que é só executado ocasionalmente para monitorar o estado do sistema. Neste caso, a prioridade deste processo poderia ser modificada. Para isto, o Erlang possui a função *process_flag*( ) que é usada como mostra o quadro 32.

Quadro 32: Função *process_flag*( )

```
Process_flag(priority, Pri)
```

No quadro 31, *Pri* é a nova prioridade do processo no qual a chamada foi executada e pode ter o valor normal ou baixo. Os processos no estado *runnable* com prioridade baixa serão executados menos freqüentemente do que os processos com prioridade normal. A prioridade *default* para todos os processos é normal.

4.5 AMBIENTE GRÁFICO

O ambiente Erlang possui dois subsistemas gráficos: *gs* e *pxw*. Neste trabalho foi utilizado o subsistema gráfico *gs*. Portanto, torna-se importante a apresentação das principais características relacionadas com este modelo gráfico.

4.5.1 MODELO GS

O subsistema *gs* foi construído em um modelo de evento. Mensagens são enviadas entre o controlador do processo e o servidor *gs*, criando os eventos. Os objetos são criados com uma hierarquia. Cada objeto tem um pai e pode ter um ou mais filhos. Quando um objeto é criado, é devolvido um identificador do objeto para o seu criador. Os objetos podem ser criados com um nome específico, permitindo ao programador a escolha do identificador.

4.5.2 FUNÇÕES GS

A interface para *gs* é construída através de seis funções básicas: *gs:start*, *gs:stop*, *gs:create*, *gs:config*, *gs:destroy* e *gs:read*.

O *gs:start* inicia o servidor *gs*. Esta função não leva nenhum argumento. Um identificador é devolvido, sendo usado como o pai para esta janela. Se a função é chamada mais de uma vez, o mesmo identificador é devolvido.

O *gs:stop* serve para interromper o *gs* e fecha qualquer janela aberta pelo servidor gráfico. Esta função não leva nenhum argumento.

A função *gs:create* é usada para criar os objetos gráficos. Esta função tem diversas formas. Os tipos de argumentos usados são:

a) *objtype*: átomo que identifica o tipo dos objetos;

b) *parent*: identificador devolvido pelo pai;

c) *options*: lista de opções para ser selecionado para o objeto;

d) *option*: opção a ser selecionado para o objeto;

e) *name*: identificador a ser usado como referência pelo objeto.

A função *gs:config* estabelece uma opção para um objeto. Esta função tem duas formas. Uma forma leva um identificador do objeto ou um nome. Outra leva um único valor de opção como argumento através de uma lista.

A função *gs:destroy* destrói um objeto gráfico e seus filhos. Esta função leva como argumento um identificador do objeto ou um nome.

A função *gs:read* lê um valor de um objeto gráfico. Esta função tem como argumentos um identificador de objeto ou um nome e uma chave.

4.5.3 OBJETOS GS

O subsistema *gs* possui uma série de objetos gráficos que podem ser usados para construção de telas ou displays.

Uma janela é uma tela que contém outros objetos de tela. Só as janelas podem ser pais, os outros objetos são seus descendentes. A janela pode ter uma janela ou a servidor como seu pai. O átomo janela é usado para denotar este tipo de objeto.

Também é encontrada uma família de objetos chamados botões. O botão é um objeto que pode ser selecionado pelo mouse para selecionar ou não selecionar. Este pode ter como pai uma janela ou um *frame*. O átomo *button* é usado para denotar um simples botão, *radiobutton* é usado para denotar um tipo de botão onde somente um dos membros de um grupo de botões pode ser apertado em um mesmo instante, e *checkbutton* denota um tipo de botão onde podem ser selecionados muitos botões de um grupo ao mesmo tempo.

Um *label* é usado para exibir uma mensagem de texto ou uma figura na tela. O *label* pode ter como pai uma janela ou um *frame*. O átomo *label* é usado para denotar este tipo de objeto.

Um *frame* é um recipiente usado por um grupo de objetos. Um *frame* pode ter como

pai uma janela ou até mesmo, um próprio *frame*. O átomo *frame* é usado para denotar este tipo de objeto.

Um *entry* permite que uma simples linha de texto seja mostrada na tela, podendo ter como pai uma janela ou um *frame*. O átomo *entry* é usado para denotar este tipo de objeto.

Um *listbox* exibe uma lista de caracteres e permite que zero ou mais sejam selecionados. Seus pais podem ser uma janela ou um *frame*. O átomo *listbox* é usado para denotar este tipo de objeto.

Um *canvas* é uma área de desenho, onde podem estar presentes os seguintes objetos: imagem, linha, retângulos, textos. Uma janela ou um *frame* pode ser seu pai e o átomo *canvas* é usado para denotar este tipo de objeto.

Uma coleção de elementos foi provida e pode ser usada para construir um *menu*. O *menu* é uma estrutura gráfica recursiva que é usada para apresentar as opções e ações. Estas escolhas podem ser selecionadas. O *menu* pode ter como pais: *menubutton*, *menuitem* em forma de cascata, uma janela ou um *frame*. Os átomos *menu*, *menuitem*, *menubutton* e *menubar* são usados para denotar os objetos usados para a construção de um *menu*.

No quadro 33 e na figura 10 são apresentadas algumas características do ambiente gráfico *gs;* características estas que serão utilizadas na implementação do protótipo proposto neste trabalho.

O código Erlang mostrado no quadro 33 mostra um simples exemplo de um servidor gráfico *gs* com três botões e um texto. Este exemplo ilustra:

a) inicializa e finaliza um servidor gráfico usando *gs:start()* e *gs:stop();*
b) cria uma janela usando *gs:create*;
c) cria um *display* dentro da janela para mostrar os resultados;
d) cria os botões “+” , “-“ e “Quit” dentro da janela. O botão “+” soma 1 no *display*, o botão “-“ diminui 1 no *display* e o botão “Quit” fecha o gráfico *gs*, retornando ao ambiente Erlang.
e) troca as opções conectadas com o objeto gráfico usando *gs:config*;
f) tratamento dos eventos gerados no *even loop*.

Quadro 33: Características do ambiente gráfico *gs*

```
-module(thrbut).
-export([init/0 ]).

 init()->
     Server =gs:start(),
     Win =gs:create(window,Server,[{width,300},{height,80},
      {title, 'Ambiente Gráfico GS'}]),
     Display =gs:create(label,Win,[{label,{text,"0"}},
     {x,0},{y,0},{width,200},{height,50}]),
     Plus=gs:create(button,Win,[{label,{text,"+"}},
     {x,0},{y,50}]),
     Minus=gs:create(button,Win,[{label,{text,"-"}},
     {x,100},{y,50}]),
     Quit =gs:create(button,Win,[{label,{text,"Quit"}},
     {x,200},{y,50}]),
     gs:config(Win,{map,true}),
     event_loop(0,Display,Plus,Minus,Quit).

 event_loop(N,D,P,M,Q)->
     receive
        {gs,P,click,Data,Args}->
            RP =N+1,
            gs:config(D,{label,{text,RP}}),
            event_loop(RP,D,P,M,Q);
        {gs,M,click,Data,Args}->
            RM =N-1,
            gs:config(D,{label,{text,RM}}),
            event_loop(RM,D,P,M,Q);
        {gs,Q,click,Data,Args}->
            gs:stop(),
            N
     end.
```

Fonte: [CAS1998]

Figura 10: Visualização da interface gerada pelo código do quadro 33

5 DESCRIÇÃO DO MODELO DE ESPECIFICAÇÃO

Os grandes e complexos sistemas, especialmente os sistemas de tempo real, têm uma propriedade em que os eventos passados e presentes, tanto internos como externos, mudam seu comportamento. Estas mudanças são mais fundamentais do que a produzir uma saída com valor diferente, através da computação, quando mudam as entradas correspondentes ([HAT1990]).

Um sistema pode, de um instante para outro, sofrer alterações de comportamento como um processador inteiramente diferente. Esta espécie de sistema é difícil de ser representado apenas através de um modelo de processo, porém o modelo de máquina finita, conforme [HAT1990], é muito útil para esta situação, e uma vez combinado com o modelo de processo, torna-se uma ferramenta poderosa para representação de qualquer requisito de sistema.

O diagrama de transição de estado é um modelo de especificação para máquinas de estado finito, principalmente as máquinas seqüenciais, que são caracterizadas por determinarem suas saídas tanto por meio de entradas atuais quanto passadas. A memória, por assim dizer, diferencia esta de outras máquinas de estado finito, as quais não serão discutidas neste trabalho.

Os sistemas não triviais, na sua grande maioria, como sistemas de controle de tráfego aéreo ou ferroviário, são principalmente máquinas seqüenciais ([HAT1990]). Um sistema de tráfego aéreo sempre terá um plano de vôo dentro dele antes da própria decolagem, que influenciará suas saídas durante todo o vôo, onde fica claro que as saídas são designadas a partir de entradas passadas, caracterizando assim, uma máquina seqüencial.

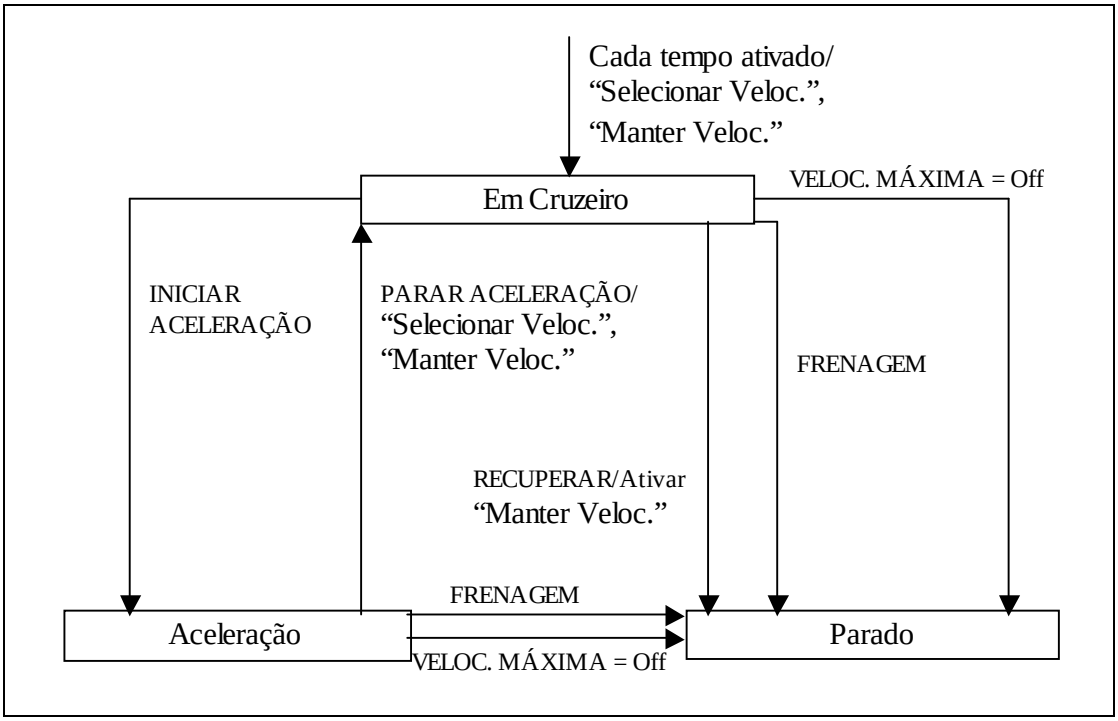
A memória de uma máquina seqüencial é representada na forma de estados, pois esta máquina sempre está em um estado especificado. Em cada um destes estados, alguns eventos, que são combinações de valores de entrada, obrigam a máquina a mudar de estado, ou a produzir saídas. As saídas das máquinas seqüenciais são chamadas de ação.

5.1 DIAGRAMA DE TRANSIÇÃO DE ESTADOS

Segundo [HAT1990], o diagrama de transição de estados (DTE) é a representação

mais familiar para uma máquina seqüencial. Um modelo de DTE é mostrado na figura 11.

Figura 11: Diagrama de transição de estados



Fonte: [HAT1990]

Os DTEs, conforme a figura 11, possuem quatro componentes para sua representação:

a) estados: os estados são representados por caixas retangulares contendo os nomes dos estados que elas representam: em cruzeiro, aceleração, parado;
b) arcos de transição: representados por linhas, com pontas de flechas, mostrando a direção da transição;
c) eventos: mostrados, pelos seus nomes, como rótulos sobre os arcos de transição por eles causados;
d) ações: mostradas, pelos seus nomes, adjacentes aos eventos que as provocam; os dois são separados por uma barra, como mostra o exemplo, EVENTO / "Ação".

Geralmente, um dos estados é denominado como o estado inicial, no qual a máquina será encontrada quando for ativada. Existe a possibilidade da máquina ser posicionada em um estado inicial desejado por algum processo externo de partida. O estado inicial é mostrado com um arco de transição entrando nele, porém, vindo de nenhum outro estado.

Em qualquer estado determinado, quando acontece um evento que esteja associado com uma transição a partir desse estado, a máquina irá para o estado indicado pelo arco de transição e, simultaneamente, realizará a ação associada. Caso ocorra algum evento que não está associado com uma transição a partir daquele estado, nada acontecerá.

As transições também podem voltar ao estado que deixaram. Isto acontece quando, em um estado especificado, é solicitado um evento para ocasionar uma ação, e não para mudar de estado. O contrário também pode ocorrer, se o evento muda o estado, mas não produz nenhuma ação.

O DTE convenciona que as ações tanto podem ser acionadas pelas saídas de transições como pelos estados. Outra convenção importante utilizada neste modelo de especificação é a de que as ações, embora associada com as transições, que são por natureza passageiras, supostamente continuam eficientes até ocorrer a próxima transição. Portanto, um processo ativado por ação particular permanecerá ativado e respondendo às entradas de dados de mudança até a próxima transição ([HAT1990]).

Em casos relativamente simples, o DTE é uma das melhores representações para máquinas seqüenciais. Porém, quando a quantidade de estados e transições é muito grande, a sua compreensão fica dificultada. Para estes casos mais complexos, podem ser adotadas duas representações alternativas:

a) a tabela de transição de estado (TTE);
b) a matriz de transição de estado (MTE).

5.1.1 TABELA DE TRANSIÇÃO DE ESTADO

O quadro 34 apresenta a tabela de transição de estado para o diagrama de transição de estado mostrado na figura 11. A tabela tem quatro colunas. A primeira coluna contém uma lista de cada um dos estados. A segunda coluna mostra, para cada estado atual, todos os eventos que causam a transição a partir dele. A terceira mostra a ação associada com cada transição, quando existe ação. A quarta coluna apresenta o estado para o qual cada transição é direcionada. A TTE pode ser apresentada em várias páginas, de acordo com a necessidade de especificação para o sistema em questão.

Quadro 34: Tabela de transição de estado

Estado Atual	Evento	Ação	Próximo Estado
Início	Cada Tempo Ativado	"Selecionar Veloc.", "Manter Veloc"	Em Cruzeiro
Em Cruzeiro	VELOC. MÁXIMA = Off	X	Inativo
	FRENAGEM	X	Inativo
	INICIAR ACELERAÇÃO	X	Acelerando
Inativo	Recuperar	"Manter Veloc"	Em Cruzeiro
Acelerando	FRENAGEM	X	Inativo
	VELOC. MÁXIMA = Off	X	Inativo
	PARAR ACELERAÇÃO	"Selecionar Veloc.", "Manter Veloc"	Em Cruzeiro

Fonte: [HAT1990]

5.1.2 MATRIZ DE TRANSIÇÃO DE ESTADO

O quadro 35 mostra a matriz de transição de estado.

Quadro 35: Matriz de transição de estado

Evento / Estado	Cada Tempo Ativado	VELOC. MÁXIMA = Off	FRENAGEM	INÍCIO ACELER.	RECUPE-RAÇÃO	PARAR ACELER.
Início	"Selecionar Veloc." "Manter Veloc."					
	Em Cruzeiro					
Em Cruzeiro						
		Inativo	Inativo	Em Aceler.		
Inativo					"Manter Veloc."	
					Cruzeiro	
Em Aceler.						"Selecionar Veloc." "Manter Veloc."
						Cruzeiro

Fonte: [HAT1990]

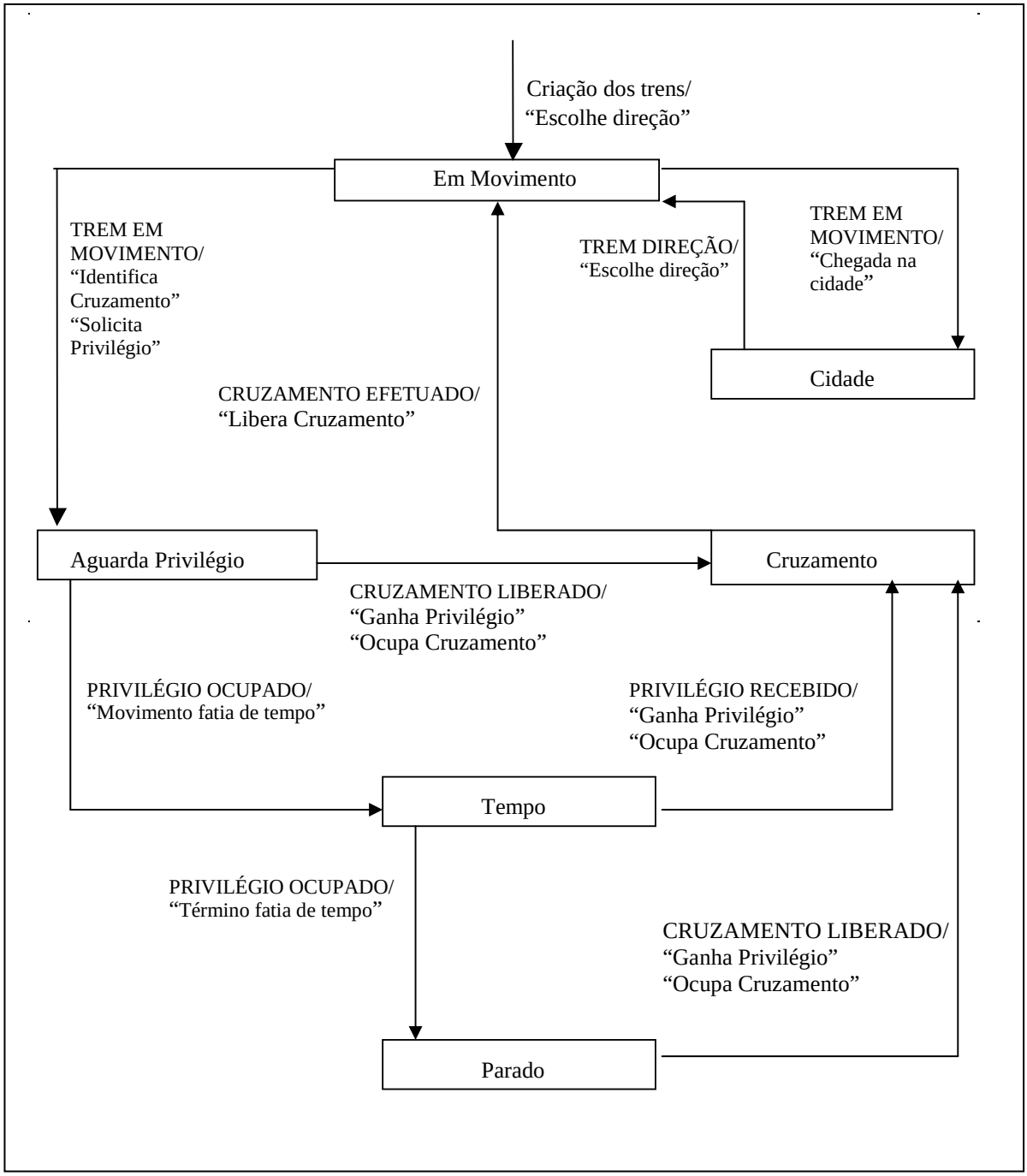
Na tabela, os estados são colocados no lado esquerdo da matriz, e os eventos ao longo da linha de cima. Os elementos da matriz mostram a ação causada pelo evento acima daquele elemento quando a máquina está no estado à esquerda daquele elemento.

Segundo [HAT1990], esta representação tem a vantagem de fornecer uma prova visual rápida do que ocorre enquanto a máquina está em um determinado estado. Tem também a vantagem de apresentar todas as combinações possíveis de estados e eventos, proporcionando uma forma de teste de integralidade.

6 ESPECIFICAÇÃO DO PROTÓTIPO

Conforme apresentado no capítulo 5 deste trabalho, o protótipo foi especificado através do diagrama de transição de estados (DTEs). A figura 12 apresenta o diagrama de transição de estados do controle da malha ferroviária implementado neste trabalho.

Figura 12: DTEs do protótipo



Na malha ferroviária implementada, tem-se os seguintes estados, com seus devidos eventos e ações:

a) ‘Em Movimento’: neste estado, o trem está em movimento na malha ferroviária, passando pelos cruzamentos e cidades da mesma. Durante este estado, ocorre o evento ‘TREM EM MOVIMENTO’, sendo que este pode gerar as ações: “Chegada na cidade”, “Identifica Cruzamento” e “Solicita Privilégio”. Quando acontece a ação “Chegada na cidade”, o trem vai para o estado ‘Cidade’. Com a ação “Identifica Cruzamento” e “Solicita Privilégio”, o trem entra no estado ‘Aguarda Privilégio’;
b) ‘Cidade’: neste estado, através do evento ‘TREM DIREÇÃO’, ocorre a ação ‘Escolhe Direção’, que passa novamente o trem para o estado ‘Em Movimento’, apontando o próximo caminho que o trem irá trilhar;
c) ‘Aguarda Privilégio’: neste estado, se o cruzamento estiver liberado, ocorre o evento ‘CRUZAMENTO LIBERADO’, e através da ação ‘Ganha Privilégio’ e ‘Ocupa Cruzamento’, o trem entra no estado ‘Cruzamento’. Se o cruzamento estiver ocupado, ocorre o evento ‘PRIVILÉGIO OCUPADO’ que acarreta a ação “Movimento fatia de tempo”. Com estão ação, o trem continua seu caminho até o estado ‘Tempo’.
d) ‘Cruzamento’: deste estado, o trem passa para o estado ‘Em Movimento’, através do evento ‘CRUZAMENTO EFETUADO, gerando a ação ‘Libera Cruzamento’, que deixará o cruzamento livre para outro trem.
e) ‘Tempo’: neste estado, se o privilégio for recebido antes do término da fatia de tempo, ocorre o evento ‘PRIVILÉGIO RECEBIDO’, e através das ações ‘Ganha Privilégio’ e ‘Ocupa Cruzamento’, o trem vai para o estado ‘Cruzamento’. Se o privilégio não for recebido e fatia de tempo que foi determinada acabar, ocorre o evento ‘PRIVILEGIO OCUPADO’, e através da ação ‘Término fatia de tempo’, o trem passa para o estado ‘Parado’;
f) ‘Parado’: quando ocorrer o evento ‘CRUZAMENTO LIBERADO’, o trem passará para o estado ‘Cruzamento’ através das ações ‘Ganha Privilégio’ e ‘Ocupa Cruzamento’, continuando seu trajeto na malha ferroviária.

7 IMPLEMENTAÇÃO E APRESENTAÇÃO DO PROTÓTIPO

Neste capítulo serão apresentadas as principais características do protótipo implementado, assim como os algoritmos mais importantes utilizados para o desenvolvimento desta implementação. Também será apresentada a tela principal do protótipo.

7.1 PROPRIEDADES DO PROTÓTIPO

O protótipo permite a simulação de uma malha ferroviária entre doze cidades, com a circulação de no máximo cinqüenta trens. Este limite foi imposto na implementação para facilitar a visualização da malha, pois, segundo a literatura e conforme os testes efetuados, a linguagem não tem limitação em relação ao número de processos concorrentes. O que ocorre, com a criação de um número elevado de processos, é uma desaceleração gradativa na execução dos mesmos.

A malha ferroviária do protótipo foi elaborada com a criação de um ambiente gráfico, onde são apresentadas as doze cidades, as ferrovias entre elas e os três cruzamentos existentes na malha.

As ferrovias foram implementadas sem exclusividade, isto é, sem restrição de número de trens. Por este motivo, dois trens podem ocupar a mesma posição em um determinado instante. As ferrovias são tratadas como vias de mão dupla, portanto, para ir, o trem passa pela via da direita, e para voltar, pela via da esquerda. Vias estas, representadas em uma única ferrovia.

As cidades, representadas no protótipo pelas letras de A até L, podem receber vários trens ao mesmo tempo. O trem que chega em uma cidade, escolhe, conforme sua posição, a próxima direção a seguir, ficando, assim em movimento na tela até que a aplicação seja finalizada. As cidades, são também, no ambiente gráfico, botões, que quando clicados, criam os trens e os colocam em movimento na malha.

Os cruzamentos encontrados na malha ferroviária só recebem um trem de cada vez. Ao se aproximar do cruzamento, o trem solicita a sua passagem no mesmo, caso o cruzamento

esteja liberado, o trem ganha exclusividade e passa pelo cruzamento, deixando o mesmo ocupado durante a sua passagem. Quando termina a passagem, o trem libera o cruzamento. Caso um trem solicite sua passagem por um cruzamento e o mesmo esteja ocupado, ele continua sua trajetória durante um tempo determinado, chegando desta forma, mais próximo do cruzamento, mesmo que não tenha recebido a mensagem indicando que o cruzamento está liberado. Se não receber a liberação, depois de um determinado tempo, o trem fica parado, aguardando em uma fila, até receber uma mensagem que indica que o cruzamento foi liberado. Se o trem receber a liberação antes do tempo máximo estipulado, ele prossegue no cruzamento, sem ter que parar antes do mesmo. O tempo implementado no tratamento do cruzamento demonstra a utilização dos conceitos de tempo real propostos no início deste trabalho.

7.2 DESCRIÇÃO DAS PRINCIPAIS FUNÇÕES DO PROTÓTIPO

No início da implementação são criados os três processos cruzamento que ficarão ativos durante toda a execução do protótipo. No quadro 36 é apresentada a função *start* responsável por esta criação.

Quadro 36: Criação dos processos cruzamento

```
%%-----------------------------------------------------------------------------
%%A função start inicia o programa criando três processos para controle dos
%%cruzamentos
%%-----------------------------------------------------------------------------

start()->
       start(1).
start(N) ->
       io:format("Pid: ~w~n", [self()]),
       PC1 = spawn(tcc, mensagem, [N, []]),
       PC2 = spawn(tcc, mensagem, [N, []]),
       PC3 = spawn(tcc, mensagem, [N, []]),
       io:format("Pid PC1: ~w~n", [PC1]),
       io:format("Pid PC2: ~w~n", [PC2]),
       io:format("Pid PC3: ~w~n", [PC3]),
       iniciar(PC1,PC2,PC3).
```

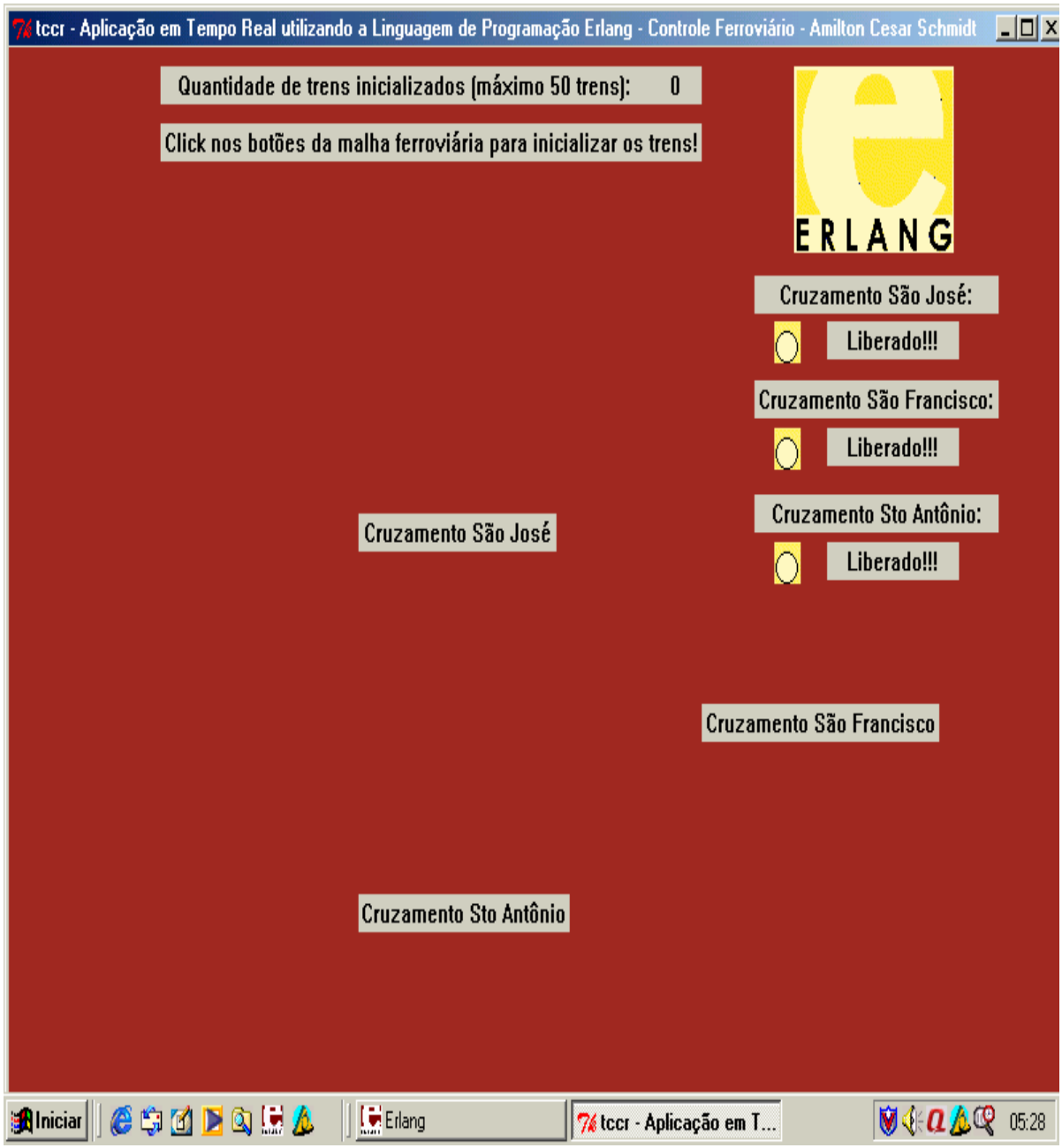
A função *start* chama a função iniciar passando o *Pid* dos processos cruzamento que foram criados. As identificações destes processos serão utilizadas durante toda a execução. A

função iniciar é responsável pela criação da tela principal do protótipo, com a apresentação da malha ferroviária, das cidades e dos cruzamentos, assim como, todas as informações necessárias para a utilização do protótipo. No quadro 37 é apresentada a criação da tela com suas características, tais como, a colocação na tela do logo da linguagem Erlang, a criação do botão ‘Pare’ que finaliza a aplicação, a mensagem e o contador de trens, a posição dos cruzamentos na malha ferroviária com a nomeação de cada um deles. A figura 13 mostra a representação desta tela.

Quadro 37: Criação da tela principal

```
%%--------------------------------------------------------------------
%%A função iniciar cria a tela principal do programa
%%--------------------------------------------------------------------
iniciar(PC1,PC2,PC3)->
       Server =gs:start(),
       Win =gs:create(window,Server,[{width,800},{height,600},
              {title, 'TCC - Aplicação em Tempo Real utilizando a Linguagem
             de Programação Erlang - Controle Ferroviário - Amilton Cesar
Schmidt'}]),
       Canvas = create(canvas, Win,[{width,800},{height,600}, {bg,brown}]),
       Contador = gs:create(label,Win,[{label,{text,"Quantidade de trens
             Inicializados {máximo 50 trens):"}},{x,115},{y,10},
             {width,370},{height,20}]),
       Trens = gs:create(label,Win,[{label,{text,"Click nos botões da malha
             Ferroviária   para inicializar os trens!"}},{x,115},{y,40},
             {width,410},{height,20}]),
       Cruz1 = gs:create(label,Win,[{label,{text,"Cruzamento São José"}},
              {x,265},{y,245},{width,150},{height,20}]),
       Cruz2 = gs:create(label,Win,[{label,{text,"Cruzamento São
             Francisco"}},{x,525},{y,345},{width,180},{height,20}]),
       Cruz3 = gs:create(label,Win,[{label,{text,"Cruzamento Sto
             Antônio"}},   {x,265},{y,445},{width,160},{height,20}]),
       Cruz4 = gs:create(label,Win,[{label,{text,"Cruzamento São José:"}},
              {x,565},{y,120},{width,185},{height,20}]),
       Cruz5 = gs:create(label,Win,[{label,{text,"Cruzamento São
             Francisco:"}},{x,565},{y,175},{width,185},{height,20}]),
       Cruz6 = gs:create(label,Win,[{label,{text,"Cruzamento Sto
             Antônio:"}},{x,565},{y,235},{width,185},{height,20}]),
       Cruz7 = gs:create(label,Canvas,[{label,{text,"Liberado!!!"}},
              {x,620},{y,144},{width,100},{height,20}]),
       Cruz8 = gs:create(image,Canvas,[{load_gif, "pie0.gif"},{coords,
             [{580,144}]}]),
       Cruz9 = gs:create(label,Canvas,[{label,{text,"Liberado!!!"}},
             {x,620},{y,200},{width,100},{height,20}]),
       Cruz10 = gs:create(image,Canvas,[{load_gif, "pie0.gif"},
             {coords, [{580,200}]}]),
       Cruz11 = gs:create(label,Canvas,[{label,{text,"Liberado!!!"}},
             {x,620},{y,260},{width,100},{height,20}]),
       Cruz12 = gs:create(image,Canvas,[{load_gif, "pie0.gif"},
             {coords, [{580,260}]}]),
       Logo = gs:create(image,Canvas,[{load_gif, "logo.gif"},
             {coords, [{595,10}]}]),
       Display = gs:create(label,Win,[{label,{text,"0"}},{x,485},{y,10},
             {width,40},{height,20}]),
```

Figura 13: Representação da tela principal



No quadro 38, são criados os botões que representam as doze cidades da malha ferroviária. Cada cidade é representada por uma letra maiúscula do alfabeto. Quando um dos botões da tela é clicado, o trem é criado e a partir da posição da cidade na tela, começa o movimento na malha ferroviária. Através dos botões, poderão ser inicializados cinqüenta trens que ficarão em movimento na tela até que o botão ‘Pare’ seja acionado. A figura 14 mostra a tela com a implementação dos botões.

Quadro 38: Botões que representam as cidades

```
%%-------------------------------------------------------------------------
%%Na continuação da função iniciar são criados os botões de acionamento dos %%trens
%%-------------------------------------------------------------------------

   AA = gs:create(button, Win, [{x,40}, {y,120},{width,20},{height,20}, {label, {text, "A"}}]),
   AB = gs:create(button, Win, [{x,240},{y,120},{width,20},{height,20}, {label, {text, "B"}}]),
   AC = gs:create(button, Win, [{x,500},{y,120},{width,20},{height,20}, {label, {text, "C"}}]),
   BA = gs:create(button, Win, [{x,100},{y,220},{width,20},{height,20}, {label, {text, "D"}}]),
   BB = gs:create(button, Win, [{x,500},{y,220},{width,20},{height,20}, {label, {text, "E"}}]),
   CA = gs:create(button, Win, [{x,240},{y,320},{width,20},{height,20}, {label, {text, "F"}}]),
   CB = gs:create(button, Win, [{x,740},{y,320},{width,20},{height,20}, {label, {text, "G"}}]),
   DA = gs:create(button, Win, [{x,40}, {y,420},{width,20},{height,20}, {label, {text, "H"}}]),
   DB = gs:create(button, Win, [{x,500},{y,420},{width,20},{height,20}, {label, {text, "I"}}]),
   DC = gs:create(button, Win, [{x,700},{y,420},{width,20},{height,20}, {label, {text, "J"}}]),
   EA = gs:create(button, Win, [{x,240},{y,520},{width,20},{height,20}, {label, {text, "K"}}]),
   EB = gs:create(button, Win, [{x,500},{y,520},{width,20},{height,20}, {label, {text, "L"}}]),
   FA = gs:create(button, Win, [{x,40}, {y,10}, {width,60},{height,50}, {label, {text, "PARE"}}]),
        linha(Canvas),
        cria_trem(0, Canvas, Display, AA, AB, AC, BA, BB, CA, CB, DA, DB, DC,
            EA, EB, FA, PC1, PC2, PC3).
```

Figura 14: Representação dos botões da malha ferroviária

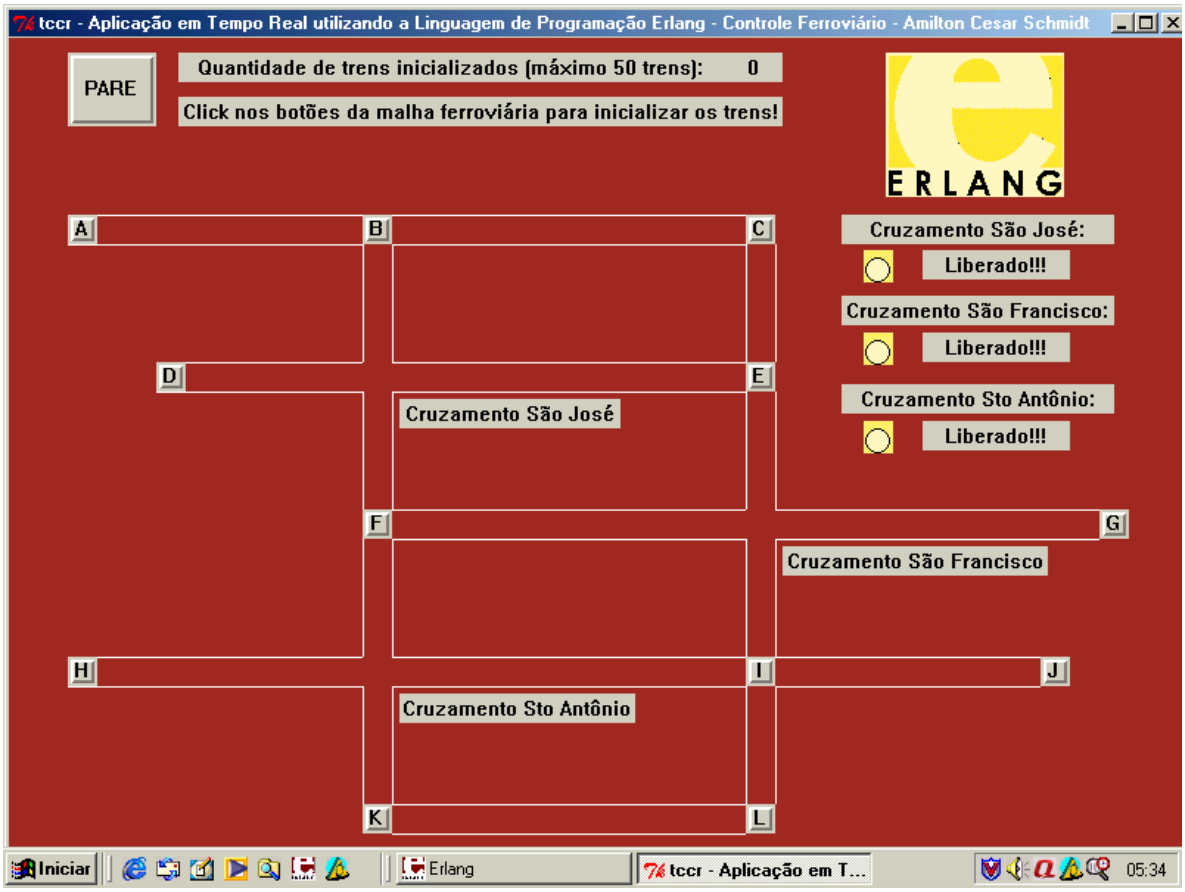
Após a criação dos botões, são chamadas as funções `linha` e `cria_trem`. A função linha, como mostra o quadro 39 e a figura 15, é responsável pela criação das linhas que representam as ferrovias. A função `cria_trem`, apresentada no quadro 40, é responsável pela criação dos processos `trem1, trem2, trem3 e trem4`. Esta função é acionada através do *click* nos botões das cidades. Os processos `trem1, trem2, trem3 e trem4` são funções que iniciam o movimento dos trens na malha. A função `trem1` representa o movimento para direita, a `trem2`, o movimento para esquerda, a `trem3`, o movimento para baixo, e a `trem4`, o movimento para cima. As funções `linha` e `cria_trem` são apresentadas no quadro 39 e 40 respectivamente, de forma resumida.

Quadro 39: A função linha

```
%%-------------------------------------------------------------------------
%%A função linha cria as linhas que representam os trilhos da malha %%ferroviária
%%-------------------------------------------------------------------------

linha(C) ->
       create(line, C, [{coords, [{40,120},  {240,120}]}, {fg, white},{fg, white}]),
       create(line, C, [{coords, [{40,140},  {240,140}]}, {fg, white},{fg, white}]),
       create(line, C, [{coords, [{260,120}, {500,120}]}, {fg, white},{fg, white}]),
       create(line, C, [{coords, [{260,140}, {500,140}]}, {fg, white},{fg, white}]),
       create(line, C, [{coords, [{100,220}, {240,220}]}, {fg, white},{fg, white}]),
       create(line, C, [{coords, [{100,240}, {240,240}]}, {fg, white},{fg, white}]),
       create(line, C, [{coords, [{260,220}, {500,220}]}, {fg, white},{fg, white}]),
       create(line, C, [{coords, [{260,240}, {500,240}]}, {fg, white},{fg, white}]),
       create(line, C, [{coords, [{260,320}, {500,320}]}, {fg, white},{fg, white}]),
       create(line, C, [{coords, [{260,340}, {500,340}]}, {fg, white},{fg, white}]),
       create(line, C, [{coords, [{520,320}, {740,320}]}, {fg, white},{fg, white}]),
       create(line, C, [{coords, [{520,340}, {740,340}]}, {fg, white},{fg, white}]),
       create(line, C, [{coords, [{40,420},  {240,420}]}, {fg, white},{fg, white}]),
       create(line, C, [{coords, [{40,440},  {240,440}]}, {fg, white},{fg, white}]),
       create(line, C, [{coords, [{260,420}, {500,420}]}, {fg, white},{fg, white}]),
       create(line, C, [{coords, [{260,440}, {500,440}]}, {fg, white},{fg, white}]),
       create(line, C, [{coords, [{520,420}, {700,420}]}, {fg, white},{fg, white}]),
       create(line, C, [{coords, [{520,440}, {700,440}]}, {fg, white},{fg, white}]),
       create(line, C, [{coords, [{260,520}, {500,520}]}, {fg, white},{fg, white}]),
       create(line, C, [{coords, [{260,540}, {500,540}]}, {fg, white},{fg, white}]),
       create(line, C, [{coords, [{240,140}, {240,220}]}, {fg, white},{fg, white}]),
       create(line, C, [{coords, [{260,140}, {260,220}]}, {fg, white},{fg, white}]),
       create(line, C, [{coords, [{240,240}, {240,320}]}, {fg, white},{fg, white}]),
       create(line, C, [{coords, [{260,240}, {260,320}]}, {fg, white},{fg, white}]),
       create(line, C, [{coords, [{240,340}, {240,420}]}, {fg, white},{fg, white}]),
       create(line, C, [{coords, [{260,340}, {260,420}]}, {fg, white},{fg, white}]),
       create(line, C, [{coords, [{240,440}, {240,520}]}, {fg, white},{fg, white}]),
       create(line, C, [{coords, [{260,440}, {260,520}]}, {fg, white},{fg, white}]),
       create(line, C, [{coords, [{500,140}, {500,220}]}, {fg, white},{fg, white}]),
       create(line, C, [{coords, [{520,140}, {520,220}]}, {fg, white},{fg, white}]),
       create(line, C, [{coords, [{500,240}, {500,320}]}, {fg, white},{fg, white}]),
       create(line, C, [{coords, [{520,240}, {520,320}]}, {fg, white},{fg, white}]),
       create(line, C, [{coords, [{500,340}, {500,420}]}, {fg, white},{fg, white}]),
       create(line, C, [{coords, [{520,340}, {520,420}]}, {fg, white},{fg, white}]),
       create(line, C, [{coords, [{500,420}, {500,520}]}, {fg, white},{fg, white}]),
       create(line, C, [{coords, [{520,440}, {520,520}]}, {fg, white},{fg, white}]).
```

Figura 15: Representação das linhas



Quadro 40: A função cria_trem

```
%%-----------------------------------------------------------------------
%%A função cria_trem inicializa os trens na malha ferroviária a partir do
%%click nos botões
%%-----------------------------------------------------------------------

cria_trem (N, C, D, A1, A2, A3, B1, B2, C1, C2, D1, D2, D3, E1, E2, F1,
    PC1, PC2, PC3) ->
    receive
        {gs, F1, click, Data, Args} ->
             gs:stop();
        {gs, A1, click, Data, Args} ->
             CT = N + 1,
             if N == 50 ->
                   bye;
                true ->
                   gs:config(D, {label, {text, CT}}),
                   spawn (tcc, trem1,[C, 50, 120, blue, PC1, PC2, PC3]),
                   cria_trem (CT, C, D, A1, A2, A3, B1, B2, C1, C2, D1, D2,
                              D3, E1, E2, F1, PC1, PC2, PC3)
             end;
```

A função `cria_trem` apresentada no quadro 40 cria o processo `trem1` que é uma função responsável pelo movimento do trem na malha ferroviária. No quadro 41 é mostrada a implementação desta função.

Quadro 41: A função trem

```
trem1 (C, X, Y, W, PC1, PC2, PC3) ->
      cria1(C, X, Y, W),
      if
            X == 40, Y == 120; X == 240, Y == 120; X == 500, Y == 120;
            X == 100, Y == 220; X == 240, Y == 320; X == 500, Y == 220;
            X == 740, Y == 320; X == 40, Y == 420; X == 500, Y == 420;
            X == 700, Y == 420; X == 240, Y == 520; X == 500, Y == 520 ->
                  A = direcao(C, X, Y, W, PC1, PC2, PC3);
            X == 180, Y == 220 ->
                  Xatual_a = tempo1(C, X+5, Y, W, PC1, PC2, PC3),
                  io:format("hora entrada: ~w~n", [time()]),
                  figura(C,1),
                  A = cruzamento1 (C, Xatual_a+5, Y, W, PC1, PC2, PC3),
                  io:format("retornou do cruzamento 1: ~w~n", [time()]);
            X == 440, Y == 320 ->
                  Xatual_a = tempo1(C, X+5, Y, W, PC1, PC2, PC3),
                  io:format("hora entrada: ~w~n", [time()]),
                  figura(C,2),
                  A = cruzamento1 (C, Xatual_a+5, Y, W, PC1, PC2, PC3),
                  io:format("retornou do cruzamento 1: ~w~n", [time()]);
            X == 180, Y == 420 ->
                  Xatual_a = tempo1(C, X+5, Y, W, PC1, PC2, PC3),
                  io:format("hora entrada: ~w~n", [time()]),
                  figura(C,3),
                  A = cruzamento1 (C, Xatual_a+5, Y, W, PC1, PC2, PC3),
                  io:format("retornou do cruzamento 1: ~w~n", [time()]);
            true ->
                  A = X,
                  deleta1(C,X,Y)
      end,
      io:format("trem1 X: ~w  Y: ~w PID: ~w~n", [A,Y,self()]),
      trem1(C, A+5, Y, W, PC1, PC2, PC3).
cria1(C, X, Y, W) ->
      create(rectangle,C,[{coords,[{X,Y+5},{X+20,Y+15}]},
            {fg, black}, {fill, W}]),
      receive
            after 250 -> true
      end.
deleta1(C,X,Y) ->
      create(rectangle,C,[{coords,[{X,Y+5},{X+20,Y+15}]},
      {fg, brown}, {fill, brown}]).
```

A função `trem1` possui várias responsabilidades que são comuns nas funções `trem2`, `trem3` e `trem4`, as quais são:

a) identificar cidades e chamar a função direção para escolher a nova direção a ser seguida;
b) identificar cruzamentos e chamar a função tempo e cruzamento. A função tempo é responsável pelo tratamento da aproximação do cruzamento enquanto ele estiver ocupado. Também é esta função que envia a solicitação de exclusividade para o cruzamento. Caso esta solicitação seja atendida, a função cruzamento é chamada para tratar a passagem pelo cruzamento e esta função é responsável pelo envio da mensagem de liberação do semáforo;
c) criar os retângulos na tela que representam os trens, e simular o seu movimento, através das chamadas das funções `cria1` e `deleta1` retângulos.

A função `cruzamento1` chamada pela função `trem1` é responsável pelo tratamento do cruzamento, isto é, pelo movimento do trem durante o cruzamento e também por enviar uma mensagem de liberação do cruzamento para a função semáforo. A função cruzamento é apresentada no quadro 42.

Quadro 42: A função cruzamento

```
cruzamento1 (C, 260, 220, W, PC1, PC2, PC3) ->
      deleta1(C,255,220),
      v(PC1),
      io:format("hora saida: ~w~n", [time()]),
      figura(C,4),
      io:format("Liberou cruzamento PID: ~w~n", [self()]),
      260;
cruzamento1 (C, 520, 320, W, PC1, PC2, PC3) ->
      deleta1(C,515,320),
      v(PC2),
      io:format("hora saida: ~w~n", [time()]),
      figura(C,5),
      io:format("Liberou cruzamento PID: ~w~n", [self()]),
      520;
cruzamento1 (C, 260, 420, W, PC1, PC2, PC3) ->
      deleta1(C,255,420),
      v(PC3),
      io:format("hora saida: ~w~n", [time()]),
      figura(C,6),
      io:format("Liberou cruzamento PID: ~w~n", [self()]),
      260;
cruzamento1 (C, X, Y, W, PC1, PC2, PC3) ->
      deleta1(C,X-5,Y),
      cria1(C, X, Y, W),
      cruzamento1 (C, X+5, Y, W, PC1, PC2, PC3).
```

Na função `trem1` também é chamada a função direção que é responsável pela escolha da próxima direção que o trem irá percorrer na malha ferroviária. Desta forma, o trem iniciado em qualquer dos botões da tela que representam as cidades ficará em movimento na malha até que o botão ‘Pare’ seja acionado. Toda vez que o trem chegar em uma cidade, ele chamará a função direção que lhe indicará o trilho que este trem deverá percorrer. A direção do trem é determinada conforme o horário que o mesmo chegou na cidade. São os segundos da hora em que o trem chegou em uma cidade que determinarão para onde o trem irá seguir. A função direção é apresentada no quadro 43.

Quadro 43: A função direção

```
direcao(C, X, Y, W, PC1, PC2, PC3) ->
      DIR1 = time(),
      io:format("HORA CIDADE: ~w~n", [DIR1]),
      DIR2 = element(3,DIR1),
      io:format("elemento: ~w~n", [DIR2]),
      DIR3 = round(DIR2/10),
      io:format("ELEMENTO ARRENDODADO: ~w~n", [DIR3]),
      if
            X == 40, Y == 120; X == 100, Y == 220; X == 40, Y == 420 ->
                  trem1(C, X+5, Y, W, PC1, PC2, PC3),
                  direcao(C, X, Y, W, PC1, PC2, PC3);
            X == 740, Y == 320; X == 700, Y == 420 ->
                  trem2(C, X-5, Y, W, PC1, PC2, PC3),
                  direcao(C, X, Y, W, PC1, PC2, PC3);
            X == 240, Y == 120 ->
                  DIR3,
                  if DIR3 == 1 ; DIR3 == 4 ->
                        trem1(C, X+5, Y, W, PC1, PC2, PC3),
                        direcao(C, X, Y, W, PC1, PC2, PC3);
                  DIR3 == 2; DIR3 == 5 ->
                        trem2(C, X-5, Y, W, PC1, PC2, PC3),
                        direcao(C, X, Y, W, PC1, PC2, PC3);
                  DIR3 == 0; DIR3 == 3; DIR3 == 6 ->
                        trem3(C, X, Y+5, W, PC1, PC2, PC3),
                        direcao(C, X, Y, W, PC1, PC2, PC3)
                  end;
```

Uma das partes mais importantes da implementação é o tratamento do cruzamento. Para este tratamento foram utilizados conceitos de tempo real. O tempo real foi implementado através da determinação de um tempo, durante o qual, o trem se aproxima do cruzamento mesmo que não tenha recebido a mensagem informando que o cruzamento está liberado. O tempo determinado na implementação foi 1.500 milisegundos. Este é tempo máximo em que

o trem poderá continuar em movimento sem receber a mensagem de liberação do cruzamento. Se o trem receber a mensagem de liberação do cruzamento antes que este tempo termine, o trem continuará seu trajeto sem parar, isto é, não perderá tempo, aguardando parado pela liberação do cruzamento. A função tempo é apresentada no quadro 44.

Quadro 44: A função tempo

```
tempo1(C, X, Y, W, PC1, PC2, PC3) ->
      io:format("X tempo: ~w  Y: ~w~n", [X,Y]),
      PidTimer1 = spawn (tcc,timer, [self(),1500,alarm]),
      PidMandaP1 = spawn (tcc, mandaP1,[self(),C, X, Y, W, PC1, PC2, PC3]),
      Xatual_a = andaAteCruzamento1 (PidTimer1,C, X, Y, W, PC1, PC2, PC3),
      esvazia_fila (),
      Xatual_a.
andaAteCruzamento1 (PidTimer1,C, X, Y, W, PC1, PC2, PC3) ->
      receive
            liberouPrivilegio1 ->
                  cancel (PidTimer1),
                  Xatual_a = X-5;
            alarm ->
                  receive
                        liberouPrivilegio1 -> true,
                        Xatual_a = X-5
                  end
            after 0 ->
                  deleta1 (C, X-5, Y),
                  cria1 (C, X, Y, W),
                  Xatual_a = andaAteCruzamento1 (PidTimer1,C, X+5, Y, W,
                  PC1, PC2, PC3)
      end,
      Xatual_a.
mandaP1(Pid,C, X, Y, W, PC1, PC2, PC3) ->
      io:format("X: ~w Y: ~w Pid do MandaP: ~w~n", [X, Y, self()]),
      if X == 185, Y == 220 ->
            p(PC1),
            Pid ! liberouPrivilegio1;
      X == 445, Y == 320 ->
            p(PC2),
            Pid ! liberouPrivilegio1;
      X == 185, Y == 420 ->
            p(PC3),
            Pid ! liberouPrivilegio1

      end.
```

A função tempo é responsável pela solicitação do cruzamento e pelo movimento de aproximação do trem do cruzamento durante o tempo determinado na implementação. Esta função foi tratada através do *timeout* independente apresentado no capítulo 4, no item 4.4.3.

Além dos conceitos de tempo real, também foram utilizados conceitos de processos concorrentes no tratamento do cruzamento. O cruzamento foi implementado como um semáforo binário. Isto significa que ou o cruzamento está liberado ou está ocupado. Esta parte do programa foi encontrada em [CAS1998] e foi transcrita e adaptada para o tratamento do cruzamento na malha ferroviária deste trabalho. O semáforo binário é mostrado no quadro 45.

Quadro 45: Semáforo binário

```
%%--------------------------------------------------------------------------
%%P(s): se s > 0, então s = s - 1; senão coloca na fila de espera
%%--------------------------------------------------------------------------

p(S) ->
       S ! {semaforo, p, self()},    % cont exige o nome do processo
       % espera pela mensagem prosseguir indicando para fila existente
       receive
              {semaforo, cont} ->
                 true
       end.

%%--------------------------------------------------------------------------
%%V(s): se a fila de espera não estiver vazia, desperta um; senão s = s + 1
%%--------------------------------------------------------------------------

v(S) ->
       S ! {semaforo, v}.

%%--------------------------------------------------------------------------
%%A função mensagem
%%--------------------------------------------------------------------------

mensagem(S, L) ->
       io:format("INICIO MENSAGEM S: ~w L: ~w~n", [S,L]),
       {NewS, NewL} =
              receive
              {semaforo, p, Pid} ->
                     io:format("MENSAGEM no P - L: ~w PID: ~w~n", [L, Pid]),
                     if
                            % P(s): se s > 0, então s = s - 1;
                            S > 0 ->
                                   Pid ! {semaforo, cont},
                                   {S - 1, L};
                            true ->
                                   % senão coloca na fila de espera
                                   {S, [Pid | L]}
                            end;

              % V(s): se a fila não estiver vazia
              {semaforo, v} when length(L) =/= 0 ->
                     [H|T] = L,
                     % desperta um;
                     H ! {semaforo, cont},
                     {S, T};
              % se a lista está vazia
              {semaforo, v} ->
                     io:format("LIBEROU S= ~w L= ~w~n", [S,L]),
                     % se s = s + 1
                     {S + 1, L}
              end,
       io:format("MENSAGEM L: ~w NewS: ~w, NewL: ~w~n", [L,NewS,NewL]),
       mensagem(NewS, NewL).
```

Fonte: [CAS1998]

A função tempo é responsável pelo envio de uma mensagem de solicitação do cruzamento para o semáforo. Caso o cruzamento não esteja ocupado, a solicitação é atendida. Caso o cruzamento esteja ocupado, o trem que solicitou o cruzamento entra em uma lista que é administrada como uma pilha. Isto significa que o último a entrar na lista é o primeiro a ser atendido. Quando o trem sai do cruzamento, é enviada uma mensagem para o semáforo, liberando o cruzamento para outros trens.

7.3 APRESENTAÇÃO DO PROTÓTIPO

Na figura 16 é apresentada a tela onde é visualizada a simulação do funcionamento da malha ferroviária.

Figura 16: Tela do protótipo com seu funcionamento

Esta tela contém as seguintes informações:

a) o botão Pare que quando acionado fecha a aplicação;
b) o contador de trens inicializados na malha ferroviária;
c) as cidades representadas pelas letras de A até L. Quando o botão de uma cidade é acionado, um trem é criado na ferrovia;
d) os retângulos que representam os trens na malha ferroviária;
e) as informações dos cruzamentos anotados com ocupado ou liberado.

8 CONSIDERAÇÕES FINAIS

Este trabalho foi, na sua essência, uma pesquisa da linguagem de programação Erlang e dos conceitos de tempo real e processos concorrentes. Durante esta pesquisa, verificou-se que é uma linguagem declarativa com facilidades para a representação de processos concorrentes e tempo real.

Um dos grandes desafios deste trabalho foi o aprendizado da linguagem de programação Erlang, de como utilizar seus mecanismos e produzir uma aplicação satisfatória através destes, levando em consideração os conceitos de concorrência e tempo real.

Ao final deste trabalho, foi possível a apresentação do protótipo de um controle de uma malha ferroviária, onde foram utilizados os conceitos de tempo real e processos concorrentes, sendo este um dos objetivos estabelecidos e portanto, alcançado.

Pode-se também concluir que a linguagem Erlang possui um ambiente gráfico de simples entendimento e implementação, possibilitando a criação de uma interface de fácil utilização para os usuários. Outro fato interessante a ser citado, é que a linguagem não apresentou um limite de processos simultâneos dentro de uma aplicação, o que demonstra o seu poder de representação dos conceitos de concorrência.

8.1 DIFICULDADES ENCONTRADAS

No decorrer do trabalho, foram encontradas algumas dificuldades, dentre elas pode-se citar:

a) o aprendizado da linguagem e de suas estruturas, por diferir dos moldes tradicionais de uma linguagem orientada por instruções seqüenciais. A linguagem em questão é declarativa, estilo Prolog;
b) literatura sobre a linguagem com erros, como pode ser visto no exemplo apresentado no livro [CAS1998], na página 85, quando mostra a descrição do modelo do semáforo binário;
c) não foi identificado usuários da linguagem no Brasil, dificultando uma possível interação, e também não foi encontrada literatura em português sobre a linguagem.

8.2 LIMITAÇÕES

O protótipo desenvolvido apresenta as seguintes limitações:

a) não foi dada exclusividade para um trem em uma linha. Esta foi tratada como uma via de mão dupla;
b) não foi efetuada uma avaliação do desempenho da linguagem;
c) no tratamento do cruzamento, não escolhe direção, segue a mesma de sua trajetória;
d) os trens vindos de uma mesma direção ficam parados na mesma posição quando precisam aguardar o cruzamento ser liberado;
e) não foi identificada uma função para administrar as cores dos trens. Foram somente utilizadas as cores básicas.

8.3 EXTENSÕES

Para finalizar, são apresentadas algumas sugestões de extensões em relação a pesquisa desenvolvida neste trabalho:

a) pesquisar e utilizar o modelo *Specification and Description Language* (SDL), sugerido em [ARM1993];
b) implementar o protótipo com exclusividade em relação às ferrovias;
c) desenvolver uma avaliação do desempenho da linguagem Erlang, fazendo uma comparação com outras linguagens;
d) desenvolver um algoritmo mais genérico para a implementação do protótipo da malha ferroviária;
e) desenvolver uma aplicação na área de telecomunicações utilizando Erlang, visto que a finalidade primeira quando da criação da linguagem, foi para esta área.

REFERÊNCIAS BIBLIOGRÁFICAS

[AMO1988] AMORIM, Cláudio Luiz; BARBOSA, Valmir Carneiro; FERNANDES, Edil Severiano Tavares. **Uma introdução à computação paralela e distribuída**. Campinas : UNICAMP, 1988.

[ARM1993] ARMSTRONG, Joe et al. **Concurrent programming in Erlang**. New Jersey : Prentice Hall, 1993.

[CAS1998] CASTRO, Maurice. **Erlang in real time**. Melbourne : Rmit, 1998.

[DEI1984] DEITEL, Harvey M. **An introduction to operating systems**. Austin : Addisson Wesley, 1984.

[ERI2000] ERICSSON Utvecklings AB. **Erlang**, 01/03/2000. Endereço Eletrônico: http:www.erlang.se. Data da consulta: 02/06/2000.

[FAR2000] FARINES, Jean-Marie; FRAGA, Joni da S.; OLIVEIRA, Rômulo S. de. **Sistemas de tempo real**. São Paulo : IME-USP, 2000.

[GHE1991] GHEZZI, Carlo e JAZAYERI, Mehdi. **Conceitos de linguagens de programação.** Rio de Janeiro : Campus, 1991.

[HAN1977] HANSEN, Per B. **The architecture of concurrent programs**. New Jersey : Prentice Hall, 1977.

[HAT1991] HATLEY, Derek J. PIRBHAI, Imtiaz A. **Estratégias para especificação de sistemas em tempo real**. São Paulo : MacGraw-Hill, 1991.

[KAV1992] KAVI, Krishna M. **Real-time systems abstractions, languages and design methodologies**. Washington : Braun-Brumfield, 1992.

[LAP1993] LAPLANTE, Phillip A. **Real-time systems design and analysis.** New York : Board, 1993.

[MAC1994] MACHADO, Francis B.; MAIA, Luiz Paulo. **Introdução à arquitetura de sistemas operacionais**. Rio de Janeiro : Livros Técnicos e Científicos, 1994.

[MAG1990] MAGALHÃES, Maurício Ferreira. **Sistemas de tempo real: anais do 8º congresso brasileiro de automática**. Belém : Universidade Federal do Pará, 1990.

[RIP1993] RIPPS, David L. **Guia de implementação para programação em tempo real**. Rio de Janeiro : Campus, 1993.

[SON1995] SON, Sang H. **Advances in real-time systems**. New Jersey : Prentice Hall, 1995.

[SEB2000] SEBESTA, Roberto W. **Conceitos de linguagens de programação.** Porto Alegre : Bookman, 2000.

[STA1988] STANKOVIC, J. **Misconceptions about real-time computing.** Los Alamitos : IEEE Computer Society Press, 1988.