UNIVERSIDADE FEDERAL DO MARANHÃO
CENTRO TECNOLÓGICO
CURSO DE PÓS-GRADUAÇÃO EM ENGENHARIA DE ELETRICIDADE

Um Controlador de Tráfego Ferroviário Adaptável Simulado com Máquinas de Estado Finitas

REGINALDO ISAIAS BRAGA

São Luis
2004

Um Controlador de Tráfego Ferroviário Adaptável Simulado com Máquinas de Estado Finitas

Dissertação de Mestrado submetida à Coordenação do curso de Pós-Graduação em Engenharia de Eletricidade da UFMA como parte dos requisitos para obtenção do título de mestre em Engenharia Elétrica na área de Ciência da Computação.

Por

REGINALDO ISAIAS BRAGA

Maio, 2004

Um Controlador de Tráfego Ferroviário Adaptável Simulado com Máquinas de Estado Finitas

REGINALDO ISAIAS BRAGA

DISSERTAÇÃO APROVADA EM 02 / 07 / 2004

Prof. Dr. Zair Abdelouahab
(Orientador)

Prof. Dr. Paulo Romero
(Membro da Banca Examinadora)

Profa. Dra. Maria Del Rosário Girardi
(Membro da Banca Examinadora)

Um Controlador de Tráfego Ferroviário Adaptável Simulado com Máquinas de Estado Finitas

MESTRADO

Área de Concentração: CIÊNCIAS DA COMPUTAÇÃO

REGINALDO ISAIAS BRAGA

Orientador: Prof. Dr. Zair Abdelouahab

Curso de Pós-Graduação

Em Engenharia de Eletricidade da

Universidade Federal do Maranhão

A meus filhos Thiago e Caio

“O supremo grau da sabedoria é ter sonhos suficientemente grandes para não perdê-los de vista, enquanto os perseguimos”.

William Faulkner

AGRADECIMENTOS

O caminho foi árduo e cheio de obstáculos, porém, reconheço com muita alegria, que não haveria de vencê-los sem a colaboração de várias pessoas. O meu muito obrigado...

A Deus, que está presente de forma contínua e renovadora em todos os momentos de minha vida, me conduzindo nos momentos mais difíceis e abençoando cada um dos meus passos.

A minha família, por saber que minhas conquistas serão sempre comemoradas como suas próprias conquistas. Especialmente, aos meus filhos (Thiago e Caio) e Esposa (Ângela): Assim como tenho certeza que sou uma referência para eles tenham certeza que são a razão de minha vida.

Ao Prof. Dr. Zair Abdelouahab, por ter me orientado na realização deste trabalho, pela confiança depositada e incentivo constante.

A Cia. Vale do Rio Doce pelo incentivo ao auto-desenvolvimento de seus empregados e pela confiança depositada que possibilitou a realização desta conquista.

Aos professores do CPGEE da UFMA, por toda competência e dedicação com que desempenham seu ofício.

Aos amigos do curso de mestrado, pelas experiências compartilhadas e todo apoio nos momentos de maior dificuldade.

Por fim, agradeço a todos aqueles que direta ou indiretamente contribuíram para a realização deste trabalho.

RESUMO

Este trabalho trata da implementação de um simulador de controle de tráfego ferroviário cujos requisitos funcionais possam ser adaptáveis, isto é, modificados, complementados ou mesmo redefinidos, através de uma abordagem baseada em máquinas de estado finitas.

Dependendo de seus requisitos, um controlador de tráfego ferroviário pode assumir características simples ou críticas em termos de segurança ou tolerância a falhas. A modelagem baseada na abordagem de máquinas de estado provê as condições para a adaptabilidade de alguns requisitos funcionais. Esta característica é melhor enfatizada quando usamos ferramentas automatizadas de definição de máquinas de estado, que permitem a fácil integração com o restante da aplicação.

É apresentada uma discussão detalhada sobre as considerações de segurança que devem ser seguidas em um projeto de missão e segurança crítica, onde é dada ênfase nas questões relacionadas à segurança do software. Uma metodologia e ferramenta (*SpecTRM-RL*) que a implementa, derivada de pesquisas recentes em software de segurança é mostrada. Estas considerações desempenham um papel fundamental no projeto e análise de requisitos em sistemas de controle metro-ferroviários. Por serem estes classificados como de "missão e segurança críticas" e principalmente por estarem migrando gradativamente de dispositivos com lógica de controle baseada em relês para controladores microprocessados onde no todo ou em parte os requisitos de segurança são implementados por software.

Também de fundamental importância no contexto deste trabalho são as técnicas de tolerância a falha tanto em hardware como em software. Uma seleção das mais usuais em sistemas que demandam elevada confiabilidade foi detalhada com objetivo de enfatizar a aplicabilidade no estudo de caso de um simulador de intertravamento ferroviário.

ABSTRACT

This thesis deals with implementation of a railway control interlocking simulator whose functional requirements can be adaptive, that is, modified, complemented or same redefined, through an approach based on Finite State Machines.

Depending on its requirements, a controller of rail traffic can assume simple or critics characteristics in terms of safety or fault tolerance. The modeling based on the Finite State Machine approach provides the conditions for the adaptability of some functional requirements. This characteristic is emphasized better when we used automated tools of definition of state machines, which allow the easy integration with the remaining of the software application.

A detailed discussion about safety's considerations in design of mission and safety critical systems is presented. Emphasis is given in the subjects related to the safety software. A methodology and tool (*SpecTRM-RL*) that implements it, derived of recent researches in safety software, is shown. These considerations are fundamental in the design and requirements analysis in railway's interlocking control systems. These kinds of systems can be classified as "mission and safety critical" and they are progressively migrating for relay based controllers to microprocessor-based controllers where in the whole or partly safety's requirements are implemented by software.

Fault tolerance techniques either in software or hardware are fundamental in the context of this work. A selection of the most usual in systems that demand high reliability was detailed with the goal of emphasizing its potential application in the case study of a railway control-interlocking simulator.

SUMÁRIO

SUMÁRIO ..... **X**

LISTA DE TABELAS ..... **XII**

LISTA DE FIGURAS ..... **XIII**

ABREVIATURAS E SÍMBOLOS ..... **XIV**

1. INTRODUÇÃO **15**

1.1 MOTIVAÇÃO 16
1.2 OBJETIVOS 18
1.3 ORGANIZAÇÃO DO TRABALHO 19

2. ASPECTOS DE SEGURANÇA CRÍTICA RELACIONADOS AO SOFTWARE **21**

2.1 RISCOS ENVOLVIDOS EM SISTEMAS AUTOMATIZADOS 21
2.2 CONFIABILIDADE X SEGURANÇA 23
2.3 UMA ABORDAGEM PARA SOFTWARE DE SEGURANÇA CRÍTICA 24
2.3.1 Análise de Risco em Software (Software Hazard Analysis) *29*
2.3.2 Um Exemplo de Ferramenta Centrada em Segurança (SpecTRM-RL) *32*
2.4 CONSIDERAÇÕES FINAIS 35

3. TOLERÂNCIA A FALHAS EM SISTEMAS COMPUTACIONAIS **37**

3.1 CONCEITOS 37
3.1.1 Dependabilidade *37*
3.1.2 Redundância *38*
3.1.3 Terminologia relativas a Falhas, Erros e Defeito *39*
3.1.4 Abordagens Usuais na Aplicação da Tolerância a Falhas *41*
DETECÇÃO DE ERROS 42
CONFINAMENTO 42
RECUPERAÇÃO 42
TRATAMENTO 42
3.2 TÉCNICAS DE TOLERÂNCIA A FALHAS EM HARDWARE 43
3.2.1 Replicação Ativa ou Redundância Dinâmica *44*
3.2.2 Replicação Passiva *45*
3.3 TÉCNICAS DE TOLERÂNCIA A FALHAS EM SOFTWARE 46
3.3.1 Técnicas de Programação Versão Única (Single Version Programming) *47*
3.3.1.1 Estruturação do Software e Atomicidade de Ações 47
3.3.1.2 Detecção de Erros 48
3.3.1.3 Tratamento de Exceções 50
3.3.1.4 Salvaguardas (*Checkpoints*) e Reinicialização 50
3.3.1.5 Pares de Processos 51
3.3.2 Programação em Versões Múltiplas *52*
3.3.2.1 Blocos de Recuperação 54
3.3.2.2 Programação *N-Version* 55
3.3.2.3 Blocos de Recuperação por Consenso 57
3.3.3 Injeção de Falhas no Software para Avaliação da Tolerância a Falhas *58*
3.4 REDUNDÂNCIA MODULAR TRIPLA *(TMR)* 60
3.4.1 Arquitetura TMR Distribuída *63*
3.4.2 Votadores *64*
3.4.2.1 Votadores implementados por software (Algoritmos de Seleção de Saídas) 66
3.4.2.2 Votadores Implementados por Hardware 68
3.5 CÓDIGOS DE REDUNDÂNCIA CÍCLICA (CRC) 69
3.5.1 Considerações Básicas *70*
3.5.2 Teoria de Operação e Representação Polinomial *71*

3.5.3 Considerações Adicionais ... *73*
3.6 DESAFIOS VIGENTES ... 73
3.7 CONSIDERAÇÕES FINAIS ... 74

4. MODELAGEM DE UM CONTROLADOR DE TRÁFEGO FERROVIÁRIO ... 76

4.1 CONTROLE DE TRÁFEGO FERROVIÁRIO ... 76
4.1.1 Definições Básicas e Fundamentais ... *77*
4.1.2 Controle de Tráfego Centralizado ... *79*
4.2 ESPECIFICAÇÃO DE UM MODELO DE CONTROLADOR DE TRÁFEGO FERROVIÁRIO ... 83
4.2.1 Variáveis de Entrada e Saídas ... *84*
4.2.2 Estados e Transições ... *89*
4.2.3 Critérios de Completeza ... *93*
4.3 CONSIDERAÇÕES FINAIS ... 97

5. (ESTUDO DE CASO) - IMPLEMENTAÇÃO DO CONTROLADOR DE TRÁFEGO FERROVIÁRIO SIMULADO ... 99

5.1 MODELO COM MÁQUINAS DE ESTADO FINITAS ... 99
5.2 ARQUITETURA FÍSICA E LÓGICA ... 100
5.3 UM MODELO ADAPTÁVEL COM MÁQUINAS DE ESTADO ... 103
5.4 FUNCIONALIDADES ... 104
5.5 CONSIDERAÇÕES FINAIS ... 110

6. CONCLUSÕES ... 111

6.1 TRABALHOS FUTUROS ... 113

REFERÊNCIAS BIBLIOGRÁFICAS ... 115

ANEXO A ... 121

MÁQUINAS DE ESTADO FINITAS (*FSM FINITE STATE MACHINE*) ... 121
DEFINIÇÕES BÁSICAS ... 121
DEFINIÇÃO FORMAL DE MÁQUINAS DE ESTADO FINITAS ... 121
IMPLEMENTANDO UMA *FSM:* COMPILADOR DE MÁQUINAS DE ESTADO (*SMC*) ... 123
INTEGRANDO UMA *FSM* A UMA CLASSE JAVA ... 125

ANEXO B ... 127

REQUISITOS GERAIS DE SEGURANÇA PARA UM SISTEMA DE CONTROLE DE METRO-FERROVIÁRIO ... 127
CONCEITOS FUNDAMENTAIS DE SEGURANÇA METRO-FERROVIÁRIA ... 127
DETALHAMENTO DOS REQUISITOS GERAIS DE SEGURANÇA ... 128

LISTA DE TABELAS

Tabela 3-1 Fases de Anderson e Lee.....42
Tabela 4-1 Predicados de Entrada.....88
Tabela 4-2 Predicados de Saída.....89
Tabela 4-3 Descrição dos Estados do Controlador de Tráfego Ferroviário.....90
Tabela 4-4 Transição de Estados.....91
Tabela 4-5 Transição de Estados (cont.).....91
Tabela 4-6 Transição de Estados (cont.).....92
Tabela 4-7 Transição de Estados (cont.).....92
Tabela 4-8 Transição de Estados (cont.).....93
Tabela 5-1 Booleanas de Intertravamento.....103

LISTA DE FIGURAS

Figura 2-1 Análise de Risco no Ciclo de Vida de um Sistema 26
Figura 2-2 Metodologia de Projeto Dirigida a Segurança [LEVESON, 1995] 28
Figura 2-3 Exemplo de Representação de Máquina de Estado 31
Figura 2-4 Modelo de Processo [LEVESON, 1995] 31
Figura 2-5 SpecTRM-RL (Estrutura Lógica) e Funcionalidades 33
Figura 2-6 SpecTRM –RL Intent Specification (Estrutura Hierárquica) [LEVESON, 1995] 35
Figura 3-1 Modelo de 3 Universos : Falha, Erro e Defeito 40
Figura 3-2 Representação Lógica de Pares de Processo 52
Figura 3-3 Representação Lógica de um Bloco de Recuperação 55
Figura 3-4 Representação Alternativa de um Bloco de Recuperação 55
Figura 3-5 Modelo de Programação N-Versões 56
Figura 3-6 Bloco de Recuperação por Consenso 58
Figura 3-7 Redundância Modular Tripla (*TMR*) 60
Figura 3-8 Redundância Modular N (NMR) 61
Figura 3-9 Redundância Modular Tripla (TMR) Triplicada 62
Figura 3-10 Redundância Modular Tripla (TMR) em Múltiplos Estágios 62
Figura 3-11 Redundância Modular Tripla Distribuída em Rede Local 64
Figura 3-12 Votador em Processador Compartilhado 65
Figura 3-13 Circuitos Lógicos de Votadores Simples 69
Figura 3-14 Tabelas Verdade (Aritmética Módulo 2) 71
Figura 4-1 Controle de Tráfego Metro-Ferroviário (Subsistema de Via) 79
Figura 4-2 Controle de Tráfego Metro-Ferroviário (Subsistema de Bordo) 79
Figura 4-3 - Arquitetura de um Centro de Controle Metro-Ferroviário 81
Figura 4-4 Centro de Controle Centralizado - Modelo Conceitual 82
Figura 4-5 Modelo Conceitual de um Sistema de Controle e Supervisão 83
Figura 4-6 Fluxo de Dados do Módulo Controlador (Predicados de Entrada e Saída) .87
Figura 5-1 Módulo Controlador (Máquina de Estado) 99
Figura 5-2 Módulo Controlador (Arquitetura Lógica) 102
Figura 5-3 Módulo Controlador (Arquitetura Física) 102
Figura 5-4 Característica de Adaptabilidade 103
Figura 5-5 Interface Homem-Máquina do módulo Gerador de Eventos 104
Figura 5-6 Eventos de Inicialização e Fim de Inicialização do Controlador 106
Figura 5-7 Inicialização com Timeout Expirado 107
Figura 5-8 Requisição de Alinhamento de Rota 108
Figura 5-9 Eventos de Ocupação dos Trechos de uma Rota 109
Figura 5-10 Transitando para o Estado Inseguro 110
Figura A-1 Implementando FSM com State Machine Compiler (SMC) 124

ABREVIATURAS E SÍMBOLOS

AEI	*Automatic Equipment Identification*
ATC	*Automatic Train Control*
ATO	*Automatic Train Operation*
CAD	*Computer Aided Design*
CASE	*Computer Aided Software Engineering*
CBTC	*Communication Based Train Control*
CCITT	*Comité Consultatif International Telépfonique et Telégraphique*
CRC	*Cyclic Redundancy Code*
CTC	*Centralized Traffic Control*
CVRD	*Cia. Vale do Rio Doce*
ECC	*Error Correction Code*
EFC	*Estrada de Ferro Carajás*
FCS	*Frame Check Sequence*
FEC	*Front End Communication*
FSM	*Finite State Machine*
LAN	*Local Area Network*
MIT	*Massachusetts Institute of Technology*
MTBF	*Mean Time Between Failures*
MTTF	*Mean Time to Fail*
RGS	*Requisitos Gerais de Segurança*
SFI	*Software Fault Injection*
SMC	*State Machine Compiler*
SpecTRM-RL	*Specification Tools Requirements Methodology Requirements Language*
TCAS	*Traffic Alert and Collision Avoidance System*
TMR	*Triple Modular Redundancy*
UCLA	*University of California, Los Angeles*
UML	*Unified Modeling Language*

1. Introdução

Sistemas críticos em geral, são aqueles cujos requisitos de funcionamento exigem considerações diferenciadas relativas a diversos fatores tais como: Tempo de resposta, confiabilidade, disponibilidade e em casos específicos segurança. Quando esta última característica é fator essencial o sistema pode ser caracterizado como *sistema de segurança crítica*. Para se alcançar os níveis de segurança desejados, considerações adequadas de projeto são necessárias.

Os sistemas de controle com as características acima descritas, devem ter alto grau de *dependabilidade*, terminologia que significa elevado grau de confiança no funcionamento. Uma discussão detalhada sobre esta característica será apresentada no Capítulo 3.

A razão da existência de características diferenciadas no projeto de sistemas de controle e segurança críticos é a minimização da probabilidade da ocorrência de falhas cujas conseqüências possam ser catastróficas no sentido de perdas materiais ou humanas.

Podemos citar alguns exemplos de sistemas que em suas concepções de projeto, requisitos de controle e segurança críticos estão fortemente presentes: Sistema de controle de tráfego aéreo, plantas nucleares, químicas, controle metro-ferroviário, dentre outros.

São várias as técnicas aplicadas para se atingir níveis aceitáveis de *dependabilidade*, em geral técnicas de tolerância a falhas são largamente utilizadas, em ambos os aspectos: Hardware e Software. Muitas destas técnicas serão abordadas em detalhes ao longo do presente trabalho.

A análise de segurança de sistemas de missão e segurança crítica, requer considerações diferenciadas (rigorosas), em especial quando relacionadas ao software de controle. Pois apesar de controverso, é bastante difundida a idéia de que a segurança de um sistema é dependente não apenas do software ou hardware de controle, mas também do contexto no qual o sistema é submetido em condições reais. Essencialmente, isto quer dizer que não basta um software de controle "*estritamente correto*", para que o sistema possa ser considerado *seguro* sob todos os aspectos. Estes conceitos estão

baseados no trabalho de [LEVESON, 1986], [LEVESON, 1995] e [SAFEWARE, 2001].

O presente trabalho se refere a proposta de implementação de um controlador de tráfego ferroviário simulado adaptável através da abordagem de máquinas de estado finitas, em que são considerados aspectos de tolerância a falhas e de especificação de sistemas de segurança crítica. De fato o controle de tráfego metro-ferroviário constitui uma aplicação em que os conceitos aqui apresentados relativos a sistemas de missão crítica são bastante relevantes e aplicáveis. Por questões de simplicidade não se pretende explorar todos os aspectos de um sistema de controle de tráfego ferroviário real, mas contribuir com uma ferramenta que constitua uma aplicação prática de conceitos relevantes explorados no trabalho. Essencialmente o modelo básico de controle metro-ferroviário é baseado no trabalho de [CAMARGO JR, 1996] cujo resultado é usado na implementação do simulador. Uma técnica largamente utilizada em arquiteturas físicas de sistemas tolerantes a falhas denominada TMR (*Triple Modular Redundancy*) foi explorada. Isto facilitou a exemplificação de aspectos importantes de tolerância a falhas que o projeto de software do simulador necessita considerar.

Outro aspecto bastante relevante explorado, é o uso de ferramentas específicas para a implementação do modelo de controle de tráfego metro-ferroviário. Ferramentas como uma linguagem de especificação de requisitos e modelagem denominada SPECTRM-RL *(Specification Tools Requirements Methodology – Requirements Language)*, a qual fornece mecanismos adequados para especificação de requisitos, formalização, simulação e validação de aspectos de segurança crítica [LEVESON, 2000]. Uma outra ferramenta, denominada SMC *(State Machine Compiler)* foi usada na implementação do núcleo do controlador de tráfego metro-ferroviário simulado, o qual é baseado na abordagem de máquinas de estado finitas [MARTIN, 1998].

Segue uma caracterização detalhada das idéias que motivaram este trabalho, bem como sua organização estruturada.

1.1 Motivação

O controle de tráfego ferroviário em malhas de densidade média e alta, exige níveis de segurança no controle do processo (licenciamento e movimentação de trens) que demanda significativa automação. Este tipo de controle deve ter

características não dependentes de fatores estritamente manuais e que minimizem a probabilidade de ocorrência de falhas decorrentes de fatores humanos e dos próprios componentes de hardware e software do sistema.

Nos últimos anos os sistemas de controle de tráfego ferroviários nos quais estão baseados os requisitos de segurança crítica, estão migrando de forma irreversível de controladores discretos ou baseados em lógica a relês para sistemas microprocessados [VERNAZZA, 1990], [RAO, 1987]. Grande parte dos requisitos de segurança, agora é baseada em software. Independentemente das vantagens vindas da atualização tecnológica, em aspectos relacionados a custos de implantação, manutenção e gerenciamento, o processo de verificação e garantia de segurança tornou-se dramaticamente complexo. O software é fator chave dessa complexidade.

Técnicas de redundância são comumente utilizadas no projeto de controladores de tráfego ou em geral de sistemas de controle que exigem alto grau de *dependabilidade.* Tais técnicas são baseadas nos conceitos de tolerância a falhas. O objetivo é mascarar falhas, de tal forma que as funções desses sistemas continuem disponíveis para o cumprimento de sua missão, mesmo na presença dessas falhas.

A redundância modular tripla (***TMR***) é um exemplo das muitas técnicas de redundância utilizadas, tendo aplicações reais em sistemas de controle metro-ferroviário. Neste tipo de arquitetura podemos ver aplicações conjugadas de vários conceitos de tolerância a falha relacionadas a hardware e software. A *Replicação* de hardware e software e a abordagem da *diversidade* de projeto aplicada ao software, são exemplos relevantes. Estes conceitos e técnicas serão detalhados no Capítulo 3. [AVIZIENIS, 1976] e [POMALES, 2000] são referências importantes.

Os problemas relacionados com a análise, verificação e garantia de segurança em sistemas críticos fortemente baseados em computadores, tiveram sua complexidade significativamente aumentada. O uso de metodologias emergentes que suportam e enfatizam o foco na segurança estão influenciando fortemente o projeto de sistemas desta natureza. O uso efetivo destas metodologias e a verificação de sua efetividade são fatores motivadores. Questões relativas à *adaptabilidade* de requisitos funcionais de software, constituem fatores relevantes no contexto deste trabalho. A possibilidade de se complementar ou modificar regras funcionais num ambiente de simulação possibilita a realização de ensaios e análises que podem cobrir as especificidades de uma aplicação como um controlador de intertravamento ferroviário.

Motivação adicional vem de desafios relacionados com a grande dificuldade de se verificar e garantir a segurança relacionada a componentes de software. Bem como sua adaptabilidade, isto é, como tornar o software adaptável a diferentes cenários de requisitos funcionais. Estas questões têm ocupado um grande número de pesquisadores no desenvolvimento de metodologias voltadas à especificação de requisitos e modelagem em "*Safety Software*". Ferramentas automatizadas CASE *(Computer Aided Software Engineering)* que suportam estas metodologias têm surgido e melhorado substancialmente o processo de criação de software de tempo real para processos críticos como o controle de tráfego ferroviário. A aplicação de resultados já consolidados ou emergentes relacionados com tolerância a falhas e software de segurança em ramos da indústria como o transporte ferroviário, constitui relevante fator motivador. Isto pode ser visto como uma contribuição numa área da indústria onde a tecnologia aplicada na solução de problemas específicos, tem se mostrado proeminente nos últimos anos. A introdução de ferramentas de simulação adaptáveis também é relevante neste segmento.

1.2 Objetivos

Este trabalho tem como principal objetivo a implementação de um simulador de intertravamento ferroviário, cujos requisitos funcionais (regras de negócio e de segurança) sejam adaptáveis através da modelagem com máquinas de estado finitas. Também é objetivo a aplicação de técnicas de tolerância a falhas por software nesta implementação.

Outro importante objetivo é discorrer detalhadamente sobre conceitos e técnicas de tolerância a falhas aplicadas a software de controle. O controle de tráfego ferroviário pode ser caracterizado como um processo crítico, logo os aspectos de *dependabilidade* são bastante relevantes, como também os de *segurança crítica*.

Uma discussão detalhada de considerações relevantes sobre os aspectos de segurança crítica relacionados ao software que implementa requisitos desta natureza, é outro objetivo do trabalho, bem como a apresentação de uma ferramenta de modelagem dirigida à especificação e desenvolvimento de sistemas de segurança. Esta ferramenta automatizada é baseada numa metodologia resultante de pesquisas recentes em software de segurança e denomina-se SpecTRM-RL (*(Specification Tools and Requirements Methodology – Requirements Language)*).

Por ser o conceito de máquinas de estado finitas (*FSM - Finite State Machine*) fundamental na modelagem de software de controle tolerante a falhas. Tem-se por objetivo desenvolver o núcleo do modelo do controlador de tráfego ferroviário simulado utilizando esta abordagem. Para tanto se pretende basear a implementação da máquina de estado no compilador SMC (*State Machine Compiler*).

Com o uso de máquinas de estado na modelagem do simulador de intertravamento ferroviário pretende-se dotar a implementação com um nível inicial de adaptabilidade a certos requisitos funcionais de controle.

1.3 Organização do Trabalho

Além das seções introdutórias descritas no Capítulo 1, o trabalho está organizado da seguinte forma. O Capítulo 2 descreve sobre os aspectos de segurança crítica relacionados ao software. São enfatizados os problemas referentes à complexidade inerente do software e seus reflexos na análise de segurança de sistemas que no todo ou em parte tem requisitos de segurança críticas implementados em software. Questões ainda em aberto sobre tais aspectos, que constituem desafios interessantes de pesquisa, também são discutidos neste capítulo.

No Capítulo 3 são descritos com razoável grau de detalhamento os fundamentos dos mecanismos de tolerância a falhas, tanto de hardware como de software. Relativo ao hardware, é enfatizado as técnicas de *replicação ativa* e *passiva*, todas baseadas no conceito de *redundância*. Relativo a tolerância a falhas de software, as técnicas de programação em versão simples ou múltiplas *(Single Version e N-Version)* respectivamente, são exploradas. Conceitos relacionados a redundância por software, como o do principio da diversidade de projeto o qual fundamenta algumas teorias ainda controversas sobre confiabilidade do software, são discutidos. Ênfase é dada na descrição de uma técnica de redundância específica TMR (*Triple Modular Redundancy*) devido a sua aplicabilidade em sistemas reais e ao estudo de caso objeto deste trabalho.

O Capítulo 4 descreve um modelo de controlador de tráfego ferroviário baseado no trabalho de [CAMARGO JR., 1996]. Neste capítulo são apresentados conceitos fundamentais sobre sistemas de controle de tráfego metro-ferroviário, caracterizando-o como um importante segmento da indústria em que as questões como a garantia de segurança é fundamental.

Este modelo foi utilizado como núcleo base no estudo de caso descrito no Capítulo 5, que implementa um simulador adaptável de um intertravamento ferroviário baseado na abordagem de máquinas de estado.

As conclusões relacionadas ao trabalho são apresentadas no Capítulo 6. São enfatizadas além dos resultados obtidos, as questões fundamentais que motivaram o trabalho, bem como as questões abertas que podem servir de base para trabalhos futuros. Com relação ao estudo de caso, uma perspectiva evolutiva é vislumbrada, com diversas melhorias que podem ser aplicadas ao projeto do controlador de tráfego ferroviário simulado.

Adicionalmente são descritos no anexo A, aspectos relevantes da teoria de máquinas de estado finitas, as quais foram aplicadas neste trabalho. Neste anexo, também pode ser encontrado detalhes do compilador de máquinas de estados SMC, em cuja abordagem está baseada a implementação em linguagem Java do controlador de tráfego ferroviário simulado. Por fim, no anexo B é apresentada uma lista dos principais requisitos gerais de segurança de um sistema de controle de tráfego metro-ferroviário, segundo [FDTE 542.401. AS.001].

2. Aspectos de Segurança Crítica Relacionados ao Software

Este Capítulo descreve as principais questões relacionadas com software de segurança (*Safety Software*). Enfatizando os problemas que surgiram na área de análise de segurança de sistemas de controle automatizados, na medida em que estes se tornaram cada vez mais complexos. Conseqüência dessa complexidade é a crescente automação dos sistemas de controle de processos, nos quais requisitos de segurança são implementados.

A maioria dos acidentes ocorridos em sistemas que envolvem requisitos de segurança crítica, não são resultados de princípios científicos não conhecidos, mas nas falhas em se aplicar práticas padrões de engenharia bem conhecidas [LEVESON, 1995].

2.1 Riscos Envolvidos em Sistemas Automatizados

A crescente automação em sistemas de controle complexos tem levado a mudanças significativas no papel dos operadores e também a novos tipos de erros humanos induzidos pela tecnologia aplicada [LEVESON, 1995].

Uma interessante analogia pode ser usada para representar um software de computador. Se pensarmos num computador como uma máquina de propósito geral, este mesmo computador executando um software específico pode ser visto como uma máquina de propósito especial ou específica. Desta forma quando se muda o software deste computador o mesmo é transformado numa nova máquina de propósito específico. Isto significa que para cada versão diferente de um software, aquele computador de propósito geral é transformado numa nova máquina que realiza uma tarefa específica [SAFEWARE, 2001].

O software permite ao projetista se concentrar nas tarefas a serem executadas sem se preocupar como as mesmas serão feitas fisicamente. Isto de fato é uma característica poderosa, um grande benefício com o qual os projetistas de sistemas de controle baseados em tecnologia de dispositivos discretos não dispunham no inicio da era de automação. Entretanto a facilidade de se modificar o software e consequentemente de forma abstrata de se modificar o propósito de uma máquina, ao mesmo tempo em que se constitui num grande benefício, é também a fonte dos riscos em sistemas de segurança crítica, baseados em computadores.

O poder e a flexibilidade do software em sistemas computacionais, de fato constituem o seu maior problema no contexto de garantia de segurança *(safety software)*. Sem limites físicos que possam ajudar a manter as soluções simples, as mesmas tendem a uma complexidade que favorece a violação de requisitos e restrições de segurança. Na ausência de limites físicos a complexidade do software também favorece a ausência de disciplina no projeto, codificação e modificações do mesmo, ocasionando sérias implicações em sistemas de segurança critica automatizados.

A análise de segurança do software se baseada apenas em métodos puramente matemáticos (formais), como Z, VDM, Statecharts [WILLIAMS, 1994], é muito problemática devido à complexidade inerente ao software. Em sistemas reais a quantidade de estados é enorme. Provas de corretude formais em geral apresentam as seguintes inconveniências:

- Podem ser iguais ou maiores em tamanho que o próprio código fonte
- São em geral mais difíceis de formular que o código
- São mais difíceis de entender que o próprio código

Outra abordagem para modelar e provar corretude é através de métodos de medidas ou avaliação de qualidade do software, mas infelizmente ainda não se chegou a formas verdadeiramente efetivas de se realizar tais medidas [LEVESON, 1995].

Para se garantir a segurança do software, no processo de verificação poderia ser usada tanto a abordagem de "*blackbox*" como a de "*whitebox*". Na primeira todas as condições de entradas devem ser verificadas. Não somente as condições válidas, mas também todas as condições de entradas impróprias. Na segunda abordagem, a lógica do programa é verificada através de testes exaustivos de todos os caminhos possíveis no fluxo de execução dos algoritmos. Na realidade ambas as abordagens são impraticáveis, devida a quantidade astronômica de casos.

Mesmo que hipoteticamente fosse possível proceder as verificações mencionadas no parágrafo anterior, ainda assim não seria possível garantir que o software seria seguro. Em essência um software pode estar livre de erros e mesmo assim não ser seguro num contexto mais amplo. A verificação lógica de algoritmos pode identificar erros nos caminhos existentes, no entanto não seria capaz de identificar a ausência de caminhos necessários a tornar o software seguro. A segurança do software

não estaria garantida devido a critérios de completeza, isto é, devido a não completeza de sua especificação de requisitos.

O problema primário de segurança *(safety)* em sistemas computacionais é a ausência de restrições apropriadas ao projeto. A tarefa principal do engenheiro de segurança é identificar as restrições de projeto necessárias à segurança e garantir que as mesmas sejam efetivamente consideradas no projeto do sistema e no software [LEVESON, 1995].

2.2 Confiabilidade x Segurança

Existe numerosa literatura sobre teoria de confiabilidade em sistemas computacionais, no entanto, em sua maioria é relacionada com os aspectos de hardware. Ainda é controversa a aplicação dos conceitos de confiabilidade em segurança de sistemas baseados em software. Procura-se a seguir estabelecer diferenças claras, mesmo que baseadas em conjecturas, a cerca das diferenças entre os conceitos inerentes a segurança crítica de software e confiabilidade. O objetivo é fundamentar a assertiva que segurança e confiabilidade são conceitos distintos e que não devem ser confundidos no sentido que induza se considerar um software *confiável* como *seguro*, o que não é verdade.

A teoria da confiabilidade lida com conceitos como falhas e defeitos *(ver definições no Capítulo 3)* e essencialmente com a predição de indicadores como tempo médio entre falhas (*MTBF – Mean Time Between Failures*) e tempo médio para falhar (*MTTF - Mean Time to Fail*) dentre outros. A idéia básica é conseguir estimar tais indicadores a partir do conhecimento do comportamento da taxa de falhas de componentes específicos, a qual pode ser modelada através de uma distribuição probabilística. Esses conceitos são aplicáveis a componentes físicos, os quais podem ser acometidos de falhas de fabricação ou mesmo a fadiga por diversos tipos de desgastes [BRAGA, 1997], [SAMPAIO, 2000], [LEWIS, 1987]. O software é um produto intelectual de projeto, não possui características físicas das quais se possam modelar estatisticamente sua *taxa de falhas*. Alguns autores divergem até no fato de ser apropriado ou não dizer se o software falhou, tomando por base a definição de *falha*. De fato se considerarmos o comportamento do software determinístico, sob as mesmas circunstâncias, a sua execução será sempre da mesma forma [SAFEWARE, 2001], [ABBOTT, 1990].

Sem dúvida alguma as técnicas usuais aplicadas em confiabilidade de sistemas computacionais tais como: *Redundância Paralela, Replicação Ativa e Passiva* e outras discutidas no Capítulo 3 que trata de tolerância a falhas, são benéficas e essenciais ao projeto de sistemas de missão crítica. Em essência todas essas técnicas objetivam incrementar a confiabilidade dos componentes críticos de um sistema reduzindo sua taxa de falhas. Entretanto com relação à segurança de processos críticos cujo controle é baseado em software, as técnicas de confiabilidade não são suficientes e muitas vezes nem aplicáveis na garantia de segurança, pois estaria se assumindo que todos os acidentes neste contexto seriam resultantes de falhas de componentes físicos.

A maioria dos acidentes em sistemas críticos controlados por software, os quais foram investigados seriamente, foram resultantes de erros ou omissões nas especificações de requisitos ou então devido à operação fora dos limites especificados. Tais acidentes não foram devidos a falta de confiabilidade de componentes físicos. Acidentes também podem ocorrer devido a interações de componentes todos operando dentro de suas especificações operacionais [SAFEWARE, 2001]. Acidentes causados por erros na implementação do software, não poderiam ser evitados pela simples adoção de redundância. Neste caso os erros não são causados por falhas aleatórias, pois diferentes cópias do mesmo software apresentarão o mesmo defeito. Outro fator de considerável relevância é que a aplicação de redundância no software aumentará sua complexidade, a qual tornará a questão de segurança também mais complexa de se garantir.

De acordo com pesquisas recentes em "*Safety Software"* baseadas em casos reais de acidentes cujas causas estavam relacionadas com softwares de controle crítico. Concluiu-se que a reutilização de componentes de software é uma das causas mais comuns destes acidentes. Exemplos significativos de casos reais são: Ariane 5, U.K. ATC (United Kingdom Air Traffic Control) [LEVESON, 1986] [SAFEWARE, 2001].

Conclusivamente, a garantia de *software seguro* num contexto sistêmico, requer muito mais que simplesmente ter um *software correto*.

2.3 Uma Abordagem para Software de Segurança Crítica

Os aspectos de *segurança em software* constituem parte essencial de uma abordagem mais abrangente relacionada com os aspectos de *segurança do sistema* a ser desenvolvido o qual efetivará o controle de um processo crítico.

Podemos considerar a segurança do sistema *(System Safety)* como uma abordagem sistemática, disciplinada e planejada. Tendo como objetivo primordial a prevenção ou redução a níveis aceitáveis dos riscos de ocorrência de acidentes no decorrer do ciclo de vida de um sistema. Um dos objetivos primários de uma abordagem de segurança de sistemas é o *gerenciamento dos riscos*. Consideremos *risco* como um estado do sistema o qual conjuntamente com condições específicas no ambiente desse sistema poderá levar a um acidente.

Em geral as metodologias que sistematizam uma abordagem para o desenvolvimento de sistema de segurança devem trazer um conjunto de ferramentas automatizadas ou não que incluam facilidades de análise e projeto centradas no gerenciamento de riscos. Em diversas áreas de aplicação e engenharia existem padrões e normalizações que recomendam boas práticas de segurança no desenvolvimento de sistemas afins, como exemplo as recomendações MIL-STD-882.

O gerenciamento de riscos é comumente baseado num processo que envolve pelo menos as seguintes fases: *Identificação, avaliação, eliminação e controle*. Pesquisas têm mostrado que uma boa abordagem para segurança de sistemas é aquela que é incorporada ao próprio processo de desenvolvimento do sistema, isto é, o projeto do sistema já deve ter em sua concepção inicial considerações que objetivem as questões de segurança. O estabelecimento dos requisitos de segurança nas fases iniciais do projeto é muito mais efetiva que nas fases finais. Muitos esforços em construir sistemas seguros falham devido à avaliação dos requisitos de segurança serem feitos na fase final do projeto ao invés do início [LEVERSON, 1995].

A análise de riscos constitui uma etapa primordial no ciclo de vida de um sistema de segurança crítica. A mesma deve ser baseada na premissa de que um sistema deve ser considerado como um todo e não apenas como um conjunto de componentes isolados. Os aspectos de segurança sistêmicos devem considerar os riscos numa perspectiva mais ampla do que a questão de falhas de componentes. Conforme explicado na *Seção 2.2* confiabilidade não garante segurança em sistemas automatizados. A maioria das causas de acidentes investigados em processos controlados por sistemas computacionais foi resultante da complexidade das interações entre os diversos componentes do sistema. Não devido às falhas específicas destes componentes.

Qualquer que seja a abordagem para lidar com a segurança no desenvolvimento de sistemas, a mesma deve considerar a análise de riscos e o projeto com requisitos para sua eliminação ou controle. De fato em aplicações reais de engenharia, como por exemplo, o controle de tráfego metro-ferroviário, por vezes a eliminação de alguns riscos é economicamente inviável, então se adota a estratégia de controle do mesmo. Tal controle pode ser implementado através de estratégias de minimização da possibilidade de ocorrência de riscos, o que pode ser efetivado pela determinação criteriosa das condições de transição do sistema de um estado seguro para um estado de risco.

É importante ressaltar que a análise de riscos e seu respectivo controle constituem um processo contínuo e interativo no ciclo de vida de um sistema. A Figura 2-1 abaixo mostra claramente esta idéia.

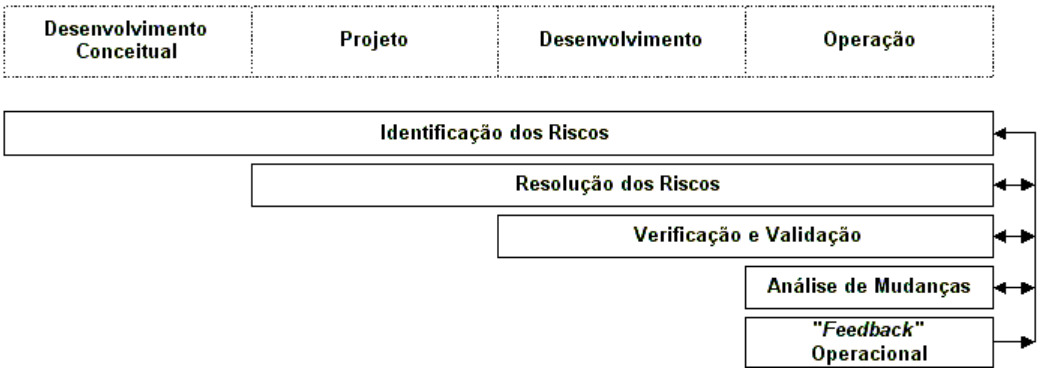


Figura 2-1 Análise de Risco no Ciclo de Vida de um Sistema

Pode-se adotar a seguinte precedência na resolução dos riscos identificados durante a fase de análise de riscos do processo que suporta o desenvolvimento de sistema de segurança crítico.

1. Eliminação do risco
2. Prevenção ou minimização da possibilidade de ocorrência
3. Controle do risco na eventualidade de ocorrência do mesmo
4. Minimização dos danos na eventualidade de ocorrência do mesmo

A Figura 2-2 abaixo mostra um diagrama que ilustra um processo de uma metodologia de suporte ao desenvolvimento de sistemas de segurança crítica.

Apenas as etapas relacionadas à segurança *(Safety)* serão enfatizadas no diagrama da Figura 2-2. Como se pode ver, a primeira etapa no processo de desenvolvimento de sistemas de segurança crítico é a ***Análise Preliminar de Riscos*** *(Preliminary Hazard Analysis)* da qual se origina uma lista denominada ***relação de riscos***. A partir da lista de riscos identificados é possível o estabelecimento dos requisitos de segurança e restrições do sistema.

A segunda fase no processo é denominada ***Análise de Riscos do Sistema*** *(System Hazard Analysis),* a qual com base na lista de riscos oriunda da fase anterior objetiva identificar as causas potenciais dos riscos identificados. É importante ressaltar que este tipo de análise considera interações entre componentes do sistema, que embora estejam funcionando corretamente, podem levar a estados de risco. A razão conforme explicada anteriormente é que a análise de riscos transcende o escopo da análise de falhas de componentes.

A Fase 2 do processo possibilita que a especificação dos requisitos do software seja elaborada e verificada através de uma coleção de técnicas e ferramentas que executam análises específicas, conforme lista que segue:

- Completeza
- Simulação e Animação
- Análise de riscos do Software (*State Machine Hazard Analysis*)
- Robustez
- Análise de Erros Humanos
- Análise de Fatores Humanos (Usabilidade, Carga de Trabalho)

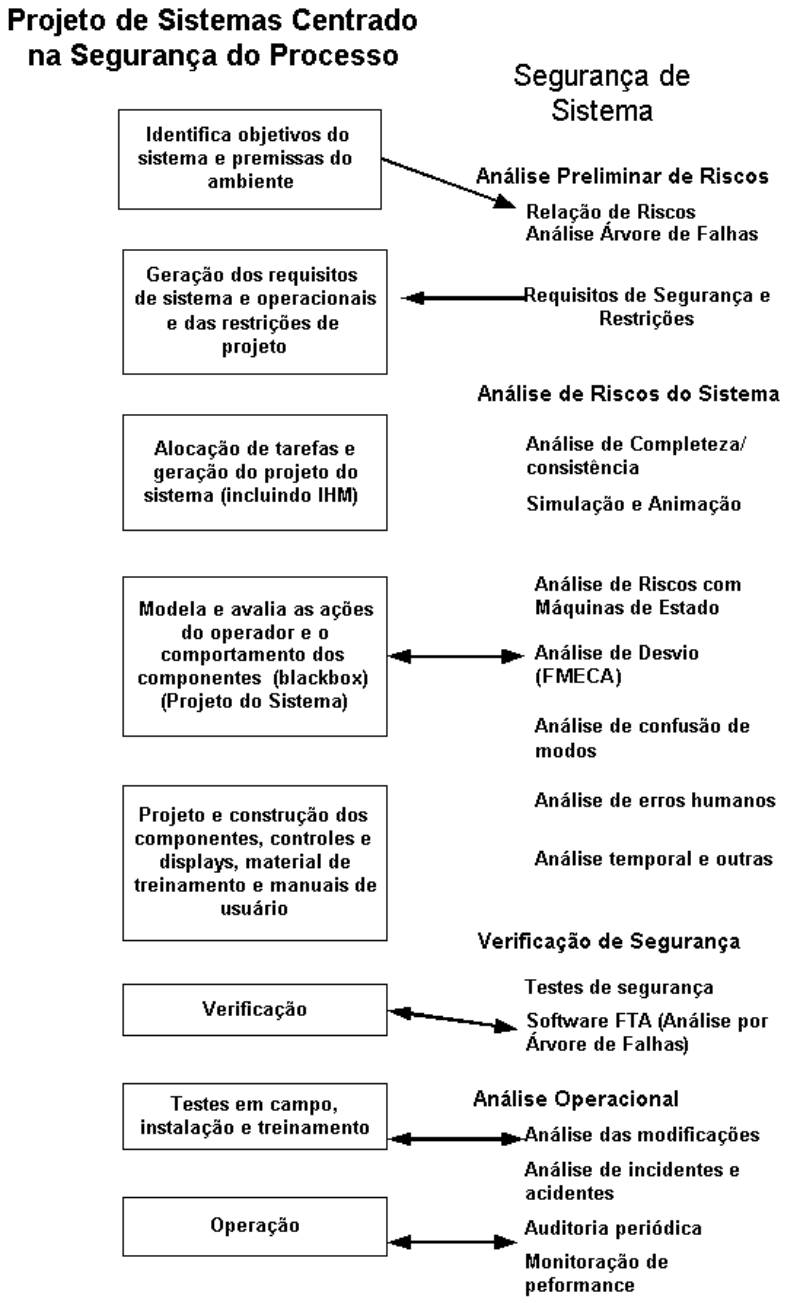


Figura 2-2 Metodologia de Projeto Dirigida a Segurança [LEVESON, 1995]

Os tipos de ferramentas listadas acima, as quais suportam a análise de segurança da especificação de requisitos em sistema de segurança críticos foram disponibilizadas pela empresa *Safeware Engineering Corporation*, através de um ambiente integrado denominado *SpecTRM-RL.* Ver Seção 2.3.2 para maiores detalhes.

Na fase de implementação do software, algumas recomendações são bastante relevantes, tais como:

- Separação das funções críticas relacionadas a requisitos de segurança, das funções não críticas. Isto facilitará a revisão e validação de forma independente.

- Uso de práticas de programação defensiva, como técnicas de redundância, detecção de erros e garantia de integridade de dados. Ver *Capítulo 3*, onde exemplos destas técnicas são descritos em detalhes.
- Escolha de linguagens que provêm facilidades a implementação de softwares de missão crítica.

A fase seguinte no processo conforme o diagrama da Figura 2-2 é a fase de ***verificação de segurança***. Esta fase em sistemas dessa natureza exige que os testes considerem todas as situações que potencialmente poderiam levar a condições de risco. Não basta testar as condições de normalidade do sistema, mais especialmente verificar o comportamento do software sob condições de anormalidade. Outro objetivo desta fase é realizar uma verificação de aderência aos requisitos de segurança do sistema os quais foram mapeados em componentes de software. A idéia é que sejam gerados cenários de testes com o objetivo de ratificar a correto comportamento e completeza dos requisitos de segurança do sistema.

Algumas das técnicas descritas na Seção 3.3.3 são análogas às requeridas nesta fase do processo de desenvolvimento de sistemas de segurança crítica.

Por fim a última fase do processo é relativa a análise do sistema em condições operacionais. A questão é que deve ser previsto em caso de evolução do sistema, com a eventual mudança de requisitos, uma reavaliação da análise de segurança do sistema. O eventual impacto nos aspectos de segurança do sistema resultantes de modificações evolutivas, deve ser criteriosamente avaliado. Investigações de acidentes ocorridos em processos críticos tiveram suas causas associadas a modificações em requisitos de software cuja verificação das implicações das mudanças nos aspectos de segurança não foram suficientemente avaliadas.

2.3.1 Análise de Risco em Software (*Software Hazard Analysis*)

A análise de risco do Software pode ser considerada uma forma de análise de risco de subsistemas. No contexto global do sistema (processo a ser controlado) apenas parte dos requisitos de segurança, podem ter sido implementado por software.

Essencialmente a análise de risco em software, objetiva validar que o comportamento do software especificado atende ou não as restrições de segurança definidas no projeto do sistema. Desta forma pelo menos os seguintes critérios devem

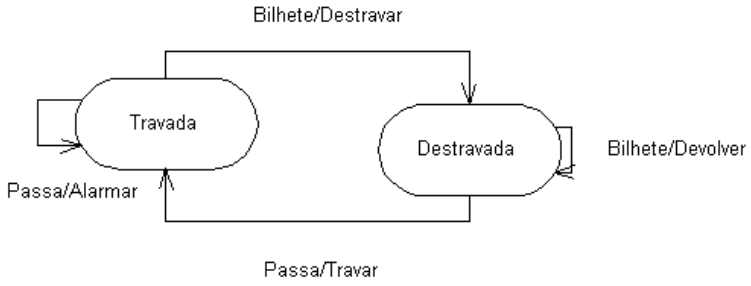
ser considerados neste tipo de análise, no sentido de como eles poderiam contribuir para que o sistema transitasse a estados de riscos.

- Condições de operação e performance normais
- Condições de operação degradada
- Falhas Funcionais
- Funções inesperadas (funções previstas, mas em momentos ou seqüências erradas).

Um dos aspectos fundamentais na análise de riscos em software, é que se deve ter em mente que embora os requisitos funcionais do software em geral especifiquem o que o software deve fazer, as restrições de segurança especificam o que ele não deve fazer. Adicionalmente deve ficar claro que as restrições de segurança que impõem aquilo que o software não deve fazer, não significam o oposto dos requisitos que o software deve fazer. Em suma é muito mais fácil garantir o que o software deve executar do que aquilo que não deve.

Como dito anteriormente métodos formais embora úteis, são de difícil compreensão para aqueles não versados em matemática discreta. O que os torna pouco práticos na análise de segurança de softwares complexos. O processo de análise de riscos em software necessita de resultados e métodos que possam ser facilmente lidos, revisados e discutidos por especialistas multidisciplinares. Deve ser lembrado que os aspectos de segurança de um sistema complexo envolvem não apenas engenheiros de software, mas especialistas em diversas áreas de engenharia afins ao processo que se quer automatizar.

Máquinas de estado finitas *(FSM Finite State Machine)* é uma poderosa e flexível ferramenta que permite modelar e analisar sistemas digitais e software. Por ser fundamentada matematicamente os modelos baseados em máquinas de estado podem ser analisados. Além do mais *FSM* possuem uma representação gráfica intuitiva e de fácil compreensão. Ver na Figura 2-3 um exemplo da representação de um modelo simples em *FSM*.



Estado Atual	Evento	Novo Estado	Ação
Travada	Bilhete	Destravada	Destravar
Travada	Passa	Travada	Alarmar
Destravada	Bilhete	Destravada	Devolver
Destravada	Passa	Travada	Travar

Figura 2-3 Exemplo de Representação de Máquina de Estado

Quando o número de estados é muito grande, aumentando substancialmente a complexidade da análise de riscos do software, níveis de abstração maiores podem ser utilizados. Especificações do tipo "*blackbox*" provêem formas de se traduzir o comportamento de softwares complexos em função de suas entradas, saídas e condições externamente observáveis. São omitidos os detalhes internos da implementação do software.

Modelos de processo constituem uma forma de representação *blackbox* do comportamento do software em termos de modelos baseados em máquinas de estado. Abaixo a Figura 2-4 mostra um exemplo clássico da teoria de controle referente a um modelo de processo.

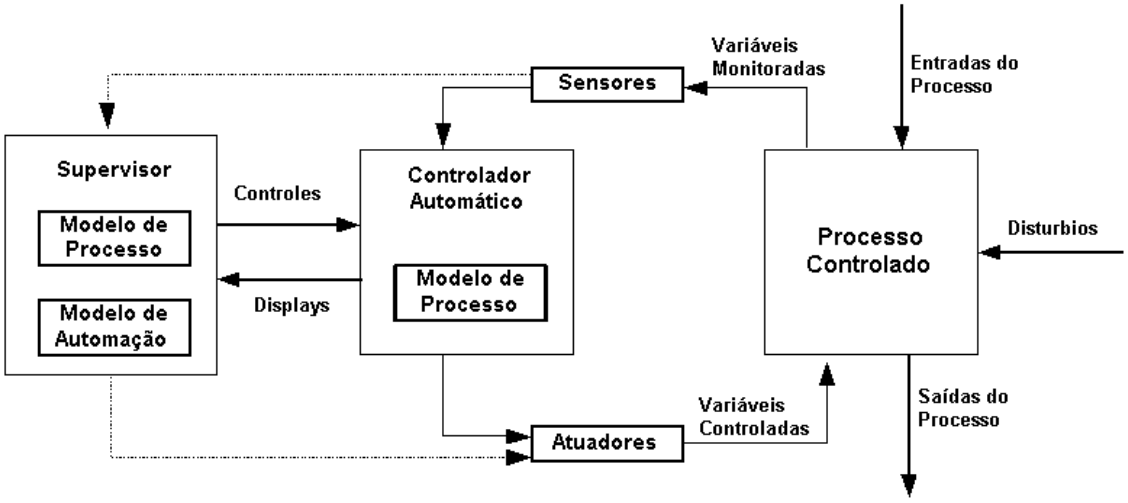


Figura 2-4 Modelo de Processo **[LEVESON, 1995]**

A ferramenta e linguagem de modelagem *SpecTRM-RL (Specification Tools and Requirements Methodology – Requirements Language)* provê recursos que permitem definir modelos de processo conforme descritos anteriormente. Adicionalmente os modelos de processo definidos nesta linguagem são executáveis, isto é, podem ser simulados e analisados formalmente. Ela combina uma linguagem de especificação de requisitos e de modelagem. Embora a linguagem seja baseada na teoria de máquinas de estado, sua sintaxe utiliza-se de abstrações que a tornam mais intuitiva. Este ambiente implementa os mais recentes critérios de completeza utilizados no desenvolvimento de software em sistemas de segurança crítica de tal forma a permitir uma análise sob esta perspectiva.

Uma descrição mais detalhada da metodologia e do ambiente integrado *CAD (Computer Aided Design)* para suporte ao desenvolvimento de sistemas de segurança crítica *(SpecTRM-RL)* pode ser encontrado na seção seguinte.

2.3.2 Um Exemplo de Ferramenta Centrada em Segurança (*SpecTRM-RL*)

Ferramentas que automatizam o processo de desenvolvimento de software constituem um valioso recurso no gerenciamento do processo de desenvolvimento com ganhos qualitativos e quantitativos de produtividade. Quando tais ferramentas são especificamente orientadas ao desenvolvimento de sistemas automatizados de segurança crítica, tem-se então a possibilidade de se controlar de forma mais efetiva e formal uma característica significativamente complexa: Garantia de Segurança em Software, conforme discutido nas seções anteriores.

SpecTRM-RL (*Specification Tools and Requirements Methodology*) é um ambiente *CAD* integrado o qual implementa resultados de pesquisas recentes no auxilio ao gerenciamento de especificações de requisitos e projeto de sistemas de segurança crítica. Este ambiente provê um conjunto de ferramentas de modelagem, especificação e análise que auxiliam na rastreabilidade dos requisitos, critérios de projeto e restrições de segurança durante o ciclo de desenvolvimento de um sistema de controle automatizado. É um exemplo de tecnologia transferida da universidade para indústria [LEVESON, 1998b].

A empresa *Safeware Engineering Corporation* tem disponibilizado o ambiente integrado comercialmente, implementando suporte aos resultados das pesquisas da professora Nancy G. Leveson que desenvolveu a metodologia. Esta

metodologia vem sendo usada experimentalmente em sistemas reais como: Sistemas de gerenciamento de vôo e alerta de colisão TCAS II nos EUA, bem como para verificação de segurança de atualizações que a NASA vem efetivando no Sistema de Controle de Tráfego Aéreo dos EUA.

Uma das principais características de *SpecTRM* é que a mesma provê modelos executáveis. Isso permite que simulações sejam realizadas no modelo em questão, com visualização gráfica de visões apropriadas que auxiliam no entendimento do modelo que representa o processo a ser automatizado. Outra característica singular é que a análise é centrada diretamente na especificação de requisitos através de uma forma bastante legível, ao contrário de muitas abordagens que requerem uma tradução complexa do projeto do sistema para uma linguagem de modelagem matemática.

O modelo e especificação executáveis no ambiente *SpecTRM* constitui a própria especificação do sistema. A Figura 2-5 abaixo mostra um resumo geral do modelo sob o qual está baseado *SpecTRM*, bem como suas características funcionais.

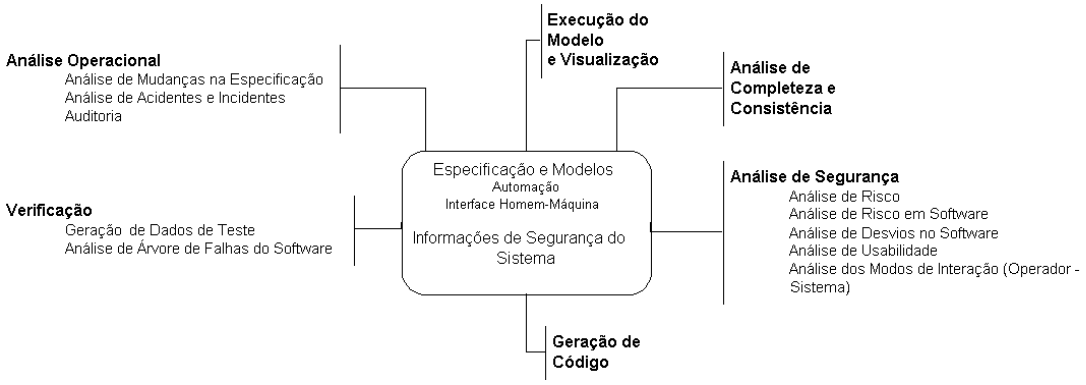


Figura 2-5 SpecTRM-RL (Estrutura Lógica) e Funcionalidades

A especificação e desenvolvimento de sistemas com foco em segurança, requerem um razoável esforço, o qual pode ser simplificado com o uso de ferramentas adequadas como *SpecTRM-RL* e outras. Uma das características base da metodologia na qual *SpecTRM-RL* é fundamentada, se chama de *Intent Specification*[1]. Essencialmente é uma nova forma de estruturação de especificação de sistemas desenvolvida pela

[1] *Intent Specification* é uma forma estruturada de especificação de sistemas de segurança crítica, desenvolvida pela pesquisadora Nancy Leveson (Universidade de Washington e MIT), a qual é centrada num princípio fundamental de solução de problemas e abstração usados por pessoas para tornar tarefas complexas, intelectualmente gerenciáveis [Leveson ,1998]

professora Nancy G. Leveson (Universidade de Washington e *MIT*). Esta abordagem enfatiza substancialmente a forma como se escreve a especificação de requisitos e o registro de aspectos críticos e relevantes do projeto. Objetiva-se atender a requisitos cognitivos essenciais a boa compreensão do projeto [LEVESON, 1997].

Intent Specification tem por premissa gerar registros documentais de projeto que sejam:

- Legíveis e facilmente compreensíveis
- Revisáveis
- Rastreáveis

Características como as citadas acima, contribuem para que o ciclo de vida do sistema seja evolutivo de uma forma controlada em relação às restrições de segurança. Isto evitaria que mudanças em funcionalidades do sistema, em especial no software, pudessem afetar restrições de segurança, simplesmente devido ao fato de determinadas características do projeto não terem sua razão suficientemente clara. Muitas vezes os responsáveis pela manutenção evolutiva de um sistema não necessariamente participaram de seu desenvolvimento. Uma questão crucial para os leitores de uma especificação é saber a razão pela qual uma determinada característica do sistema foi projetada da forma que foi.

Intent Specification no ambiente *SpecTRM-RL* estrutura a especificação do sistema e do software em níveis hierárquicos de abstração. As abstrações são organizadas em três dimensões: *Intent, Refinamento e Decomposição*. Conforme Figura 2-6.

A dimensão vertical (*Intent*) designa níveis específicos nos quais o problema está sendo considerado. As dimensões relativas à decomposição e refinamento possibilitam que os analistas possam direcionar sua atenção a níveis de detalhamento conforme a conveniência da situação.

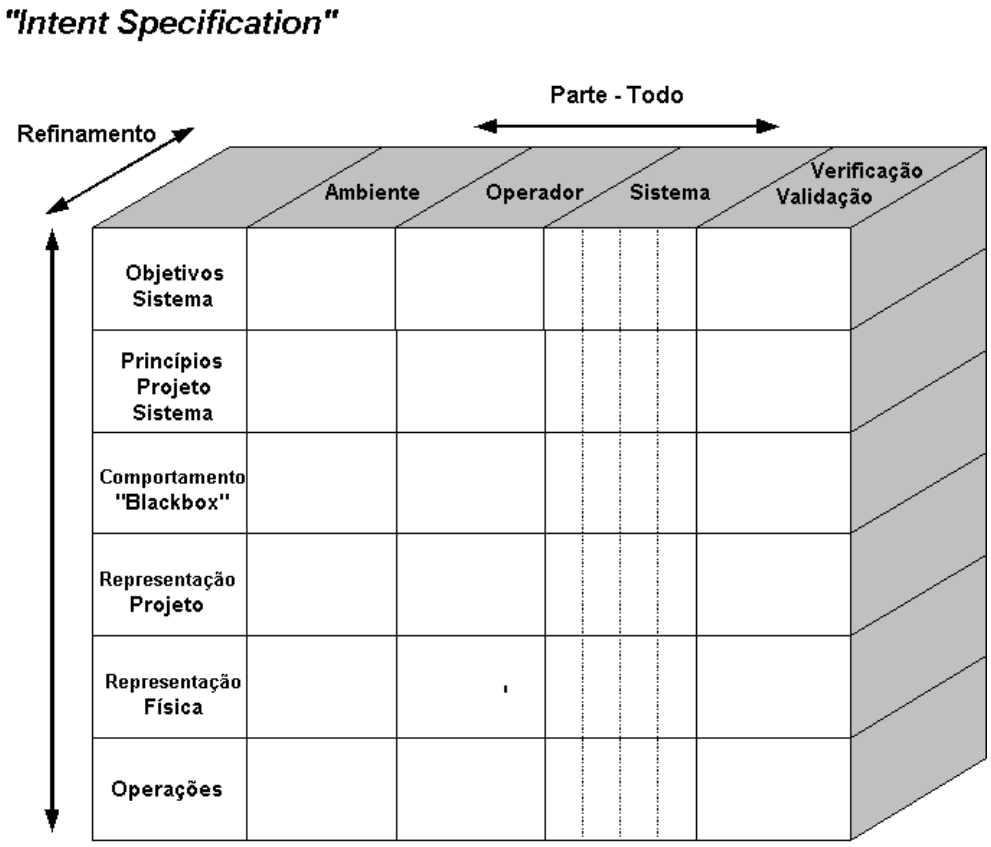


Figura 2-6 SpecTRM –RL Intent Specification (Estrutura Hierárquica) [LEVESON, 1995]

A dimensão vertical (*Intent*) possui 6 níveis de abstração. Os níveis mais elevados 1 e 2 fornecem basicamente informações gerais do sistema como objetivos, restrições de segurança que podem ter sido mapeadas durante a fase de análise preliminar de riscos. O nível 3 fornece recursos para a construção de modelos executáveis os quais são usados para analisar o comportamento do software do sistema numa abordagem "*blackbox*". Estes são formalmente analisáveis. Os níveis 4 e 5 contêm informações de projeto e implementação enquanto que o nível 6 informações operacionais do sistema, considerando-o já em produção [SAFEWARE, 2003].

2.4 Considerações Finais

Neste Capítulo foram apresentados os principais aspectos relacionados ao projeto de sistemas de controle com requisitos de segurança implementados em software. Foram discutidas as principais diferenças entre confiabilidade e segurança no contexto do software, de tal forma a concluir que um software confiável não necessariamente pode ser considerado seguro.

Foi também feita uma análise de uma metodologia de projeto para sistemas de controle e segurança crítica. Esta metodologia é implementada numa linguagem de modelagem e especificação de requisitos (*SpecTRM-RL*). Este ambiente se mostra bastante promissor no suporte à elaboração de especificação de requisitos focadas em segurança, devido a sua diferencial característica de estruturação da especificação e projeto baseada no conceito de *Intent Specification*.

No contexto deste trabalho o uso da ferramenta *SpecTRM-RL* poderá ser utilizada na evolução do simulador de controle de tráfego ferroviário, validando os aspectos de segurança e completeza dos requisitos das funções controle de intertravamento ferroviário, conforme serão apresentadas no Capítulo 4.

Complementares aos aspectos de segurança, os aspectos de confiabilidade de sistemas computacionais são fortemente considerados através do uso de conceitos de tolerância a falhas como redundância e diversidade de projeto, os quais serão detalhados no capítulo que segue.

3. Tolerância a Falhas em Sistemas Computacionais

Este Capítulo descreve de forma detalhada e abrangente aspectos relevantes sobre técnicas de tolerância a falhas em hardware e software. Uma descrição sumarizada dos principais conceitos envolvidos é apresentada, onde se enfatiza definições como **dependabilidade**. A seguir são apresentados os fundamentos das principais técnicas de tolerância a falhas em hardware como as de *replicação ativa* e *passiva*. Especial ênfase foi dada às técnicas de tolerância a falhas por software como as de programação em versão única (*Single Version*) e múltipla (*N-Version*). Devido a sua relevância a alguns dos objetivos deste trabalho, como a da aplicação de técnicas eminentemente baseadas em software em substituição as de hardware. Também devido a sua importância aos objetivos do trabalho descrevem-se de forma mais detalhada a técnica de redundância modular tripla *TMR (Triple Modular Redundancy)* e os códigos de redundância cíclica *(CRC - Cyclic Redundancy Codes).*

3.1 Conceitos

Inicialmente procuraremos apresentar as principais definições e conceitos empregados usualmente em tolerância a falhas: Dependabilidade, redundância e terminologias relacionadas com falhas, erros e defeitos. Sendo essencial a sua correta compreensão.

Outro aspecto relevante nesta seção é uma discussão sobre as principais abordagens e estratégias utilizadas em tolerância a falhas com sua respectiva terminologia associada.

3.1.1 Dependabilidade

Técnicas de tolerância a falhas são largamente utilizadas em sistemas de missão e segurança crítica, como forma de alcançar os requisitos de **dependabilidade**. Estes embora não signifique garantia de segurança, constitui importante indicador de qualidade sem os quais certamente a segurança também estaria comprometida.

Dependabilidade essencialmente caracteriza um conjunto de conceitos relacionados com o grau de confiança no funcionamento de um sistema ou em outras palavras a confiança requerida nos serviços a serem fornecidos por um dado sistema

[LAPRIE, 1989]. Dependabilidade ou confiança no funcionamento de um sistema é genericamente um conceito que compreende diferentes atributos associados a requisitos em sistemas de missão crítica. Os mais significativos são: **Confiabilidade** e **Disponibilidade**, além de **Segurança**, cujas definições são respectivamente expressas como:

- Probabilidade de o sistema fornecer um serviço correto num determinado instante.
- Probabilidade do sistema está pronto para executar sua missão num determinado instante.
- Probabilidade do sistema num dado instante, executar corretamente suas funções ou suspender seu funcionamento de tal forma a levar o mesmo para um estado de segurança intrínseco.

Um sistema pode ser considerado tolerante a falhas se na eventualidade da ocorrência de falhas em seus componentes ou subsistemas, o mesmo ainda continuar executando suas funções no todo ou em parte. O nível de tolerância a falhas de um sistema é associado ao grau de redundância utilizado em seus componentes e aos mecanismos específicos utilizados no controle dos componentes redundantes [SAMPAIO, 2000].

3.1.2 Redundância

A redundância introduzida num sistema pode ser de **hardware**, **software** ou **tempo** [AVIZIENIS, 1976]. Alguns autores incluem a redundância de **informação** à classificação anterior.

A redundância de **hardware** basicamente compreende componentes do sistema relacionados ao seu hardware como: Processadores, Canais de Comunicação, dispositivos E/S, etc. A redundância por **software** em geral é composta pelos componentes de software escritos para gerenciar os componentes de hardware redundantes, como também tolerar falhas da lógica de aplicação do sistema. A redundância por **tempo** é refletida em características que permitem que operações críticas sejam executadas repetidas vezes, com alocação de tempo adicional para estas ações. Constituem exemplos deste tipo de redundância a retransmissão de mensagens num sistema de comunicação, em que atrasos são considerados para possibilitar a

recuperação do sistema caso mensagens sejam inicialmente perdidas. A redundância de **informação** aparece nos códigos de correção de erros, redundância cíclica, comumente usados na codificação de informações em memória, ou protocolos de comunicação, com objetivo de melhor prover integridade de dados.

A redundância é utilizada tanto nas abordagens de detecção de falhas quanto no mascaramento das mesmas, como será visto posteriormente. Entretanto o nível de redundância difere e consequentemente o custo (econômico ou de processamento). O mascaramento exige um nível de redundância maior que o usado para detecção de falhas. .

Uma exposição mais detalhada sobre técnicas de redundância será apresentada nas seções 3.2 a 3.5.

3.1.3 Terminologia relativas a Falhas, Erros e Defeito

Muitos autores têm dedicado atenção à definição precisa e contextual de termos usados em tolerância a falhas. Os trabalhos de [LAPRI, 1995], [ANDERSON & LEE, 1981], [ANDERSON & LEE, 1990], servem de base para terminologia utilizada sobre falhas, erros e defeitos e sua semântica associada a sistemas computacionais. A seguir são definidos mais precisamente alguns termos que serão referenciados neste trabalho.

- **Defeito *(failure):*** Desvio do comportamento desejado por um sistema segundo sua especificação. Em geral a ocorrência de um defeito degrada ou impossibilita que um serviço ou a missão de um sistema seja executado. Defeitos não podem ser tolerados, no entanto evitar que os mesmos ocorram deve ser um objetivo fundamental na especificação de um sistema.
- **Erro *(error)*:** Elemento causal de um defeito, isto é, na presença de um erro, uma seqüência específica de ações pode levar o sistema a apresentar um defeito.
- **Falha *(fault):*** Causa física ou algorítima a partir da qual um erro é gerado.

É comum utilizar-se do modelo de 3 universos: **Falha**, **Erro** e **Defeito** para exemplificar e melhor caracterizar as definições apresentadas acima. Ver Figura 3-1.

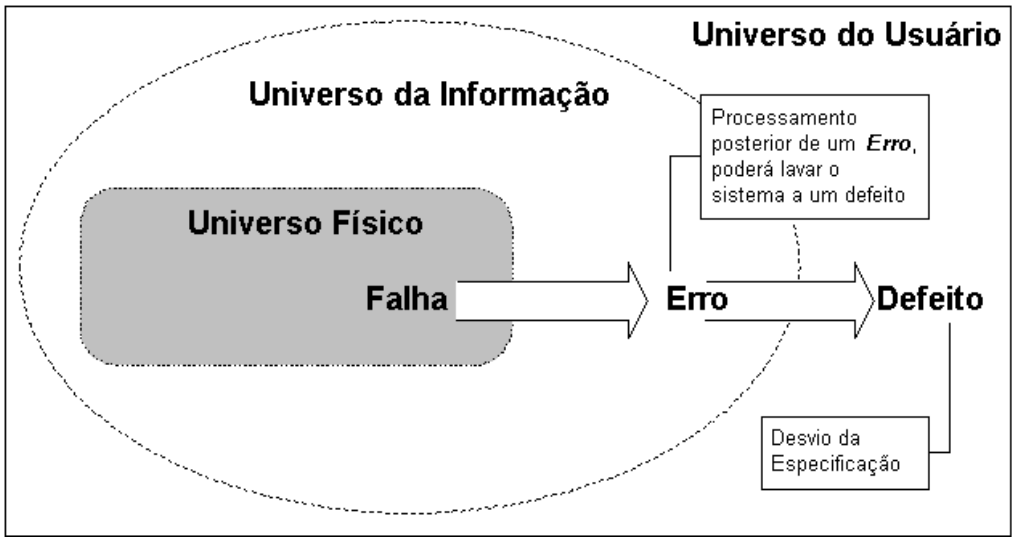


Figura 3-1 Modelo de 3 Universos : Falha, Erro e Defeito

Conforme mostra a Figura 3-1, falhas estão associadas ao universo físico, enquanto que os erros decorrentes de uma falha ao universo da informação. Por sua vez um defeito ocorrido após o posterior processamento de um erro está associado ao universo do usuário.

Exemplificando as definições anteriores se considerarmos um chip de memória que apresenta uma falha em um de seus bits, grampeado em um nível lógico 1 (**falha** no universo físico). Esta **falha** pode provocar uma interpretação errada da informação armazenada em uma estrutura de dados (**erro** no universo da informação). Como resultado o sistema representado pode deixar de gerar um alarme crítico relativo ao processo, deixando de alertar os operadores sobre uma condição anormal levando a situações de risco (**defeito** no universo do usuário). É interessante observar que uma **falha** não necessariamente leva a um **erro**. Aquela porção da memória pode nunca ser usada. Adicionalmente um **erro** não necessariamente conduz a um **defeito**. No exemplo, um alarme crítico poderia eventualmente ser disparado a partir de outros dados redundantes do processo.

Uma descrição detalhada da caracterização semântica de **falhas** em um sistema computacional pode ser encontrada em [JALOTE, 1994] e [CRISTIAN, 1991].

No contexto deste trabalho, é enfatizada uma natureza específica de falhas denominadas **Falhas de Concepção**, que essencialmente compreendem falhas de projeto em suas diversas fases como: *Especificação de Requisitos, Projeto e Implementação*. Tais falhas estão fortemente associadas a causas de acidentes em

sistemas de missão e segurança críticas cujos requisitos de segurança estão implementados em software.

3.1.4 Abordagens Usuais na Aplicação da Tolerância a Falhas

Diversos autores ocuparam-se na classificação das técnicas de tolerância a falhas, mas embora pequenas variações possam ser encontradas em geral duas linhas de aplicação são bem definidas e aceitas [LALA, 1994], são elas:

Mascaramento de falhas: Abordagem que procura evitar que **erros** se manifestem na presença de **falhas**, uma vez que o objetivo é mascarar as falhas do universo físico, se as mesmas ocorrerem. Nesta abordagem verifica-se o uso intensivo de redundância.

Detecção, Busca e Reconfiguração: Abordagem na qual se enfatiza os mecanismos de detecção de falhas e sua respectiva localização, de tal forma que mecanismos manuais ou automáticos de reconfiguração possam ser efetivados. Por exemplo: Reinicialização parcial ou total do sistema e retransmissão de mensagens.

Técnicas específicas que implementam estas duas abordagens estão descritas detalhadamente nas seções seguintes.

[ANDERSON & LEE, 1981], apresentaram uma classificação em fases na aplicação de técnicas de tolerância a falhas, as quais são comumente referenciadas. Tal classificação é denominada de **4 fases de aplicação**: *Detecção, confinamento, recuperação e tratamento*. A mesma é baseada na idéia de sequenciamento de procedimentos que devem ser executados após a ocorrência de falhas simples ou múltiplas.

Alguns mecanismos de tolerância a falhas associados a cada uma das fases de *Anderson* e *Lee* podem ser vistos na Tabela 3-1 que segue:

Fases	Mecanismos
Detecção de Erros	▪ Replicação ▪ Temporizadores (*Watch Dog Timers*) ▪ Codificação: Paridade, Redundância Cíclica, *Hamming* ▪ Testes de Consistência ▪ Diagnóstico
Confinamento	▪ Ações Atômicas ▪ Isolamento de Processos ▪ Hierarquia de Processo ▪ Exclusão Mútua de regras de Permissibilidade
Recuperação	▪ Recuperação por retorno (*Backward Error Recovery*) ▪ Recuperação por Avanço (*Forward Recovery)*
Tratamento	▪ Diagnóstico e Raparo

Tabela 3-1 Fases de Anderson e Lee

Essencialmente a **fase de detecção**, consiste em detectar erros após a ocorrência de falhas, as quais estavam latentes.

A **fase de confinamento** é decorrente da característica de latência das falhas, que até ocorrerem podem ter propagado dados corrompidos. No entanto, definir limites de propagação são critérios dependentes de decisões de projeto. Portanto as restrições de fluxos de dados e facilidades para detecção de erros devem ser previstas em fases anteriores a implementação. A fase de confinamento muitas vezes não é implementada em situações reais, devido ao alto grau de complexidade e custo envolvido na sua opção. Em seu lugar as técnicas de mascaramento de falhas são preferidas.

A **fase de recuperação** ocorre após a detecção de erros. Em geral os dois mecanismos apresentados na Tabela 3-1, são os mais utilizados: Recuperação por retorno (*Backward Recovery*) e Recuperação por avanço (*Forward Recovery*).

Dependendo da natureza do sistema, a decisão de uso de mecanismos de recuperação deve ser criteriosamente verificada. Pois a complexidade contextual envolvida na recuperação de defeitos pode ser razoável. Por exemplo, a mesma é razoavelmente simples em sistemas convencionais monoprocessados, no entanto pode ser muito complexa em sistemas distribuídos [JANS, 1997].

Em sistemas de tempo real, a recuperação de falhas por retorno (*backward recovery*) é proibitiva, devido a restrições temporais. O *overhead* de tempo necessário à recuperação de um erro, pode não ser suportado pelo processo de tempo real, o que implicaria perda em de eventos.

Em sistemas de segurança crítica, a recuperação de erros, também não é considerada, devido à criticidade relacionada com os requisitos de segurança. O sistema em caso de erros necessita assumir um estado de segurança intrínseco com elevado grau de confiança. Para tanto os mecanismos de recuperação necessitariam ser validados quanto a segurança, talvez utilizar de redundância, etc. O custo e complexidade seriam elevados e no contexto prático seria difícil garantir os níveis de segurança, devido as razões detalhadas no Capítulo 2.

Por último a **fase de tratamento da falhas**, é considerada com o objetivo de localizar a origem da falha (módulo afetado) e reparar o sistema num contexto automático ou manual.

Uma das abordagens mais utilizadas é o que se chama de **mascaramento de falhas**, que essencialmente objetiva a continuidade da missão de um sistema, mesmo na presença de falhas. Quando os erros não se manifestam (são mascarados), não há necessidade de implementação dos mecanismos de detecção, confinamento ou recuperação. No entanto isto apenas é válido para os erros efêmeros não permanentes. No caso de erros duradouros, mesmo com o mascaramento, mecanismos de detecção e recuperação devem ser considerados. Em geral para garantir os requisitos de disponibilidade (*availability*) do sistema.

As técnicas de mascaramento de falhas têm em sua essência o uso de um ou mais tipos de redundância, conforme conceituação e classificação apresentada na Seção 3.1.2. As mais utilizadas em sistemas tolerantes a falhas reais são citadas a seguir:

- Redundância de hardware (Replicação de Componentes)
- Codificação (Redundância Cíclica, Códigos de Correção de Erros - *ECC*)
- Redundância de software (Programação *Single Version* e *N-Version*)

As seções que seguem mostram de forma mais detalhada os esquemas de redundância de hardware e software mais largamente utilizados.

3.2 Técnicas de Tolerância a Falhas em Hardware

Na tolerância a falhas de hardware é bastante usado o conceito de replicação, que nada mais é que redundância. Se considerarmos que um determinado sistema computacional deve tolerar falhas, abstraindo-se os defeitos de projeto (concepção), em geral replica-se componentes de software em processadores distintos,

com o objetivo de mascarar falhas em processadores específicos. Caso o objetivo seja tolerar defeitos de concepção, isto implica na necessidade de se usar algum tipo de **diversidade** no projeto do hardware.

Existem várias técnicas de replicação (redundância) de hardware em sistemas tolerantes a falhas. Duas das mais utilizadas são as denominadas: **Replicação Ativa**, **Replicação Passiva ou Replicação Híbrida** (combinação das anteriores).

Uma discussão detalhada sobre as estratégias de replicação ativa e passiva pode ser encontrada em [VASCONCELOS, 1997].

3.2.1 Replicação Ativa ou Redundância Dinâmica

A idéia básica deste tipo de redundância é prover mecanismos de detecção, localização e recuperação, contudo sem o uso de mascaramento de falhas, para se alcançar a tolerância a falhas. Adequada para aplicações que suporta uma determinada latência de falhas em seus módulos, de tal forma a ser possível a detecção das mesmas.

Os módulos redundantes dinâmicos usualmente assumem duas configurações:

- *Hot Standby:* No qual o módulo redundante assume a continuidade do funcionamento após a detecção da falha. Os módulos redundantes são conectados ao sistema energizados. Podendo assumir "*a quente*" as funções do módulo que apresentou a falha.
- *Cold Standby:* Neste caso os módulos redundantes não se encontram energizados. A continuidade das funções ocorre somente após a conexão manual dos módulos redundantes.

Neste tipo de replicação temos "***n***" processadores paralelos, nos quais estão baseadas réplicas de componentes de software computando um conjunto de dados de entradas os quais produzem saídas idênticas igualmente ordenadas. Estes dados são submetidos à mesma variação determinística de estados, caso o processamento ocorra sem que haja falhas. Dependendo da semântica de falhas adotada, as saídas de todas as réplicas são submetidas a um processo de votação majoritária. Este processo é realizado por um componente externo denominado **votador**, no qual é efetivada a escolha da saída apropriada.

O uso de replicação na tolerância a falhas requer que algumas condições [SCHNEIDER, 1990] sejam satisfeitas:

- **Consistência de Entrada**: Todas as réplicas devem receber necessariamente os mesmos valores de entradas.
- **Determinismo das Réplicas**: A partir de um mesmo estado inicial, a computação de um mesmo conjunto ordenado de dados de entradas, deve produzir em todas as réplicas na ausência de falhas, os mesmos conjuntos ordenados de valores de saída. A abordagem de **máquina de estado** é uma forma usual de garantia desta condição.

No Capítulo 4 estudaremos um caso real de um controlador de tráfego ferroviário simulado, cuja implementação é baseada no conceito de replicação ativa de hardware, conforme descrito nesta seção.

3.2.2 Replicação Passiva

Neste tipo de replicação, apesar de existir um grupo de réplicas de processadores, apenas um deles, denominado de **réplica primária**, processa o conjunto de dados de entrada. As demais réplicas denominadas de **réplicas secundárias** só serão ativadas em caso de falha na réplica primária. Esta estratégia pressupõe uma semântica de falhas dos processadores por paradas. Isto significa que em caso de falha num processador primário, o mesmo será inibido, havendo um chaveamento para um processador secundário. Notar que neste tipo de replicação não há necessidade de um mecanismo de validação dos resultados como um votador, significando que na ausência de falhas todas as saídas originadas de uma réplica do grupo são consideradas corretas.

Em geral, neste tipo de replicação as réplicas secundárias guardam estados de variáveis do ambiente, com o objetivo de dar continuidade ao processamento em caso de falha da réplica primária. Para que isto ocorra, o processador principal precisa ter uma política de atualização do ambiente das réplicas secundárias, através do envio de salvaguardas *(checkpoints).* Estes salvaguardas podem ser emitidos periodicamente ou sempre que houver uma mudança num estado da réplica primária [SAMPAIO, 2000].

É importante ressaltar que no caso da replicação passiva, as réplicas secundárias não realizam processamento efetivo, apenas a atualização de estados internos.

Outro fator importante advém do fato de que essa abordagem acarreta um tempo de processamento extra durante a execução dos mecanismos de recuperação de falhas, executadas pela réplica secundária. Este tempo extra pode não ser compatível com requisitos de tempo real em alguns sistemas. Adicionalmente vale salientar que do ponto de vista de performance a abordagem de replicação passiva é inferior à replicação ativa.

3.3 Técnicas de Tolerância a Falhas em Software

Diferentemente do hardware, o software não possui características físicas as quais possam ser modeladas em termos de sua taxa de falhas, de tal forma que um modelo estatístico pudesse ser usado para inferir quão provável uma falha acometeria o mesmo. O Software é essencialmente um produto intelectual. Suas falhas são decorrentes basicamente de falhas de projeto (concepção).

Técnicas usuais em teoria da confiabilidade aplicadas a sistemas computacionais não são suficientes para mensurar e garantir qualidade do software. Nem tão pouco os aspectos de segurança crítica associados ao sistema. Conforme discutido na Seção 2.2.

O Software tolerante a falhas baseia-se no princípio da **diversidade de projeto**, que basicamente significa o desenvolvimento de componentes de software utilizando diversidade para uma mesma especificação funcional de requisitos. O objetivo esperado é que as réplicas do software redundantes sejam independentes em seus modos de falhas, isto é, não falhem da mesma forma em decorrência das mesmas falhas de projeto.

Diversas técnicas aplicadas para se chegar a tolerância a falhas em software são analisadas nas seções seguintes. Basicamente divididas segundo [LYU, 1995] em duas categorias: Programação em versão simples ou única *(Single-Version)* e programação em múltiplas versões *(N-Version).*

As técnicas baseadas na abordagem de versão única (*Single Version*) de um componente de software são focadas na melhoria da tolerância a falhas de uma única porção do software. Em geral através da incorporação de mecanismos no projeto de software, que tornem possível a *detecção*, *confinamento* e *tratamento de erros* causados pela ativação de falhas de projeto. Já as técnicas baseadas no conceito de versões múltiplas (*N- Version*), usam múltiplas versões ou variações de uma determinada porção

do software, estruturados de tal forma a minimizar a probabilidade que falhas de projeto em uma versão causem defeitos no sistema.

3.3.1 Técnicas de Programação Versão Única (Single Version Programming)

O conceito de programação versão única (*single version*) é baseado no uso da redundância aplicada a uma única versão de um componente de software, com objetivo de detectar falhas, bem como promover a recuperação do sistema [LYU, 19 95]. Segue uma breve análise de algumas das técnicas mais representativas desta categoria, segundo [LYU, 1995], tais como:

- Considerações sobre a estruturação do software e ações específicas
- Detecção de Erros
- Tratamento de exceções
- Salvaguardas e reinicialização (*checkpoint and restart*)
- Pares de processos
- Diversidade de dados

3.3.1.1 Estruturação do Software e Atomicidade de Ações

A **arquitetura do software** define a base de implementação da tolerância a falhas. O uso de técnicas de modularização com objetivo de decompor o problema em componentes de software gerenciáveis é tão importante para aplicação da tolerância a falhas como o é para o próprio projeto do sistema [POMALES, 2000]. A decomposição modular do projeto deve considerar proteções internas, de forma a manter confinada em um módulo específico o componente de software anormal. Restringindo a propagação de seus efeitos a outros módulos do sistema.

O fechamento do sistema (*system closure*) constitui um principio de tolerância a falhas, o qual define que nenhuma ação pode ser permitida caso não esteja explicitamente autorizada. Este princípio estabelece as seguintes premissas:

- A nenhum elemento do sistema é dado mais capacidade do que aquela estritamente necessária para execução de suas funções
- Qualquer restrição deve ser removida antes que uma determinada capacidade possa ser usada.

A razão do uso deste princípio é tornar mais simples e seguro o tratamento de erros. Limitando as chances dos mesmos se propagarem e ocasionarem mais danos antes de serem detectados.

A estruturação temporal de atividades entre módulos do sistema de acordo com a modularização definida, também constitui importante fator na implementação de software tolerante a falhas. Este tipo de estruturação é refletida no uso de exclusão mútua em determinadas ações (acessos a regiões críticas) de partes do software. A atomicidade de ações permite a referida exclusão mútua, não havendo interação com o resto do sistema durante a execução de uma ação atômica. A noção de atomicidade define a base necessária a implementação do *confinamento de erros* e *mecanismos de recuperação*. Numa ação atômica, há apenas duas possibilidades: O seu término normal ou seu cancelamento caso algum erro seja detectado durante sua execução.

3.3.1.2 Detecção de Erros

Segundo [ABBOTT, 1990], a aplicação efetiva de tolerância a falhas em software de versão única, pressupõe que os módulos resultantes da decomposição estrutural da arquitetura possuam duas propriedades básicas: auto-proteção (*self-protection*) e auto-verificação (*self-checking)*. A auto-proteção é uma propriedade presente em módulos de software que possuem a habilidade de se proteger contra distúrbios externos. Por exemplo, através de mecanismos de detecção de erros nos dados passados por outros módulos do sistema com os quais haja interação. A auto-verificação é a capacidade que um componente de software deve ter de detectar erros internos e tomar medidas apropriadas para evitar a propagação destes erros nos outros módulos do sistema [SHIN, 1984].

É importante notar que o grau de abrangência no sistema dos mecanismos de detecção de erros é determinado pelo custo da redundância adicional e sobrecarga de dados em tempo de processamento [ZAHEDI, 1991]. O uso real de redundância de software para se atingir os níveis requeridos de tolerância a falhas é baseado na relação custo/benefício de fatores como: Funcionalidades, performance, complexidade e segurança [POMALES, 2000].

[ANDERSON & LEE, 1981] propôs a seguinte classificação de técnicas de detecção de erros, as quais podem ser aplicadas na implementação de mecanismos que

atendam os requisitos das propriedades de **auto-proteção** e ***auto-verificação*** mencionadas anteriormente:

- ***Checks por Replicação***: Faz uso de replicação de componentes de software, de tal forma que a detecção de erros fica baseada na comparação das saídas destes componentes. Também é aplicável em software de versões múltiplas, conforme será discutido na Seção 3.3.2.
- ***Checks de Temporização*** : Aplicável em módulos de software com restrições temporais, tais como, "*deadlines*". A detecção de erros pode ser baseada nos resultados destes *checks*, os quais implementam formas de monitorar desvios no comportamento esperado do módulo. Rotinas de temporização tipo "*Watchdogs"* são exemplos práticos de como se detectar componentes de software travados, através de *checks* de temporização.
- ***Checks por Computação Reversa:*** Baseado na idéia de realizar a computação das saídas de um determinado módulo de software, no intuito de se determinar as entradas relativas. A detecção de erros ocorre se a comparação das entradas calculadas a partir de saídas não corresponderem com as entradas reais do processo. Esta técnica á aplicável quando não existe uma grande complexidade na realização da computação reversa, de tal forma que possa comprometer outros requisitos de sistemas como performance ou requisitos de tempo real.
- ***Checks por Codificação de Dados***: Baseado na redundância de informação, com um relacionamento pré-determinado entre a informação real e a informação redundante. A detecção de erros é baseada na verificação destes relacionamentos antes e depois das respectivas computações. Exemplos de *checks* por codificação: *Checksum*, códigos de redundância cíclica *(CRC´s),* manipulação aritmética de dados, aqueles baseados em propriedades matemáticas entre o dado real e sua porção redundante. Em sistemas que necessitam de alta confiabilidade relacionada a integridade de dados, como a comunicação inter-processos em sistema de segurança crítica, é usual a aplicação deste tipo de detecção de erros.
- ***Checks de Racionalidade de Dados***: Baseado no conhecimento de propriedades semânticas dos dados, tais como: Sequenciamento, faixa de valores, taxa de mudanças, etc. Em geral estas propriedades são dependentes de requisitos específicos de um determinado sistema.

- ***Checks Estruturais de Dados***: Neste caso é usado o conhecimento de propriedades relacionadas com estruturas de dados conhecidas. Filas, listas, árvores binárias, possuem elementos que podem ser articulados no sentido de se detectar inconsistências relativas a propriedades, como por exemplo, o número de elementos, ponteiros, etc.

3.3.1.3 Tratamento de Exceções

O tratamento de exceções essencialmente é caracterizado pela interrupção do processamento normal de um componente de software, com objetivo de tratar condições anormais detectadas. No contexto de software tolerante a falhas, as exceções são disparadas pelos mecanismos de detecção de falhas implementados, de tal forma a requisitar o inicio de ações apropriadas de recuperação.

O correto desenvolvimento de mecanismos de detecção de erros auxilia consideravelmente na contenção de erros em determinadas partes da arquitetura estruturada do software. Sendo isto essencial no desenvolvimento de mecanismos efetivos de tratamento de exceções.

3.3.1.4 Salvaguardas (*Checkpoints*) e Reinicialização

Mecanismos de salvaguardas *(checkpoints)* e de reinicialização são frequentemente utilizados em software de versão única como abordagens de recuperação de falhas [PRADHAN, 1996].

Falhas de software como discutidas no Capítulo 2, estão fortemente associadas à falhas de projeto. Aquelas que permanecem latentes após o desenvolvimento, em geral não são previsíveis, isto é, suas condições de ativação são dependentes de estados e condições internas do software. Este tipo de falha tem um comportamento similar às falhas transientes de hardware, aparecendo inesperadamente, degradando o sistema através da manifestação de defeitos, para então desaparecerem de forma aparente. Tornando-se latentes novamente, sem que uma razão determinada para sua manifestação fique explicita [GRAY, 1986]. Devido às características supracitadas, muitas vezes um simples procedimento de reinicialização do módulo de software é suficiente para que sua execução seja bem sucedida após a reinicialização.

A idéia da renicialização pode ser considerada conceitualmente simples e sua aplicabilidade pode se generalizada para diversos níveis do sistema. Outra

importante característica desta abordagem é sua independência das conseqüências associadas à falha ocorrida [ANDERSON & LEE, 1981]. A reinicialização também é conhecida como recuperação de erros para trás *(backward error recovery).*

Existem basicamente dois tipos de mecanismos de recuperação: **Estático e Dinâmico**. A recuperação estática promove o retorno do módulo de software para sua condição inicial ou simplesmente para um estado pré-determinado. A recuperação dinâmica é baseada no uso de **salvaguardas** *(checkpoints)* que basicamente são fotografias de estados em diversos pontos do processamento, que permitem a continuidade da execução do módulo, mesmo após a detecção de falhas. Os estados guardados são obtidos durante o progresso da computação, não havendo necessidade, portanto de descartar todo o trabalho realizado até o instante da detecção do erro.

Em aplicações reais, o uso de *checkpoints* deve considerar algumas restrições, pois em situações ideais a detecção de erros é considerada instantânea, o que não ocorre na prática. Isto é, há um atraso entre a ocorrência da falha no módulo de software e sua efetiva detecção. A existência dos atrasos acima mencionados é determinante para a estratégia de implementação dos *checkpoints*. Por exemplo, em casos que os *checkpoints* ocorrem em intervalos regulares. Se o tempo de detecção de falhas for maior que este intervalo, então os *checkpoints* poderão ser invalidados, anulando a efetividade do mecanismo de recuperação [SAMPAIO, 2000].

Ainda discorrendo sobre o uso de uma estratégia dinâmica ou estática de *checkpoints.* A abordagem estática permite que o mecanismo de detecção de erro seja colocado no final de cada módulo da arquitetura estruturada do sistema, não sendo necessário *checks* distribuídos no código. O problema com essa abordagem é que na presença de falhas aleatórias o tempo esperado para execução do módulo cresce exponencialmente com o processamento incluindo reinicializações. Pois estas retornam o programa ao inicio da execução do módulo [POMALES, 2000]. A abordagem dinâmica de *checkpoints* teoricamente focaliza a redução do tempo de execução na presença de falhas aleatórias. Salvando estados do sistema em pontos intermediários da execução de um módulo do software.

3.3.1.5 Pares de Processos

Essencialmente a idéia de pares de processos em software de versão única tolerante a falhas, se baseia na existência de duas versões idênticas do software

executado em processadores distintos. Em geral neste caso utiliza-se como mecanismo de recuperação a técnica de **reinicialização** e *checkpoint* discutidas na Seção 3.3.1.4.

A Figura 3-2 abaixo mostra logicamente a arquitetura de pares de processo, bastante utilizada em processos reais com software tolerante a falhas.

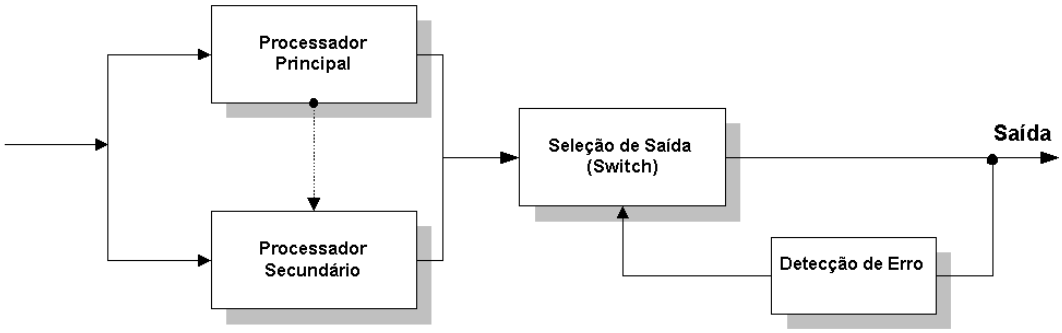


Figura 3-2 Representação Lógica de Pares de Processo

Têm-se dois processadores denominados de **primário** e **secundário** respectivamente. O primário executa o processamento das entradas do processo gerando as respectivas saídas, além de gerar os *checkpoints*, isto é, salva o contexto do processamento que produziu as atuais saídas. Quando na presença de falhas detectadas pelo mecanismo de detecção de erro implementado, o processador secundário carrega o último *checkpoint* produzido e então assume o papel do processador primário que falhou. Este por sua vez quando reparado e retornar a situação de disponível passará a executar o mesmo papel do processador secundário, recebendo os *checkpoints* do processador primário. Neste tipo de redundância pode ser enfatizado o fato da abordagem de recuperação utilizada permitir que o sistema ou módulo permaneça em serviço contínuo mesmo na presença de falhas, o que prioriza a questão da disponibilidade de sistemas de missão crítica.

3.3.2 Programação em Versões Múltiplas

A programação em **múltiplas versões** em software tolerante a falhas é baseada no uso de duas ou mais versões de uma parte do software de controle do sistema, executando sequencialmente ou em paralelo.

Um dos trabalhos mais abrangentes e referenciados no intuito de estabelecer os fundamentos de uma metodologia para desenvolvimento de software de múltiplas

versões, foi desenvolvido por **Algirdas Avizienis** e sua equipe na UCLA, desde 1970 [POMALES, 2000], [CHEN & AVIZIENIS, 1978].

As múltiplas versões são usadas como alternativas, implementado diferentes formas de detecção de erros. Essencialmente esta abordagem é baseada na idéia de que as diferentes versões do software devem falhar diferentemente na presença das mesmas falhas[LYU & HE, 1993]. A diversidade de projeto é um conceito fundamental neste contexto.

[AVIZIENIS, 1995] propôs uma metodologia de desenvolvimento de software de múltiplas versões a qual considera todo o esforço de desenvolvimento do sistema, desde a fase de especificação de requisitos até a fase operacional do ciclo de vida. Essencialmente o objetivo é minimizar a probabilidade de falhas de projeto, eliminando ou minimizando qualquer que seja a fonte de falhas de modo comum inclusive nas saídas. Em geral os princípios de engenharia de software são válidos, o que diferencia dos projetos de propósito geral é a introdução do suporte a diversidade de projeto.

Várias são as considerações a serem tomadas no projeto de software tolerante a falhas, baseados na abordagem de programação de múltiplas versões. Seguem algumas relevantes:

- Escolha do número de versões a serem desenvolvidas
- Avaliação do tipo de diversidade a ser empregada: Especificação, projeto, codificação e procedimentos de testes. Considerando também a combinação entre eles.
- Avaliação do uso de diversidade aleatória (não gerenciável) ou determinística (gerenciável ou forçada), com objetivo de minimizar a probabilidade de ocorrência de falhas de modo comum.
- Regras de isolamento entre equipes de desenvolvimento, com objetivo de reduzir a probabilidade de erros de modo comum na fase de projeto.

Devido a complexidade inerente do software, é consenso entre vários pesquisadores que a aplicação prática da diversidade de projeto, com base na atual tecnologia e metodologias disponíveis, ainda constitui mais **arte** que propriamente uma **ciência**. Os benefícios da aplicação da diversidade de projeto ainda são difíceis de mensurar, em especial a questão de garantia de independências de falhas.

Outra questão crucial no uso prático da abordagem da programação em múltiplas versões é o **custo de desenvolvimento**. Na verdade dependendo do nível de criticidade o custo pode ser proibitivo para determinadas aplicações. No entanto alternativas tem sido usadas para contornar esta questão, tais como: Utilizar diversidade apenas nas partes realmente críticas e imprescindíveis do sistema ao invés do todo.

Existem duas questões críticas no uso de software tolerante a falhas com a abordagem de múltiplas versões, são elas: A garantia de independência de falhas das versões múltiplas do software e o desenvolvimento de algoritmos de seleção apropriados [POMALES, 2000].

Nas seções seguintes serão discutidas três técnicas básicas que se enquadram nesta categoria. No entanto existem várias outras propostas de diversos pesquisadores em software tolerante a falhas que constituem variantes ou combinações das técnicas básicas aqui apresentadas.

3.3.2.1 Blocos de Recuperação

A técnica de blocos de recuperação foi inicialmente introduzida por [RANDELL, 1975]. Essencialmente combina o uso de salvaguardas *(checkpoints)* e a abordagem de reinicialização com múltiplas versões do software, de tal forma que uma versão diferente seja executada sempre que uma falha for detectada [POMALES, 2000].

Um bloco de recuperação é constituído de um módulo principal, o qual realiza as funções especificadas para um determinado componente de software. Um módulo de aceitação que verifica e valida as saídas produzidas por uma tarefa específica e $k\,(k>0)$ módulos alternativos ao principal. Estes serão executados sequencialmente no caso de não aceitação das saídas produzidas pelo módulo anterior. A Figura 3-3 mostra esquematicamente como ocorre o fluxo de execução de um bloco de recuperação.

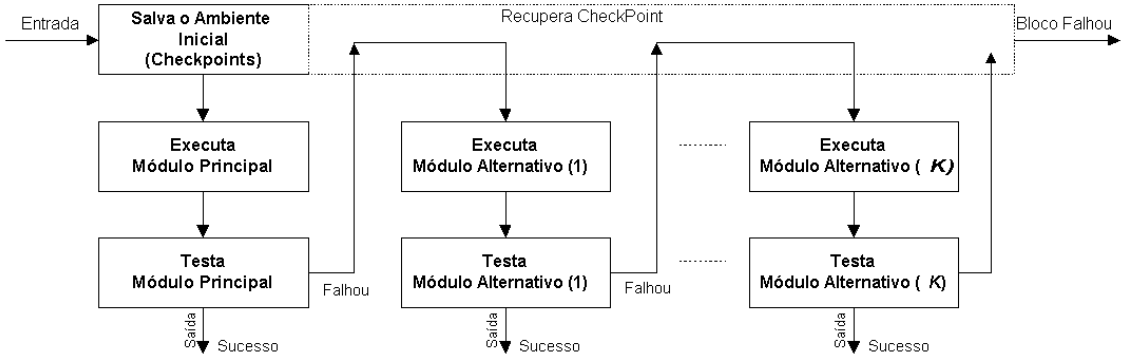


Figura 3-3 Representação Lógica de um Bloco de Recuperação

Alternativamente o modelo lógico de um bloco de recuperação também é mostrado na Figura 3-4 abaixo:

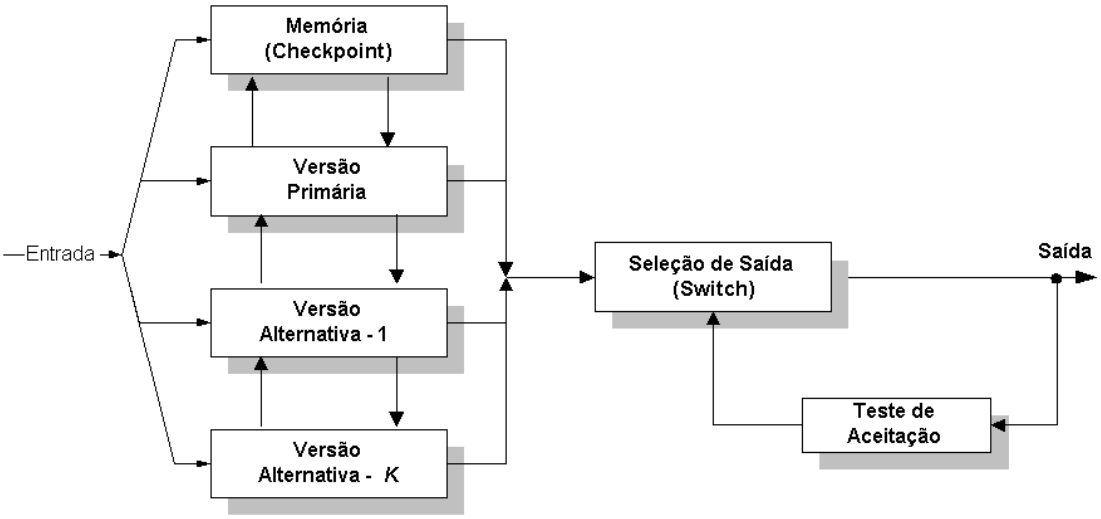


Figura 3-4 Representação Alternativa de um Bloco de Recuperação

Como podemos ver a criação dos *checkpoints* se dá antes da execução da versão principal do módulo de software. Sendo este *checkpoint* necessário para prover uma forma de recuperação do estado dinâmico do ambiente após uma falha ter ocorrido. Provendo também para a versão alternativa um ponto de inicialização válido após a ocorrência de uma falha.

3.3.2.2 Programação *N-Version*

A programação *N-Version*, proposta por [AVIZIENIS, 1985] é baseada no uso de duas ou mais versões de um mesmo software, o qual pode ser executado sequencialmente ou em paralelo [POMALES, 2000]. Podem implementar diferentes

formas de detecção de erros. Sendo usada em pares com objetivo de detectar falhas pela replicação de testes ou em grandes grupos para mascarar falhas através de mecanismos de votação [POMALES, 2000]. A Figura 3-5 mostra genericamente a abordagem da programação *N-Version*.

É importante ressaltar que todas as versões devem atender a mesma especificação de requisitos funcionais, sendo a decisão de corretude das saídas baseada na comparação entre elas. O uso de algoritmos genéricos de decisão, em geral elementos **votadores**, estabelece uma diferença fundamental com a técnica de **blocos de recuperação**. Esta última requer testes de aceitação dependentes da aplicação.

Nesta abordagem as N-versões devem processar o mesmo conjunto de entradas, de tal forma que suas saídas respectivas sejam avaliadas por um algoritmo de decisão majoritário. O resultado final será válido caso no máximo $\lceil (N-1)/2 \rceil$ saídas sejam consideradas incorretas, considerando que as saídas restantes (corretas) sejam idênticas.

O requisito anterior não é válido para as aplicações em que é possível a ocorrência de múltiplos resultados corretos [JALOTE, 1994]. Para estes casos a votação majoritária não se aplica, devendo ser substituída por estratégias alternativas de votação. Como **votação inexata**, onde valores de saída são considerados corretos, desde que estejam dentro de uma faixa de tolerância. Em geral a votação inexata é utilizada na validação de saídas oriundas de operações matemáticas com ponto flutuante.

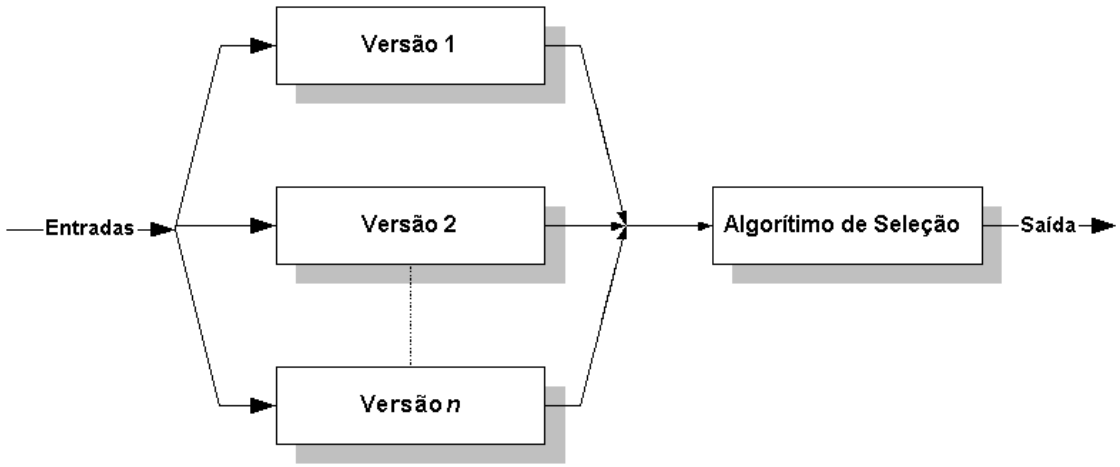


Figura 3-5 Modelo de Programação N-Versões

Segundo [KNIGHT & LEVESON, 1986], é impossível assumir a independência de falhas decorrentes de falhas de projeto ou concepção. No entanto diversas técnicas de validação de N-Versões de software têm sido desenvolvidas nos últimos anos. O objetivo é detectar e eliminar falhas de projeto em especial, as **falhas de especificação de requisitos**.

Independentemente da não possibilidade de se garantir a total independência de falhas de concepção, essas falhas somente poderão causar problemas se um número majoritário de versões do software for acometido da mesma falha de projeto. Este tipo de situação na prática é razoavelmente minimizada pelo uso da diversidade de projeto conjuntamente com metodologias padronizadas e recomendadas para garantia de qualidade de software [KNIGHT & LEVESON, 1989].

3.3.2.3 Blocos de Recuperação por Consenso

[SCOTT, 1987] propôs os **blocos de recuperação por consenso**, que essencialmente é uma abordagem que combina os blocos de recuperação discutido na Seção 3.3.2.1 e a programação N-Versões na Seção 3.3.2.2. O objetivo é melhorar a confiabilidade do sistema se comparada com a conseguida usando apenas uma destas técnicas isoladamente. A motivação básica para esta proposta é que segundo o autor, os testes de aceitação em blocos de recuperação carecem de uma formalização, o que implica numa efetividade menor que a esperada. Além do mais a complexidade envolvida no desenvolvimento de testes é considerável, por outro lado o uso de algoritmos de seleção (votadores) na programação *N-Version* pode não ser apropriada em muitas situações reais. Em especial aquelas cujo processo sob controle admite múltiplas saídas corretas.

Tomando por base o exposto no parágrafo anterior, a técnica de blocos de recuperação por consenso usa um algoritmo de seleção como uma primeira camada de proteção e decisão. Caso uma falha seja detectada neste estágio, uma segunda camada composta pelos testes de aceitação usuais ou similares aqueles dos blocos de recuperação seriam acionados. A Figura 3-6 detalha o modelo lógico da abordagem de blocos de recuperação por consenso.

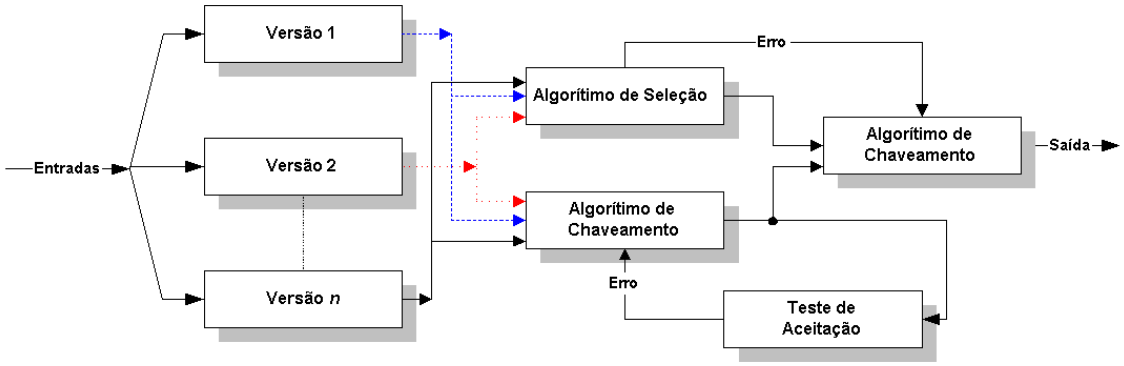


Figura 3-6 Bloco de Recuperação por Consenso

Algumas considerações adicionais sobre esta técnica merecem ser ressaltadas. A confiabilidade esperada por esta abordagem, como mencionado anteriormente, deve ser maior que a alcançável pelo uso de uma das técnicas base isoladamente. Alguns estudos de confiabilidade comprovam esta premissa, mesmo levando em consideração o aumento considerável da complexidade de desenvolvimento. Os módulos de software resultantes podem ter potencialmente uma maior confiabilidade [SCOTT, 1987], no entanto, isto não é totalmente garantido, apenas um estudo de verificação de confiabilidade pode confirmar essa premissa. Pois também o aumento de complexidade pode afetar substancialmente o projeto do software, tendo como resultante um sistema menos confiável que o esperado.

3.3.3 Injeção de Falhas no Software para Avaliação da Tolerância a Falhas

Ao processo de teste de módulos de software sob condições de anormalidades previstas, tais como geração de entradas impróprias ou manipulação forçada de estados internos, dar-se o nome de injeção de falhas no software *(Software Fault Injection - SFI).* Em sistemas tolerantes a falhas, objetiva-se com esta prática avaliar a efetividade dos mecanismos de tolerância empregados no projeto, bem como o próprio projeto de software.

Como discutido no Capítulo 2, é muito difícil garantir a corretude do software, devido a vários fatores, essencialmente devido a sua complexidade inerente. Baseado nesta premissa as técnicas *SFI* tentam mostrar o comportamento do sistema sob a presença de falhas. O que proporciona informações úteis relacionadas aos pontos falhos do software sob determinadas circunstancias. De posse destes dados, é possível tornar o código menos susceptível aos efeitos danosos verificados. Como conseqüência tem-se uma melhor robustez nos mecanismos de tolerância a falhas considerados no

projeto. Minimizando a probabilidade de propagação de determinadas falhas nas saídas críticas do sistema.

Segundo [AVRESKY, 1992], o processo *SFI* pode ser usado para atingir dois importantes objetivos em sistemas de missão e segurança crítica *(dependable and safety systems)*: **Eliminação de Falhas e Previsão de Falhas**. O primeiro se refere à possibilidade que o processo *SFI* proporciona de avaliar a efetividade do software, se o mesmo for utilizado como parte da estratégia de testes. Torna-se possível uma avaliação dos algoritmos e mecanismos de tolerância a falhas no sentido de sua corretude. O segundo se refere à possibilidade de se avaliar a robustez de partes do software ou de sua plataforma. Como por exemplo, taxas de falhas do sistema operacional ou mesmo a latência associada ao tratamento da falha. Esta última significa o tempo decorrido a partir de um instante de observação de uma falha e sua manifestação através de um erro.

[LAI, 1995] afirma que a abordagem *SFI*, tem duas importantes vantagens em relação a abordagem tradicional de testes de software (*test cases*), considerando seqüências normais de entradas:

- Possibilitar num ambiente controlado, dentro de uma janela de tempo limitada, ensaios acelerados das taxas de falhas do sistema.
- Melhor conhecimento acerca do comportamento de algumas características do sistema, como propagação de erros e padrão de saídas, quando determinados mecanismos de tolerância a falhas são submetidos sistematicamente a injeção de falhas.

Segundo [POMALES, 2000] existem basicamente dois modelos básicos de *SFI***:** *Injeção de Falhas e Injeção de Erros.*

A **injeção de falhas** tenta simular prováveis falhas de projeto, através da modificação do código fonte, de várias formas, como substituir o código de determinados componentes por outro semanticamente diferente. Já a **injeção de erros** objetiva a modificação de estados (dados) do sistema no intuito de simular a manifestação de falhas. Na prática existem várias maneiras de modificar estados de um sistema, tais como: Sistema operacional pode prover suporte para modificar a prioridade de processos, o que possibilitaria atribuir baixa prioridade a processos de alta prioridade. Ferramentas de depuração (*debuggers*) em geral permitem que se modifiquem variáveis de estado em tempo de execução. Mecanismos de gerenciamento

de mensagens poderiam ser usados para simular corrupção de dados nas camadas de comunicação inter-processos.

É relevante observar que os métodos de *SFI*, constituem uma classe dinâmica de testes de software. A mesma só tem sentido, no contexto do software em tempo de execução, onde seqüências particulares de entradas e perfis de estados internos podem ser seguidos ou manipulados.

Numerosas pesquisas têm sido realizadas no campo de metodologias e sistemas de testes experimentais de software, do qual os métodos *SFI*, fazem parte. Para uma referência mais completa sobre o tema, podem ser consultados os trabalhos de [IYER, 1996], [KOOPMAN, 1997] e [LEE, 1995], dentre muitos outros.

3.4 Redundância Modular Tripla *(TMR)*

A Redundância Modular Tripla *(TMR - Triple Modular Redundancy)* constitui um exemplo de técnica de redundância largamente utilizada, em geral com objetivo de mascarar falhas. Nela estão presentes os conceitos de replicação ativa de hardware e de programação em versões múltiplas ou única, tendo como objetivo garantir níveis de alta confiabilidade, disponibilidade e integridade dos dados.

Em um sistema *TMR* são usadas três versões (iguais ou diferentes) de componentes de hardware e/ou software, os quais implementam uma determinada lógica funcional. As saídas destes três módulos são conectadas a um elemento votador o qual realiza uma votação majoritária, neste caso uma votação 2 de 3. Isto é, duas das 3 saídas necessitam serem idênticas para que a saída final seja considerada correta, conforme mostrado na Figura 3-7.

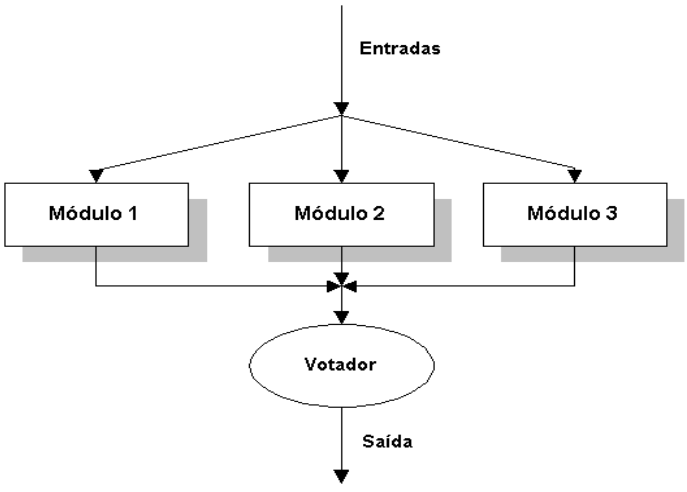


Figura 3-7 Redundância Modular Tripla (*TMR*)

Na verdade a redundância modular tripla é um caso particular de uma modalidade genérica de redundância denominada **Redundância Modular N** ***(NMR)***, ver Figura 3-8, onde devemos considerar um número *N* impar (quantidade de módulos redundantes). Essa abordagem genérica tolera até $(N-1)/2$ falhas. O custo é fator limitante do uso de configurações *NMR* em aplicações práticas, exceto em situações muito específicas onde os requisitos de *dependabilidade* sejam extremamente críticos. Como exemplos reais podem ser citados o '*Space Shuttle da NASA'* com arquitetura *NMR* (N=5) e projetos ligados a aviação civil como: *Boeing 777* e *Airbus*, todos com seus sistemas computacionais de bordo em arquitetura *NMR*.

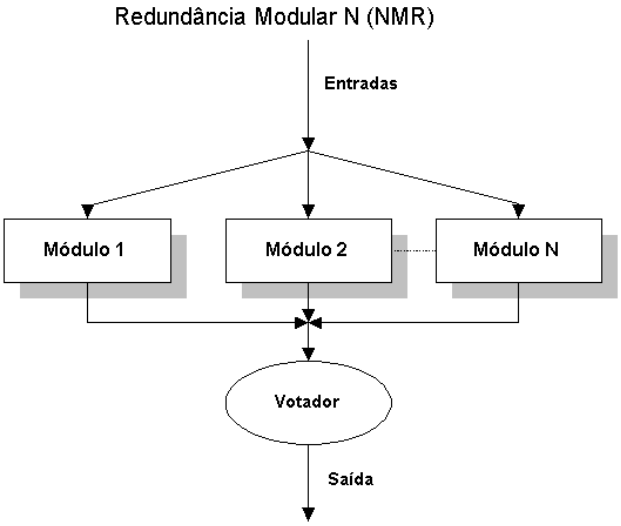


Figura 3-8 Redundância Modular N (NMR)

A modalidade *TMR* é mais usual por ser a mais simples arquitetura *NMR* possível, em sua configuração convencional, como mostrado na Figura 3-7. Porém existem variações interessantes, como *TMR* com estágio de votação triplicada, conforme mostrado na Figura 3-9.

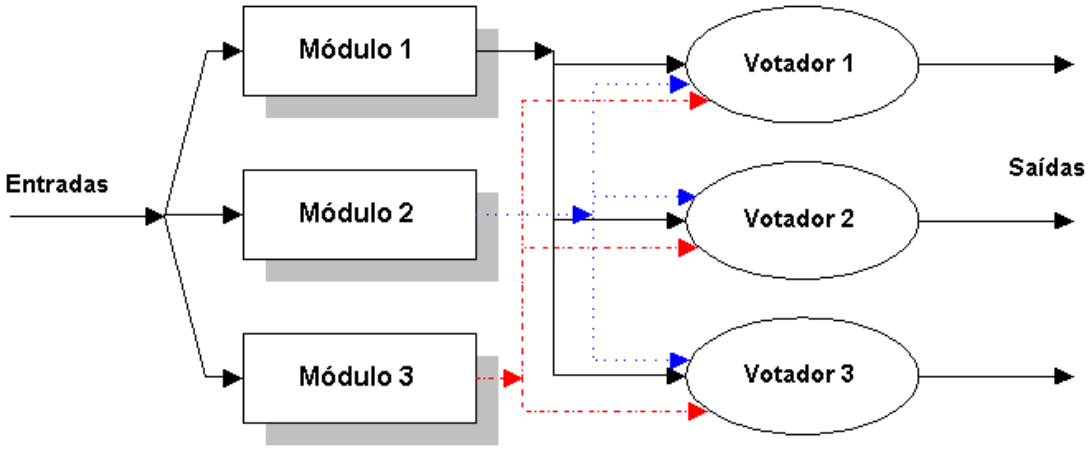


Figura 3-9 Redundância Modular Tripla (TMR) Triplicada

Neste caso, 2 das 3 saídas redundantes somente serão consideradas corretas se e somente 2 dos 3 votadores estivem funcionando corretamente.

Mais uma vez dependendo dos requisitos de confiabilidade e disponibilidade do sistema de controle, a arquitetura com estágio de votação triplicada pode ser extendida para uma abordagem de múltiplos estágios, conforme Figura 3-10.

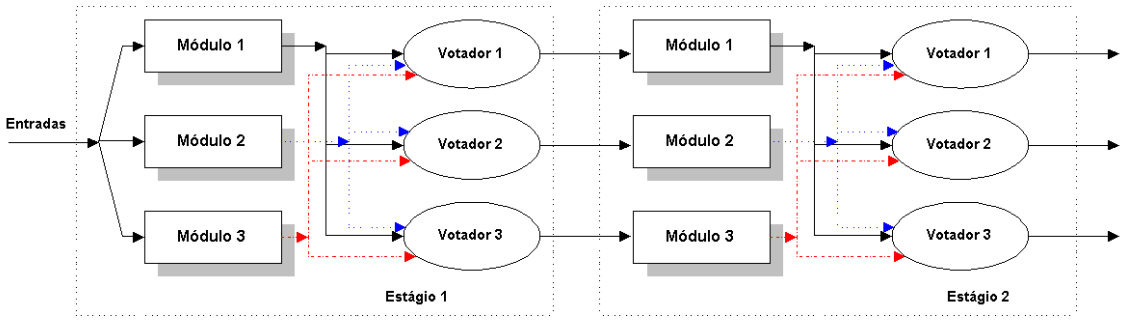


Figura 3-10 Redundância Modular Tripla (TMR) em Múltiplos Estágios

Neste caso específico pode-se verificar uma implementação de algumas técnicas de tolerância a falhas teoricamente mostradas nas seções 3.1, 3.2 e 3.3. Por exemplo, o propósito de múltiplos estágios é prover um nível maior de detecção e correção de falhas por meio de mascaramento, isto é, falhas devem ser mascaradas entre os estágios e também serão confinadas em um estágio específico.

Um dos principais problemas relacionados com a arquitetura *TMR* é a sincronização entre as saídas dos módulos redundantes, para o estagio de votação. O uso de *clock* de sincronismo comum é bastante utilizado no caso de votadores baseados em hardware. Protocolos de sincronização são utilizados em votadores baseados em

software. Uma discussão mais detalhada sobre votadores será apresentada na Seção 3.4.2.

Podemos ressaltar algumas das vantagens relevantes da arquitetura *TMR*, as quais são motivadoras para seu uso extensivo em sistemas tolerantes a falhas:

- Prover a possibilidade de mascaramento de falhas
- Cobertura tanto de falhas permanentes como falhas intermitentes
- Dispensa a necessidade de estágios prévios de detecção de falhas, devido à capacidade de imediato mascaramento das mesmas.
- Relativa facilidade de conversão de sistemas redundantes convencionais para *TMR*.

Há numerosas referências sobre o uso e análise de confiabilidade relacionadas a arquitetura *TMR*. Pode-se encontrar uma análise matemática mais detalhada das relações de confiabilidade da arquitetura *TMR* em [BRAGA, 1997], [CAMARGO JR., 1996b].

3.4.1 Arquitetura TMR Distribuída

Como será visto no Capítulo 4, sugere-se uso da arquitetura *TMR* distribuída numa rede de computadores *LAN*, como ambiente de destes do simulador de controle de tráfego ferroviário. De fato o estudo de caso proposto neste trabalho possibilita a aplicação e ensaios de diversas técnicas de tolerância a falhas discutidas anteriormente. A Figura 3-11 mostra a plataforma de testes proposta para os ensaios do simulador. Deve ficar claro que muitos dos requisitos de segurança são dependentes da metodologia de projeto utilizada, conforme os critérios discutidos no Capítulo 2.

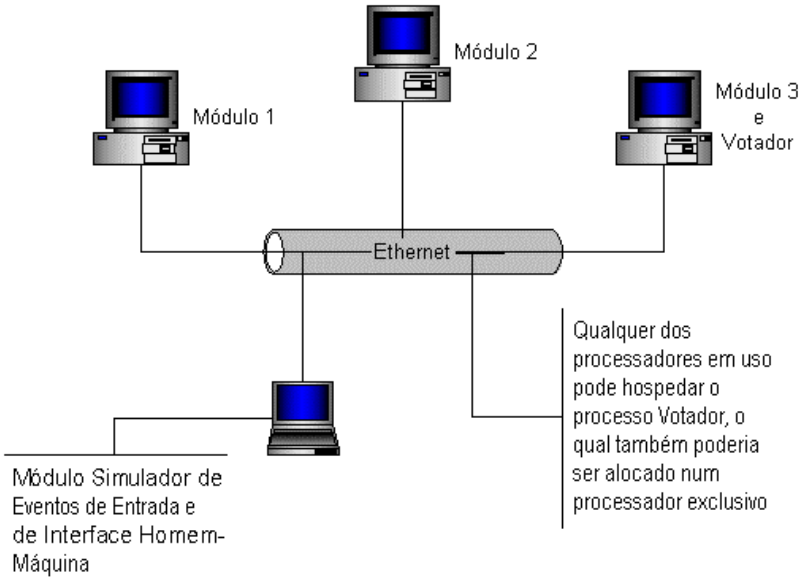


Figura 3-11 Redundância Modular Tripla Distribuída em Rede Local

Seguem algumas das principais questões decorrentes desta abordagem, que devem ser fortemente consideradas no desenvolvimento real de uma aplicação prática.

- O mecanismo de seleção de saídas (votador) deverá ser implementado por software, utilizando quaisquer das técnicas discutidas na Seção 3.4.2.1.
- Uso de um protocolo de sincronização de mensagens para garantir um processo de votação adequado.
- Uso um mecanismo seguro de integridade de mensagens entre os módulos redundantes e o módulo votador, como códigos de redundância cíclica *(CRC)*, conforme mostrado na Seção 3.5.
- A aplicação desta abordagem em processos de tempo real, ainda é dependente da infra-estrutura de comunicação. Sendo válido no contexto de simulação.
- Constitui uma plataforma de testes válida, para fins de simulação, onde se pode avaliar a efetividade de técnicas de software tolerante a falhas. Pode-se também verificar a corretude da lógica funcional de negócio, independentemente da disponibilidade de arquiteturas proprietárias de hardware e software.

3.4.2 Votadores

Segue uma breve discussão sobre as principais abordagens encontradas para resolver questões relativas à seleção de saídas provenientes de arquiteturas redundantes,

em que a alta confiabilidade é um requisito imperativo tanto por questões de segurança crítica (*safety*) ou disponibilidade (*availability*).

A escolha do tipo de votador a ser utilizado numa determinada aplicação depende de alguns fatores, conforme segue:

Considerações de Hardware

- Análise de Custo/Beneficio comparando vantagens e desvantagens dos requisitos de velocidade e custo relativo ao número de votadores a ser usado:
    - Velocidade no qual o processo de votação deve ser executado (em geral um votador implementado por hardware é significativamente mais rápido que um processo de votação por software).
    - O número de diferentes votadores.
- Relativa simplicidade na escolha do votador, no entanto necessita de um mecanismo de sincronização temporal tolerante a falhas.

Considerações de Software

- Um votador implementado por software necessita de uma interface tolerante a falhas com os módulos redundantes, para obter os dados de saídas.
- Deverá haver na arquitetura em uso a disponibilidade de um processador que possa executar o processo de votação, não necessariamente um processador exclusivo.
- Flexibilidade requerida para futuras mudanças no mecanismo de votação. Neste caso a escolha de uma implementação baseada em software é bastante vantajosa.

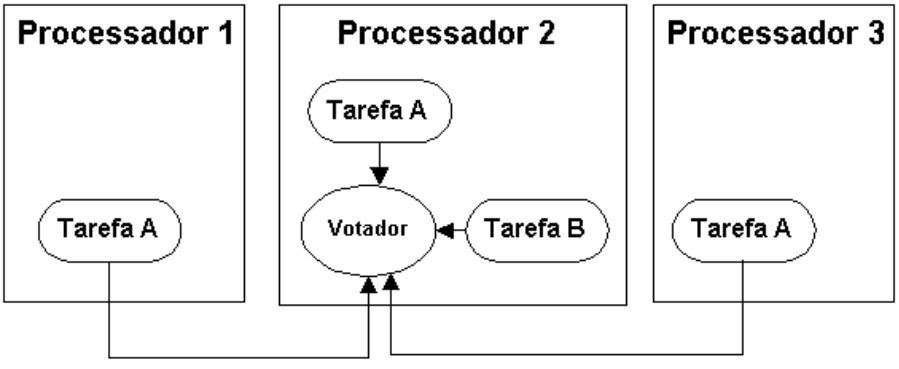


Figura 3-12 Votador em Processador Compartilhado

A Figura 3-12 acima ilustra uma das vantagens do uso de votadores baseados em software, que é a possibilidade de compartilhar processadores já destinados a outras funções para também hospedar o processo responsável pelo algoritmo de seleção de saída. Neste caso específico o processador 2 além de processar réplicas das tarefas A e B também executa o processo de votação final da arquitetura *TMR*.

A seguir serão apresentadas técnicas aplicáveis no desenvolvimento de mecanismos de votação baseados em software e algumas arquiteturas de projeto de votadores baseados em hardware. Todas estas abordagens são aplicáveis a sistemas *TMR*.

3.4.2.1 Votadores implementados por software (Algoritmos de Seleção de Saídas)

Testes de aceitação implementados em software, apesar de simplificar a abordagem de seleção de saídas, muitas vezes não são suficientes para garantir níveis de confiabilidade requeridos em determinadas aplicações. Aplicações estas com razoável grau de complexidade em termos de redundância de hardware ou software, como *TMR*, por exemplo. A razão é a complexidade no desenvolvimento de testes de aceitação efetivos, devido ao nível de acoplamento dos testes com a aplicação [SHIN, 1989]. Testes de aceitação avaliam se determinados resultados são aceitáveis, mas não especificamente corretos [POMALES, 2000].

Níveis suplementares de validações se fazem necessários, principalmente nos casos em que são empregadas múltiplas versões de software de controle. Nesta camada da arquitetura redundante, algoritmos de seleção de saídas devem ser usados, em especial, quando todas as versões dos componentes da arquitetura redundante são consideradas igualmente confiáveis. Desta forma a seleção das saídas deve ser baseada em critérios de comparação cruzada entre as saídas disponíveis de cada versão.

Em aplicações de missão e segurança críticas, a escolha ou o desenvolvimento de mecanismos de votação para seleção de saídas devem considerar as conseqüências de uma seleção errada por tais mecanismos. A corretude das saídas de um módulo de software de versão única, corretamente desenvolvido, deve ser aceitável para uma faixa de entrada razoável. Em implementações de versões múltiplas, as técnicas de votação por comparação cruzada devem ser desenvolvidas tendo em mente que a seleção da saída deve ser correta com altíssima probabilidade.

Algumas considerações adicionais devem existir dependendo do foco principal da aplicação.

- Se os principais requisitos são de segurança crítica (*Safety Critical*) os algoritmos de seleção empregados (votadores), devem detectar saídas erradas de tal forma a evitar a propagação destes resultados às saídas principais do sistema. Saídas inconsistentes podem implicar em situações de riscos catastróficos. Em geral nestas aplicações os algoritmos votadores geram alarmes apropriados associados à condição de erro encontrado, levando necessariamente o sistema para um estado de segurança intrínseco. Procedimentos de recuperação ou reinicialização são adotados para que o sistema volte a um estado que possibilite novamente alcançar níveis de alta confiabilidade na seleção das saídas.
- Caso o foco principal seja a disponibilidade (*Availability)*, o algoritmo de seleção de saídas pode considerar que uma saída válida sempre será disponibilizada, independentemente de sua corretude. Esta abordagem é aceitável se a execução subseqüente do software não é dependente de saídas possivelmente incorretas geradas previamente.

Um processo genérico de seleção de saídas baseado em duas etapas (**filtragem e seleção final por arbitragem**) foi proposto por [ANDERSON & LEE, 1986]. A etapa de filtragem objetiva a eliminação das saídas provenientes de versões individuais que podem ser detectadas como incorretas através de testes de aceitação diretamente implementados no código. Nestes casos são avaliados critérios relativos à corretude de dados, temporizações e outras características determinantes. Após a etapa de filtragem as saídas são repassadas para o estagio de seleção da saída principal, onde um algoritmo de seleção (votador) é utilizado para produzir a saída final.

Várias técnicas de votação têm sido propostas por vários pesquisadores no campo de software tolerante a falhas e sistemas de missão crítica (*dependable systems*).

[CROLL, 1995] propôs um algoritmo de votação baseado em técnicas de inteligência artificial. O comportamento básico é similar a de um **votador majoritário** ou de pluralidade, onde a determinação da saída final é baseada em critérios de similaridade ou proximidade (faixa aceitável) das saídas dos módulos redundantes. Havendo divergência o algoritmo de seleção se comporta como um **votador de média ponderada**, o qual associa pesos com base no registro estatístico de falhas obtido

durante o funcionamento na modalidade de votador majoritário. Os registros de falhas contêm informações sobre divergências detectadas relativas a valores e bases temporais de cada versão individual da arquitetura redundante.

[BASS, 1995] propôs o uso de um **votador predidivo**, o qual se baseia no histórico dos resultados prévios para produzir um valor de saída esperado. A seleção da saída final é baseada na comparação de proximidade dos valores de saída de cada versão individual com esse valor esperado.

[LORCZAK, 1989] apresentou quatro variedades de votadores para uso em sistemas redundantes: ***Votador Majoritário Formalizado, Votador Mediano Generalizado, Votador de Pluralidade Formalizado e Votadores de Média Ponderada***. Todos baseados na definição de *espaços métricos*[2], que pode ser encontrada em detalhes no seu trabalho.

Em essência estes votadores se baseiam na seguinte abordagem: Os valores de saídas resultantes dos módulos redundantes são considerados iguais (por meio de uma votação inexata) se suas distâncias métricas (ver definição de espaços métricos) forem menores que um determinado valor predefinido ε. No **Votador Majoritário Formalizado**, as saídas das versões são comparadas segundo a igualdade de seus espaços métricos. Se mais da metade desses valores forem concordantes, o votador seleciona o valor do grupo concordante com sendo a saída final.

3.4.2.2 Votadores Implementados por Hardware

Votadores implementados por hardware são largamente utilizados em sistemas redundantes de missão e segurança críticas, principalmente em arquiteturas proprietárias de hardware.

Em caráter ilustrativo são mostrados na Figura 3-13 votadores "*hardware based"* simples:

---

[2] Definição de espaço métrico [LORCZAK, 1989]:
Seja $\boldsymbol{X}$ o espaço de saídas do módulo de software (conjunto de saídas possíveis). Seja *d* um número real de uma função definida pelo produto cartesiano $\boldsymbol{X}$ x $\boldsymbol{X}$ com as seguintes propriedades: (1) d(x,y) = 0; (2) d(x,y) = 0 $\Rightarrow$ x = y; (3) d(x,y) = d(y,x); (4) d(x,z) = d(x,y) + d(y,z), $\forall$ x,y,z $\in$ $\boldsymbol{X}$. Logo ($\boldsymbol{X}$,*d*) é um espaço métrico.

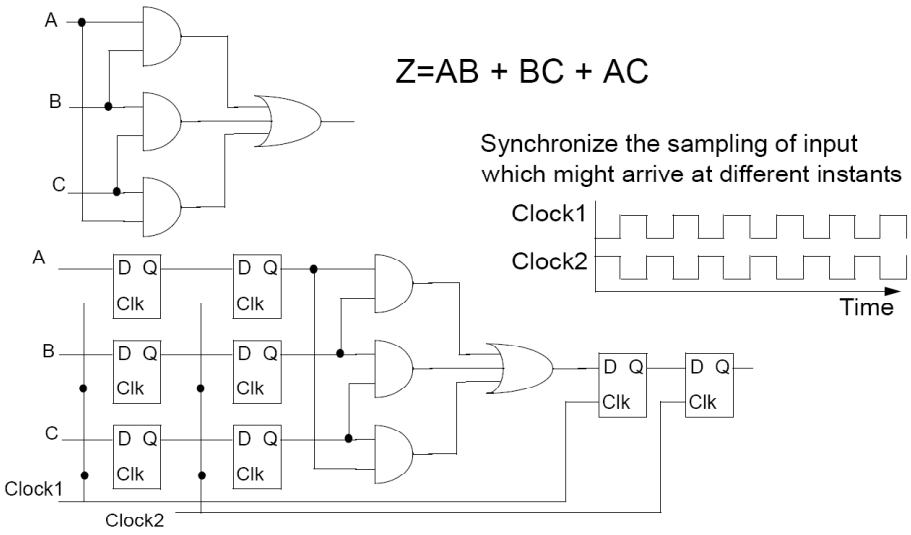


Figura 3-13 Circuitos Lógicos de Votadores Simples

Existem projetos de votadores desenhados especificamente para uso com a arquitetura *TMR*. Como é o caso de um votador denominado "*word-voter*" proposto por [MITRA, 2000], o qual realiza a votação de palavras binárias. Este tipo de votador é uma alternativa mais confiável que os votadores convencionais baseados na abordagem "*bit-wise*" que realizam a votação bit a bit.

3.5 Códigos de Redundância Cíclica (CRC)

Em sistemas de comunicação nos quais são exigidos requisitos de alta confiabilidade, são implementados mecanismos que garantem a integridade das mensagens transmitidas. Em essência, estes mecanismos devem prover meios de detecção de erros com alta confiabilidade, caso mensagens tenham sido corrompidas por distúrbios externos, como ruído no meio de comunicação. Existem algoritmos que atendem em diversos níveis de confiança os requisitos acima expostos. Alguns mais simples usados em aplicações não críticas, outros mais sofisticados, que permitem a detecção de diversas modalidades de erros. Uma das técnicas mais utilizadas é a codificação das mensagens, com o objetivo de prover meios ao receptor de verificar se as mensagens recebidas estão corrompidas ou não. Dentre estas técnicas, uma largamente difundida que se enquadra nesta categoria é o ***CRC*** (*Cyclic Redundancy Code*).

A mesma abordagem de prevenção contra corrupção de dados também é largamente utilizada em dispositivos de armazenamento de dados. Pois soluções de

espelhamento em dispositivos redundantes, apesar de ideal, em geral apresentam alto custo. Desta forma a codificação dos dados é empregada com objetivo de possibilitar que rotinas de verificação periódicas possam detectar se dados foram corrompidos. Algoritmos de codificação usando *CRC* são bastante utilizados.

Dependendo da criticidade da aplicação algoritmos de correção de erros são também utilizados, após a fase de detecção dos mesmos.

3.5.1 Considerações Básicas

Algoritmos que implementam *CRC* executam ciclicamente operações matemáticas nos bytes de dados de informação, o que essencialmente significa um algoritmo cíclico que gera informação redundante, deixando intacta a informação original [PETERSON, 1972].

Como dito anteriormente o *CRC* executa operações matemáticas simples em um bloco de dados. Estas operações retornam um número como resultado, o qual representa unicamente o conteúdo da informação original além de sua organização posicional. Desta forma este número pode ser utilizado para validação dos dados originais através de comparação. Deve-se a esta flexibilidade e relativa facilidade de implementação a difusão do uso do *CRC* em sistemas de comunicação para verificação da integridade de dados transmitidos ou de dados armazenados.

A idéia básica do *CRC* é representar as mensagens como números binários que por sua vez são equivalentes aos coeficientes de um polinômio algébrico. Procedendo-se com a divisão destes números por um número binário fixo, denominado **polinômio gerador**, o qual é escolhido criteriosamente para atender níveis específicos de capacidade de detecção de erros. O resto da divisão acima mencionada é considerado o *CRC* resultante da operação. Desta forma na recepção pode ser executada a mesma operação de tal forma a possibilitar a comparação do *CRC* calculado e transmitido. Este procedimento será mais formalmente apresentado na Seção 3.5.2. que segue.

É importante ressaltar que todas as operações aritméticas realizadas no calculo do *CRC*, basicamente adição, subtração e divisão são efetivadas em aritmética módulo 2, a qual apresenta propriedades particulares se comparada a aritmética módulo 10 usual. Seguem algumas propriedades relevantes:

- A adição e subtração produzem os mesmos resultados, e podem ser substituídas por uma operação lógica ***XOR***.
- Todas as operações não levam em consideração o "*carry*"

A Figura 3-14 mostra as tabelas verdade das operações com aritmética módulo 2.

+	0	1
0	0	1
1	1	0

–	0	1
0	0	1
1	1	0

X	0	1
0	0	0
1	0	1

/	0	1
0	*	*
1	0	1

Figura 3-14 Tabelas Verdade (Aritmética Módulo 2)

3.5.2 Teoria de Operação e Representação Polinomial

Os códigos de redundância cíclica executam todos os cálculos baseados na aritmética polinomial, com poucas diferenças se comparada com os conceitos tradicionais da álgebra.

Os componentes da operação da divisão polinomial (dividendo, divisor, quociente e resto) são representados por números binários que equivalem a polinômios com coeficientes binários.

Mais formalmente [PETERSON, 1972], podemos descrever a teoria de operação do código de redundância cíclica (*CRC*) conforme a seguir:

- Considerar uma mensagem de dados *M* com *k bits* de tamanho, sendo que o módulo transmissor gerará um bloco adicional de *n bits* denominado genericamente de *FCS* (*Frame Check Sequence*), neste caso específico *CRC*. O bloco resultante a ser transmitido consiste de *(k + n) bits* o qual é exatamente divisível por um número pré-determinado *P* a ser denominado de polinômio gerador.
- O receptor por sua vez realiza uma divisão polinomial binária (*módulo 2*) da mensagem recebida pelo mesmo número (polinômio gerador). Se não houver

resto nesta divisão, assume-se que a mensagem transmitida não foi corrompida, isto é, não há erros.

Simbolicamente temos:

$T = (k + n)$, frame de bits a ser transmitido, com $n < k$

M = mensagem com os primeiros k bits de T

$F = n$ bit FCS, os últimos n bits de T

P = Polinômio Gerador de tamanho $n + 1$ bits

Deseja - se que T / P não tenha resto na divisão

Multiplicando M por 2^n, temos :

$T = 2^n M + F$

Significando deslocar n bits a esquerda e colocando "0" nas posições deslocadas do resultado. Ao adicionarmos F concatena - se M a F

Queremos que T seja divisível por P, desta forma suponhamos $2^n M$ dividido por P

$$\frac{2^n M}{P} = Q + \frac{R}{P} \qquad (3.1)$$

Temos um quociente e um resto. Como a divisão é binária, o resto sempre será um bit menor que o divisor. Usaremos o resto R como o FCS ou nosso CRC.

$T = 2^n M + R$

Verificando que R satisfaz a condição :

$$\frac{T}{P} = \frac{2^n M + R}{P}, \qquad (3.2)$$

Substituindo 3.1 em 3.2, temos

$\frac{T}{P} = Q + \frac{R}{P} + \frac{R}{P}$, como na adição binária módulo 2, $(R + R) = 0$ temos que :

$\frac{T}{P} = Q + \frac{R + R}{P} = Q$

Não há resto na divisão, logo T é divisível por P

Desta forma vemos que o código *CRC* para uma mensagem de dados é facilmente gerado. Bastando para isso dividir $\frac{2^n M}{P}$ e usar o resto da divisão como o código *CRC* a ser adicionado na mensagem principal. Na recepção, o receptor irá dividir T por P não obtendo resto na divisão se não tiver havido erro na transmissão.

O divisor P a ser usado no cálculo de *CRC´s*, também conhecido como *polinômio gerador* tem de ser determinado com critério, isto é, o seu grau o qual determina o tamanho do código *CRC* está diretamente ligado com o poder de detecção

de erros. Dependendo do tipo de aplicação, existem polinômios geradores específicos, cuja determinação exige esforço na verificação de sua efetividade na detecção de combinações de erros prováveis, que podem ocorrer com os dados originais.

A representação simbólica de polinômios geradores largamente utilizados em implementações reais é apresentada a seguir:

$$G(x) = X^{12} + X^{11} + X^3 + X^2 + X + 1 \qquad (CRC-12)$$
$$G(x) = X^{16} + X^{15} + X^2 + 1 \qquad (CRC-16)$$
$$G(x) = X^{16} + X^{12} + X^5 + 1 \qquad (CRC-16\ CCITT)$$
$$G(x) = X^{32} + X^{26} + X^{23} + X^{22} + X^{16} + X^{12} + X^{11} + X^{10} + X^8 + X^7 + X^5 + X^4 + X^2 + X + 1 \quad (CRC-32)$$

3.5.3 Considerações Adicionais

Seguem os tipos de erros detectáveis por códigos de redundância cíclica *CRC*

- Todos os erros em um ou dois bits da mensagem original
- Todos os erros ocorridos em um número impar de bits.
- Todos os erros em grupos de bits de número menor ou igual ao grau do plinômio gerador
- A maioria, mas não a totalidade dos erros em grupos de bits com número superior ao grau do polinômio gerador.

Implementações de alta performance de *CRC* podem ser executadas por hardware, ou por algoritmos de software que usam uma abordagem denominada "*Table Driven*".

Uma discussão mais detalhada sobre a implementação real de algoritmos *CRC* pode ser encontrada em [WILLIAMS, 1993].

3.6 Desafios Vigentes

Apesar da evolução progressiva da confiabilidade de sistemas computacionais, muitas são as questões abertas que carecem de soluções definitivas. Vários são os desafios a serem superados. Seguem alguns os quais são relevantes no contexto deste trabalho, que embora não tenha sido focado especificamente nestas questões, podem ser consideradas com alvo de evoluções futuras.

- Como tirar proveito do modelo de sistemas distribuídos em aplicações críticas de tempo real. Em especial quando a plataforma utilizada não é constituída de hardware confiável, isto é, hardware com considerações de projeto que garantam falhas seguras ou tolerância a falhas. As questões de sincronismo de mensagens têm em muito sua complexidade aumentada, se tornando crítico para aplicações de segurança.
- Requisitos de alto desempenho impõem sérias restrições a requisitos de confiabilidade, disponibilidade e segurança (*safety*). Portanto como estabelecer um equilíbrio para atendimento de todos envolve uma razoável complexidade ainda não superada.
- O crescimento vertiginoso da complexidade de sistemas computacionais, principalmente no software, tornou a garantia de segurança ainda mais complexa. O desenvolvimento de metodologias e ferramentas automáticas que auxiliem na modelagem de sistemas de segurança minimizando falhas de concepção ou identificando-as em fases preliminares de projeto tem ocupado um grande número de pesquisadores em software de segurança crítica.

3.7 Considerações Finais

Neste Capítulo foram apresentados os principais conceitos e técnicas relativas a tolerância a falhas aplicada a sistemas computacionais tanto relativas a hardware como software. O uso da tolerância a falhas em sistemas de controle e segurança crítica constitui um fator fundamental no atendimento à requisitos de missão crítica como confiabilidade e disponibilidade contribuindo significativamente com questões relativas a segurança.

A arquitetura *TMR* foi enfatizada devido a sua aplicabilidade em sistemas reais de controle metro-ferroviário. Podemos notar sua aplicabilidade como base de um ambiente de testes baseado em LAN em substituição aos ambientes de hardware e software proprietários usuais na indústria. Adicionalmente esta arquitetura permite uma aplicabilidade de vários conceitos e técnicas conjugadas de tolerância a falhas apresentadas individualmente neste capítulo.

Um ponto a ser destacado pela relevância no contexto do trabalho é a utilização de mecanismos de seleção de saídas (votadores) baseados em software, devido a sua flexibilidade e não necessidade de alocação de processador exclusivo, o

que facilita a implementação de uma plataforma de teste, abstraindo-se das complexidades relativas aos votadores baseados em hardware.

4. Modelagem de Um Controlador de Tráfego Ferroviário

Neste Capítulo é detalhado um modelo simplificado de um controlador de tráfego ferroviário (intertravamento), no qual será baseado o estudo de caso a ser implementado no Capítulo 5.

O objetivo principal deste capítulo é o detalhamento formal de um modelo de controle de tráfego ferroviário, no contexto de um intertravamento, isto é, que implementa as regras de negócio e segurança do licenciamento de trens num trecho ferroviário. São introduzidos conceitos fundamentais sobre o controle de tráfego ferroviário, para então se detalhar o modelo de controle com as suas variáveis de entrada e saídas, as quais são base para definição do espaço de estados e transições que regem o comportamento dinâmico do modelo.

4.1 Controle de Tráfego Ferroviário

O controle de tráfego metro-ferroviário tem como principal objetivo automatizar o processo de licenciamento do tráfego de trens numa linha férrea. Tal processo de automação assume requisitos críticos devido a necessidade de controle seguro (*safety*) do processo. Composições ferroviárias em linhas com tráfego de média e grande densidade e com perfis de velocidade médio ou alto, exigem características de segurança crítica e também de *dependabilidade*.

A segurança em sistemas de controle metro-ferroviário desempenha papel fundamental, pois acidentes ferroviários em geral constituem eventos catastróficos podendo envolver grandes perdas humanas e materiais. Os primeiros sistemas de controle foram consolidados a partir dos conceitos de relés de falha segura (*Fail-Safe*) e lógica a relés. Estes atingiram seus propósitos de forma plena, no que se refere ao atendimento de requisitos de segurança, uma vez serem os mesmos comprovadamente imunes a vários tipos de interferências [CAMARGO JR., 1996].

Na Seção seguinte, são apresentados conceitos fundamentais relacionados ao controle de tráfego metro-ferroviário, bem como tecnologias aplicadas em diversas arquiteturas proeminentes e sua evolução tecnológica.

4.1.1 Definições Básicas e Fundamentais

Em termos gerais um sistema de controle de tráfego ferroviário pode ser dividido em dois subsistemas:

- Subsistemas de Via
- Subsistemas de Bordo

Os ***subsistemas de via*** também são denominados de sistemas de ***intertravamento***, os quais são responsáveis pelo processamento seguro das informações adquiridas dos ***elementos de via***. É relevante ressaltar que as permissões de circulação de trens sejam liberadas no trecho de domínio do intertravamento levando em consideração perfis de velocidade específicos ao contexto operacional.

Subsistemas embarcados podem genericamente ser divididos em dois subsistemas denominados ***Automatic Train Operation*** (*ATO*) e ***Automatic Train Protection*** (*ATP*). O primeiro é responsável pela operacionalidade do sistema e o segundo pelas funções de segurança, como por exemplo, supervisionar os limites máximos de velocidades permitidos num determinado trecho e aplicação de frenagem em caso de violação desses limites. Conforme ilustrado na Figura 4-1.

Os subsistemas mencionados acima, tanto de via como de bordo, que implementam funções de segurança, são projetados tendo como premissa o princípio de *falha segura* (***Fail-Safe***). Tal princípio essencialmente estabelece que na presença de falhas previsíveis de hardware e de forma mais controversa de software, o sistema deve ser conduzido para um estado de segurança intrínseco. No caso dos subsistemas de bordo significa parada dos trens e dos subsistemas de via o bloqueio para os trechos de seu domínio. Em outras palavras os trens devem ser parados com alto grau de confiabilidade e segurança e nenhuma permissão de circulação de trens deve ser emitida por um intertravamento no seu trecho de domínio.

Seguem os elementos fundamentais os quais são controlados por um típico sistema de controle de tráfego metro-ferroviário, conforme Figura 4-1:

- **Trem** (composto basicamente por locomotivas e vagões, os quais apresentam grande variedade em função de tecnologias específicas).
- **Via** (representa os pátios de cruzamento, zonas de manobra e trechos).
- **Equipamentos de via**, relacionados a seguir, com suas respectivas funções:

- ***Circuito de Via***: Destinados a prover a função de detecção segura da presença de trens em determinados trechos de via. Prover indicações de ocupação/desocupação desses trechos as quais são analisadas pelo sistema de controle de tráfego, dentro de um contexto específico.
- ***Máquinas de Chave***: Equipamentos utilizados para definir a direção de percurso de um trem. Em geral são dispositivos comandados local ou remotamente pelo sistema de controle. O comando das máquinas de chaves em geral é elétrico, no entanto em condições de emergência podem ser comandadas por alavanca manual. Apresentam duas posições básicas: *Normal e Reversa*. A posição normal direciona o trem na mesma direção em que ele se encontrava, enquanto que a posição reversa indica mudança de direção de percurso. Outro importante aspecto de uma máquina de chave, relacionada com segurança é a sua condição de travamento, que indica a não possibilidade de movimentação da mesma, quando ela se encontra no estado travada. Esta condição é garantida pela retirada de alimentação elétrica do motor de deslocamento de posição.
- ***Sinaleiros***: Também conhecidos como *bloqueios* indicam a permissão ou proibição de entrada dos trens numa determinada região.
- ***Dispositivos de Código de Velocidade***: Equipamentos que transmitem códigos de velocidade aos trens, em um determinado trecho de via.
- ***Dispositivos de Detecção de Caixa/Roda Quentes***: Equipamentos de monitoração das temperaturas de rolamentos e rodas de vagões. O objetivo é identificar aqueles cujos eixos estão com alta probabilidade de rompimento, o que poderia causar graves acidentes.
- ***Dispositivos de Detecção de Descarrilamento***: Dispositivo de via com objetivo de detectar se algum trem está descarrilado.

As Figuras 4-1 e 4-2 ilustram os conceitos e definições abordadas anteriormente sobre subsistemas de via e bordo respectivamente.

Implementações reais de sistemas de controle de tráfego podem diferir em vários aspectos, como conceituais, funcionais, em parte devido a critérios operacionais ou mesmo de custo, como também do tipo de tecnologia aplicada.

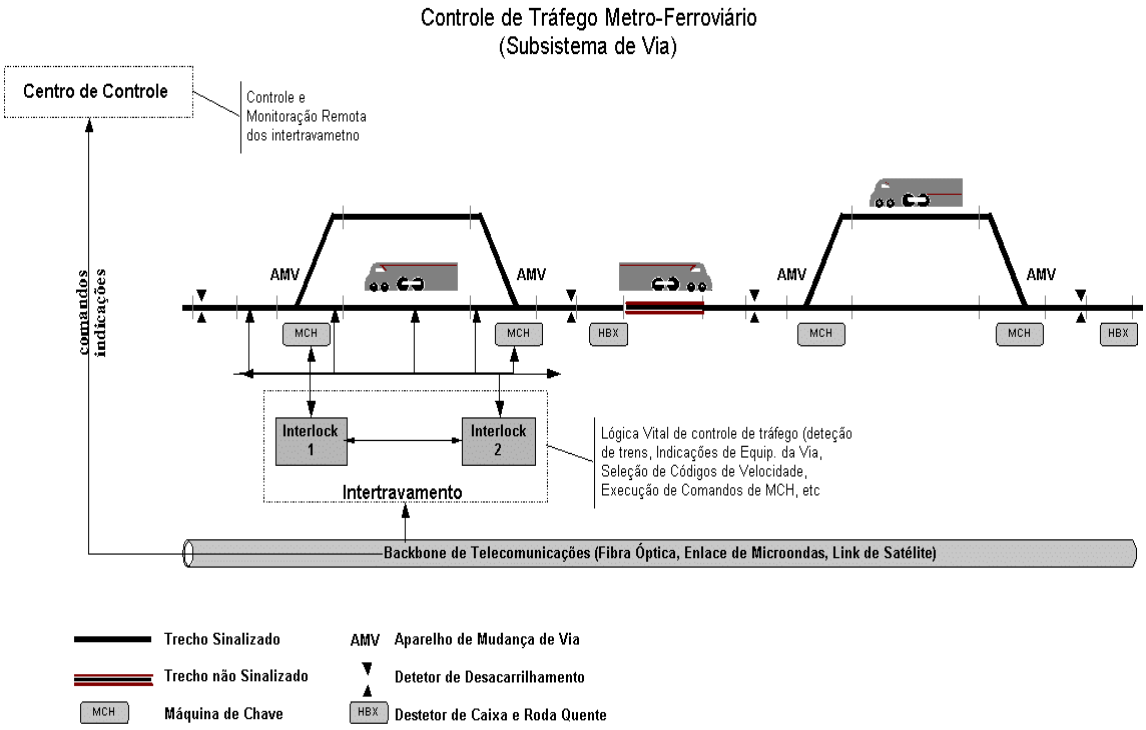


Figura 4-1 Controle de Tráfego Metro-Ferroviário (Subsistema de Via)

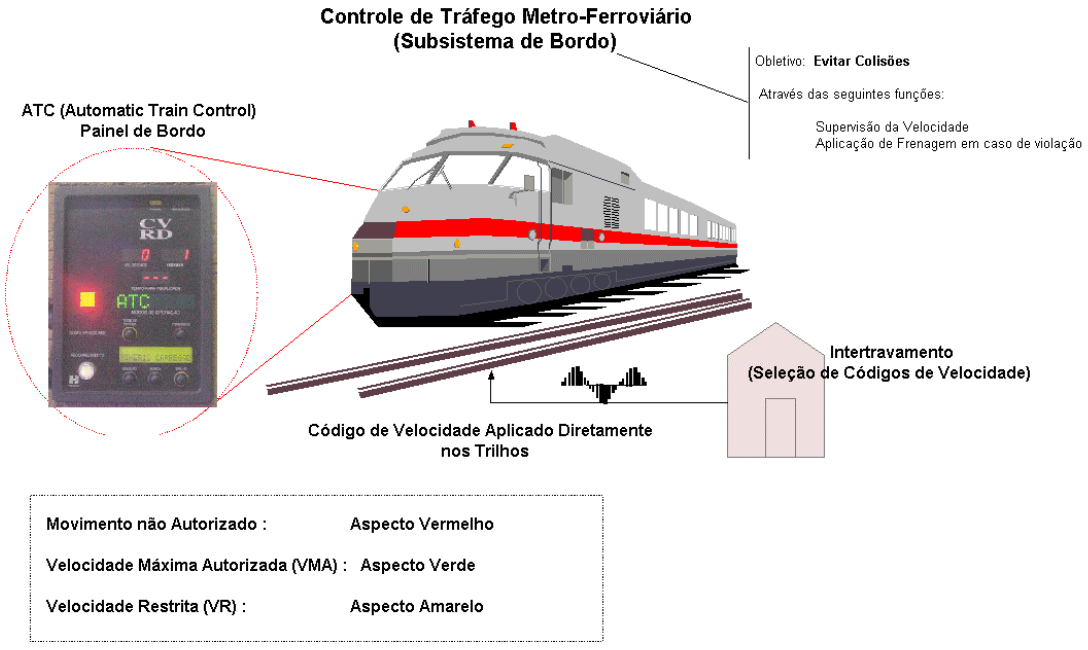


Figura 4-2 Controle de Tráfego Metro-Ferroviário (Subsistema de Bordo)

4.1.2 Controle de Tráfego Centralizado

O Controle de Tráfego Centralizado constitui uma arquitetura bastante utilizada no controle de tráfego metro-ferroviário. O objetivo é a partir de um único

ponto denominado de *CTC* (*Centralized Traffic Control*), efetivar o controle e monitoração dos subsistemas de campo, denominados de intertravamento vitais. Estes, conforme mencionado anteriormente, controlam de forma segura a movimentação dos trens num determinado trecho ferroviário.

Em geral o *CTC*, possui interfaces com equipamentos de comunicação de dados, de tal forma que haja conectividade com os intertravamento distribuídos ao longo da via. Existe uma grande variedade de arquiteturas atualmente aplicadas a *CTC*, variedade esta também refletida na diversidade de tecnologias aplicadas, tanto de software quanto de hardware. A Figura 4-3 mostra a arquitetura geral de um *CTC* típico de uma aplicação metro-ferroviária.

Atualmente a complexidade dos sistemas de controle de tráfego ferroviário tem aumentado vertiginosamente. Evolui-se de um patamar que tradicionalmente limitava-se a supervisão e controle dos subsistemas de via, com funções tradicionais de telecomandos e monitoração em tempo real de indicações do estado da via e equipamentos de campo. Para um escopo bem mais abrangente, no qual o uso extensivo de novas tecnologias de software e comunicações tornou-se proeminente.

Tais tecnologias tem permitido uma extensão funcional dos *CTC´s* tradicionais para arquiteturas integradas que possibilitam um melhor controle e monitoração remota dos trens em movimento. Um exemplo é uso de tecnologias de rastreamento *positivo* com o uso do *GPS* (Sistema de Posicionamento Global) e *transpondes*[3] instalados na via. Estes aspectos são evidenciados nas Figuras 4-3 e 4-4.

[3] Dispositivos eletrônicos passivos ou ativos que armazenam informações que podem ser lidas através de um conjunto leitor e antena. A leitura se faz com a irradiação de RF por uma antena que ao atingir o transponder possibilita que o mesmo transmita de volta o conteúdo gravado em sua memória de dados.

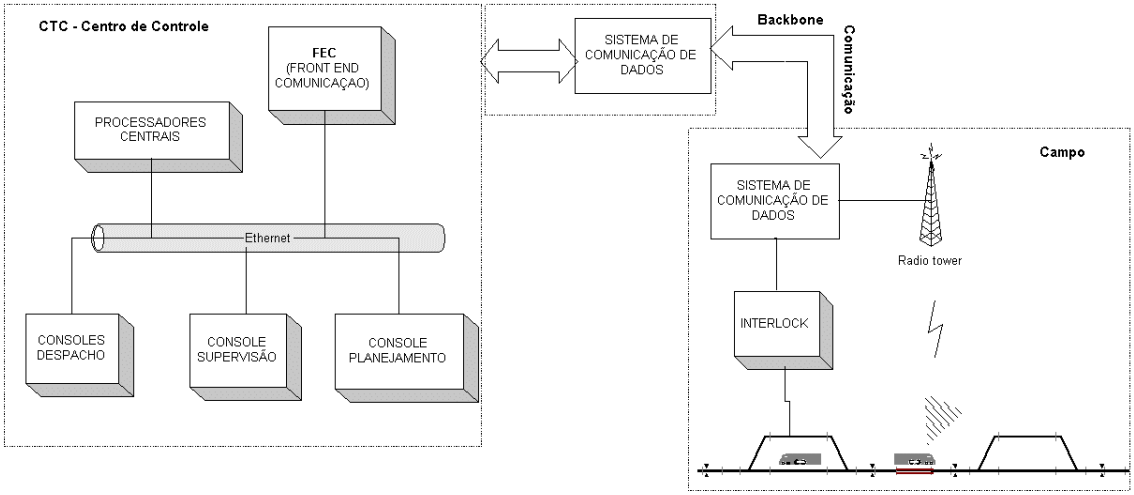


Figura 4-3 - Arquitetura de um Centro de Controle Metro-Ferroviário

Consequentemente a complexidade no desenvolvimento do software de controle de tais sistemas tem crescido bastante. Isto tem levado a uma demanda razoável no desenvolvimento de normalizações de garantia de qualidade de software que assegurem o sucesso no desenvolvimento de grandes e complexas aplicações.

A expansão de funcionalidades nos *CTC´*s tem sido evidenciada pela inclusão de funções que transcendem o controle dos dispositivos de campo, incorporando apoio a manutenção e o gerenciamento da ferrovia [VERNAZZA, 1990].

A Figura 4-4 mostra o modelo conceitual da arquitetura de um *CTC* real. São evidenciados componentes que refletem funcionalidades tradicionais e avançadas. Estas somente presentes em arquiteturas com tecnologia de hardware, software e comunicações de dados avançadas [BRAGA, 2003].

Basicamente temos os módulos principais do software aplicativo (*Core System*), responsáveis pelas funções de processamento de alto nível provenientes de requisições dos usuários finais, bem como o tratamento das informações provenientes dos subsistemas de campo. Funções como *rastreamento de trens* também é implementada nesta parte do sistema. Existem interfaces tanto com os subsistemas de campo como com as aplicações clientes que provêm às funcionalidades de interface homem-máquina. Interfaces estas com objetivo de desacoplar os componentes de alto nível das especificidades de sistemas proprietários de campo.

Os principais subsistemas ilustrados abaixo são os de bordo e via. Nestas arquiteturas avançadas com recursos de comunicação de dados móvel, tem se utilizado de funções de rastreamento positivo baseadas no *GPS*, conforme explicado nas seções

anteriores. Um subsistema bastante emergente nas arquiteturas atuais é o de *identificação automática de vagões*, aqui genericamente designado por ***AEI*** *(Automatic Equipment Identification)*. Essencialmente responsável pela leitura de dispositivos de armazenamento de informações com funções de comunicação em rádio freqüência RF, denominados de *transpondes*. Estes identificam precisamente a composição de um comboio ferroviário. Interfaces com outros subsistemas relevantes ao controle e monitoração dos trens estão também disponíveis, como os dispositivos de detecção de caixas de rolamento e rodas quentes (*HotBox/Wheel*).

Devido à ênfase que se tem dado nas arquiteturas mais recentes de *CTC* a requisitos de gestão da ferrovia. Módulos de gerenciamento são providos no intuito de fornecer informações gerenciais resultantes da monitoração em tempo real do processo de circulação de trens. Desta forma interfaces com sistemas corporativos são implementadas, bem como a disponibilização de informações históricas a clientes externos. A Figura 4-4, que representa o modelo conceitual de um *CTC* de última geração (**Centro de Controle Operacional da EFC - CVRD**), ilustra estas idéias.

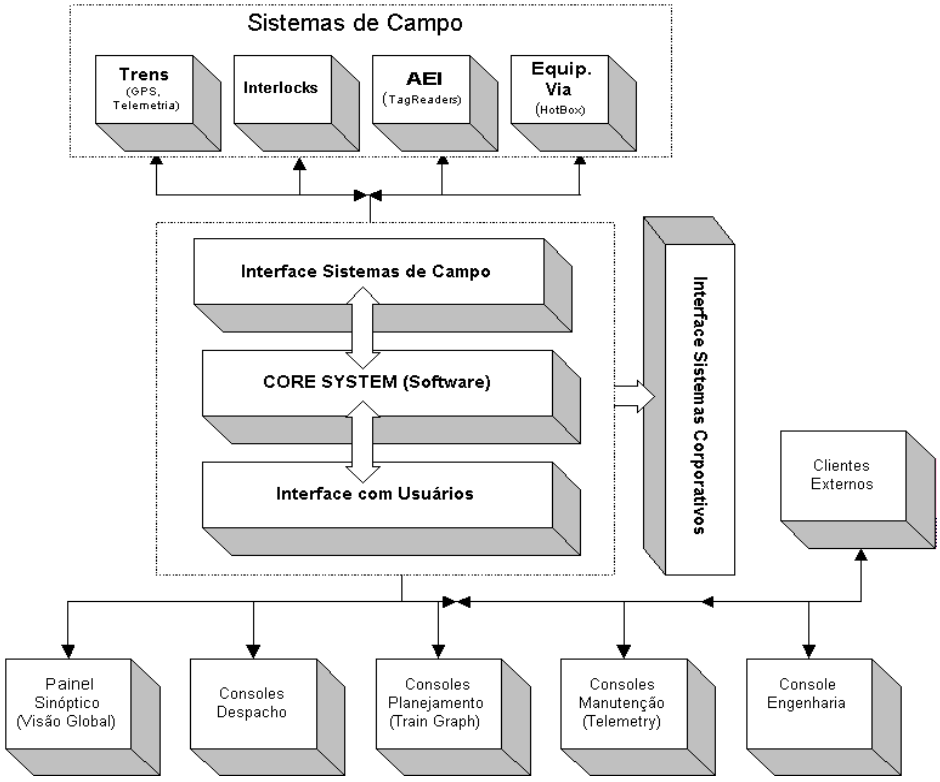


Figura 4-4 Centro de Controle Centralizado - Modelo Conceitual

4.2 Especificação de um Modelo de Controlador de Tráfego Ferroviário

Com objetivo de facilitar a compreensão de um sistema de controle de tráfego metro-ferroviário, algumas simplificações funcionais de sistemas reais foram adotadas, destacando os aspectos mais relevantes.

O modelo em questão trata-se de um controlador de via, não tendo sido considerado subsistemas de bordo. Adicionalmente, devido à complexidade da arquitetura de um controlador de via real, o qual possui vários módulos e equipamentos com funcionalidades diversas, apenas o *módulo de controle* foi enfatizado. Portanto o módulo controlador é representado de forma mais significativa pelo modelo detalhado nas seções seguintes.

O modelo de um controlador de tráfego metro-ferroviário simples [CAMARGO JR. 1996] está conceitualmente definido abaixo, conforme Figura 4-5.

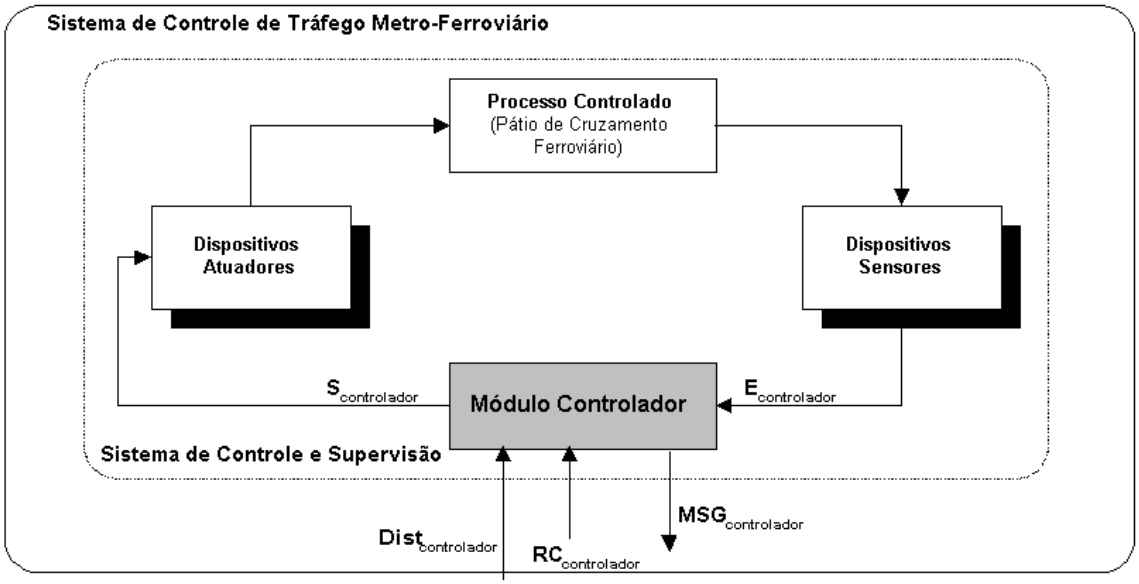


Figura 4-5 Modelo Conceitual de um Sistema de Controle e Supervisão

Temos acima um modelo básico de supervisão e controle aplicado ao processo de controle de tráfego ferroviário. Propositalmente o modelo foi simplificado, uma vez que queremos enfatizar apenas o *módulo controlador*, como sendo o elemento a ser considerado no detalhamento que segue.

Como ilustrado na Figura 4-5 o módulo controlador possui um conjunto de variáveis de entrada e saída, conforme descrição a seguir:

- Entradas
  - $E_{controlador}$ (Entradas provenientes de dispositivos sensoriados no campo)
  - $RC_{controlador}$ (Requisições de comandos dos operadores do sistema)
  - $Dist_{controlador}$ (Distúrbios externos que podem afetar o controlador)
- Saidas
  - $S_{controlador}$ (Saídas que afetam dispositivos atuadores em campo)
  - $MSG_{controlador}$ (Mensagens aos operadores do sistema)

4.2.1 Variáveis de Entrada e Saídas

Segue uma descrição sucinta de todas as entradas e saídas consideradas no modelo do controlador de tráfego metro-ferroviário. A Figura 4-1 pode ser referenciada para uma melhor visualização dos elementos ferroviários aos quais as entradas e saídas do controlador se relacionam.

Entradas do Controlador (*$\mathbf{E_{controlador}}$*)

- ***Ocupação e Desocupação dos Techos de Via.*** Informação proveniente do campo, que basicamente indica se um determinado trecho de via encontra-se ocupado ou não por um trem.
- ***Estados da Máquina de Chave (Normal, Reverso e Travada)***. Informação de posicionamento de máquina de chave, indicando se o elemento de posicionamento denominado "*ponta de agulha*", encontra-se na posição *Normal* ou *Reversa*. A indicação de travamento indica se a máquina de chave encontra-se eletricamente travada, o que se significa que nenhum movimento da mesma será possível.
- ***Indicações dos Aspectos de Sinaleiro (Aberto, Fechado)***. Os sinaleiros indicam basicamente o estado de bloqueio para o cumprimento de rotas. Estas por sua vez definem o percurso autorizado de um trem na entrada e saída de pátios de cruzamento num trecho ferroviário. Sinaleiro *aberto* significa liberação de percurso, enquanto que *fechado* significa bloqueio, indicando proibição de percurso.

- ***Indicações de Código de Velocidade***. Informação que reflete o código de velocidade selecionado para um determinado trecho ferroviário, de acordo com um contexto específico.

Saidas do Controlador ($\mathbf{S_{controlador}}$)

- ***Comando de posicionamento da máquina de chave*.** Comandos que posicionam uma máquina de chave em uma das duas posições válidas: *Normal* ou *Reversa*, conforme solicitado ao controlador.
- ***Comando de travamento da máquina de chave***. Comando que coloca uma máquina de chave num estado que impossibilita um movimento da mesma.
- ***Comando de Geração de Código de Velocidade***. Comando que envia a um determinado trecho de via um código de velocidade específico, o qual deverá ser detectado pelo trem, quando o mesmo atingir este trecho.
- ***Comando de Abertura e Fechamento de Sinais***. Comandos destinados a predispor os bloqueios de entrada e saída de um trecho ferroviário em um estado de abertura ou fechamento, que respectivamente liberaria ou proibiria o percurso de uma rota para um trem.

Requisições de Comandos do Sistema (*$\mathbf{RC_{controlador}}$ *)

- ***Iniciação do Controlador***. Requisição para ativar a seqüência de ativação do controlador
- ***Terminação do Controlador***. Requisição para ativar a seqüência de terminação do controlador.
- ***Requisição de Alinhamento de Rota***. Requisições de alinhamento de rotas, que essencialmente significam uma autorização de movimentação para um trem. Uma rota estabelece um percurso para um trem, o qual pode conter alguns elementos de via definidos anteriormente, tais como: Trechos de Via, Máquinas de Chave, Sinaleiros. Neste modelo uma rota estabelecida significa a possibilidade segura de percurso de um trem. Pois o trecho de destino da mesma encontra-se desocupado, bem como os circuitos de via das máquinas de chave que estão no percurso da mesma. Essas por sua vez estão posicionadas convenientemente e travadas e os bloqueios de entrada da rota (sinaleiros) estão

abertos e com códigos de velocidade específicos já estabelecidos na via. Este últimos serão detectados pelo subsistema de bordo do trem.

- ***Requisição de Cancelamento de Rota***. Requisição de cancelamento de rota efetuado pelo operador de tráfego. Refere-se a rotas já previamente estabelecidas como descritas no item anterior.
- ***Requisição de Proibição de Alinhamento de Rota***. Requisição efetivada pelo operador de tráfego. Com objetivo de impedir temporariamente que rotas com destino a determinados trechos sejam estabelecidas. Em geral em situações de anormalidade ou manutenção.
- ***Requisição de Cancelamento da Proibição de Alinhamento de Rota***. Requisição efetivada pelo operador de tráfego, no intuito de retirar a condição de proibição de alinhamento de rotas descrita no item anterior. Em geral como a retirada de bloqueios e proibições são procedimentos que envolvem alguns critérios de segurança, são recomendados níveis de confirmação auxiliar para ratificação da execução de comandos desta natureza.

O módulo controlador pode sofrer *Distúrbios Externos***,** essencialmente caracterizados por interferências eletromagnéticas, eletrostáticas e correntes de fuga, os quais podem afetar o correto funcionamento do sistema [CAMARGO, JR. 1996].

Mensagens geradas pelo controlador, no intuito de exteriorizar aos operadores e mantenedores do sistema, constituem um importante recurso que pode ser usado para o diagnóstico e reposição do sistema em caso de falhas. Elas melhoram também características de *usabilidade*, pois facilitam o uso correto do sistema mostrando condições específicas dos estados do processo, falhas, confirmação de requisições de comandos e histórico de eventos.

A implementação da lógica funcional deste controlador como será descrito na Seção 4.3, utiliza a abordagem de *máquinas de estado*. As transições dos estados definidos são regidas por variações nas entradas do controlador que de uma forma geral pode ser dividido em dois objetos denominados: *Objeto de controle* e *Objeto funcional*. As entradas, provenientes do campo ou operadores do sistema, são inicialmente tratadas pelo *objeto funcional,* o qual enviará o resultado ao *objeto de controle*. A estes resultados dar-se o nome de *predicados de entrada*. Analogamente as saídas do *objeto de controle* dar-se o nome de *predicados de saída* que são enviados ao *objeto funcional.*

Este por sua vez baseado nestes predicados gerará um conjunto de saídas que constituem o resultado final para o acionamento de dispositivos atuadores na via. Este fluxo de dados e sinais de controle está ilustrado na Figura 4-6.

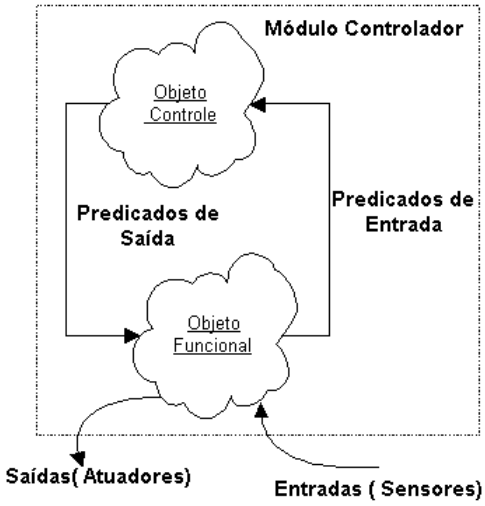


Figura 4-6 Fluxo de Dados do Módulo Controlador (Predicados de Entrada e Saída)

Essencialmente a máquina de estado do controlador processará os predicados de entrada e saída do sistema. Portanto é essencial o entendimento destes elementos para a compreensão do funcionamento do controlador de tráfego ferroviário simulado descrito na Seção 4.3. As Tabelas 4-1 e 4-2 mostram respectivamente uma descrição detalhada de todos os predicados de entrada e saída previstos no módulo controlador.

Tabela de Predicados de Entrada da Máquina de Estados (Objeto de Controle)	
inicia	Sinal para inicialização do Controlador
fim_inicia	Sinal de fim do processo de inicialização do controlador
termina	Sinaliza inicio do processo de terminação do controlador
fim_termina	Sinaliza final do processo de finalização do controlador
req_alin_rota	Sinal de requisição de uma rota
rota_ok	Sinaliza que uma rota foi completamente estabelecida
rota_nao_ok	Sinaliza que a uma rota requisitada não pode ser estabelecida
travamento_ok	Sinaliza travamento de uma ou mais máquinas de chave associadas ao percurso de uma rota
travamento_nao_ok	Sinaliza que uma ou mais máquinas de chave

	associadas ao percurso de uma rota requisitada não tiveram seu travamento efetivado.
req_canc_alin_rota AND (¬timeout)	Sinaliza uma requisição de cancelamento de uma rota, a qual não se encontra em contagem de tempo de segurança[4].
timeoutcanc	Sinaliza final de contagem de tempo de segurança ou seu cancelamento
req_bloq_alin_rota	Sinaliza uma requisição de bloqueio de alinhamento de rota para um determinado trecho de via
req_canc_bloq_alin_rota	Sinaliza uma requisição de cancelamento de bloqueio de alinhamento de rota vigente em um determinado trecho de via.
perigo_superado	Sinaliza que o sistema foi conduzido para um estado não mais caracterizado por riscos iminentes. Significa a saída de estado de risco, dentro de regras pré-estabelecidas.
liberação	Sinaliza a liberação para o controlador retornar ao um estado normal de operação do sistema, após ter saído de um estado de risco.

Tabela 4-1 Predicados de Entrada

Tabela de Predicados de Saída da Máquina de Estados (Objeto de Controle)	
ativa_inic	Comanda que o processo de inicialização do controlador deve ser ativado.
ativa_control	Ativa as funções de controle do controlador de tráfego ferroviário.
ativa_term	Comanda a ativação do processo de terminação do controlador.
desativa_control	Desativa as funções de controle do controlador de tráfego ferroviário.
ativa_alin	Ativa processo de alinhamento de rotas para um determinado trecho ferroviário.
cmd_mch AND trav_mch	Comando de máquina de chave para uma posição válida e comando de travamento desta máquina de chave.
cmd_absinal AND cmd_codveloc	Comando de abertura de sinal e comando de envio de um código de velocidade válido.
cmd_fecsinal AND cmd_codveloc AND ativacancaut	Comando de fechamento de sinal e comando de envio de um código de velocidade válido e ativação de procedimento de cancelamento automático de uma rota.

[4] Contagem de Tempo de Segurança, também conhecido "Timelocking", significa o tempo que uma determinada rota permanece travada, quando é recebida uma requisição de cancelamento. Em geral serve para garantir que um trem não tenha sua rota cancelada de imediato, quando o mesmo já se encontrava com sinal de abertura indicando liberação de percurso. Caso o trem inicie o cumprimento da rota concedida, antes do termino do tempo, a requisição de cancelamento não será efetivada, caso contrário sim, após o termino da contagem de tempo de segurança.

cmd_fecsinal AND cmd_codveloc AND ativacanc	Comando de fechamento de sinal e comando de envio de um código de velocidade válido e ativação de procedimento de cancelamento de uma rota.
Msg AND cmd_codveloc AND cmd_fecsinal	Comando de fechamento de sinal e comando de envio de um código de velocidade válido e uma mensagem ao controlador
Msg	Mensagens alusivas aos estados do processo, condições de falha, confirmação de requisições de comando e histórico de eventos.

Tabela 4-2 Predicados de Saída

4.2.2 Estados e Transições

Segue uma descrição dos estados considerados na modelagem adotada do controlador de tráfego ferroviário, o qual será responsável pela liberação da circulação de trens num trecho ferroviário conhecido como "pátio de cruzamento". Conforme Figura 4-1.

Esta estruturação em estados com sua semântica específica é derivada de [CAMARGO JR. 1996], constituindo um modelo representativo de implementações reais de controle de tráfego metro-ferroviário vitais.

	ESTADOS	***DESCRIÇÃO***
0	Repouso	Neste estado o sistema está desligado, não se encontrando ativadas suas funções de controle e supervisão.
1	Inicialização	Estado no qual as variáveis de via e controle estão sendo inicializadas, bem como estão sendo ativadas as funções de supervisão
2	Espera	Estado no qual o sistema se encontra com suas funções de controle e supervisão ativadas, aguardando requisições do operador, eventos de entrada da via ou eventos gerados pelo próprio módulo controlador.
3	Terminação	Estado no qual o sistema se encontra em processo de finalização ou desligamento, suas funções de controle e supervisão estão sendo desativadas, no entanto as condições de via e do próprio módulo controlador continuam sendo verificadas.
4	Alinhamento Rota	Estado no qual o sistema processará uma requisição de alinhamento de rota. Uma série de verificações deve ser efetivada para que se possa liberar a rota em questão. As verificações devem iniciar pela checagem de não ocupação dos trechos de via que fazem parte do percurso da rota e do não travamento das máquinas de chave relativas a rota. Numa fase posterior a verificação acima descrita são enviados comandos para o posicionamento e travamento das máquinas de chave envolvidas. Após a confirmação da efetivação desses comandos a rota é então

		liberada. Liberação esta efetivada através de comandos para abertura de sinais e geração de códigos de velocidade específicos.
5	Cancelamento Automático de Rotas	O sistema entra neste estado quando o trem atinge o 1º trecho de via envolvido na rota. Neste estado na medida em que o trem cumpre a rota, os trechos de via já percorridos são liberados. O mesmo acontece com as máquinas de chave as quais são destravadas e liberadas. Quando o trem atingir o último trecho de via da rota (destino da mesma) o sistema deve transitar para o estado de espera.
6	Cancelamento Pelo Operador	Neste estado o operador solicitou explicitamente um cancelamento de uma rota previamente alinhada ou em alinhamento. Deve-se prever um "*timeout*" para temporizar a possibilidade de requisição de cancelamento de uma rota cujo alinhamento iniciou-se anteriormente, mas ainda não foi completamente efetivada. Uma vez aceito a requisição de cancelamento, as seguintes ações devem ser tomadas: Fechamento dos sinais de entrada e código de velocidade totalmente restrito. Deve ser considerada outra temporização para que seja checado que não houve invasão do trecho de via inicial da rota. Havendo invasão o sistema deve transitar para o estado de cancelamento automático (**Estado 5**). Não havendo invasão a rota em questão é efetivamente cancelada e o sistema deve transitar para o estado de espera (**Estado 2**).
7	Bloqueio Alinhamento Rotas	O sistema transita para este estado quando uma solicitação de bloqueio é recebida do operador. Por se tratar de uma requisição restritiva, a mesma dever ter preferência sobre todas as outras. O que efetivamente se bloqueia é um trecho de via, então nenhuma rota com destino a este trecho poderá ser liberada. O sistema só poderá deixar este estado após o recebimento de uma requisição de cancelamento do bloqueio em questão.
8	Inseguro	Estado no qual o sistema não é conduzido espontaneamente. Para que o sistema venha a assumir este estado, é necessário que alguma condição incoerente aconteça. Neste modelo as condições decorrentes de falhas de hardware foram abstraídas por questões de simplicidade. Logo serão consideradas condições incoerentes eventos não esperados quando o sistema estiver num estado qualquer. Eventos estes que podem ser decorrentes da movimentação inadequada do trem, entradas impróprias do operador ou eventos espúrios decorrentes de falhas de software ou indiretamente de hardware.
9	Degradado	Estado atingido caso seja detectado um evento que leve o sistema para o estado inseguro (**Estado 8**) e o tratamento deste evento consiga transitar o sistema para este estado. Caracterizado por um nível de degradação funcional, porém seguro.

Tabela 4-3 Descrição dos Estados do Controlador de Tráfego Ferroviário

Seguem as Tabelas 4-4 a 4-8 que representam todas as possíveis transições dos estados considerados no modelo do controlador de tráfego.

Essencialmente estas tabelas descrevem o funcionamento do núcleo principal do controlador de tráfego metro-ferroviário exemplificado. Na Seção 4.3 pode ser visto alguns aspectos relacionados com a implementação da *máquina de estado* que executa todas as transições baseadas nas variações dos predicados de entrada, bem como as respectivas ações especificadas nos predicados de saída.

Basicamente a forma de ler estas tabelas é a seguinte: Tomamos uma linha que representa um estado válido do controlador e então para cada predicado de entrada (evento de transição) referenciado pela coluna, temos na intersecção a respectiva ação a ser tomada (predicado de saída) e o estado para qual o controlador deve transitar.

	inicia	**fim_inicia**	**termina**	**fim_termina**
Estado 0	ativa_inic/Estado 1	Null/Estado 0	Null/Estado 0	Null/Estado 0
Estado 1	Null/Estado 1	ativa_control/Estado 2	ativa_term/Estado 3	ativa_term/Estado 3
Estado 2	Null/Estado 2	ativa_term/Estado 9	ativa_term/Estado 3	ativa_term/Estado 3
Estado 3	Null/Estado 3	ativa_term/Estado 9	ativa_term/Estado 3	desativa_control /Estado 0
Estado 4	Null/Estado 4	ativa_term/Estado 9	ativa_term/Estado 3	ativa_term/Estado 3
Estado 5	Null/Estado 5	ativa_term/Estado 9	ativa_term/Estado 3	ativa_term/Estado 3
Estado 6	Null/Estado 6	ativa_term/Estado 9	ativa_term/Estado 3	ativa_term/Estado 3
Estado 7	Null/Estado 7	ativa_term/Estado 9	ativa_term/Estado 3	ativa_term/Estado 3
Estado 8	Null/Estado 8	ativa_term/Estado 9	ativa_term/Estado 3	ativa_term/Estado 3
Estado 9	Null/Estado 9	ativa_term/Estado 9	ativa_term/Estado 3	ativa_term/Estado 3

Tabela 4-4 Transição de Estados

	req_alin_rota	**rota_ok**	**rota_nao_ok**	**travamento_ok**
Estado 0	Null/Estado 0	Null/Estado 0	Null/Estado 0	Null/Estado 0
Estado 1	Null/Estado 1	Null/Estado 1	Null/Estado 1	Null/Estado 1
Estado 2	ativa_alin/Estado 4	Null/Estado 2	Null/Estado 2	Null/Estado 2
Estado 3	Null/Estado 3	Null/Estado 3	Msg/Estado 3	Null/Estado 3
Estado 4	´-/Estado4	cmd_mch AND trav_mch /Estado 4	Msg/Estado 4	cmd_absinal AND cmd_codveloc /Estado 2
Estado 5	Null/Estado 5	Msg/Estado 5	Msg/Estado 5	Msg/Estado 5
Estado 6	Null/Estado 6	Msg/Estado 6	Msg/Estado 6	Msg/Estado 6
Estado 7	Null/Estado 7	Msg/Estado 7	Msg/Estado 7	Msg/Estado 7
Estado 8	Null/Estado 8	Msg/Estado 8	Msg/Estado 8	Msg/Estado 8
Estado 9	Null/Estado 9	Msg/Estado 9	Msg/Estado 9	Msg/Estado 9

Tabela 4-5 Transição de Estados (cont.)

	travamento_nao_ok	ocp(1o trecho)	ocp(trecho_interm)	ocp(ultimo_trecho)
Estado 0	Null/Estado 0	Null/Estado 0	Null/Estado 0	Null/Estado 0
Estado 1	Null/Estado 1	Msg AND cmd_codveloc AND cmd_fecsinal/Estado 8	Msg AND cmd_codveloc AND cmd_fecsinal/Estado 8	Msg AND cmd_codveloc AND cmd_fecsinal/Estado 8
Estado 2	Null/Estado 2	cmd_fecsinal AND cmd_codveloc AND ativacancaut/Estado 5	Msg AND cmd_codveloc AND cmd_fecsinal/Estado 8	Msg AND cmd_codveloc AND cmd_fecsinal/Estado 8
Estado 3	Null/Estado 3	Msg AND cmd_codveloc AND cmd_fecsinal/Estado 8	Msg AND cmd_codveloc AND cmd_fecsinal/Estado 8	Msg AND cmd_codveloc AND cmd_fecsinal/Estado 8
Estado 4	Msg/Estado 2	Msg AND cmd_codveloc AND cmd_fecsinal/Estado 8	Msg AND cmd_codveloc AND cmd_fecsinal/Estado 8	Msg AND cmd_codveloc AND cmd_fecsinal/Estado 8
Estado 5	Msg AND cmd_codveloc AND cmd_fecsinal/Estado 8	Null/Estado 5	Null/Estado 5	Null/Estado 2
Estado 6	Msg AND cmd_codveloc AND cmd_fecsinal/Estado 8	Msg /Estado 5	Msg AND cmd_codveloc AND cmd_fecsinal/Estado 8	Msg AND cmd_codveloc AND cmd_fecsinal/Estado 8
Estado 7	Null/Estado 7	Msg AND cmd_codveloc AND cmd_fecsinal/Estado 8	Msg AND cmd_codveloc AND cmd_fecsinal/Estado 8	Msg AND cmd_codveloc AND cmd_fecsinal/Estado 8
Estado 8	Null/Estado 8	Null/Estado 8	Null/Estado 8	Null/Estado 8
Estado 9	Null/Estado 8	Null/Estado 8	Null/Estado 8	Null/Estado 8

Tabela 4-6 Transição de Estados (cont.)

	req_canc_alin_rota AND (~timeout)	timeoutcanc	req_bloq_alin_rota	req_canc_bloq_alin_rota
Estado 0	Null/Estado 0	Null/Estado 0	Null/Estado 0	Null/Estado 0
Estado 1	Null/Estado 1	Null/Estado 1	Null/Estado 1	Null/Estado 1
Estado 2	cmd_fecsinal AND cmd_codveloc AND ativacanc/Estado 6	Null/Estado 2	Msg AND cmd_codveloc AND cmd_fecsinal/Estado 7	Msg/Estado 2
Estado 3	Null/Estado 3	Null/Estado 3	Null/Estado 3	Null/Estado 3
Estado 4	Msg/Estado 2	Msg/Estado 4	Msg AND cmd_codveloc AND cmd_fecsinal/Estado 7	Msg/Estado 4
Estado 5	Msg/Estado 5	Msg/Estado 5	Msg AND cmd_codveloc AND cmd_fecsinal/Estado 7	Msg/Estado 5
Estado 6	Msg/Estado 6	Msg/Estado 2	Msg AND cmd_codveloc AND cmd_fecsinal/Estado 7	Msg/Estado 6
Estado 7	Msg/Estado 7	Msg/Estado 7	Null/Estado 7	Msg/Estado 2
Estado 8	Null/Estado 8	Null/Estado 8	Null/Estado 8	Null/Estado 8
Estado 9	Null/Estado 9	Null/Estado 9	Null/Estado 9	Null/Estado 9

Tabela 4-7 Transição de Estados (cont.)

	perigo_superado	liberação
Estado 0	Null/Estado 0	Null/Estado 0
Estado 1	Null/Estado 1	Null/Estado 1
Estado 2	Null/Estado 9	Null/Estado 2
Estado 3	Null/Estado 3	Null/Estado 3
Estado 4	Null/Estado 9	Null/Estado 4
Estado 5	Null/Estado 9	Null/Estado 5
Estado 6	Null/Estado 9	Null/Estado 6
Estado 7	Null/Estado 9	Null/Estado 7
Estado 8	Msg/Estado 9	Null/Estado 8
Estado 9	Null/Estado 9	Msg/Estado 2

Tabela 4-8 Transição de Estados (cont.)

4.2.3 Critérios de Completeza

[CAMARGO, JR. 1996] Definiu formalmente vários critérios de verificação de completeza, para validação de segurança de um sistema de controle de tráfego metro-ferroviário. Estes critérios são aplicados sobre um modelo de transição de estados [SHAW, 1992].

A seguir seguem alguns particularmente relevantes no entendimento da máquina de estado referente ao núcleo do controlador de tráfego ferroviário simulado, implementado neste trabalho. Estes critérios podem ser implementados na lógica funcional deste simulador.

- Deve ser efetivada uma análise de consistência nas variáveis de entrada do controlador, por exemplo: Um código de velocidade inconsistente (códigos enviados e recebidos não correspondem a um código válido). Neste caso o sistema deve ser conduzido ao *estado inseguro*, devendo ser gerado um predicado de saída alusivo a esta condição. Abordagem análoga (verificação de consistência) deve ser considerada com outras variáveis de entrada críticas como Ocupação/Desocupação de trechos de via. Os mesmos devem ser considerados ocupados em caso de detecção de inconsistências. Em caso de inconsistência relativa a posição de máquinas de chave, o posicionamento deve ser considerado *indefinido* sem o travamento respectivo. Inconsistências nas indicações de rotas devem levar a condição de rotas não concedidas. Antes do envio de comandos (posicionamento e travamento de máquina de chave, alinhamento de rota,

abertura e fechamento de sinal, etc.), devem ser verificadas as respectivas consistências para o envio. Se houver qualquer discordância nas verificações, os comandos devem ser abortados.

- O sistema deve ser inicializado num *estado seguro*. Para isto as seguintes precauções devem ser providas: Inicialização das variáveis de entrada do controlador devem corresponder ao estado seguro. Em outras palavras, todos os trechos de via devem ser *ocupados*, todas as posições de chave devem inicialmente *não possuir correspondência*, estados de travamento das máquinas de chave no estado de *não travadas*, códigos de velocidade *restritivos* para cada trem em um trecho de via. Os comandos de saída devem ser iniciados num estado de *não acionamento*. De forma análoga, o processo de *terminação* deve ser efetivado de forma segura. O sistema após em funcionamento só deve ser conduzido ao estado de repouso (**estado 0**), se o mesmo já estando no estado de terminação (estado 3) receber o evento de *fim de terminação.*
- Para cada estado do sistema são previstas transições de saída para cada predicado de entrada. Mas se em um dado momento nenhum predicado de entrada for válido o sistema deve permanecer no estado em que o mesmo se encontra por um determinado tempo. Logo algumas temporizações *(timeouts)* deverão ser definidas para alguns dos estados no modelo em questão.
- Devem ser atribuídas prioridades aos predicados de entrada, pois existe a possibilidade de mais de um deles ser verdadeiro num dado instante. As prioridades devem enfatizar as condições inseguras, isto é, na ocorrência de eventos simultâneos, os eventos que possuem predicados de saída correspondentes a situações de risco, devem ser prioritariamente processados.

- Deve ser avaliado o significado da permanência em demasia do controlador em um determinado estado. Para alguns a permanência indeterminada não implica em situações de anormalidade, os estados **0** (*repouso)* e **2** (*espera)* são exemplos. No caso do **estado 2** (*espera*) poderia ser problemático caso a aplicação tivesse que considerar a existência de tempos de *headway* (intervalos de chegada de trens em estações de metrô). Os demais estados **1, 3, 4, 5 e 6** certamente significam problemas. Os estados **7, 8 e 9** têm sua permanência por um tempo prolongado, num contexto de risco ou não, dependente de regras de negócio do sistema. A razão é que se trata de estados restritivos, degradação e inseguro.

- Considerações temporais devem ser avaliadas para cada variável de entrada. Pois uma aplicação real de controle de tráfego ferroviário pode ser considerada em diversos aspectos uma aplicação de *tempo real*. Para cada variável de entrada (ocupação de trecho de via, indicações de posição e travamento de máquinas de chaves, sinais de abertura e fechamento de sinaleiros, códigos de velocidade), deve ser determinada a noção de tempo de validade. De tal forma a indicar se esta informação está ou não envelhecida.

- Deve ser considerada a relação de dependência temporal entre cada predicado de entrada, pois em casos reais a seqüência temporal de alguns eventos é relevante.

- Deve ser avaliado a capacidade do sistema de processar numa taxa de tempo Δt uma quantidade Q de entradas do processo. Por exemplo: Quantas requisições de alinhamento de rotas podem ser processadas num dado intervalo temporal? Qual a capacidade de varredura dos elementos de campo? De tal forma que a atualização dos estados da via não envelheça causando interpretações errôneas do estado dinâmico do sistema. A capacidade que deve ser avaliada para algumas variáveis de entrada e saída tanto do ponto de vista mínimo como máximo. Por exemplo, qual o mínimo e máximo tempo admissível para uma variável que represente um estado de elementos de via seja atualizada? Logo é intuitivo concluir que a capacidade mínima e máxima neste caso é dependente da quantidade de elementos que podem ser atualizadas numa varredura através do subsistema de comunicação.

- É muito importante a monitoração da integridade física dos caminhos de aquisição de dados de entrada. Esta situação pode levar a desatualização do sistema em relação a indicações críticas. Caso esta condição seja detectada, o sistema deve ser levado ao estado *inseguro*, sendo emitidos alertas convenientes aos operadores. Com relação à falta de integridade física relativa aos comandos, que levaria a indisponibilidade dos mesmos, inclusive aqueles restritivos. Isto poderia gerar situações perigosas. Logo se detectado falta de integridade desta natureza, o sistema deve ser levado ao estado *inseguro*. Neste trabalho serão abstraídas questões de hardware, as quais seriam responsáveis pela monitoração de falhas de integridade física no sistema. A nível de software pode-se simular um estado desta natureza, a fim de que sejam testadas as ações que seriam

tomadas face à ocorrência de problemas com o caminho físico de alguma entrada ou saída crítica do controlador.

- Devem ser avaliadas as condições de sobrecarga da capacidade do sistema. Situação esta a ser abstraída da implementação, devido à necessidade de mecanismos de monitoração em tempo real. No entanto a situação de *falha* já detectada, pode ser emulada para que haja a transferência do sistema para o estado apropriado (*degradado*).
- Devem ser avaliadas as ações a serem tomadas quando o sistema transitar ao *estado degradado*. Por exemplo, deve ser analisado o tempo necessário de permanência neste estado, para que haja condições de estabilização da capacidade de entrada do sistema.
- Dependendo das características do processo sendo controlado é altamente relevante a análise e verificação das condições de sequenciamento de ações potencialmente perigosas. No estudo de caso em questão não existem seqüências de ações que levariam diretamente o sistema ao *estado inseguro*. Apenas inconsistências de estados e entradas de dados críticos podem levá-lo a este estado.
- Devem ser criteriosamente verificados para todos os estados, se para todos os predicados de entrada haverá ou não dentro de um intervalo temporal pré-determinado a ocorrência de outros predicados de entrada. De tal forma a possibilitar a identificação de ações a serem tomadas caso ocorram eventos dentro do tempo de latência considerado.
- Alguns aspectos de interação homem-máquina devem ser criteriosamente analisados. Um caso típico refere-se à situação de um trecho de via bloqueado para rotas, havendo uma requisição de desbloqueio e imediatamente um novo bloqueio. Isto pode caracterizar um desbloqueio acidental e a conseqüente tentativa de correção da entrada indevida. Existem algumas abordagens para lidar com este tipo de situação. Pode-se atribuir um tempo de latência para os pedidos de desbloqueio. Desta forma dentro deste intervalo, qualquer requisição de desbloqueio deve ser desconsiderada. Outra abordagem é um procedimento denominado de "*check before operate*", baseado no qual o sistema pede

explicitamente a confirmação do desbloqueio. Sendo aguardado um tempo específico, caso excedido, o desbloqueio não será efetivado.

- Devem ser definidos critérios de verificação contra-comandos, de tal forma que indicações espúrias sejam detectadas. Em outras palavras, existem determinadas indicações que somente devem mudar se houver comandos relativos. Neste caso o sistema deve transitar para o *estado inseguro*.
- Devem ser definidos critérios relativos a verificação entre os tempos de supervisão de indicações de entrada, que correspondem a ações solicitadas por uma variável de controle (comandos). Em geral devem ser determinados tempos mínimos e máximos, que tenham a seguinte semântica: Caso a informação de supervisão ocorra precocemente, pode indicar problemas com o sistema de supervisão. Caso ocorra de forma tardia, eventos alusivos a não efetivação do comando podem surgir mesmo este tendo sido realizado no campo. Para ambos os casos o sistema deve ser conduzido aos estados inseguro ou degradado, dependendo da criticidade da indicação de entrada.
- Verificações de percurso dos estados definidos no modelo constituem importante critério de detecção de situações anômalas. Essencialmente para cada estado definido, deve ser verificado qual o caminho que leva a este estado a partir do *estado de repouso*.
- Para cada estado definido, deve ser verificado qual o caminho não vazio que o leva ao mesmo estado.
- Verificações sobre a existências de incoerências (relativas a seqüência de ocorrência de predicados de entrada) que levem ao *estado inseguro*, devem ser efetivadas.

4.3 Considerações Finais

Neste Capítulo foram apresentados os conceitos fundamentais sobre o controle de tráfego ferroviário, sendo descrito formalmente um modelo simplificado de um módulo controlador (intertravamento ferroviário).

No Capítulo 5, este modelo servirá de base para a implementação do núcleo de um simulador de um módulo de controle de tráfego ferroviário. Todo o

comportamento dinâmico do modelo descrito nas tabelas de estados e transições será implementado através de uma máquina de estado.

A abordagem de máquinas de estado possibilita que técnicas de tolerância a falhas como a redundância modular tripla (*TMR*), seja mais facilmente explorada num ambiente de simulação, como será visto na Seção 5.2 do capítulo seguinte.

5. (Estudo de Caso) - Implementação do Controlador de Tráfego Ferroviário Simulado

Neste Capítulo descreveremos os aspectos relevantes da implementação do controlador de tráfego ferroviário simulado, como estudo de caso. O objetivo é prover um ambiente de testes onde os requisitos e regras do modelo descrito anteriormente nas seções 4.1 e 4.2 possam ser verificados em termos práticos. A implementação de todas as características do modelo apresentado fica reservada para trabalhos futuros. Tendo sido desenvolvido um núcleo do controlador e discutido vários aspectos relevantes que devem ser considerados na implementação plena.

5.1 Modelo com Máquinas de Estado Finitas

Essencialmente o núcleo do controlador é baseado numa máquina de estados *FSM (Finite State Machine).* Esta responde aos eventos de transição, gerados por um módulo denominado *gerador de eventos*, o qual coloca o simulador num dos estados correspondentes às regras definidas nas tabelas de transição. A dinâmica da máquina de estado é transitar de um estado a outro ou permanecer no estado atual de acordo com o contexto. Também é efetivada a atualização de variáveis internas no processo ou tomada de ações que devem ser traduzidas em saídas do módulo controlador. Esta dinâmica, bem como a arquitetura lógica da *FSM* implementada é ilustrada na Figura 5-1.

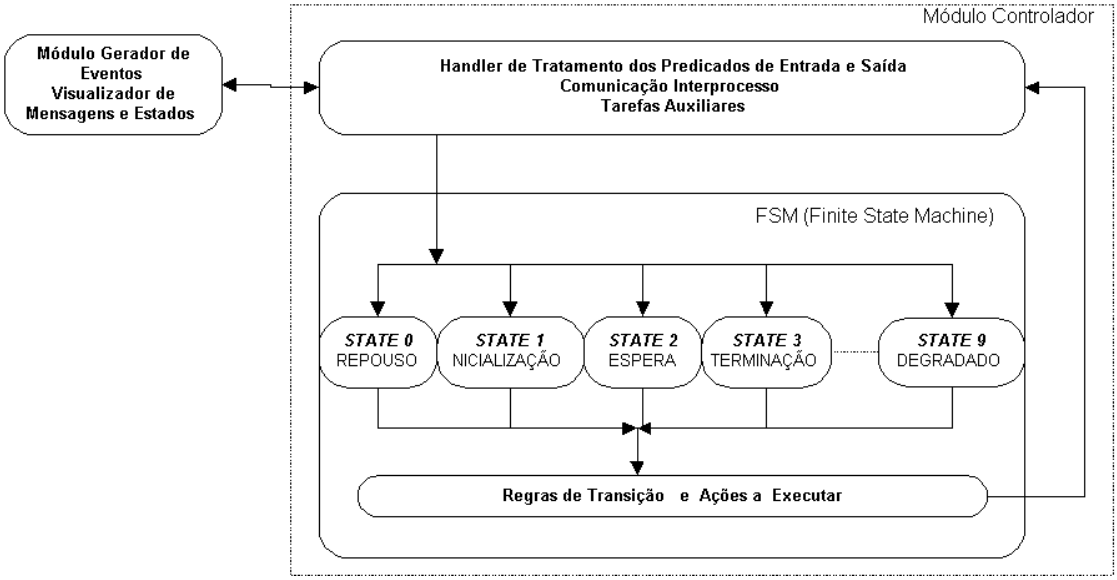


Figura 5-1 Módulo Controlador (Máquina de Estado)

A implementação de uma arquitetura *TMR* distribuída é baseada na replicação da máquina de estado em três processadores distintos cujas saídas serão enviadas a um processo votador fisicamente localizado em outro processador ou não. Detalhes da arquitetura física e lógica do controlador são descritas na Seção 5.2.

No controlador de tráfego ferroviário simulado, cujas funcionalidades são ilustradas na Seção 5.4 a implementação da máquina de estado foi baseada num compilador específico denominado ***SMC*** (*State Machine Compiler*). Esta ferramenta se mostrou bastante útil, possibilitando não só a definição da máquina de estado pretendida, como também sua integração com o código Java do simulador. Um breve detalhamento sobre máquinas de estado e dessa ferramenta pode ser encontrada no ***Anexo A***.

5.2 Arquitetura Física e Lógica

A Figura 5-2 mostra a arquitetura lógica do simulador, onde temos um módulo denominado *Gerador de Eventos Visualizador de Estados e Mensagens* o qual é responsável pela geração de todas as entradas previstas na máquina de estados correspondente ao núcleo do controlador de tráfego ferroviário. São simuladas desta forma tanto as indicações que seriam provenientes do processo, bem como as requisições de comandos originadas de um operador do sistema, o qual poderia estar localizado num centro de controle, como ilustrado na Figura 4-3 da Seção 4.1.2. Adicionalmente este módulo é responsável pela visualização de mensagens relativas aos estados internos de cada controlador (máquina de estado), bem como falhas, alertas ou mesmo mensagens de condições operacionais derivadas do resultado final da votação.

Temos os *módulos controladores* cada qual contendo réplicas da mesma versão do software (máquina de estado) que implementa as funções de controle de tráfego propriamente ditas. Estes controladores estão distribuídos fisicamente em processadores distintos conforme uma arquitetura de redundância modular tripla *TMR*. Dependendo da disponibilidade de processadores no ambiente de rede, podemos ter o processo votador fisicamente separado dos módulos controladores, conforme é mostrado na Figura 5-3. Caso não haja disponibilidade de um processador dedicado pode-se colocar o processo votador compartilhando o processador que hospeda um dos módulos

controladores. Esta possibilidade constitui uma vantagem de se adotar a abordagem de votação por software, conforme discutido na Seção 3.4.2.1.

Uma vez gerado um evento (quaisquer dos previstos no modelo), o mesmo tem de ser replicado para os três módulos controladores, os quais de forma assíncrona (pois são independentes) devem processar as respectivas mensagens. As saídas respectivas também de forma assíncrona devem ser enviadas para o estágio de votação, o qual efetivará uma votação majoritária *2 em 3*. Isto significa que uma saída considerada correta será gerada se pelo menos 2 das saídas dos controladores forem idênticas. Deve ser garantido através de técnicas de sincronismo apropriadas, que a votação seja relacionada com saídas associadas às mesmas informações de entrada. Caso contrário aspectos de segurança seriam comprometidos pela ausência de consistência. Também a disponibilidade do processo poderia ser comprometida, pois haveria divergências freqüentes no processo de votação. Portanto as condições apresentadas na Seção 3.2.1 [SCHNEIDER, 1990] precisam ser garantidas.

Outro aspecto fundamental decorrente da abordagem distribuída da arquitetura *TMR*, é a necessidade de alta confiabilidade na integridade das mensagens trocadas entre os diversos módulos do sistema. O uso de métodos de detecção de erro como códigos de redundância cíclica (*CRC*) deve ser fortemente considerado. Em sistemas de missão e segurança crítica recomenda-se o uso *CRC* de 32 bits de cumprimento, conforme descritos na Seção 3.5. Desta forma uma mensagem gerada pelo módulo *Gerador de Eventos*, conforme Figura 5-2, é composta além de dados relativos a informações do processo, como por exemplo, posição de máquinas de chave, mas também de dados adicionais de controle que a caracterizam unicamente. Estes dados adicionais de controle são comumente designados como "*overhead"* da mensagem.

A verificação de integridade da mensagem (cálculo do *CRC*) é efetivada no lado receptor (módulo controlador). Caso a mensagem recebida esteja integra, será considerada, caso contrario uma mensagem de erro alusiva ao fato, deve ser enviada de volta ao transmissor. Processo análogo é efetivado na troca de mensagens entre os demais módulos do ambiente de simulação tais como: Módulos controladores e votador e entre votador e o módulo gerador de eventos.

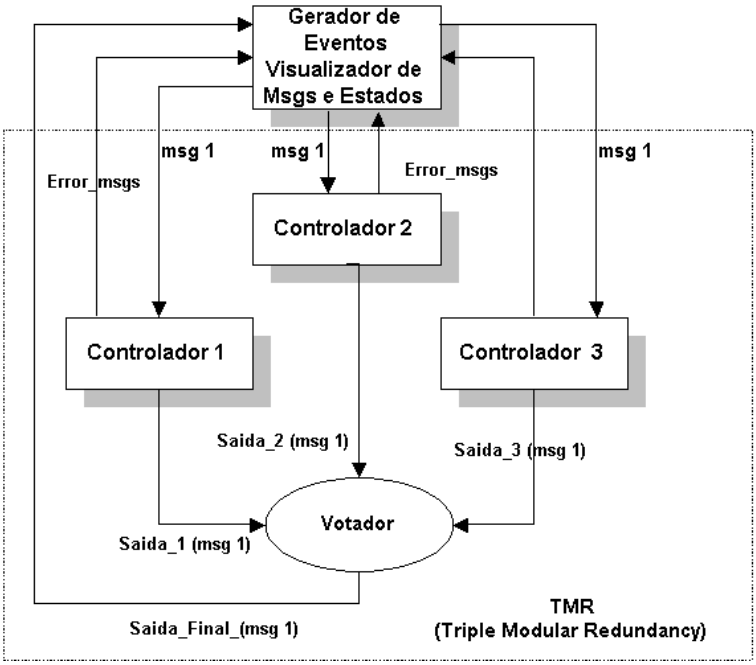


Figura 5-2 Módulo Controlador (Arquitetura Lógica)

A Figura 4-9 mostra como poderiam ser distribuídos fisicamente num ambiente de rede local os processos lógicos apresentados na Figura 4-8.

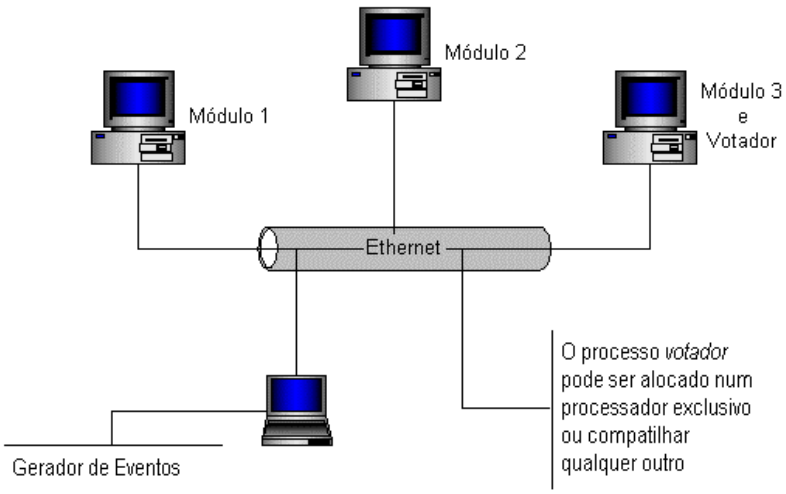


Figura 5-3 Módulo Controlador (Arquitetura Física)

Uma alternativa refere-se ao cenário de termos disponível um processador exclusivo para hospedar o processo de votação por software. Outra seria o uso compartilhado de um processador já destinado a outro propósito.

5.3 Um Modelo Adaptável com Máquinas de Estado

O modelo adaptável ao qual se refere este trabalho significa basicamente a possibilidade de se modificar sem grande esforço, regras funcionais do simulador de intertravamento ferroviário. Vejamos como exemplo básico a seguinte regra na Tabela 5-1:

Estado Atual	Estado de Alinhamento de Rota
Evento	Alinhamento_ok
Saída	Comando de Movimentação da Máquina de Chave e Comando de Travamento da Chave.
Estado Seguinte	Estado de Alinhamento de Rota

Tabela 5-1 Booleanas de Intertravamento

Se quisermos mudar esta regra, de tal forma a incluir mais ações nas saídas, ou mesmo associar esta ação a outro evento ou ainda transitar para outro estado. O uso de máquinas de estado no modelo lógico do sistema permite teoricamente uma fácil adequação, mais ainda, se forem usadas ferramentas apropriadas para implementar máquinas de estados integradas ao restante do software.

A Figura 5-4 abaixo mostra a estrutura em camadas do simulador enfatizando as camadas nas quais alguns requisitos funcionais podem ser adaptados [BRAGA, 2004].

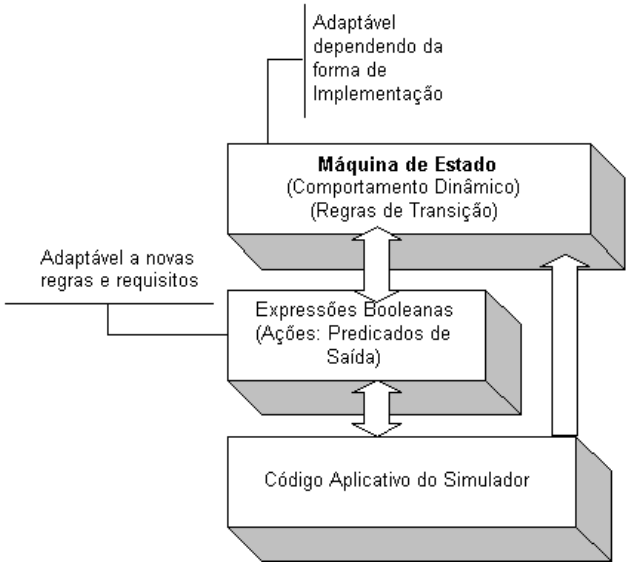


Figura 5-4 Característica de Adaptabilidade

5.4 Funcionalidades

Será detalhado nesta Seção o uso real do simulador de controle de tráfego ferroviário em seu estágio inicial de desenvolvimento em linguagem *Java*.

Usaremos algumas situações práticas para exemplificar o uso deste simulador. A interface principal é apresentada na Figura 5-5, a qual representa essencialmente o módulo *Gerador de Eventos e Visualização*.

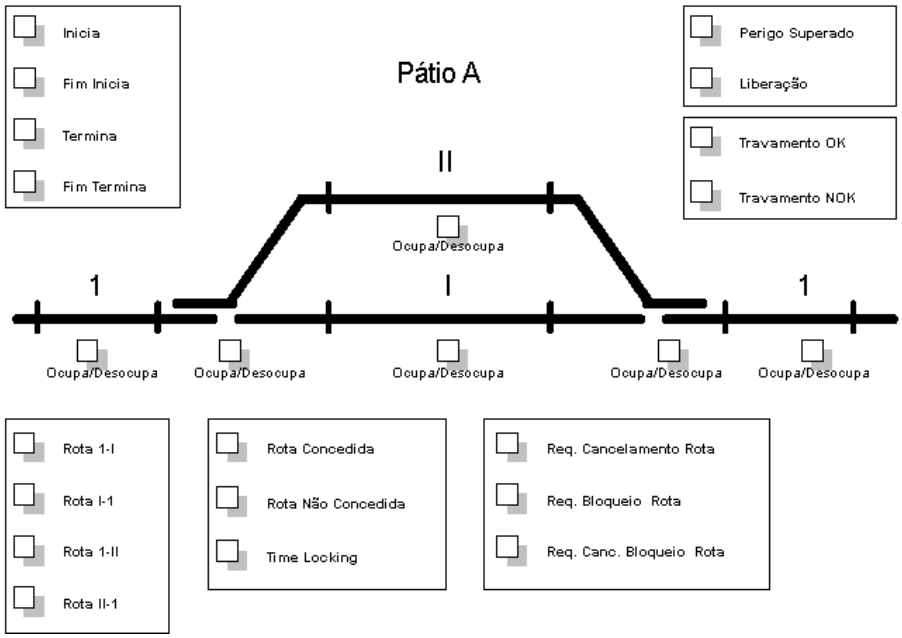


Figura 5-5 Interface Homem-Máquina do módulo Gerador de Eventos

Tem-se o sinóptico de um pátio típico de cruzamento ferroviário com trecho singelo e duas linhas de desvio. Os quadros com caixas de verificação agrupam as entradas previstas no modelo descrito na Seção 4.2.

O 1º grupo de entradas (*Inicia, Fim Inicia, Termina e Fim Termina*) são os comandos de inicialização e terminação do sistema.

O 2º grupo de entradas (*Perigo Superado e Liberação*) são entradas que possibilitam a máquina de estado do controlador transitar para um estado de segurança degradada e então ser liberado para um estado normal de espera de comandos.

O 3º grupo de entradas (*Travamento OK, Travamento NOK*) é designativo do estado de travamento das máquinas de chave associadas às entradas e saídas do pátio. Quando uma rota de entrada ou saída de um pátio é totalmente concedida, um dos

estágios que antecedem o estabelecimento pleno da rota é o de travamento das máquinas de chave que estão no percurso desta rota.

O 4º grupo de entradas (*Rota 1-1, Rota I-1, Rota 1-II, Rota II-1*) corresponde às requisições de rotas ao módulo controlador. Este deve transitar para o estado apropriado e efetivar todas as consistências requeridas para que uma rota seja estabelecida.

O 5º grupo (*Rota Concedida, Rota Não Concedida, Time Locking*) corresponde às indicações resultantes do estado do processamento das rotas em estado de alinhamento. Basicamente se as condições para o estabelecimento de uma rota forem satisfeitas então uma indicação de *Rota Concedida* deverá ser a saída válida do controlador para o módulo *Gerador de Eventos e Visualizador de Estados*. Caso contrário uma saída de *Rota Não Concedida*, deverá ser retornada.

O 6º grupo de entradas (*Req. Cancelamento de Rota, Req. de Bloqueio de Rota, Req. de Canc. de Bloqueio de Rota*) são requisições de restrições a possíveis requisições de rotas ou de cancelamento de rotas já concedidas ou em processo de requisição.

Adicionalmente tem-se saídas que representam o estado de ocupação de todos os circuitos de via. Isto possibilita simular manualmente o sequenciamento de ocupações e desocupações resultantes da movimentação de trens entrando ou saindo do pátio de cruzamento.

Por último, antes de detalharmos alguns cenários com o simulador, vale salientar que a implementação considerou por simplicidade uma área de domínio correspondente a um dos lados de um pátio de cruzamento. Isto corresponde à metade de um pátio de cruzamento ferroviário como mostrado nas Figuras 4-1 e 5-5.

A seguir são ilustrados alguns cenários que mostram o funcionamento do controlador de tráfego ferroviário simulado.

Primeiramente deve-se iniciar o sistema gerando um evento de *inicialização*. No processo de inicialização são necessárias várias ações que garantam a consistência das variáveis de controle. É necessário, portanto considerar um tempo máximo para o ciclo de inicialização finalize. Após este tempo o controlador deve gerar um evento de *fim de inicialização*, o que leva a máquina de estados correspondente a transitar para o *estado de espera*. A Figura 5-6 mostra a seqüência descrita.

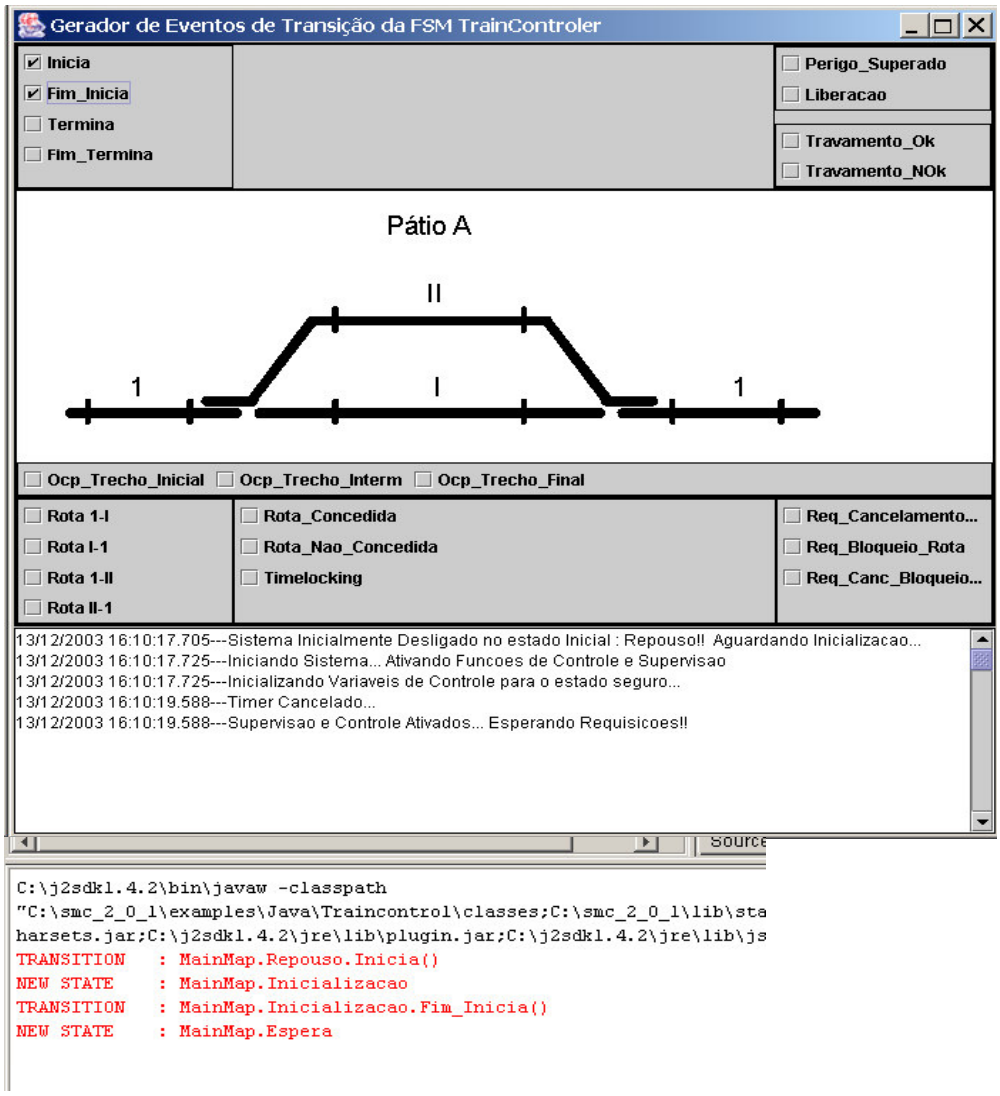


Figura 5-6 Eventos de Inicialização e Fim de Inicialização do Controlador

Podemos verificar que o sistema fica em estado de espera, aguardando requisições válidas de comandos.

Caso o evento *fim de inicialização* não tivesse sido gerado dentro do tempo esperado, o que denotaria algo de errado no processo de inicialização, o controlador assumiria o estado *seguro degradado*, conforme mostra a Figura 5-7.

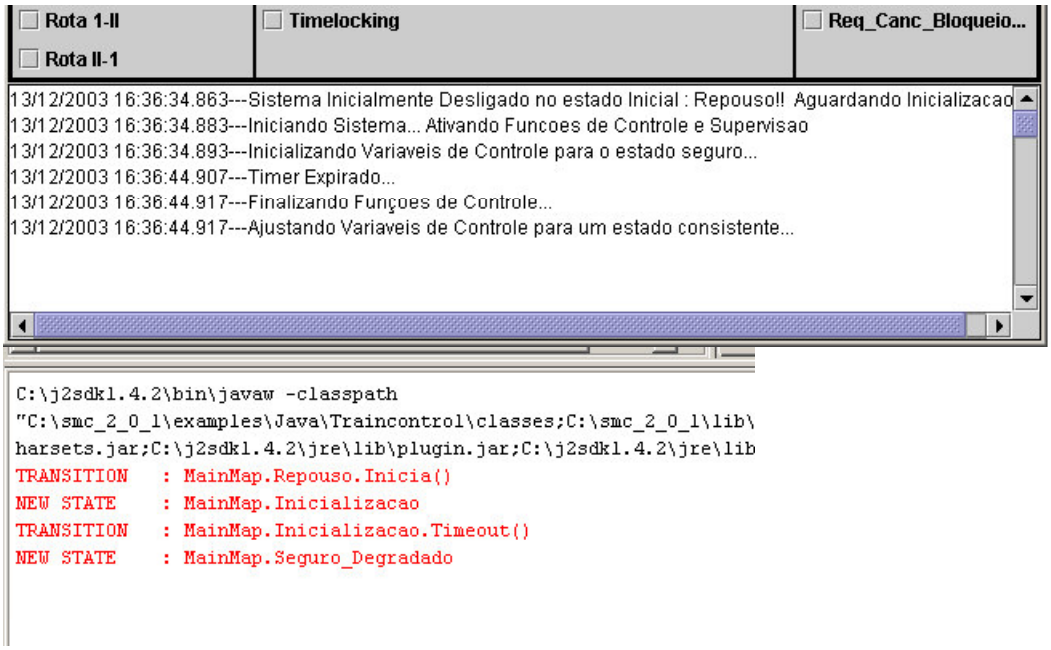


Figura 5-7 Inicialização com Timeout Expirado

Pode-se ver que o sistema transitou para o estado: *seguro degradado*, do qual somente sairá com um evento de *liberação*.

Qualquer que seja a entrada considerada inválida (fora da janela temporal ou mesmo numa seqüência não esperada) a máquina de estado do módulo controlador transitará para um estado apropriado. Serão tomadas as ações apropriadas: Mensagens alusivas e transição ou não para outro estado.

A seguir temos um exemplo de uma requisição de rota (*Rota 1-I*), com as respectivas indicações de travamento de máquina de chave e de rota concedida. Estas geradas pelo módulo controlador.

Após iniciarmos o módulo controlador em condições normais, ficando o mesmo no estado de *espera*. Requisita-se a rota 1-I, conforme Figura 5-8. Assumindo neste caso que a rota é possível e não havendo, portanto nenhum impedimento ou entrada imprópria, o comportamento da lógica funcional do controlador será a seguinte:

- Indicar que a máquina de chave no percurso da rota em questão se encontra travada, isto liberará o bloqueio da rota, com a conseqüente abertura do sinal de velocidade para a mesma.
- Indicar o estado da rota como concedida, significando que o sinal de velocidade foi liberado para a via devendo ser captado pelo subsistema de bordo. Esta

condição autoriza a movimentação de um trem que se encontre na origem desta rota.

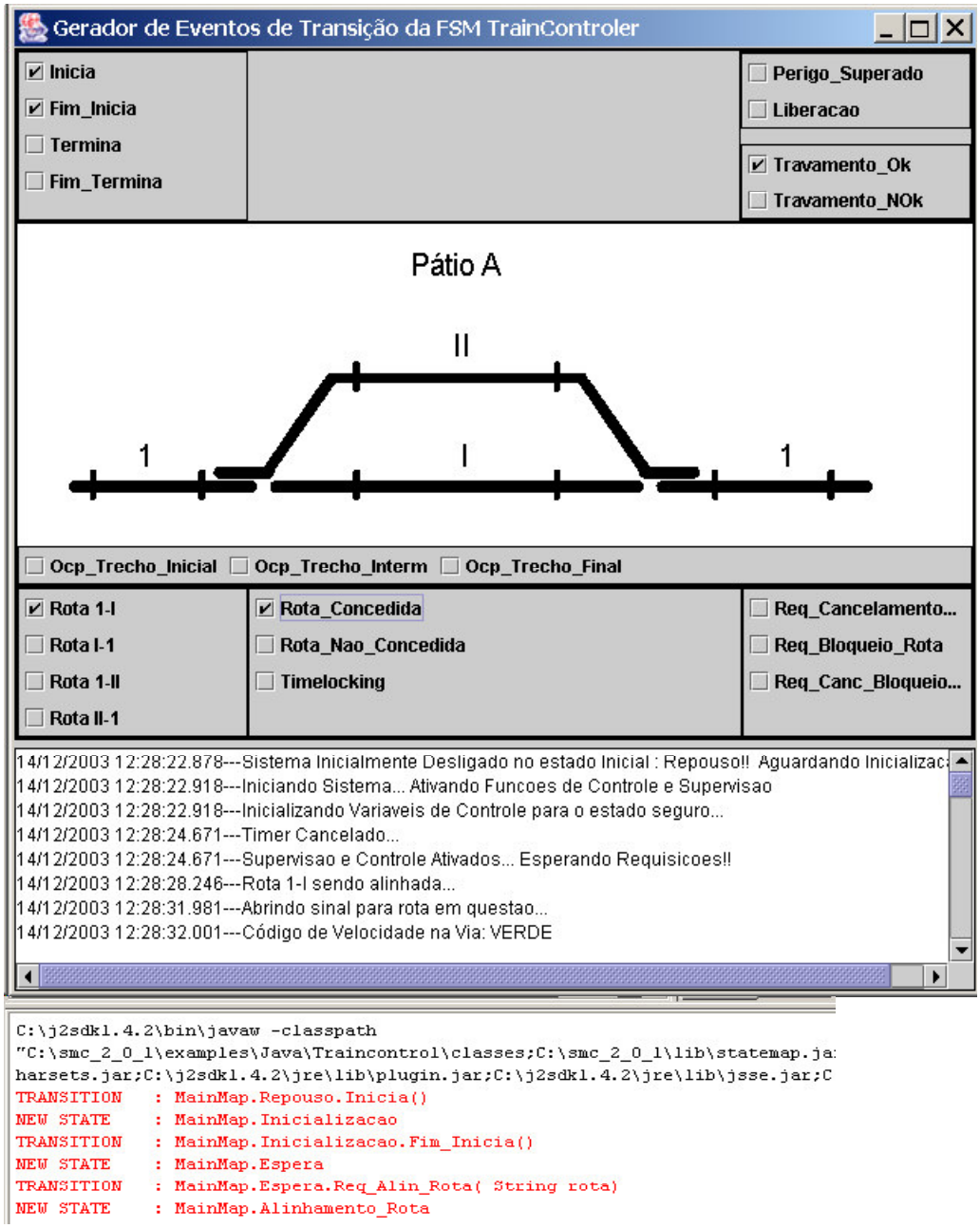


Figura 5-8 Requisição de Alinhamento de Rota

Na Figura 5-9 temos os eventos de ocupação dos trechos inicial, intermediário e final da rota 1-I requisitada no passo anterior.

É relevante notarmos que a lógica funcional de segurança do controlador deve garantir que não sejam liberadas autorizações para outras rotas conflitantes no mesmo domínio. Também a partir do evento de *ocupação do trecho inicial* da rota, o controlador deve transitar para o estado de *cancelamento automático de rota*. Fato este

que denota o cenário de cumprimento real de uma rota por um trem. Após a *ocupação do trecho final* da rota o controlador transita para o *estado de espera*, aguardando novos eventos ou requisições de comandos.

Na Figura 5-10 é mostrado uma situação em que o controlador transita para o estado inseguro. Neste caso específico, aproveitando o exemplo anterior, temos a situação em que a seqüência dos eventos de ocupação dos trechos da rota 1-I, não foi a esperada, isto é, houve uma ocupação do trecho de destino (final), antes da ocupação do trecho intermediário. Por questões de segurança o controlador deve entender que esta situação pode ser potencialmente perigosa, como algum outro trem invadindo sem autorização o trecho de destino desta rota. Neste caso ir para o estado *inseguro*, também deve significar que ações restritivas devem ser tomadas pela lógica funcional do módulo controlador. Ações tais como: Fechamento de todos os bloqueios de entrada e saídas da sua área de domínio, retirada de todos os códigos de velocidade que possam ter sido liberados. Isto significa levar o sistema de controle de tráfego ferroviário para um estado de segurança intrínseco, o qual garanta a parada de todos os trens em sua área de domínio.

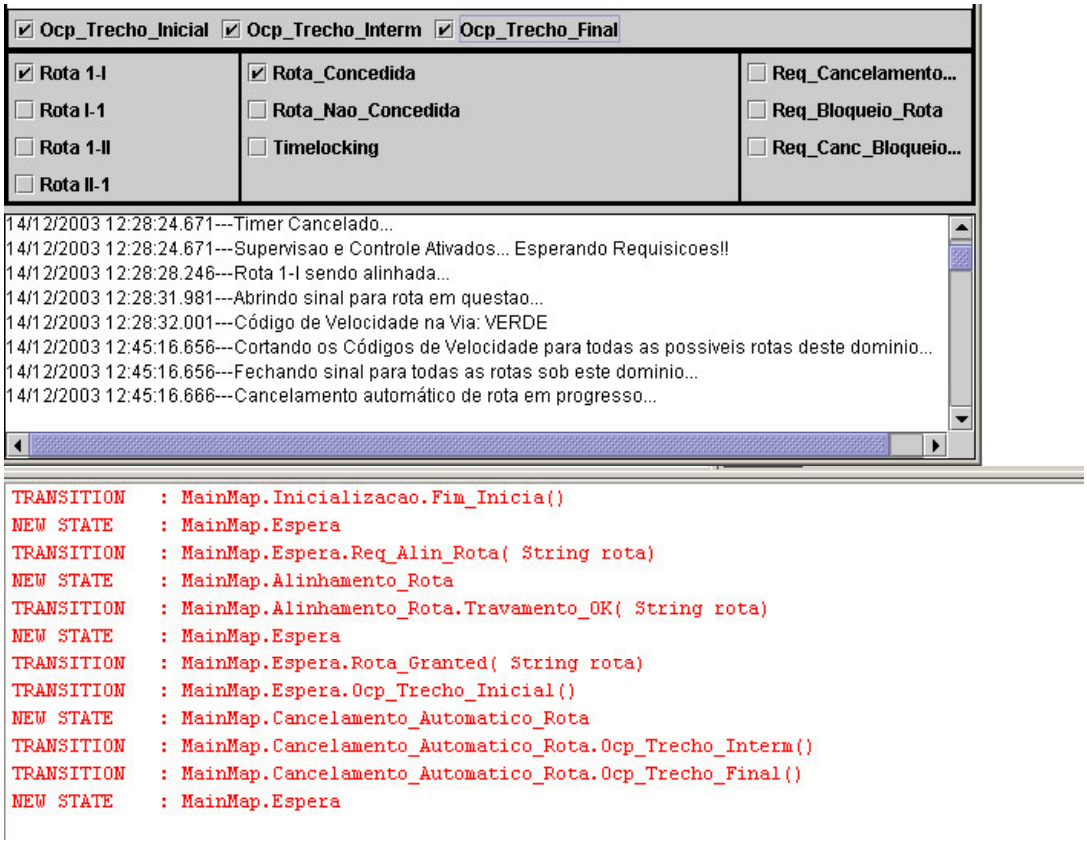


Figura 5-9 Eventos de Ocupação dos Trechos de uma Rota

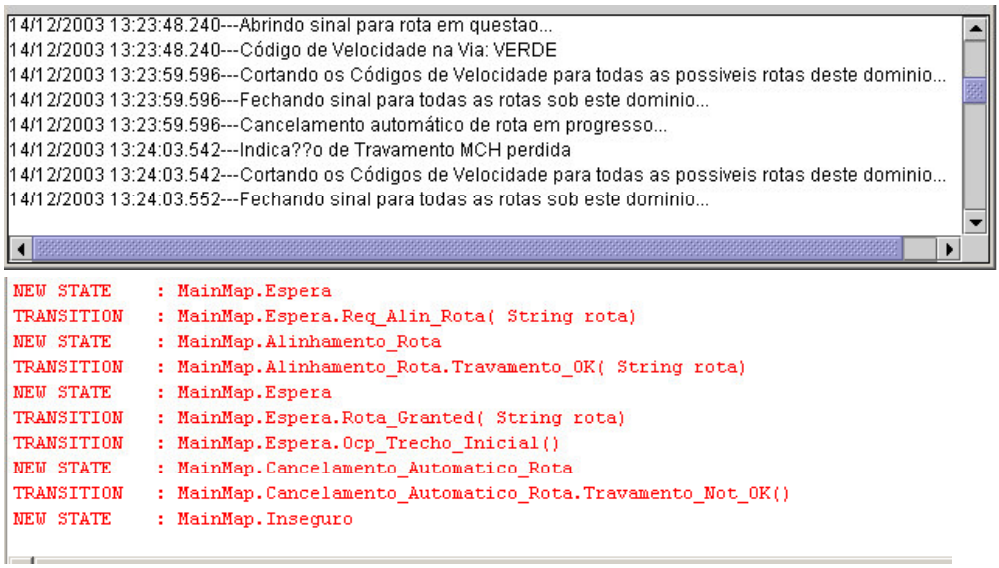


Figura 5-10 Transitando para o Estado Inseguro

Todas as condições implementadas na máquina de estado do controlador são expressas nas tabelas de transições de estados da Seção 4.2.2 (Tabelas 4-4 a 4-8).

5.5 Considerações Finais

Neste Capítulo foi apresentado um estudo de caso (simulador de um intertravamento ferroviário) em que foi detalhada toda uma lógica funcional associada a uma aplicação de missão crítica. Foi mostrada a possibilidade do uso potencial de técnicas de tolerância a falhas em software, em um ambiente que tenta simular as condições lógicas de um intertravamento ferroviário baseado num modelo simplificado.

Um ponto a ser destacado é como o uso de máquinas de estado é apropriado para se dotar o ambiente de simulação adaptável a diferenças em determinados requisitos funcionais, possibilitando desta forma a característica de adaptabilidade desejada ao simulador, mesmo em seu estágio inicial.

Usando-se o compilador de máquinas de estado (SMC), modificações no modelo funcional da máquina de estado mostraram-se relativamente simples, bem como sua efetivação na implementação do simulador em linguagem JAVA. Esta característica contribui significativamente para se alcançar um nível inicial de adaptabilidade pretendido.

Com a implementação do núcleo deste simulador, acredita-se ter contribuído com uma ferramenta numa área da indústria (ferroviária) onde predominantemente existe dependência de hardware e software proprietários.

6. Conclusões

A concepção do simulador de um controlador de tráfego ferroviário (intertravamento) adaptável em termos de alguns requisitos funcionais foi possível com o uso de máquinas de estado em seu modelo conceitual. Ferramentas práticas para a implementação integrada de máquinas de estado com aplicações em linguagens populares mostraram-se um recurso interessante na implementação mesmo que preliminar do protótipo deste simulador.

Um dos principais objetivos do trabalho acredita-se alcançado foi a seleção e discussão detalhada das principais técnicas de tolerância a falhas em software e hardware, as quais podem ser aplicáveis na implementação de um controlador de tráfego ferroviário simulado, cujo modelo simplificado foi discutido no Capítulo 4. Desta forma a implementação de um protótipo de controlador de tráfego ferroviário, constituiu uma contribuição que pode ser útil na análise de sistemas de controle de tráfego ferroviário reais, onde tanto os aspectos de engenharia de sistemas tolerante a falhas, bem como os de segurança crítica são altamente relevantes.

Temos visto vários simuladores de operação de trens ou mesmo de despacho e distribuição de tráfego. Este trabalho se refere à simulação de intertravamentos ferroviários, isto é, a parte do controle de tráfego ferroviário que se encarrega do licenciamento de trens sob regras e restrições bem definidas. Uma solução mesmo que introdutória sobre uma ferramenta adaptável em relação a alguns requisitos funcionais de um intertravamento ferroviário constitui uma contribuição interessante neste segmento da indústria. A possibilidade de se complementar ou modificar regras funcionais num ambiente de simulação possibilita a realização de ensaios e análises que podem cobrir as especificidades de um intertravamento ferroviário específico.

A área metro-ferroviária (campo de aplicação alvo desta implementação) é um segmento de aplicações de missão crítica relacionadas a controle de processo. Desta forma a evolução desta implementação para suportar características de tolerância a falhas deve ser considerada. Em geral as implementações reais são tolerantes a falhas em hardware e software. Existe uma grande variedade de requisitos relacionados à

intertravamentos de controle ferroviários, logo uma ferramenta de simulação adaptável neste sentido é de grande valia neste segmento..

É consenso entre vários pesquisadores em "*Safety Systems*" que a garantia de segurança deve ser parte de uma metodologia de projeto, devendo ser verificada desde suas fases iniciais e continuamente reavaliada em caso de mudanças de requisitos. A garantia de segurança não deve ser limitada, portanto, apenas aos aspectos de arquitetura física ou lógica ou mesmo de implementação do software.

Aspectos relevantes relativos a software de segurança são importantes, devendo ser considerados como premissas no desenvolvimento de aplicações reais. Seguem algumas conclusões relativas a software de segurança (*Safety Software*), relevantes no contexto deste trabalho e a qualquer projeto que envolva o controle de intertravamentos ferroviários:

- Não existem soluções simples para se conseguir níveis de segurança aceitáveis em software. Software Seguro requer tempo, esforço, conhecimento e experiência.
- Simplicidade é uma técnica efetiva no projeto de software de segurança crítica. Deve-se projetar software cuja complexidade seja gerenciável.
- Um dos mais notórios fatores de risco em sistemas de segurança crítica é a complacência.
- Segurança e Confiabilidade são conceitos distintos, não se deve confundi-los.
- Segurança é um problema para sistemas computacionais, cuja solução está associada a questões multidisciplinares, isto é, trabalho conjunto de equipes de especialistas em diversas áreas, em todas as fases do ciclo de vida.
- Realizar considerações de segurança em fase de projeto é mais efetivo que inserir dispositivos de proteção inseridos após o sistema desenvolvido.
- O uso de técnicas de tolerância a falhas no projeto do software é recomendado como uma forma de atender os requisitos de dependabilidade.

Tem se conseguido progressos significativos no desenvolvimento de ferramentas e metodologias que auxiliam na garantia de qualidade de software, onde são focados os aspectos de segurança, embora ainda seja inviável garantir formalmente a segurança de sistemas complexos baseadas em software. Um exemplo bastante

enfatizado neste trabalho é a ferramenta *CAD* denominada *SpecTRM-RL*, que essencialmente implementa um metodologia voltada para modelagem e especificação de requisitos de sistemas de segurança. Através desta pode-se ter uma visão real e abrangente de quão necessário e crítico é o gerenciamento de riscos em projetos *"Safety Critical",* e a seu mapeamento nas fases iniciais do projeto.

Enfim, acredita-se ter alcançado plenamente os objetivos da pesquisa, em função dos resultados obtidos:

- Implementação do núcleo do simulador usando a abordagem de máquinas de estado, sendo que a característica de adaptabilidade foi facilitada com o uso de uma ferramenta específica *(SMC)* de criação e integração de *FSM*.
- Identificação e análise de uma metodologia para o desenvolvimento de sistemas de missão e segurança críticas com requisitos de segurança fortemente baseados em software.
- Análise e seleção de técnicas de tolerância a falhas por software potencialmente aplicáveis no desenvolvimento de sistemas de controle de intertravamentos ferroviários.

6.1 Trabalhos futuros

Como mencionado anteriormente o estudo de caso apresentado neste trabalho, constitui apenas um protótipo em termos de implementação, podendo o mesmo evoluir em diversos aspectos, como:

- Implementação completa do modelo físico e lógico apresentados, com a possível implementação e verificação de efetividade de diferentes técnicas de votação por software.
- Considerar a evolução do modelo para arquiteturas de controle de tráfego ferroviário mais avançado [SHEIKH, 1990]. Suporte aos conceitos de controle de tráfego baseados em comunicação (*CBTC – Communication Based Train Control*) [MOORE & POLIVKA, 2003], seria apreciável.
- São vários os desafios ainda não resolvidos, nas áreas de software de segurança (Safety Software) e tolerância a falhas apresentados nesta pesquisa que podem servir de base para trabalhos futuros de forma isolada ou mesmo num contexto

evolutivo do controlador de tráfego ferroviário simulado apresentado. Os desafios mencionados na Seção 3.6 são relevantes neste contexto.

- A característica de adaptabilidade do simulador de controle de tráfego ferroviário apresentada neste trabalho poderia ser significativamente melhorada, se for implementado um editor de expressões booleanas que representassem os predicados de entrada e saída do modelo. Este módulo poderia também gerar o código Java necessário à implementação dos métodos relativos às ações dos predicados de saída invocados pelo módulo controlador através da máquina de estado. Uma boa referência para a implementação deste módulo seria o compilador de máquinas de estado (*SMC*), o qual possui código fonte livre e apresenta característica análogas, mesmo que para outro propósito.

O uso combinado de um controlador de tráfego ferroviário simulado, com o ambiente *SpecTRM-RL*, podem constituir elementos poderosos e úteis na análise de soluções de engenharia de sistemas aplicados a área de controle de tráfego ferroviário. Campo este em que o uso de sistemas computacionais de controle tem encontrado utilização crescente. Desta forma enquanto a análise de requisitos de segurança e sua modelagem com verificações de completeza podem ser efetivamente executadas no ambiente *SpecTRM*, o simulador de intertravamento ferroviário poderia implementar o modelo numa perspectiva mais próxima das regras funcionais do negócio em questão: Controle de Tráfego Ferroviário, o qual constitui um cenário válido para se testar técnicas de projeto de sistemas vitais predominantes baseados em software.

REFERÊNCIAS BIBLIOGRÁFICAS

[ABBOTT, 1990] Abbott, R. J., Resourceful Systems for Fault Tolerance, Reliability, and Safety, ACM Computing Surveys, Vol. 22, No. 1, March 1990, pp.35-68.

[ANDERSON & LEE, 1981] Anderson, T.; Lee, P. A. Fault Tolerance -Principles and Practice. Englewood Cliffs, Prentice-Hall, 1981.

[ANDERSON & LEE, 1990] Lee, P. A., Anderson, T., Fault-Tolerance: Principles and Practice- Second Edition. Spring Verlag, 1990.

[AVIZIENIS, 1976] Avizienis, A., 'Fault-tolerant systems.' IEEE Transaction on Computers, volume C-25, num. 12, pp. 1491–1501, December1976.

[AVIZIENIS, 1985] Avizienis, A., 'The n-version approach to fault tolerant software.' *IEEE Transaction on Software Engineering*, volume 11, num. 12, pp. 1491–1501, December 1985.

[AVIZIENIS, 1995] Avizienis, A. The Methodology of N-Version Programming, in R. Lyu, editor, Software Fault Tolerance, John Wiley & Sons, 1995.

[AVRESKY, 1992] Avresky, D. et al, Fault Injection for the Formal Testing of Fault Tolerance, Digest of Papers of the Twenty-Second International Symposium on Fault-Tolerant Computing, Boston, Massachusetts, July 8 – 10, 1992, pp. 345 - 354.

[BASS, 1995] Bass, J.M. Voting in Real-Time Distributed Computer Control Systems, PhD Thesis, University of Sheffield, October 1995.

[BRAGA, 1997] Braga, R. I., Cadeias de Markov Aplicadas a Teoria da Confiabilidade de Sistemas Computacionais. Monografia, Universidade Federal do Maranhão - São Luís, Setembro 1997. 65p.

[BRAGA, 2003] Braga, R. I. 'Dispatch and Management System – Carajás Railroad of Cia Vale do Rio Doce' Proceedings of International Heavy Haul Association 2003 Specialist Technical Session , pp. 4.29 – 4.36, Dallas Texas, USA, May 5-9 2003.

[BRAGA, 2004] Braga, R. I. An Adaptive Simulated Train Traffic

Control with Finite State Machines. To be Published in Proceedings of "Integrated Design & Process Technology Symposium on System Design and Software Engineering", Kusadasi, Izmir, Turkey, Jun 28 – Jul 2, 2004.

[CAMARGO JR., 1996] Camargo Jr., J.B.. Estudo da Segurança em Sistemas de Controle Metro-Ferrroviários. Tese de Doutorado, Escola Politécnica da Universidade de São Paulo, 1996.

[CAMARGO JR., 1996b] Camargo Jr., J.B. Confiabilidade, Disponibilidade e Segurança de Sistemas Computacionais. São Paulo: FDTE EPUSP, 1996. 184p.

[CHEN & AVIZIENIS, 1978] Chen, L.; Avizienis, A. N-version programming: A fault tolerance approach to reliability of software operation. In: Annual International 23 Symposium on Fault-Tolerant Computing, 8, 1978. Proceedings. New York, IEEE, 1978. p. 3-9.

[CRISTIAN, 1991] Cristian, F., 'Understanding fault-tolerant distributed systems.' Communications of the ACM, volume 34, num. 2, pp. 56–78, feb. 1991.

[CROLL, 1995] Croll, P. R. et al, Dependable, Intelligent Voting for Real-Time Control Software, Engineering Applications of Artificial Intelligence, vol. 8, no. 6, 1995.

[FDTE 542.401. AS. 001] Determinação, Análise e Classificação das Funções de Segurança do Sistema Paraíso-Clínicas, Março 1991.

[GRAY, 1986] GRAY, Jim. Why Do Computers Stop and What Can Be Done About It?, Proceedings of the Fifth Symposium On Reliability in Distributed Software and Database Systems, January 13-15, 1986, pp. 3-12.

[IYER, 1996] Iyer R. K. and Tang, D. Experimental Analysis of Computer System Dependability, in Fault Tolerant Computer System Design, Dhiraj K. Pradhan,

[JALOTE, 1994] Jalote, P., Fault Tolerance in Distributed Systems. Prentice Hall, 1994.

[JANS, 1997] Jansch-Porto, I. E. S; Weber, T. S. Recuperação em Sistemas Distribuídos. In: XVI Jornada de Atualização em Informática, XVII Congresso da SBC, Brasília, 2-8 agosto de 1997. anais. pp 263-310

[KNIGHT & LEVESON, 1986] Knight, J. C., Leveson, N. G., 'An experimental evaluation of the assumption of independence in multi-version programming.' IEEE Transactions on Software Engineering, volume SE-12, num. 1, pp. 96–109, Janeiro 1986.

[KNIGHT & LEVESON, 1989] Knight, J.C., Leveson, N.G., Ammann, P., Testing Software Using Multiple Versions. Technical Report 89029N, Software Production Consortium, jun. 1989.

[KOOPMAN, 1997] Koopman, P. et al, Comparing Operating Systems Using Robustness Benchmarks, Proceedings of the 1997 16th IEEE Symposium on Reliable Distributed Systems, October 1997, pp. 72 – 79.

[LAI, 1995] Lai, M.Y. and Wang, S. Y. Software Fault Insertion Testing for Fault Tolerance, in Software Fault Tolerance, Michael R. Lyu, editor, John Wiley & Sons, 1995, pp. 315 – 333.

[LALA, 1994] Lala, J. H. and Harper, R. E. "Architectural Principles for Safety-critical Real-time Applications", Proc. of the IEEE, Vol. 82, No. 1, pp. 25-40, Jan. 1994.

[LAPRIE, 1989] Laprie, J.C., 'Dependability: A Unifying Concept for Computing and Fault Tolerance.' T. Anderson, editor, *Distributed Systems*, BSP Professional Books, 1989.

[LEE, 1995] Lee I. and Iyer, R. K. Software Dependability in the Tandem GUARDIAN System, IEEE Transactions on Software Engineering, Vol. 21, No. 5, May 1995, pp. 455 – 467.

[LEVESON, 1986] Leveson, N. G. Software Safety: Why, What, and How, ACM Computing Surveys, Vol. 18, No. 2, June 1986, pp. 125 – 163.

[LEVESON, 1995] Leveson, N.G. Safeware: System Safety and Computers, Addison-Wesley, 1995.

[LEVESON, 1997] Leveson, N.G.; Reese, J.D. Software Deviation

Analysis: A Safeware Technique. In AIChe 31st Annual Loss Prevention Symposium, March 1997.

[LEVESON, 1998] Leveson, N.G, Brown, M. Modeling Controller Tasks for Safety Analysis. HESSD 1998 2nd Workshop on Human Error, Safety, and System Development April 1-2, 1998, Seattle, Washington, USA

[LEVESON, 1998b] Leveson, N.G.; Reese, J.D., Heimdahl, M. "SpecTRM: A CAD System for Digital Automation" in Digital Avionics System Conference, November 1998.

[LEVESON, 1999] Leveson, N.G., Heimdahl, M. P. E., Reese, J. D. Designing Specification Languages for Process Control Systems: Lessons Learned and Steps to the Future. ESEC / SIGSOFT FSE 1999: 127-145

[LEVESON, 2000] Leveson, N.G. Intent Specifications: An Approach to Building Human-Centered Specifications. IEEE Trans. Software Eng. 26(1): 15-35 (2000)

[LEWIS, 1987] Lewis, E.E., Introduction to Reliability Engineering. New York: John Wiley & Sons, 1987. 400p.

[LORCZAK, 1989] Lorczack, P. R. et al, A Theoretical Investigation of Generalized Voters for Redundant Systems, Digest of Papers FTCS-19: The Nineteenth International Symposium on Fault-Tolerant Computing, 1989, pp. 444 - 451.

[LYU & HE, 1993] Lyu, M., He, Y., 'Improving the n-version programming process through the evolution of a design paradigm.' IEEE Transactions on Reliability, volume 42,num. 2, pp. 179–189, jun. 1993.

[LYU, 1995] Lyu, M., Software Fault Tolerance. Wiley Press, 1995.

[MARTIN, 1998] Martin, R.C., UML Tutorial: Finite State Machines, C++ Report, SIGS Publications Inc., ISSN 1040-6042, June 1998.

[MITRA, 2000] Mitra, S., and McCluskey, E.J. "Word-Voter: A New Voter Design for Triple Modular Redundant Systems," 18th IEEE VLSI Test Symposium, pp. 465-470, Montreal, Canada, Apr. 30-May 4, 2000.

[MOORE & POLIVKA, 2003] Moore, W.J., Polivka, A. 'Positive Train Control for Heavy Haul Operations' Proceedings of International Heavy Haul Association 2003 Specialist Technical Session , pp. 3.75 – 4.36, Dallas Texas, USA, May 5-9 2003.

[PETERSON, 1972] Peterson, W. & E. Weldon, Error-Correcting Codes, MIT Press, Second Edition, 1972.

[POMALES, 2000] Pomales, W. T., "Software Fault Tolerance: A Tutorial". Langley Research Center, Hampton Virginia. NASA/TM-2000-210616.

[PRADHAN, 1996] Pradhan D. K., Fault-Tolerant Computer System Design, Prentice-Hall, Inc.,1996.

[RANDELL 1975] Randell, B. System Structure for Software Fault Tolerance, IEEE Transactions on Software Engineering, Vol. SE-1, No. 2, June 1975, pp. 220 – 232.

[RAO, 1987] Rao, V. P.; Venkatachalam, P. A. Microprocessor Based Railway Interlocking Control With Low Accident Probability. IEEE Transaction on Vehicular Technology, v.35,n.3, p141-7, August 1987.

[SAFEWARE, 2001] Safeware Engineering Corporation, Software Safety, 2001.

[SAFEWARE, 2003] Safeware Engineering Corporation, SpecTRM-RL User´s Manual.

[SAMPAIO, 2000] Sampaio, L.M.R., Serviços de Processamento Tolerantes a Faltas para Sistemas Distribuídos Assíncronos. Dissertação de Mestrado, Universidade Federal da Paraíba, Centro de Ciências e Tecnologia, Coordenação de Pós-Graduação em Informática, Campina Grande - Pb, Outubro de 2000.

[SCHNEIDER, 1990] Schneider, F. B., 'Implementing fault tolerant services using the state machine approach: a tutorial.' ACM Computing Surveys, volume 22, num. 4, pp. 299–319, dezembro 1990.

[SCOTT, 1987] Scott, R. K., Gault, J. W. and McAllister, D. F. Fault-Tolerant Software Reliability Modeling, IEEE Transactions on Software Engineering, Vol. SE-13, No. 5, May 1987, pp. 582 – 592.

[SHAW, 1992] Communicating real time state machines. IEEE Transactions on Software Engineering, v.18, n.9, p.805-16, Sept. 1992.

[SHEIKH, 1990] Sheikh, A. U. H., Coll, D.C., Ayers, R. G., Bailey, J.H. ATCS: Advanced Train Control System Radio Data Link Design Consideration. IEEE Transactions on Vehicular Technology, Vol.39 n.3, pag.256-62, Aug.1990.

[SHIN, 1984] Shin, K. G., Lee, Y. H. Error detection process: model, design and its impact on computer performance. IEEE Transactions on Computer, v.33, n.6, p.130-47, June 1984.

[SHIN, 1989] Shin, K. G., Dolter, J. W. Alternative majority voting methods for real time computing systems. IEEE Transactions on Reliability, v.38, n.1, p.58-64, Apr 1989.

[VASCONCELOS, 1997] Vasconcelos, S.R.A., Provendo Serviços para Tolerância a Faltas em Sistemas Distribuídos. Dissertação de mestrado, COPIN - Universidade Federal da Paraíba, Campina Grande, julho 1997.

[VERNAZZA, 1990] Vernazza, G., ZUNINO, R. A distributed Intelligence methodology for railway traffic control. IEEE Transactions on Vehicular Technology, Vol.39 n.3, pag.263-70, Aug.1990.

[WILLIAMS, 1993] Williams, R. N. A Painless guide to CRC error detection algorithms. Version 3, 1993

[WILLIAMS, 1994] Williams, L.G. Assessment of safety critical specification. IEEE Software, V.11, n.1, p.51-60, Jan. 1994.

[ZAHEDI, 1991] Zahedi, F., Asharafi, N. Software reliability allocation based on structure, utility, price and cost. IEEE Transactions on Software Engineering , v.17, n.4, p.345-56, Apr 1991.

ANEXO A

Máquinas de Estado Finitas (*FSM Finite State Machine*)

FSM consiste em uma ferramenta poderosa ao suporte da modelagem dinâmica de aplicações gerais, em específico sistema de controle de processo de tempo real. Esta abordagem provê uma forma simples e discernível de descrever e implementar lógica de controle para muitas aplicações complexas do mundo real. A notação usada em *FSM* permite que lógicas de controle complexas sejam representadas em diagramas relativamente pequenos, no entanto, simples de serem entendidos, verificados, devido às regras simples que estabelecem seus fundamentos.

Este anexo descreve noções básicas sobre *FSM*, em específico, as que foram usadas neste trabalho. Adicionalmente são descritos aspectos básicos da ferramenta *SMC* (*State Machine Compiler*), apresentando as principais características e funcionalidades.

Informações mais detalhadas sobre o compilador de máquinas de estado *SMC* podem ser encontradas no seu manual disponível no site http://www.oma.com .

Definições Básicas

- ***Estado*** é uma condição que um sistema assume num dado momento.
- ***Evento*** é uma ocorrência significante ou notável, a partir do qual transições ocorrerão.
- ***Transição*** é o relacionamento entre dois estados indicando que um evento ocorreu. Na presença de uma transição o sistema passará de um estado prévio para outro subseqüente. É válido que o estado subseqüente seja o mesmo estado anterior a ocorrência de uma transição.

Definição Formal de Máquinas de Estado Finitas

Para se definir formalmente máquinas de estado finitas, é necessário especificar algumas questões, as quais são essenciais para esta definição.

- Quantos e quais são os estados de uma determinada FSM ?

- Qual o conjunto de eventos de entrada da FSM ?
- Como são definidas as transições de um estado para outro na FSM ?
- Qual o estado inicial da FSM ?
- Qual o estado final da FSM ?

Estas cinco questões nos conduzem diretamente a definição formal de uma FSM. Conforme segue:

Uma Máquina de Estado Finita M é uma 5 - tupla $M = (Q, \Sigma, \delta, q_o, F)$, onde
Q é um conjunto finito de estados
Σ é o alfabeto descritivo das entradas da FSM
$\delta : Q \text{ x } \Sigma \rightarrow Q$ é a função de transição dos estados
$q_o \in Q$ é o estado inicial
$F \subset Q$ é o conjunto de estados finais

Para exemplificar a definição anterior, tomemos como exemplo simples a FSM de uma chave "pushbutton", a qual liga e desliga a cada evento de acionamento da chave ("Push").

Seja a FSM PB,
$PB = \{\{On, Off\}, \{PUSH\}, \{(On, PUSH) \rightarrow Off, (Off, PUSH) \rightarrow On\}, Off, \{On\}\}$

Em máquinas de estado complexas, as funções de transferências (transições dos estados) usualmente são expressas na forma tabular. Segue mais um exemplo de uma FSM, no intuito de exemplificar esta abordagem.

Segue a definição formal de uma FSM relativa a uma "trava de porta", cujo segredo de abertura são os dígitos na seguinte seqüência : "3141". Logo podemos de forma simples representar os seguintes estados:

q_o : Estado "idle"
q_1 : Estado "3"
q_2 : Estado "31"
q_3 : Estado "314"
q_4 : Estado "3141"
q_5 : Estado "Erro"

Logo a definição formal da FSM *DL*, seria

$$DL = \{\{q_o, q_1, q_2, q_3, q_4, q_5\}, \{0,1,2,\ldots 9\}, \delta, q_o, \{q_4\}\}$$

Onde,

$\delta =$

	0	1	2	3	4	5	6	7	8	9
q_o	q_5	q_5	q_5	q_1	q_5	q_5	q_5	q_5	q_5	q_5
q_1	q_5	q_2	q_5	q_5	q_5	q_5	q_5	q_5	q_5	q_5
q_2	q_5	q_5	q_5	q_5	q_3	q_5	q_5	q_5	q_5	q_5
q_3	q_5	q_4	q_5	q_5	q_5	q_5	q_5	q_5	q_5	q_5
q_4	q_5	q_5	q_5	q_5	q_5	q_5	q_5	q_5	q_5	q_5
q_5	q_5	q_5	q_5	q_5	q_5	q_5	q_5	q_5	q_5	q_5

Existe numerosa literatura sobre formalização matemática de máquinas de estado finita, no entanto, para o propósito deste trabalho a definição formal apresentada é suficiente.

Implementando uma *FSM:* Compilador de Máquinas de Estado (*SMC*)

O compilador SMC originalmente desenvolvido por *Robert C. Martin* e melhorado por *Charles W. Rapp*, permite definir e implementar *FSM* em linguagens populares como (C++, Java e TCL).

Basicamente o processo de implementação da *FSM* com o compilador *SMC* é composto dos passos seguintes:

- Escrever o *script SMC* com as características da *FSM* desejadas, isto é, seguindo as regras de sintaxe do compilador *SMC*. Explicitar todos os estados, transições e requisições de ações previstas no modelo da *FSM*. O arquivo contendo o script deve ser do tipo <FSM_name>.sm
- Compilar o script *SMC* para uma das linguagens suportadas (Java, C++, TCL), como resultado teremos o código de uma classe na linguagem alvo.

- Integrar a classe que implementa a *FSM*, obtida da compilação do script *SMC*, com a classe de aplicação. É importante salientar que todas as ações previstas na *FSM* devem ser implementadas como métodos da classe de aplicação, os quais podem ser chamados a partir da *FSM*.

A Figura A-1 abaixo ilustra o processo descrito nos passos acima.

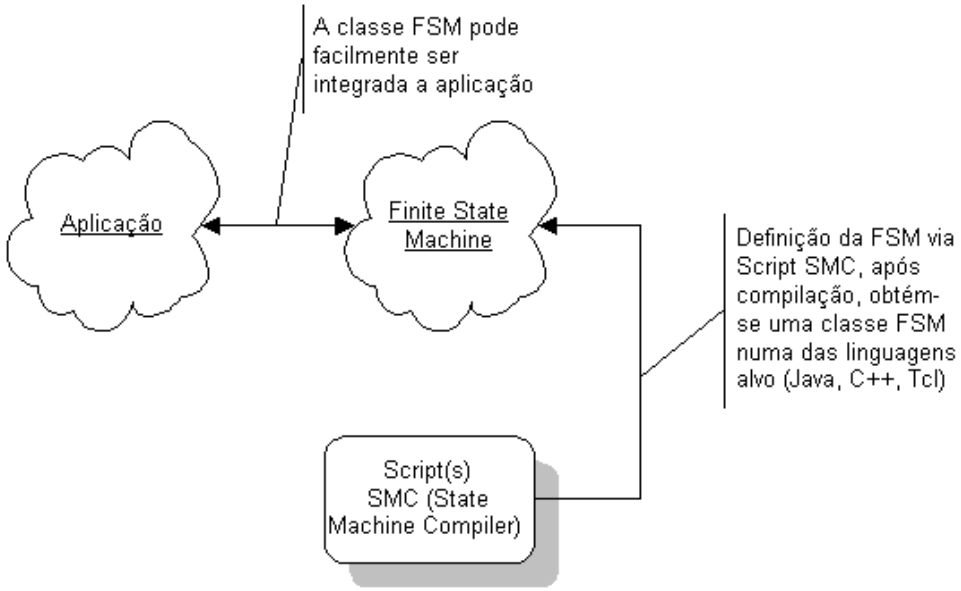
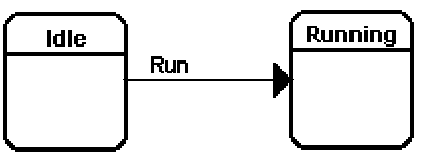


Figura A-1 Implementando FSM com State Machine Compiler (SMC)

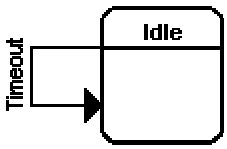
A seguir são mostrados alguns dos elementos fundamentais da sintaxe do SMC. Para ilustrar como traduzir do modelo *FSM* para a sintaxe específica do compilador de máquinas de estado, usaremos a notação *UML* (*Unified Modeling Language*). Esta linguagem possui um conjunto bastante rico de notação relacionada com diagramas de estado para suporte a modelagem dinâmica. Evidentemente poderia se usar qualquer outra notação diferente da *UML* como *Harel State Machine*.

- Transição Simples sem Ações




```
// State
Idle
{
        // Trans    Next State      Actions
        Run         Running         {}
}
```

- Transição Loopback (Permanecendo no mesmo estado após uma transição)




```
// State
Idle
{
        // Trans    Next State      Actions
        Timeout     nil             {}
}
```

ou

```
// State
Idle
{
        // Trans    Next State      Actions
        Timeout     Idle            {}
}
```

- Transição com ações

```
// State
Idle
{
    // Trans
    Run
            // Next State
            Running
                    // Actions
                    {
                        StopTimer("Idle");
                        DoWork();
                    }
}
```

Evidentemente há uma variedade maior de casos suportados pelo compilador SMC, que permitem uma boa flexibilidade na definição de *FSM*, tais como: *Transições condicionais, Transições com passagens de argumentos, ações pré-definidas em transições, transições default e transições PUSH e POP* que possibilitam melhor estruturar a complexidade de *FSM* extensas. As transições *push e pop* suportadas pelo *SMC* são análogas em sentido à noção de super-estados da UML, no entanto a sua implementação no *SMC* é mais próximo do conceito de chamada de sub-rotinas.

Uma descrição detalhada de todos os aspectos relacionados com a sintaxe do compilador *SMC*, pode ser encontrado no seu *User´s Guide*.

Integrando uma *FSM* a uma classe Java

Para integrar uma *FSM* gerada pelo compilador *SMC* em uma classe de aplicação Java, partindo das seguintes premissas :

- Uma classe Java (aplicação) <*My_Application.java*>

- Script *SMC* da *FSM* <*My_application.sm*>
- Compilação bem sucedida do script <*My_Application.sm*>

Devem-se seguir os seguintes passos:

1. Adicionar na classe de aplicação Java <*My_Application.java*> a seguinte declaração: `private My_ApplicationContext _fsm;`

   Não necessariamente a variável `_fsm` necessita ser do tipo *private*, bem com poderia ter outro nome. Já o tipo `My_ApplicationContext` é o mesmo da classe Java criada pelo compilador *SMC* a qual implementa a *FSM*.

2. No construtor da classe <*My_Application.java*>, adicionar a seguinte declaração:

   ```
   _fsm = new _ApplicationContext (this);
   ```

   Isto de fato cria uma instância do objeto *FSM* .

2. Sempre que se desejar disparar uma transição prevista na FSM, deve-se escrever:

   ```
   _fsm.<transition>();
   ```

   Onde "<*transition*>" é o nome do evento de transição previsto na *FSM*.

Todas as ações previstas na *FSM*, devem ser implementadas na classe de aplicação Java <*My_Application.java*> como métodos públicos ou pacote, de tal forma que o código da *FSM* gerado pelo compilador *SMC* possa acessá-los.

ANEXO B

Requisitos Gerais de Segurança para um Sistema de Controle de Metro-Ferroviário

Este anexo descreve os Requisitos Gerais de Segurança (RGS) aplicáveis a um Sistema de Controle Metro-Ferroviário. O objetivo da apresentação desses requisitos é ilustrar este trabalho com um conjunto de RGS aplicáveis a diversos sistemas de sinalização metro-ferroviária. Nesse sentido, tais requisitos não constituem em um padrão adotado pelas empresas operadoras, mas definem um conjunto coerente de exigências de segurança [CAMARGO, 1996]

Conceitos Fundamentais de Segurança Metro-Ferroviária

Tomando por base aspectos de segurança usuais em sistemas de sinalização metro-ferroviária, os requisitos aqui descritos se referem as seguintes premissas fundamentais de segurança:

- Prevenção de Colisões
- Prevenção de Descarrilamentos
- Prevenção de Atropelamentos

Desta forma podemos afirmar que o objetivo essencial de qualquer sistema de controle metro-ferroviário que implemente requisitos de segurança críticas é executar de *forma segura* (entenda-se com alto nível de confiança e no sentido de falha segura), as funções básicas de controle de movimentação dos trens, dentre as quais podemos citar:

- Proteção dos aparelhos de mudanças de via, onde estão localizados as máquinas de chave, cujo travamento em caso de ocupação dos respectivos de via ou no caso de rotas estabelecidas, é necessário ser garantido.
- Controle de Tráfego
- Seleção dos Códigos de Velocidade

De fato existem outras funções de controle não citadas, que também devem fazer parte de análise de segurança de sistemas de controle metro-ferroviários, no entanto, no contexto deste trabalho as citadas são essenciais.

Detalhamento dos Requisitos Gerais de Segurança

A seguir são apresentados alguns Requisitos Gerais de Segurança (***RGS***), definidos em [FDTE 542.401. AS. 001].

- RGS_1: Só pode haver alinhamento de urna rota em condições normais do sistema se não houver tráfego estabelecido no sentido oposto ao bloqueio de saída dessa rota.
- RGS_2: Se houver proibição de um bloqueio como saída de uma rota, não poderá haver alinhamento da rota que utilize esse bloqueio como saída.
- RGS_3: Se houver proibição de um bloqueio como entrada de urna rota, não poderá haver alinhamento de rota que utilize esse bloqueio como entrada.
- RGS_4: Só poderá haver alinhamento de rota em condiç6es normais do sistema se todos os circuitos de via pertencentes ã rota estiverem desocupados.
- RGS_5: Só poderá haver alinhamento de rota se n~o houver outra rota conflitante com a primeira. Define-se como Rota Conflitante qualquer rota que possa vir a provocar urna colisão em relação a uma outra rota.
- RGS_6: Um bloqueio só pode ser aberto se todas as máquinas de chave envolvidas estiverem eletricamente travadas em posições que definam uma rota prevista no intertravamento.
- RGS_7: Só poderá haver cancelamento de urna rota alinhada por desocupação seqüencial, ou seja, cancelamento automático, ou por cancelamento pelo operador.
- RGS_8: O cancelamento de rota pelo operador não deverá ser efetivado se os circuitos de via pertencentes à rota alinhada já tiverem sido ocupados pelo trem.
- RGS_9: O cancelamento de uma rota pelo operador só deverá ser efetivado após uma temporização suficiente para parar o trem antes que ele ultrapasse o bloqueio de entrada. Essa temporização deverá ser de 60 segundos. Se o trem

não estiver a uma distância do bloqueio que permita a sua parada, o trem irá ocupar os circuitos de via pertencentes a rota e o RGS_8 deverá ser garantido.

- RGS_10: No cancelamento de uma rota por desocupação seqüencial, cada circuito de via só deverá deixar de fazer parte da rota após ter sido desocupado pelo trem.
- RGS_11: Só poderá haver destravamento de uma máquina de chave se esta não pertencer a nenhuma rota e o circuito de via da máquina de chave estiver desocupado.
- RGS_12: O perfil seguro de velocidade que se antecede a uma ocupação deve ser respeitado pelo intertravamento.
- RGS_13: O perfil seguro de velocidade que antecede a um bloqueio fechado deve ser respeitado pelo intertravamento.
- RGS_14: No alinhamento de rota, a seleção de código de velocidade deverá obedecer ao sentido de tráfego estabelecido e ao perfil de velocidade imposto pelas condições da via.
- RGS_15: O trecho correspondente ao fim de via deverá estar com c6digo de velocidade de 0 Km/h e com sinalização apropriada.
- RGS_16: Só poderá haver geração de código de velocidade superior a 0 Km/h para um trem que percorrer uma rota já alinhada em condições normais.
- RGS_17: Quando o sistema receber um comando de restrição de velocidade, ele deverá impor um limite máximo de velocidade em todos os circuitos de via envolvidos que estejam com um valor de código de velocidade acima desse limite.
- RGS_18: Só poderá haver abertura de um bloqueio como entrada de uma rota de chamada após um intervalo de tempo de 60 segundos, contado a partir da imposição de código de velocidade de 0 Km/h aos circuitos de via pertencentes á rota. A finalidade desse tempo é permitir ao trem, que eventualmente esteja trafegando em sentido oposto, parar completamente antes da abertura do bloqueio para a rota de chamada.

- RGS_19: Na ocorrência de violação de bloqueio, deverá ser imposto código de velocidade de 0 Km/h nos circuitos de via pertencentes à região invadida, com o fechamento imediato de todos os bloqueios abertos dessa região. A violação de bloqueio ocorre quando o trem não respeita o sinaleiro vermelho, ou o bloqueio fechado.
- RGS_20: Só poderá haver efetivação da inversão de sentido de tráfego se o bloqueio para o qual o(s) trem(s) se dirigirá(ão) não estiver sendo utilizado como saída de uma outra rota.
- RGS_21: A efetivação do modo de manutenção em uma região da via só pode ocorrer se não houver nenhuma rota alinhada na região. Rotas que estejam alinhadas devem entrar em processo de cancelamento. Os códigos de velocidade nos trechos da região em manutenção devem ser 0 Km/h e devem ser gerados perfis de frenagem nos trechos que antecedem os bloqueios de acesso à região.
- RGS_22: Uma vez efetivado o modo manutenção, nenhuma rota pode ser alinhada na respectiva região da via.
- RGS_23: A permanência de pessoas e a movimentação de veículos dentro de uma região em manutenção devem estar regulamentadas por procedimentos operacionais.
- RGS_24: Quando o trem estiver em operação manual, a distância segura entre dois trens deve ser garantida por procedimentos operacionais.
- RGS_25: Só poderá haver alinhamento de rota de chamada se não existir qualquer outra rota de chamada em sentido oposto ou proibição de bloqueio de saída.
- RGS_26: A simulação de ocupações na via não pode de forma alguma gerar condições que levem a situações inseguras.
- RGS_27: Um bloqueio deve permanecer fechado enquanto houver pelo menos uma requisição de proibição de abertura desse bloqueio.

Em função da arquitetura adotada pelo sistema de controle metro-ferroviário, novos requisitos deverão ser gerados visando cobrir aspectos de segurança decorrentes da arquitetura adotada.