JORGE MARTINS SECALL

**Avaliação comparativa do impacto do emprego de técnicas de
programação defensiva na segurança de sistemas críticos**

Tese apresentada à Escola Politécnica da
Universidade de São Paulo como parte dos
requisitos para a obtenção do grau de Doutor em
Engenharia.

Área de concentração: Engenharia Elétrica,
Sistemas Digitais.

Orientador Prof. Livre-Docente João Batista
Camargo Júnior

São Paulo

2007

Este exemplar foi revisado e alterado em relação a versão original, sob responsabilidade única do autor com a anuência de seu orientador.

São Paulo, 26 de março de 2007

Assinatura do autor: _____

Assinatura do orientador: _____

FICHA CATALOGRÁFICA

Secall, Jorge Martins

Avaliação comparativa do impacto do emprego de técnicas de programação defensiva na segurança de sistemas críticos – Edição Revisada/ J.M. Secall. -- São Paulo, 2007. 201p.

Tese (Doutorado) - Escola Politécnica da Universidade de São Paulo. Departamento de Engenharia de Computação e Sistemas Digitais.

1.Sistemas críticos (Segurança) 2.Softwares 3.Programação defensiva 4.Injeção de falhas I.Universidade de São Paulo. Escola Politécnica. Departamento de Engenharia de Computação e Sistemas Digitais II.t.

DEDICATÓRIA

Aos meus pais, Jordi e Maria, que com sua maravilhosa combinação de senso crítico e sensibilidade, produziram o ambiente que me formou.

Ao meu filho Jorge e a minha esposa Rosângela, cuja compreensão, abnegação e apoio proporcionaram o vigor necessário para conduzir uma empreitada como essa, simultaneamente à labuta da vida cotidiana.

AGRADECIMENTOS

A Nelson de Carvalho Scaglione, meu chefe e companheiro, pelo apoio incondicional, pelo respeito à liberdade criativa e à pluralidade de idéias.

Ao Prof. Dr. João Batista Camargo Júnior, meu orientador, pela atenção, confiança, apoio irrestrito e compreensão. Sem seu apoio nas horas mais difíceis este trabalho simplesmente não existiria.

A Rafael Zorzetti Pereira pela dedicação na produção das versões do *software* de teste e a execução dos testes necessários à comprovação das idéias apresentadas nessa tese, em especial pela confiança em embarcar numa empreitada desconhecida.

A Alexander J. Zumalde pela produção do ambiente de teste e das organizadas telas do simulador. Agradeço a sua dedicação gratuita e compromissada, mesmo em sua adversidade particular.

A Mônica de Fátima Loureiro pela revisão cuidadosa e pelas valiosas sugestões para a melhora do texto.

A José Carlos Mora e Marcelo Sanchez Turrini, pela ajuda inestimável no processamento dos dados com a ferramenta Weibull++.

Ao Metrô – Companhia do Metropolitano de São Paulo, pela confiança depositada e demonstrada, pelos investimentos materiais, pela disponibilização do tempo e dos recursos necessários à execução desse Doutorado.

“Caminante, no hay camino, se hace camino al andar”.

Antonio Machado

RESUMO

Com o objetivo da redução do tempo de desenvolvimento de produtos comerciais, *hardwares* padronizados, como microcontroladores e microprocessadores dedicados, têm sido largamente empregados em aplicações críticas, transferindo para o *software* elementos até então de responsabilidade exclusiva do *hardware*. Técnicas de programação defensiva são mecanismos preventivos contra a ocorrência de falhas de *hardware* ou de *software*. Para a verificação da segurança de sistemas de aplicações críticas, técnicas de injeção de falhas foram desenvolvidas, propiciando o teste dos mecanismos de tolerância a falhas em condições muito semelhantes às do ambiente operacional real. A introdução de técnicas de programação defensiva aumenta a segurança dos sistemas de aplicações críticas. Não há, na literatura pesquisada, qualquer referência a uma avaliação quantitativa das técnicas de programação defensiva. Esta tese é a descrição de um trabalho experimental, que visa esta avaliação quantitativa, e se organiza em algumas etapas. Primeiro, algumas técnicas de programação defensiva são apresentadas, caracterizadas e eleitas como objeto de avaliação. A seguir, técnicas de injeção de falhas são descritas e uma delas é eleita como meio de teste do trabalho experimental. A partir daí, as técnicas de programação defensiva são verificadas sob o enfoque da técnica de injeção de falhas escolhida. O resultado é uma avaliação quantitativa relativa da eficácia de algumas técnicas de programação defensiva na capacidade de tolerância a falhas inseguras de sistemas de aplicações críticas. Ao final, indicações de continuidade do trabalho são apresentadas. O ambiente metroferroviário, em que trabalha o autor, foi utilizado como estudo de caso. Entretanto, as considerações e conclusões desta tese se aplicam a qualquer sistema de missão crítica.

Palavras-chave: 1.Sistemas críticos (Segurança) 2.*Softwares* 3.Programação defensiva 4.Injeção de falhas

ABSTRACT

Aiming the reduction of commercial systems' time to the market, standardized *hardware*, as microcontrollers and embedded microprocessors, has been broadly employed for critical applications, transferring to the *software* issues that once exclusively relied on the *hardware* design. Defensive programming techniques are preventive engines against *hardware* and *software* faults. In order to verify the safety of critical application systems, fault injection techniques were developed, allowing for the testing of fault tolerant techniques under conditions quite close to actual operational environments. The introduction of defensive programming techniques increases the safety of critical application systems. There are no references, on a large research base, on quantitative evaluations of defensive programming techniques. This thesis describes an experimental work towards a relative quantitative evaluation, organized in a few stages. First, some defensive programming techniques are shown, characterized and selected as the evaluation target. Following, fault injection techniques are described and one of them is selected as the agent of the experimental work. From this point on, the defensive programming techniques are verified under the fault injection technique chosen. The result is a relative quantitative evaluation on the efficiency of some defensive programming techniques on the unsafe fault tolerance capacity of critical application systems. Finally, indications for further work are presented. The railway environment, where the author works, was employed as a case study. However, the reasoning and the conclusions of this thesis are applicable to any critical mission system.

Keywords: 1.Critical Systems (Safety) 2.*Softwares* 3.Defensive Programming 4.Fault Injection

LISTA DE ILUSTRAÇÕES

Figura 1 - Nível de integridade de segurança de uma aplicação.	28
Figura 2 - O ciclo de vida da segurança.	29
Figura 3 - O "medidor" de SIL.	38
Figura 4 - Avaliando o SIL de várias técnicas de programação defensiva.	39
Figura 5 - Qualidade do Serviço e Dependabilidade do Sistema.	41
Figura 6 - Curvas da confiabilidade, da probabilidade e da densidade de probabilidade de falhas.	42
Figura 7 - Efeitos das Falhas dentro de um Sistema.	45
Figura 8 - Uma câmara de vácuo com uma fonte de Califórnio ²⁵² colocada sobre um <i>chip</i> com o encapsulamento removido.	82
Figura 9 - O processo de formação do Califórnio 252.	84
Figura 10 - Taxa de falhas pelo tempo com bombardeio de prótons.	85
Figura 11 - Injeção de falhas por <i>bursts</i> de alta energia.	87
Figura 12 - Arranjo básico do <i>hardware</i> do experimento (Estação de testes).	91
Figura 13 - Vista interna do computador empregado para o PC1, mostrando a placa de E/S instalada.	94
Figura 14 - A placa de E/S PXB-721 da Qualitech.	95
Figura 15 - Características funcionais dos 8255 da placa de E/S.	96
Figura 16 - A placa de desenvolvimento AVR Total Flex da Symphony.	97
Figura 17 - Adaptador ATMEGA128 da AVR Total Flex.	98
Figura 18 - O processador ATMEGA128 e a sua capacidade de I/O.	100
Figura 19 - A placa do gravador AVR ISP da Symphony.	101
Figura 20 - A interface de programação STK300 da Atmel.	101
Figura 21 - O diagrama de comunicação entre os módulos.	102
Figura 22 - Diagrama de blocos dos arranjos de injeção de falhas por RF.	105
Figura 23 - Transmissor Alinco DX-77T utilizado como gerador principal do experimento.	106
Figura 24 - <i>Transverter</i> Ten Tec 1210 30MHz - 146MHz.	107
Figura 25 - Detalhe do ajuste de potência do <i>transverter</i> .	108
Figura 26 - Diagrama de blocos do <i>transverter</i> Ten Tec 1210.	108
Figura 27 - Variações harmônicas do sinal interferente.	109
Figura 28 - Padrão interferente utilizado no experimento.	111
Figura 29 - Interface dedicada para geração do padrão interferente.	111
Figura 30 - Foto do experimento montado, já com a placa USB.	113
Figura 31 - Arranjo final do <i>hardware</i> do experimento com a introdução da interface USB.	113
Figura 32 - Interface de E/S USB para geração do padrão interferente.	114
Figura 33 - Amplificador de radiofrequência de 35W empregado no experimento.	115
Figura 34 - Wattímetro e sintonizador de antena OPEK SWR-20ATM.	116
Figura 35 - Watímetro e medidor de ROE Hansen FS5.	117
Figura 36 - Carga ôhmica de terminação para o <i>stripline</i> .	117
Figura 37 - Antena ICOM FA-B44V utilizada no experimento.	118
Figura 38 - Modelos de blindagens experimentais empregadas.	119
Figura 39 - Arranjo experimental com bobina sob o ATMEGA128.	120
Figura 40 - Stripline, como a utilizada no experimento (Tradução nossa).	121
Figura 41 - Campo elétrico no interior de uma <i>stripline</i> em 50 e 200MHz.	122
Figura 42 - Uso de stripline como arranjo de teste da norma MIL-STD-461.	123
Figura 43 - A stripline construída para o experimento desta tese.	124
Figura 44 - Esboço do sistema a ser controlado.	130
Figura 45 - Visualização da interface operacional que se apresenta ao usuário.	131
Figura 46 - Janela da aplicação de verificação sobre a IHM, apontando uma falha insegura.	134
Figura 47 - Interface do sistema de desenvolvimento integrado do compilador mikroPascal.	148
Figura 48 - Resultado de uma série de testes.	151
Figura 49 - A técnica de comparação das séries de testes.	152
Figura 50 - Comparação dos dados com desvios padrões computados.	156
Figura 51 - Análise comparativa das médias das diversas técnicas de programação defensiva: $t[s]$ até a falha insegura.	157
Figura 52 - Comparação através do pico da função densidade de probabilidade de falhas inseguras e das médias.	158
Figura 53 - Comparação das médias com o pico das distribuições e o MTTF.	159
Figura 54 - Tamanho do arquivo em relação a robustez à falhas inseguras.	160
Figura 55 - Comparação normalizada das técnicas de programação defensiva.	161
Figura 56 - Comparativo das funções Segurança $S(t)$.	162
Figura 57 - Comparativo das probabilidades de falha insegura $F(t)$.	163
Figura 58 - Comparativo das densidades de probabilidade de falhas inseguras.	164
Figura 59 - Função Segurança $S(t)$ para o programa Exp_1.	173
Figura 60 - Probabilidade de falhas inseguras $F(t)$ para o programa Exp_1.	173
Figura 61 - Densidade de probabilidade de falhas inseguras $f(t)$ para o programa Exp_1.	174
Figura 62 - Função Segurança $S(t)$ para o programa Exp_Back_Recovery.	174
Figura 63 - Probabilidade de falhas inseguras $F(t)$ para o programa Exp_Back_Recovery.	175
Figura 64 - Densidade de probabilidade de falhas inseguras $f(t)$ para o programa Exp_Back_Recovery.	175
Figura 65 - Função Segurança $S(t)$ para o programa Exp_Div.	176
Figura 66 - Probabilidade de falhas inseguras $F(t)$ para o programa Exp_Div.	176
Figura 67 - Densidade de probabilidade de falhas inseguras $f(t)$ para o programa Exp_Div.	177
Figura 68 - Função Segurança $S(t)$ para o programa Exp_Div2.	177
Figura 69 - Probabilidade de falhas inseguras $F(t)$ para o programa Exp_Div2.	178
Figura 70 - Densidade de probabilidade de falhas inseguras $f(t)$ para o programa Exp_Div2.	178
Figura 71 - Função Segurança $S(t)$ para o programa Exp_dupla.	179
Figura 72 - Probabilidade de falhas inseguras $F(t)$ para o programa Exp_dupla.	179

Figura 73 - Densidade de probabilidade de falhas inseguras $f(t)$ para o programa Exp_dupla.	180
Figura 74 - Função Segurança $S(t)$ para o programa Exp_dupla6.	180
Figura 75 - Probabilidade de falhas inseguras $F(t)$ para o programa Exp_dupla6.	181
Figura 76 - Densidade de probabilidade de falhas inseguras $f(t)$ para o programa Exp_dupla6.	181
Figura 77 - Função Segurança $S(t)$ para o programa Exp_dupla7.	182
Figura 78 - Probabilidade de falhas inseguras $F(t)$ para o programa Exp_dupla7.	182
Figura 79 - Densidade de probabilidade de falhas inseguras $f(t)$ para o programa Exp_dupla7.	183
Figura 80 - Função Segurança $S(t)$ para o programa Exp_Forward_Recovery.	183
Figura 81 - Probabilidade de falhas inseguras $F(t)$ para o programa Exp_Forward_Recovery.	184
Figura 82 - Densidade de probabilidade de falhas inseguras $f(t)$ para o programa Exp_Forward_Recovery.	184
Figura 83 - Função Segurança $S(t)$ para o programa ExpBCH.	185
Figura 84 - Probabilidade de falhas inseguras $F(t)$ para o programa ExpBCH.	185
Figura 85 - Densidade de probabilidade de falhas inseguras $f(t)$ para o programa ExpBCH.	186
Figura 86 - Função Segurança $S(t)$ para o programa ExpChecksum.	186
Figura 87 - Probabilidade de falhas inseguras $F(t)$ para o programa ExpChecksum.	187
Figura 88 - Densidade de probabilidade de falhas inseguras $f(t)$ para o programa ExpChecksum.	187
Figura 89 - Função Segurança $S(t)$ para o programa ExpChecksum2.	188
Figura 90 - Probabilidade de falhas inseguras $F(t)$ para o programa ExpChecksum2.	188
Figura 91 - Densidade de probabilidade de falhas inseguras $f(t)$ para o programa ExpChecksum2.	189
Figura 92 - Função Segurança $S(t)$ para o programa ExpPar.	189
Figura 93 - Probabilidade de falhas inseguras $F(t)$ para o programa ExpPar.	190
Figura 94 - Densidade de probabilidade de falhas inseguras $f(t)$ para o programa ExpPar.	190
Figura 95 - Função Segurança $S(t)$ para o programa ExpPar2.	191
Figura 96 - Probabilidade de falhas inseguras $F(t)$ para o programa ExpPar2.	191
Figura 97 - Densidade de probabilidade de falhas inseguras $f(t)$ para o programa ExpPar2.	192

LISTA DE TABELAS

Tabela 1 - Percentagem relativa de paradas das primeiras centrais telefônicas microprocessadas.....	18
Tabela 2 - Normas aplicadas no Metrô de São Paulo.....	25
Tabela 3 - Os níveis SIL segundo a norma IEC61508.....	26
Tabela 4 - As exigências de programação defensiva para cada SIL, segundo a norma IEC61508.....	27
Tabela 5 - Probabilidade de falhas inseguras por SIL.....	38
Tabela 6 - A classificação dos riscos.....	46
Tabela 7 - A Matriz de riscos.....	47
Tabela 8 - Projeto de arquitetura de <i>software</i> segundo a 61508.....	50
Tabela 9 - Capacidade de detecção/correção de um código com distancia <i>d</i>	69
Tabela 10 - Capacidade de correção dos códigos BCH.....	70
Tabela 11 - Características dos métodos para injeção de falhas.....	78
Tabela 12 - Resumo dos sinais do ponto de vista da placa de E/S.....	103
Tabela 13 - Detalhamento dos sinais de interface do ponto de vista da placa de E/S.....	104
Tabela 14 - Tamanho de células TEM em relação à faixa de frequências.....	123
Tabela 15 - Experimento para determinação do par frequência/potência ideal para o teste.....	125
Tabela 16 - As entradas e saídas de cada aplicação da plataforma de testes.....	126
Tabela 17 - A codificação das informações com a interpretação das combinações binárias.....	127
Tabela 18 - Dados erráticos no caso de sobrecarga de falhas no semicondutor.....	153
Tabela 19 - Comparação estendida da Tabela 18 entre as versões sem proteção e com BCH.....	153
Tabela 20 - Conjunto de dados, resultado dos experimentos. Tempo para falha insegura [s].....	154
Tabela 21 - Análise estatística básica dos dados dos experimentos.....	155

LISTA DE ABREVIATURAS E SIGLAS

AD	Analógico Digital
ALARP	<i>As Low As Reasonably Practicable.</i> (Tão baixo quanto razoavelmente praticável).
ANATEL	Agência Nacional de Telecomunicações
ANSI	<i>American National Standards Institute.</i> (Instituto nacional de normas dos Estados Unidos da América).
API	<i>Application Programming Interface.</i> (Interface de programação para aplicações).
AVR	Marca fantasia de microcontroladores da Atmel (http://www.atmel.com), fabricante do ATMEGA128 utilizado nessa pesquisa.
BART	<i>Bay Area Rapid Transit.</i> (Metrô de São Francisco – Estados Unidos).
CDMS	Confiabilidade, Disponibilidade, Manutenibilidade e Segurança. Versão em Português para RAMS.
CENELEC	<i>Comité Européen de Normalisation Electrotechnique.</i> (Comitê europeu de normalização electrotécnica).
CMOS	<i>Complementary metal oxide semiconductor.</i> (Semicondutor de óxido metálico complementar).
COR	<i>Carrier Operated Relay.</i> (Relê Operado por Portadora).
CW	<i>Continuous Wave.</i> (Onda contínua – Portadora telegráfica).
DoD	<i>Department of Defense.</i> (Departamento de Defesa dos Estados Unidos da América).
DUT	<i>Device Under Test.</i> (Dispositivo sob teste).
EISA	<i>Extended Industry Standard Architecture.</i> (Arquitetura estendida padrão da Indústria – Barramento de PCs).
EMC	<i>ElectroMagnetic Compatibility</i> (Compatibilidade eletromagnética).
EMI	<i>Electromagnetic interference</i> (Interferência eletromagnética).
FDTE	Fundação para o Desenvolvimento Tecnológico da Engenharia

FMEA	<i>Failure Modes and Effects Analysis.</i> (Análise dos modos de falhas e seus efeitos).
FMECA	<i>Failure Modes and Effects Critical Analysis.</i> (Análise dos modos de falha críticos e seus efeitos).
FORTTRAN	<i>Formula Translation.</i> (Linguagem de programação científica pioneira, ainda em uso).
FTA	<i>Fault Tree Analysis.</i> (Análise de árvore de falhas).
GAMAB	<i>Globalement Au Moins Assez Bon.</i> (Globalmente aceito como satisfatório).
GAS	Grupo de Análise de Segurança
HAZOP	<i>Hazard and Operability Studies.</i> (Estudos de perigos e operabilidade).
HF	<i>High Frequency.</i> (Frequência alta, tipicamente abaixo de 30 MHz).
IEC	<i>International Electrotechnical Commission.</i> (Comissão Eletrotécnica Internacional).
IHM	Interface Homem Máquina. (Tela de interação entre o operador e o processo a ser controlado).
ISO	<i>International Organization for Standardization.</i> (Organização internacional para a normatização).
ISP	<i>In System Programmer.</i> (Programador AVR no próprio circuito, sem gravador externo).
JNI	<i>Java Native Interface.</i> (Interface Nativa Java).
JVM	<i>Java Virtual Machine.</i> (Máquina Virtual Java).
LSB	<i>Less Significant Bit.</i> (<i>bit</i> menos significativo).
METRÔ	Companhia do Metropolitano de São Paulo
MISRA	<i>Motor Industry Software Reliability Association.</i> (Associação para a Confiabilidade de <i>Software</i> da Indústria Automotiva).
MOS	<i>Metal oxide semiconductor.</i> (Semicondutor de óxido metálico).

MSB	<i>Most Significant Bit</i> (<i>bit</i> mais significativo).
MTTF	<i>Mean Time To Failure.</i> (Tempo médio para falhar).
MTTR	<i>Mean Time to Repair.</i> (Tempo médio até o reparo).
MTTUF	<i>Mean Time To Unsafe Failure.</i> (Tempo médio para uma falha insegura).
NASA	<i>National Air Space Agency.</i> (Agência espacial dos Estados Unidos da América).
NUREG	<i>Nuclear Regulatory Commission.</i> (Comissão de regulamentação nuclear).
PHA	<i>Preliminary Hazard Analysis.</i> (Análise preliminar de perigos).
RAMS	<i>Reliability, Availability, Maintainability (or Maintainability) and Safety.</i> (Confiabilidade, disponibilidade, manutenabilidade e segurança).
RF	Radiofrequência
RGS	Requisitos Gerais de Segurança
ROE	Relação de ondas estacionárias.
SEU	<i>Single Event Upset.</i> (Distúrbio de evento único).
SIL	<i>Safety Integrity Level.</i> (Nível de integridade de segurança).
SL	<i>Signed Long.</i> (Variável lógica de uma <i>word</i> – 16 <i>bits</i> - mais sinal).
SS	<i>Signed Short.</i> (Variável lógica de um <i>byte</i> – 8 <i>bits</i> - mais sinal).
TEM	<i>Transverse ElectroMagnetic</i> (Eletromagnética transversa).
UC	<i>Unsigned Character.</i> (Variável lógica de um <i>byte</i> – 8 <i>bits</i> - codificando caracter).
UHF	<i>Ultra High Frequency.</i> (Frequência ultra alta, tipicamente de 300 MHz a 3GHz).
VHF	<i>Very High Frequency.</i> (Frequência muito alta, tipicamente de 30 a 300 MHz).

SUMÁRIO

1	INTRODUÇÃO	17
1.1	Objetivos	17
1.2	Justificativa	17
1.3	Conclusão	20
2	TÉCNICAS ATUAIS PARA A AVALIAÇÃO DO NÍVEL DE SEGURANÇA	21
2.1	Padrões nacionais, corporativos e particulares	21
2.2	Normas	22
2.2.1	A IEC 61508 e os SIL – <i>Safety Integrity Levels</i>	25
2.2.2	O “medidor” de SIL	27
2.2.3	Ciclo de Vida CDMS (Confiabilidade, Disponibilidade, Manutenibilidade e Segurança)	30
2.2.3.1	Fase 1: Concepção	30
2.2.3.2	Fase 2: Definição do Sistema e as Condições de Aplicação	30
2.2.3.3	Fase 3: Análise de Riscos	31
2.2.3.4	Fase 4: Requisitos do Sistema	32
2.2.3.5	Fase 5: Distribuição dos Requisitos do Sistema	33
2.2.3.6	Fase 6: Projeto e Implementação	33
2.2.3.7	Fase 7: Fabricação	34
2.2.3.8	Fase 8: Montagem	34
2.2.3.9	Fase 9: Validação do Sistema	35
2.2.3.10	Fase 10: Aceitação do Sistema	35
2.2.3.11	Fase 11: Operação e Manutenção	36
2.2.3.12	Fase 12: Análise de Desempenho	36
2.2.3.13	Fase 13: Modificações e Retrabalhos	37
2.2.3.14	Fase 14: Descomissionamento e Desmontagem	37
2.2.4	A injeção de falhas na determinação do SIL	37
2.2.5	A CENELEC EN50128	39
3	TÉCNICAS DE PROGRAMAÇÃO DEFENSIVA	48
3.1	A programação defensiva segundo a norma IEC61508	49
3.1.1	O objetivo da programação defensiva segundo a norma IEC61508	51
3.1.2	Técnicas de programação defensiva segundo a norma IEC61508	51
3.2	Técnicas de programação defensiva baseadas em análises estáticas	53
3.2.1	A padronização como proteção contra erros	55
3.2.2	Uso das convenções dos compiladores como programação defensiva	62
3.3	Técnicas de programação defensiva baseadas em análise dinâmica	64
3.3.1	Diretrizes para a programação defensiva dinâmica	66
3.3.2	Detectores de falhas	67
3.3.3	Interceptando exceções comuns	68
3.3.4	Códigos de detecção e correção de erros	68
3.3.5	Estratégias dinâmicas de correção e/ou prevenção de erros	71
3.4	Seleção de técnicas de programação defensiva para o experimento	72
4	TÉCNICAS DE INJEÇÃO DE FALHAS	75
4.1	Injeção por <i>hardware</i> versus injeção por <i>software</i>	77
4.2	Injeção de falhas por simulação	78
4.3	Injeção de falhas por <i>software</i>	79
4.4	Injeção de falhas por <i>hardware</i>	80
4.4.1	Partículas carregadas	81
4.4.2	Radiofrequência	86

4.4.2.1	Radiofrequência irradiada.....	87
4.4.2.2	Radiofrequência conduzida	88
4.5	Escolha da técnica de injeção de falhas para o experimento.....	88
5	ARRANJO EXPERIMENTAL.....	90
5.1	O <i>Hardware</i> de testes	93
5.1.1	A estação de testes.....	93
5.1.1.1	O cartão de entradas e saídas digitais	94
5.1.2	A placa-alvo de testes	96
5.1.3	A comunicação entre os módulos	102
5.2	Injetor de falhas	104
5.2.1	O rádiotransmissor	106
5.2.2	O <i>transverter</i> 146/30	107
5.2.3	A interface do programa gerador de falhas com o rádio	110
5.2.4	O cartão de E/S USB	112
5.2.5	Amplificador de RF.....	114
5.2.6	Acoplamento de impedâncias, Watímetro e monitor de relação de ondas estacionárias.....	115
5.2.7	Os sistemas irradiantes.....	117
5.3	O <i>software</i> empregado	125
5.3.1	O simulador do mundo real	127
5.3.2	A aplicação injetora de falhas	128
5.3.3	A aplicação de visualização	130
5.3.4	A aplicação de verificação.....	132
5.3.5	As aplicações-alvo do teste.....	136
5.3.5.1	Determinação da linguagem de programação do programa-alvo ..	136
5.3.5.2	O núcleo básico das aplicações alvo.....	137
5.3.5.3	Duplicação de Variáveis.	137
5.3.5.4	Block Recovery, tipo Forward Recovery.....	139
5.3.5.5	Block Recovery, tipo Back Recovery.....	139
5.3.5.6	Diversidade.....	140
5.3.5.7	Polinômio de recuperação tipo BCH.....	141
5.3.5.8	Check Sum.	142
5.3.5.9	Paridade	143
5.3.5.10	Determinação da carga de trabalho – <i>workload</i> significativo	143
6	REALIZAÇÃO EXPERIMENTAL	145
6.1	O processo de desenvolvimento.....	145
6.1.1	Definição e formalização dos sinais de interface entre os módulos de <i>software</i> e <i>hardware</i>	145
6.1.2	Montagem e testes da configuração do <i>hardware</i> da estação de testes....	145
6.1.3	Desenvolvimento e testes do módulo de <i>software</i> do simulador	146
6.1.4	Desenvolvimento e testes do módulo de <i>software</i> da IHM (JAVA).....	146
6.1.5	Desenvolvimento e testes do módulo de <i>software</i> do microcontrolador (ADA, C e PASCAL), na versão referência, sem emprego de técnicas de programação defensiva	146
6.1.6	Teste funcional da estação de testes.....	148
6.1.7	Desenvolvimento do sistema de injeção de falhas	149
6.1.8	Desenvolvimento e testes de cada versão do módulo de <i>software</i> do microcontrolador (PASCAL).....	150
6.2	Procedimento.....	150
6.3	Resultados obtidos	151
7	CONSIDERAÇÕES FINAIS	166

7.1	Conclusão	166
7.2	Recomendações para pesquisas futuras.....	167
8	REFERÊNCIAS	170
9	APENDICE A.....	173
10	APENDICE B.....	193

1 INTRODUÇÃO

1.1 OBJETIVOS

A introdução de mecanismos de tolerância a falhas, como as técnicas de programação defensiva, aumenta a robustez dos sistemas de aplicações críticas. Entretanto, as avaliações dos níveis de integridade de segurança, proposta pela IEC61508, são apenas uma avaliação analítica do quanto as recomendações da norma foram aplicadas ao produto final. Não há, na literatura pesquisada, qualquer referência a uma avaliação quantitativa das técnicas de programação defensiva, mesmo em autores que muito se aprofundaram no tema como ARLAT *et al.* (2003), LEVESON (2002) ou LOPES (2005).

O objetivo desta tese é desenvolver um método capaz de avaliar quantitativamente o impacto das técnicas de programação defensiva na segurança dos sistemas de aplicação crítica quando expostos a falhas aleatórias. O ambiente metroferroviário em que trabalha o autor foi utilizado como estudo de caso mas, as considerações e conclusões deste trabalho se aplicam a qualquer sistema de missão crítica, como controle de aeronaves, centrais nucleares, sistemas médicos, etc.

1.2 JUSTIFICATIVA

A história do desenvolvimento da humanidade está intimamente ligada à inventividade do homem. Frequentemente, no afã de desenvolver soluções para otimizar a vida cotidiana, o homem primitivo se colocava em situações de risco de morte, portanto críticas¹.

Sistemas críticos têm, dessa maneira, estado presentes há séculos. A preocupação formal com o desenvolvimento de sistemas críticos tem, entretanto, uma história recente.

A partir do estabelecimento de um novo patamar tecnológico, com o advento da segunda guerra mundial, as técnicas computacionais para sistemas de missão crítica tem apresentado um desenvolvimento acelerado nos últimos trinta anos. Esse processo acompanhou o desenvolvimento da computação como um todo,

¹ Não há consenso na literatura para o conceito de sistemas críticos. No geral entende-se como crítica uma aplicação de *software* que envolva o controle da segurança de sistemas físicos que protejam a vida humana ou patrimônios ativos expressivos.

onde a busca por dependabilidade² tem sido uma constante, agregando fatores relacionados à segurança que ganharam importância maior desde os primórdios do Programa Espacial Americano.

Com o objetivo da redução do tempo de desenvolvimento de produtos comerciais, *hardwares* padronizados, como microcontroladores e microprocessadores dedicados, têm sido largamente empregados em aplicações críticas, transferindo para o *software*, elementos até então de responsabilidade exclusiva do *hardware*. Como consequência, técnicas eficientes de falha segura³ tiveram que, progressivamente, se compatibilizar e têm sido substituídas por técnicas de computação tolerantes a falhas⁴. Essas técnicas passaram a incluir o que se convencionou chamar de técnicas de programação defensiva, que são mecanismos preventivos contra a ocorrência de falhas de *hardware* ou de *software*.

Ao mesmo tempo, aparece a necessidade de efetuar testes em sistemas de aplicações críticas. Esses sistemas precisam ser testados e, para isso, como uma forma de verificação da segurança desses sistemas, técnicas de injeção de falhas foram desenvolvidas para estressar esses sistemas, propiciando o teste dos mecanismos de tolerância a falhas em condições semelhantes ou mesmo mais exigentes que as do ambiente operacional real, reduzindo, desta forma, os tempos de teste para valores factíveis.

A questão do *software* adquiriu contornos especiais, já que os primeiros sistemas complexos baseados em *software* apresentaram um desempenho sofrível, com um enorme impacto dos programas na confiabilidade dos sistemas, como ilustrado por PAULA JR. *et al.* (1987) na Tabela 1.

Tabela 1 - Percentagem relativa de paradas das primeiras centrais telefônicas microprocessadas.

Causa da parada	%
Falha <i>hardware</i> , ambiente	20
Falha <i>software</i> , recuperação incorreta	50
Procedimentos incorretos	30

Fonte: WALLACE E BARNES (1984) *apud* PAULA JR. *et al.* (1987).

² O termo “dependabilidade” não existe formalmente em Português. Trata-se de uma tradução livre do Inglês *dependability*, que tem a intenção de traduzir a soma dos quatro fatores que aparecem sob o acrônimo *RAMS*, *reliability*: confiabilidade, *availability*: disponibilidade, *maintainability* ou *maintenability*: manutenibilidade e *safety*: segurança. Alguns destes termos tampouco existem em português. Serão entretanto empregados nesta tese pela inadequação de explicá-los a cada ocorrência.

³ O termo do inglês *fail safe*, se refere à técnica que, fazendo uso do conhecimento dos modos de falha de cada componente eletrônico, implementa mecanismos que levem o sistema a uma condição segura a cada vez que estes modos ocorram.

⁴ O termo vem do inglês *fault tolerant*, e é empregado para definir sistemas que absorvem a ocorrência da falha, continuando em funcionamento normal, do ponto de vista operacional.

Esta necessidade se reforça nos países como o Brasil, em que, na área metroferroviária por exemplo, inexistente o arcabouço legal para a garantia das condições mínimas de segurança dos sistemas críticos complexos. O Brasil não tem se manifestado pelo tema, sendo a responsabilidade pública o único motor das empresas que assumem o compromisso de administrar a segurança destes sistemas. Apenas empresas com um perfil muito diferenciado buscam um planejamento de riscos adequado nesta área. Desta maneira, acidentes acontecem, e é neste momento que o poder público se manifesta. O resultado é uma tendência do poder público, em especial do judiciário, de entender que as empresas devam buscar a ausência absoluta de risco, o que é uma impossibilidade física. O autor se manifestou recentemente no sentido de que “embora esta seja uma decisão corajosa, é preciso que o Brasil, enquanto Nação soberana determine os riscos que está disposto a aceitar na área metroferroviária. Esta atitude demonstraria domínio e controle da técnica, ao invés do desconhecimento que a demanda por ‘risco zero’ denuncia” (SECALL, 2004). Parece fundamental que o Brasil, enquanto nação soberana, defina os níveis de risco que está disposto a aceitar, sejam eles maiores, menores ou em consonância com a prática mundial.

Normas têm sido desenvolvidas para aplicações de missão crítica. Entre as razões para este desenvolvimento encontra-se a conveniência de garantir que as tecnologias de computação tolerantes a falhas estivessem presentes nos novos desenvolvimentos de sistemas críticos. Normas são um acordo entre a sociedade, os operadores e os fornecedores em que os interesses são balanceados pelo próprio mecanismo do processo de desenvolvimento normativo. Há diversas normas específicas, como as *CENELEC-Comité Européen de Normalisation Electrotechnique*, *DoD-Department of Defense*, *NASA-National Air Space Agency*, etc. que visam segmentos de mercado delimitados, como os militares ou a aviação civil, e outras normas mais genéricas, como a diretiva europeia 61508 da *IEC-International Electrotechnical Commission*. Essa norma define um conceito de medida da tolerância à falhas: os níveis de integridade de segurança, ou *SILs-Safety Integrity Levels*.

Quando se considera a questão das normas é preciso relembrar que na maioria dos países elas são instrumentos indicativos, de caráter orientativo, e não uma obrigação legal. FRIEND (1992), p.119 (tradução nossa), lembra que:

normas têm pouco uso se não forem invocadas. Normas comerciais são empregadas pelos fabricantes para fornecer aos compradores uma medida da confiança de que o *software* foi construído adequadamente. Não há nenhuma obrigação de uso de técnicas normalizadas, exceto pela insistência do comprador.

1.3 CONCLUSÃO

A partir desse panorama, questiona-se acerca da possibilidade de prover os desenvolvedores de sistemas voltados para aplicações críticas de um critério objetivo que permita a seleção das técnicas de programação defensiva adequadas a cada nível de integridade de segurança (que está diretamente correlacionado com o ambiente operacional específico).

Esta tese desenvolve uma investigação para essa seleção por meio de uma pesquisa experimental, que envolve experimentos de laboratório, dados computacionais e simulação, e se organiza em algumas etapas. Primeiramente, um conjunto de técnicas de programação defensiva é apresentado, caracterizado e eleito como objeto de avaliação. A seguir, técnicas de injeção de falhas são descritas, sendo uma delas eleita como meio de teste do trabalho experimental. Em seguida as técnicas de programação defensiva são verificadas individualmente, sob o enfoque da técnica de injeção de falhas escolhida obtendo-se uma série de amostras. O resultado é uma avaliação quantitativa relativa da eficácia das técnicas de programação defensiva na capacidade de tolerância a falhas inseguras de sistemas de aplicações críticas. Ao final, indicações de continuidade do trabalho são apresentadas.

A abordagem apresentada para a comparação dos resultados das diversas técnicas de programação defensiva é estatística, proporcional ao seu comportamento sistêmico global, quando submetida a um estímulo interferente igual ao empregado para as outras técnicas. Essa premissa faz supor que, delimitado o espaço amostral e escolhida a dimensão prática adequada para a amostra, o escopo de validade das comparações pode ser estendido a outras soluções, funcionando em outros ambientes operacionais, validando a técnica. Essa discussão está aprofundada na conclusão final deste trabalho.

O autor trabalha com sinalização ferroviária há vários anos, no Metrô de São Paulo. A percepção do desenvolvimento de aplicações para sistemas críticos – e sistemas de transporte de massa certamente o são – vem majoritariamente dessa experiência, fundamentada recorrentemente na literatura internacional.

2 TÉCNICAS ATUAIS PARA A AVALIAÇÃO DO NÍVEL DE SEGURANÇA

“O uso de tecnologia computacional em sistemas de segurança é um problema complexo” (CAMARGO JR. *et al.*, 2001). A maior parte dos sistemas críticos atuais está sendo desenvolvido com padrões proprietários, estabelecidos pelos consumidores ou pelos produtores de *software*. A própria característica sigilosa, do ponto de vista da informação industrial, faz com que o intercâmbio de informações não seja livre. Parte dos esforços de pesquisa realizados dentro da academia, estão, por vezes, cobertos por manifestos de confidencialidade que encampam laboratórios de pesquisa, pesquisadores e agregados e que permitem a publicação de trabalhos apenas dentro de um escopo extremamente limitado da pesquisa global efetivamente realizada.

O conhecimento específico, dessa maneira, fica restrito aos círculos das organizações que direta ou indiretamente financiaram a realização dos estudos associados. Entretanto, “mais e mais sistemas computacionais estão supervisionando e controlando sistemas críticos de segurança, substituindo técnicas de *hardware* de falha segura. Esses sistemas são responsáveis pela segurança de indivíduos bem como das propriedades e do meio ambiente em seu domínio (ALMEIDA JR. *et al.*, 2003)”.

2.1 PADRÕES NACIONAIS, CORPORATIVOS E PARTICULARES

A padronização e a organização no desenvolvimento de *softwares* é fundamental para sistemas críticos. É intuitivo que o cuidado em repetir as mesmas boas práticas, documentando e fundamentando cada passo contribui muito no sentido de evitar erros. Desenvolver sistemas críticos sem considerar padrões rígidos de projeto e codificação pode ser considerado como impraticável.

A partir dessa constatação diversas corporações estabeleceram ou instituíram seus próprios padrões. Esta é uma realidade do mercado e a iniciativa dos órgãos normativos visa, de alguma maneira, organizar este vazio regulatório existente em algumas áreas de aplicação no Brasil.

Os processos de propriedade industrial, associados a esta constatação, favorecem ainda mais o aparecimento de padrões corporativos específicos.

Alguns governos têm demonstrado preocupações com sistemas críticos. Em alguns casos isso foi motivado por grandes acidentes, que geraram comoção

pública, pressionando por uma ação institucional. A Inglaterra, por exemplo, introduziu a política de *safety cases* após o grave acidente de Paddington. Nesta metodologia, nenhum produto destinado as funções de segurança de sistemas ferroviários pode ser utilizado sem que uma análise de segurança tenha sido conduzida e registrada, através de uma metodologia bem descrita em uma série de manuais publicados pelo HMRI-*Her Majesty Railway Inspectorate*, conhecidos como “os livros amarelos” (devido a cor de sua capa).

Este capítulo não visa sistematizar o conhecimento referente à estrutura normativa existente para sistemas de aplicação crítica, nem apresentar as práticas de programação mais adequadas no momento. O seu objetivo é subsidiar o leitor a compreender as técnicas propostas nessa pesquisa sem ter que se valer de documentos auxiliares. Dessa maneira, o próximo item apresenta normas que estão mais afeitas a área de especialidade do autor, como a IEC61508 (*INTERNATIONAL ELECTROTECHNICAL COMMISSION*, 2005), por exemplo, que é uma tentativa internacional de oferecer uma solução padronizada para a sistematização dos conhecimentos das dezenas de normativas internacionais, como citado por SQUAIR (2006), garantindo uma uniformidade de tratamento.

Embora haja uma quantidade expressiva de iniciativas normativas, os princípios básicos que norteiam estas normas existentes são comuns e estão descritos com bom nível de detalhe por SQUAIR (2006).

2.2 NORMAS

A estrutura normativa para sistemas críticos provém majoritariamente de instituições militares. Essa necessidade remonta à Segunda Grande Guerra, que, em função das necessidades impostas pelas condições de combate, impôs as primeiras exigências correlacionadas à confiabilidade de uma maneira geral e à segurança, propriamente dita, em um momento seguinte.

Entretanto, há características particulares, com relação a programas desenvolvidos para sistemas críticos, que os diferem de outras áreas do conhecimento. Em muitas áreas as normas são escritas quando o estado da arte é, de alguma maneira, conhecido. A produção de normas trata da escolha das melhores práticas entre as existentes e da imposição, ou recomendação⁵, de que

⁵ Dependendo da legislação de cada país.

os projetos seguintes à sua adoção as apliquem como uma maneira de padronizar os resultados, por razões mercadológicas ou governamentais.

No caso dos sistemas para aplicações críticas, normas foram desenvolvidas quando o conhecimento era bastante insuficiente. O autor acredita que este conhecimento global ainda está muito aquém das necessidades. Esta situação é ainda mais crítica no setor metroferroviário, origem do autor. As fronteiras de pesquisa de hoje ainda estão muito próximas da prática de engenharia. Esse ambiente faz com que as normas possam se tornar obsoletas, ou mesmo completamente inaceitáveis, antes mesmo de que sua aplicação seja largamente aceita. Neste caso específico, as normas estão próximas da pesquisa.

Desta forma, não parece apropriado adotar normas de *software* para aplicações críticas com o entendimento de que a segurança intrínseca necessária esteja assegurada. A adoção das normas garantirá uma produção uniforme com relação ao *status quo*, que, todavia, tem uma chance não desprezível de não atender a um nível de segurança desejado.

Conforme já citado, SQUAIR (2006), pág. 25, (tradução nossa) apresenta uma discussão completa sobre o assunto com uma conclusão que resume o panorama atual da estrutura normativa da segurança de sistemas críticos:

Pelo fato de focar a 'integridade' e a 'correção' (i.e. a falta de erros de projeto) dos *softwares*, as normas de segurança de *software* atuais estão aparentemente abandonando os princípios originais de segurança de sistemas e despendendo um enorme esforço para obter um resultado pouco provável. De maneira semelhante, quando olhamos para a argumentação do 'nível de integridade' inserido nas normas de segurança atuais descobrimos que o fato de ter ou não ter⁶ por si só é logicamente fraco e representa uma base fraca para demonstrar a segurança de um sistema. "É talvez sábio lembrar a nós mesmos, de tempos em tempos, dos limites de nosso conhecimento e de que a engenharia tem tanto de aplicação da experiência – eu sugiro arte, como de medidas científicas e números em uma página.

Se isso é verdade, então segurança de *software* é algo pragmático, não diz respeito à beleza e à elegância de nossas teorias, mas da aplicação da experiência das pessoas e organizações e de como nós podemos fazer o que é necessário para proporcionar um produto seguro. Se as normas atuais estiverem erradas então nós temos que mudá-las e garantir que elas representem o que nós aprendemos ao longo do último meio século a respeito da segurança de *software*.

Embora nem todas as normas avaliadas tenham sido citadas, os conceitos apresentados nesta pesquisa se referem à experiência do autor com as normas

⁶ Os mecanismos de proteção, nota do autor.

correntemente aplicadas pelo Metrô de São Paulo (vide Tabela 2), que faz uso da metodologia descrita em CAMARGO JR. *et al.* (2001).

Em uma ferrovia, de carga ou urbana - como o Metrô de São Paulo, a primeira missão crítica que demanda métodos de *software* adequados é a circulação de trens. Para isto, desde os princípios das ferrovias, foram desenvolvidos procedimentos e sistemas dedicados à garantia da manutenção da distância segura entre trens e da proteção das zonas de manobra dos mesmos, chamadas de desvios. Ao conjunto de técnicas dedicadas a essa função, chamamos de Sinalização Ferroviária ou em inglês *signalling* (os anglo-saxões mantêm a ortografia antiga com a letra “L” dupla como forma de diferenciação de outros tipos de sinais). A sinalização ferroviária progrediu de dispositivos unicamente mecânicos até sofisticados sistemas multiprocessados, em seus mais de cem anos de evolução.

No Metrô de São Paulo as preocupações com a segurança de sistemas de sinalização remontam a seus primórdios. A linha 1 azul, inaugurada nos primeiros anos da década de 1970, importou um sistema de sinalização originalmente desenvolvido pela companhia norte americana Westinghouse para o sistema de trens denominado *BART-Bay Area Rapid Transit*, da cidade de São Francisco, na Califórnia. Esse era um sistema radicalmente novo para a época, o que motivou as autoridades americanas a demandarem uma análise da segurança do sistema efetuada por uma entidade autônoma, independente dos projetistas ou compradores do sistema. A entidade escolhida foi o mundialmente conhecido escritório de engenharia Battelle⁷, que desenvolveu um conjunto de métodos para esta análise de segurança.

Quando a Westinghouse vendeu o sistema para o Metrô de São Paulo o conceito da necessidade dessas análises veio em seu bojo e passou a fazer parte da cultura tecnológica do Metrô de São Paulo. As primeiras análises para o sistema instalado no Brasil foram realizadas pela própria Battelle, mas, na sequência, providenciou-se a sua transferência para a Universidade de São Paulo-USP.

Embora não haja no Brasil uma estrutura normativa para as questões de segurança na área metroferroviária, o Metrô adota as principais referências internacionais. A comunidade dos grandes Metrôs do mundo tem se agrupado e

⁷ <http://www.battelle.org/>

buscado otimizar recursos com o compartilhamento de experiências⁸. Muito embora as normas originalmente tivessem uma característica mandatória, hoje elas assumem um papel orientador. A prática de hoje mostra empresas desenvolvendo métodos que são submetidos aos órgãos normativos que os aprovam ou não.

As principais normativas aplicadas pelo Metrô de São Paulo para aplicações de missão crítica estão listadas na Tabela 2. É um conjunto que historicamente se iniciou com as normas MIL americanas e incorporou progressivamente as normas Européias.

Tabela 2 - Normas aplicadas no Metrô de São Paulo.

EN 50126	<i>Railway Applications: The Specification and Demonstration of Dependability, Reliability, Availability, Maintainability and Safety (RAMS).</i>
EN 50128	<i>Railway Applications: Software for Railway Control and Protection System.</i>
EN 50129	<i>Railway Applications: Safety Related Electronic Systems for Signalling.</i>
IEC-61508	<i>Functional Safety: Safety Related Systems.</i>
MIL-HDBK-217F	<i>Reliability Prediction of Electronic Equipment</i>
MIL-STD-1629A	<i>Procedures for performing a Failure Mode, Effects and Criticality Analysis.</i>
MIL-STD-470B	<i>Maintainability Program for Systems and Equipment</i>
MIL-STD-471A	<i>Maintainability Verification / Demonstration / Evaluation.</i>
MIL-STD-721C	<i>Definitions of Terms for Reliability and Maintainability.</i>
MIL-STD-756B	<i>Reliability Modeling and Prediction.</i>
MIL-STD-757	<i>Reliability Evaluation from Demonstration Data</i>
MIL-STD-781D	<i>Reliability Testing for Engineering Development, Qualification and Production.</i>
MIL-STD-785B	<i>Reliability Program for Systems and Equipment Development and Production</i>
NBR 5462	Confiabilidade e Manutenabilidade - Terminologia

Fonte: Documentos internos da Cia. do Metrô sem acesso público⁹.

2.2.1 A IEC 61508 E OS SIL – SAFETY INTEGRITY LEVELS

Em toda a bibliografia analisada para esta tese, essa norma tem se mostrado como a referência mais recorrente entre as normas existentes para sistemas de

⁸ Como o CoMET-Community of Metros, por exemplo.

⁹ Embora não haja um documento público formal, todos os editais de contratação de equipamentos específicos, afeitos ao controle de trens do Metrô de São Paulo – e que são documentos públicos -, apresentam a exigência de cumprimento dessas normas ou a comprovação da obediência a normas equivalentes, como as TÜV, por exemplo.

aplicação crítica. O seu desenvolvimento concentrou as preocupações de todos os interessados em aplicações ao redor do mundo, evoluindo da norma 1508 original para a versão atual.

Esta norma estabelece quatro níveis de segurança, denominados SIL1 a SIL4. A Tabela 3 resume as características de cada um destes níveis.

Tabela 3 - Os níveis SIL segundo a norma IEC61508.

SIL	Requisitos quanto a falhas	Probabilidade média de falha funcional	Probabilidade de falha de segurança [p/ hora]
1	Falhas relativas a segurança, que não são reveladas durante a operação normal, devem ser reveladas por testes de verificação, implementados através de testes periódicos.	$\geq 10^{-5}$ e $< 10^{-4}$	$\geq 10^{-9}$ e $< 10^{-8}$
2	Falhas relativas a segurança, que não são reveladas durante a operação normal, devem ser reveladas por testes de verificação.	$\geq 10^{-4}$ e $< 10^{-3}$	$\geq 10^{-8}$ e $< 10^{-7}$
3	<ul style="list-style-type: none"> Para subsistemas sem cobertura de diagnósticos <i>on-line</i> o sistema deve estar apto a ativar uma função de segurança na presença de uma falha insegura simples; Falhas relativas a segurança, que não são reveladas durante a operação normal, devem ser reveladas por testes de verificação, implementados através de testes periódicos. 	$\geq 10^{-3}$ e $< 10^{-2}$	$\geq 10^{-7}$ e $< 10^{-6}$
4	<ul style="list-style-type: none"> Para subsistemas com cobertura de diagnósticos <i>on-line</i> o sistema deve estar apto a ativar uma função de segurança na presença de uma falha insegura simples; Após a detecção da primeira falha e após a implementação da função de segurança definida, uma falha adicional de segurança não deve implicar em uma violação do estado seguro; Falhas relativas a segurança, que não são reveladas durante a operação normal, devem ser reveladas por testes de verificação; Análises quantitativas de <i>hardware</i> devem se basear na assunção dos piores casos. 	$\geq 10^{-2}$ e $< 10^{-1}$	$\geq 10^{-6}$ e $< 10^{-5}$

Fonte: Norma IEC61508 (INTERNATIONAL ELECTROTECHNICAL COMMISSION, 2005).

Ao estabelecer os Níveis de Integridade de Segurança – SIL, a IEC61508 estabelece quais as providências recomendadas para a obtenção de cada um deles. Conforme o destaque em cinza na Tabela 4, por exemplo, extraída da norma em referência, a Programação Defensiva, cujo estudo é objeto desta tese, é recomendada para o SIL 2 e altamente recomendada para os SIL 3 e 4.

Tabela 4 - As exigências de programação defensiva para cada SIL, segundo a norma IEC61508.

Técnica/medida		SIL1	SIL2	SIL3	SIL4
	...				
3	Programação Defensiva	---	R	AR	AR
	...				
R-Recomendável					
AR-Altamente Recomendável					

Fonte: International Electrotechnical Commission (2005).

2.2.2 O “MEDIDOR” DE SIL

Embora a norma IEC61508 (*INTERNATIONAL ELECTROTECHNICAL COMMISSION*, 2005) seja dedicada a sistemas de aplicação crítica como um todo, a partir de agora se abordará a questão da segurança intrínseca dentro da ótica dos sistemas computacionais e mais especificamente do *software* empregado em aplicações críticas, que é o objetivo deste estudo.

Entende-se aqui, como desejável, a possibilidade de determinar o nível do SIL de uma peça pronta de *software*. Essa questão ganha particular relevância quando se considera que o conceito de componentes tem sido portado do *hardware* para o *software*, ou seja, peças de *software* com funções padronizadas são desenvolvidas e disponibilizadas como caixas funcionais. Ao projetista de um produto de *software* resulta a necessidade de conhecer as entradas e saídas de cada uma dessas caixas e não necessariamente os algoritmos detalhados em seu interior.

Como a IEC61508 (*INTERNATIONAL ELECTROTECHNICAL COMMISSION*, 2005) pressupõe o conhecimento detalhado de cada uma das etapas, sua aplicação prejudica em muito a produtividade no desenvolvimento de sistemas, atenuando um pouco as enormes vantagens advindas da transferência para o *software* de funções originalmente pertencentes ao *hardware*.

Conhecidas todas as fases de desenvolvimento de um produto de aplicação crítica, é possível determinar seu nível de integridade, de acordo com a IEC61508, como mostrado na Figura 1.

Figura 1 - Nível de integridade de segurança de uma aplicação.

Para que um determinado nível SIL seja atingido, várias recomendações devem ser cumpridas, conforme já ilustrado na Tabela 4. A verificação do atendimento das condições de segurança advém da introdução ou não dos mecanismos de proteção sugeridos pela norma. Entretanto, a norma não esclarece¹⁰ a origem destes mecanismos, nem o escopo de sua validação.

Embora as recomendações sejam aparentemente intuitivas e pertencentes ao senso comum, não parece adequado que uma estrutura normativa desenvolvida para sistemas de aplicações críticas não justifique - de maneira formal - a motivação teórica e sua comprovação associada, que motivaram a sua introdução na norma e, por decorrência, em todos os produtos associados ao seu emprego. Entretanto, é necessário compreender que as normas não apresentam este formalismo em função do fato de que ele não está desenvolvido.

As recomendações são contabilizadas e a sua presença ou não, e em que grau, é que motivam a classificação de uma aplicação como pertencente a um determinado nível SIL ou outro. Estas recomendações são uma coletânea da prática internacional disseminada para as aplicações de missão crítica.

O problema é que as recomendações da norma são analíticas e dependem do acompanhamento do desenvolvimento do programa ao longo de todo o seu ciclo de vida.

A norma é muito mais detalhada e extensa do que a informação aqui apresentada, o que serve de parâmetro da forma analítica como os requisitos de

¹⁰ Nem mesmo nos *drafts* da norma, a que o autor teve acesso.

segurança são avaliados nas normas CENELEC citadas e por consequência na IEC61508 que é a base conceitual adotada.

É importante notar que muito pouco pode ser feito por um projeto se este já se encontra em andamento, sem ter obedecido as normas em suas fases preliminares. A metodologia indicada pelas normas descritas é seqüencial, pressupondo sempre que a fase anterior foi completamente satisfeita.

Figura 2 - O ciclo de vida da segurança.

Fonte: EN50126 (BRITISH STANDARDS, 1999).

O item seguinte apresenta uma descrição resumida de cada uma das atividades requeridas por cada uma das fases do ciclo de vida.

2.2.3 CICLO DE VIDA CDMS (CONFIABILIDADE, DISPONIBILIDADE, MANUTENABILIDADE E SEGURANÇA)

De maneira resumida, os objetivos, requisitos e produtos bem como as atividades de verificação e validação a serem desenvolvidas durante cada uma das fases do ciclo de vida apresentado na Figura 2, podem ser descritos como:

2.2.3.1 FASE 1: CONCEPÇÃO

O objetivo desta fase deve ser o desenvolvimento de um nível de compreensão do sistema suficiente para permitir que todas as tarefas subseqüentes do ciclo de vida do CDMS sejam satisfatoriamente desenvolvidas. Esta fase visa analisar:

- a) O escopo, o contexto e o propósito do sistema;
- b) O ambiente do sistema;
- c) As implicações gerais de CDMS do sistema;
- d) As implicações de CDMS relativas a qualquer análise financeira;
- e) As implicações de CDMS relativas a qualquer estudo de viabilidade do sistema;
- f) Interação com outros sistemas;
- g) Interação com o ser humano;
- h) Requisitos prévios de CDMS e os valores de CDMS de sistema similares e/ou mais antigos;
- i) Identificar as fontes de perigo com relação ao desempenho dos parâmetros CDMS;
- j) Políticas e metas de segurança estabelecidas;
- k) Legislação vigente a respeito de segurança.
- l) Identificação do escopo da gestão dos requisitos para as atividades subseqüentes do ciclo de vida CDMS.

2.2.3.2 FASE 2: DEFINIÇÃO DO SISTEMA E AS CONDIÇÕES DE APLICAÇÃO

Esta fase visa:

- a) Definir os parâmetros básicos do sistema;
- b) Definir os limites do sistema;
- c) Estabelecer as condições que influenciam as características do sistema;
- d) Definir o escopo da Análise de Segurança do sistema;
- e) A análise preliminar de CDMS para alcançar os objetivos;
- f) A identificação preliminar dos perigos;
- g) Identificar os subsistemas associados com os perigos identificados;
- h) Identificar as causas iniciadoras de acidentes que devem ser consideradas, incluindo falhas de componentes, erros dos procedimentos, erros humanos e mecanismos dependentes de falhas;
- i) Definir o critério inicial de tolerância aos riscos;
- j) Estabelecer a política geral de CDMS para o sistema, incluindo o conceito dos requisitos de segurança e a política para a resolução dos conflitos entre disponibilidade e segurança;
- k) Estabelecer o plano de segurança para o sistema. O plano de segurança deve ser acordado entre o poder concedente e a indústria metroferroviária, devendo ser implementado, revisto e mantido ao longo de todo o ciclo de vida do sistema.

2.2.3.3 FASE 3: ANÁLISE DE RISCOS

Esta fase deve permitir:

- a) Sistemáticamente identificar e priorizar todos os perigos razoavelmente previsíveis associados ao ambiente da aplicação;
- b) Identificar a seqüência de eventos que causam o perigo;
- c) Avaliar a freqüência de ocorrência de cada perigo;
- d) Avaliar a gravidade da conseqüência de cada perigo;
- e) Avaliar o risco de cada perigo para o sistema.
- f) Determinar e clarificar a aceitabilidade do risco associado com cada perigo identificado, tendo considerado o risco em termos de quaisquer

conflitos entre a disponibilidade e os requisitos do custo do ciclo de vida do sistema.

- g) Estabelecer o Registro Histórico de Perigos como base para a gestão de riscos. O Registro Histórico de Perigos deve ser atualizado toda vez que ocorrer uma mudança nos perigos identificados ou for identificado um novo perigo, ao longo do ciclo de vida.

2.2.3.4 FASE 4: REQUISITOS DO SISTEMA

Os requisitos CDMS do sistema em consideração devem incluir:

- a) A definição do sistema e seus limites;
- b) Os parâmetros básicos do sistema;
- c) Os requisitos funcionais e de suporte do desempenho, incluindo os requisitos funcionais de segurança e os requisitos de integridade para cada função da segurança;
- d) Os requisitos do suporte logístico;
- e) As interfaces;
- f) As características do entorno do sistema;
- g) Os níveis de tolerância aos riscos para os perigos identificados;
- h) As medidas externas necessárias para a obtenção dos requisitos;
- i) Os requisitos de suporte do sistema;
- j) Os detalhes dos limites das análises;
- k) Os detalhes de quaisquer premissas adotadas.
- l) Os critérios de aceitação para os requisitos CDMS gerais;
- m) O processo de demonstração e aceitação para os requisitos CDMS gerais, facilitados pelo plano de validação de CDMS;
- n) O Plano CDMS detalhado para as tarefas das fases subseqüentes do ciclo de vida. O Plano de CDMS deve incluir as tarefas que forem julgadas serem as mais efetivas para a obtenção dos requisitos CDMS para o sistema em consideração. O Plano CDMS deve ser

acordado entre o poder concedente e a indústria metroferroviária e deve ser implementado ao longo do ciclo de vida do sistema.

- o) Aperfeiçoamento do Plano de Segurança para assegurar que todas as futuras tarefas planejadas estão consistentes com os requisitos de CDMS emergentes do sistema.

2.2.3.5 FASE 5: DISTRIBUIÇÃO DOS REQUISITOS DO SISTEMA

Esta fase deve:

- a) Alocar os requisitos funcionais para os respectivos sub-sistemas, componentes e instalações externas;
- b) Alocar os requisitos de segurança para os respectivos sub-sistemas, componentes e redução dos riscos associados às instalações externas;
- c) Especificar os respectivos sub-sistemas, componentes e instalações externas para obter a totalidade dos requisitos CDM do sistema, incluindo o impacto de causas comuns e múltiplas falhas;
- d) Rever o Plano de CDM.
- e) Especificar os requisitos para a conformidade com os requisitos dos sub-sistemas, componentes e instalações externas;
- f) Rever e atualizar o Plano de Segurança e o Plano de Validação, para assegurar que as tarefas planejadas estão consistentes com os requisitos do sistema que foram distribuídos. Áreas muito importantes incluem requisitos de independência do pessoal envolvido e o controle das interfaces do sistema nas quais a funcionalidade de segurança pode ser comprometida.

2.2.3.6 FASE 6: PROJETO E IMPLEMENTAÇÃO

Espera-se nesta fase:

- a) Projetar os sub-sistemas e componentes para alcançar os requisitos CDMS;

- b) Desenvolver o projeto dos sub-sistemas e componentes para alcançar os requisitos CDMS.
- c) Estabelecer planos, no contexto do CDMS, para as futuras tarefas do ciclo de vida;
- d) Definir, verificar e estabelecer os processo de fabricação capazes de produzir sub-sistemas e componentes com CDMS validados;
- e) Preparar o Caso de Segurança genérico para o sistema, justificando que o sistema, conforme projetado e independente da aplicação, atinge os requisitos de segurança. O Caso de Segurança requer a aprovação do poder concedente.
- f) Preparar uma Aplicação do Caso de Segurança.

2.2.3.7 FASE 7: FABRICAÇÃO

Nesta fase deve-se:

- a) Verificar e implementar os processos de fabricação;
- b) Estabelecer os planos de suporte dos sub-sistemas e componentes.

2.2.3.8 FASE 8: MONTAGEM

Deve-se nesta fase:

- a) Montar e instalar a combinação total de todos os sub-sistemas, componentes e instalações externas necessárias para formar o sistema completo, de acordo com o plano de instalação;
- b) Documentar o processo de instalação;
- c) Rever e atualizar o Plano de Segurança ao longo da instalação para assegurar que qualquer mudança no sistema ou procedimentos sejam registradas e efetivamente gerenciadas nas futuras tarefas do ciclo de vida;
- d) Iniciar o treinamento do pessoal;
- e) Disponibilizar os procedimentos de suporte;
- f) Estabelecer a provisão de sobressalentes;

- g) Estabelecer a provisão de ferramentas.

2.2.3.9 FASE 9: VALIDAÇÃO DO SISTEMA

É importante notar que os requisitos da fase 10, Aceitação do Sistema, podem ser integrados com os requisitos da fase 9, se apropriado para o sistema em consideração. Se este for o caso, então os produtos desta fase devem demonstrar que os requisitos da fase 10 foram adequadamente preenchidos na realização da fase 9. Espera-se nesta fase:

- a) Validar a combinação total dos sub-sistemas, componentes e medidas de redução de riscos externos, de acordo com o plano de validação e os registros do processo de validação;
- b) Comissionar a combinação total dos sub-sistemas, componentes e medidas de redução de riscos externos de acordo com o plano de comissionamento e registrar o processo de comissionamento;
- c) Realizar período de operação provisória, se requerido, para possibilitar a resolução de problemas com o sistema em serviço. Quando o período de operação provisório faz parte da aceitação do sistema, deve ser considerado a necessidade da segurança do sistema ser demonstrada antes do início da operação remunerada;
- d) Preparar a Aplicação do Caso de Segurança para o sistema, se já não foi elaborado na fase 6, para justificar que o sistema, como especificamente tratado nesta aplicação, está em conformidade com os requisitos de segurança do sistema. O Relatório Específico de Segurança requer a aprovação do poder concedente.

2.2.3.10 FASE 10: ACEITAÇÃO DO SISTEMA

Os objetivos desta fase incluem:

- a) A determinação de todas as tarefas de verificação e validação do sistema, especificamente a verificação e a validação do CDM e a avaliação dos casos específicos de segurança de acordo com o plano de segurança;

- b) Aceitar formalmente o sistema para a entrada em serviço, se apropriado.
- c) Revisar e atualizar o Registro Histórico de Perigos, de maneira a registrar quaisquer perigos residuais identificados durante as fases de validação e aceitação do sistema, de maneira a assegurar que os riscos destes perigos estão efetivamente administrados.

2.2.3.11 FASE 11: OPERAÇÃO E MANUTENÇÃO

Esta fase deve incluir:

- a) A monitoração do sistema e a implementação dos procedimentos de manutenção e operação, particularmente no que diz respeito as questões do desempenho e do custo associado ao ciclo de vida do sistema;
- b) A garantia da conformidade dos requisitos CDMS do sistema, ao longo desta fase.

2.2.3.12 FASE 12: ANÁLISE DE DESEMPENHO

Esta fase deve contemplar:

- a) O estabelecimento e a implementação de um processo de revisão regular para a coleta de dados estatísticos de performance operacional e dos parâmetros CDMS;
- b) O estabelecimento e a implementação de um processo de revisão regular para a aquisição, análise e a avaliação dos dados de desempenho e dos parâmetros de CDMS;
- c) O estabelecimento e a implementação de um processo de revisão regular para verificar que as premissas assumidas nos estudos de segurança permanecem válidas;
- d) A análise do desempenho, os dados CDMS e as estatísticas que influenciam;
- e) Novos procedimentos operacionais e de manutenção;
- f) Mudanças no apoio logístico do sistema.

2.2.3.13 FASE 13: MODIFICAÇÕES E RETRABALHOS

Esta fase deve incluir:

- a) O estabelecimento de um Plano de Segurança;
- b) O estabelecimento, a implementação e uma revisão regular de um processo para o controle das modificações e retrabalhos, no contexto dos parâmetros CDMS.

2.2.3.14 FASE 14: DESCOMISSIONAMENTO E DESMONTAGEM

Esta fase deve contemplar:

- a) O estabelecimento do impacto do descomissionamento e da desativação em qualquer sistema ou instalação externa associado com o sistema a ser descomissionado;
- b) A elaboração de um plano de descomissionamento;
- c) O fornecimento e a análise do desempenho dos parâmetros CDMS como parâmetro inicial para futuros sistemas, incluindo o custeio.

2.2.4 A INJEÇÃO DE FALHAS NA DETERMINAÇÃO DO SIL

Para produtores de sistemas de aplicação crítica, uma situação ideal, mas até agora não resolvida, seria a obtenção de um “medidor” de nível SIL, que possibilitasse a determinação do SIL de um determinado sistema após a sua implementação.

Uma das possibilidades para obter um medidor de SIL seria excitar a aplicação com falhas, medindo o comportamento do sistema. Essas técnicas são conhecidas como técnicas de injeção de falhas. Essa abordagem, ilustrada na Figura 3, permitiria, por exemplo, avaliar um programa desconhecido ou não documentado adequadamente quanto ao seu SIL. Esta é uma possibilidade teórica, a qual esta tese pretende dar alguma contribuição. Se formalmente comprovada esta possibilidade seria importante fazê-la constar na estrutura das normas.

A estrutura hoje apresentada pelas normas estudadas é analítica, baseando sua avaliação na verificação de que o código fonte contém os elementos necessários para a obtenção de um nível determinado de aderência ao padrão e, portanto, de segurança.

Figura 3 - O "medidor" de SIL.

A existência desse “medidor” poderia ser muito útil na avaliação da eficácia e da eficiência das diversas técnicas de programação defensiva. Bastaria medir o nível SIL de uma aplicação sem a utilização de técnicas de programação defensiva e, a partir daí, introduzir cada uma das técnicas de programação defensiva, isoladamente ou em grupos de eficácia, medindo o novo SIL alcançado. Dentro desse conceito, o “medidor” de SIL ideal seria um instrumento, virtual ou físico, capaz de analisar um determinado código executável quanto à sua segurança crítica, ou seja, quanto à sua capacidade de conduzir o sistema em que está inserido a uma condição segura quando da ocorrência de um evento potencialmente danoso à integridade do sistema crítico em discussão (falha insegura).

A cada dia mais e mais sistemas eletrônicos, incluindo os de aplicações de missão crítica, se baseiam na flexibilidade advinda do uso extensivo de *software* na configuração e implementação. Na medida em que isto acontece, a tecnologia das empresas passa a se concentrar nas aplicações e programas, o que diminui a probabilidade de que as empresas concordem voluntariamente em abrir suas aplicações para uma avaliação analítica por um órgão independente. Este problema é recorrente na área metroferroviária a que pertence o autor.

A IEC61508 (*INTERNATIONAL ELECTROTECHNICAL COMMISSION*, 2005) classifica os sistemas críticos e estabelece os níveis SIL da maneira listada na Tabela 5:

Tabela 5 - Probabilidade de falhas inseguras por SIL.

Nível de integridade de segurança	Grande demanda ou modo de operação contínuo (Probabilidade de falhas seguras por hora)
4	$\geq 10^{-9}$ to $< 10^{-8}$
3	$\geq 10^{-8}$ to $< 10^{-7}$
2	$\geq 10^{-7}$ to $< 10^{-6}$
1	$\geq 10^{-6}$ to $< 10^{-5}$

Fonte: IEC61508 (*International Electrotechnical Commission*, 2005).

A Figura 4 mostra o modelo adotado para a implementação de um medidor de SIL concebido pelo autor. Este é um dispositivo teórico e hipotético. Esta tese visa trazer uma contribuição no sentido de sua implementação mas a sua execução prática ainda está distante do conhecimento acumulado até o momento.

Apesar da impossibilidade do estabelecimento prático de um medidor de SIL desta natureza, essa concepção pode ser parcialmente posta em prática, trazendo vantagens comparativas.

Figura 4 - Avaliando o SIL de várias técnicas de programação defensiva.

Assim sendo, as questões básicas que se apresentam e para as quais esta tese pretende contribuir para a compreensão são:

- em quanto cada uma das técnicas de programação defensiva efetivamente influencia o nível de segurança da aplicação?
- há prevalência entre elas?
- seria possível definir quais as mais importantes ou significativas?
- tudo isso seria possível para *softwares* genéricos?

2.2.5 A CENELEC EN50128

A CENELEC¹¹-*Comité Européen de Normalisation Electrotechnique* é o órgão normalizador europeu para a área de engenharia eletro-eletrônica. Para os sistemas metroferroviários em particular, a CENELEC tem publicado normas que estabelecem os padrões de segurança e qualidade exigidos para esses sistemas

¹¹ <http://www.cenelec.org>

de transporte de massa de alta capacidade. A 50128 faz parte da família EN5012X, que se alicerça na norma EN50126 que trata da dependabilidade de sistemas críticos ferroviários. Como as bases teóricas da 50128 estão listadas na norma BSEN50126 (*BRITISH STANDARDS*, 1999), essa última está aqui referenciada¹².

A sigla RAMS está definida na norma BSEN50126 (*BRITISH STANDARDS*, 1999) como “*Reliability, Availability, Maintainability and Safety*” e, às vezes, aparece em Português como CDMS, que representa a tradução literal dos termos em Inglês, respectivamente confiabilidade, disponibilidade, manutenibilidade¹³ e segurança. De acordo com a norma BSEN50126 (*BRITISH STANDARDS*, 1999), o conjunto dos atributos de RAMS pode ser resumidamente descrito pelo conceito de dependabilidade. Dependabilidade é uma característica operacional dos sistemas no longo prazo e é obtida pela aplicação de conceitos de engenharia, métodos, ferramentas e técnicas estabelecidas ao longo do ciclo de vida do sistema.

A dependabilidade de um sistema pode ser caracterizada como um indicador qualitativo do grau em que o sistema, ou o subsistema e os componentes abrangidos por esse sistema, pode ser confiável para funcionar conforme o especificado, de maneira disponível e segura (*British Standards*, 1999).

A palavra dependabilidade também pode ser interpretada como uma medida do grau em que se pode depender de um sistema de aplicação crítica associado. As normas CENELEC levam em conta que “a meta de um sistema metro-ferroviário é alcançar um nível definido de tráfego em um determinado tempo e com segurança” (*BRITISH STANDARDS*, 1999).

A dependabilidade do sistema metro-ferroviário descreve a confiança com que o sistema pode garantir a realização dessa meta. A dependabilidade do sistema metro-ferroviário tem uma influência clara na qualidade com que o serviço é entregue ao cliente. A qualidade de serviço também é influenciada por outras características relativas à funcionalidade e desempenho do sistema, como a frequência do serviço, a regularidade do serviço e a estrutura tarifária. Essa relação é mostrada na Figura 5:

¹² Evitando o *apud* em cada citação.

¹³ Ou manutenibilidade. O termo não consta dos principais dicionários gerais, mas aparece no dicionário especializado de BRANCO FILHO (2000) como manutenibilidade. Entretanto, como a escolha da NBR5462 - Confiabilidade e Manutenibilidade – Terminologia, é para manutenibilidade, este termo será empregado nesta tese.

Figura 5 - Qualidade do Serviço e Dependabilidade do Sistema.

Fonte: *draft* da BSEN50126 (BRITISH STANDARDS, 1999).

Os atributos da dependabilidade (confiabilidade, disponibilidade, manutenibilidade e segurança) estão sempre correlacionados no contexto de sistemas metroferroviários. Esses conceitos aparecem há muito tempo na literatura, provenientes das normas militares norte-americanas desenvolvidas no Pós Segunda Guerra Mundial. De maneira geral pode-se definir confiabilidade a partir de uma determinada taxa de falhas típica de um sistema que é chamada convencionalmente λ , que é o número de falhas em um determinado período de tempo. Essa taxa de falhas (λ) é definida formalmente pela BSEN50126 (BRITISH STANDARDS, 1999) como sendo o número de falhas por equipamento por unidade de tempo, e é calculada para um dado conjunto de unidades de um equipamento, como o número de falhas em um período estabelecido, dividido pelo tempo de operação acumulado por essas unidades nesse período. Ou seja,

$$\lambda = \frac{n}{\sum_1^N t_i}$$

Onde:

- n = número de falhas ocorridas no período estabelecido, no dado conjunto de unidades;
- N = número total de unidades no dado conjunto;
- t_i = tempo total de operação da unidade (i) no período estabelecido.

A confiabilidade está relacionada com o MTTF (*Mean Time To Failure*) que é um tempo médio para uma falha ocorrer, sendo calculado por: $MTTF = \frac{1}{\lambda}$. É importante lembrar que esta expressão vale apenas para um λ constante. A confiabilidade $R(t)$ é definida como a probabilidade do sistema funcionar, de acordo

com a sua especificação, no período de tempo t , dado que ele estava funcionando no instante inicial t_0 . A área abaixo da curva da confiabilidade $R(t)$ corresponde ao valor do MTTF. De maneira análoga, podemos definir $F(t) = 1 - R(t)$ como a probabilidade cumulativa de falhas. Seguindo esta linha de raciocínio, podemos definir a função de densidade de probabilidade de falhas $f(t)$ (*pdf*¹⁴) como a derivada da probabilidade cumulativa de falhas, que pode ser expressa por $f(t) = \frac{dF(t)}{dt} = -\frac{dR(t)}{dt}$. A Figura 6 ilustra o tipo destas curvas:

Figura 6 - Curvas da confiabilidade, da probabilidade e da densidade de probabilidade de falhas¹⁵.

Considerando λ_u como a taxa de falhas inseguras (do inglês *unsafe*) obtém-se respectivamente a segurança $S(t)$, a probabilidade de falhas inseguras e a densidade de probabilidade de falhas inseguras, conforme mostrado na Figura 6. Estas são as curvas analisadas no experimento proposto por esta tese. O trabalho proposto nesta tese visa avaliar a segurança de peças de *software* através destes conceitos apresentados.

¹⁴ Do inglês *probability density function*.

¹⁵ Notas de aula do Prof. Guido Stolfi para a disciplina PTC0527 – EPUSP – 2002.

A manutenibilidade, por sua vez, descreve a dificuldade temporal em se manter um sistema. Este conceito está relacionado com o MTTR (*Mean Time to Repair*), que é o tempo médio entre a parada de um sistema por falha e a devolução desse sistema ao serviço após o reparo.

A disponibilidade assintótica pode então ser definida como a proporção entre os tempos de funcionamento e de parada por falha, de maneira porcentual, ou seja, da porcentagem do tempo total de vida de um sistema em que ele está efetivamente disponível para o serviço. A disponibilidade assintótica pode ser expressa por:

$$Ass = \frac{MTTF}{MTTF + MTTR}$$

O MTUUF – *Mean Time To Unsafe Failure* corresponde à área abaixo da curva da segurança $S(t)$, utilizando aqui a taxa de falhas inseguras λ_u .

Os mecanismos de segurança implementados para garantir um nível adequado de segurança freqüentemente conduzem o sistema à indisponibilidade. Uma deficiência em qualquer um desses elementos, ou a má gestão de requisitos de disponibilidade e de segurança conflitantes no projeto, pode resultar em uma dificuldade para se obter um sistema seguro.

Ainda segundo a BSEN50126 (*BRITISH STANDARDS*, 1999), visando garantir a segurança na operação de sistemas complexos e atingir os objetivos de disponibilidade desejados, devem ser satisfeitos os requisitos de segurança, confiabilidade e de manutenibilidade, além de se controlar as atividades de manutenção em longo prazo e o ambiente operacional.

Teoricamente, a segurança pública, no que se refere ao vandalismo e infrações legais, também poderia ser considerada como um componente com impacto em todos os parâmetros RAMS. Para isso um metrô deve ser considerado como um grande sistema complexo. Essa alternativa não foi, porém, considerada nessa norma. A literatura não trata esse tema, mas a experiência do autor na operação do Metrô de São Paulo indica essa possibilidade. Uma plataforma extremamente lotada, por exemplo, faz com que os usuários segurem as portas dos trens no seu momento de fechamento, diminuindo por consequência a confiabilidade do serviço de trens nas estações subseqüentes. O deslocamento das equipes de manutenção, por sua vez, também fica prejudicado, aumentando o

MTTR e, por consequência, afetando a disponibilidade do sistema. A segurança intrínseca, na visão do autor, não deve ser afetada pela segurança pública. Voltando para a norma BSEN50126 (*BRITISH STANDARDS*, 1999), os conceitos técnicos de confiabilidade e disponibilidade estão baseados no conhecimento:

- de todos os possíveis modos de falha do sistema, no contexto do sistema específico e no ambiente;
- da probabilidade de ocorrência de um modo de falha do sistema ou alternativamente, da taxa de ocorrência de modos de falha do sistema;
- da causa e do efeito de cada falha ou modo de falha na funcionalidade do sistema;
- assuntos relativos aos recursos humanos.

Além disto, esta conceituação visa a provisão:

- da detecção e localização eficiente de falhas;
- do restabelecimento eficiente do sistema sob falha;
- da manutenção com custo efetivo reduzido ao longo da vida útil requerida para o sistema.

Os conceitos de segurança e risco estão fortemente relacionados. O risco é avaliado pelo produto da probabilidade de ocorrência de um estado inseguro com a severidade da ocorrência de um eventual acidente. Por conseguinte, segurança e risco são baseados principalmente no conhecimento:

- de todas as funções do sistema relacionadas à segurança;
- de todos os possíveis modos de falha do sistema relacionados à segurança na aplicação especificada e no ambiente (um subconjunto de todos os modos de falha possíveis do sistema);
- de todos os possíveis perigos para a operação segura do sistema e a sua frequência de ocorrência;
- da probabilidade da ocorrência de um modo de falha relacionado à segurança, ou alternativamente, a taxa de ocorrência dos modos de falha relacionados à segurança;

- das consequências de um evento perigoso ou de um modo de falha relacionado à segurança;
- da probabilidade de que um modo de falha relacionado à segurança ou um evento perigoso conduzam a um evento indesejável ou acidente, quando evidenciado em conjunto com uma cadeia de outros incidentes ocorrendo simultaneamente;
- das características de proteção da segurança e as medidas de redução do risco;
- da influência do fator humano na operação segura do sistema.

A partir dessas considerações, o risco pode ser definido como complemento da probabilidade de funcionamento seguro do sistema, ou seja, o risco pode ser considerado como a probabilidade de funcionamento inseguro do sistema. Essas correlações estão mostradas na Figura 7.

Segundo o *draft* da BSEN50126 (*BRITISH STANDARDS*, 1999) - observação esta que foi retirada da versão publicada da norma, mas com a qual o autor concorda - para qualquer sistema, funcionando como parte de uma aplicação, os defeitos interiores ao sistema têm sempre um efeito na aplicação. Todas as falhas afetam, de alguma maneira, a confiabilidade do sistema. Entretanto, falhas específicas vão afetar a segurança da aplicação. O ambiente também pode influir na funcionalidade do sistema, conforme ilustrado na Figura 7.

Figura 7 - Efeitos das Falhas dentro de um Sistema.

Fonte: *draft* da BSEN50126 (*BRITISH STANDARDS*, 1999).

O risco tolerável para um sistema metro-ferroviário qualquer depende do nível de segurança estabelecido como referência. Em alguns casos, existe

legislação que requer a apresentação formal de evidências para demonstrar a adequação da segurança do sistema.

A norma BSEN50126 (*BRITISH STANDARDS*, 1999) estabelece claramente como deve ser a gestão da dependabilidade do sistema, desde seu projeto até seu descomissionamento. Entretanto, estes pontos não serão abordados aqui por não serem foco desta pesquisa.

Um aspecto importante dessa norma é sua abordagem quanto à avaliação de riscos, fornecendo uma metodologia numérica de mensuração de riscos, onde se leva em conta a severidade do risco e sua probabilidade de ocorrência. Embora o resultado seja numérico, a sua determinação é apenas qualitativa, na medida em que são atribuídos valores para a percepção do risco que é, por definição, uma avaliação subjetiva.

Esta subjetividade, por consequência, levará a valores diferentes em função das características do avaliador ou mesmo do meio social em que ele está inserido. O que é tolerável para uma sociedade pode ser intolerável para outra. A classificação dos riscos da BSEN50126 (*BRITISH STANDARDS*, 1999) é apresentada na Tabela 6:

Tabela 6 - A classificação dos riscos.

Risco	Avaliação do Risco	Redução/Controle do Risco
R1	Intolerável	Deve ser eliminado
R2	Indesejável	Deve ser aceito somente quando a redução de risco é impraticável
R3	Tolerável	Aceitável com o controle adequado
R4	Negligenciável	Aceitável

Fonte: norma BSEN50126 (*BRITISH STANDARDS*, 1999).

A Tabela 7 apresenta a matriz de riscos derivada dos conceitos da Tabela 6.

Este tipo de classificação visa subsidiar as ações organizadas necessárias para mitigar cada um dos níveis de risco envolvidos em um projeto de aplicação crítica.

A Tabela 7 procura correlacionar a probabilidade de materialização de um determinado perigo com a sua consequência ou impacto no sistema de aplicação crítica associado. Por exemplo, um perigo que se materializado pode provocar uma interrupção na linha de 20 a 60 minutos (negligenciável) e que aconteça até 5

vezes por ano (poucas vezes por ano) aparece na Tabela 7 como R2, que na Tabela 6 está classificado como indesejável. Por outro lado, o mesmo perigo provocando uma interrupção superior a 50 vezes por ano (poucas vezes por mês) seria um R1, classificado na Tabela 6 como intolerável e que deve ser eliminado.

Tabela 7 - A Matriz de riscos.

Qualificação das probabilidades de ocorrência	Qualificação das consequências		Consequência						
			7 Trivial	6 Negligenciável	5 Marginal	4 Sério	3 Crítico	2 Catastrófico	1 Desastroso
	Fatalidades(mortes)						<5	5-50	51-500
	Dano Grave					<5	5-50	51-500	>500
	Dano Leve				<5	5-50	51-500	>500	>500
	Interrupção da Linha			20-60min	Poucas horas	1 dia	1 sem.	1 mês	Poucos meses
	Interrupção de Estação		<20min	Poucas horas	1 dia	1 sem.	1 mês	Poucos meses	1 ano
Poucas vezes por semana	>500	>=100/ano	R3	R1	R1	R1	R1	R1	R1
Poucas vezes por mês	50	>=10 e <100/ano	R4	R1	R1	R1	R1	R1	R1
Poucas vezes por ano	5	>=1 e <10/ano	R4	R2	R1	R1	R1	R1	R1
Poucas vezes em 10 anos	0,5	>=0,1 e <1/ano	R4	R3	R2	R1	R1	R1	R1
Provável de ocorrer	$5,0 \times 10^{-1}$	$\geq 1,0 \times 10^{-2}$ e $< 1,0 \times 10^{-1}$	R4	R3	R3	R2	R1	R1	R1
Improvável de ocorrer	$5,0 \times 10^{-3}$	$\geq 1,0 \times 10^{-3}$ e $< 1,0 \times 10^{-2}$	R4	R4	R3	R3	R2	R1	R1
Muito improvável de ocorrer	$5,0 \times 10^{-4}$	$\geq 1,0 \times 10^{-4}$ e $< 1,0 \times 10^{-3}$	R4	R4	R4	R3	R3	R2	R1
Remoto	$5,0 \times 10^{-5}$	$\geq 1,0 \times 10^{-5}$ e $< 1,0 \times 10^{-4}$	R4	R4	R4	R4	R3	R3	R2
Improvável	$5,0 \times 10^{-6}$	$\geq 1,0 \times 10^{-6}$ e $< 1,0 \times 10^{-5}$	R4	R4	R4	R4	R4	R3	R3
Inacreditável	$5,0 \times 10^{-7}$	$< 1,0 \times 10^{-6}$	R4	R4	R4	R4	R4	R4	R3

Fonte: norma BSEN50126 (BRITISH STANDARDS, 1999).

Para atingir estes padrões de dependabilidade estabelecidos, foram criadas normas de *software* específicas, como a CENELEC 50128 e 50129, por exemplo.

A sua base está no estabelecimento formal de boas práticas de programação. As boas práticas de programação, por sua vez, correntemente aceitas para sistemas críticos, são abordadas em profundidade em HATTON (1995).

O capítulo 3, Técnicas de programação defensiva, descreve essas técnicas e sua aplicação.

3 TÉCNICAS DE PROGRAMAÇÃO DEFENSIVA

“Técnicas de programação defensiva” é um nome genérico atribuído às diversas alternativas de projeto, que podem ser implementadas em um sistema programado para aplicações críticas, visando o incremento de seu nível de integridade de segurança. Tratam-se de mecanismos de descoberta e tratamento de erros que se aplicam em diversos contextos que serão descritos neste capítulo. Essas técnicas são fundamentais em sistemas de aplicação crítica, tendo em vista que elas aparecem em todas as normas publicadas e referenciadas nesse estudo, como a própria IEC61508 (*INTERNATIONAL ELECTROTECHNICAL COMMISSION*, 2005), que reúne as bases de todas as outras. Há mesmo uma sensibilidade de que, entre elas, algumas são mais significativas ou importantes do que outras, conforme demonstrado na Tabela 8.

Essa visão não se constitui numa avaliação quantitativa, que permita avaliar:

- em quanto cada uma dessas técnicas contribui efetivamente para o incremento do nível de segurança de uma determinada aplicação;
- em quanto cada uma dessas técnicas (ou uma combinação das mesmas) é mais efetiva do que outras (ou uma combinação de outras).

A resposta ao primeiro item resgata as preocupações de Squair (2006) quanto à necessidade de verificar a aderência de sistemas de aplicações críticas aos princípios básicos de segurança, ao invés de apenas verificar a correção e a integridade como apregoado pelas normas vigentes. Essa é, entretanto, uma alternativa impraticável no momento, já que inexistem as bases teóricas e experimentais necessárias a essa empreitada.

A busca pela resposta ao segundo item é o objetivo deste trabalho. O autor acredita que a sua resposta, associada à continuidade deste trabalho, como está proposto no capítulo 7.2, “Recomendações para pesquisas”, podem contribuir para a construção das bases experimentais que permitam a concepção de um modelo teórico capaz de representar o impacto das estratégias de projeto de *software* no nível intrínseco de segurança de sistemas de aplicações críticas.

Este capítulo visa, primeiro, listar as técnicas de programação defensiva que aparecem na norma 61508, a norma civil que, na opinião do autor, concentra os princípios teóricos que nortearam todas as outras. A seguir serão descritas de maneira geral, incluindo mas não se limitando às normas, as técnicas de

programação defensiva mais representativas na visão do autor. Por último está apresentado um subconjunto dessas técnicas que foi aplicado no experimento proposto nesta tese.

3.1 A PROGRAMAÇÃO DEFENSIVA SEGUNDO A NORMA IEC61508

A norma IEC61508 (*INTERNATIONAL ELECTROTECHNICAL COMMISSION*, 2005) lista algumas técnicas de Programação Defensiva da forma ilustrada na Tabela 8.

Esta compilação das principais técnicas apontada pela norma, de alguma forma, reúne o estado da arte no assunto. Em anos recentes, alguns pesquisadores, como LOPES (2004), têm levantado preocupações com os controles de fluxo de dados, através da modelagem de como as informações fluem em um programa e de que maneira este fluxo se organiza perante a arquitetura utilizada. Estas técnicas não estão relacionadas pela norma, provavelmente por seu desenvolvimento mais recente, mas serão comentadas quando forem discutidas as técnicas de programação defensiva dinâmicas.

O aspecto mais interessante destas indicações é o fato de que não foi possível localizar referências no texto da norma que permitam compreender a razão ou não da inclusão ou da exclusão de uma determinada técnica específica. Por consequência, o desenvolvedor de sistemas de missão crítica não tem resposta na norma para pontos estruturais do desenvolvimento de um projeto. Por exemplo, há técnicas classificadas com altamente recomendadas. Todas devem ser utilizadas? Uma delas apenas garante a classificação em um determinado nível SIL? Se não, quantas são necessárias no mínimo? Elas tem o mesmo peso ou algumas são mais importantes do que outras?

Essas perguntas e dúvidas enfrentadas pelos projetistas são o objetivo principal desta tese, que pretende oferecer uma abordagem numérica e pragmática para o problema.

No caso específico do Brasil, que, na área metroferroviária em que trabalha o autor, tem empresas desenvolvedoras de *software* para aplicações críticas, a situação é ainda mais difícil. Como não há uma estrutura normativa nacional para o assunto, existe a tendência de adoção de normas internacionais. Mas como há culturas diferentes, estas exigências variam de cliente a cliente, dificultando a padronização e elevando os custos dos desenvolvimentos nacionais.

Tabela 8 - Projeto de arquitetura de *software* segundo a 61508.

Técnica/Medida		SIL1	SIL2	SIL3	SIL4
1	Detecção e diagnóstico de falhas	-	R	AR	AR
2	Detecção de erros e códigos de correção	R	R	R	AR
3a	Programação por assertivas de falhas ¹⁶	R	R	R	AR
3b	Técnicas de <i>airbag</i> ¹⁷	-	R	R	R
3c	Programação diversa ¹⁸	R	R	R	AR
3d	Bloco de recuperação ¹⁹	R	R	R	R
3e	Recuperação para trás ²⁰	R	R	R	R
3f	Recuperação para a frente ²⁰	R	R	R	R
3g	Mecanismos de recuperação de falhas por novas tentativas	R	R	R	AR
3h	Memorização de casos executados	-	R	R	AR
4	Degradação “graciosa” ²¹	R	R	AR	AR
5	Inteligência artificial – Correção de falhas	-	NR	NR	NR
6	Reconfiguração dinâmica	-	NR	NR	NR
Uma técnica/medida numerada deve ser selecionada de acordo com o nível de integridade de segurança. Onde técnicas/medidas alternativas ou equivalentes estiverem disponíveis elas estão indicadas pela letra que segue o número ²² . Somente uma das técnicas/medidas alternativas ou equivalentes deve ser satisfeita.					
(R indica <u>Recomendável</u> , AR indica <u>Altamente Recomendável</u> e NR indica <u>Não Recomendável</u> .)					

Fonte: *International Electrotechnical Commission* (2005).

¹⁶ Uma assertiva é uma declaração da linguagem de programação que habilita o teste das hipóteses refletidas no programa. Por exemplo: se há no código um procedimento ou função que calcula a velocidade de uma partícula, uma assertiva pode garantir que a velocidade calculada seja inferior à velocidade da luz.

¹⁷ Testes do tipo *airbag* consistem em interceptar e bloquear os comandos do sistema se eles não respeitarem um conjunto de propriedades de segurança especificadas durante o desenvolvimento do sistema (KLEIN, 1991 *apud* LUSSIER *et al.* (2005)). No caso do Metrô de São Paulo este é um conjunto muito bem definido e denominado RGSs, ou requisitos gerais de segurança.

¹⁸ Votação entre *softwares* diversos (equipes, linguagens, etc.), de maneira a mitigar os erros sistemáticos através da diversidade.

¹⁹ O mesmo que a recuperação para trás quando há componentes redundantes e comparação. Para isso recupera-se um componente e a comparação é refeita (RANDELL, 1975 *apud* LYU (1995), p. 581).

²⁰ O sistema armazena os seus estados em blocos e quando detecta um erro restaura o sistema com o estado do último bloco confiável (para trás) ou para um estado seguro conhecido – geralmente degradado – (para frente) (RANDELL, 1975 *apud* LYU (1995), p. 570).

²¹ Enquanto um programa estiver apto a satisfazer o seu conjunto mais forte de requisitos ele segue a sua especificação preferencial, mas se mudanças no ambiente o forçarem a satisfazer um conjunto mais fraco, ele, então, permitirá computações mais fracas, que são indesejáveis mas toleráveis.

²² Só acontece nas técnicas/medidas numeradas como 3.

É interessante lembrar que estes conceitos estabelecidos pela IEC61508 estão nos alicerces de todas as normas desenvolvidas para sistemas de aplicação crítica conhecidas pelo autor, sejam elas militares ou civis.

3.1.1 O OBJETIVO DA PROGRAMAÇÃO DEFENSIVA SEGUNDO A NORMA IEC61508

Segundo a IEC61508 (*INTERNATIONAL ELECTROTECHNICAL COMMISSION*, 2005), a meta global que justifica a introdução de técnicas de programação defensiva é produzir programas que detectem anomalias em fluxos de controle, fluxos de dados ou valores de dados durante a sua execução e reagem a estes de uma maneira predeterminada e aceitável.

Estas técnicas visam contribuir para a robustez das aplicações, tornando os programas mais imunes, tanto às condições normalmente presentes no entorno ou às que apresentam uma probabilidade significativa de ocorrência.

Ao longo dos próximos itens discutir-se-ão as diversas naturezas das técnicas de programação defensiva.

3.1.2 TÉCNICAS DE PROGRAMAÇÃO DEFENSIVA SEGUNDO A NORMA IEC61508

Muitas das técnicas já discutidas na norma IEC61508 (*INTERNATIONAL ELECTROTECHNICAL COMMISSION*, 2005) podem ser usadas durante a programação para verificar anomalias de controle ou de dados. Elas podem ser aplicadas sistematicamente ao longo da programação de um sistema para diminuir a probabilidade de um processamento errôneo de dados.

Softwares tolerantes a falhas são projetados para acomodar até mesmo algumas de suas próprias deficiências de projeto. Duas áreas sobrepostas de técnicas defensivas podem ser identificadas:

- prevenção contra requisitos errôneos;
- prevenção contra deficiências oriundas de erros no projeto ou na codificação.

Há outras técnicas de programação defensiva, além das mostradas na Tabela 8, como por exemplo:

- verificação de variáveis quanto ao intervalo (range);
- verificação da plausibilidade dos valores;

- verificação dos parâmetros dos procedimentos quanto ao tipo (*type*), a dimensão (*dimension*) e ao intervalo (*range*) na entrada do procedimento.

As três recomendações, a seguir, ajudam a assegurar que os dados manipulados pelo programa são adequados, tanto em termos da função do programa quanto do significado físico das variáveis:

- parâmetros de apenas-leitura e de escrita-leitura devem ser separados e seus acessos verificados;
- funções devem tratar todos os parâmetros como de apenas-leitura;
- constantes literais não devem estar acessíveis para escrita. Isso ajuda a detectar a escrita acidental e o uso errôneo de variáveis.

Softwares tolerantes à falhas são projetados para “esperar” falhas no seu próprio ambiente ou para utilizar condições externas nominais ou esperadas e se comportar de uma maneira pré-definida. As técnicas incluem o seguinte:

- variáveis de entrada ou variáveis intermediárias com significado físico devem ser verificadas quanto à plausibilidade;
- o efeito das variáveis de saída deve ser verificado, preferencialmente, pela observação direta de mudanças associadas ao estado do sistema;
- o programa deve verificar a sua configuração. Isso pode incluir tanto a existência e a acessibilidade de um *hardware* esperado, como a verificação de que ele mesmo está completo. Isso é particularmente importante para a manutenção da integridade após procedimentos de manutenção.

Algumas das técnicas de programação defensiva, como as verificações de sequência do fluxo de controle, também dão conta de falhas externas.

Conceitualmente, as técnicas de programação defensiva podem ser classificadas em dinâmicas (ou de *runtime*) e estáticas. As primeiras são aquelas presentes no código compilado e que manifestam sua ação em tempo de execução, já as estáticas são aquelas presentes ou utilizadas na produção do código fonte e que se manifestam em tempo de programação ou de compilação, contribuindo para a robustez do código objeto gerado. Serão abordadas, a partir de agora, as técnicas de programação defensiva em uma conceituação geral, que vai

além daquilo proposto pelas normas. É importante salientar, que embora os exemplos apresentados estejam escritos em uma linguagem específica (em geral C ou JAVA) eles são conceitos teóricos gerais que se aplicam para praticamente qualquer linguagem de programação.

3.2 TÉCNICAS DE PROGRAMAÇÃO DEFENSIVA BASEADAS EM ANÁLISES ESTÁTICAS

Ao tratar das técnicas estáticas, nesta pesquisa, visa-se descrever algumas das diversas regras ou técnicas correntemente utilizadas para aumentar a qualidade intrínseca do produto de programação, no que tange à sua aptidão para sistemas críticos. Dessa forma, é possível prosseguir com a analogia proposta e considerar a introdução de técnicas de programação defensiva como “vacinação” ou “inoculação” de programas.

Quando se escrevem aplicações de sistemas críticos, busca-se um comportamento predeterminado do sistema com relação a seus requisitos especificados em projeto e ao seu ambiente. Para que isso aconteça, é preciso que sejam definidos e qualificados os padrões correntes da programação defensiva.

Autores como MADAU (1999) afirmam, de maneira simbólica, que programas podem ficar “doentes” e que devem ser inoculados. O uso do termo “doente” se justifica com relação aos termos “erro de programa” ou falha, pelo fato de que programas, inerentemente, não falham. Dadas as mesmas condições operacionais, programas devem se comportar de maneira previsível. Em certas situações o comportamento dos programas pode não ser o desejável e este comportamento pode induzir uma falha no sistema como um todo, mas o *software* em si mesmo não falha. Ao menos não da maneira como falha um sistema mecânico devido à fadiga.

O *software* pode ter sido codificado correta ou incorretamente em relação a uma especificação de requisitos incorreta. Os sintomas para a “doença” podem ser falhas parciais ou completas de sistemas e estão freqüentemente escondidos para a maioria das condições operacionais. Após um programa ter sido submetido à verificação e validação, ele é normalmente considerado “saudável” – prosseguindo com a analogia proposta – e pronto para liberação ou comissionamento²³.

²³ Entrada em serviço operacional.

Freqüentemente, será só na operação convencional que os problemas se mostrarão evidentes. É por isso que uma aplicação crítica deve ser submetida a testes sob condições próximas às reais para que sua validade seja comprovada. Nesse momento é que as técnicas de injeção de falhas encontram a sua aplicação.

Nem mesmo os compiladores estão imunes a problemas sintomáticos, na medida em que eles são sistemas de *software* complexos. No início da computação, os compiladores apresentavam características individuais únicas, com cada implementação – de cada fabricante – apresentando técnicas completamente diferentes. Isso se deve ao fato de que regras para a construção de compiladores simplesmente inexistiam. Mesmo quando as regras de programação foram definidas pelos comitês de padronização das linguagens, alguns produtores de compiladores, especialmente aqueles de sistemas dedicados, continuaram tratando as normas como a opinião do comitê de como o seu produto deveria se comportar. Isso é particularmente grave nas linguagens de larga difusão, como a Linguagem C, por exemplo, em que historicamente o comitê de padronização não conseguiu sequer concordar com certos aspectos da própria linguagem de programação, como a introdução das técnicas de programação orientada a objetos, por exemplo.

As preocupações da programação defensiva devem levar em consideração as limitações dos padrões de codificação e dos compiladores, transformando o ato de programar defensivamente em um hábito.

Considerando-se a analogia com a direção de um automóvel, é possível comparar os métodos usados para evitar um acidente com as técnicas de programação defensiva. Os fabricantes de automóveis desenvolvem veículos que são inerentemente estáveis e previsíveis. Por outro lado, o ambiente por onde estes veículos transitam é menos previsível e os usuários dos veículos devem ser considerados como inclinados a falhar. Para reduzir a chance de um acidente, os governos estabelecem leis que devem ser seguidas. Os motoristas obedecem a essas leis, e como uma precaução adicional, desenvolvem um estilo de direção defensiva visando evitar acidentes. Quando algum motorista se envolve em um acidente, as conseqüências são tipicamente muito menos graves do que a de motoristas que desenvolveram um estilo agressivo de condução.

Uma situação análoga se passa com o desenvolvimento de aplicações. Hoje, o comportamento dos vários compiladores é bem definido, mesmo para situações

que não aparecem nas normas. Entretanto, os ambientes dos usuários ou dos programadores são menos previsíveis. Não se pode desconsiderar que programadores são quase sempre submetidos a pressões com prazos de entrega apertados, cronogramas carregados e situações adversas de convivência ocupacional.

3.2.1 A PADRONIZAÇÃO COMO PROTEÇÃO CONTRA ERROS

A primeira grande regra dos sistemas de programação defensiva estática diz respeito à padronização. Cada projeto de desenvolvimento de sistemas deve ter um padrão de codificação formalmente documentado. Um padrão de codificação consiste numa série de regras que tratam de deficiências na norma da linguagem utilizada e/ou idiossincrasias dos compiladores e, que também define um formato ou estilo usado na escrita de código. São alguns itens típicos tratados em um padrão de codificação:

- uso de ponteiros antes de sua inicialização;
- uso de algoritmos recursivos;
- alocação dinâmica de memória;
- desvios incondicionais.

Normalmente, o padrão de codificação também ajuda a melhorar a clareza do código. Cada programador sempre tem suas práticas particulares de codificação. Dessa maneira, quando várias pessoas estão trabalhando em um projeto, um padrão comum deve ser estabelecido. Os ganhos possíveis incluem:

- geração de código: um padrão reduz a probabilidade de codificar um erro, já que reproduzir estruturas é menos suscetível a erros do que reescrevê-las de maneira semelhante;
- revisão de código: um padrão aumenta a eficácia de revisões e inspeções visuais, já que a reprodução de estruturas semelhantes cria um padrão visual que ressalta diferenças;
- qualidade: um padrão aumenta a qualidade intrínseca global do produto, tornando-o comparável com padrões estabelecidos e produtos semelhantes. Se menos erros são gerados e é mais fácil detectá-los

resultam menos erros. Se há menos erros, a qualidade intrínseca é melhor;

- manutenção: um padrão aumenta a manutenibilidade de um produto de *software*, já que é mais fácil localizar, identificar e analisar suas estruturas.

Processos de desenvolvimento de padrões ou normas apresentam sempre a dificuldade de lidar com características pessoais, levantando questões intangíveis como “gosto” pessoal. Pessoas freqüentemente entendem uma padronização diferente da sua como uma crítica pessoal. É por isso que adotar normas de escopo mundial apresenta benefícios em relação a padrões corporativos.

Entretanto, especialmente quando se trata de sistemas críticos, padrões de codificação adequados podem simplesmente não estar disponíveis. Se este é o caso, devem ser utilizadas as porções das normas existentes mais adequadas, completando-se o seu conteúdo com as necessidades do projeto.

Existem padrões para determinadas áreas, como o *Motor Industry Software Reliability Association* (MISRA) de Abril de 1998. Destinado às aplicações automotivas dedicadas, este padrão é constituído de 127 regras, sendo que 34 delas são definidas como recomendações²⁴.

Um outro padrão é o desenvolvido pela *U.S. Nuclear Regulatory Commission* em Junho de 1996 (NUREG CR-6463), denominado *Review Guidelines on Software Languages for use in Nuclear Power Plant Safety Systems*.

Um método para verificar a conformidade de um programa com um padrão se dá por meio da utilização de ferramentas de verificação estáticas, que se executam²⁵ em tempo de compilação, como procedimentos e verificadores sintáticos e de consistência formal. Estas ferramentas, apesar de poderosas, não conseguem, usualmente, garantir a conformidade total (dependendo de cada padrão de codificação).

Uma outra forma de verificar e obrigar a conformidade com o padrão se dá por meio de revisões sistemáticas do produto codificado (repetição) e inspeções do código em simulação de tempo de execução (*run through*). Um dos primeiros

²⁴ Pode ser encontrado em: <http://www.misra.org.uk>.

²⁵ Automáticas (*softwares* de verificação) ou manuais (treinamento do programador em boas práticas).

problemas encontrados na elaboração de um padrão de codificação é a estrutura dos nomes das variáveis. O objetivo é usualmente o de ser possível visualizar uma linha de código e imediatamente os tipos das variáveis e constantes utilizadas. Algumas linguagens do passado traziam, dentro de uma sintaxe exigente, as informações que nas linguagens modernas ficam por conta do programador. O FORTRAN IV (*FORmula TRANslation*) era uma dessas linguagens. Uma estrutura rigorosa de tipos é, portanto uma vantagem importante no desenvolvimento de *software* para sistemas críticos, o que nos leva a constatar as vantagens inerentes das estruturas do PASCAL e do ADA em relação à Linguagem C. Por exemplo, tomemos a seguinte linha em C:

```
Energia=Massa *(Velocidade_da_Luz_ao_Quadrado);
```

As informações disponíveis na equação dizem pouco a respeito da natureza da estrutura de tipos. Seria, entretanto, possível obter uma boa quantidade de informações dessa equação sem necessariamente recorrer a um documento de definições para verificar as variáveis e constantes utilizadas na fórmula. Da maneira como está, a tarefa de inspecionar essa linha com respeito à estrutura de tipos pode ser demorada e sujeita a erros devidos à existência de referências cruzadas. Determinar se as variáveis são locais ou globais ou mesmo se devem existir preocupações quanto a estouro nos cálculos pode ser muito difícil. Assumindo, como exemplo, a seguinte convenção abreviada para a nomeação de variáveis:

- as variáveis serão minúsculas, com um prefixo de dois ou três caracteres indicando o seu tipo, seguidas por um caractere maiúsculo para o começo do nome da variável (*ss* = *signed short*, *sl* = *signed long*, *uc* = *unsigned character*, *sc* = *signed character* e assim por diante);
- todas as variáveis locais terão caracteres maiúsculos e minúsculos intercalados na definição de seus nomes;
- todas as variáveis globais terão seus nomes em maiúsculas;
- todas as constantes serão em maiúsculas e se o tipo for estabelecido por um *#define* (diretiva da Linguagem C para estabelecimento de constantes em tempo de compilação), receberão um prefixo de dois caracteres maiúsculos seguidos por um *underscore* para definir o seu tipo.

Entende-se que, seguindo essas regras, pode-se reescrever a equação anterior da seguinte maneira:

```
slENERGIA=usMassa *(SL_VELOCIDADE_DA_LUZ_AO_QUADRADO);
```

Uma análise dessa linha de código revelaria que `slENERGIA` é do tipo *signed long* e é uma variável global.

`SL_VELOCIDADE_DA_LUZ_AO_QUADRADO`, por sua vez, é uma constante embutida no *define* como um *signed long*. A variável `usMassa` é uma variável local de tipo *unsigned short*. Uma verificação de consistência pode ser facilmente realizada, já que os tipos das variáveis estão claramente identificados. O produto de `usMassa` com `SL_VELOCIDADE_DA_LUZ_AO_QUADRADO` força `usMassa` como um tipo intermediário *signed long integer*. As vantagens da convenção dos nomes são claras e a linha de código apresentada é muito menos suscetível a erros do que a original. De maneira análoga, com a simples visualização da seguinte linha de código numa revisão:

```
ssPRODUTO = (ssNumero1 *ssNumero2) / scDivisor;
```

Será possível questionar se o produto intermediário vai caber em um *signed short* ou se seria mais seguro colocar o produto intermediário em um *signed long* antes da divisão. Colocar um dos elementos do produto como um *signed long* força o produto intermediário de `ssNumero1` e `ssNumero2` a ser promovido para um *signed long*:

```
ssPRODUTO=((signed long)ssNumero1*ssNumero2)/scDivisor;
```

Uma notação importante, com essas propriedades, é a chamada reversa ou Polonesa, que é um exemplo de uma normalização de variáveis que pode diminuir a probabilidade de erros em um código.

O autor entende que não se devem utilizar números codificados manualmente em expressões. A manutenção de um código se torna uma tarefa muito difícil quando um código está contaminado com números “mágicos”. Um número mágico tem significado para o programador no momento da codificação, mas, até mesmo para ele, perde significado ao longo do tempo. Mesmo documentado adequadamente, esse procedimento dificulta sobremaneira o exame de qualquer código. O exemplo ilustra o ponto:

$y = m * x + b$, onde m é a inclinação da linha, b é o ponto onde ela intercepta y (em $x = 0$), e x (entrada) e y (saída) são os pontos nos eixos x e y . Mesmo satisfazendo os requisitos de nomes de variáveis é possível escrever:

```
ssY = (31.872983 * ssX) + 45.3232;
```

Depois de algum tempo, essa linha pode ser de difícil compreensão. O problema pode ser menor se houver comentários adequados (embora a mesma constante possa ser utilizada em outros pontos do código, aumentando sua complexidade). A seguir compara-se a declaração anterior com a seguinte:

```
ssY=(SS_INCLINACAO_PRESSAO*ssX)+ SS_INTERCEPTACAO_PRESSAO;
```

Pode-se agora observar claramente que:

- a equação trabalha com algum tipo de constantes de pressão;
- as palavras inclinação e interceptação indicam ao leitor que a equação pode ser a de uma reta.

As constantes podem estar definidas em um arquivo de parâmetros, de definição de constantes ou no cabeçalho do arquivo. Qualquer mudança no valor da constante vai se propagar por todo o espaço de definição do programa evitando tarefas de procura de ocorrências de uma variável ou constante alteradas.

Estabelecer um padrão de indentação e adotá-lo por todo o projeto deve ser uma regra. O objetivo é prover a facilidade de leitura e, conseqüentemente, inibir a possibilidade de codificação com erro.

Considere-se o seguinte exemplo:

```
for (x=0; x<= LIMITE; ++x)

{for (y=1; y<= Y_LIMITE; ++y)

{z+= x*y;

if (x == TESTE) {

printf("x passa");

t=x/MEDIA;}}

z /= 4;}}
```

Fica muito difícil inspecionar ou verificar uma solução como essa. Este pedaço de código pode dar a impressão visual que a linha de código $z /= 4$; é

executada fora do *loop*. A ausência da indentação pode facilitar a digitação do programa quando o programador tem pressa, mas compromete a compreensão dos fluxos de dados na análise do código fonte.

Considere-se a seguir uma solução mais adequada:

```
for (x=0; x<= X_LIMITE; ++x)
{
    for (y=1; y<= Y_LIMITE; ++y)
    {
        z += x*y;
        if (x ==TEST)
        {
            printf(„x passa“);
            t = x/MEDIA;
        }
        z /= 4;
    }
}
```

Com o emprego da indentação fica mais claro ao desenvolvedor que $z /= 4$; faz parte do *loop*. Três regras básicas podem deixar programas muito mais claros, para quase todas as linguagens modernas:

- Deve-se usar apenas uma declaração por linha;
- Todas as chaves devem estar sozinhas em suas linhas;
- Toda chave de fechamento deve estar alinhada com a sua chave de abertura.

Parênteses completos sempre devem ser usados para macros e equações, de maneira a reduzir as possibilidades de erro quando se utilizam pré-processadores. Um erro comum está na não utilização de parênteses na definição de números positivos, já que é mais comum o seu uso para os negativos. Considere-se o exemplo de definição de um número positivo:

```
#define POS_NO    10
```

Quando o pré-processador for executado ele vai encontrar a localização de POS_NO e vai substituí-la pelo valor 10. Como a operação dos pré-processadores

não está padronizada²⁶, a compilação da seguinte linha de código é inteiramente dependente do compilador:

```
ssY = .POS_NO;
```

O ponto decimal antes de POS_NO pode ser interpretado como sendo mesmo um ponto decimal e a linha vai aparecer para o compilador como:

```
ssY = .10;
```

Alguns compiladores vão perceber essa falha²⁷ e gerar erros. Outros vão compilá-la normalmente e o erro só aparecerá por depuração. Entretanto, o problema poderia ser adequadamente resolvido, para qualquer compilador, usando parênteses:

```
#define POS_NO      (10)
```

O pré-processador vai substituir POS_NO por (10) e o código que aparecerá para o compilador será:

```
ssY = .(10);
```

Dessa forma, o compilador indicará um erro, já que o ponto decimal é definido em C como um operador, com atributos e restrições similares a todos os outros.

O uso de pré-processadores pode ser uma ferramenta importante para a produtividade na geração de código e praticamente todos os ambientes de desenvolvimento integrados (*IDEs – Integrated Development Environments*) fazem uso deles. São robôs que varrem o código e antecipam todos os cálculos e atribuições contidos no programa fonte.

Entretanto, a introdução de mecanismos automáticos requer cuidados especiais, já que existe potencial para a introdução de erros. Para sistemas críticos em especial, eles nunca devem ser utilizados na definição de macros complexas, já que é possível, dessa forma, introduzir uma série de operações automáticas que não são, de maneira usual, auditáveis ou verificáveis. A operação dos pré-processadores está muito mal definida nas normas das linguagens, em especial no caso do C. A ausência de estrutura normativa faz com que sua implementação dependa quase que exclusivamente dos projetistas dos compiladores (MCGETTRICK, 2000). Os

²⁶ Há apenas padrões de mercado.

²⁷ Um número inteiro será erroneamente interpretado como decimal.

pré-processadores têm várias aplicações válidas que podem ajudar na clareza e na manutenibilidade das aplicações. Eles são úteis na definição de constantes declaradas e no pré-cálculo de constantes. Esta característica permite ao usuário forçar o pré-processador a calcular as expressões matemáticas dentro de um tipo constante. O valor desse tipo fica, então, substituído em todo o escopo do código.

3.2.2 USO DAS CONVENÇÕES DOS COMPILADORES COMO PROGRAMAÇÃO DEFENSIVA

A maioria dos desenvolvedores de aplicações confia, sem se preocupar como, que os compiladores promovam e equalizem²⁸ os diversos tipos de variáveis. Quando os resultados adequados não são obtidos, a única alternativa é, por vezes, passar a verificação diretamente em Linguagem *Assembler* da função desejada, que, por vezes, é simplesmente uma equação que não funciona (*INTERNATIONAL ORGANIZATION FOR STANDARDIZATION / INTERNATIONAL ELECTROTECHNICAL COMMISSION*, 1999). Em Linguagem C, a regra de promoção é:

“Os tipos integrais *character*, *short integer*, ou um *integer bit field* e suas variedades com e sem sinal serão convertidas para *signed integer* se o *signed integer* puder representar todos os valores do tipo original; se não, ele será convertido para *unsigned integer*” (*INTERNATIONAL ORGANIZATION FOR STANDARDIZATION / INTERNATIONAL ELECTROTECHNICAL COMMISSION*, 1999).

Dessa maneira, *chars*, *shorts*, e *bit fields* acabam convertidos em algum tipo de inteiro (com ou sem sinal), antes que uma expressão seja executada internamente. Para evitar problemas potenciais, operadores não inteiros devem ser embutidos em expressões. Quando tipos como *char*, *short*, ou *bit fields* forem empregados em expressões, eles devem sempre estar confinados no tipo inteiro apropriado, de maneira a evitar erros de conversão. Se o programador não o fizer, a Linguagem C automaticamente converterá esses tipos integrais para inteiros antes da execução das expressões, e a conversão pode não apresentar o resultado adequado. Mesmo se o comportamento do compilador para a conversão destes tipos for observado, quando um novo compilador for empregado em um novo programa, o seu comportamento poderá ser diferente.

²⁸ Façam a consistência de tipos, promovendo ou rebaixando variáveis.

Operandos de precisão aritmética mista, em expressões, devem ter seus tipos fixados, já que, nestes casos, as sub-expressões são calculadas na precisão apropriada para o operando. Os resultados desejados podem não ser obtidos se a precisão resultante for maior do que a precisão do operando da expressão. Para eliminar este erro, os operandos devem ser condicionados a precisão final do resultado.

Códigos reutilizáveis são uma promessa de trazer para a engenharia de *software* os benefícios e vantagens desenvolvidos pela engenharia de componentes. Na verdade, a própria análise numérica das rotinas proposta por esta tese é uma iniciativa nesta direção. Entretanto, é preciso muito cuidado para que a implementação de códigos portáteis se torne um benefício e não uma fraqueza dos sistemas. No caso específico da Linguagem C, uma das fraquezas da padronização ANSI está na definição dos tamanhos dos tipos.

A portabilidade de tipos se torna um problema na Linguagem C porque os produtores de compiladores devem determinar o tamanho dos tipos baseados nessas restrições. A solução é definir substitutos para os tipos padrão do C, que devem aparecer no cabeçalho do arquivo, definidos a partir de tipos padrão. Os tipos substitutos são então utilizados em todo o código para condicionar e definir variáveis. Quando se muda de compilador, apenas o cabeçalho necessita ser mudado para adequar os novos tamanhos dos tipos. Essa pode ser uma fonte potencial de problemas de segurança se meios adequados não forem introduzidos de maneira a garantir que os cabeçalhos certos sejam empregados para cada compilador de um projeto.

A ordem dos campos de *bits*, por exemplo, também depende dos compiladores. Existem duas ordens de campos de bits que são comuns:

- O bit mais significativo vem primeiro (MSB), ou;
- O bit menos significativo vem primeiro (LSB).

Quando se define uma variável *bit field* a diretiva `#if` deve ser utilizada para inserir a ordem de *bits* desejada para a compilação em função de um *flag* de configuração. Alguns compiladores permitem uma linha de comando com a opção de reverter a ordem dos *bits*. Confiar na opção do compilador, entretanto, reduz a portabilidade do programa - sua capacidade de trabalhar em ambientes diferentes - e abre espaço para erros.

Outro exemplo destas convenções se dá quando se admite que macros calculem constantes antecipadamente, usando a máxima resolução do pré-processador, o que é usualmente feito em ponto flutuante. É necessário truncar ou arredondar (para baixo ou para cima), antes de colocá-lo em uma variável inteira (assumindo que se está utilizando matemática inteira no projeto). Se a precisão da variável ou constante força o arredondamento para o inteiro mais próximo, deve-se adicionar ou subtrair 0,5 do número e depois truncá-lo. Neste caso, a possibilidade de erro aparece quando se está tentando arredondar números com sinal.

Estes princípios encontram especial aplicação na área de atuação do autor, com a recente introdução das tecnologias de blocos móveis nos sistemas de sinalização ferroviários e/ou metroviários.

Até o presente momento, os sistemas de sinalização destes sistemas de transporte eram baseados na detecção da ocupação dos trens através do emprego de trechos de via discretos. Dessa maneira, a informação de ocupação era discretizada na informação da ocupação ou não de um determinado trecho de via discreto e específico. As variáveis eram então exclusivamente booleanas²⁹.

Nos novos sistemas por blocos móveis a informação da posição dos trens é introduzida através de sensores, que têm sua informação codificada em ponto flutuante e a posição efetiva dos trens é dada por cálculo matemático, introduzindo as questões de arredondamento e suscitando os cuidados e recomendações descritos anteriormente.

3.3 TÉCNICAS DE PROGRAMAÇÃO DEFENSIVA BASEADAS EM ANÁLISE DINÂMICA

A melhor forma de garantir a qualidade de um projeto de *software* é projetando-o segundo uma estrutura normativa desde o início, na fase de concepção, e mantendo a estratégia ao longo de todas as fases do ciclo de vida do sistema, conforme indicado na IEC61508 (*INTERNATIONAL ELECTROTECHNICAL COMMISSION*, 2005) ou na BSEN50126 (*BRITISH STANDARDS*, 1999), que é mais específica para a situação ferroviária. Dessa forma, as partes se encaixarão mais perfeitamente e a funcionalidade de cada parte será mais simples, de maneira que menos erros poderão ser cometidos ao se implementar as várias partes.

²⁹ Lógicas: FALSE ou TRUE apenas.

As técnicas estáticas descritas no item anterior visam diminuir os erros no código, mas, é difícil não introduzir erros durante a codificação, e uma maneira efetiva de se encontrar esses erros é utilizando-se as técnicas dinâmicas: ou seja, “aquelas que envolvem a execução do programa e a observação de seu comportamento.” (JACKSON E DEVADAS, 2005).

Existem dois problemas relacionados com a abordagem estática: o primeiro é que problemas originários das fases projeto, embora não devessem, com o tempo acabam emaranhados na implementação em decorrência dos próprios problemas da implementação, sendo muito difícil encontrá-los. Todas as medidas de proteção possíveis devem ser tomadas para evitar que essa situação aconteça com *softwares* voltados a sistemas de aplicação crítica. O segundo problema é que, sabe-se que o custo de se consertar um erro em uma peça de *software* cresce dramaticamente quanto mais tarde o erro for encontrado, dentro do processo de implementação. Em estudos da IBM e da TRW, BOEHM *et al.* (1982) descobriram que, para se consertar um erro de especificação durante a implementação, o custo poderia chegar a 1000 vezes o valor da correção do erro, caso ele tivesse sido descoberto antes do processo de implementação ser iniciado.

De maneira análoga é errôneo imaginar que apenas técnicas estáticas são suficientes. Embora, grandes progressos tenham sido alcançados na tecnologia de análise estática, ainda está muito longe a capacidade de descobrir todos. Mesmo que se tenha construído uma prova matemática de que o programa está correto, seria, no mínimo, uma tolice não testá-lo. O problema fundamental da atividade de teste é expresso em um famoso aforismo de DIJKSTRA (1974): “Os testes podem revelar a presença de erros, mas nunca sua ausência.”

No entanto, a atividade de teste é incompleta, por sua própria natureza. É necessária muita cautela nas conclusões a respeito dos parâmetros RAMS de um programa, mesmo que ele tenha passado por uma grande bateria de testes. De fato, o problema de se determinar quando um *software* é suficientemente confiável para que seja utilizado, é um grande problema. Existem poucos recursos para se resolver essa questão. Portanto, é melhor não considerar os testes como um meio de se estabelecer o quanto um programa é confiável, mas sim, apenas como uma forma de se encontrar erros. Há uma diferença sutil, mas muito importante entre estes pontos de vista.

3.3.1 DIRETRIZES PARA A PROGRAMAÇÃO DEFENSIVA DINÂMICA

“A programação defensiva é uma abordagem que visa aumentar a confiabilidade sobre um programa através da inserção de verificações redundantes.” (JACKSON E DEVADAS, 2005). Esta metodologia funciona da seguinte maneira: quando se está escrevendo um código, imagina-se as condições que devem ser validadas e mantidas em determinados pontos do código, em outras palavras, as invariantes. Então, ao invés de apenas assumir que essas invariantes são válidas, deve-se testá-las explicitamente. Esses testes são denominados ‘Certificações em Tempo de Execução’. Se uma certificação falha - ou seja, a invariante não é validada - se reporta o erro e abandona a execução. As técnicas de recuperação aplicadas nos experimentos dos capítulos a seguir são exemplos de certificações em tempo de execução.

Quando se empregam as certificações em tempo de execução, primeiramente, elas não devem ser utilizadas para amenizar a má codificação, e sim, para assegurar que um código seja isento de erros de codificação, da maneira mais efetiva. A programação defensiva não significa escrever um código inferior e depois inserir diversas certificações em tempo de execução. No longo prazo, é mais efetivo escrever um bom código desde o início, pois um código ruim representa, freqüentemente, uma desordem que pode chegar ao ponto em que não seja possível consertá-lo, a menos que se comece tudo de novo.

As certificações em tempo de execução devem ser escritas ao longo da codificação, e não depois. Quando se estiver escrevendo o código tem-se as invariantes em mente, e escrevê-las é uma boa forma de documentação. Se forem deixadas para depois, é menos provável que sejam feitas. As certificações em tempo de execução têm um custo. Elas podem desordenar o código, portanto elas devem ser utilizadas com parcimônia. Obviamente se deseja escrever as certificações que são mais propensas a encontrar erros de codificação. Para isso, normalmente, devem-se utilizar as certificações dessa forma (JACKSON E DEVADAS, 2005):

- No início de um procedimento, para verificar que o estado no qual o procedimento é invocado está como se espera - ou seja, para se verificar a pré-condição. Isso faz sentido, pois uma grande proporção dos erros está relacionada à compreensão errada das interfaces entre os procedimentos;
- No final de um procedimento complicado, para verificar que o resultado está correto - ou seja, para se verificar a pós-condição. Em um procedimento que calcula a raiz quadrada, por exemplo, se pode

escrever uma certificação que calcule o quadrado do resultado para verificar se o valor é igual ao argumento (ou, nesse caso, pelo menos aproximado). Este tipo de certificação é, às vezes, chamado de auto-verificação.

Quando uma operação que possui efeitos externos está para ser executada, em uma máquina de radioterapia, por exemplo, é interessante verificar se a intensidade do feixe de radiação está dentro de limites aceitáveis, antes de se ligar o equipamento. As certificações em tempo de execução podem, também, diminuir o desempenho da execução.

Com o aumento exponencial da capacidade computacional das máquinas modernas, essa não é uma questão preocupante, especialmente para sistemas críticos. Entretanto, a prática de se escrever certificações em tempo de execução, para se testar o código e depois desabilitá-las para o lançamento da versão oficial, implica em expor o usuário a riscos que estavam mitigados no ambiente de testes, o que não faz sentido do ponto de vista da lógica simples. A boa prática indica que se uma certificação em tempo de execução é necessária, deve-se preocupar com o custo do desempenho apenas quando houver evidências de que o custo da certificação é realmente significativo.

Todavia, não faz sentido escrever certificações absurdamente custosas. Supondo-se, por exemplo, que se tem um *array* e um índice no qual um elemento foi inserido. É sensato verificar se o elemento está lá. Mas não é sensato verificar se o elemento não está em nenhum outro lugar, procurando por ele em todo o *array* do começo ao fim, pois isso transformaria uma operação que pode ser executada em tempo constante, em uma operação que levará um tempo proporcionalmente linear (em relação ao tamanho do *array*) para ser completada. Os exemplos apresentados para técnicas dinâmicas a partir deste ponto do texto estão, como exemplo, codificados em Linguagem JAVA, da mesma forma que o foram em Linguagem C para as técnicas estáticas descritas no item 3.2. A razão para isto advém exclusivamente do fato de que os exemplos foram simplesmente adaptados das referências disponíveis, mantendo as linguagens originais.

3.3.2 DETECTORES DE FALHAS

Os detectores de falhas são componentes do *software* com a função de supervisionar constantemente o código em execução em busca de uma condição de falha. Eles estão entre os primeiros dispositivos de detecção dinâmica de falhas de

ação independente. “Tais componentes fornecem uma abstração do mecanismo de detecção de falhas que permite a simplificação dos algoritmos distribuídos, resumindo a detecção a simples consultas ao detector.” (SAMPAIO *et al.*, 2003). Os detectores são revelados por suas propriedades abstratas ao invés de implementações específicas (CHANDRA E TOUEG, 1996 *apud* SAMPAIO *et al.*, 2003), mas encontram aplicação prática imediata.

3.3.3 INTERCEPTANDO EXCEÇÕES COMUNS

O ambiente *Java Virtual Machine* (JVM) já inclui certificações em tempo de execução para importantes classes de erros (JACKSON E DEVADAS, 2005):

- Chamada de método em uma referência nula;
- Acessar um array além de seus limites;
- Realizar uma operação inválida de *downcast*.

Estes erros fazem com que exceções não verificadas sejam desprezadas. Além disso, as próprias classes da API³⁰ Java trabalham com exceções em condições de erro. É uma boa prática interceptar todas essas exceções. Uma maneira simples de se fazer isso é escrever um manipulador no topo do programa, no método *main*.³¹ Destaca-se que existem algumas exceções atribuídas pela JVM que não se deve tentar manipular, como os *stack overflows* e os *out-of-memory errors*, por exemplo, que indicam que o programa ficou sem recursos. Nessas circunstâncias, não faz sentido tornar as coisas piores ao se tentar continuar a computação, já que o próprio sistema está informando, com essas mensagens, da sua incapacidade de continuar com sucesso.

3.3.4 CÓDIGOS DE DETECÇÃO E CORREÇÃO DE ERROS

Códigos de detecção e/ou correção de erros são usualmente classificados em termos da sua habilidade em detectar ou corrigir determinadas classes de erros que afetam um determinado numero de bits da informação a ser protegida. Desta maneira, diz-se que um código de correção de erros é e-erro detectável se qualquer falha que cause até e bits errados for detectável. O significado maior desta

³⁰ *Application Programming Interface*. Um dos propósitos primários de uma API é descrever como aplicações e programadores podem acessar um conjunto de funções sem necessariamente ter acesso ao código fonte ou detalhes internos da biblioteca que contém as funções.

³¹ Um manipulador capaz de encerrar o programa apropriadamente (por exemplo, com uma mensagem de erro para o usuário, tentando, em seguida, fechar todos os arquivos abertos).

afirmação é o de que, nesta condição, o erro não é capaz de transformar uma determinada informação binária em outra. Analogamente, um código de correção de erros é e -erro corrigível se para qualquer falha que cause até e bits errados, o conjunto de todos os bits corretos de saída puder ser determinado. Isto significa que dois erros e_1 , e_2 afetando informações w_1 , w_2 , respectivamente, não resultarão na mesma informação.

A distância de Hamming d corresponde ao mínimo número de *bits* em que quaisquer duas palavras de dados válidos diferem. A capacidade de detecção e correção de falhas de um código pode ser expressa em termos de d como mostrado na Tabela 9.

Tabela 9 - Capacidade de detecção/correção de um código com distancia d .

d	Capacidade de detecção/correção
1	Nenhuma
2	Detecção de 1 <i>bit</i> errado, sem capacidade de correção
3	Detecção de 2 <i>bit</i> errado, correção de 1 <i>bit</i> errado
...	...
$e+1$	Detecção de e <i>bits</i> errados, correção de $\lceil e/2 \rceil$ ³²
...	...
$2e+1$	Detecção de $2e$ <i>bits</i> errados, correção de e <i>bits</i> errados

Fonte: BREUER E FRIEDMAN (1976).

Em geral, *bits* adicionais, chamados de bits de verificação, devem ser gerados por um programa de maneira que os dados finais constituam um código com capacidade de detecção/correção de erros. O código de paridade é o mais simples destes códigos.

Para aplicações mais complexas e críticas podem ser aplicados códigos de resíduo. Estes códigos tem a propriedade desejável de que, para as operações aritméticas de soma, subtração e multiplicação, os *bits* de verificação do resultado podem ser derivados a partir dos *bits* de verificação dos operandos. Esta propriedade é denominada verificação independente.

São alguns exemplos de códigos de detecção e recuperação de erros:

- BCH (do qual o Hamming acima descrito é um caso particular);
- Reed-Muller, Usado em transmissões de sondas espaciais;

³² $\lceil e/2 \rceil$ indica o menor inteiro não menor do que $e/2$.

- Golay, que com 12 *bits* de dados codificados em 24 *bits*, corrige até 3 *bits* errados e é usado em transmissões de sondas espaciais. Apesar de apresentar uma menor capacidade de correção de erros, permite maior velocidade de transmissão do que o anterior;
- Reed-Solomon (RS), que é empregado nos CDs, mas também em sondas espaciais;
- Códigos de convolução, etc.

Os códigos BCH (Bose-Chaudhuri-Hocquenghem), de 1960 apresentam uma abordagem pragmática para o problema da correção de erros em *bits*. Os códigos BCH podem corrigir um número de erros tão grande quanto se queira (desde que o bloco tenha bits suficientes). Desta maneira, o código de Hamming faz parte dos códigos BCH. A Tabela 10 ilustra a capacidade de correção dos códigos do tipo BCH, onde k é o número de *bits* de informação e n é o número de bits total.

Tabela 10 - Capacidade de correção dos códigos BCH.

k	n	$R=k/n$	Erros corrigidos
4	7	0.57	1
5	15	0.33	3
24	63	0.38	7
64	127	0.5	10
247	255	0.97	1
171	255	0.67	11
11	1023	0.01	255

Os códigos de Hamming clássicos (4;7 e semelhantes) são adequados à correção de erros isolados. Entretanto, um caso diferente são os erros de rajada, como aqueles devidos a fenômenos atmosféricos ou danos na superfície de um meio de armazenamento como um CD ou disco rígido de um PC. Nestes casos é preciso usar códigos de correção de erros especialmente dedicados para estas situações.

Este é o caso dos códigos RS (Reed-Solomon), desenvolvidos nos laboratórios Bell em 1960. Com estes códigos se consegue corrigir 4000 *bits* errados seguidos (correspondentes a 3 mm de uma pista de CD estragada). Trata-se de um código periódico ou *interleaved*. Entretanto, o rendimento destes códigos é ruim, o

que revela as preocupações com a carga de trabalho no caso das aplicações críticas.

No caso do CD, por exemplo, começa-se com um bloco de 24 *bytes* de dados e aplica-se uma codificação de Reed-Solomon obtendo-se 32 *bytes*. Um *frame* contém 33 *bytes* (32 de dados + correção de erros + 1 *byte* de *subcoding*). Cada *byte* é transformado em 14 *bits* para evitar demasiados zeros consecutivos + 3 *bits* de separação. Um *frame* tem 27 *bits* iniciais de sincronização mais $33 \times (14+3)$ *bits* de dados. Desta forma, 24 *bytes* (192 *bits*) de dados se transformam em 588 *bits* escritos no CD.

3.3.5 ESTRATÉGIAS DINÂMICAS DE CORREÇÃO E/OU PREVENÇÃO DE ERROS

A programação em N versões, como salientado por BURNS e WELLINGS (1991), é o equivalente de software da redundância de hardware, em que as falhas no interior de cada módulo estão ocultas do ponto de vista do exterior. Esta é uma relação estática, já que tanto cada versão do *software* tem uma relação fixa com as outras versões como elas são executadas na presença ou não de falhas. No caso das redundâncias dinâmicas, os componentes redundantes apenas entram em operação quando um erro for detectado. O experimento desta tese, quando fizer o uso do procedimento denominado Seguro, estará implementando esta técnica.

As estratégias de *back* e de *forward recovery* são exemplos de redundâncias dinâmicas. A técnica de recuperação a frente (*forward recovery*) visa continuar o processamento a partir de um ponto de falha através do exercício de correções seletivas nos estados do sistema. Para sistemas críticos, esta técnica envolve a colocação de cada uma das variáveis do sistema em sua posição de maior segurança operacional. Alguns exemplos de técnicas de recuperação a frente são o uso de ponteiros redundantes em estruturas de dados ou o emprego dos códigos de correção anteriormente descritos.

A recuperação atrás (*back recovery*), por sua vez, se baseia no retorno do sistema a uma condição de segurança conhecida anterior ao momento da falha. Neste momento uma parte alternativa do programa é executada, restaurando o sistema.

Segundo BURNS e WELLINGS (1991), estas técnicas de restauração de estados tem a desvantagem de que o estado de falha é removido sem que haja qualquer

informação a respeito da origem da falha. De maneira análoga, em sistemas críticos muitas informações podem ser dependentes no tempo, o que significa que o estado seguro armazenado pode não ser mais seguro no instante de sua restauração, gerando uma condição tão potencialmente insegura quanto a falha original.

Uma forma de contornar esta situação são os certificados de validade de uma informação em tempo de execução, freqüentemente denominados de *time stamps*.

Certificações em tempo de execução podem complicar a compreensão e a execução de programas, na medida em que elas se misturam às diretivas funcionais do programa, utilizando as mesmas variáveis e construções.

Isso é especialmente ruim se não for possível identificar quais partes do código são certificações e quais estão realizando a computação propriamente dita. Algumas linguagens de programação, como a Linguagem EIFFEL, já vêm com mecanismos de certificação embutidos (JACKSON E DEVADAS, 2005).

3.4 SELEÇÃO DE TÉCNICAS DE PROGRAMAÇÃO DEFENSIVA PARA O EXPERIMENTO

As técnicas de programação defensiva relacionadas pelo autor foram adotadas no experimento proposto nesta tese em função dos seguintes critérios:

- 1) Representatividade da técnica, por seu largo emprego. Relevância para sistemas de aplicação crítica em uso. São uma prática de mercado? Esse critério visa validar as técnicas em uso;
- 2) Exeqüibilidade do seu teste na plataforma de testes concebida para o experimento proposto nessa pesquisa.

Desta maneira, foram escolhidas as técnicas de:

1. Duplicação de variáveis: cada variável é guardada em duas posições de memória que são comparadas antes do uso. Para cada variável serão efetuadas as mesmas operações em tempo de execução e a verificação se dará por consistência, conforme sugerido por Vargas et al. (2003) para melhora da imunidade à *EMI-ElectroMagnetic Interference*;
2. Polinômio de recuperação tipo BCH (Bose, Ray-Chaudhuri, Hocquenghem): cada variável é guardada com informação de recuperação de corrupção através de código BCH;

3. *Block Recovery*, tipo *Forward Recovery*: um estado de segurança para todas as variáveis do sistema está programado no programa e, na detecção de uma falha, este estado é restabelecido e utilizado como novo *start up* para o sistema;
4. *Block Recovery*, tipo *Backward* ou *Back Recovery*: a cada intervalo, um estado seguro para todas as variáveis do sistema é armazenado e, na detecção de uma falha, este estado é restabelecido e utilizado como novo *start up* para o sistema;
5. Diversidade: cada linha de código é executada duas vezes e o resultado comparado (redundância temporal);
6. *Check sum* (soma binária) da área de dados: a cada operação de escrita o *check sum* da área de dados é recalculado e armazenado. A cada operação de leitura o *check sum* é antes calculado e comparado com o valor armazenado, validando o dado armazenado;
7. Paridade: cada variável é guardada com informação de recuperação de corrupção através de código de paridade;

No sentido prático, a partir das sete técnicas escolhidas, foram desenvolvidas, para esta tese, 13 versões de programas, com as seguintes características básicas:

- Exp_1: experimento sem utilização de técnicas;
- Exp_Back_Recovery: experimento que utiliza *back recovery*;
- ExpBCH: experimento que utiliza a técnica de codificação BCH;
- Exp_Div: experimento que utiliza técnica de diversidade com a *procedure* Seguro;
- Exp_dupla: experimento que utiliza a duplicação de variáveis de entrada e de saída com a *procedure* Seguro;
- Exp_dupla6: experimento que utiliza a duplicação de variáveis de entrada e saída sem a *procedure* Seguro;
- ExpChecksum: experimento que verifica a soma binária da área de dados;

- Exp_Div2: experimento que utiliza a técnica de diversidade sem a *procedure* Seguro;
- Exp_dupla7: experimento que utiliza a duplicação de todas as variáveis do programa sem a *procedure* Seguro;
- ExpCheckSum2: experimento que verifica a soma da área de dados sem a *procedure* Seguro;
- ExpPar: experimento que verifica a paridade para a validação do dado com a *procedure* Seguro;
- ExpPar2: experimento que verifica a paridade para a validação do dado sem a *procedure* Seguro;
- Exp_For_Recovery: experimento que utiliza *forward recovery*.

A *procedure* Seguro contém o conjunto de dados mais restritivo (seguro) para todas as variáveis do sistema. Quando uma possível falha de segurança é detectada pela própria aplicação do microcontrolador, este procedimento é executado, todas as variáveis vão para a restrição, e o processo é reiniciado.

A partir desta escolha das técnicas de programação defensiva que serão exercitadas no experimento, o próximo capítulo descreve as principais técnicas de injeção de falhas disponíveis, escolhendo uma para a elaboração do experimento descrito no capítulo 5.

4 TÉCNICAS DE INJEÇÃO DE FALHAS

Técnicas de injeção de falhas são uma forma de implementar anomalias em sistemas computacionais. Há um grupo pequeno de pesquisadores que trabalha com o conceito de validação de programas por meio da submissão dos mesmos a técnicas de injeção de falhas, construindo o conceito de *benchmarking* de segurança. Os trabalhos como os de ARLAT *et al.* (2001), LEVESON (2002) ou ARLAT *et al.* (2003) são textos relevantes nessa área, e fundamentam, nesta pesquisa, o estudo das técnicas de injeção de falhas.

Entretanto, a proposta aqui apresentada não tem o viés do *benchmarking* dos autores citados, mas o da análise comparativa da seleção de técnicas de programação defensiva discutidas no capítulo 3, quando submetidas a uma técnica de injeção de falhas. Esta é uma abordagem do autor para a qual não se encontrou paralelo na literatura pesquisada. Os trabalhos que mais se aproximam são aqueles desenvolvidos no Rio Grande do Sul, na Pontifícia Universidade Católica do Rio Grande do Sul, no grupo SiSC. LOPES (2004), LOPES (2005) e VARGAS *et al.* (2003) são exemplos de trabalhos deste grupo, com que o autor não teve contato direto, referidos nesta tese. O trabalho de LOPES (2005), que guarda semelhança na abordagem, visa trabalhar a interferência eletromagnética como um risco, propondo meios para mitigá-los. No trabalho desta tese propõe-se o uso desta fragilidade como uma chance de introduzir falhas de maneira acelerada, verificando a eficácia de técnicas de proteção para falhas introduzidas por qualquer fonte potencial de erros.

A descrição e a sistematização dos métodos de injeção de falhas apresentados neste capítulo não visam esgotar o tema, mas subsidiar o leitor na compreensão das possibilidades da metodologia da injeção de falhas e no entendimento das razões que levaram a escolha do método de radiofrequência, descrito no subitem final deste capítulo, para a execução do experimento apresentado.

Os conceitos essenciais a respeito de injeção de falhas estão inevitavelmente ligados aos conceitos da própria computação. Computação, na sua forma mais pura, trata das transições de entradas em saídas em que estados intermediários são criados ao longo do processo como sub-componentes da

execução do *software* (ARLAT *et al.*, 2001). Sub-componentes incluem estruturas como:

- declarações de código;
- funções;
- módulos;
- objetos.

A injeção de falhas é simplesmente o ato de adicionar uma transição e verificar que efeitos a transição adicional provocará na saída. De uma maneira simplificada, a injeção de falhas pode ser aplicada a (ARLAT *et al.*, 2003):

- entradas;
- sub-componentes do código;
- estados dos dados.

O objetivo é verificar o que acontecerá com a saída, ou seja, implementar técnicas de programação defensiva específicas e verificar se essas técnicas concedem imunidade contra as falhas injetadas, procurando quantificar essa imunidade, conforme descrito na parte experimental. As técnicas básicas de injeção de falhas podem ser classificadas em (ARLAT *et al.*, 2003):

- falhas injetadas por *hardware*: variam do uso de isótopos radioativos (tipicamente ^{252}Cf) à injeção direta de sinais interferentes nas entradas e posições intermediárias;
- falhas injetadas por *software*: variam de aplicações geradoras de erros num sistema multitarefa a procedimentos com disparo aleatório ou em processadores conectados por compartilhamento de memória.

O objetivo deste capítulo não é descrever essas técnicas em uma análise comparativa concludente. As referências contêm diversas fontes a respeito dessas variadas técnicas, como VOAS E MCGRAW (1998), SIVENCRONA *et al.* (2003), LOPES (2004) e KARNIK (2004).

O objetivo aqui é elencar as técnicas que despontam como alternativas adequadas ao padrão de testes proposto pelo trabalho.

4.1 INJEÇÃO POR *HARDWARE* VERSUS INJEÇÃO POR *SOFTWARE*

Infelizmente não é possível eleger um sistema de injeção de falhas universal, adequado a todas as situações, já que há vantagens e desvantagens em cada sistema. Não só algumas estratégias de injeção de falhas não são capazes de fazer determinadas tarefas, como há claras diferenças com relação à eficácia com que algumas formas de teste são realizadas, conforme discutido por ARLAT *et al.* (2003). As vantagens das técnicas de injeção por *hardware* são grandes, na medida em que permitem um controle externo aos processos, de maneira independente do processo-alvo. Essas técnicas são, em geral, de difícil aplicação, o que complica sobremaneira o caráter experimental dos poucos grupos que se aventuram nessa área. ARLAT *et al.* (2003) apresentam uma comparação importante entre as técnicas de injeção de falhas implementadas por programa e as implementadas de maneira física, caracterizando os seus respectivos impactos. Para isso, três técnicas físicas e uma implementada por *software* são comparadas e analisadas na aplicação a um único sistema distribuído, operando em tempo real.

Não há, entretanto, uma metodologia estabelecida para a eleição de quais técnicas são adequadas a cada tipo de teste desejado. As informações disponíveis sobre algumas das técnicas - ou um subconjunto delas como o artigo descrito no parágrafo anterior - estão pulverizadas em diversas fontes, como LOPES (2004), que ilustra técnicas de radiação eletromagnética induzida; KARNIK (2004), com falhas em semicondutores e SIVENCRONA *et al.* (2003), que emprega o bombardeio radioativo. Uma sistematização dessas informações - que, entretanto, foge ao escopo desta tese - seria muito benéfica para a pesquisa nessa área e deveria ser objeto de um trabalho investigativo.

Dependendo dos tipos de falhas e do esforço requerido para criá-las, um projetista de testes deve poder escolher qual dos métodos é mais apropriado à sua aplicação. Falhas permanentes, por exemplo, como *stuck-at* (que corresponde à fixação de um valor em um ponto escolhido do circuito), são melhor injetadas por *hardware*, pois assim é possível controlar o local certo onde a mesma vai ser inserida, sem interferir com o processo intercorrente. Neste mesmo caso, métodos de *software* poderiam gerar um alto *overhead* no código e, eventualmente, nem mesmo conseguir gerar a falha desejada pelo projetista do teste, devido ao esgotamento da capacidade operacional do sistema computacional escolhido. A

questão do *workload* de processadores envolvidos com aplicações de sistemas críticos será discutida adiante nesta tese.

De maneira similar, quando se deseja simplesmente corromper dados, a abordagem por *software* tem se mostrado suficiente, e falhas como *bit-flips* podem ser geradas por ambas abordagens quase indistintamente. Neste último caso, devem ser considerados outros fatores, como: custos, equipamentos adicionais, precisão, etc.

A Tabela 11 sistematiza essa comparação, mostrando as vantagens e desvantagens das técnicas de injeção por *hardware* e *software* nos aspectos de custo, risco de danos, acessibilidade do *hardware* e controlabilidade dos processos. A técnica escolhida para o experimento desta tese foi do tipo *hardware*. A principal razão para esta escolha está na capacidade única que as técnicas de injeção por *hardware* têm de ser independentes do processo sob teste. Além disto, há a intenção de dar uma contribuição adicional, advinda do fato de que há uma quantidade expressivamente menor de trabalhos empregando técnicas de injeção de falhas por *hardware*.

Tabela 11 - Características dos métodos para injeção de falhas.

	Hardware	Software
Custo	Alto	Baixo
Risco de danos	Alto	Não existe
Pontos diretamente acessíveis	Pinos e partes internas do circuito em teste	Memória
Controlabilidade	Alta (<i>Pin-level</i>), Baixa (Íons)	Alta

Fonte: LOPES (2004).

4.2 INJEÇÃO DE FALHAS POR SIMULAÇÃO

A injeção de falhas por simulação se baseia no princípio de que todo o ambiente onde a aplicação sob teste está sendo executada se encontra sob simulação. Nessa situação há controle de todas as variáveis do ambiente e as variações indutoras de erros podem ser implementadas.

Uma grande desvantagem dessa abordagem se origina do fato de que para a sua implementação, todas as fontes potenciais de problemas devem ser conhecidas. Esta não é uma situação provável, na medida em que se as situações

potencialmente indutoras de falhas fossem conhecidas, elas seriam eliminadas através de mecanismos de proteção específicos inseridos no código.

Desta maneira, pode-se concluir que há benefícios no uso de técnicas de injeção de falhas randômicas ou aleatórias. Se a sua dispersão for adequada e for possível obter um número adequado de repetições, então será possível obter informações estatisticamente significativas dentro do espaço amostral escolhido. FOLKESSON (1999), apresenta uma interessante comparação entre as diversas técnicas e em especial sobre as influências da carga de trabalho (*workload*) na eficácia das várias técnicas de injeção de falhas.

Técnicas com estas características randômicas ou aleatórias também podem ser simuladas, mas há limitantes importantes. Gerar uma variável efetivamente aleatória dentro de um domínio de validade tem sido objeto recorrente de estudo para matemáticos, já que é muito difícil garantir a aleatoriedade necessária para simulações estatísticas, como o emprego computacional do método de Monte Carlo.

De maneira geral as técnicas de simulação têm natureza similar às técnicas de injeção de falhas por *software*, descritas a seguir.

4.3 INJEÇÃO DE FALHAS POR SOFTWARE

Esse é o primeiro dos dois grandes grupos de técnicas de injeção de falhas, constituído como a família de técnicas mais comum de injeção de falhas. Sua popularidade deve-se, substancialmente, à facilidade e à versatilidade com que as falhas podem ser introduzidas. Alguns autores se referem a essas técnicas como técnicas de simulação por *software*, mas é preciso lembrar que elas apresentam caráter diverso das técnicas descritas no item 4.2.

Há várias possibilidades para implementá-las, mas pode-se dividi-las em duas grandes famílias:

- mecanismos de injeção de falhas embutidos na própria aplicação sob teste;
- mecanismos de injeção de falhas instalados fora da aplicação sob teste.

O primeiro caso encontra aplicação em sistemas *stand alone*, onde a inexistência de um sistema multitarefa impede a utilização de um novo programa rodando na mesma máquina. Suas limitações são grandes, na medida em que é

improvável obter independência dos dois processos sob o mesmo programa. A introdução do erro na parte sob teste do código, por consequência, também trará problemas no próprio código injetor e no eventual código detector de anomalias, que, entretanto, pode ser o próprio sistema operacional³³, onde se insere a aplicação.

O segundo caso é o mais usual, com variações. A variedade mais utilizada é a implementação de tarefas individuais para a injeção de falhas, para a detecção de anomalias e para a tarefa sob teste. Preservando-se uma tarefa *scheduler* pode-se garantir uma independência limitada dos processos e obter resultados válidos dentro de um domínio estabelecido.

O maior problema dessa abordagem se origina do fato, como no caso das técnicas de simulação, de que, para a sua implementação, todas as fontes potenciais de problemas devem ser conhecidas. Esta não é uma situação provável, na medida em que se as situações potencialmente indutoras de falhas fossem conhecidas, elas seriam eliminadas através de mecanismos de proteção específicos inseridos no código. Já as técnicas de *hardware* trazem vantagens importantes nesse aspecto.

O trabalho de LOPES (2005) apresenta algumas das principais técnicas de injeção de falhas por *software*, incluindo os conceitos de mutantes ou sabotadores.

4.4 INJEÇÃO DE FALHAS POR *HARDWARE*

As técnicas de injeção de falha por *hardware* pressupõem a interação física entre o mecanismo injetor de falhas e o dispositivo computacional que executa os programas sob teste. Segundo HSUEH *et al.* (1997) *apud* LOPES (2005), elas podem ser classificadas em duas categorias: com contato e sem contato. Na categoria sem contato estão o bombardeamento radioativo ou a interferência eletromagnética, enquanto que nas com contato encontra-se principalmente as interferências chamadas de *pin-level*, que atuam abrindo, curto-circuitando ou injetando sinais interferentes em cada pino individual do dispositivo sob testes.

Há também a possibilidade de introdução de falhas através da aplicação de vibrações mecânicas, embora não se tenha encontrado referências bibliográficas a respeito. A característica mais marcante dessa família de técnicas diz respeito ao

³³ Sistema operacional no sentido de operação dos sistema na execução de sua função.

caráter independente do dispositivo gerador das interações com relação ao dispositivo sob teste, já que as ações de controle efetuadas no mecanismo injetor não guardam qualquer relação com o computador-alvo. Apesar da dificuldade de execução, já que são procedimentos laboratoriais, essas técnicas podem trazer um novo alento à pesquisa, superando algumas dificuldades das técnicas de injeção de falhas por *software*. Técnicas como a radiofrequência induzida ou irradiada e o bombardeio iônico, estão entre estas técnicas. É descrito, a seguir, um conjunto das principais técnicas de injeção de falhas correntemente disponíveis.

4.4.1 PARTÍCULAS CARREGADAS

“Circuitos eletrônicos armazenam informação sob a forma de cargas armazenadas em um nó de circuito ou como uma corrente fluindo entre dois nós de um circuito.” (KARNIK, 2004). Como é de conhecimento comum, a densidade dos circuitos eletrônicos cresce a cada ano, aproximando-se rapidamente do limite físico da matéria. O advento dos circuitos CMOS³⁴ contribuiu enormemente para esse fato e hoje já se produzem circuitos integrados com comprimentos de canal de 20nm, com uma espessura de óxido de 2nm e área da pastilha que se aproxima de 1 μ m² (1mm x 1mm).

Concomitantemente há, no meio ambiente da Terra, uma grande quantidade de partículas atômicas e subatômicas em circulação. Conforme KARNIK (2004), mais de 92% do ambiente da terra consiste de nêutrons, ~4% são píons, ~2% são prótons e ~2% são múons ao nível do mar, gerados, principalmente, pela interação dos raios cósmicos com a atmosfera terrestre.

Essas partículas carregadas têm a capacidade de gerar a ionização direta de dispositivos semicondutores. Nas memórias e dispositivos semicondutores digitais, essa interação física provoca um fenômeno denominado *bit flip*, ainda sem tradução em português. Como o próprio nome indica, uma partícula carregada agindo sobre uma célula semicondutora MOS saturada (característica dos sistemas digitais), tem a tendência de mudar o seu estado, de “0” para “1” ou de “1” para “0”, indistintamente (KARNIK, 2004).

Falhas naturais provenientes desse fenômeno têm sido descritas teoricamente desde 1962 (WALMARK E MARCUS, 1962 *apud* KARNIK, 2004) e

³⁴ CMOS-Complementary metal oxide semiconductor, técnica de construção de dispositivos semicondutores.

falhas reais atribuídas a raios cósmicos tem sido descritas desde 1975 (BINDER *et al.*, 1975 *apud* KARNIK, 2004).

A incidência destas falhas na natureza é esporádica e, devido às necessidades de teste impostas pelos programas espaciais³⁵, surgiu a necessidade de desenvolvimento de alguma estratégia para ensaios acelerados. Como forma de testar esses semicondutores, dispositivos geradores de cargas, muito mais freqüentes do que as naturais, foram desenvolvidos.

Embora estas falhas sejam temporais para níveis baixos de exposição, é possível provocar danos permanentes em semicondutores sob exposição intensa e prolongada, que provoca a destruição da estrutura cristalina. Há tecnologias mais recentes de semicondutores que apresentam maiores graus de imunidade, como os dispositivos SOI-*Silicon On Insulator*, empregados pela IBM desde 2001 ou os MIRT-*Multiple Independent Redundant Transistors*, mas que ainda apresentam aplicações de escopo reduzido.

KARLSSON, *et al.* (1994) e FOLKESSON (2005) ambos *apud* LOPES (2005), SIVENCORONA *et al.* (2003) e FOLKESSON (1999) são exemplos da aplicação do Califórnio Cf^{252} com esta finalidade. Estes dispositivos bombardeiam diretamente as partículas semicondutoras, provocando os *bit flips* descritos anteriormente. A Figura 8 mostra um arranjo típico.

Figura 8 - Uma câmara de vácuo com uma fonte de Califórnio²⁵² colocada sobre um *chip* com o encapsulamento removido.

Fonte: SIVENCORONA *et al.* (2003).

Neste arranjo, a fonte radioativa é instalada no interior de um cilindro de chumbo com uma única saída circular, provida de um obturador mecânico, acionado

³⁵ Os dispositivos expostos ao espaço livre são mais vulneráveis a este tipo de falhas, já que a atmosfera da Terra atenua grande parte das radiações cósmicas presentes no espaço interplanetário.

por um solenóide comandado remotamente pelo controle do experimento. A fonte é colocada diretamente sobre o *chip*, que tem o seu encapsulamento removido para facilitar o bombardeamento.

Uma micro câmara de vácuo é então acoplada sobre o *chip*, provendo conexões elétricas vedadas para os sinais do *chip* e para o comando do obturador. O vácuo visa remover as interações das partículas carregadas com as moléculas do ar, o que retiraria energia das partículas radioativas destinadas a provocar falhas no *chip*.

Com a introdução das técnicas de injeção de falhas, esses dispositivos apareceram como candidatos naturais a introdutores de falhas em semicondutores microprocessados que executam aplicações críticas, onde se deseja verificar a robustez das estruturas de código.

FOLKESSON (1999) empregou uma fonte comercial de Califórnio Cf^{252} , com uma atividade nominal de 15 kBq. Neste caso, perto de 3% de todas as desintegrações geram fragmentos de fissão (íons pesados) e o restante delas gera apenas partículas alfa (α).

As partículas α não afetam a maioria dos circuitos porque elas depositam muito menos cargas no silício do que os fragmentos da fissão (uma transferência linear de energia de aproximadamente $1.6 \text{ MeV mg}^{-1}\text{cm}^2$ versus $41\text{-}45 \text{ MeV mg}^{-1}\text{cm}^2$ para os fragmentos de fissão (GUNNEFLO *et al.*, 1987 *apud* FOLKESSON, 1999).

Este procedimento aparece como ideal para os objetivos do experimento proposto nesta tese, já que é capaz de gerar SEUs—*Single Upset Events* uniformemente distribuídos, contribuindo para a validade estatística da comparação entre as técnicas de programação defensiva propostas no experimento apresentado no capítulo 5.

A grande dificuldade advém do fato de que é necessário manipular elementos radioativos, câmaras de vácuo e *chips* sem encapsulamento para fazer uso destas técnicas, o que dificulta sobremaneira a sua execução.

Além disto, o Califórnio é raro, e de difícil obtenção, como mostrado na Figura 9. O Califórnio, elemento 98, foi produzido e identificado pela primeira vez por Thompson, Street Jr., Ghiorso e Seaborg na Universidade de Berkeley, Estados Unidos, em Fevereiro de 1950 (SEABORG, 1969).

Figura 9 - O processo de formação do Califórnio 252.

Fonte: SEABORG (1969).

No lado esquerdo da Figura 9 percebe-se que, a partir do Plutônio 239, que já é um elemento controlado, de custo alto e de difícil obtenção, apenas 0,3% podem se transformar em Califórnio 252. Do lado direito, observa-se que para formar 10mg de Califórnio seriam necessários dois anos de irradiação contínua num fluxo neutrônico de 10^{15} nêutrons/cm². SEABORG (1969), mostra que grandes reatores adequados para este fim, como MTR-*Materials Test Reactor* da Universidade de Idaho, nos Estados Unidos, conseguiam um fluxo de apenas $3 \cdot 10^{14}$ nêutrons/cm². Ao mesmo tempo, FOLKESSON (1999) indica problemas importantes com a qualidade da fonte de Califórnio 252 empregada por ele em seus estudos de técnicas de injeção de falhas, que prejudicaram em muito a comparação pretendida em seu trabalho. Isto se deve, provavelmente, ao fato de que no processo de produção do Califórnio 252 aparece naturalmente o Califórnio 254 que tem meia vida de apenas 61 dias e é muito difícil de manusear, já que sofre fissão espontânea, emitindo nêutrons neste processo, como indicado por SEABORG (1969).

Apesar de todas estas dificuldades, a habilidade que as fontes de Califórnio têm de gerar SEUs em dispositivos semicondutores não pode ser desprezada. A maioria dos SEUs acontece em memórias MOS, mas também podem ocorrer em portas lógicas. Grande parte das dificuldades experimentais, que serão apresentadas neste trabalho, são fruto da baixa controlabilidade, ou melhor, da criticidade do controle do método eletromagnético proposto na geração dos SEUs necessários a comprovação da eficácia das técnicas de programação defensiva.

Apesar de relativamente bem conhecidas, é provável que estas técnicas (bombardeio por partículas carregadas e EMI) possam se beneficiar de estudos voltados exclusivamente ao desenvolvimento operacional de seu processo experimental.

Um trabalho muito recente de LHO E KIM (2007) mostra o uso de prótons, provenientes de um acelerador de partículas, com a mesma finalidade de provocar SEUs. É interessante notar que ele emprega o mesmo processador de nosso experimento, o ATMEGA128 da Atmel, para supervisionar o seu experimento. Algumas das taxas de introdução de falhas obtidas por ele estão exemplificadas na Figura 10.

Figura 10 - Taxa de falhas pelo tempo com bombardeio de prótons.

Fonte: LHO E KIM (2007).

Estas taxas, da ordem de 8 por minuto, são perfeitamente adequadas para a realização do experimento de comparação de técnicas de programação defensiva propostas neste trabalho.

O Instituto de Física da Universidade de São Paulo, da qual também faz parte a Escola Politécnica a que pertence o autor, possui um acelerador de partículas do tipo Pelletron, com um potencial de 8MV. Este acelerador é provavelmente capaz de gerar partículas aceleradas com energia apropriada para provocar os *bit flips*, entretanto, os experimentos de LHO E KIM (2007) foram realizados com uma energia de 30MeV o que pressupõe um tempo de exposição maior para alcançar o mesmo número de SEUs. Foi considerada a possibilidade de seu emprego em um projeto conjunto entre a Escola Politécnica e o Instituto de Física. A idéia, entretanto, foi descartada devido a duas causas básicas: o elevado tempo de *set up* da máquina e o grande número de pesquisadores e projetos de pesquisa que disputam seu uso.

Entretanto, acreditamos que é válido experimentar com este acelerador, o que poderia ser feito em um trabalho futuro

4.4.2 RADIOFREQUÊNCIA

A expectativa desta tese é de que a radiofrequência induzida possa provocar efeitos ou SEUs muito similares aos dos íons pesados discutidos anteriormente dentro de um ambiente de testes mais facilmente manipulável. Embora a natureza física seja diferente, há a expectativa de que a radiofrequência irradiada possa provocar um *bit flip* através da indução de uma tensão superficial na pastilha semicondutora. Verificar esta alternativa foi uma premissa deste trabalho, já que a obtenção das condições técnicas adequadas para o uso de radiofrequência se mostra muito mais factível do que para o análogo radioativo.

Considerando as dificuldades inerentes à pesquisa experimental, procurou-se trabalhar com equipamentos de fácil obtenção, de maneira a que os experimentos sejam facilmente reproduzíveis. Desta maneira, a experiência do autor como físico e radioamador contribuíram para tornar realidade este experimento, como será descrito no próximo capítulo. De maneira análoga, esta tese procura ilustrar e documentar com detalhes os procedimentos, equipamentos e materiais utilizados, de maneira a incentivar a reprodutibilidade e a proliferação de novos grupos.

Há, entretanto, uma alternativa a esta técnica da radiofrequência, que consiste na geração de *bursts* de alta energia, como os sugeridos pela norma IEC 801-4 (CEI/IEC), ou seja, trens de pulsos com uma duração de 15 ms, um período de 300 ms, uma frequência de 1.25, 2.5, 5 ou 10 kHz, e níveis de tensão entre 225 V e 4.400 V. Estes pulsos tendem a ser similares aos provocados pelo chaveamento de cargas indutivas por relês e chaves.

A Figura 11 ilustra duas formas de acoplamento destes pulsos ao DUT – *Dispositive Under Test*. No arranjo da esquerda, alternativa 1 da figura, o DUT é colocado no interior de duas placas paralelas. Há pelo menos duas variações para a conexão. Na primeira, as duas placas tem o mesmo potencial e a interferência encontra seu caminho entre as placas e o terra da alimentação dos circuitos, que deve ser o mesmo do gerador de interferências; outra possibilidade é colocar uma placa no terra do gerador de interferências e a outra no sinal interferente.

No arranjo da direita, alternativa 2, uma bobina conectada ao sinal interferente e seu terra é colocada diretamente sobre o DUT (no caso um circuito integrado). Ambos os arranjos foram experimentados neste projeto desta tese.

Figura 11 - Injeção de falhas por *bursts* de alta energia.

Fonte: FOLKESSON (1999).

Embora o uso destas técnicas em baixa frequência apareça em trabalhos como o de LOPES (2005), com bons resultados, elas não foram experimentadas nesse trabalho de pesquisa.

4.4.2.1 RADIOFREQUÊNCIA IRRADIADA

Emissões de radiofrequência apresentam a característica de induzir tensões em superfícies condutoras através do estabelecimento de correntes de Foucault (*eddy currents*).

Como as informações em memórias semicondutoras estão armazenadas como cargas elétricas, a submissão à radiofrequência apresenta propriedades semelhantes ao bombardeio iônico descrito anteriormente, sem, entretanto, apresentar as desvantagens advindas do manuseio de elementos radioativos.

LOPES (2005) afirma que “Em consequência do dramático aumento da densidade de integração associado à redução progressiva da tensão de alimentação, sistemas eletrônicos estão cada vez mais susceptíveis à ocorrência de falhas transientes.”, o que aponta para dois fatos importantes e antagônicos. A redução da tensão de alimentação dos dispositivos microprocessados típicos aumenta a probabilidade de sucesso na injeção de falhas por meios eletromagnéticos, entretanto, a mesma constatação aponta para uma fragilidade adicional destes dispositivos e para a necessidade de investigação e desenvolvimento de técnicas de mitigação adequadas

A desvantagem da radiação eletromagnética frente ao bombardeio radioativo aparece em função da dispersão. É mais difícil, devido às características da distribuição dos campos elétricos e magnéticos dos sistemas irradiantes, garantir

uma distribuição uniforme, e, portanto, estatisticamente significativa, da interferência provocada.

4.4.2.2 RADIOFREQUÊNCIA CONDUZIDA

A técnica de injeção de falhas por radiofrequência conduzida consiste, em geral, na aplicação direta (galvânica) da radiofrequência no circuito sob teste.

No caso desta tese, por exemplo, esta técnica foi empregada e abandonada por não apresentar os resultados esperados devido a dificuldades experimentais. Como para ter um espaço de validade estatístico adequado desejávamos uma distribuição uniforme das falhas na pastilha, optamos por injetar a radiofrequência modulando a alimentação de 5V. Isto provoca múltiplas falhas simultâneas na pastilha, mesmo para níveis muito baixos de potência, dificultando o diagnóstico do comportamento do sistema.

4.5 ESCOLHA DA TÉCNICA DE INJEÇÃO DE FALHAS PARA O EXPERIMENTO

Os critérios para a escolha da técnica de injeção do experimento foram:

1. Representatividade estatística: capacidade de fornecer um grande número de dados em um tempo razoável de experimentação, de maneira a validar estatisticamente as medidas realizadas;
2. Baixa probabilidade de vínculos ou vícios: capacidade do processo de injeção de ser independente do DUT ou das aplicações ou programas sendo executadas em seu interior;
3. Possibilidade física de execução: disponibilidade dos equipamentos e dispositivos necessários.

Seguindo estes critérios, optou-se por uma técnica de *hardware*, dadas as considerações da Tabela 11. Apesar das desvantagens descritas anteriormente, o procedimento da radiofrequência irradiada foi a técnica eleita para o experimento desta tese. A limitação que decorre desta escolha aparece na disponibilidade e no investimento necessários à obtenção do instrumental adequado. A forma encontrada foi fazer uso de equipamentos desenvolvidos originalmente para radioamadores, adaptando-os quando necessário. Foi concebido assim um sistema de injeção de falhas por radiofrequência que utiliza exclusivamente equipamentos de radiocomunicação de custo razoável.

Esta alternativa foi a possível diante das limitações, mas trouxe para esta tese um trabalho de pesquisa e desenvolvimento em instrumentação não esperado para o projeto.

É importante notar que o ajuste adequado dos sistemas irradiantes, potências e frequências empregados é fundamental para o sucesso do experimento realizado. Diferentemente dos métodos que fazem uso de partículas pesadas, que são pontuais, os métodos de radiação agem pela indução de tensões nos condutores e semicondutores do circuito integrado. São desta forma dependentes da geometria e da existência de materiais e estruturas diferentes no interior destes componentes. Isto poderia provocar uma distribuição privilegiada de falhas ligadas aos pinos ou a longas faixas de alumínio, por exemplo, viciando os testes. Desta maneira, parece provável que as potências e frequências ideais para provocar eventos únicos em pastilhas sejam dependentes de seu modelo físico, o que ainda está por ser investigado em futuras pesquisas.

5 ARRANJO EXPERIMENTAL

Conforme descrito anteriormente, o objetivo proposto por esta tese é o desenvolvimento de um método prático para avaliar quantitativamente o impacto da programação defensiva na segurança dos sistemas críticos. Assim, considerando-se o caráter empírico representado por um método experimental, entende-se que essa se justifica pela utilidade que possa ter para o desenvolvimento da ciência e da tecnologia. A expectativa é de que a condução de vários experimentos possam levar à construção de um modelo teórico adequado. Com esta abordagem pragmática, espera-se que outros trabalhos empíricos sejam empreendidos ou desenvolvidos, até que se forme uma massa crítica capaz de permitir uma abordagem puramente analítica para o problema maior que se apresenta: quais técnicas conhecidas de programação defensiva podem beneficiar o nível de integridade de segurança de um sistema complexo em funcionamento e em que intensidade. Este é, na essência, o motivo pelo qual se incentiva a reprodução e o aperfeiçoamento do experimento aqui apresentado.

Partindo desse raciocínio, optou-se pelo desenvolvimento de uma plataforma de testes simplificada que pudesse servir para a experimentação dos conceitos desta tese, mas que fosse aproveitável como base para outros trabalhos posteriores do mesmo grupo de pesquisa, o GAS - Grupo de Análise de Segurança do PCS – Departamento de Engenharia da Computação e Sistemas Digitais da Escola Politécnica da Universidade de São Paulo, que estão sugeridos no capítulo 7. A premissa adotada nesta tese foi a de se utilizar *software* livre, sempre que possível, e tentar manter os custos baixos. Um dos diferenciais importantes para o desenvolvimento da pesquisa aplicada em países como o Brasil está relacionado aos custos.

O experimento proposto consta de um *hardware* dedicado, simulando a execução de uma aplicação crítica que, para o estudo de caso proposto, é um simulador de um sistema de sinalização ferroviária, descrito no item 5.3. Esse *hardware* se baseou em uma placa de desenvolvimento de microcontroladores da empresa Atmel, facilmente disponível no mercado nacional. A placa executa uma aplicação de teste, simulando um controlador de movimentação de trens real, que pode ser configurada através da inclusão ou remoção das diversas técnicas de programação defensiva. A introdução seletiva das técnicas de programação defensiva permite a análise comparada proposta.

A plataforma de testes é composta por uma unidade injetora de falhas, 2 PCs denominados 1 e 2, uma placa de desenvolvimento AVR Total Flex, uma unidade de programação ISP e uma placa de entradas e saídas com barramento EISA, além dos cabos e dos *softwares* associados.

A parte mais elaborada do *software*, que está discutida no item 5.3, é executada no PC1 e tem três funções básicas:

- um simulador, que gera as entradas e aceita as saídas que, em um sistema real, seriam disponibilizadas pelas instalações em campo;
- uma Interface Homem Máquina (IHM) que permite a emissão de comandos e a supervisão visual do sistema;
- uma aplicação de verificação, que monitora o comportamento do sistema em busca de inconsistências e erros. Esta é a aplicação que controla o teste.

A Figura 12 ilustra o arranjo geral do experimento (estação de testes):

Figura 12 - Arranjo básico do *hardware* do experimento (Estação de testes).

O dispositivo sob teste é a placa AVR Total Flex, executando a aplicação que inclui as técnicas de programação defensivas.

Funcionalmente, a estação de testes se comporta da seguinte maneira:

- À semelhança de um sistema de controle de trens real, a placa AVR Total Flex tem a função de controladora do sistema de segurança. Nela reside um *software* que contém equações booleanas com todas as regras de movimentação de trens e dispositivos de mudança de vias³⁶. A segurança do sistema depende, intrinsecamente, do funcionamento adequado do conjunto

³⁶ Máquinas de chave, dispositivos usualmente eletromecânicos ou eletrohidráulicos responsáveis pela movimentação dos trilhos nas regiões em que os trens podem passar de uma via a outra.

de regras que está contido neste *software*. Os requisitos de segurança especificados estão codificados neste dispositivo, que está denominado como AVR nesta tese.

- O PC1 está conectado ao microcontrolador AVR através de sinais discretos (54 sinais), via a interface de entradas e saídas digitais padrão Quatech PXB-721³⁷. Através dessas linhas o simulador do PC1 recebe e transmite indicações e comandos de/para o AVR, simulando o mundo real. Destes sinais, 53 estão conectados aos *ports* de entrada/saída do microcontrolador e 1 está ligado diretamente ao *reset* do mesmo;
- A aplicação IHM simula a interface do operador do sistema real e permite a interação com o sistema de maneira análoga a um sistema real;
- A partir deste funcionamento normal introduzem-se falhas no sistema através da irradiação de Radiofrequência - RF, produzida pelo injetor de falhas e disparada pelo PC1 através de um sinal dedicado da interface de entradas e saídas digitais, padrão Quatech PXB-721, que sinaliza o *status* de início de testes. Inicialmente este sinal era acionado por um pequeno programa em Linguagem C no PC1 e o sinal acionava diretamente o gerador de RF através de uma pequena interface dedicada. Com a constatação de interferência entre os processos do PC1 (um prejudicava o desempenho do outro no ambiente multitarefa), este pequeno aplicativo foi transferido para o PC2, que através de uma interface USB lê o sinal dedicado do PC1 e produz o padrão de testes necessário, detalhado a seguir³⁸;
- A falha é detectada pela aplicação de verificação, executada no PC1, que registra o tempo entre o início da amostra de teste e a constatação de uma falha de tipo insegura. Esta seqüência é continuamente repetida até que se obtenha um conjunto de dados grande o suficiente para ser estatisticamente significativo;

³⁷ Placa padrão *Paralell Expansion Board*, tipo Quatech PXB-721, http://www.quatronix-cn.com/pdf/acq/PXB_721.pdf

³⁸ Esta pequena modificação ilustra os problemas encontrados na pesquisa experimental. O compilador utilizado no PC1 (Linux) não conseguia incluir as bibliotecas da placa USB necessárias para uso no PC2 (Windows), o que obrigou portar a aplicação para outro compilador apto para a tarefa (o interpretador Ch).

- Os registros temporais obtidos são comparados e tabulados gerando a saída dos resultados para a análise quantitativa comparada investigada por esta tese.

O programador ISP, a direita na Figura 12, é por onde se carrega o programa do microcontrolador. O ambiente de desenvolvimento do *software* do microcontrolador está instalado no PC2, razão pela qual o ISP está conectado a esta máquina. O ISP está detalhado em outras partes dessa tese.

Os dois itens seguintes descrevem com mais detalhes as características de cada um destes componentes.

5.1 O *HARDWARE* DE TESTES

O sistema escolhido é dedicado e simples, não redundante e sem sistemas defensivos implementados por *hardware*, como *watchdogs*³⁹ por exemplo. O motivo desta posição é evitar - já que se busca validar quantitativamente as técnicas de programação defensiva - que os efeitos das falhas injetadas sejam mascarados por um dispositivo de *hardware* que atuasse prioritariamente.

5.1.1 A ESTAÇÃO DE TESTES

Esta estação está baseada em um computador tipo PC, equipado com uma placa de entradas e saídas digitais. As características técnicas desta máquina não são, em absoluto, críticas para a aplicação desenhada. O objetivo básico da escolha foi a disponibilidade de um *slot* disponível com barramento EISA, já que havia a disponibilidade de uma placa de entradas e saídas, já existente e nesse padrão.

A CPU foi carregada com o sistema operacional Linux, em sua distribuição Ubuntu 5.10 e está denominada como PC1 na Figura 12. Essa escolha foi efetuada em função da familiaridade com a plataforma de desenvolvimento Java sobre este sistema operacional. Inicialmente foi considerada, e efetivamente empregada nos testes iniciais, uma máquina de 2 processadores Pentium II de 267MHz, com a intenção de executar a aplicação de verificação em um segundo processador, mostrada na Figura 12, de modo a conferir um grau de independência maior para este processo. Essa alternativa, entretanto, foi descartada em função da dificuldade de implementar modificações que seriam necessárias no *kernel* do sistema

³⁹ Dispositivos tipo *timetick*, que rearmam ou geram interrupções de alto nível na ausência de sinais de correto funcionamento do *software*.

operacional. Uma máquina COMPAQ DESKPRO bastante comum, com o *slot* EISA necessário, foi empregada. Esta máquina está equipada com um processador Pentium II, com 512 kbytes de *cache*, ilustrada na Figura 13.

Figura 13 - Vista interna do computador empregado para o PC1, mostrando a placa de E/S instalada.

Originalmente ela contava com 128Mbytes de memória RAM que foram ampliados para 576Mbytes como tentativa, infrutífera, de resolver uma interferência entre a tarefa geradora do padrão de falhas e as aplicações do simulador, de detecção de falhas e da IHM do PC1. Este comprometimento mostrou que a intenção de utilizar processadores duplicados era realmente fundamentada. Com a impossibilidade de utilizar a máquina com dois processadores, a tarefa de injeção de falhas foi transferida para o PC2.

5.1.1.1 O CARTÃO DE ENTRADAS E SAÍDAS DIGITAIS

Havia a disponibilidade de uma placa de 72 entradas ou saídas digitais bastante antiga na empresa em que trabalha o autor (Metrô de São Paulo, que se disponibilizou a cedê-la para o projeto). Essa placa foi escolhida por sua implantação muito simples, baseada em 3 controladores 8255 da INTEL⁴⁰. Esses antigos dispositivos possuem 24 entradas bidirecionais cada, agrupadas em 3 grupos de 8 linhas A, B e C. Os grupos A e B devem ser configurados totalmente como entradas ou saídas. O grupo C pode ser partilhado em 2, sendo permitido que se configure 4 sinais como entradas e quatro como saídas. A placa possui ainda algumas portas lógicas e uma chave *dip* que permite configurar o *range* de endereçamento a ser obedecido pela placa.

⁴⁰ Fabricante de semicondutores.

A sua concepção simples faz com que não haja nem *firmware* nem *drivers* específicos. A placa pode ser acessada a partir de qualquer sistema de programação, simplesmente através da escrita ou leitura de seus *ports* de E/S.

Essa é uma vantagem tangível, já que não se desejava um sistema sob o qual não se tivesse o controle direto. O objetivo, neste caso, é obter um sistema cujas falhas induzidas fossem seguramente devidas ao sistema de injeção de falhas adotado, e não por um mal funcionamento, cujo mecanismo não estivesse acessível e documentado.

A sua única desvantagem era o barramento EISA, pouco freqüente em PCs modernos, o que foi solucionado através da escolha de uma estação de trabalho disponível na Universidade, mostrada na Figura 13. A Figura 14 ilustra a disposição física da placa:

Figura 14 - A placa de E/S PXB-721 da Qualitech.

Fonte: *site* Qualitech⁴¹.

A Figura 15 mostra o diagrama de blocos dos dispositivos da placa. A empresa Qualitech⁴¹, fabricante da placa, descontinuou o seu fornecimento com barramento EISA. Entretanto, esta placa continua disponível, em versão idêntica para barramento PCI, denominada PXB-721/PCI através de sua subsidiária na China³⁷.

Seu circuito é constituído por 3 dos controladores de entrada e saída do tipo 8255 da INTEL discutidos anteriormente. O arranjo dos *ports* destes componentes, formado por três *ports* de entrada e saída, com uma capacidade total de 24 *bits*, está mostrado na Figura 15, fazendo uso de todas as facilidades oferecidas pelo *chip*. Em termos de endereçamento, a placa se relaciona com o barramento EISA em dois

⁴¹ <http://www.qualitech.com>

grupos de 12 *bits*, A e B. O grupo A é formado pelo *port* A e pela metade superior do *port* C, enquanto o grupo B é formado pelo *port* B e pela metade inferior do *port* C.

O restante dos circuitos da placa realiza a interface com o barramento EISA, e inclui um *buffer* bidirecional para os dados.

Figura 15 - Características funcionais dos 8255 da placa de E/S.

Fonte: *data sheet* site INTEL⁴².

5.1.2 A PLACA-ALVO DE TESTES

A placa empregada é de fabricação nacional, produzida pela empresa Symphony, modelo AVR Total Flex e está ilustrada na Figura 16. A Interface AVR Total Flex é uma placa para desenvolvimento e treinamento nos microcontroladores da família AVR (*RISC-Reduced Instruction Set Computer*) de alto desempenho. Foi concebida para permitir o desenvolvimento e o teste de sistemas de forma prática e rápida, permitindo alto desempenho, com aplicativos totalmente compilados.

⁴² <http://www.intel.com>

- 90S2313 - 10 MHz (módulo *Fast BASIC* 2k);
- Atmega 8 - 16 MHz (módulo *Fast BASIC* 8k);
- *Chips* auxiliares;
- LCD gráfico 128x64;
- LCD gráfico 240x128;
- 2 canais RS-232 Módulo *Fast BASIC* 2k e 8k Módulo RF (rádio modem);
- Módulo servos/sensores Módulo potência L-293 Soquete PLCC;
- Módulo Atmega 128;
- Módulo *Proto Board* LCD 16 x 2 ou gráfico.

Essa placa, conforme citado, suporta o emprego de diversos microcontroladores, mas devido à configuração necessária à plataforma de testes, foi escolhido o processador ATMEGA128, que vai acoplado a essa placa através de um adaptador especial, conforme ilustrado na Figura 17. A escolha desse conjunto placa de desenvolvimento/adaptador do ATMEGA128 foi direcionada pela geometria do conjunto, que favorece a introdução de antenas e bobinas sob ou sobre o *chip* (no experimento antenas foram efetivamente colocadas sobre o *chip*, em diversos arranjos e uma bobina foi experimentada sob o *chip*, validando esta escolha). A Figura 16, com o processador desencaixado, e a Figura 17, com ele no lugar, ilustram essa possibilidade.

Figura 17 - Adaptador ATMEGA128 da AVR Total Flex.

Fonte: site Symphony⁴³.

O processador ATMEGA128 foi escolhido em função da grande disponibilidade de *ports* de entradas e saídas digitais disponibilizadas por esta pastilha. Seria possível estabelecer uma comunicação serial entre o dispositivo sob

⁴³ <http://www.symphony.com.br>

testes e o PC1, entretanto, os dispositivos seriais teriam a tendência de falhar antes dos dispositivos sob testes, inviabilizando a detecção de quem falhou: o dispositivo sob testes ou o dispositivo de comunicação que conduz a informação do dispositivo. A razão para isto se deve ao fato de que, nos padrões seriais, a informação de cada um dos *bits* passa pelo mesmo *hardware*. Por decorrência, uma falha simples neste componente indisponibiliza 100% do sistema.

Esse *chip*, ATMEGA128, possui uma grande versatilidade e é totalmente auto-suficiente, permitindo incluir a pastilha como um todo no ambiente de bombardeio por radiofrequência induzida. São necessários externamente apenas um cristal e dois pequenos capacitores para o estabelecimento do relógio principal.

A Figura 18 ilustra o diagrama de blocos do processador ATMEGA128 escolhido, com as suas características principais.

Nas bordas superior e inferior observam-se os blocos *port drivers*, que são os *ports* bidirecionais do microcontrolador, único acesso ao mundo exterior. É importante salientar o caráter multifuncional de cada um dos terminais externos do microcontrolador. Como exemplo, os pinos do *Port F* podem ser tanto entradas ou saídas digitais convencionais ou entradas analógicas, em função da programação do *chip*. Esta característica trouxe problemas na fase de desenvolvimento dos *softwares*. Como os *bits* de programação conectados a interface ISP também são entradas/saídas digitais, e todas as entradas/saídas disponíveis são empregadas neste projeto, se o programa simulador do campo do PC1 for executado simultaneamente a uma tentativa de programar o *chip*, este acabará destruído.

As setas largas, em formato de I na Figura 18, representam o barramento interno do microcontrolador. A este barramento estão conectados:

- Os registradores de dados de todos os *ports* (numerados de A até G);
- Os diversos tipos de memórias (*RAMs*, *flash*, *EPROMs*, etc.);
- Os conversores análogo digitais;
- Os controladores seriais;
- Os mecanismos de controle por *hardware*, como cão de guarda, geradores e verificadores de paridade;
- Relógios de tempo real, controladores de interrupção, etc.

Figura 18 - O processador ATMEGA128 e a sua capacidade de I/O.

Fonte: *datasheet* site Atmel: http://www.atmel.com/dyn/products/product_card.asp?part_id=2018

O ATmega128 é um microcontrolador CMOS de baixa potência de consumo, de 8-bits e baseado na avançada arquitetura AVR RISC. Através da execução de instruções poderosas em ciclo único de *clock* o ATmega128 alcança índices de desempenho de 1 MIPS-*Mega Instructions Per Second* por MHz, permitindo a otimização da velocidade de processamento em função do consumo de energia. É importante notar que, para o projeto desta tese, são empregados apenas as memórias Flash e SRAM e os *ports* de entrada e saída. Todos os outros periféricos internos ao microcontrolador são mantidos desativados.

Para a sua programação a placa AVR Total Flex utiliza uma tecnologia desenvolvida pela empresa Atmel denominada *In System Program* (ISP) em seu

padrão STK200/300, e para isto faz uso de um adaptador denominado Gravador AVR ISP da Symphony. Figura 19.

Figura 19 - A placa do gravador AVR ISP da Symphony.

Fonte: *site* Symphony⁴³.

A sua concepção é antiga, já que os gravadores atuais se conectam aos PCs através de interfaces USB, entretanto ela é funcional e facilmente reproduzível com componentes comuns. A Figura 20 ilustra o seu diagrama esquemático. O conector denominado *target circuit* está ligado aos sinais homônimos do microcontrolador indicados na Figura 18.

Figura 20 - A interface de programação STK300 da Atmel.

Fonte: *site* LancOS⁴⁴.

Esta interface se constitui quase que exclusivamente por um 74HC244 que é um *buffer tri state* de 8 vias, com a única função de adaptar os níveis de tensão do microcontrolador aos do *port* paralelo Centronics e vice versa.

⁴⁴ <http://www.lancos.com>

5.1.3 A COMUNICAÇÃO ENTRE OS MÓDULOS

A Figura 12, no início do capítulo, mostra o diagrama de blocos geral da plataforma de testes. Há, entretanto, 54 sinais de comunicação de *hardware* entre a placa AVR Total Flex e a placa de E/S digitais instalada no PC1. Destes, 53 são referentes a *ports* de E/S do ATMEGA128, configurados adequadamente para a posição de entrada ou saída, necessários ao arranjo experimental, e 1 sinal gerado pela placa de E/S e conectado a linha de *reset* do ATMEGA128, que tem a função de reinicializar o processador a cada seqüência de testes, retomando o programa do início após a detecção de uma falha. A Figura 21 ilustra o processo de comunicação entre os diversos módulos, com os sinais codificados no formato *axb*, onde *a* é o número de informações (quantidade de rotas, de ocupações, etc.) e *b* é o número de *bits* de que essa informação é composta:

Figura 21 - O diagrama de comunicação entre os módulos.

Os números indicados na figura significam [número de informações]x[número de *bits* da informação]. Por exemplo, as ocupações dos circuitos de via são dadas por 12 informações de 1 *bit* cada, já que temos 12 circuitos de via (trechos de via) e a única informação necessária é se o circuito está ocupado (valor lógico 1 ou TRUE) ou desocupado (valor lógico 0 ou FALSE). As mesmas informações utilizadas pelo simulador estão também disponíveis para a aplicação de verificação, que é

executada na mesma máquina, PC1. A comunicação entre a IHM e o simulador não tem indicações de sinais físicos, já que as duas aplicações são executadas na máquina PC1 e as informações são representadas por sinais lógicos.

Do ponto de vista da placa de E/S os sinais de interfaceamento foram mapeados pelos seus três ports H1, H2 e H3 da maneira mais conveniente, já que havia sobra de capacidade da placa. Foram utilizados 54 sinais e a placa é capaz de controlar 72 sinais programados para entrada ou saída. A Tabela 12 resume os sinais do ponto de vista da placa de E/S que estão detalhados na Tabela 13. Nesta tabela, os valores de A1 a F1 e A2 a F2 referem-se aos 6 circuitos de via das duas vias concêntricas, 1 e 2 respectivamente.

Tabela 12 - Resumo dos sinais do ponto de vista da placa de E/S.

Saída	Descrição	Endereço	Entrada	Descrição	Endereço
12	Ocupações	0x302 = {0,1,2,3,4,5,6,7} 0x306 = {0,1,2,3}	24	Códigos de velocidade	0x300 = {0,1,2,3,4,5,6,7} 0x301 = {0,1,2,3,4,5,6,7} 0x304 = {0,1,2,3,4,5,6,7}
4	Comandos de rota	0x306 = {4,5,6,7}	6	Status da chave	0x305 = {0,1,2,3,4,5}
4	Posição das chaves	0x30A = {0,1,2,3}	2	Comando da chave	0x305 = {6,7}
1	Comando local	0x30A = {4}			
21	↵Total de <i>bits</i> de saída		32	↵Total de <i>bits</i> de entrada	

O número de informações empregadas nesta configuração é bastante reduzido, devido a limitação imposta pelo número de *ports* disponíveis no microcontrolador e pela decisão de não empregar controladores de entrada e saída externos ao microcontrolador.

A consequência esta decisão pode ser observada tanto na redução do número de informações empregadas no simulador – no simulador não há sinaleiros, por exemplo – como na modificação do padrão das informações existentes no simulador em relação a forma com que estas informações aparecem em sistemas reais. Só há, por exemplo, 4 códigos de velocidade. O Metrô de São Paulo tem oito. Isto se deve ao fato de que só foi possível disponibilizar 2 *bits* para a codificação dos códigos de velocidade dos circuitos de via.

Conforme explicitado anteriormente, o caráter multifuncional dos pinos dos microcontroladores modernos aumenta em muito a possibilidade de que erros operacionais sejam cometidos, o que aconteceu algumas vezes no experimento

proposto por essa tese. Por decorrência alguns *chips* pararam de funcionar ao longos dos testes, o que demandou um grande esforço para adquiri-los e substituí-los.

Tabela 13 - Detalhamento dos sinais de interface do ponto de vista da placa de E/S.

Saídas		Entradas	
A1:ocupação	0X302[0]	A1:codigo_velocidade (LSB)	0x300[0]
B1:ocupação	0X302[1]	A1:codigo_velocidade (MSB)	0x300[1]
C1:ocupação	0X302[2]	B1:codigo_velocidade (LSB)	0x300[2]
D1:ocupação	0X302[3]	B1:codigo_velocidade (MSB)	0x300[3]
E1:ocupação	0X302[4]	C1:codigo_velocidade (LSB)	0x300[4]
F1:ocupação	0X302[5]	C1:codigo_velocidade (MSB)	0x300[5]
A2:ocupação	0X302[6]	D1:codigo_velocidade (LSB)	0x300[6]
B2:ocupação	0X302[7]	D1:codigo_velocidade (MSB)	0x300[7]
C2:ocupação	0X306[0]	E1:codigo_velocidade (LSB)	0x301[0]
D2:ocupação	0X306[1]	E1:codigo_velocidade (MSB)	0x301[1]
E2:ocupação	0X306[2]	F1:codigo_velocidade (LSB)	0x301[2]
F2:ocupação	0X306[3]	F1:codigo_velocidade (MSB)	0x301[3]
X01_rota:comando (LSB)	0X306[4]	A2:codigo_velocidade (LSB)	0x301[4]
X01_rota:comando (MSB)	0x306[5]	A2:codigo_velocidade (MSB)	0x301[5]
X02_rota:comando (LSB)	0X306[6]	B2:codigo_velocidade (LSB)	0x301[6]
X02_rota:comando (MSB)	0x306[7]	B2:codigo_velocidade (MSB)	0x301[7]
X01_chave:posição (LSB)	0x30A[0]	C2:codigo_velocidade (LSB)	0x304[0]
X01_chave:posição (MSB)	0x30A[1]	C2:codigo_velocidade (MSB)	0x304[1]
X02_chave:posição (LSB)	0x30A[2]	D2:codigo_velocidade (LSB)	0x304[2]
X02_chave:posição (MSB)	0x30A[3]	D2:codigo_velocidade (MSB)	0x304[3]
Modo:[local],[auto]	0x30A[4]	E2:codigo_velocidade (LSB)	0x304[4]
		E2:codigo_velocidade (MSB)	0x304[5]
		F2:codigo_velocidade (LSB)	0x304[6]
		F2:codigo_velocidade (MSB)	0x304[7]
		X01_chave:status (bit0)	0x305[0]
		X01_chave:status (bit1)	0x305[1]
		X01_chave:status (bit2)	0x305[2]
		X02_chave:status (bit0)	0x305[3]
		X02_chave:status (bit1)	0x305[4]
		X02_chave:status (bit2)	0x305[5]
		X01_chave:comando	0x305[6]
		X02_chave:comando	0x305[7]

5.2 INJETOR DE FALHAS

Conforme descrito anteriormente, o dispositivo injetor de falhas foi montado com base em equipamentos desenvolvidos originalmente para uso por radioamadores. As excelentes especificações técnicas dos equipamentos utilizados, de propriedade do autor, justificaram a escolha e viabilizaram o projeto.

O fato do autor desta tese ser radioamador habilitado no Brasil pela ANATEL – Agência Nacional de Telecomunicações, com o prefixo de PU2YIG, autoriza o uso experimental e controlado das frequências empregadas, em conformidade com a legislação brasileira⁴⁵.

⁴⁵ http://www.anatel.gov.br/index.asp?link=/biblioteca/Decretos/dec_91836.htm?Cod=775, Decreto nº 91.836 de 24 de outubro de 1985.

Foram experimentados dois arranjos, um operando na faixa de VHF, que se mostrou inadequado conforme será descrito adiante, e outro, efetivo, operando em HF.

Ambos os arranjos aparecem ilustrados nos diagramas de blocos apresentados na Figura 22.

Figura 22- Diagrama de blocos dos arranjos de injeção de falhas por RF.

No arranjo superior a potência pode ser variada continuamente (0 à 8 ou 0 à 35W, colocando ou retirando o amplificador) e a frequência pode ser variada de 116 à 146MHz. O arranjo inferior permite 4 níveis discretos de potência (ajustáveis no transmissor de 5, 10, 50 ou 100W) e a frequência pode ser variada de 0 à 30MHz.

Os sistemas irradiantes formam uma parte importante do processo de experimentação descrito nesta tese e seus diversos arranjos estão apresentados a frente no texto.

É interessante notar que o sinal de modulação está aplicado diretamente a entrada do transmissor projetada para a conexão de um manipulador telegráfico. Apesar da existência de um manipulador eletrônico neste rádio (capaz de gerar seqüências de pontos e traços automaticamente) ele foi desligado no *software* interno do rádio, através de seus próprios menus de controle.

5.2.1 O RÁDIOTRANSMISSOR

Como gerador de radiofrequência utilizou-se um transceptor comercial, desenvolvido para radioamadores pela empresa Japonesa Alinco⁴⁶, modelo DX-77T, ilustrado na Figura 23.

Figura 23 - Transmissor Alinco DX-77T utilizado como gerador principal do experimento.

Ele foi programado inicialmente para operar na frequência de 30MHz, com potência máxima de 10W, em modo CW – *Continuous Wave*, sendo utilizado somente como transmissor. Este sinal foi chaveado inicialmente pela interface de E/S do PC1 através de uma interface proprietária mostrada na Figura 12. A partir dos problemas de interferência entre aplicações do PC1 ele passou a ser chaveado pela interface USB , conectada ao PC2.

Em ambos os casos foi utilizada a entrada projetada para telegrafia do transceptor, que foi programado para operar sem o uso do manipulador eletrônico interno. A saída do transmissor era então convertida para 146MHz@8W pelo *transverter* e amplificada para 35W pelo amplificador.

Foram encontrados problemas com esta técnica de injeção, que saturava o processador com falhas, impossibilitando o funcionamento dos mecanismos de proteção projetados. Desta forma, operou-se o rádio na frequência de 7,8MHz@10W diretamente (sem o *transverter* ou o amplificador), com o mesmo padrão de chaveamento do sinal.

Uma característica importante deste equipamento é a presença de um circuito protetor, que limita, automaticamente, a sua potência na presença de uma relação

⁴⁶ <http://www.alinco.com/Products/DX-77T.shtml>

de ondas estacionárias alta, evitando danos permanentes ao equipamento em situações comuns no ambiente de experimentação.

As características principais do rádio são:

- Faixa de operações: 500 kHz - 30 MHz;
- Modos de operação J3E (USB/LSB), A1A (CW), A3E (AM), F3E (FM);
- Estabilidade de frequência: $\pm 10\text{ppm}$;
- Impedância de saída: 50 ohms, não balanceada;
- Alimentação: 13.8 VDC $\pm 15\%$;
- Consumo: $\sim 20\text{A}@13.8\text{Vcc}$ na transmissão (máx.);
- Potência de saída: SSB, CW, FM: 100W ou 10W;
- Emissões espúrias: $< -50\text{dB}@30\text{MHz}$.

5.2.2 O TRANSVERTER 146/30

O *transverter* tem a função de transladar o sinal de $30\text{MHz}@10\text{W}$ do gerador para $146\text{MHz}@8\text{W}$, no centro da faixa do amplificador utilizado. Para isto ele efetua a soma de 116MHz, através de um modulador balanceado.

Este equipamento é bidirecional, entretanto, para o propósito dos experimentos aqui descritos foi utilizado apenas no sentido de transmissão. A Figura 24 ilustra o equipamento:

Figura 24 - Transverter Ten Tec 1210 30MHz - 146MHz.

Fonte: PERNER (1999).

O objetivo de seu emprego reside no fato de que se buscava trabalhar na mais alta frequência possível. A motivação para isto reside em dois fatos principais: em primeiro lugar, quando o comprimento de onda λ da radiação transmitida se

aproxima das dimensões físicas do semiconductor – ou pelo menos o comprimento de onda das harmônicas mais distantes – a transferência de energia, e portanto a possibilidade de provocar um SEU aumenta; em segundo lugar, o dimensionamento do sistema irradiante, responsável por acoplar a potência produzida à pastilha, passa a ter dimensões mais adequadas.

A frequência de 146MHz obtida, entretanto, era chaveada por uma função degrau, gerando harmônicas que eram o verdadeiro sinal interferente. Na primeira fase do experimento, este *transverter* proporcionava a única forma de ajuste contínuo da potência do sistema, através do *trim pot* R54 da Figura 26, que aparece à direita na Figura 25 abaixo. O transistor de saída BLW81 está à esquerda. Possibilitar o acesso a este ajuste é o motivo do *transverter* estar sem a tampa superior na fotografia da Figura 30.

Figura 25 - Detalhe do ajuste de potência do *transverter*.

Este *transverter*, também concebido para o uso de radioamadores, possui uma boa estabilidade da frequência de saída.

Figura 26 - Diagrama de blocos do *transverter* Ten Tec 1210.

Fonte: PERNER (1999) (Tradução nossa).

Neste diagrama, percebe-se a entrada de 13,8Vcc que entra a esquerda inferior, diretamente para um regulador de tensão que gera as tensões reguladas necessárias ao equipamento. Assim que um sinal de RF entra na carga resistiva de 50W, que aparece no lado superior esquerdo, ele coloca o transverter no modo de transmissão. O *transverter* tem um oscilador local trabalhando em 58MHz (C16, Y1 e Q3) que é dobrado para 116MHz (Q4 e Q5). Este sinal é misturado com o sinal recebido do transmissor Alinco (0 à 30MHz), produzindo um sinal de 116 à 146MHz, em baixa potência. Este sinal é então passado por um filtro passa faixa (D13, Q6) e amplificado até 8W (Q7 à Q10). D17 e D18 fazem uma amostragem da potência refletida. Se ela subir acima de um patamar o conjunto U2A e U2B atua no amplificador reduzindo sua potência de maneira a proteger o transistor de saída (Q10, BLW81). R54 altera artificialmente o balanço de U2B, conseguindo-se desta maneira um ajuste da potência de saída.

As harmônicas do sinal eram o maior interesse inicial, mas, talvez pela presença do filtro de Q6, não se concretizou. É interessante notar que a mudança porcentual de frequência de uma harmônica de ordem mais alta para a seguinte, como ilustrado na Figura 27, (entre a décima e a décima primeira harmônicas mostradas em (D) e (E), por exemplo) é muito menor do que entre a fundamental e a segunda harmônica (B e C). Isto torna as variações de impedância mais rápidas, na medida em que o sistema irradiante (do qual o DUT – *Device Under Test* passa a fazer parte), se torna mais longo em termos de comprimentos de onda λ .

Figura 27 - Variações harmônicas do sinal interferente.

Fonte: ESTADOS UNIDOS. AMERICAN RADIO RELAY LEAGUE (1976).

Esta constatação nos fez acreditar que uma condição mais adequada poderia ser obtida através da diminuição da frequência, o que se tornou possível com o emprego da *stripline*, discutida no item 5.2.7.

A unidade utilizada foi montada pelo autor a partir de um *kit* comercial fabricado pela empresa norte americana Ten Tec, modelo 1210⁴⁷. Este *transverter* deixou de ser empregado quando a frequência do experimento passou para 7,8MHz.

5.2.3 A INTERFACE DO PROGRAMA GERADOR DE FALHAS COM O RÁDIO

O objetivo do experimento é introduzir falhas no cartão sob testes. Desta maneira buscou-se o padrão de radiofrequência induzida que pudesse provocar, mais rapidamente, a introdução de uma falha insegura no sistema. Esta determinação é empírica. Há, entretanto, normas para testes de dispositivos eletrônicos com relação a *EMI – ElectroMagnetic Interference*, tanto conduzida como na IEC61000 4-17 (*INTERNATIONAL ELECTROTECHNICAL COMMISSION*, 2001) e na IEC61000 4-29 (*INTERNATIONAL ELECTROTECHNICAL COMMISSION*, 2000), ou irradiada como na EN 61000-4-3 (*EUROPEAN COMMITTEE FOR TECHNICAL STANDARDIZATION*, 2006) e na NBR12304 (*ASSOCIAÇÃO BRASILEIRA DE NORMAS TÉCNICAS*, 1992), nacional.

Os padrões de testes destas normas, como indicado por Vargas et al. (2003), visam testar a imunidade de circuitos microprocessados à radiofrequência.

É importante salientar que o padrão interferente utiliza uma senóide de alta frequência (que é o próprio sinal de RF, inicialmente em 146MHz e finalmente em 7,8MHz), modulado por uma onda quadrada descrita a seguir. A idéia é otimizar a interferência ao longo de uma porção significativa do espectro de radiofrequência. A representação teórica de uma função degrau por uma série de Fourier apresenta uma soma de infinitas parcelas, acrescentando harmônicas infinitas ao padrão interferente, embora progressivamente atenuadas. No caso específico deste experimento acontece uma limitação na geração destas harmônicas. Estes rádios foram projetados para operar dentro das faixas de radioamadorismo regulamentadas pela Anatel e portanto possuem excelentes filtros em suas saídas que visam exatamente suprimir estas harmônicas que poderiam favorecer o padrão interferente. Entretanto, estes são filtros para alta potência, que por limitação

⁴⁷ <http://www.tentec.com>

5.2.4 O CARTÃO DE E/S USB

Originalmente a interface mostrada na Figura 29 foi concebida conectada ao mesmo PC1 da aplicação de verificação da ocorrência de falhas. O controle da geração do padrão interferente da Figura 28 era realizado por uma tarefa rodando sob o sistema operacional Ubuntu do PC1.

Como a tarefa de verificação e o simulador, que rodam nesta máquina, são aplicações com demandas expressivas de capacidade computacional, notou-se que a execução destas tarefas causava alterações no padrão interferente desejado, alongando os tempos e diminuindo as transições ou degraus. A causa por trás desta constatação advém do fato de que os sistemas operacionais empregados não são de tempo real.

Desta maneira, concebeu-se uma alteração, passando-se a gerar o padrão interferente na máquina PC2, equipada com o sistema operacional Windows XP, e que originalmente era apenas utilizada na programação do microcontrolador AVR, através da interface ISP.

Como originalmente esta máquina não era utilizada ao longo da execução do experimento, conseguiu-se desta forma gerar o padrão interferente desejado sem interferência entre processos. Assim, foi desenvolvida uma aplicação em Linguagem C no PC2, fazendo uso de uma DLL especial para controle de uma interface de E/S conectada através do canal série USB.

A Figura 31 ilustra, em blocos, como ficou o arranjo experimental, mostrado originalmente na Figura 12. A Figura 30 apresenta uma foto do conjunto de testes montado.

Funcionalmente não há alterações. A partir desta modificação, o PC1 no início da tarefa de verificação zera o seu cronômetro interno e seta um *bit* da placa de E/S do PC1 que é lido pela placa de E/S USB do PC2. A partir da detecção deste sinal o *software* do PC2 passa a gerar o padrão interferente, para a entrada do transceptor Alinco DX-77, através do mesmo relê da Figura 29, agora conectado a saída da interface USB. Quando o *software* de detecção de falhas do PC1 percebe uma falha insegura, ele para o cronômetro e remove o mesmo *bit* que disparou a geração da interferência. O PC2 percebe esse sinal e interrompe a geração do padrão interferente.

Figura 30 - Foto do experimento montado, já com a placa USB.

Figura 31 - Arranjo final do *hardware* do experimento com a introdução da interface USB.

A interface USB foi construída pelo autor a partir de um *kit* fabricado pela empresa Belga Velleman (Figura 32)⁴⁸, modelo K8055, e possui as seguintes características básicas:

- 5 entradas digitais (0 = terra, 1= aberto ou +V), com um botão de teste para cada entrada;
- 2 entradas analógicas com atenuação e ganho ajustáveis, com conexão de testes ao +5V por *jumpers*;

⁴⁸ <http://www.velleman.be/ot/en/product/view/?id=351346>

- 8 saídas digitais em coletor aberto (Max. 50V/100mA), com indicação a LED;
- 2 saídas analógicas:
 - 0 à 5V, resistência de saída 1,5k Ω ;
 - PWM 0 a 100%, saídas em coletor aberto (Max. 100mA / 40V) com indicação a LED.
- Tempo médio de conversão de 20ms por comando;
- Alimentação via USB, consumo médio de 70mA;
- Interface de *software* DLL pronta, programável em qualquer linguagem de programação.

Figura 32 - Interface de E/S USB para geração do padrão interferente.

Fonte *site* Velleman⁴⁸.

5.2.5 AMPLIFICADOR DE RF

O Amplificador de radiofrequência empregado na primeira fase do experimento foi construído pelo autor a partir de um *kit* produzido pela empresa norte americana Communication Concepts, Inc.⁴⁹, modelo 335A (Figura 33). As suas características principais são:

- Frequência de operação: 144 a 148 MHz;
- Potência de RF de entrada: 250 mW a 3 Watts (5 watts Max.);
- Potência de RF de saída: 2 a 35 watts (típico);

⁴⁹ <http://www.communication-concepts.com/335a.htm>

- Modo de operação: classe AB;
- Aplicação: SSB-FM-CW;
- COR – *Carrier Operated Relay*, relê operado pela portadora de entrada;
- Alimentação em corrente contínua: 13.6 V;
- Consumo de corrente contínua: 4.5 a 5 A;
- Conexões de E/S padrão BNC;
- Conexão de alimentação: conector *Cinch-Jones* de 4 pinos.

Figura 33 - Amplificador de radiofrequência de 35W empregado no experimento.

Este equipamento foi desenvolvido para uso de radioamadores, e, por esta razão, tem sua banda de passagem estreita, com apenas 4MHz de largura, o que limitava a frequência do ensaio. Como a hipótese inicial era de que a frequência exata do teste no arranjo proposto para o experimento não fosse importante, a frequência de testes inicial foi estabelecida em 146MHz, bem no centro da banda passante do amplificador. Como este amplificador tem um filtro passa faixas em sua saída a emissão de harmônicos muito distantes da portadora fica proporcionalmente atenuada pela curva deste filtro. Este fenômeno se dá também com a saída direta do *transverter* ou a saída do rádio de HF, empregados em várias fases dos experimentos.

5.2.6 ACOPLAMENTO DE IMPEDÂNCIAS, WATÍMETRO E MONITOR DE RELAÇÃO DE ONDAS ESTACIONÁRIAS

Para os diversos experimentos realizados em VHF (146MHz), com diversos arranjos de sistemas irradiantes (no caso induzido) ou cargas (no caso conduzido), as condições das impedâncias encontradas variavam muito. Esta situação, conforme

descrito, prejudicava a eficiência da transferência de potência do amplificador ou do conversor para o experimento, além de colocar em risco a integridade dos mesmos.

De maneira a monitorar as potências diretas e refletidas e sua relação (ROE-Relação de Ondas Estacionárias⁵⁰), além de otimizar a transferência de potência, foi empregado um instrumento combinado *SWR Tuner*, modelo SWR-20ATM da OPEK⁵¹, mostrado na Figura 34. Este instrumento combina um Wattímetro com um sintonizador de antenas de VHF/UHF (*Ultra High Frequency*), adequado às necessidades dos experimentos em VHF descritos nesta tese.

Figura 34 - Wattímetro e sintonizador de antena OPEK SWR-20ATM.

Este instrumento tem as seguintes características:

- Frequência de trabalho: 136 a 175MHz (VHF) e 420 a 460MHz (UHF);
- Alcance em potência: 0 a 60W em duas escalas (15 e 60W);
- Relação de ondas estacionárias: 1,0 a 5,0;
- Acoplamento de impedâncias: 20 a 125 Ω ;
- Perda por inserção: <0,5dB;
- Tolerância: $\pm 10\%$ no fundo de escala.

Para o caso dos experimentos realizados em HF-*High Frequency* (7,8MHz) este instrumento se mostrou inadequado, já que a rede de acoplamento de impedâncias interferia com o sinal gerado. Neste caso, a carga foi sempre uma carga ôhmica, puramente resistiva, de 50 Ω conectada através de um *stripline*, dispensando o acoplador de impedâncias. Para monitoração da potência em HF foi empregado um Watímetro/monitor de ROE Hansen FS5, ilustrado na Figura 35.

⁵⁰ Do Inglês SWR-Stationary Wave Rate.

⁵¹ <http://www.opektech.com/>

Figura 35 - Watímetro e medidor de ROE Hansen FS5.

Suas principais características são:

- Alcance watímetro: 100W em duas escalas, 10 e 100W;
- Alcance medidor de ROE: 1 a 3;
- Resposta em frequência 3 a 144MHz;
- Impedância: 50 a 52 Ω ou 73 a 75 Ω comutável por chave.

O resistor empregado, mostrado na Figura 36, é um resistor de óxido de Berílio de 50 Ω , normalmente empregado como carga fictícia em estações repetidoras de telefonia celular, adaptado a um dissipador de alumínio e a um conector BNC. Este resistor se mostrou capaz de suportar potências contínuas de 100W com aquecimento aceitável do dissipador.

Figura 36 - Carga ôhmica de terminação para o *stripline*.

5.2.7 OS SISTEMAS IRRADIANTES

Para a transmissão do sinal da saída do amplificador ou diretamente do *transverter*, experimentado em alguns casos, foram empregadas diversas configurações de sistemas irradiantes.

Inicialmente foi utilizada uma antena tipo *rubber duck*, usualmente empregada em transceptores comerciais de VHF-*Very High Frequency*, modelo FA-B44V da empresa ICOM⁵². Esta antena foi posicionada diretamente sobre o chip durante os experimentos, e está ilustrada na Figura 37.

A motivação para a escolha da produção da radiação induzida em 146MHz está diretamente relacionada com o comprimento de onda λ do sinal induzido, que é dado por $\lambda = \frac{c}{f}$, onde $c = 3 \cdot 10^8 \frac{m}{s}$ é a velocidade de propagação das ondas eletromagnéticas no vácuo (muito próxima da do ar) e f a frequência. Para 146MHz, $\lambda \cong 2,06m$, o que permite construir uma antena bobinada de $\frac{1}{4}\lambda$ muito pequena, com pouco mais de 10cm, adequada às condições do experimento projetado. A escolha de um múltiplo adequado de λ para a antena é muito importante, já que isto garante um acoplamento de impedâncias correto na saída do amplificador de 35W. Um tamanho inadequado de antena implicaria em uma relação de ondas estacionárias muito alta, que levaria à presença de uma potência refletida que poderia arruinar o amplificador. De maneira análoga, a potência efetivamente transmitida seria reduzida, diminuindo, por consequência, a probabilidade da introdução de uma falha no sistema sob teste.

Figura 37 - Antena ICOM FA-B44V utilizada no experimento.

O emprego desta antena trouxe consequências indesejadas, na medida em que introduziu falhas em todos os subsistemas do experimento, incluindo os monitores de vídeo dos computadores. Este já era um reflexo esperado. Conforme já explanado anteriormente, a razão para a escolha de comunicação fio a fio entre o PC do simulador do mundo real e o DUT foi o de que a susceptibilidade dos controladores de comunicação seriais ou de rede comprometeria o experimento

⁵² <http://www.icomamerica.com/>

como um todo, já que uma falha global destes controladores impediria a detecção da falha específica ocorrida.

Esta precaução se mostrou realista. Havia interferência por indução eletromagnética em alguns fios, mas não em todos, e foi possível monitorar e detectar a origem dos problemas.

A primeira variação adotada foi a blindagem da fiação com papel alumínio aterrado, que se mostrou simultaneamente ineficaz e de difícil aplicação.

O próximo passo foi projetar uma blindagem mecânica de alumínio para a antena, mantendo uma janela para irradiação sobre o *chip*. O maior problema encontrado no projeto desta blindagem foi o desconhecimento do padrão de irradiação da antena empregada para o campo próximo, e a indisponibilidade de instrumental para determiná-lo. Este campo poderia ser calculado, mas é difícil prever analiticamente o efeito de blindagens, de maneira que a abordagem empírica foi adotada. A cada modelo projetado e construído, havia uma completa transformação do diagrama de irradiação e da impedância da antena. A impedância podia ser adequadamente compatibilizada através do acoplador do watímetro mas o padrão irradiante foi sempre inadequado. A Figura 38 ilustra alguns dos modelos projetados e fabricados.

Figura 38 - Modelos de blindagens experimentais empregadas.

Todos estes modelos se mostraram inadequados. As blindagens foram muito eficazes contra as interferências indesejadas, mas a antena se mostrou incapaz de induzir falhas no DUT quando blindada. Isto se deve provavelmente ao fato de que este tipo de blindagem aparentemente aniquila o campo magnético, deixando

apenas o campo elétrico, que se mostra incapaz de, mesmo com potências na antena superiores a 40W, aplicar no *chip* uma tensão suficiente para provocar um *bit flip* no substrato MOS.

Com a inadequação dos resultados obtidos com a antena *rubber duck* foi necessário partir para o projeto de uma outra antena, que pareceu mais adequada. Mantendo a necessidade de que o elemento irradiante tivesse dimensões compatíveis com o microcontrolador, optou-se pelo desenho de uma antena do tipo *small loop*.

A alternativa encontrada foi o emprego de uma bobina entre o adaptador do *chip* e a placa de circuito impresso principal da interface AVR Totalflex, como mostrado nas ilustrações da Figura 39.

Figura 39 - Arranjo experimental com bobina sob o ATMEGA128.

Este arranjo se mostrou muito eficaz no sentido de induzir falhas no microcontrolador. Devido à sua limitação física (a bobina tinha 20 espiras de fio 18AWG e conseqüente baixa indutância) e às limitadas capacidades de acoplamento de impedâncias do acoplador, só foi possível operar este sistema com

os mesmos 146MHz dos sistemas anteriores. A grande vantagem desta concepção foi o confinamento completo da irradiação que, com a colocação de uma tampa de alumínio simples sobre o perfil quadrado da Figura 39, não provocava qualquer interferência mensurável na fiação ou qualquer outra parte do circuito impresso da placa. Além disto, o campo total, em função do formato toroidal da bobina, passou a compreender uma parcela significativa de campo magnético, que pode ser modelado como o *small loop* presente em qualquer livro texto de antenas, como o da *ESTADOS UNIDOS. AMERICAN RADIO RELAY LEAGUE* (1976), por exemplo.

O maior problema encontrado com todos os sistemas descritos até aqui foi o fato de que havia um *threshold* na injeção das falhas. Abaixo de um certo nível de potência as falhas não se manifestavam e acima deste havia uma “inundação” de falhas no semiconductor. Esta saturação provavelmente introduzia falhas tanto nas rotinas a serem testadas como nos mecanismos de correção de falhas, agindo em vários componentes internos do complexo microcontrolador, como sugerido por AVIŽIENIS *et al.* (2004): “Alguns erros (por exemplo um *burst* de radiação eletromagnética) podem causar erros simultaneamente em mais de um componente. Estes erros são chamados de erros múltiplos relacionados”.

A solução encontrada foi uma guinada completa nas diretrizes adotadas para o sistema de injeção de falhas por radiofrequência, com o dispositivo ilustrado na Figura 40.

Figura 40 - Stripline, como a utilizada no experimento (Tradução nossa).

Fonte: WILLIAMS (1992)

Em 1977 GROENVELD E JONG (1977) *apud* WILLIAMS (1992) desenvolveram um dispositivo muito simples, que se constituía em uma secção de linha de transmissão aberta, formada por duas placas paralelas. Este dispositivo foi concebido para

receber o sinal de transmissão de radiofrequência por um lado e dissipá-lo em uma terminação ou carga fictícia adequada no outro lado, , colocando-se o equipamento sob teste em seu interior.

A primeira grande vantagem deste dispositivo, além da sua simplicidade, reside no fato de que o campo elétrico no seu interior é homogêneo, sendo dado simplesmente pela relação $E=V/h$ [V/m], onde V é a tensão medida entre as placas e h é a distância entre elas. Este fato propicia uma oportunidade interessante de pesquisa, adicional a este trabalho, onde se investigaria o correlacionamento entre os níveis absolutos do campo elétrico interferente e taxa de introdução de falhas em sistemas de aplicações críticas. Além disso, na *stripline* o campo entre as placas se propaga em modo TEM-*Transverse ElectroMagnetic*, que, segundo WILLIAMS (1992), tem as mesmas características que a propagação do espaço livre.

O campo elétrico no interior de uma *stripline* está ilustrado na Figura 41:

Figura 41 - Campo elétrico no interior de uma *stripline* em 50 e 200MHz.

Fonte: BITTINGER (1993).

É possível notar a homogeneidade do campo elétrico mostrado a esquerda, para 50 MHz, onde só há uma deformidade, provocada pelo efeito das bordas. Entretanto, para 200MHz, já é possível observar uma grande irregularidade na distribuição do campo elétrico. Esta informação é provavelmente a razão pela qual foi possível obter os melhores resultados desta tese em baixa frequência, ao redor de 8MHz.

Como vantagem adicional, o seu tamanho físico guarda uma relação pequena com a faixa de frequências úteis do dispositivo. Células TEM são uma extensão do conceito das *striplines*, onde se instala um septo entre as placas, ligado ao sinal e as duas placas ao terra, com um fechamento lateral. Este arranjo confina totalmente o campo e seu volume típico é o dobro de uma *stripline* equivalente.

As *striplines* aparecem na literatura, algumas vezes, denominadas como células TEM abertas.

A tabela a seguir dá uma idéia dos volumes de células TEM em relação ao tamanho máximo dos dispositivos passíveis de teste, o que permite inferir o tamanho das *striplines* equivalentes:

Tabela 14 - Tamanho de células TEM em relação à faixa de frequências.

Tamanho da célula [cm ²]	Tamanho máximo do dispositivo sob teste [Largura x Comprimento x Altura]	Faixa de frequências [MHz]
30,5	15x15x5	0-500
61	20x20x7,5	0-300
91,5	30,5x30,5x10	0-200
122	40,5x40,5x15	0-150
183	61x61x20	0-100

Fonte: WILLIAMS (1992).

As *striplines* têm sido incluídas em diversas normas, como na MIL-STD-461 [197-?] *apud* MERTEL (1995) ou nas IEC801-3 (1984) e EN55020 [198-?] *apud* WILLIAMS (1992), com vários arranjos, como mostrado na Figura 42, que ilustra a sua aplicação pela norma MIL-STD-461 que visa verificar a imunidade de dispositivos a ruídos de natureza eletromagnética.

Figura 42 - Uso de stripline como arranjo de teste da norma MIL-STD-461.

Fonte: MERTEL (1995).

Embora a sua concepção inicial fosse voltada às avaliações de susceptibilidade eletromagnética, elas nos apareceram como dispositivos ideais para a injeção de falhas. A Figura 43 ilustra a *stripline* construída para o experimento, tanto na posição de operação da placa como em detalhe construtivo.

Figura 43 - A stripline construída para o experimento desta tese.

O uso desta *stripline*, conjugado com a terminação resistiva mostrada, permitiram a experimentação com uma extensa variação de frequências (0 à 30MHz e 116 à 146MHz) e de potências (0 à 100W, em várias configurações).

O resultado desta experimentação foi a obtenção de um conjunto frequência/potência ideal, que permitiu a avaliação comparativa efetiva das técnicas de programação defensiva escolhidas. Nesta aspecto, a *stripline* representou uma nova diretiva no desenvolvimento do trabalho experimental. Antes de sua introdução não foi possível comprovar as hipóteses levantadas no início do projeto. Após a sua introdução isto foi factível.

Avaliações qualitativas/quantitativas, utilizando versões dos programas com e sem a utilização de técnicas de programação defensiva, foram conduzidas, indicando que a melhor sensibilidade para o experimento se encontrava entre 7,5 e 8MHz@10W. Esta frequência provavelmente é própria do microcontrolador em uso, associada a suas características físicas.

Esta hipótese poderia ser comprovada repetindo-se os testes com o emprego de um outro processador.

Um experimento foi então conduzido, medindo o tempo até uma falha insegura ocorrer, de modo a refinar o valor mais adequado. Seus resultados estão descritos na Tabela 15 a seguir.

Tabela 15 - Experimento para determinação do par frequência/potência ideal para o teste.

freqüência/potência	7,50MHz@10W		7,65MHz@10W		7,8MHz@10W		8,0MHz@10W	
versão de software	Exp_1	Exp_dupla6	Exp_1	Exp_dupla6	Exp_1	Exp_dupla6	Exp_1	Exp_dupla6
Dados medidos.	8,1	21,9	1,5	2,7	5,1	17,4	3,0	1,5
Tempo[s] até falhar de modo inseguro.	5,4	36,6	8,7	2,4	12,3	9,3	0,6	5,4
	32,0	24,0	18,6	1,2	4,2	7,2	0,6	1,2
	2,1	18,9	0,6	61,2	15,9	2,4	1,5	0,6
	8,4	3,6	1,8	3,3	2,1	32,1	9,0	0,6
	3,6	62,4	0,6	8,1	15,3	7,8	1,5	2,1
	8,4	11,1	3,0	0,9	5,7	10,8	1,2	4,5
	4,2	51,3	20,4	22,8	10,5	21,0	3,6	7,5
	1,2	38,2	18,3	3,0	11,4	1,8	1,8	1,2
	50,0	14,1	6,3	4,5	19,5	27,9	3,6	1,8
média	19,9	28,2	7,9	11,1	10,2	13,8	2,60	2,64
desvio padrão	27,5	18,6	8,1	18,8	5,8	10,4	2,5	2,3
variância	682,2	311,4	59,1	317,4	29,9	98,1	5,6	5

O programa Exp_1 realiza as funções sem quaisquer técnicas de programação defensiva, enquanto o programa Exp_dupla6 implementa a duplicação de todas as variáveis de entrada e saída do microcontrolador. As medidas realizadas com 8MHz apresentaram uma variância muito baixa, indicando que se estava inundando o *chip* de falhas novamente, e, portanto, foi descartada, embora a introdução da técnica de programação defensiva podia ser percebida, através da pequena variação da média (2,5%). Para a frequência de 7,5MHz, embora a variação da média seja a maior (41%) as médias dos tempos são muito altas, o que tornaria os experimentos a serem realizados extremamente longos, descartando este par. Entre 7,65MHz e 7,8MHz a decisão foi tomada pela frequência de 7,8MHz@10W, que mantinha o desvio padrão inferior a média.

É necessário lembrar aqui que estes valores são adequados para o conjunto *stripline*, microcontrolador, placa de desenvolvimento empregados, já que não é possível descartar completamente a hipótese de que possam haver modos de falha privilegiados. É interessante notar que, de maneira análoga ao que acontecerá em todo o experimento, as variâncias são muito maiores do que a média, indicando uma distribuição não normal, o que está discutido à frente com mais detalhes.

5.3 O SOFTWARE EMPREGADO

Com exceção do *software* ou programa-alvo, que é executado no *hardware* da placa AVR, a linguagem de desenvolvimento das aplicações é indiferente. Isso se dá em função da enorme diferença temporal entre essas aplicações e o sistema em

tempo real da placa AVR, muito mais lento. Para fins de simulação esses tempos foram acelerados em escala, mas, mantiveram ainda, uma proporção, que fez com que os programas da estação de trabalho ou testes fossem consideravelmente mais rápidos do que o programa-alvo.

Do ponto de vista das aplicações de *software*, os sinais a serem tratados por cada tarefa estão listados na Tabela 16.

Tabela 16 - As entradas e saídas de cada aplicação da plataforma de testes.

IHM	<i>Entrada</i>	Ocupação do simulador via JNI <i>Status</i> da chave do AVR (6 bits, 3 por chave) Código de velocidade do AVR (24 bits, 2 por circuito)
	<i>Saída</i>	Comando de rota para AVR (4 bits, 2 por chave)
Simulador	<i>Entrada</i>	Comando de chave do AVR (2 bits, 1 por chave) Código de velocidade do AVR (24 bits, 2 por circuito)
	<i>Saída</i>	Posição da chave para AVR (4 bits, 2 por chave) Ocupação para AVR (12 bits, 1 por circuito) Ocupação para IHM, via JNI
Verificador	<i>Entrada</i>	Todas
	<i>Saída</i>	<i>Reset</i> para AVR (1 bit)
AVR	<i>Entrada</i>	<i>Reset</i> (1 bit ligado no <i>reset</i>) Comando de rota da IHM (4 bits, 2 por chave) Ocupações do simulador (12 bits, 1 por circuito) Posição da chave do simulador (4 bits, 2 por chave) Modo da IHM (1 bit)
	<i>Saída</i>	Comando da chave para simulador (2 bits, 1 por chave) Código de velocidade para simulador (24 bits, 2 por circuito) Código de velocidade para IHM (24 bits, 2 por circuito) <i>Status</i> da chave para IHM (6 bits, 3 por chave)

Estes sinais são, por vezes, compostos, onde a informação está codificada nos vários bits que a compõe. Este arranjo foi adotado para diminuir o número de bits necessário para produzir uma plataforma de testes de complexidade significativa.

Esta estratégia, entretanto, traz o risco de introdução de erros combinacionais nos cabos de conexão. Este tipo de interferência não foi percebido no projeto, mas existe a possibilidade de que ocorra. Este tipo de erro ocorre, principalmente, quando uma determinada quantidade de bits '1' se faz presente, simultaneamente, em um cabo de vias paralelas.

A Tabela 17 ilustra essa interpretação para todos os sinais envolvidos na plataforma de testes.

Tabela 17 - A codificação das informações com a interpretação das combinações binárias.

Item	Codificação	Comunicação	Quantidade
Ocupações	Ocupado = 1 Desocupado = 0	AVR <--- PC	12x1
Códigos de velocidade	0 Km/h = 00 50 Km/h = 01 75 Km/h = 10 100Km/h = 11	AVR ---> PC	12x2
Comandos de rota	Rota AB = 00 Rota DB = 01 Rota DC = 10 Rota não requisitada = 11	AVR <--- PC	2x2
Status da chave	Ausente = 000 AB alinhada = 001 DC alinhada = 010 DB alinhada = 011 AB + DC alinhadas = 100 Em movimento = 101 Em normal sem rota = 110 Em reverso sem rota = 111	AVR ---> PC	2x3
Posição da chave	Em normal = 00 Em reverso = 01 Em movimento = 10 Ausente = 11	AVR <--- PC	2x2
Comando da chave	Reverso = 0 Normal = 1	AVR ---> PC	2x1
Comando modo (auto / local)	Local = 0 Auto = 1	AVR <--- PC	1x1
Reset		1	
Total		53 bits	

5.3.1 O SIMULADOR DO MUNDO REAL

Trabalhar com um sistema físico real traria complicações e um dispêndio de tempo que pôde ser superado por meio do uso de técnicas de simulação. Essa tarefa visou simular o comportamento de um sistema no mundo real. Como exemplo, a placa AVR envia pedidos de mudança de posição das máquinas de chave que são recebidos pelo simulador. O simulador, após um atraso proporcional ao tempo real, devolve uma indicação ao AVR, indicando que a solicitação foi consolidada.

A simulação de elementos de campo pode ser dividida em “Máquina de chave” e “Ocupações de circuitos de via”. As máquinas de chave possuem uma implementação que leva em consideração o comando – Normal ou Reverso - recebido via *hardware* e o tempo aproximado de movimentação de uma máquina de chave real.

Já as ocupações de circuito de via são produzidas a partir da interação entre os componentes “Trem”, “Circuito de via”, “Desvio de intertravamento” e “código de velocidade”. De forma resumida, os trens recebem um código de velocidade por meio de seleção proveniente das entradas de *hardware*. Com este código o trem calcula repetidamente a posição de sua frente e de sua traseira dentro de um determinado circuito. Assim que os cálculos retornam que sua frente atingiu uma das extremidades do próximo circuito, a partir de sua extensão, este ocupa o circuito seguinte. Este circuito pode ser fixo ou cambiável. Um circuito é cambiável em casos onde um trem passa por um desvio de intertravamento.

O mesmo processo se dá para a traseira do trem, para desocupações do circuito anterior.

Os eventos de campo podem ser calculados a partir de tempos reais ou tempos reduzidos. Com tempos reduzidos, pode-se criar uma simulação de circulação de trens acelerada, o que pode ser alterado através do preenchimento de um campo pertencente a um dos painéis de controle da interface.

5.3.2 A APLICAÇÃO INJETORA DE FALHAS

Este programa roda no PC2, verificando o início do ciclo de testes disparado pelo PC1 através da interface USB e acionando o transmissor com o padrão interferente. Este programa foi elaborado no interpretador Ch na Linguagem C:

```
/* File name: controle.cpp. Programa criado para o controle da injeção de falhas.``*/
#include <K8055D_C.h>
#include <stdio.h>
#include <time.h>
int main() {
    void wait(int ms_delay);
    int intCardAddress = 0; // USB card address
    int i;
    OpenDevice(intCardAddress); //Open card
    while(1){
        if(!ReadDigitalChannel(1))
        {
            SetDigitalChannel(1);
            wait(1000);
            ClearDigitalChannel(1);
            wait(250);
        }
        if(!ReadDigitalChannel(1))
        {
            SetDigitalChannel(1);
            wait(250);
            ClearDigitalChannel(1);
            wait(250);
        }
    }
}
```

```

    }
    if(!ReadDigitalChannel(1))
    {
        SetDigitalChannel(1);
        wait(250);
        ClearDigitalChannel(1);
        wait(250);
    }
    if(!ReadDigitalChannel(1))
    {
        SetDigitalChannel(1);
        wait(250);
        ClearDigitalChannel(1);
        wait(250);
    }
}
CloseDevice();
return 0;
}
void wait(int ms_delay)
{
    int init = (int)clock();
    int final = init + ms_delay * (CLOCKS_PER_SEC / 1000);
    int actual;
    while(actual < final)
        actual = (int)clock();
}

```

Inicialmente são incluídas as bibliotecas padrão de entrada/saída e de temporização, bem como o arquivo `K8055D_C.h`, que contém as funções específicas da placa USB. A seguir é construído o protótipo da função `wait`, que espera um argumento inteiro, que é traduzido em uma parada espontânea do programa com duração deste inteiro em ms. A função `wait` é construída a partir da função `clock()` da biblioteca `time.h`, que devolve tempos absolutos que são somados para a determinação do atraso requerido. A seguir o endereço USB da placa é estabelecido em zero, que também foi configurado nos *jumpers* da placa. Na seqüência, o programa começa a produzir o sinal de modulação no canal 1 de saída da interface USB. Inicialmente o sinal é ligado por 1s e desligado por 1/4s e a partir daí é ligado por 1/4s e desligado também por 1/4s, por três vezes, quando então o ciclo é reiniciado. É importante notar que este ciclo só acontece enquanto a entrada 1 da placa USB, permanecer habilitada. Este sinal habilita a geração do sinal interferente a partir do seu recebimento, proveniente do cartão E/S digital do PC1. Foram usados 4 *loops* com `if` para garantir uma aparada mais rápida do sinal interferente na detecção de uma falha pela tarefa verificadora.

5.3.3 A APLICAÇÃO DE VISUALIZAÇÃO

Essa aplicação é executada na estação de trabalho de testes, com as funções exclusivas de visualizar a aparência do suposto sistema real e emitir comandos para o sistema alvo. Este sistema é constituído por uma via ferroviária dupla com duas zonas de manobra, constantes de travessões e quatro máquinas de chave no total. Além disto há 6 circuitos de via, responsáveis pela informação de posicionamentos dos trens, numerados de A1 a F1 para a via 1 e de A2 a F2 para a via 2, conforme ilustrado na Figura 44.

Os componentes visuais que representam os circuitos de via estão contidos em uma estrutura que permite compor elementos que representam circuitos de via simples e regiões de intertravamento, assim como componentes que exibem códigos de velocidade. Também foram inseridos componentes que exibem o estado de bloqueios de intertravamento e recebem comandos para seleção de rotas “origem-destino”. Estes comandos foram exclusivamente utilizados para testes (debug) do simulador, não sendo usados durante avaliação de erros provocados por interferência eletromagnética. Com a mesma finalidade foram criados componentes que simulavam a ocupação de um circuito escolhido.

Figura 44 - Esboço do sistema a ser controlado.

Para monitoração do estado de cada entrada e saída de *hardware*, foram criados dois outros painéis auxiliares. Em um deles visualiza-se o estado de cada entrada ou saída em tempo real. Em outro são registrados as mudanças de estado de cada uma delas, a fim de se criar um relatório de registro de eventos. Toda a implementação da interface visual baseou-se em Java – *Swing*.

Todos os componentes que acessam o *hardware* ou recebem eventos dele, o fazem por meio de uma única classe que utiliza JNI – *Java Native Interface*. Resumidamente, este *framework* possibilita que classes Java se conectem com o *hardware* de uma máquina através de arquivos de biblioteca dinâmica. No sistema operacional Windows estes arquivos são reconhecidos pela extensão DLL, já em sistemas Linux, como no nosso caso, isto se dá através da extensão SO. Desse modo, foi necessário a criação de um arquivo de conversão com implementação em C e parametrização JNI.

Esse conjunto foi implementado na tela do PC, através da IHM, conforme apresentado na Figura 45:

Figura 45 - Visualização da interface operacional que se apresenta ao usuário.

Essa interface foi projetada para permitir a interação do operador com o sistema de maneira análoga à existente em um sistema de controle ferroviário convencional.

5.3.4 A APLICAÇÃO DE VERIFICAÇÃO

Para garantir a independência dos testes, o processo de supervisão do experimento concebido deveria se manter isolado do sistema sob testes. Isso foi obtido fazendo-se com que a missão de supervisão fosse executada por uma tarefa exclusiva para essa aplicação, mas que, ao mesmo tempo compartilhasse a CPU com as outras aplicações, possibilitando a supervisão de maneira pouco interferente. Os recursos de independência do sistema multitarefa, *multi-thread* e preemptivo do Linux, foram um dos fatores importantes que motivaram a escolha deste sistema operacional para o PC1.

Objetivava-se obedecer um conceito clássico, denominado exclusão mútua: “Dois (ou mais) processos podem ser executados concorrentemente em completa segurança contanto que eles acessem apenas aqueles recursos (como memórias ou dispositivos de entrada e saída) que sejam privativos a eles” (LEIGHT, 1988). Muito embora essa afirmação teórica seja inegavelmente consistente, tal independência é, na prática, impossível. Os processos precisam se comunicar para executar a função para a qual foram concebidos. O mesmo autor realça o fato de que, para que a segurança seja mantida, apenas o acesso aos recursos necessita ser mutuamente exclusivo.

Se a região comum for denominada como região crítica, um determinado processo p deve apenas monitorar o acesso a um recurso crítico e aguardar se ele estiver sendo utilizado por um processo $p1$. Para isso p e $p1$ devem gerar um sinal de que desocuparam o recurso. Esses sinais deverão ser monitorados reciprocamente. Como o *kernel* do sistema operacional usado na máquina não prevê esse recurso de maneira explícita, ele foi implementado no código das aplicações com o conceito de semáforo.

Essa é a aplicação principal do sistema, responsável pela verificação de cada alteração de entrada e de saída do sistema sob teste. Inicialmente ela foi concebida para verificar todo o conjunto de *bits* de E/S, verificando se o conjunto é próprio, impróprio ou inválido:

- por conjunto próprio entende-se uma combinação de entradas e saídas segura e animada de significado físico;
- por conjunto impróprio entende-se uma combinação de entradas e saídas insegura, e animada de significado físico;
- por conjunto inválido entende-se uma combinação de entradas e saídas sem significado físico.

Essa verificação, na prática, demandaria a monitoração simultânea em tempo real dos 53 sinais de interface e a sua comparação com uma tabela de estados que contaria com 2^{53} linhas, o que representa uma grande dificuldade prática.

A solução encontrada foi efetuar um levantamento de todas as situações inseguras, com significado físico, e introduzi-las no software de verificação.

A Figura 46 ilustra uma janela da aplicação de verificação sobre a aplicação de visualização (IHM), indicando a presença de código diferente de zero dentro do intertravamento sem rota alinhada.

É possível observar que a tarefa de verificação não tem interface gráfica, o que foi feito para manter o seu tamanho e carga de execução nos menores níveis possíveis. Embora seja impossível eliminar a interferência desta tarefa na tarefa do simulador, já que elas são executadas na mesma máquina, esta providência visou minimizar esta interferência.

Desta forma, a tarefa de verificação se comunica com o usuário através de uma console de comandos simples do sistema operacional Ubuntu.

O programa verificador de falhas foi concebido para ser executado paralelamente com o simulador, indicando como saída o tempo decorrente entre o início da simulação e a ocorrência de uma falha insegura. Sua descrição completa pode ser encontrada no APENDICE B.

Este programa monitora o tempo decorrido entre o início do processo de injeção de falhas e o instante em que uma falha de segurança é detectada. Entretanto, o Ubuntu não é um sistema de tempo real, o que faz com que exista a possibilidade de concorrência por recursos, o que poderia comprometer a eficácia da tarefa de verificação. Como forma de diminuir a possibilidade de que esta situação ocorra, a memória do PC1 foi incrementada, de maneira a diminuir a disputa por recursos entre a tarefa de monitoração e as outras que compartilham o mesmo PC1.

Figura 46 - Janela da aplicação de verificação sobre a IHM, apontando uma falha insegura.

O programa desenvolvido em Linguagem C é executado em um *loop* infinito, até a ocorrência de uma falha insegura. Ele lê todas as variáveis utilizadas pelo simulador e através de equações booleanas efetua a verificação de falhas inseguras. Caso ocorra uma falha insegura ocorre uma quebra do *loop* infinito e o tempo decorrido de execução do programa é calculado. Exemplos de leitura das variáveis seguem a seguir:

```
/*Leitura dos dados*/
H1_PA = inb(0x300);
```

```
/*Leitura das ocupações dos circuitos de via*/
A1 = getBit (H1_PC, 1);
```

```
/*Leitura dos alinhamentos de rotas*/
X01_AB = (getBit (H2_PB, 1) && !getBit (H2_PB, 2) && !getBit (H2_PB, 4)) || ( !getBit (H2_PB, 1) && !getBit (H2_PB, 2) && getBit (H2_PB, 4));
```

```
/*Leitura das posições das máquinas de chave*/
X01_NORMAL = !getBit (H3_PC, 1) && !getBit(H3_PC, 2);
```

```
/*Leitura dos códigos de velocidade dos circuitos de via*/
V0_A1 = !getBit (H1_PA, 1) && !getBit (H1_PA, 2);
V50_A1 = getBit (H1_PA, 1) && !getBit (H1_PA, 2);
V75_A1 = !getBit (H1_PA, 1) && getBit (H1_PA, 2);
```

```
V100_A1 = getBit (H1_PA, 1) && getBit (H1_PA, 2);
```

Após a leitura das variáveis, as equações booleanas as utilizarão na verificação da ocorrência de falhas inseguras. As falhas inseguras do sistema foram divididas em quatro tipos: velocidade diferente de zero no intertravamento quando não há rota alinhada, erro no perfil de velocidade, falsa desocupação e movimentação insegura da máquina de chave. Exemplos de booleanas correspondentes às falhas acima descritas são mostradas abaixo:

```
/*Booleana que verifica código de velocidade igual a zero quando não há rota alinhada.*/
if ( !V0_A1 && (!X01_AB || X01_DB))
{
    falha[0] = 1;
    break;
}

/*Booleana que verifica perfil de velocidade*/
if (A1 && (!V0_F1 || (X02_NORMAL && (V75_E1 || V100_E1 || V100_D1))))
{
    falha[1] = 1;
    break;
}

/*Booleana que verifica falsa desocupação*/
if (!A1 && antA1 && !B1)
{
    falha[2] = 1;
    break;
}

/*Booleana que verifica movimentação insegura da máquina de chave*/
if ( !X01_NORMAL && !X01_REVERSO && (B1 || B2 || X01_AB || X01_DC || X01_DB))
{
    falha[3] = 1;
    break;
}
```

Caso venha a ocorrer alguma falha insegura, o *loop* infinito é desfeito e, através do vetor de falha, é possível identificar a falha insegura ocorrida:

```
if( falha[0] )
printf("Rota não alinhada e velocidade diferente de zero no intertravamento\n\n");
if( falha[1] )
    printf("Falha insegura no perfil de velocidade\n\n");
if( falha[2] )
    printf("Falsa desocupação\n\n");
if( falha[3] )
    printf("Movimentação insegura da máquina de chave\n\n");
```

O tempo de execução do programa é calculado através da função `clock()` da biblioteca `time.h`. A função `clock()` retorna o número de pulsos de *clock* que foi dado ao processador. Logo, conhecida a taxa de pulsos por segundo é possível saber

quanto tempo foi gasto desde o início da execução do *loop* até a ocorrência de uma falha insegura. Assim o tempo de execução do programa até encontrar uma falha insegura é calculado da seguinte forma:

```
t0 = clock(); /*guarda o número de pulsos de clock inicial.
while (1)
{
    Leitura das variáveis;
    Verificação de falhas inseguras;
}
tf = clock();
tempo_gasto = ( (double) (tf - t0) ) / CLOCKS_PER_SEC;
```

5.3.5 AS APLICAÇÕES-ALVO DO TESTE

Esses são os *softwares* que são executados na placa do microcontrolador AVR, *hardware* especializado conectado a estação de testes por 54 linhas independentes de sinais (53 mais uma da linha de *reset*, que não ocupa *port* lógico do microcontrolador). Os *softwares* desse módulo são o alvo de todos os testes teóricos desenvolvidos nesse trabalho. É nele que são introduzidas e retiradas, seletivamente, todas as técnicas de programação defensivas segundo o experimento.

5.3.5.1 DETERMINAÇÃO DA LINGUAGEM DE PROGRAMAÇÃO DO PROGRAMA-ALVO

Há diversos critérios para a escolha de uma linguagem para uma aplicação em tempo real e muito já se publicou a esse respeito. Para essa aplicação o objetivo foi compreender a extensão dos conceitos aplicados na norma que estabelece os critérios de *Safety Integrity Levels*, a IEC61508 (*INTERNATIONAL ELECTROTECHNICAL COMMISSION*, 2005). Essa norma recomenda altamente a aplicação da Linguagem ADA (e não recomenda a aplicação da Linguagem C) para o desenvolvimento de sistemas críticos.

O estudo das vantagens reais das diversas linguagens e/ou compiladores para o nível de integridade de segurança de sistemas críticos ainda está por ser pesquisada.

Por conveniência, foi utilizado o PASCAL no experimento desta tese. O compilador empregado foi o *MikroPASCAL for AVR*, da empresa Mikroelektronika⁵³.

⁵³ <http://www.mikroe.com/en/compilers/mikroPASCAL/avr/>

5.3.5.2 O NÚCLEO BÁSICO DAS APLICAÇÕES ALVO

As técnicas de programação defensiva são aplicadas ao programa que tem por objetivo o controle de tráfego dos trens no simulador. Basicamente este programa lê as variáveis de entrada vindas do simulador, executa as booleanas necessárias para o controle de tráfego e retorna as variáveis de saída que são utilizadas pelo simulador.

As variáveis de entrada consistem em ocupações de circuitos de via, comandos de rota e posição de máquinas de chave, já as variáveis de saída consistem em códigos de velocidade dos circuitos de via, informação de posicionamento das máquinas de chave e comando para as máquinas de chave.

Após receber os dados vindos do simulador, as variáveis de entrada são armazenadas e em seguida booleanas são executadas. Essas booleanas de controle são responsáveis pelo correto código de velocidade nos circuitos de via e possíveis alinhamentos de rota. Após serem executadas as booleanas, os valores das variáveis de saída são enviadas ao simulador. O programa básico para o controle de tráfego tem a seguinte forma:

```
while (true)
begin
    varreEntradas; //Atualiza os valores das variáveis de entrada
                    com os dados vindos do simulador.
    Execução das booleanas relativas aos códigos de velocidade dos circuitos de via.
    Execução das booleanas relativas aos alinhamentos de rotas.
    varreSaídas; //Envia ao simulador os valores dos dados
                 relativos às variáveis de saída.
end;
```

5.3.5.3 DUPLICAÇÃO DE VARIÁVEIS.

Três programas foram feitos utilizando a técnica de duplicação de variáveis. No primeiro programa cada variável de entrada e saída foi duplicada. Na leitura, cada dado de entrada é guardado em duas posições de memória e depois comparados para sua validação. Caso ocorra uma diferença entre os dados de entrada, uma variável falha é ativada e no final do programa um estado Seguro é imposto caso seja detectada uma falha ao longo do programa. Assim a validação do dado de entrada fica da seguinte forma:

```
A1_ocupacao1:= PORTB.7;
A2_ocupacao2:= PORTB.7;
if (not(A1_ocupacao1 xor A1_ocupacao2)) then A1_ocupacao:= A1_ocupacao1;
else falha:= true;
if falha then Seguro;
```

No caso das variáveis de saída, o resultado de cada booleana é guardado em duas posições de memória, que depois são comparadas para sua validação. Caso ocorra uma diferença entre os resultados das booleanas, uma variável falha é ativada e no final do programa um estado Seguro é imposto caso seja detectada uma falha. Assim a validação do dado de saída fica da seguinte forma:

```
if (X02_rota_comando_LSB and not X02_rota_comando_MSB and not E1_ocupacao and not
F2_ocupacao and not (X02_AB_alinhada or X02_DC_alinhada)) then
begin
    X02_chave_comando1:=false;
    X02_chave_comando2:=false;
end;
if (not(X02_chave_comando1 xor X02_chave_comando2)) then X02_chave_comando:=
X02_chave_comando1;
else
falha:= true;
if falha then
Seguro;
```

O estado Seguro impõe nas saídas códigos de velocidade nulos, impede o movimento das máquinas de chaves e alinhamentos de rotas.

No segundo programa cada variável de entrada e saída foi duplicada. Na leitura, cada dado de entrada é guardado em duas posições de memória e depois, através de uma equação lógica, ocorre sua validação. Assim a validação do dado de entrada fica da seguinte forma:

```
A1_ocupacao1:= PORTB.7;
A2_ocupacao2:= PORTB.7;
A1_ocupacao:= A1_ocupacao1 or (A1_ocupacao1 xor A1_ocupacao2);
```

Quando A1_ocupacao1 for igual a A2_ocupacao2 ocorre a validação do dado de entrada, caso contrário é imposto o valor verdadeiro para A1_ocupacao.

No caso das variáveis de saída, o resultado de cada booleana é guardado em duas posições de memória e depois, através de uma equação lógica, ocorre sua validação. Assim a validação do dado de saída fica da seguinte forma:

```
if (X02_rota_comando_LSB and not X02_rota_comando_MSB and not E1_ocupacao and not
F2_ocupacao and not (X02_AB_alinhada or X02_DC_alinhada)) then
begin
    X02_chave_comando1:=false;
    X02_chave_comando2:=false;
end;
X02_chave_comando:= X02_chave_comando1 or (X02_chave_comando1
xor X02_chave_comando2);
```

Quando X02_chave_comando1 for igual a X02_chave_comando2 ocorre a validação do dado de saída, caso contrário é imposto o valor verdadeiro para X02_chave_comando.

No terceiro programa usa-se a mesma lógica do segundo programa, sem a utilização do estado Seguro, porém todas as variáveis do programa foram duplicadas.

5.3.5.4 BLOCK RECOVERY, TIPO FORWARD RECOVERY.

Neste experimento a verificação da falha é feita no próprio programa. Após todas as booleanas serem calculadas, as saídas são verificadas através do VerificadorFalha.

O VerificadorFalha irá verificar a consistência dos dados de saída, analisando se há falhas como: velocidade não nula no intertravamento quando não houver rota alinhada, erro no perfil de velocidade, falsa desocupação e movimentação insegura das máquinas de chave.

Caso exista alguma falha, o verificador torna verdadeiro a variável falha e no final do programa um estado Seguro para todas as variáveis é imposto nas saídas.

Caso não existam falhas, os valores das variáveis de saída são enviadas normalmente.

```
VerificadorFalha; //Seta a variável falha caso ocorra uma falha insegura.
if falha then
    Seguro;        //Estado de segurança para todas as variáveis de saída.
    VarreSaidas;   //Envia os dados de saída correspondentes aos resultados das
                  booleanas.
```

O estado Seguro impõe nas saídas códigos de velocidade nulos, impede o movimento das máquinas de chaves e alinhamentos de rotas. Assim são enviados dados seguros para o simulador.

5.3.5.5 BLOCK RECOVERY, TIPO BACK RECOVERY.

Neste experimento a verificação da falha é feita no próprio programa. Após todas as booleanas serem calculadas, as saídas são verificadas através do VerificadorFalha.

O VerificadorFalha verifica a consistência dos dados de saída, analisando se há falhas como: velocidade não nula no intertravamento quando não houver rota alinhada, erro no perfil de velocidade, falsa desocupação e movimentação insegura das máquinas de chave.

A cada iteração, o sistema armazena todos os valores das saídas na memória. Com isso a cada iteração é possível obter o estado atual das variáveis de saída e seu estado anterior.

Caso exista alguma falha, o verificador torna verdadeiro a variável falha e no final do programa o estado anterior é restabelecido e utilizado como novo “Start Up” para o sistema, visto que ele é seguro.

Caso não existam falhas os valores das variáveis de saída são enviadas normalmente.

```

VerificadorFalha;      //Seta a variável falha caso ocorra erro nos valores
                        das saídas.
if falha then
EstadoAnterior;      //Estado anterior e seguro para todas as variáveis de saída.
VarreSaidas;          //Envia os dados de saída correspondentes aos
                        resultados das booleanas.

```

5.3.5.6 DIVERSIDADE.

Neste experimento cada booleana do programa que define o valor de uma variável de saída, é executada duas vezes.

Os valores são guardados nas variáveis temporárias div1 e div2. Após serem calculadas, essas duas variáveis são comparadas para a validação da saída.

Foram feitos dois programas utilizando a técnica de diversidade. No primeiro programa, caso as booleanas sejam iguais, a variável de saída é validada, caso contrário um estado Seguro é imposto para os valores das variáveis de saída. No segundo programa, caso as booleanas sejam iguais, a variável de saída é validada, caso contrário não há alteração da variável de saída.

No exemplo seguinte tem-se a booleana que define o comando da chave X01. Depois de calculada duas vezes, há uma comparação e caso haja discrepância um estado Seguro é imposto para as variáveis de saída, no caso do primeiro programa. Para o segundo programa a saída não é validada e não existe o estado Seguro.

Assim a validação do dado de saída para o primeiro programa fica da seguinte forma:

```

div1:= not X01_rota_comando_LSB and not X01_rota_comando_MSB and
        not B1_ocupacao and not B2_ocupacao and not X01_DB_alinhada;
div2:= not X01_rota_comando_LSB and not X01_rota_comando_MSB and
        not B1_ocupacao and not B2_ocupacao and not X01_DB_alinhada;
if (div1 xor div2) then
    Seguro;

```



```

if (not(div1 xor div2)) then
    X01_chave_comando:= div1;

```

O estado Seguro impõe nas saídas códigos de velocidade nulos, impede o movimento das máquinas de chaves e alinhamentos de rotas. Assim são enviados dados seguros para o simulador.

A validação do dado de saída para o segundo programa fica da seguinte forma:

```

div1:= not X01_rota_comando_LSB and not X01_rota_comando_MSB and
        not B1_ocupacao and not B2_ocupacao and not X01_DB_alinhada;
div2:= not X01_rota_comando_LSB and not X01_rota_comando_MSB and
        not B1_ocupacao and not B2_ocupacao and not X01_DB_alinhada;
if (not(div1 xor div2)) then
    X01_chave_comando:= div1;

```

5.3.5.7 POLINÔMIO DE RECUPERAÇÃO TIPO BCH.

Neste experimento os dados de entrada são guardados com informações de recuperação de corrupção tipo BCH. As portas de entrada possuem valores contidos em um *byte*. Assim um PORT de entrada é da forma 0xVV (V em hexadecimal).

Após serem lidos todos os *ports* de entrada, os dados contidos são codificados através do algoritmo BCH. Dado o *byte* de entrada correspondente ao *Port*, o algoritmo BCH gera um outro byte que é acrescentado ao dado de entrada, assim o nosso novo dado é da forma 0xCCVV (C e V em hexadecimal). O valor 0xCCVV corresponde à codificação do byte de entrada através do algoritmo BCH. O algoritmo BCH garante que depois de codificado o dado, qualquer alteração ou perda de algum dos *bits* ao longo do programa, o dado de entrada é restaurado perfeitamente na decodificação.

Assim o programa codifica os valores dos PORTB, PORTF e PORTG no início do programa e ao longo da execução de cada booleana decodifica o código para corrigir possíveis erros dos valores dos PORT. Uma simplificação da aplicação da técnica é mostrada a seguir:

```

encode; //Codifica os valores dos PORT de entrada.
reload; //Decodifica os valores dos PORT, corrigindo uma possível alteração dos dados de
        entrada e atualiza os valores das variáveis.
if (not X02_rota_comando_LSB and X02_rota_comando_MSB and not
    E1_ocupacao and not F2_ocupacao and not X02_DB_alinhada) then
begin
    X02_chave_comando:= true;
end; // Execução da booleana.

```

5.3.5.8 CHECK SUM.

Neste experimento, na operação de leitura no início do programa, é realizada a soma binária dos valores dos dados de entrada e, ao longo do programa, é feita uma verificação dessa soma (*Check Sum*) para a validação do dado de saída obtido através de uma booleana. Dois experimentos foram feitos utilizando esta técnica. No primeiro programa, caso a verificação não seja válida, um estado Seguro é imposto para todas as variáveis de saída. No segundo programa, caso a verificação não seja válida, as variáveis de saída não são alteradas.

O resultado da soma foi armazenado em uma variável temporária *sum*. Antes de executar cada booleana do programa, a soma dos dados de entrada é recalculada e comparada com a variável *sum*. Desta forma, é possível detectar possíveis alterações da área de dados e impor um estado seguro até que se estabeleça um novo estado estável.

Assim a validação do dado de saída para o primeiro programa fica da seguinte forma:

```

Procedure checksum;
var temp:integer;
begin
    temp:= PortaB + PortaF + PortaG;
    if(temp <> sum) then Seguro;
end; //Recalcula a soma da área de dados e compara com a obtida no início do programa.
sum:= PortaB + PortaF + PortaG;
checksum; //Chama a procedure checksum.
if (not X01_rota_comando_LSB and not X01_rota_comando_MSB and not B1_ocupacao and
not B2_ocupacao and not X01_DB_alinhada)then
begin
    X01_chave_comando:=true;
end; //Execução da booleana.

```

O estado seguro impõe nas saídas códigos de velocidade nulos, impede o movimento das máquinas de chave e impede alinhamentos de rotas. Assim são enviados dados seguros ao simulador. A validação do dado de saída para o segundo programa fica da seguinte forma:

```

Procedure checksum;
var temp:integer;
begin
    temp:= PortaB + PortaF + PortaG;
    if(temp <> sum) then falha:= true;
end; //Recalcula a soma da área de dados e compara com a obtida no início do programa.
sum:= PortaB + PortaF + PortaG;
checksum; //Chama a procedure checksum.
if (not X01_rota_comando_LSB and not X01_rota_comando_MSB and not B1_ocupacao and
not B2_ocupacao and not X01_DB_alinhada)then

```

```

begin
    X01_chave_comando:=true;
end; //Execução da booleana.
if not falha then
    VarreSaidas;

```

5.3.5.9 PARIDADE

Neste experimento, na operação de leitura no início do programa é gerado um *byte* que será acrescentado ao dado de entrada. Esse *byte* corresponderá à paridade do mesmo. Dessa forma o dado que será analisado ao longo do programa pode ser escrito na forma 0xPPVV (PP = paridade, VV = valor).

O dado de entrada é composto por um byte que pode ser escrito na forma 0xVV (VV = valor). No início do programa o dado é codificado: verifica-se o número de *bits* '1' e, caso o número seja par, o dado resultante é escrito na forma 0xFFVV e caso contrário 0x00VV.

Antes de executar cada booleana do programa, é recalculado o *bit* de paridade e comparado com o obtido no início do programa. Desta forma é possível detectar possíveis alterações da área de dados e impedir a validação dos dados de saída.

Dois programas foram feitos utilizando esta técnica. No primeiro programa, caso não haja uma consistência na análise da paridade, um estado Seguro é imposto para as variáveis de saída. Para o segundo programa a saída não é validada e não existe o estado Seguro. Assim a validação do dado de saída fica da seguinte forma:

```

getParidade; // gera o bit de paridade equivalente aos valores dos PORT de entrada.
reload; // recarrega as variáveis de entrada através da consistência na análise da paridade.
if (not X02_rota_comando_LSB and X02_rota_comando_MSB and not
E1_ocupacao and not F2_ocupacao and not X02_DB_alinhada) then
begin
    X02_chave_comando:= true;
end; // Execução da booleana.

```

5.3.5.10 DETERMINAÇÃO DA CARGA DE TRABALHO – *WORKLOAD* SIGNIFICATIVO

Parece intuitivo que a simples introdução de técnicas de programação defensiva no código a ser protegido tem a tendência de diminuir a disponibilidade do código, pela simples razão de que passa a haver mais código. Por outro lado, o código acrescentado tem carácter protetor e, portanto, a tendência de diminuir a susceptibilidade do sistema a falhas inseguras. A partir deste *trade off* é que

pretendemos avaliar a eficácia das técnicas de programação defensiva em estudo nesta tese.

Por outro lado, pode suceder que um programa maior ou menor (e, portanto com uma carga de trabalho maior ou menor) possa se beneficiar de maneira diferenciada das vantagens oferecidas pelas técnicas de programação defensiva. Dessa maneira, este trabalho representa o comportamento dessas técnicas sob a ótica das aplicações produzidas para o experimento, sendo prematuro generalizar estes resultados.

O aprofundamento da influência do *workload* no comportamento conjunto ou isolado das técnicas de programação defensiva é uma necessidade para a validação dessas técnicas, embora fora do escopo e da contribuição projetados para este trabalho de pesquisa.

O resultado de experimentos realizados por FOLKESSON (1999), indicam que variações na carga de trabalho (*workload*) do domínio de teste afetam significativamente o erro de cobertura estimado para cada programa sob testes.

O objetivo de utilizar técnicas de injeção de falhas é o de poder admitir uma cobertura de erro estimado c que possa ser utilizada em um modelo analítico para avaliar numericamente a segurança associada ao sistema como um todo. A cobertura dos erros pode ser então calculada com base nos fatores de cobertura obtidos para diferentes seqüências de entradas usando uma soma ponderada: $c = \sum_{i=1}^n w_i \cdot c_i$, onde c_i é a cobertura estimada para a seqüência de entrada i e o fator de ponderação w_i é a probabilidade de ocorrência do erro para a seqüência de entrada i durante a operação real do sistema.

Infelizmente há uma série de problemas práticos envolvidos na estimativa de cobertura de erros realizada desta maneira. Primeiramente, é difícil determinar a distribuição de probabilidade das seqüências de entrada que ocorrerão em uma operação real. Em segundo lugar, o número de seqüências de entrada possíveis em um sistema real é muito grande, o que torna impossível realizar experimentos de injeção de falhas de maneira a estimar fatores de cobertura para cada seqüência de entrada.

6 REALIZAÇÃO EXPERIMENTAL

6.1 O PROCESSO DE DESENVOLVIMENTO

O trabalho foi desenvolvido em diversas etapas sequenciais, e que estão descritas a seguir:

6.1.1 DEFINIÇÃO E FORMALIZAÇÃO DOS SINAIS DE INTERFACE ENTRE OS MÓDULOS DE SOFTWARE E HARDWARE

Nesta etapa, foram definidos todos os sinais de comunicação entre os diversos módulos. A maior dificuldade foi motivada pelo fato de que se objetivava ter todos os comandos e indicações transmitidos por sinais de *hardware* individuais, representados por fios simples para cada sinal. A razão para esta opção é a de que a o uso de controladores seriais, como Ethernet, RS232, etc., não é recomendado para este tipo de aplicação, já que no momento de injeção de falhas, uma falha simples neste controlador inviabilizaria completamente a visão do que se passa no sistema sob teste.

Desta forma foi necessário compatibilizar a necessidade de construir um sistema complexo o suficiente para ser significativo sob o ponto de vista da representação de um sistema real, mas que, ao mesmo tempo, fizesse uso de um número de bits de interface compatíveis com os microcontroladores existentes no mercado. A hipótese de emprego de controladores de entrada e saída externos ao microcontrolador foi descartada para manter o conceito de ter todo o controlador, que representa um sistema de sinalização real, encerrado em um mesmo chip, objeto dos procedimentos de injeção de falhas.

6.1.2 MONTAGEM E TESTES DA CONFIGURAÇÃO DO HARDWARE DA ESTAÇÃO DE TESTES

Para a montagem do *hardware*, foi escolhida a placa de desenvolvimento já descrita, de fabricação nacional. Entretanto, esta placa não foi fornecida com os diagramas esquemáticos pelo fornecedor, que considerava esta documentação como de sua propriedade industrial. Esta foi uma dificuldade inicial, somente resolvida com o levantamento deste diagrama esquemático por observação da placa de circuito impresso fornecida. A partir do momento em que se dominou a uso desta placa com as ferramentas de *software*, passou-se ao processo de conexão com a interface de entrada e saída instalada no PC1. Esta conexão foi efetuada através de

cabos multivias providos de conectores (prensados, no padrão dos discos duros normais de PCs) no lado da interface e soldados a conectores lineares do lado da placa de desenvolvimento. Além dos 53 sinais de entrada e saída foram providas linhas especiais para reset, alimentação em 5Vcc e diversos terras, misturados aos sinais de maneira a diminuir a possibilidade de introdução de interferências por indução na cabeação.

6.1.3 DESENVOLVIMENTO E TESTES DO MÓDULO DE SOFTWARE DO SIMULADOR

O *software* do simulador de campo foi inicialmente concebido para ser desenvolvido em Linguagem C. De fato, uma versão efetiva chegou a funcionar nesta linguagem. Entretanto esta versão não foi documentada adequadamente e a transferência do seu desenvolvedor para o exterior tornou esta versão inútil. Uma nova versão foi desenvolvida em Linguagem JAVA, documentada formalmente, e empregada em todos os testes descritos nesta tese. O motivo da escolha da Linguagem JAVA para esta segunda abordagem visava facilitar o interfaceamento com a interface homem máquina, também construída nesta linguagem. É interessante lembrar que este simulador replica as condições físicas de campo de um sistema metroviário real, através do emprego das equações cinemáticas para a determinação da posição dinâmica dos trens nas vias.

6.1.4 DESENVOLVIMENTO E TESTES DO MÓDULO DE SOFTWARE DA IHM (JAVA)

Apesar de ser o programa, entre todos os desenvolvidos, com mais apelo visual, a interface homem máquina não trouxe nenhuma dificuldade de grande porte. Isto se deve ao fato de que o seu desenvolvimento foi realizado por um elemento do grupo, Alexander J. Zumalde, com grande experiência no desenvolvimento de simuladores para sistemas metroferroviários, destinados ao treinamento de operadores e técnicos de manutenção. As suas características funcionais estão descritas em outras partes desta tese.

6.1.5 DESENVOLVIMENTO E TESTES DO MÓDULO DE SOFTWARE DO MICROCONTROLADOR (ADA, C E PASCAL), NA VERSÃO REFERÊNCIA, SEM EMPREGO DE TÉCNICAS DE PROGRAMAÇÃO DEFENSIVA

Seguindo as recomendações de programação para sistemas críticos apresentadas por Burns e Wellings (1991), a primeira abordagem adotada foi a de construir as aplicações em Linguagem ADA. O objetivo era abstrair as limitações

inerentes das outras linguagens. Desta maneira, utilizando-se uma linguagem nativamente mais robusta, seria possível se ater a eficácia intrínseca das técnicas de programação defensiva em estudo.

Entretanto, não foi possível obter-se um compilador ADA comercial para o microcontrolador escolhido. O compilador obtido foi um compilador experimental, com fontes abertos liberados sob licença de *software* livre, denominado AVR-ADA⁵⁴.

Foram realizados diversos experimentos com este compilador. Entretanto, houve diversas dificuldades em seu uso. Primeiramente, trata-se de um compilador isolado, com interface por linha de comando e saída de erros por gravação de arquivo. Este conjunto, diferentemente de um sistema de desenvolvimento integrado moderno, torna as operações de depuração bastante morosas e improdutivas. Além deste fato, as experiências com este compilador aconteceram concomitantemente com o teste integrado da estação de testes, que implicou em falhas múltiplas. Nestas ocorrências de desenvolvimento ficava impossível determinar a origem dos não funcionamentos encontrados. Desta maneira, havemos por bem fixar pontos e esta ferramenta foi abandonada em detrimento de um compilador moderno com sistema de desenvolvimento integrado.

Esta ação, entretanto, implicou no emprego de uma nova linguagem, pela simples inexistência de uma ferramenta adequada para a Linguagem ADA. Optou-se então pelo emprego da Linguagem PASCAL, origem da Linguagem ADA e que apresenta uma estrutura formal de tipos bastante robusta.

Neste ponto foi experimentado um compilador da Linguagem C, que apesar de funcional, também foi descartado em função da estrutura de tipos frágil desta linguagem.

O desenvolvimento se iniciou com uma distribuição gratuita, de maneira a testar os conceitos. Após esta validação foi adquirida uma licença de desenvolvimento do compilador, com seu ambiente de desenvolvimento integrado, mikroPascal, da empresa mikroElektronika⁵⁵.

Este compilador se mostrou uma poderosa ferramenta de produtividade e seu uso tornou viável o projeto dentro da disponibilidade temporal existente. A Figura 47 ilustra a interface gráfica do sistema do sistema de desenvolvimento adotado.

⁵⁴ <http://avr-ada.sourceforge.net/>

⁵⁵ <http://www.mikroe.com/en/compilers/mikrobasic/avr/>

Figura 47 - Interface do sistema de desenvolvimento integrado do compilador mikroPascal.

O produto final desta etapa foi uma versão funcional do controlador, executando equações booleanas que encerram as regras de movimentação do ambiente de testes escolhido, que simula uma via ferroviária real, com duas regiões de manobra, circuitos de via com 4 códigos de velocidade e dois trens em circulação.

6.1.6 TESTE FUNCIONAL DA ESTAÇÃO DE TESTES

Nesta fase foi realizado o teste integral da estação de testes, com a interface homem máquina gerando os comandos, que são enviados ao microcontrolador via a interface de entradas e saídas do PC1. O microcontrolador calcula as ações e envia comandos ao simulador, que, sendo executado no mesmo PC1, recebe os comandos e simula as ações, enviando o resultado simultaneamente para a interface homem máquina, que esta sendo executada no mesmo PC1 (via passagem de parâmetros) e para o microcontrolador (via interface de entradas e saídas).

O resultado desta fase foi um sistema simulado funcional, com comandos e parâmetros programáveis.

6.1.7 DESENVOLVIMENTO DO SISTEMA DE INJEÇÃO DE FALHAS

Nesta fase foi concebido e construído o sistema de injeção de falhas. De maneira diferente das fases anteriores, esta fase demandou a construção experimental de dispositivos mecânicos, principalmente o desenvolvimento dos sistemas irradiantes adequados. Conforme descrito anteriormente, esta foi a fase mais delicada do projeto, já que nas fases iniciais do processo de experimentação não foi possível obter o processo de injeção de falhas mais adequado. Após um grande processo experimental, com diversas frustrações de expectativa descritas anteriormente foi possível construir e experimentar a stripline que tornou possível a realização dos experimentos descritos.

Uma vez obtido o padrão de radiofrequência adequado: frequência, potência e modulação, foi necessário automatizar o experimento para auferir um fluxo adequado à tomada de dados.

Para isto foi desenvolvida uma pequena aplicação em Linguagem C, executada no PC2, que cria o padrão modulante e o transmite ao radiotransmissor adotado. Esta aplicação é disparada e parada pela tarefa de verificação de falhas que automatiza o experimento.

Nesta etapa foi desenvolvida a tarefa de *software* que verifica a ocorrência de falhas inseguras. Esta tarefa, ao ser executada, zera um relógio de tempo real e dispara um *bit* que, enviado via placas de entradas e saídas dos PCs 1(EISA) e 2(USB), inicia o processo de injeção de falhas. A partir deste ponto a tarefa inicia a monitoração de todas as linhas de dados da interface de entradas e saídas do PC2, procurando por combinações correspondentes à ocorrência de falhas inseguras. Assim que uma combinação insegura é encontrada, a tarefa para o relógio, para o processo de modulação do PC2 e informa em sua console o tempo e a natureza da falha insegura detectada.

A partir da conclusão desta fase era possível introduzir falhas no programa sem nenhuma técnica de programação defensiva, o Exp_1, medindo-se o tempo até a ocorrência da falha insegura.

6.1.8 DESENVOLVIMENTO E TESTES DE CADA VERSÃO DO MÓDULO DE SOFTWARE DO MICROCONTROLADOR (PASCAL).

Esta é a parte mais sensível de todos os *softwares* desenvolvidos para o projeto, já que todas as técnicas de programação defensiva, objeto de estudo desta tese, são introduzidas neste programa.

É importante ressaltar que o processo de desenvolvimento destas versões foi bastante complicado. O experimento tem muitas interações no domínio do tempo, o que implica em adequar cada uma das versões para que funcione adequadamente.

Para isto foram desenvolvidas treze versões de *software*, contendo as técnicas de programação defensiva descritas anteriormente. Foram efetuados testes sem a injeção de falhas, e, ao fim desta fase havia 13 versões de software completamente operacionais empregando as técnicas descritas.

A partir deste ponto foi realizado o teste de cada uma das versões contra o injetor de falhas e realizada a coleta de dados. Em conclusão, foi realizada a formatação, a análise dos dados e a elaboração das conclusões, que passaremos a descrever.

6.2 PROCEDIMENTO

O experimento foi concebido para realizar diversas séries de testes, reunindo o conjunto de 7 técnicas escolhidas para avaliação, distribuídas nas treze versões de programa já descritas. Cada um dos treze programas, contendo combinações das sete técnicas de programação defensiva escolhidas, foi exercitado individualmente, seguindo o seguinte procedimento:

- Em primeiro lugar, uma nova técnica é introduzida no código de segurança do processador AVR;
- A seguir a plataforma de testes é colocada em funcionamento e, sincronizadamente, inicia-se a introdução de falhas através da irradiação por RF no padrão temporal escolhido e executa-se a tarefa de verificação, zerando e disparando o seu cronômetro;

- No momento do evento de uma falha insegura o cronômetro é parado e o tempo t_f registrado, repetindo o procedimento por 50 vezes até o final da série⁵⁶;
- Os 50 valores foram tabulados e colocados em um gráfico, gerando uma informação como a da Figura 48, para cada uma das 13 séries.

Figura 48 - Resultado de uma série de testes.

6.3 RESULTADOS OBTIDOS

Os gráficos do APENDICE A ilustram a sensibilidade do sistema sob teste com relação à introdução das diversas técnicas de programação defensiva no núcleo de execução da aplicação crítica que representa a carga de trabalho. A Figura 49 ilustra a forma desta comparação.

Por esta ilustração pode-se observar que a técnica de programação defensiva 1 apresentou um tempo médio t_{f1} até apresentar uma falha insegura detectada pela tarefa de avaliação.

De maneira análoga pode-se inferir que a técnica 2 produziu uma melhor imunidade quanto a produção de falhas inseguras, na medida em que a aplicação demorou um tempo $t_{f2} > t_{f1}$ para apresentar uma falha insegura. A diferença, anotada na Figura 49, dá uma idéia comparativa do quanto melhor a técnica 2 é em relação ao emprego da técnica 1.

⁵⁶ Embora a tarefa de verificação informe a natureza das falhas ocorridas, estas não foram registradas. Este registro poderia dar boas informações sobre a aleatoriedade do processo e é uma sugestão para trabalhos futuros.

Embora esta avaliação seja relativa, ela poderia ser empregada como uma avaliação de um padrão absoluto, se for possível criar um padrão único de referência, como no caso dos decibéis do tipo dBA ou dBm.

Figura 49 - A técnica de comparação das séries de testes.

Só foi possível executar a comparação efetiva das técnicas de programação defensiva após meses de ensaio com as técnicas de injeção de falhas. A determinação do conjunto correto de sistema irradiante, potência aplicada e frequência de trabalho só atingiu o estado adequado após centenas de testes e variações.

Foi fundamental compreender os efeitos de “inundar” o dispositivo semicondutor de falhas, compreendendo a forma como isto mascarava a ação das técnicas de programação defensiva. A Tabela 18 ilustra algumas medidas efetuadas

com a bobina ilustrada na Figura 39, em que se conseguiu o confinamento total dos campos irradiantes. Aparentemente, os mecanismos de proteção de falhas também estão falhando e, portanto, a taxa de falhas inseguras produzida é apenas proporcional ao tamanho efetivo do código: quanto maior, maior a chance de falhar.

Tabela 18 - Dados erráticos no caso de sobrecarga de falhas no semicondutor.

Versão <i>software</i>	Exp_dupla6	Exp_dupla7	Exp_1	ExpBCH
Tempo [s]	0,6	0,6	3,0	0,6
	1,5	0,9	0,6	0,6
	0,6	0,6	0,9	4,2
	13,5	3,6	0,6	0,3
	3,9	4,5	3,6	2,7
	0,3	1,8	9,0	4,8
	0,3	0,6	3,6	2,1
	0,6	0,6	0,6	1,8
	6,9	9,6	0,9	3,3
	0,6	1,8	0,9	0,6
Média	3,1	2,5	2,4	2,1
Desvio padrão	4,5	2,9	2,7	1,6
variança	17,8	7,35	6,3	2,4

É possível notar que estes dados são indistinguíveis do ponto de vista estatístico, com o programa Exp_dupla7, que duplica todas as variáveis do programa, exibindo um improvável mesmo desempenho que os demais.

Neste processo de investigação, procurou-se aumentar o significado estatístico através da obtenção de mais dados, já que apenas dez dados poderiam ser uma amostra insuficiente. Obteve-se, então, mais 20 amostras para os programas Exp_1 (sem nenhuma técnica de proteção) e ExpBCH, que implementa a recuperação de dados através do uso do algoritmo BCH, conforme mostrado na Tabela 19.

Tabela 19 - Comparação estendida da Tabela 18 entre as versões sem proteção e com BCH.

Programa	Tempo [s]										$\bar{x}$	δ	σ^2
Exp_1	0,6	2,7	8,7	2,7	0,6	0,6	3,3	8,1	0,6	0,6	2,7	3,0	8,4
	0,6	0,6	3,9	7,5	1,2	2,4	0,6	0,6	0,6	9,9			
ExpBCH	5,4	1,2	3,0	3,9	0,6	0,6	0,6	1,8	4,2	0,6	1,8	1,5	2,1
	1,8	0,6	2,1	0,6	1,8	0,9	0,6	1,2	0,6	0,6			

Os resultados estatísticos da Tabela 19 incorporam os 20 dados dessa tabela mais os dez da Tabela 18 e ainda assim não houve melhora que permitisse a distinção efetiva dos dados. Estes resultados ruins, motivaram as alterações no experimento que, com a introdução da *stripline* da Figura 43 e os ajustes de frequência e potência determinados pelo estudo da Tabela 15, passaram a ter os

resultados esperados com a introdução das técnicas de programação defensiva. A tabela a seguir apresenta a totalidade dos dados que foram analisados:

Tabela 20 - Conjunto de dados, resultado dos experimentos. Tempo para falha insegura [s]

Exp_1	Exp_Back_Recovery	ExpBCH	Exp_div	Exp_dupla	Exp_dupla6	ExpCheckSum	Exp_Div2	Exp_dupla7	ExpCheckSum2	ExpPar	ExpPar2	Exp_For_Recovery
0,3	3,6	1,5	12	2,1	6	1,8	5,1	30,3	35,4	2,1	24	3,3
1,2	9,9	3,6	4,5	5,7	9,3	22,2	36,9	30,3	11,7	2,1	8,4	8,4
2,7	6,6	2,1	13,8	8,1	4,2	2,1	49,5	5,7	37,2	1,2	7,5	30
1,2	15,6	4,8	3,9	7,8	29,1	4,8	27,6	39,6	36,3	1,5	3,6	4,5
0,3	15,9	1,5	12,6	8,4	4,5	18	8,7	33,31	86,4	1,5	1,8	1,5
0,9	8,7	1,5	5,7	8,1	28,2	1,8	48,3	51	7,8	1,8	9,6	11,7
3,6	5,7	15,6	29,1	3	9,6	11,1	3	34,5	50,4	6	20,7	9
1,2	4,8	6,6	17,4	1,5	7,8	8,7	43,2	41,1	20,1	3,6	1,2	40,2
0,3	6	6,3	2,4	1,5	16,5	12	42	28,81	105,9	1,5	42,6	42
0,9	4,5	3,3	5,1	5,1	23,7	6,6	5,4	10,8	8,4	2,7	0,6	2,1
1,2	3,9	1,2	13,2	3,9	5,4	9,9	26,7	28,8	21,9	1,5	0,3	4,2
2,1	7,2	2,4	5,4	3	25,5	6,3	15,6	10,5	39	1,8	4,8	8,7
1,5	12,9	1,5	3,6	2,7	3,3	7,8	33,3	91,8	41,1	1,2	1,2	39
6	3	1,2	2,4	1,2	3,3	2,1	27	86,7	29,1	0,9	0,3	1,5
0,9	4,5	8,7	12,6	28,5	30,3	1,8	7,2	94,5	86,7	0,9	6,6	12
3,3	3,6	21	12,6	6,9	5,1	1,2	41,7	30	19,8	2,4	1,5	19,2
3,3	3,3	19,2	2,7	1,8	10,2	11,1	3	6,6	38,1	2,1	0,6	38,7
0,9	3	1,8	3	1,2	3	2,4	35,4	35,7	36	7,2	4,5	30,3
0,6	31,5	3,3	5,1	8,4	4,8	11,7	39,9	44,7	25,5	3,9	3,3	22,8
2,1	4,8	2,1	13,5	6	24,6	4,5	5,4	46,8	36,61	2,1	6,9	45,6
3	1,8	1,5	3,9	8,4	28,8	4,5	34,5	26,4	9	9	3,6	15,91
2,4	2,7	2,7	3,6	6	26,1	6	5,4	38,1	23,7	3,6	32,4	39,6
0,9	4,8	4,8	10,8	6,6	9,9	3	42	41,7	13,2	0,9	6	35,1
0,6	28,5	8,1	3,9	1,8	19,8	3	6,3	40,2	18,3	3,3	11,7	3,3
3,3	2,4	9,9	4,8	15,6	23,1	2,1	48,9	50,7	32,7	3	2,1	4,5
1,2	3,3	6,3	4,2	1,8	14,4	16,8	32,4	30,3	4,8	4,8	4,5	12,3
0,9	2,1	1,8	2,7	8,7	27	4,2	6	46,5	9	1,5	22,5	20,7
0,3	3	1,2	4,8	3	9,6	28,2	41,1	42,3	27,3	2,7	4,5	42,9
2,4	5,4	6,6	5,7	1,5	3,6	4,2	27,3	47,4	21	2,4	23,1	2,4
1,2	3	10,8	7,5	3,6	4,5	1,5	16,2	38,4	41,4	2,1	18	42,9
1,5	11,1	4,5	5,7	7,2	6,9	2,4	38,4	8,1	24,3	1,2	1,2	45,3
0,6	9,6	5,7	5,7	1,8	24	15,9	41,7	46,8	9,3	2,1	2,7	51,9
1,5	3,3	12	12,3	3,6	14,1	15,6	60,6	43,5	36,3	6,3	19,2	74,1
4,8	3	5,1	16,2	1,5	6	6	4,8	43,21	47,4	1,5	12,3	18,3
1,2	2,7	3,3	3	9,6	30	10,5	37,5	42,6	5,7	0,9	0,9	35,4
3	7,5	11,1	17,1	1,5	8,1	17,7	5,1	38,1	40,2	1,2	25,5	41,7
0,6	12,9	3,6	2,4	3,6	20,4	7,2	43,2	6,3	4,2	2,1	10,5	17,7
0,9	12,6	14,1	7,8	4,5	11,1	17,1	5,7	4,2	39,6	3,9	0,6	40,5
1,8	8,1	5,4	3,3	1,5	4,5	1,2	30,3	84,6	42,3	1,2	1,8	36,3
0,9	3,3	2,4	2,4	2,1	11,4	2,1	39,9	36,3	35,7	2,4	6	3,3

Exp_1	Exp_Back_Recovery	ExpBCH	Exp_div	Exp_dupla	Exp_dupla6	ExpCheckSum	Exp_Div2	Exp_dupla7	ExpCheckSum2	ExpPar	ExpPar2	Exp_For_Recovery
1,2	2,1	5,4	8,1	15	27,6	4,8	6	67,8	7,8	4,5	15,6	42,6
0,9	2,7	3	4,8	5,1	4,5	2,7	16,8	75,3	42,9	2,1	9,6	7,8
7,5	4,2	13,2	14,4	4,8	6,9	1,5	40,8	10,8	3,6	0,6	16,5	24,3
0,6	30,9	9	8,1	6,6	19,2	20,1	4,2	94,8	91,5	0,9	4,2	39,6
2,7	16,8	8,7	12,6	1,2	4,5	3,9	11,1	32,7	45,9	2,4	3	37,8
3	2,1	24,3	12	1,2	8,4	9,6	5,1	10,8	38,4	9	6,6	3,6
0,6	29,4	2,1	28,5	23,7	18,3	26,4	33,6	44,7	33,1	5,4	1,5	46,5
1,2	4,5	2,4	22,5	9	9,9	19,2	5,1	38,4	36,3	0,9	1,8	39,6
0,6	8,4	19,5	4,8	5,8	7,2	9,6	6,6	35,4	12,3	1,8	0,6	5,1
1,5	8,3	8,9	9,9	6,7	15,9	6,7	14,7	30,6	27,5	1,8	1,2	3

A análise mais simples que se pode fazer destes dados está representada na Tabela 21, que apresenta a média, o desvio padrão e a variância de cada uma das técnicas.

Tabela 21 - Análise estatística básica dos dados dos experimentos.

Tempo para falha insegura [s]	Exp_1	Exp_Back_Recovery	ExpBCH	Exp_div	Exp_dupla	Exp_dupla6	ExpCheckSum	Exp_Div2	Exp_dupla7	ExpCheckSum2	ExpPar	ExpPar2	Exp_For_Recovery
média	1,75	7,99	6,53	8,68	5,76	13,60	8,43	24,32	39,57	31,96	2,67	8,39	24,17
desvio	1,47	7,65	5,71	6,43	5,40	9,10	7,04	16,95	22,86	22,55	1,99	9,41	18,05
variância	2,16	58,50	32,64	41,32	29,21	82,89	49,55	287,21	522,44	508,27	3,95	88,61	325,89

Estes dados, quando colocados em um gráfico, fornecem uma boa amostra comparativa entre as técnicas de programação defensiva empregadas em cada versão do programa, conforme ilustrado na Figura 51, que avalia as médias.

Apenas com estes dados já podemos realizar uma análise comparativa das técnicas de programação defensiva. Entretanto, pode-se perceber na Tabela 21 que os desvios padrões obtidos são bastante grandes, para medidas que, na verdade, apresentam uma homogeneidade e reprodutibilidade boas. Esta conclusão inicial pressupunha que a distribuição dos dados fosse uma distribuição normal ou gaussiana, o que não corresponde à realidade em nosso caso.

Há evidências de uma distribuição aleatória de falhas, o que pode ser constatado por simples observação. Quando o microcontrolador era inundado por

falhas pelo uso de conjuntos frequência/potência/sistemas irradiantes inadequados ficava patente que alguns modos de falha eram privilegiados, viciando o processo do experimento. A partir da introdução da *stripline* o comportamento mudou drasticamente, com fortes indicações de aleatoriedade que podem ser observadas refletidas nos dados da Tabela 20. A comprovação mais adequada desta aleatoriedade poderia ser obtida através do emprego de outros microcontroladores, geometricamente diferentes. Em se repedindo os resultados, haveria melhores chances de se afirmar sobre a aleatoriedade desta forma de injeção de falhas.

Antes de efetivamente comparar as médias, é interessante observar a ilustração da Figura 50, que apresenta essa comparação com os desvios padrões computados.

Figura 50 - Comparação dos dados com desvios padrões computados.

Estes desvios padrão e a análise das variâncias, encontradas para cada um dos conjuntos de dados, indicam que a distribuição normal pressuposta pelo emprego da média não deve corresponder adequadamente à realidade.

Entretanto, a análise das médias é muito mais simples de ser efetuada e, se as diferenças entre as técnicas for grande, como indicam os dados, a comparação simples das médias poderia ainda assim fornecer um parâmetro de comparação de simples obtenção, sem a necessidade de emprego de ferramentas computacionais

mais sofisticadas. A Figura 51 apresenta a comparação das médias para as diversas técnicas.

Figura 51 - Análise comparativa das médias das diversas técnicas de programação defensiva: t[s] até a falha insegura.

Para realizar esta comparação, de maneira mais adequada, foi empregado o método dos mínimos quadrados. Foram ajustadas várias distribuições ao conjunto de dados, obtendo os menores chi-quadrados, e portanto a função densidade de probabilidade de falhas inseguras $f(t)$ que melhor define o fenômeno físico observado. A raiz da primeira derivada de $f(t)$, $f'(t) = \frac{df(t)}{dt} = \frac{d^2 F(t)}{dt^2}$ define então um ponto, cuja avaliação promoverá uma comparação muito mais precisa entre as técnicas de programação defensivas.

A razão para o emprego deste máximo reside no simples fato de que ele representa o ponto onde houve uma incidência maior de falhas em um determinado instante ao redor deste ponto, o que leva a suposição de que este tempo é uma característica da técnica de programação defensiva em análise sob as condições do experimento proposto. Na realidade, este ponto representa a posição, característica da técnica, de maior incidência de falhas e corresponde ao seu histograma. A função

densidade de probabilidade de falhas, neste caso, representa foi usada como um histograma onde os intervalos de acúmulo são infinitesimais. A realização deste estudo estatístico foi conduzida para cada uma das 13 séries de dados.

Para o trabalho estatístico optou-se pelo emprego de uma ferramenta comercial de análise de confiabilidade, denominada Weibull++, cujos resultados passaremos a descrever. O APENDICE A apresenta todos os gráficos gerados por este programa para este estudo. A Figura 52 mostra a mesma comparação da Figura 51 em azul e inclui, em marrom, a comparação dos picos das funções densidade de probabilidade de falhas inseguras.

Figura 52 - Comparação através do pico da função densidade de probabilidade de falhas inseguras e das médias.

É interessante notar que, embora a maioria do posicionamento relativo se mantenha, há algumas inversões de posição. Isto se deve provavelmente ao fato de que, como algumas melhores distribuições foram escolhidas pelo programa como sendo exponenciais, fica impreciso determinar o pico. O programa provavelmente escolheu um ponto como pico que não tem o significado físico esperado.

Há algumas possibilidades para superar esta constatação. A primeira seria forçar o programa Weibull++ a considerar todas as distribuições iguais. Se a

natureza física do fenômeno, que estamos observando, for a mesma para todas as versões do programa – o que provavelmente é verdade – então as distribuições devem ser as mesmas. O problema é que esta distribuição do fenômeno pode não ser – e provavelmente não o é – redutível a uma das funções de distribuição elementares (normal, exponencial, etc.).

Neste caso, a alternativa mais plausível seria abrir mão da função densidade de probabilidade de falhas e efetuar as comparações com a área dos gráficos de segurança $S(t)$, ou com a probabilidade de falhas inseguras $F(t)$ em um tempo t determinado e igual para todos os programas. A área sob o gráfico das $S(t)$, mostradas no APENDICE A, representa o tempo médio para falha insegura, já que $MTTUF = \int_0^{\infty} S(t)dt$. Os resultados da aplicação desta técnica estão mostrados na

Figura 53:

Figura 53 - Comparação das médias com o pico das distribuições e o MTTF.

A observação desta figura mostra que agora há uma aderência importante da comparação dos MTTUFs com os dados da média, o que indica que os dados da média podem ser empregados para uma avaliação comparativa das técnicas. É importante verificar que os dados do MTTUF, apresentados no gráfico da Figura 53

foram obtidos para um nível de confiança de 0,95, utilizando a Matriz de Fisher como método do intervalo de confiança.

Uma segunda alternativa seria ajustar para cada conjunto de dados uma função de distribuição de Weibull (homônimo do programa), que é uma distribuição parametrizada, com dois ou três parâmetros. Desta maneira teríamos todas as funções densidade de probabilidade de falhas inseguras ajustadas à mesma função matemática, diferenciadas por três parâmetros α , β e γ . A comparação e a análise destes parâmetros deve trazer contribuições interessantes para a compreensão do fenômeno e deve ser experimentada em trabalhos futuros.

Uma importante constatação da aleatoriedade das falhas obtidas aparece ilustrada na Figura 54. Ela apresenta os tempos para falhar de maneira insegura de cada técnica, colocados no gráfico em conjunto com o tamanho do arquivo .hex de cada uma.

Figura 54 - Tamanho do arquivo em relação a robustez à falhas inseguras.

Quando havia uma “inundação” de falhas no microcontrolador, com as outras técnicas de injeção anteriores ao *stripline*, havia uma correlação entre o tempo para falha e o tamanho do arquivo. Agora, com a injeção de falhas trabalhando na proporção correta, pode-se observar que não há correlação entre o tamanho do arquivo e sua robustez a falhas inseguras. Mesmo nos casos em que a *procedure*

Seguro é empregada, a maior susceptibilidade introduzida pelo aumento do tamanho do código é amplamente compensada pelas qualidades de proteção do mecanismo de programação defensiva adotado.

Esta constatação corrobora, uma vez mais, que foi possível construir um procedimento de injeção de falhas adequado para o experimento projetado.

A Figura 55 mostra os mesmos dados da Figura 51 normalizados, de maneira a que o experimento conduzido com o arquivo Exp_1, que não contém nenhuma técnica de programação defensiva, seja 1. Este gráfico dá uma idéia comparativa do quanto melhor, do ponto de vista da segurança introduzida, cada técnica alcança.

Figura 55 - Comparação normalizada das técnicas de programação defensiva.

A partir desta figura é possível concluir que o programa Exp_For_Recovery, por exemplo, é 7,79 vezes melhor do que o programa Exp_1. Por extrapolação podemos inferir que é provável que o programa Exp_Div seja $4,97/3,74 \approx 1,33$ vezes melhor do que o programa ExpBCH e que por consequência, para as aplicações e condições do experimento, a técnica da diversidade, contida em Exp_Div contribui 33% a mais do que a técnica de recuperação BCH para a segurança da aplicação.

A Figura 56 apresenta um gráfico com as $S(t)$ de cada um dos experimentos apresentada simultaneamente.

Figura 56 - Comparativo das funções Segurança $S(t)$.

É possível observar que as linhas não são absolutamente paralelas, ou seja há pontos de cruzamento. Isto indica que a relação melhor/pior se inverte ao longo do tempo de testes e que portanto uma técnica genérica j que era melhor que k até um instante t passa a ser pior do que k para um determinado instante t' , $t' > t$.

Um fato importante do programa Weibull++ refere-se ao fato de que ele foi desenvolvido para estudos de engenharia da confiabilidade de componentes de

hardware. A possibilidade de seu emprego para a avaliação de programas, trazida pela avaliação numérica dos programas proposto nesta tese, agrega um grande conjunto de ferramentas e técnicas de análise que podem trazer benefícios para o conhecimento da segurança das aplicação de missão crítica como um todo.

Figura 57 - Comparativo das probabilidades de falha insegura $F(t)$.

A Figura 57 mostra igualmente a função $F(t)$, probabilidade de falhas inseguras, que é simplesmente o complemento da Figura 56, tendo em vista que $F(t) = 1 - S(t)$. A partir da derivada de $F(t)$ obtemos $f(t)$ que é a função densidade de probabilidade de falhas inseguras. A comparação de $f(t)$ para as diversas versões dos programas, incluindo as diversas técnicas de programação defensiva, está mostrada na Figura 58.

Figura 58 - Comparativo das densidades de probabilidade de falhas inseguras.

A comparação dos picos destas distribuições é que forneceu os dados para o gráfico comparativo das técnicas mostrado na Figura 52.

A primeira evidência que surge dos dados é a atuação da rotina Seguro, que fica muito clara se tomarmos a Figura 55 como referência. Os programas ExpPar e ExpPar2, por exemplo, fazem uso da mesma técnica de programação defensiva, a paridade. Entretanto, o programa ExpPar, quando encontra uma condição insegura, para o processamento, recompõe todos os estados e só então retoma as atividades. Já o programa ExpPar2, na mesma circunstância, recompõe apenas a variável associada ao problema e segue o processamento. Provavelmente, o *workload* adicional, introduzido pelo procedimento Seguro, aumenta as chances de ocorrência de uma falha insegura, embora o tamanho do código não seja proporcional a esta razão. O problema é que o tamanho do código não é a única variável para o estabelecimento do *workload*, já que a análise das interações e *loops* do programa é fundamental para determinar o tempo de exposição dos processos as falhas.

O mesmo fenômeno se passa com os pares ExpChecksum e ExpChecksum2, Exp_div e Exp_div2 e Exp_dupla e Exp_dupla_2, com efeitos percentualmente diferentes mas conceitualmente iguais. É interessante notar que os efeitos absolutos das técnicas ficam claros, na medida em que, por exemplo, o programa ExpPar2, otimizado e sem o procedimento Seguro, ainda apresenta um resultado pior do que o programa Exp_div, em sua versão mais primitiva e com a rotina Seguro. Esta é uma evidência de que a técnica da diversidade é bastante mais efetiva do que a técnica da paridade.

Na comparação absoluta, o programa Exp_dupla7, que replica todas as variáveis do sistema, apresentou um resultado significativamente melhor, o que pode ser comprovado pelas análise estatísticas dos dados.

Durante os testes, por vezes, o processador travava em uma posição segura. Ou seja, havia sido introduzida uma falha que levou o sistema a um travamento ou *dead lock*, sem que uma condição insegura houvesse sido alcançada. Estas ocorrências foram desprezadas e não aparecem nos dados, já que não são falhas inseguras. Da mesma maneira foram desprezadas as falhas seguras que não provocavam travamento. Nestes casos o teste simplesmente continuava.

7 CONSIDERAÇÕES FINAIS

7.1 CONCLUSÃO

O conjunto de dados avaliado nesta pesquisa está longe de ser exaustivo ou abrangente, embora contribua significativamente para o conhecimento da eficácia das técnicas de programação defensiva na implementação de sistemas críticos de segurança.

O objetivo primordial de desenvolver um método capaz de avaliar quantitativamente o impacto das técnicas de programação defensiva na segurança dos sistemas foi alcançado.

A análise dos dados indica claramente que há técnicas de programação defensiva mais eficientes do que outras.

É interessante notar que algumas técnicas apresentam um resultado muito melhor que as demais. Nos casos em que o *trade off* é vantajoso, estas técnicas devem ser consideradas como essenciais, não devendo ficar fora do desenvolvimento de qualquer sistema de aplicação crítica.

É o caso do programa Exp_dupla7, por exemplo, que replica todas as variáveis do sistema, sem ter um tamanho significativamente maior.

A abordagem apresentada para a comparação dos resultados das diversas técnicas de programação defensiva é estatística, proporcional ao seu comportamento sistêmico global quando submetida a um estímulo interferente igual ao empregado para as outras técnicas. Essa premissa faz supor que, delimitado o espaço amostral e escolhida a dimensão prática adequada para a amostra, o escopo de validade das comparações pode ser estendido a outras soluções funcionando em outros ambientes operacionais, validando a técnica.

De maneira geral podemos concluir que:

- Foi desenvolvido um procedimento experimental para avaliação comparativa das técnicas de programação defensiva em sistemas de missão crítica;
- Foi apresentada uma contribuição empírica para o modelo teórico;

- Foi possível efetivar uma avaliação quantitativa relativa à eficácia de algumas técnicas de programação defensiva no contexto do experimento proposto;
- Foi possível fazer uso de ferramentas existentes da Engenharia da Confiabilidade para a análise numérica de técnicas de *software*.

7.2 RECOMENDAÇÕES PARA PESQUISAS FUTURAS

Há muito espaço para experimentação na área explorada por esta tese. Os itens que se seguem devem ser avaliados sob o ponto de vista do impacto no nível de segurança do sistema crítico avaliado. De maneira resumida pode-se listar:

- Injeção de falhas por partículas carregadas;
- Realização de experimentos combinando os diversos tipos de programação defensiva;
- Ensaiar outras técnicas de programação defensiva:
 - Polinômio de recuperação tipo Reed-Solomon;
 - *Time stamp*;
 - *Pulsed I/O*;
 - etc.
- Realizar os mesmos testes com arquiteturas de microcontroladores diferentes;
- Comparação de linguagens;
- Comparação de compiladores;
- Comparação de sistemas operacionais;
- Uso de rotinas de recuperação em *EPROMs* ou *E²PROMs*, menos susceptíveis às falhas;
- Cálculo dos λ_u (% de falhas inseguras em relação ao número total de falhas em um Δt);
- Emprego de técnicas de ensaios acelerados.

Uma primeira alternativa para prosseguimento das pesquisas seria a realização dos experimentos combinados n a k , ou seja, mais de uma técnica implementada em um mesmo programa simultaneamente. Em nosso caso, por exemplo, o número de técnicas - ou variações de programa - escolhidos inicialmente (na concepção do experimento) foi de 13. Para um número $n=13$ de técnicas, o número de combinações seria $C_k^n = \frac{n!}{k!(n-k)!}$, onde C_k^n é o número de séries de testes, n é o número de técnicas de programação defensiva escolhidas e k é o tamanho dos subconjuntos combinacionais. Para o caso mais simples de combinação duas a duas teríamos $k=2$, que leva a $C_2^n = \frac{n(n-1)}{2}$ e $C_2^{13} = 78$ seria o número total de séries de teste. Como cada série de testes, para ter significado estatístico, deve contar com 50 repetições, como as aqui realizadas, devem ser realizados 3.900 testes individuais, medindo-se um tempo t_f até a falha insegura para cada um deles. Para isto seria necessária a produção de 78 novas versões de programa para a combinação das técnicas. Para a combinação de 3 técnicas de cada vez, por exemplo, os números seriam $C_3^{13} = 572$, e portanto 28.600 testes individuais e 572 versões de *software*.

Esta investigação deve trazer uma contribuição relevante, alinhada com a contribuição postulada por esta tese.

A linguagem e o ambiente de desenvolvimento também podem se beneficiar dessas técnicas para a sua escolha. A norma IEC61508 (*INTERNATIONAL ELECTROTECHNICAL COMMISSION*, 2005), por exemplo, indica algumas linguagens, como a Linguagem C como não recomendada para aplicações de SIL alto. As Linguagens recomendadas, no entanto, ainda não são uma unanimidade, devido aos problemas de custo/benefício que se apresentam nesta escolha.

O mesmo procedimento, que foi adotado aqui para a comparação de técnicas de programação defensiva, poderia ser utilizada também para a comparação de linguagens ou ambientes de programação. Melhor ainda, compiladores de fabricantes específicos poderiam ser classificados quanto à sua aptidão para a geração de sistemas de aplicação crítica, em um sistema de *benchmarking*.

Obviamente essa eventual classificação estaria limitada às mesmas restrições quanto à unicidade da técnica de injeção de falhas adotada já discutida

anteriormente. Por decorrência direta dessa afirmação, aparece uma nova oportunidade de continuidade para o experimento proposto: repetir os mesmos conjuntos de testes aqui apresentados com o emprego de outras técnicas de injeção de falhas. Dada a natureza estocástica do processo de avaliação empregado, acredita-se que técnicas como a de bombardeamento por partículas carregadas poderiam beneficiar enormemente os resultados. O Califórnia Cf²⁵² tem sido usado com bons resultados, especialmente na Suécia, com trabalhos como o de SIVENCORONA *et al.* (2003). O emprego de um procedimento semelhante foi considerado nas fases iniciais. A formação original do autor em física subsidiava essa escolha. Essa opção foi ocasionalmente abandonada, em função das dificuldades associadas à manipulação de elementos radioativos.

De maneira análoga, outras técnicas de programação defensiva podem ser avaliadas, como por exemplo:

- Polinômio de recuperação tipo Reed-Solomon: cada variável é guardada com informação de recuperação de corrupção através de código Reed-Solomon;
- *Time stamp*: é estabelecido um período de validade para cada dado. Se ele não for utilizado na validade, é considerado inválido e recalculado;
- *Pulsed I/O*: todos os sinais de saída passam a ser dinâmicos, com uma frequência representando os zeros e outra representando os uns, de maneira a que nenhuma entrada ou saída, fixada em zero ou um por uma falha, mascare uma falha oculta e seja interpretada como um estado válido.

Os testes realizados podem ser ampliados de maneira significativa. A abordagem aqui adotada visava a comparação de metodologias de programação com caráter defensivo, de maneira a subsidiar a escolha de um projetista de sistemas de aplicação crítica. Há, entretanto, diversas outras situações em que o projetista se depara com decisões de projeto para as quais ele não encontra suporte, no que tange ao processo de decisão.

8 REFERÊNCIAS

- ALMEIDA JR., J. R. DE *et al.* *Best practices in code inspection for safety-critical software*, IEEE Software. Maio/Junho, pg. 56-63. IEEE, 2003.
- ARLAT, J. *et al.* *Comparison of Physical and Software-Implemented Fault Injection Techniques*, IEEE Transactions on computers, VOL. 52, NO. 9, 2003.
- ARLAT, J. *et al.* *From controlled fault injection experiments to dependability benchmarks*, Workshop on Dependability in Information Society: future scenarios and R&D challenges. LAAS-CNRS, 2001.
- ASSOCIAÇÃO BRASILEIRA DE NORMAS TÉCNICAS. NBR12304: Limites e métodos de medição de radioperturbação em equipamento para tecnologia da informação (ETI). <http://www.abnt.org.br>, 1992.
- AVIŽIENIS, A. *et al.* *Basic Concepts and Taxonomy of Dependable and Secure Computing*. IEEE Transactions on dependable and secure computing, Vol. 1, N°1, 2004.
- BITTINGER, W. *Properties of open strip lines for EMC measurements*. IEEE International symposium on electromagnetic compatibility. Dallas:IEEE, 1993.
- BOEHM, B. *et al.* *The TRW software productivity system*. Proceedings of the 6th international conference on Software engineering. IEEE Computer Society Press, 1982.
- BRANCO FILHO, G. Dicionário de termos de manutenção, confiabilidade e qualidade. Rio de Janeiro: Ciência Moderna, 2000.
- BREUER, M. A. e FRIEDMAN, A. D. *Diagnosis & reliable design of digital systems*, Digital System Design Series, Rockville: Computer Science Press, 1976.
- BRITISH STANDARDS. BSEN50126:Railway applications – The specification and demonstration of Reliability, Availability, Maintainability and Safety (RAMS),1999.
- BURNS, A. e WELLINGS, A. *Real-time systems and their programming languages*, International Computer Science Series, Wokingham: Addison Wesley, 1991.
- CAMARGO JR., J. B. *et al.* *Quantitative analysis methodology in safety-critical microprocessor applications*, Reliability Engineering & System Safety 74, pg. 53-62. Amsterdam: Elsevier, 2001.
- DIJKSTRA, E. W. *Programming as a discipline of mathematical nature*, EWD 361: Am. Math. 81, 6: 608–612. Mathematical Association of America, 1974.
- ESTADOS UNIDOS. AMERICAN RADIO RELAY LEAGUE. *The ARRL Antenna Book*. Newington: ARRL, 1976.
- EUROPEAN COMMITTEE FOR TECHNICAL STANDARDIZATION. EN 61000-4-3: *Electromagnetic compatibility (EMC) -- Part 4-3: Testing and measurement techniques - Radiated, radio-frequency, electromagnetic field immunity test*. <http://www.cenelec.org>, 2006.

FOLKESSON, P. *Assessment and Comparison of Physical Fault Injection Techniques*. Department of Computer Engineering. Göteborg: Chalmers University of Technology, 1999.

FRIEND, J. R. *A review of computer controlled systems safety and quality assurance concerns for acquisition managers*. *Proceedings of the Seventh Annual Conference on Computer Assurance-COMPASS '92*. Gaithersburg: IEEE, 1992.

HATTON, L. *Safer C – Developing Software for high-integrity and safety-critical systems*, The MacGraw-Hill series in software engineering. Londres: MacGraw-Hill Book Company Europe, 1995.

INTERNATIONAL ELECTROTECHNICAL COMMISSION. IEC61000-4-17: *Electromagnetic compatibility (EMC) - Part 4-17: Testing and measurement techniques - Ripple on D.C. input power port immunity test*. Ed.1.1. <http://www.iec.ch>, 2001.

INTERNATIONAL ELECTROTECHNICAL COMMISSION. IEC61000-4-29: *Electromagnetic compatibility (EMC) - Part 4-29: Testing and measurement techniques - Voltage dips, short interruptions and voltage variations on D.C. input power port immunity tests*. Ed.1.0. <http://www.iec.ch>, 2000.

INTERNATIONAL ELECTROTECHNICAL COMMISSION. IEC61508-SER: *Functional safety of electrical/electronic/ programmable electronic safety-related systems*. Ed.1.0. <http://www.iec.ch>, 2005.

INTERNATIONAL ORGANIZATION FOR STANDARDIZATION / INTERNATIONAL ELECTROTECHNICAL COMMISSION. Guide 51: *Safety aspects – Guidelines for their inclusion in standards*, 1999.

INTERNATIONAL ORGANIZATION FOR STANDARDIZATION / INTERNATIONAL ELECTROTECHNICAL COMMISSION. Guide 73: *Risk management – Vocabulary – Guidelines for use in standards*, 2002.

JACKSON, D. E DEVADAS. S. Curso 6.170: *Laboratory in Software Engineering*. MIT-Massachusetts Institute of Technology. <http://ocw.mit.edu/OcwWeb/Electrical-Engineering-and-Computer-Science/6-170Fall-2005/CourseHome/>. Boston, 2005.

KARNIK, T. et al. *Characterization of soft errors caused by single event upsets in CMOS processes*. *IEEE Transactions on Dependable and Secure Computing*, Volume 1, número 2. IEEE, 2004.

LEIGHT, A. W. *Real time software for small systems*. Cheshire: Sigma Press, 1988.

LEVESON, N. G. *A New Approach to System Safety*. MIT, 2002.

LHO, Y. H. E KIM, K. Y. *Radiation effects of proton particles in memory devices*. *ETRI Journal*, Vol. 29, nº 1, p. 124, 2007.

LOPES, D. C. *Estimação da robustez de sistemas eletrônicos via injeção de falhas por interferência eletromagnética*. Dissertação – PUCRGS. Porto Alegre, 2005.

LOPES, D. C. Verificação da confiabilidade de sistemas embarcados através de técnicas de injeção de falhas em HW/SW. Porto Alegre, 2004.

LUSSIER, B. *et al. Fault Tolerance in Autonomous Systems: How and How Much? 4th IARP/IEEE-RAS/EURON Workshop on Technical Challenges for Dependable Robots in Human Environments*. Nagoya: LAAS-CNRS Publications, 2005.

LYU, M. R. *Handbook of software reliability engineering, IEEE Computer Society Press*. Los Alamitos: MacGraw-Hill, 1995.

MADAU, D. *Rules for Defensive C Programming*, Embedded Systems Programming. 1999.

MCGETTRICK, A. D. *Software for Safety-Critical Systems: Design Safely - Lecture 11. High integrity systems course*. Glasgow: Universidade de Strathclyde, 2000.

MERTEL, H. K. *Military EMC standards and methods: MIL-STD-461 and MIL-STD-462 in PEREZ, R. (Eds.): Handbook of electromagnetic compatibility*, pp. 553-595. San Diego: Academic Press, 1995.

PAULA JR., A. R. DE *et al.* Introdução a tolerância a falhas. Minicurso Unicamp/Telebrás. Campinas:1987.

PERNER, M. *Von 10m auf 2m und zurück mit dem Transverter-Bausatz 1210*, Funk Amateur, FA 9/99, página 1057, 1999.

SAMPAIO, L. M. R. *et al.* Detectores de falhas em sistemas assíncronos. IV Workshop de Testes e Tolerância a Falhas (WTF) do SBRC2003. Natal, 2003.

SEABORG, G. T. Os elementos transurânicos sintetizados pelo homem. São Paulo: Editora Edgard Blücher Ltda./Editora da Universidade de São Paulo, 1969.

SECALL, J. M. Operacionalidade e Segurança: a dicotomia do resultado. Revista Engenharia. São Paulo: Instituto de Engenharia, 2004.

SIVENCORONA, H. *et al. Heavy-Ion Fault Injections in the Time-Triggered Communication Protocol*, in de Lemos, R. *et al.* (Eds.):LADC 2003, LNCS 2847, pp. 69-80. Berlin-Heidelberg: Springer-Verlag, 2003.

SQUAIR, M. J. *Issues in the application of software safety standards. Proceedings of the 10th Australian workshop on Safety critical systems and software*, Volume 55, p. 13-26. Sydney: Australian Computer Society, 2006.

VARGAS, F. *et al. Introducing sw-based fault handling mechanisms to cope with EMI in embedded electronics: are they a good remedy? 9th IEEE International On-Line Testing Symposium IOLTS 2003*. Página 163, <http://ieeexplore.ieee.org/servlet/opac?punumber=8615>. 2003.

VOAS, J. M. E MCGRAW, G. *Software Fault Injection*. Inc. New York: John Wiley & Sons, 1998.

WILLIAMS, T. *EMC for product designers*. Oxford: Butterworth-Heinemann, 1992.

9 APENDICE A

Figura 59 - Função Segurança $S(t)$ para o programa Exp_1.

Figura 60 - Probabilidade de falhas inseguras $F(t)$ para o programa Exp_1.

Figura 61 - Densidade de probabilidade de falhas inseguras $f(t)$ para o programa Exp_1.

Figura 62 - Função Segurança $S(t)$ para o programa Exp_Back_Recovery.

Figura 63 - Probabilidade de falhas inseguras $F(t)$ para o programa Exp_Back_Recovery.

Figura 64 - Densidade de probabilidade de falhas inseguras $f(t)$ para o programa Exp_Back_Recovery.

Figura 65 - Função Segurança $S(t)$ para o programa Exp_Div.

Figura 66 - Probabilidade de falhas inseguras $F(t)$ para o programa Exp_Div.

Figura 67 - Densidade de probabilidade de falhas inseguras $f(t)$ para o programa Exp_Div.

Figura 68 - Função Segurança $S(t)$ para o programa Exp_Div2.

Figura 69 - Probabilidade de falhas inseguras $F(t)$ para o programa Exp_Div2.

Figura 70 - Densidade de probabilidade de falhas inseguras $f(t)$ para o programa Exp_Div2.

Figura 71 - Função Segurança $S(t)$ para o programa Exp_dupla.

Figura 72 - Probabilidade de falhas inseguras $F(t)$ para o programa Exp_dupla.

Figura 73 - Densidade de probabilidade de falhas inseguras $f(t)$ para o programa Exp_dupla.

Figura 74 - Função Segurança $S(t)$ para o programa Exp_dupla6.

Figura 75 - Probabilidade de falhas inseguras $F(t)$ para o programa Exp_dupla6.

Figura 76 - Densidade de probabilidade de falhas inseguras $f(t)$ para o programa Exp_dupla6.

Figura 77 - Função Segurança $S(t)$ para o programa Exp_dupla7.

Figura 78 - Probabilidade de falhas inseguras $F(t)$ para o programa Exp_dupla7.

Figura 79 - Densidade de probabilidade de falhas inseguras $f(t)$ para o programa Exp_dupla7.

Figura 80 - Função Segurança $S(t)$ para o programa Exp_Forwad_Recovery.

Figura 81 - Probabilidade de falhas inseguras $F(t)$ para o programa Exp_Forward_Recovery.

Figura 82 - Densidade de probabilidade de falhas inseguras $f(t)$ para o programa Exp_Forward_Recovery.

Figura 83 - Função Segurança $S(t)$ para o programa ExpBCH.

Figura 84 - Probabilidade de falhas inseguras $F(t)$ para o programa ExpBCH.

Figura 85 - Densidade de probabilidade de falhas inseguras $f(t)$ para o programa ExpBCH.

Figura 86 - Função Segurança $S(t)$ para o programa ExpCheckSum.

Figura 87 - Probabilidade de falhas inseguras $F(t)$ para o programa ExpCheckSum.

Figura 88 - Densidade de probabilidade de falhas inseguras $f(t)$ para o programa ExpCheckSum.

Figura 89 - Função Segurança $S(t)$ para o programa ExpCheckSum2.

Figura 90 - Probabilidade de falhas inseguras $F(t)$ para o programa ExpCheckSum2.

Figura 91 - Densidade de probabilidade de falhas inseguras $f(t)$ para o programa ExpCheckSum2.

Figura 92 - Função Segurança $S(t)$ para o programa ExpPar.

Figura 93 - Probabilidade de falhas inseguras $F(t)$ para o programa ExpPar.

Figura 94 - Densidade de probabilidade de falhas inseguras $f(t)$ para o programa ExpPar.

Figura 95 - Função Segurança $S(t)$ para o programa ExpPar2.

Figura 96 - Probabilidade de falhas inseguras $F(t)$ para o programa ExpPar2.

Figura 97 - Densidade de probabilidade de falhas inseguras $f(t)$ para o programa ExpPar2.

10 APENDICE B

O objetivo do programa verificador de falha é rodar paralelamente com o simulador indicando como saída o tempo decorrente entre o início da simulação e a ocorrência de uma falha insegura.

O programa feito em Linguagem C executa um *loop* infinito até a ocorrência da falha. Ele lê todas as variáveis utilizadas pelo simulador e através de equações booleanas verifica a presença de falhas inseguras. Caso ocorra uma falha insegura ocorre uma quebra do *loop* infinito e o tempo decorrido da execução do programa é calculado. A leitura das variáveis é feita da seguinte forma:

```

/*Leitura dos dados*/
H1_PA = inb(0x300);
H1_PB = inb(0x301);
H1_PC = inb(0x302);
H2_PA = inb(0x304);
H2_PB = inb(0x305);
H2_PC = inb(0x306);
H3_PA = inb(0x308);
H3_PC = inb(0x30A);

/*Leitura das ocupações dos circuitos de via*/
A1 = getBit (H1_PC, 1);
B1 = getBit (H1_PC, 2);
C1 = getBit (H1_PC, 4);
D1 = getBit (H1_PC, 8);
E1 = getBit (H1_PC, 16);
F1 = getBit (H1_PC, 32);
A2 = getBit (H1_PC, 64);
B2 = getBit (H1_PC, 128);
C2 = getBit (H2_PC, 1);
D2 = getBit (H2_PC, 2);
E2 = getBit (H2_PC, 4);
F2 = getBit (H2_PC, 8);

/*Leitura dos alinhamentos de rotas*/
X01_AB = (getBit (H2_PB, 1) && !getBit (H2_PB, 2) && !getBit (H2_PB, 4)) ||
( !getBit (H2_PB, 1) && !getBit (H2_PB, 2) && getBit (H2_PB, 4));
X01_DB = getBit (H2_PB, 1) && getBit (H2_PB, 2) && !getBit (H2_PB, 4);
X01_DC = (!getBit (H2_PB, 1) && getBit (H2_PB, 2) && !getBit (H2_PB, 4)) ||
( !getBit (H2_PB, 1) && !getBit (H2_PB, 2) && getBit (H2_PB, 4));
X02_AB = (getBit (H2_PB, 8) && !getBit (H2_PB, 16) && !getBit (H2_PB, 32)) ||
(!getBit (H2_PB, 8) && !getBit (H2_PB, 16) && getBit (H2_PB, 32));
X02_DB = getBit (H2_PB, 8) && getBit (H2_PB, 16) && !getBit (H2_PB, 32);
X02_DC = (!getBit (H2_PB, 8) && getBit (H2_PB, 16) && !getBit (H2_PB, 32)) ||
( !getBit (H2_PB, 8) && !getBit (H2_PB, 16) && getBit (H2_PB, 32));

/*Leitura das posições das máquinas de chave*/
X01_NORMAL = !getBit (H3_PC, 1) && !getBit(H3_PC, 2);
X01_REVERSO = getBit (H3_PC, 1) && !getBit(H3_PC, 2);
X02_NORMAL = !getBit (H3_PC, 4) && !getBit(H3_PC, 8);
X02_REVERSO = getBit (H3_PC, 4) && !getBit(H3_PC, 8);

```

```
/*Leitura dos códigos de velocidade dos circuitos de via*/
```

```
V0_A1 = !getBit (H1_PA, 1) && !getBit (H1_PA, 2);
V50_A1 = getBit (H1_PA, 1) && !getBit (H1_PA, 2);
V75_A1 = !getBit (H1_PA, 1) && getBit (H1_PA, 2);
V100_A1 = getBit (H1_PA, 1) && getBit (H1_PA, 2);
```

```
V0_B1 = !getBit (H1_PA, 4) && !getBit (H1_PA, 8);
V50_B1 = getBit (H1_PA, 4) && !getBit (H1_PA, 8);
V75_B1 = !getBit (H1_PA, 4) && getBit (H1_PA, 8);
V100_B1 = getBit (H1_PA, 4) && getBit (H1_PA, 8);
```

```
V0_C1 = !getBit (H1_PA, 16) && !getBit (H1_PA, 32);
V50_C1 = getBit (H1_PA, 16) && !getBit (H1_PA, 32);
V75_C1 = !getBit (H1_PA, 16) && getBit (H1_PA, 32);
V100_C1 = getBit (H1_PA, 16) && getBit (H1_PA, 32);
```

```
V0_D1 = !getBit (H1_PA, 64) && !getBit (H1_PA, 128);
V50_D1 = getBit (H1_PA, 64) && !getBit (H1_PA, 128);
V75_D1 = !getBit (H1_PA, 64) && getBit (H1_PA, 128);
V100_D1 = getBit (H1_PA, 64) && getBit (H1_PA, 128);
```

```
V0_E1 = !getBit (H1_PB, 1) && !getBit (H1_PB, 2);
V50_E1 = getBit (H1_PB, 1) && !getBit (H1_PB, 2);
V75_E1 = !getBit (H1_PB, 1) && getBit (H1_PB, 2);
V100_E1 = getBit (H1_PB, 1) && getBit (H1_PB, 2);
```

```
V0_F1 = !getBit (H1_PB, 4) && !getBit (H1_PB, 8);
V50_F1 = getBit (H1_PB, 4) && !getBit (H1_PB, 8);
V75_F1 = !getBit (H1_PB, 4) && getBit (H1_PB, 8);
V100_F1 = getBit (H1_PB, 4) && getBit (H1_PB, 8);
```

```
V0_A2 = !getBit (H1_PB, 16) && !getBit (H1_PB, 32);
V50_A2 = getBit (H1_PB, 16) && !getBit (H1_PB, 32);
V75_A2 = !getBit (H1_PB, 16) && getBit (H1_PB, 32);
V100_A2 = getBit (H1_PB, 16) && getBit (H1_PB, 32);
```

```
V0_B2 = !getBit (H1_PB, 64) && !getBit (H1_PB, 128);
V50_B2 = getBit (H1_PB, 64) && !getBit (H1_PB, 128);
V75_B2 = !getBit (H1_PB, 64) && getBit (H1_PB, 128);
V100_B2 = getBit (H1_PB, 64) && getBit (H1_PB, 128);
```

```
V0_C2 = !getBit (H2_PA, 1) && !getBit (H2_PA, 2);
V50_C2 = getBit (H2_PA, 1) && !getBit (H2_PA, 2);
V75_C2 = !getBit (H2_PA, 1) && getBit (H2_PA, 2);
V100_C2 = getBit (H2_PA, 1) && getBit (H2_PA, 2);
```

```
V0_D2 = !getBit (H2_PA, 4) && !getBit (H2_PA, 8);
V50_D2 = getBit (H2_PA, 4) && !getBit (H2_PA, 8);
V75_D2 = !getBit (H2_PA, 4) && getBit (H2_PA, 8);
V100_D2 = getBit (H2_PA, 4) && getBit (H2_PA, 8);
```

```
V0_E2 = !getBit (H2_PA, 16) && !getBit (H2_PA, 32);
V50_E2 = getBit (H2_PA, 16) && !getBit (H2_PA, 32);
V75_E2 = !getBit (H2_PA, 16) && getBit (H2_PA, 32);
V100_E2 = getBit (H2_PA, 16) && getBit (H2_PA, 32);
```

```

V0_F2 = !getBit (H2_PA, 64) && !getBit (H2_PA, 128);
V50_F2 = getBit (H2_PA, 64) && !getBit (H2_PA, 128);
V75_F2 = !getBit (H2_PA, 64) && getBit (H2_PA, 128);
V100_F2 = getBit (H2_PA, 64) && getBit (H2_PA, 128);

```

Após a leitura das variáveis, elas são utilizadas pelas booleanas na verificação de falhas. Conforme descrito anteriormente, as falhas inseguras do sistema foram divididas em quatro tipos: velocidade diferente de zero no intertravamento quando não há rota alinhada, erro no perfil de velocidade, falsa desocupação e movimentação insegura da máquina de chave. As booleanas correspondentes à verificação destas falhas são mostradas abaixo:

```

/*verifica código de velocidade igual a zero quando não há rota alinhada.*/
if ( !V0_A1 && (!X01_AB || X01_DB))
{
    falha[0] = 1;
    break;
}
if (!V0_B1 && !X01_AB && !X01_DB)
{
    falha[0] = 1;
    break;
}
if (!V0_A2 && !X01_DC && !X01_DB)
{
    falha[0] = 1;
    break;
}
if (!V0_B2 && !X01_DB && !X01_DC)
{
    falha[0] = 1;
    break;
}
if (!V0_E2 && (!X02_AB || X02_DB))
{
    falha[0] = 1;
    break;
}
if (!V0_F2 && !X02_AB && !X02_DB)
{
    falha[0] = 1;
    break;
}
if (!V0_D1 && !X02_DC && !X02_DB)
{
    falha[0] = 1;
    break;
}
if (!V0_E1 && !X02_DB && !X02_DC)
{
    falha[0] = 1;
    break;
}
}
/*Verifica perfil de velocidade*/
if (A1 && (!V0_F1 || (X02_NORMAL && (V75_E1 || V100_E1 || V100_D1))))

```

```

{
    falha[1] = 1;
    break;
}
if (B1 && (!V0_A1 || V75_F1 || V100_F1 || (X02_NORMAL && V100_E1)) &&
X01_NORMAL)
{
    falha[1] = 1;
    break;
}
if (B1 && (!V0_B2 || V75_A2 || V100_A2 || V100_F2) && X01_REVERSO)
{
    falha[1] = 1;
    break;
}
if ((C1 && (!V0_B1 || V75_A1 || V100_A1 || V100_F1)) && X01_NORMAL)
{
    falha[1] = 1;
    break;
}
if ((C1 && (!V0_B1 || V75_B2 || V100_B2 || V100_A2)) && X01_REVERSO)
{
    falha[1] = 1;
    break;
}
if (D1 && (!V0_C1 || V75_B1 || V100_B1 || V100_A1) && X01_NORMAL)
{
    falha[1] = 1;
    break;
}
if (D1 && (!V0_C1 || V75_B1 || V100_B1 || V100_B2) && X01_REVERSO)
{
    falha[1] = 1;
    break;
}
if (E1 && (!V0_D1 || V75_C1 || V100_C1 || V100_B1))
{
    falha[1] = 1;
    break;
}
if (F1 && (X02_NORMAL && (!V0_E1 || V75_D1 || V100_D1 || V100_C1)))
{
    falha[1] = 1;
    break;
}
if (A2 && (!V0_F2 || V75_E2 || V100_E2 || V100_D2) && X02_NORMAL)
{
    falha[1] = 1;
    break;
}
if (A2 && (!V0_F2 || V75_E1 || V100_E1 || V100_D1) && X02_REVERSO)
{
    falha[1] = 1;
    break;
}
if (B2 && (!V0_A2 || V75_F2 || V100_F2 || V100_E2) && X02_NORMAL)
{

```



```

        falha[1] = 1;
        break;
    }
    if (B2 && (!V0_A2 || V75_F2 || V100_F2 || V100_E1) && X02_REVERSO)
    {
        falha[1] = 1;
        break;
    }
    if (C2 && (X01_NORMAL && (!V0_B2 || V75_A2 || V100_A2 || V100_F2)))
    {
        falha[1] = 1;
        break;
    }
    if (D2 && (!V0_C2 || (X01_NORMAL && (V75_B2 || V100_B2 || V100_A2))))
    {
        falha[1] = 1;
        break;
    }
    if (E2 && (!V0_D2 || V75_C2 || V100_C2 || (V100_B2 && X01_NORMAL)))
    {
        falha[1] = 1;
        break;
    }
    if ((F2 && (!V0_E2 || V75_D2 || V100_D2 || V100_C2)) && X02_NORMAL)
    {
        falha[1] = 1;
        break;
    }
    if ((F2 && (!V0_E1 || V75_D1 || V100_D1 || V100_C1)) && X02_REVERSO)
    {
        falha[1] = 1;
        break;
    }
    /*Verifica falsa desocupação*/
    if (!A1 && antA1 && !B1)
    {
        falha[2] = 1;
        break;
    }
    if (!B1 && antB1 && !C1)
    {
        falha[2] = 1;
        break;
    }
    if (!C1 && antC1 && !D1)
    {
        falha[2] = 1;
        break;
    }
    if (!D1 && antD1 && !E1)
    {
        falha[2] = 1;
        break;
    }
    if (!E1 && antE1 && !F1 && !F2)
    {
        falha[2] = 1;
    }

```

```

        break;
    }
    if (!F1 && antF1 && !A1)
    {
        falha[2] = 1;
        break;
    }
    if (!A2 && antA2 && !B2)
    {
        falha[2] = 1;
        break;
    }
    if (!B2 && antB2 && !B1 && !C2)
    {
        falha[2] = 1;
        break;
    }
    if (!C2 && antC2 && !D2)
    {
        falha[2] = 1;
        break;
    }
    if (!D2 && antD2 && !E2)
    {
        falha[2] = 1;
        break;
    }
    if (!E2 && antE2 && !F2)
    {
        falha[2] = 1;
        break;
    }
    if (!F2 && antF2 && !A2)
    {
        falha[2] = 1;
        break;
    }
    /*Movimentação insegura da máquina de chave*/
    if ( !X01_NORMAL && !X01_REVERSO && (B1 || B2 || X01_AB || X01_DC || X01_DB))
    {
        falha[3] = 1;
        break;
    }
    if (!X02_NORMAL && !X02_REVERSO && (F2 || E1 || X02_AB || X02_DC || X02_DB))
    {
        falha[3] = 1;
        break;
    }
}

```

Caso venha a ocorrer alguma falha insegura, o *loop* infinito é desfeito e através do vetor falha é possível identificar a falha insegura ocorrida:

```

if( falha[0] )
printf("Rota não alinhada e velocidade diferente de zero no intertravamento\n\n");
if( falha[1] )
    printf("Falha insegura no perfil de velocidade\n\n");
if( falha[2] )

```

```

        printf("Falsa desocupação\n\n");
    if( falha[3] )
        printf("Movimentação insegura da máquina de chave\n\n");

```

O tempo de execução do programa é calculado através da função *clock()* da biblioteca *time.h*. A função *clock()* retorna o número de pulsos de *clock* que foi dado ao processador. Logo, conhecida a velocidade de pulsos por segundo é possível saber quanto tempo foi gasto desde o início da execução do loop até a ocorrência de uma falha insegura.

Assim o tempo de execução do programa, até encontrar uma falha insegura, é calculado da seguinte forma:

```

t0 = clock(); /*guarda o número de pulsos de clock inicial.
while (1)
{
    Leitura das variáveis;
    Verificação de falhas inseguras;
}
tf = clock();
tempo_gasto = ( (double) (tf - t0) ) / CLOCKS_PER_SEC;

```

GLOSSÁRIO

Aceitação dos riscos (*risk acceptance*): decisão de aceitar os riscos.

Acidente: um evento não intencional que causa morte, ferimento e danos materiais ou ambientais.

Análise de riscos (*risk analysis*): processo sistemático para identificar os perigos e estimar os riscos.

Falha Segura (*Fail Safe*): A incorporação de características de forma a assegurar que mal-funcionamentos afetando a segurança levem o sistema a um estado seguro. É uma característica de um sistema e de seus elementos cujo objetivo é assegurar que qualquer falha ou mal-funcionamento não resultará em uma condição insegura.

Falha: uma falha no sistema ocorre quando o serviço entregue não é exatamente aquele pretendido. Uma falha é o efeito de um erro no serviço pretendido.

Gestão de riscos (*risk management*): atividades coordenadas para dirigir e controlar uma organização em relação aos riscos (geralmente abrange o processo de análise e de avaliação de riscos, tratamento, aceitação e a comunicação dos mesmos).

Identificação de perigos (*hazard identification*): processo para localizar, relacionar e caracterizar os perigos.

Incidente: um evento não planejado, incontrolável, que sob diferentes circunstâncias poderia resultar em um acidente.

Mitigação (*mitigation*): limitação de qualquer consequência negativa de um evento particular.

Ocorrência: desvio operacional percebido pela manifestação de um sintoma (em geral representa uma degradação de uma função operacional).

Perigo (*hazard*): fonte potencial de dano (choque elétrico, corte, incêndio, etc.). Uma situação física ou um estado do sistema, frequentemente estabelecido por um evento inicial, que pode levar a um acidente.

Probabilidade (*probability*): extensão na qual é esperada a ocorrência de um evento.

Processo de análise e de avaliação de riscos (*risk assessment*): Área da Engenharia que se ocupa com os riscos, cujo conceito é liminarmente diferenciável do termo *risk analysis*. Visa analisar o impacto da introdução de modernas tecnologias na sociedade, seja através de um método quantitativo (medições ambientais, relação custo-benefício), seja através da discussão do gerenciamento do risco (*risk management*).

Projeto de Falha Segura (*Fail Safe Design*): princípio de projeto segundo o qual todas as falhas devem levar a condições seguras, isso é, todos os defeitos simples e combinações previsíveis de defeitos posteriores correlacionados devem ser de efeito restritivo.

Risco: complemento da probabilidade de funcionamento seguro do sistema.

Risco residual (*residual risk*): risco remanescente após o seu tratamento.

Risco tolerável (*tolerable risk*): risco aceito em um dado contexto e baseado nos valores correntes da sociedade.

Segurança (*safety*): livre de riscos intoleráveis.

Severidade: grau com que um determinado perigo afeta as condições de risco.

Situação perigosa (*hazardous situation*): circunstância na qual pessoas, ativos ou o meio ambiente estão expostos a um ou mais perigos.