



Universidade Federal de Mato Grosso do Sul
Câmpus de Coxim
Bacharelado em Sistemas de Informação

Biblioteca Digital: Uma Estratégia de Gestão e Preservação de Periódicos do Século XX com DSpace e Metadados Dublin Core

Vagner da Silva Bezerra

Coxim - MS
Setembro/2016



Universidade Federal de Mato Grosso do Sul
Câmpus de Coxim
Bacharelado em Sistemas de Informação

Biblioteca Digital: Uma Estratégia de Gestão e Preservação de Periódicos do Século XX com DSpace e Metadados Dublin Core

Vagner da Silva Bezerra

Trabalho de Conclusão de Curso apresentado ao Câmpus de Coxim - CPCX da Universidade Federal de Mato Grosso do Sul - UFMS, como requisito parcial para a obtenção do título de Bacharel em Sistemas de Informação sob a orientação da Profa. Ma. Juliana Wolf Pereira.

Coxim - MS
Setembro/2016



TERMO DE APROVAÇÃO

A presente monografia, intitulada **Biblioteca Digital: Uma Estratégia de Gestão e Preservação de Periódicos do Século XX com DSpace e Metadados Dublin Core**, elaborada pelo acadêmico **Vagner da Silva Bezerra** e aprovada pela Banca Examinadora composta pelos membros abaixo assinados, sendo julgada adequada para o cumprimento do requisito legal para obtenção do título de Bacharel em Sistemas de Informação.

_____, ____ de _____ de 20____.

Profa. Ma. Juliana Wolf Pereira - Orientadora

Prof. Me. Angelo Darcy Molin Brun

Prof. Dr. Gedson Faria

Câmpus de Coxim – CPCX/UFMS

Av. Márcio Lima Nantes, s/n – Vila da Barra – Estrada do Pantanal
Fone/Fax: +55 (67) 3291-0200/0202/0203 – CEP 79400-000 – Coxim/MS - Brasil
<http://www.cpcx.ufms.br> - e-mail: cpcx@ufms.br

Dedicatória

Texto de Dedicatória.

Agradecimentos

Texto de Agradecimentos.

“Se enxerguei mais longe, foi porque me apoiei sobre os ombros de gigantes.”

Isaac Newton

Resumo

Atualmente, o ser humano utiliza diversos aparelhos eletrônicos, tais como celulares, tocadores de MP3, televisores, *tablets*, e outros dispositivos usados no auxílio das atividades diárias e na melhoria da qualidade de vida. Graças a expansão da computação ubíqua os sistemas embarcados estão cada vez mais presentes no cotidiano das pessoas. No entanto, esses sistemas podem apresentar defeitos, que correspondem a uma incapacidade do sistema executar uma determinada tarefa devido a falhas em algum componente. As causas estão associadas a danos causados em algum componente, ferrugem, ou outros tipos de deteriorações; *bugs* de software; e perturbações externas, como duras condições ambientais, interferência eletromagnética, radiação ionizante, ou má utilização do sistema [1].

Os objetivos deste trabalho foram estudar possíveis causas de falhas em sistemas embarcados e ampliar as bibliotecas *FaultInjector* e *FaultRecovery*. Uma das modificações visa possibilitar ao usuário desenvolver uma máquina de estados, na qual cada estado pode ser implementado independentemente dos outros. Antes a *FaultRecovery* não entregava ao usuário uma estrutura de desenvolvimento pronto, agora ela foi modificada para atender a um padrão de projeto chamado *State*. Além dessa melhoria, foi possível criar uma classe que simplifica o uso da redundância de dados, aumentando a integridade de um sistema embarcado.

Ao final, são apresentados os resultados mostrando o tempo de execução após as modificações realizadas na biblioteca *FaultRecovery* a fim de verificar se essas alterações impactaram no desempenho do código testado. O teste realizado com a *FaultRecovery* foi executado em 5,2107 segundos, enquanto que em média o teste sem a biblioteca foi executado em 4,6854 segundos, ou seja, a biblioteca elevou o tempo de execução do teste em 0,5253 segundos. Entretanto, a biblioteca foi exposta a testes de recuperação de falhas, mostrando-se eficaz em todos eles. A classe também foi exposta a testes de integridade dos dados. Nos resultados que não utilizaram a redundância de dados, o tempo de execução médio foi de 0,0614 segundos, enquanto que nos testes com redundância o tempo médio foi de 0,3272 segundos. Neste, a classe *TData* elevou o tempo

de execução do teste em 0,2658 segundos. Entretanto, no primeiro resultado a média de falhas encontradas foi de 44% enquanto que no segundo foi de 0%.

Como resultado deste trabalho, a ideia de modificação da biblioteca *FaultRecovery* foi utilizada pelo projeto de extensão Coxim Robótica sediado na UFMS - Campus Coxim, no desenvolvimento de um programa para um robô seguidor de linha e continuará sendo utilizada em programas futuros. A classe que facilita a implementação de redundância de dados foi incluída na biblioteca *FaultRecovery*, que possibilita ao usuário definir se o seu sistema embarcado se recuperará de falhas e também poderá proteger seus dados mais importantes.

Abstract

Currently, humans use various electronic devices such as mobile phones, MP3 players, televisions, tablets, and other devices used in aid of daily activities and improving the quality of life. Due to expansion of ubiquitous computing, embedded systems are increasingly present in daily life. However these systems can malfunction, indicating a system's inability to perform a certain task because of faults in a device component or the environment, which in turn, failures are caused by [1]. According to Nelson [1] a fault is an abnormal condition. The causes are associated with damage to any component, rust or other deterioration; and external disturbances, such as harsh environmental conditions, electromagnetic interference, ionizing radiation, or misuse of the system.

The objectives of this work were to study possible causes of failures in embedded systems, modify the libraries FaultInjector and FaultRecovery. Even create a data redundancy class in which its function is to ensure the data integrity of an embedded system. One of the modifications is designed to allow the user to develop a state machine, in which each state may be implemented independently of the others. Before the FaultRecovery not delivered to the user a ready development framework, it has now been modified to meet a standard project called State.

At the end, the results are displayed showing the execution time after the changes made in the library FaultRecovery to see if these changes impacted the performance of the tested code. The test with microarray FaultRecovery was run in 5,2107 seconds, while on average the test without the library was running at 4,6854 seconds or, the library increased the test runtime held at 0,5253. However the library was exposed to disaster recovery testing, proving to be effective in all of them. The class was also exposed to the performance tests and data redundancy, the results using no data redundancy, the average run time was 0,0614 seconds, while the testing redundancy with the average time was 0,3272 seconds, the TData class raised the test runtime 0,2658 seconds. However the first result average flaws found was 44 % while the second was 0 %.

As a result of this work, the idea of modifying the library FaultRecovery was used by the extension project Cushion Robotics based in UFMS - Campus cushion in the development

of a program for a line follower cart and will be used in future programs. a class that allows the use of data redundancy in an embedded system was also created, which was included in the library FaultRecovery, which enables the user to set up your embedded system to recover from faults and can also protect your most data important.

Lista de Figuras

2.1	Modelo de três universos	24
2.2	Cinturão de Van Allen	25
2.3	Características das Falhas	29
2.4	Programação N-Versões	31
2.5	Blocos de Recuperação	32
3.1	Mapas das Regiões de Memória dos Modelos LPC1768/66/65/64	39
3.2	Diagrama de classes da biblioteca <i>FaultRecovery</i>	42
3.3	Fluxograma da biblioteca <i>FaultRecovery</i>	43
3.4	Figura que apresenta a saída com os valores das cópias consistentes para tipos primitivos após a injeção de falhas.	48
3.5	Valores das cópias consistentes após a injeção de falhas com objeto de pilha.	50
3.6	Valores das cópias consistentes após a injeção de falhas com ponteiro.	51
3.7	Figura que apresenta a saída com os valores das cópias consistentes após a atualização do objeto <i>TData</i> do tipo carro e da injeção de falhas.	51
4.1	Tempo de execução da biblioteca <i>FaultRecovery</i>	53
4.2	Tempo de execução da biblioteca <i>FaultRecovery</i> com três tamanhos de vetores diferentes.	54
4.3	Tempo de execução da Classe <i>TData</i>	56
4.4	Teste de redundância de dados da classe <i>TData</i>	57
4.5	Resultados Obtidos da Biblioteca <i>FaultRecovery</i>	58

4.6	Injeção de Falhas na Memória <i>Flash</i>	60
-----	---	----

Lista de Tabelas

2.1	Resumo dos Atributos de Dependabilidade	28
-----	---	----

Lista de Quadros

1	Uso de macro como função	41
2	Macro sendo chamada no código	41
3	Exemplo de criação de um estado	44
4	Métodos <i>createRecoveryPoints</i> e <i>run</i>	47

Lista de Siglas de Abreviações

Sumário

1	Introdução	18
1.1	Justificativa	19
1.2	Objetivos	20
1.2.1	Objetivo Geral	20
1.2.2	Objetivos Específicos	20
1.3	Organização da Proposta	22
2	Fundamentação Teórica	23
2.1	Falha, Erro e Defeito	23
2.2	Principais Fontes de Radiação e seus Efeitos nos Circuitos Eletrônicos . .	24
2.2.1	Cinturão de Van Allen	25
2.2.2	Atividade Solar	25
2.2.3	Raios Cósmicos	26
2.2.4	Partículas alpha	27
2.2.5	Efeitos Singulares ou <i>Single Event Effects</i> (SEE)	27
2.3	Dependabilidade	28
2.4	Tolerância a Falhas	28
2.5	Técnicas de Tolerância a Falhas	30
2.5.1	Técnicas de redundância baseadas em software	30
2.5.2	Diversidade ou Programação N-Versões	31
2.5.3	Blocos de Recuperação	32

2.5.4	Verificação de Consistência	32
2.6	Injeção de Falhas	32
2.6.1	Injeção de falhas por <i>Hardware</i>	33
2.6.2	Injeção de falhas por <i>Software</i>	34
2.7	<i>Design Patterns</i>	34
2.7.1	Padrão GoF (Padrões Fundamentais Originais)	35
3	Metodologia	36
3.1	Injetor de Falhas	37
3.1.1	Mapeamento de Memória	37
3.1.2	Injeção de Falhas na Memória Flash	39
3.2	FaultRecovery: Extensão da biblioteca	41
3.2.1	Refatoração e Aperfeiçoamento: Versão 1.0	41
3.2.2	Refatoração e Aperfeiçoamento: Versão 2.0	45
3.3	<i>Classe de Redundância de Dados: TData</i>	48
3.3.1	Classe TData com Tipos Primitivos	48
3.3.2	Um Exemplo Ilusório Para Utilização da Classe TData com Objetos	49
3.3.3	Classe Carro com Objeto de Pilha	49
3.3.4	Classe Carro com Ponteiro	50
4	Resultados	52
4.1	Desempenho da Biblioteca <i>FaultRecovery</i>	52
4.2	Desempenho e Eficiência da Classe TData	55
4.3	Recuperação de falhas da biblioteca <i>FaultRecovery</i>	57
4.4	Injeção de Falhas com a Biblioteca <i>FaultInjector</i>	59
5	Conclusão	61
5.1	Contribuições deste Trabalho	62

5.2	Dificuldades Encontradas	63
5.3	Trabalhos Futuros	63
	Referências	64
	Apêndices	69
	A Anexos	69

Capítulo 1

Introdução

Atualmente, o ser humano utiliza diversos aparelhos eletrônicos, tais como celulares, tocadores de MP3, televisores, *tablets*, e outros dispositivos usados no auxílio das atividades diárias e na melhoria da qualidade de vida. Em comum, esses aparelhos possuem equipamentos computacionais acoplados e por isso são denominados sistemas embarcados (ou, computadores embutidos). Esses, desempenham tarefas específicas, sendo compostos por um circuito totalmente integrado que trabalha independente de outras operações, porém seu funcionamento depende das ações de um usuário ou de eventos no meio externo [2]. Um exemplo é o sistema embarcado de um microondas, no qual o programa interno é responsável por ajustar a potência correta, selecionar e medir o tempo de acionamento do forno, e emitir um sinal quando a tarefa é concluída.

Por ser uma classe de computadores que executa tarefas de propósito específico, os sistemas embarcados geralmente possuem requisitos que combinam bom desempenho com rigorosas limitações de custo e consumo de energia [3]. Para atender aos requisitos citados, os sistemas embarcados utilizam microcontroladores ao invés de microprocessadores, uma vez que, com exceção a alguns sistemas de tempo real, dificilmente necessitam de grande poder de processamento. Conforme a definição de Malvino [4], um microcontrolador é um computador completo construído num único circuito integrado, contendo, dentre outras unidades, portas de entrada e saída seriais e paralelas, temporizadores, controles de interrupção, memórias *RAM* e *ROM*.

Segundo Patterson e Hennessy [5], os sistemas embarcados correspondem a maior classe de computadores, pois devido a sua natureza especialista, podem ser encontrados em inúmeras aplicações. Chetan [6] afirma que a quantidade de sistemas embarcados tem crescido nos últimos anos, pois com a expansão da computação ubíqua alguns equipamentos antes construídos com pouco ou nenhum recurso computacional tornaram-se mais sofisticados, incorporando algum tipo de sistema embarcado [3].

Entretanto, falhas nesses sistemas podem causar transtornos e prejuízos. Por esse motivo, equipamentos utilizados na indústria aeroespacial e militar [1], dentre outros, são desenvolvidos com estratégias de tolerância a falhas para torná-los sistemas confiáveis. Conforme Johnson [7], tolerância a falhas é a propriedade que permite a um sistema continuar funcionando adequadamente, mesmo que num nível reduzido, após a manifestação de falhas em alguns de seus componentes.

1.1 Justificativa

Embora a maioria dos sistemas embarcados tenha como requisito o baixo custo, a adoção de estratégias para a tolerância a falhas é necessária [8], pois consequências de falhas podem causar danos que variam de transtornos ao usuário a prejuízos financeiros [5]. Um exemplo são os sistemas embarcados para estações meteorológicas, utilizadas na previsão de doenças em lavouras. Estes sistemas coletam os parâmetros climáticos por meio de sensores e enviam os dados a um computador central, que disponibiliza os índices de doença para o agricultor em uma aplicação *web*. Os índices são utilizados pelos agricultores para definir o momento certo para a aplicação dos defensivos [9, 10]. Se o sistema de coleta de dados climáticos falha, o agricultor não fica ciente do momento correto para aplicação dos defensivos, podendo a plantação ser infectada e destruída pela doença.

Fenômenos da natureza, interferência eletromagnética, desgaste dos componentes de hardware [11] são alguns dos principais motivos para a causa de falhas em semicondutores, principalmente quando os dispositivos não possuem blindagem contra ruídos ou pulsos transitórios causados por prótons, íons pesados e elétrons. Dependendo da amplitude em corrente, tensão e duração, podem ser interpretados como um sinal interno do circuito, gerando erros. Quando o pulso transitório ocorre no espaço de memória, esse efeito é visto como uma inversão do valor de armazenamento no flip-flop, ou seja, um bit-flip [12].

Há também as falhas causadas por *bugs* de software, pois na indústria de *software* como um todo, existem em média de cinco a vinte falhas para cada mil linhas de código, e já existem no mercado sistemas embarcados com *softwares* contendo mais de um milhão de linhas de código. Mesmo com um padrão CMM (*Capability Maturity Model*) nível três (padrão intermediário, em que os processos são definidos e gerenciados) [13] pode-se esperar milhares de problemas e *recalls*, que em termos financeiros resultam em centenas de milhões de dólares de prejuízo [14].

Com a crescente expansão da computação ubíqua, a utilização de sistemas em-

barcados tem se tornado cada vez mais presente no cotidiano das pessoas. Logo, falhas nesses dispositivos ficarão ainda mais propícias de ocorrerem, uma vez que os sistemas embarcados estão sujeitos a diversos fatores causadores de falhas, seja em razão de *bugs* de *software*, fenômenos físicos do meio ambiente que afetam o *hardware*, interferência eletromagnética e desgaste dos componentes de *hardware* [3]. Dado o exposto é importante salientar que tolerar falhas no desenvolvimento de *software* e *hardware* é um dos grandes desafios da computação no Brasil [15].

1.2 Objetivos

1.2.1 Objetivo Geral

Este trabalho teve como objetivo ampliar o injetor de falhas (*FaultInjector*) e a biblioteca de recuperação de falhas (*FaultRecovery*) desenvolvidos por Kruger [3] em sua dissertação de mestrado. O injetor de falhas não injetava falhas na memória *flash* e seu mapeamento de memória era específico para o microcontrolador *mbed* modelo NXP 1768. Nesta nova versão, a proposta foi ampliar o injetor para funcionar em qualquer modelo da família *mbed* e remover a limitação que existia de inserir falhas apenas na memória *SRAM*.

A biblioteca *FaultRecovery* permitia a recuperação de falhas e a implementação de uma máquina de estados. No entanto, o usuário ficava responsável por criar sua própria estrutura de manipulação da máquina de estados, e isso a tornava confusa e factível a erros. Nesta ampliação, objetivou-se melhorar a arquitetura da biblioteca para facilitar seu uso, criando uma nova estrutura capaz de gerenciar automaticamente as mudanças de estados, reduzindo o número de alocações de memória e evitando cópias desnecessárias de objetos.

Por fim, outro objetivo deste trabalho foi criar uma classe na biblioteca *FaultRecovery* capaz de automatizar a redundância dos dados nos sistemas computacionais embarcados.

1.2.2 Objetivos Específicos

- Desenvolver um mapeamento das regiões de memória, para estender a utilização da biblioteca *FaultInjector* aos demais modelos da família de microcontroladores *mbed* LPC176X;

- Pesquisar as técnicas de tolerância a falhas presentes na literatura, identificar quais podem ser adicionadas a biblioteca de tolerância a falhas atual.
- Melhorar o sistema injetor de falhas para que ele possa injetar falhas em outras regiões de memória ainda não exploradas (*flash*).
- Ampliar a biblioteca de recuperação de falhas *FaultRecovery*, para que seja possível desenvolver uma máquina de estados, na qual cada estado seja implementado independentemente para facilitar a modularização.
- Implementar uma classe (TData) capaz de fazer redundância automática dos dados;
- Realizar testes de desempenho e eficiência após as modificações da biblioteca *FaultRecovery*.
- Realizar testes de desempenho e eficiência após a criação da classe *TData*.
- Após a modificação do injetor de falhas, verificar se falhas estão sendo injetadas na memória *flash*.

1.3 Organização da Proposta

No Capítulo 2 são apresentados os conceitos utilizados neste trabalho de acordo com a literatura estudada. Na Seção 2.1 explica-se os conceitos de falha, erro e defeito ou modelo de três universos. Na Seção 2.2 são descritas as principais fontes de radiação e seus efeitos nos circuitos eletrônicos. O conceito de “dependabilidade” é explicado na Seção 2.3 e na Seção 2.4 explica-se o conceito geral de tolerância a falhas e os atributos necessários para que uma falha seja definida. Na seção 2.5 são apresentadas as principais técnicas de tolerância a falhas e na seção 2.6 as principais técnicas de injeção de falhas.

As modificações realizadas nas bibliotecas e a criação da classe *TData* são exibidas no Capítulo 3, dividido em três seções. Na Seção 3.1 são apresentadas as implementações e as modificações realizadas na biblioteca *FaultInjector*. Na Seção 3.2 é exibida a extensão da biblioteca *FaultRecovery*. Na Seção 3.3 são exibidas as implementações realizadas para criação da classe *TData*, sua utilização é explicada mediante exemplos.

No Capítulo 4 são exibidos os resultados encontrados após os testes de tempo de execução e tolerância a falhas em que foram expostas as bibliotecas *FaultInjector*, *FaultRecovery* e a classe *TData*. No Capítulo 5 são exibidas as considerações finais deste trabalho.

Capítulo 2

Fundamentação Teórica

Neste Capítulo são apresentados os conceitos utilizados neste trabalho de acordo com a literatura estudada. Na seção 2.1 explica-se os conceitos de falha, erro e defeito ou modelo de três universos. Na Seção 2.2 são descritas as principais fontes de radiação e seus efeitos nos circuitos eletrônicos. Na Seção 2.3 explica-se o conceito de "dependabilidade". Na Seção 2.4 explica-se o conceito de tolerância a falhas e os atributos necessários para que uma falha seja considerada como tal. Na Seção 2.5 são apresentadas as principais técnicas de tolerância a falhas e na Seção 2.6 as principais técnicas de injeção de falhas.

2.1 Falha, Erro e Defeito

Quando aplicado a sistemas digitais os termos falha, erro e defeito possuem significados diferentes. Defeito indica uma incapacidade do sistema executar uma determinada tarefa devido a erros em algum componente do dispositivo ou no ambiente, que por sua vez, são causados por falhas [1].

Segundo Nelson [1] uma falha é uma condição física anômala. As causas estão associadas a erros de projeto, tais como erros de especificação do sistema: problemas de fabricação; danos causados em algum componente, ferrugem, ou outros tipos de deteriorações; e perturbações externas, como duras condições ambientais, interferência eletromagnética, radiação ionizante, ou má utilização do sistema. Falhas resultantes de erros de projetos e fatores externos são especialmente difíceis de serem protegidos com uma modelagem prévia, porque suas ocorrências e efeitos são difíceis de serem previstos, por exemplo, que o *hardware* utilizado seja exposto a um alto nível de radiação. Já Johnson [7] explica os conceitos de falha, erro e defeito utilizando um modelo

de universo, nos quais falhas são associadas ao universo físico, erros ao universo da informação e defeitos ao universo do usuário.

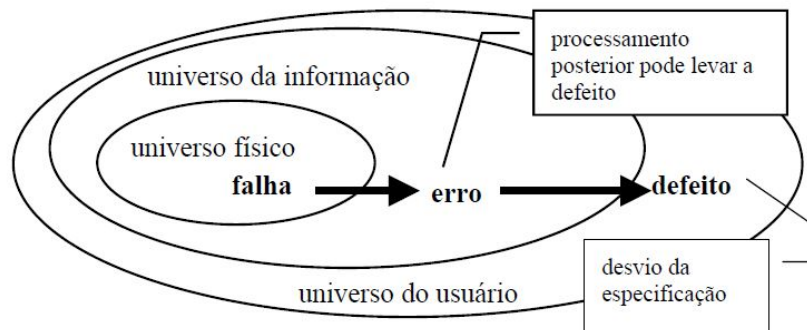


Figura 2.1: Modelo de 3 universos: falha, erro e defeito. Retirado de Weber [16].

Um exemplo de uma falha no universo físico é um chip de memória com uma falha do tipo grudado-em-zero (*stuck-at-zero*). Esta falha pode gerar um erro no universo da informação, uma vez que pode-se influenciar uma interpretação equivocada da informação armazenada em um dispositivo eletrônico, alterando o seu valor e como resultado esta alteração se torna um defeito, perceptível ao universo do usuário, no qual o sistema pode negar autorização de embarque para todos os passageiros de um voo [1]. Na Figura 2.1 é mostrado a simplificação do exemplo anterior.

2.2 Principais Fontes de Radiação e seus Efeitos nos Circuitos Eletrônicos

Em 1962 ocorreu uma falha no Satélite de Telecomunicações *Telstar* após um teste nuclear realizado em alta altitude pelos Estados Unidos, surge então a primeira evidência de que a radiação pode perturbar a operação de circuitos eletrônicos [17]. Após este acontecimento, a comunidade científica, agências espaciais e órgãos militares passaram a estudar os efeitos da radiação nos circuitos eletrônicos. Um segundo fato que levou a exploração desse assunto foi a queda de um avião Airbus A320 em fevereiro de 1990 na cidade de Bangalore na Índia, investigações preliminares sugeriram que os computadores de controle poderiam ter realizado algumas especificações segundos antes do início da queda [18].

A radiação está presente tanto no espaço quanto na atmosfera, podendo alterar a resposta ou danificar componentes eletrônicos expostos a íons pesados (partículas

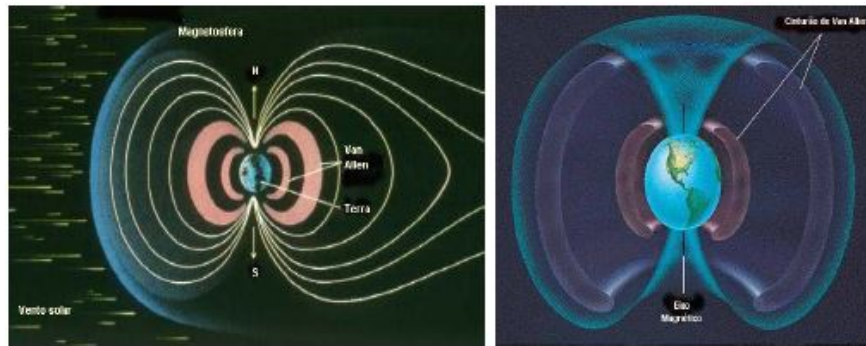


Figura 2.2: Cinturão de Van Allen [21].

carregadas), como por exemplo, os transistores. Circuitos eletrônicos podem sofrer efeitos indesejados e as principais partículas responsáveis por isso são prótons, elétrons, nêutrons, íons pesados e partículas alfa, além da radiação eletromagnética (como, por exemplo, raio-x). As principais fontes de radiação de origem espacial são os cinturões de *Van Allen*, os raios cósmicos [19] e a atividade solar [20]. Estas partículas podem gerar pulsos transitórios nos transistores que dependendo de sua amplitude em tensão, corrente e duração podem ser interpretados como sinal interno do circuito, gerando erros.

2.2.1 Cinturão de Van Allen

São imensas regiões de radiação dentro da magnetosfera repleta de prótons e elétrons energéticos presos pelo campo magnético da terra. Na Figura 2.2 mostra-se dois cinturões de elétrons (interno e externo), sendo o cinturão externo o que contém partículas com maior energia. O cinturão interno contém elétrons cuja energia é menor que 5 MeV e pode ser encontrado numa região de aproximadamente 100 km a 10.000 km de altitude. Já o cinturão externo contém elétrons cuja energia pode alcançar até 7 MeV e situa-se em altitudes de aproximadamente 20.000 km até 60.000 km [19]. Um terceiro cinturão de elétrons foi observado após uma tempestade magnética em 24 de março de 1991 [17].

2.2.2 Atividade Solar

O sol é responsável por grande parte da radiação presente no espaço. A atividade solar segue uma variação regular e periódica com 11 anos de duração, e este período é comumente conhecido como ciclo solar. O ciclo solar é a recorrência de periódicas manchas solares na superfície do Sol. Durante o ciclo solar ocorre uma mudança periódica na energia solar, radiação e a ejeção de material solar [22]. O período de 11 anos

correspondente ao ciclo solar está dividido em aproximadamente 7 anos de alta atividade e 4 anos de baixa atividade [20].

Durante a baixa atividade solar o sol emite rajadas de partículas energéticas no espaço. Essas podem ser chamadas de erupções solares, que são compostas principalmente de prótons, com uma menor quantidade de partículas alfa (5% a 10%), íons pesados e elétrons, ou seja, as explosões solares emitem uma quantidade relativamente menor do que o fluxo de raios cósmicos que viajam pelo sistema solar.

Durante a alta atividade solar o sol emite uma quantidade maior de íons pesados podendo aumentar em até quatro ordens de grandeza, ou seja, o fluxo de íons é maior do que os observados para os raios cósmicos, por períodos que podem chegar a vários dias [19].

A alta temperatura da coroa solar proporciona energia suficiente para que os elétrons escapem da atração gravitacional do sol. O efeito resultante da ejeção dos elétrons gera um desequilíbrio resultando numa ejeção de prótons e íons pesados da coroa solar. O vento solar é composto por aproximadamente 95% de prótons, 4% de íons de Hélio e 1% de outros íons pesados e elétrons com uma quantidade necessária para tornar o vento solar neutro [17].

2.2.3 Raios Cósmicos

O termo raio cósmico não possui uma definição científica clara. Ele tem sido utilizado desde o início do século XX para indicar as partículas energéticas que interferem com os estudos de materiais radioativos. Segundo Stassinopoulos e Raymond [19] os raios cósmicos consistem em cerca de 85% de prótons, cerca de 14 % de partículas alfa, e cerca de 1% de materiais mais pesados como, por exemplo, núcleo de carbono e ferro. Já Boudenot [20], afirma que a composição dos raios cósmicos galácticos compreende 83% de prótons, 13% de núcleos de hélio e 3% de elétrons.

As Partículas produzidas na atmosfera da Terra surgem quando os raios cósmicos primários atingem átomos atmosféricos e criam uma chuva de partículas secundárias. Esses são também chamados de partículas em cascata. As partículas que finalmente atingem a terra são chamadas de partículas terrestres, menos de 1% do fluxo primário atinge o nível do mar, e elas são na sua maioria compostas de múons, prótons, nêutrons e píons. A primeira observação de um *Single Event Upset* (SEU) na superfície terrestre devido a raios cósmicos ocorreu no ano de 1979 [23].

2.2.4 Partículas alpha

São compostas por dois nêutrons e dois prótons provenientes de um átomo de hélio duplamente ionizado a partir do decaimento nuclear de isótopos instáveis. No final da década de 70, as partículas alfa emitidas de materiais com traços de Urânio (U) e Tório (Th) foram mostradas como a causa dominante de um SEU numa memória *RAM* na superfície terrestre [24]. Estas partículas estão presentes nos materiais utilizados para o encapsulamento de circuitos integrados, no qual é necessária uma baixa concentração de Urânio (U) e tório (Th) para reduzir a emissão e o equilíbrio de partículas alfa, mas não sendo o suficiente para eliminá-las. Em situações de não equilíbrio, foi destacado que o material utilizado no processo de solda dos dispositivos, usualmente feitos de chumbo (Pb) e estanho (Sn), os quais são extraídos de minérios que podem conter traços de Urânio (U) e Tório (Th) que causam a incidência de partículas alfa. Por isso é aconselhável que projetistas não posicionem pontos de solda próximos aos nós dos circuitos [17].

2.2.5 Efeitos Singulares ou *Single Event Effects(SEE)*

Efeitos singulares são causados por uma única partícula e podem assumir muitas formas. Estão associados a erros transientes num dispositivo causado pela indução de partículas energéticas ou radiação cósmica [25]. O impacto de uma única partícula ionizada da origem a pares de elétrons ao longo da trajetória da partícula por meio de um material semicondutor, SEE podem ser classificados em vários tipos como por exemplo:

- *Single Event Upsets(SEU)*: Um SEU ocorre quando a incidência de uma partícula num dispositivo digital provoca mudanças indesejáveis no seu estado lógico, como por exemplo, a inversão de bits de elementos de memória. Esta inversão resulta de quando um pulso transitório incide num espaço de memória, esse efeito é visto como uma inversão do valor de armazenamento no *flip-flop*, ou seja, um *bit-flip* [26].
- *Single Event Transients(SET)*: São variações temporárias na tensão de saída de corrente ou de um circuito devido à passagem de um íon pesado através de um dispositivo sensível, ou seja, pulso transiente que pode ou não ser capturado por um elemento de memória [27].

2.3 Dependabilidade

Segundo Laprie [18], dependabilidade indica a qualidade do serviço fornecido por um dado sistema e a confiança depositada no serviço fornecido. Do ponto de vista etimológico ao que diz respeito ao termo dependabilidade, do inglês *dependability*, o termo confiabilidade, do inglês *reability* seria mais apropriado: a capacidade de confiar. Apesar de que *dependability* é sinônimo de *reability* [18].

Segundo Laprie e Weber [18, 16] tolerância a falhas e dependabilidade não são propriedades de um sistema a que se possa atribuir diretamente valores numéricos. Mas todos os atributos da dependabilidade correspondem a medidas numéricas. Seus principais atributos são dependabilidade, confiabilidade, disponibilidade, segurança de funcionamento (*safety*) e segurança (*security*). Um resumo dos principais atributos é mostrado na Tabela 2.1.

Atributo	Significado
Dependabilidade (<i>dependability</i>)	qualidade do serviço fornecido por um dado sistema
Confiabilidade (<i>reliability</i>)	capacidade de atender a especificação, dentro de condições definidas, durante certo período de funcionamento e condicionado a estar operacional no início do período
Disponibilidade (<i>availability</i>)	probabilidade do sistema estar operacional num instante de tempo determinado; alternância de períodos de funcionamento e reparo
Segurança (<i>safety</i>)	probabilidade do sistema ou estar operacional e executar sua função corretamente ou descontinuar suas funções de forma a não provocar dano a outros sistema ou pessoas que dele dependam
Segurança (<i>security</i>)	proteção contra falhas maliciosas, visando privacidade, autenticidade, integridade e irrepudiabilidade dos dados

Tabela 2.1: Resumo dos atributos de dependabilidade Retirado de Weber [16].

2.4 Tolerância a Falhas

Segundo Avizienis [28] quando um sistema é capaz de automaticamente se recuperar de erros causados por falhas, e eliminar uma falha sem sofrer um defeito externamente perceptível, diz-se que este sistema é tolerante a falhas. Na construção dos primeiros computadores, estratégias para construção de sistemas mais confiáveis já eram utilizadas para tolerar possíveis falhas [29]. Apesar de envolver técnicas e estratégias tão antigas, a tolerância a falhas ainda não é uma preocupação rotineira de projetistas e usuários, ficando sua aplicação quase sempre restrita a sistemas críticos e mais recentemente a sistemas de missão crítica [16].

Um aspecto importante em tolerância a falhas é a descrição das características

das falhas. Há um conjunto de atributos que são utilizados para cumprir esta finalidade; são eles: causa, natureza, duração, extensão e valor. Na figura 2.3 é mostrado os atributos das características das falhas.

Pêgo [30] descreve os conjuntos de atributos utilizados para caracterizar uma falha:

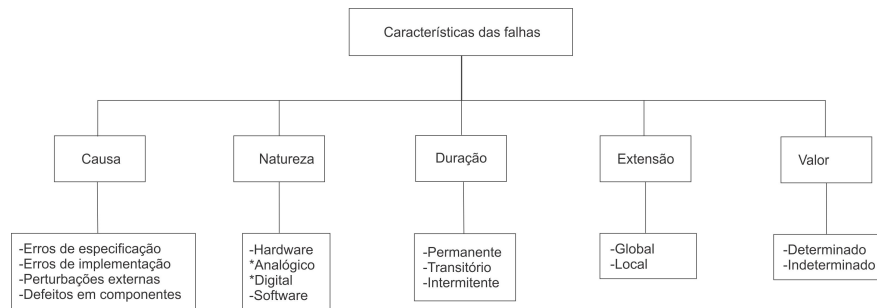


Figura 2.3: Atributos das características das falhas. Retirado de Pêgo [30].

- **Causa** - Uma falha pode ter origem em problemas de especificação, problemas de execução, defeitos em componentes do sistema operacional (que não são inco-muns para dispositivos eletrônicos), ou em fatores externos, tais como, tempesta-des, poeira, temperatura, etc.
- **Natureza** - Uma falha pode ser proveniente de *software* ou *hardware*. Neste último, a falha pode estar na parte analógica, por exemplo, em transdutores e amplificado-res, ou na parte digital, por exemplo, na unidade lógica Aritmética (ULA).
- **Duração** - Uma falha pode ser constante, o que significa que uma vez que os dados tenham sido persistidos no sistema, a falha continuará até que a manutenção adequada seja feita. Pode ainda ser transitória, quando ocorre em um período de tempo e logo em seguida, desaparece. Esse tipo de falha é geralmente provocada por causas externas. Relâmpago, por exemplo, podem provocar um erro súbito em um dispositivo, mas após o relâmpago, o dispositivo voltará ao seu funcionamento normal. Finalmente, há falhas, que são chamadas intermitentes; essas ocorrem em períodos curtos de tempo e desaparecem, mas depois elas voltam novamente. É possível que este processo se repita indefinidamente. Uma falha intermitente é a ocorrência de repetição de falhas transitórias.
- **Extensão** - A ocorrência de uma falha pode estar limitada a um escopo global ou local. Ou seja, uma falha pode afetar todo o sistema ou ser limitada a um determinado bloco.
- **Valor** - O valor de uma falha pode ser determinado ou indeterminado. Ou seja, os valores relativos de uma falha podem ser constantes ou não. Na Seção 2.1 é citado

um exemplo de uma falha que mantém um endereço de memória com um valor fixo em zero, este exemplo é chamado de valor determinado. Outra falha sem esta característica é chamada de indeterminada.

2.5 Técnicas de Tolerância a Falhas

Para mitigar(abrandar, minimizar) os efeitos citados na Subseção 2.2.5 e na Seção 2.1 são utilizadas técnicas de tolerância a falhas, nas quais envolvem alguma forma de redundância. Existem técnicas baseadas em *software* e *hardware*, neste trabalho foram citados apenas as técnicas baseadas em *software*, pois no desenvolvimento da aplicação proposta não foram utilizadas as técnicas baseadas em *hardware*, pois esta técnica demanda da utilização de um equipamento que realiza o bombardeamento de íons sobre o circuito eletrônico podendo danificar o *hardware* [16].

2.5.1 Técnicas de redundância baseadas em software

Segundo Pêgo [30] a inserção de redundância no código ou nos dados permite que seja possível detectar e até mesmo corrigir eventuais falhas. A inserção de instruções pode ser feita em programas escritos tanto em linguagem C, *assembly* e até em níveis mais baixos, a redundância temporal em nível de instruções pode ser dividida em:

- **Técnicas orientadas a dados** - Cada dado armazenado é replicado em cada operação, na checagem da consistência dos dados sendo necessária a alteração do código fonte. Pode ser implementado em linguagens de baixo nível como C, que será utilizada neste trabalho, *assembly* e até no código intermediário gerado pelo compilador [30].
- **Técnicas orientadas ao controle** - As falhas que podem modificar o fluxo correto de execução dos programas são detectadas e tratadas. Todas as técnicas no nível de instrução são baseadas na divisão do código do programa em blocos, construção de grafos e a checagem em tempo de execução sobre a correta transição entre os vértices deste grafo [30].

Weber [16] afirma que a simples replicação de componentes idênticos é uma estratégia de detecção e mascaramento de erros inútil em *software*. Componentes idênticos de *software* vão apresentar erros idênticos. Assim não basta copiar um programa

e executá-lo em paralelo ou executar o mesmo programa duas vezes em tempos diferentes. Erros de programas idênticos vão apresentar, com grande probabilidade, de forma idêntica para os mesmos dados de entrada. Segundo Brilliant *et al.*[31] outras formas de redundância de *software* como, por exemplo, diversidade ou programação n-versões, blocos de recuperação e variação de consistência não envolvem cópias idênticas.

2.5.2 Diversidade ou Programação N-Versões

A partir de um problema, são implementadas diversas soluções alternativas, sendo a resposta do sistema determinada por votação, esta técnica é ilustrada na Figura 2.4. Segundo Avizienis [32] A. *apud* Fischler *et. al.*, os esforços de programação são realizados por n indivíduos. Sempre que for possível, diferentes algoritmos e linguagens de programação são usados em cada versão. Cada versão do programa é implementada de forma independente com base na especificação inicial do problema, embora sejam diferentes na sua implementação, as N-versões são funcionalmente equivalentes e dificilmente apresentarão as mesmas falhas [32].

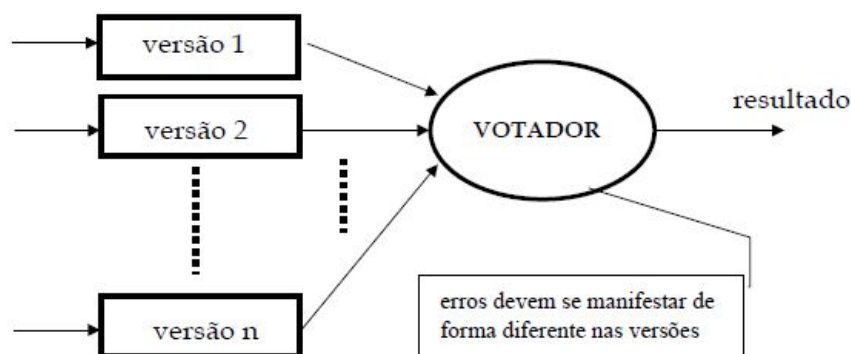


Figura 2.4: Diversidade ou Programação N-Versões. Retirado de Weber [16].

Um exemplo de programação n-versões é o sistema de bordo do *Space Shuttle*, no qual quatro computadores idênticos são utilizados em NMR (*N-Modular Redundancy*). Esta técnica consiste em replicar o *hardware* responsável pelo processamento da informação em n módulos. Um quinto computador com *hardware* diferente dos outros quatro, pode substituir os demais em caso de colapso no esquema NMR [33].

2.5.3 Blocos de Recuperação

Nesta técnica são utilizados testes de aceitação, no qual programas serão executados um a um até que o primeiro passe no teste de aceitação. A técnica de blocos de recuperação é semelhante a programação n-versões, mas nessa técnica programas secundários só serão necessários na detecção de um erro no programa primário. Esta técnica tolera $n-1$ falhas, no caso de falhas independentes nas n versões [1, 16, 34].

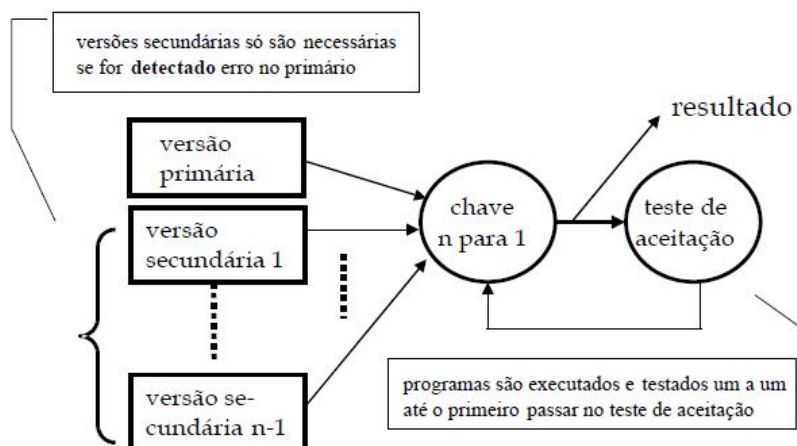


Figura 2.5: Blocos de Recuperação. Retirado de Weber [16].

2.5.4 Verificação de Consistência

É uma ampliação da programação n-versões. Nessa técnica também são utilizadas n equipes, assim como na programação n-versões, que implementam soluções independentes a partir de uma única especificação, no entanto, as equipes também desenvolvem um módulo específico para verificar a saída dos dados de seu próprio sistema em tempo real e compará-los com uma base de informações prévia para apurar a correção da informação. Os *softwares* das equipes são executados paralelamente e as saídas são submetidas a verificação. A saída passa a ser verificada em seu próprio módulo e o resultado é definido pelo primeiro módulo que passar ou por um sistema de votação [1, 3].

2.6 Injeção de Falhas

A injeção de falhas é um processo importante para validar e verificar a confiabilidade de um sistema, seja por alteração de código, simulando uma falha de *software*

[35] ou a nível de pinos (*Pin-level Injection*) injetando falhas diretamente no *hardware* [36]. Além da injeção de falhas por *software* e por *hardware* existe um terceiro tipo chamado injeção de falhas por Simulação. Arlat [36] afirma que injeção de falhas baseadas em simulação se restringem aos modelos de alto nível, como por exemplo os modelos VHDL (*VHSIC Hardware Description Language*) Linguagem de descrição de *hardware* de circuitos de alta velocidade.

Segundo Arlat *et. al.* [37] a validação e auxílio de projeto são as duas metas principais que compõem o método de injeção de falhas. Com um conjunto de testes a validação dos procedimentos de verificação são utilizados para descobrir falhas durante todo o processo de desenvolvimento. A validação dos mecanismos de tolerância a falhas é usada para detecção e recuperação de falhas com o objetivo de alcançar a dependabilidade do sistema na fase operacional. É importante ressaltar dois aspectos importantes na validação, que é a previsão de falhas, na qual os mecanismos de manipulação de falhas são avaliados em sua eficiência na estimação de medidas como cobertura e latência [37].

O auxílio ao projeto ocorre durante a fase de desenvolvimento do projeto, buscando melhorar a eficiência dos mecanismos de testes e o protocolo de tolerância a falhas por meio da injeção de falhas [37].

Uma das vantagens da utilização do método de injeção de falhas está no fato de ser uma técnica que permite a avaliação de um protótipo de sistema sob falhas, em particular ela mede a eficácia da detecção de erros do sistema e sua capacidade de correção. Outra vantagem são os efeitos das falhas no sistema que permitem revelar falhas críticas que por ventura viessem a ocorrer na execução do sistema em um ambiente real [36].

2.6.1 Injeção de falhas por *Hardware*

Esta técnica necessita de um *hardware* especial, no qual as falhas seriam originadas. Arlat [36] apresenta algumas ferramentas para injeção de falhas por *hardware* como *MESSALINE*, *RIFLE* E *AFIT*, todas utilizadas para injeção de falhas a *pin-level*, que consiste na injeção de falhas por meio de pinças ou posicionando o circuito sobre soquetes conectados a um injetor de falhas [37, 36]. Outras técnicas de injeção de falhas por *hardware* são interferências eletromagnéticas e irradiação de íons pesados [38, 37, 36], estas por sua vez podem danificar o componente sob teste [39]. O objetivo dessas técnicas visa principalmente estudar o comportamento dos mecanismos de tolerância a falhas implementados por *hardware*, porém também se pode testar os mecanismos de falhas por meio de *softwares* que não danificam os componentes sob teste [40].

2.6.2 Injeção de falhas por *Software*

Esta técnica visa modificar o estado do *hardware/software* do sistema por meio de um programa, fazendo com que o sistema se comporte como se uma falha de *hardware* estivesse ocorrendo. Pode-se emular falhas em vários níveis do sistema desde que a funcionalidade do *hardware* esteja visível através do *software*. Por isso, injetar falhas por *software* é menos dispendioso em termos de tempo e esforço do que as técnicas de *hardware* implementadas, devido a capacidade de alterar o estado de registradores e memória [35]. Um exemplo de injeção de falhas por *software* é a ferramenta *Ferrari* apresentada por Kanawati *et. al.* [35] que possibilita a injeção de falhas transitórias, bem como falhas permanentes, de modo que ele possa verificar a eficácia da detecção de erros e a correção simultânea dos mecanismos de verificação de falhas, e a capacidade para executar a injeção em código aberto. Alguns exemplos de injetores de falhas são FERRARI [35], PFI [41], SFI [42] e FIAT [43].

2.7 *Design Patterns*

Neste trabalho foi necessário a utilização de *design patterns* (padrões de projeto) para a elaboração de códigos eficientes. A ideia básica dos padrões de projeto é a de utilizar soluções conhecidas para problemas conhecidos. Cada padrão de projeto descreve um problema e sua solução, com isso, terceiros podem a utilizar sem ter a necessidade de descobri-la novamente, economizando tempo e padronizando o desenvolvimento do projeto [44]. Em geral, *design patterns* possuem alguns elementos que facilitam no desenvolvimento da sua aplicação [45], são eles:

- **Contexto** - Situação na qual o problema está sendo endereçado corretamente.
- **Problema** - Problema de design para o qual o padrão se destina.
- **Necessidade** - Quesitos de design para o qual o padrão se destina.
- **Solução e estrutura** - Solução do problema.
- **Consequências** - Vantagens e desvantagens de se utilizar o padrão.
- **Padrões associados** - Padrões similares ou utilizados para construir o padrão.

2.7.1 Padrão GoF (Padrões Fundamentais Originais)

O conceito de máquina de estados foi utilizado neste trabalho para o desenvolvimento da biblioteca *FaultRecovery*. O padrão *State* vai ao encontro da resolução deste problema.

Os padrões GoF são divididos em três grupos: padrões de comportamento, de criação e estruturais. Estes descrevem como os objetos devem ser gerenciados pelas estruturas de um programa, esses fornecem maneiras robustas de se criar objetos, e aqueles descrevem como os objetos interagem, distribuindo responsabilidades [45]. Neste trabalho utilizou-se o padrão *State*, indicado para programas que podem ser representados por uma máquina de estados, que é o caso dos *firmwares* desenvolvidos para o microcontrolador *mbed*. O padrão *State* faz parte dos padrões de comportamento e permite que parte do comportamento de um objeto seja alterado conforme o estado do objeto. Cada objeto possui atributos, que representam seu estado, e também métodos, que representam seu comportamento [44].

Capítulo 3

Metodologia

Este trabalho teve como objetivo ampliar as bibliotecas *FaultInjector* e *FaultRecovery*, ambas desenvolvidas por Kruger [3] em sua dissertação de mestrado. A biblioteca *FaultInjector* possibilita a injeção de falhas no sistema mediante a simulação do fenômeno *bit-flip* nas regiões da memória *SRAM* do microcontrolador. Já a biblioteca *FaultRecovery* proporciona o desenvolvimento de sistemas embarcados confiáveis, pois dispõe de técnicas de tolerância a falhas (baseadas na redundância de dados e de processamento) para aumentar a confiabilidade do sistema.

Foram utilizados exemplos variados para exemplificar as modificações realizadas na *FaultRecovery*, pois se pensou em criar uma biblioteca genérica que possa ser utilizada em diversos casos, seja no sistema embarcado (*firmware*) de um estacionamento, em um robô seguidor de linha ou em uma estação meteorológica.

Para a ampliação das bibliotecas citadas, utilizou-se um microcontrolador de prototipagem rápida *mbed*, modelo NXP LPC1768 [46] (o mesmo utilizado por Kruger). Este é projetado para a prototipagem de diversos equipamentos, especialmente aqueles que necessitam de conexão com a internet, portas USB e interfaces variadas para periféricos. Ele possui um núcleo ARM Cortex-M3 32 bits com 96 MHz de *clock*, 64 KB de memória *RAM* (32 KB disponíveis ao usuário e 32 KB reservados aos controladores internos do dispositivo), uma memória *flash* de 512 KB, portas *built-in Ethernet*, USB Host, CAN (*Controller Area Network*), SPI (*Serial Peripheral Interface*), I2C (*Inter-Integrated Circuit*), ADC (*Analog-to-Digital Converter*), DAC (*Digital-to-Analog Converter*), PWM (*Pulse-Width Modulation*) e outras interfaces de entrada e saída.

O *mbed* também possui um temporizador *watchdog*. Este componente consiste em um hardware responsável por reiniciar automaticamente o dispositivo em caso de falhas. O hardware temporizador é carregado com um valor inicial, decrementado a cada

vez que o *watchdog* é executado. Enquanto isso o programa principal executa em um *loop*, repondo o temporizador a cada vez que passa pelo circuito principal. Se por conta de uma falha a reposição não ocorrer e o temporizador zerar, o dispositivo é reiniciado [47].

No sítio oficial da plataforma *mbed* são fornecidas soluções para auxiliar no desenvolvimento de sistemas embarcados. Além da possibilidade de baixar bibliotecas fornecidas pela comunidade, o sítio disponibiliza um *forum* para retirada de dúvidas. A *mbed* também fornece um compilador online, com o qual após a compilação do código, um arquivo binário é gerado para ser executado no microcontrolador. Este compilador online tem a capacidade de compilar códigos para diversos modelos do *mbed* [48].

Além do modelo LPC1768, existem outros pertencentes à família *mbed* NXP LPC17X, são eles: LPC1764, LPC1765, LPC1766 [49]. Embora apresentem arquiteturas compatíveis, o tamanho e o mapa de memória *flash* e *SRAM* varia conforme cada modelo, distinguindo-se por exemplo, nos modelos LPC1768/66/65 que possuem regiões de memória *SRAM* nas mesmas faixas de endereço, mas diferentes do modelo LPC1764, conforme é mostrado na Figura 3.1 da seção a seguir.

3.1 Injetor de Falhas

A injeção de falhas é um processo importante para validar e verificar a confiabilidade de um sistema, seja por alteração de código, simulando uma falha de *software* ou a nível de pinos (*Pin-level Injection*), injetando falhas diretamente no hardware. A biblioteca *FaultInjector* permite simular o efeito do *bit-flip* em qualquer região de memória *SRAM* do microcontrolador *mbed* LPC1768, mas apresenta duas graves limitações: a falta de flexibilidade, por não executar em outros modelos de microcontroladores, e a impossibilidade de inserção de falhas nas regiões da memória *flash*. Essas limitações são descritas respectivamente nas subseções 3.1.1 e 3.1.2.

3.1.1 Mapeamento de Memória

Inicialmente, pensou-se em ampliar o uso da biblioteca *FaultInjector* mediante a criação de várias versões, cada uma específica a um modelo da família LPC176X. Assim, o programador ficaria responsável por baixar a versão correta. Entretanto, conforme as pesquisas evoluíram, percebeu-se que era possível identificar dentro do próprio código-fonte o modelo do microcontrolador, uma vez que o compilador mantém a informação do

modelo em uma *define* ¹. Com esta descoberta, foi possível criar uma única versão que automaticamente mapeia as regiões de memória.

Neste primeiro passo, observando o mapa de memória da família *mbed* LPC176X [49] foi possível constatar que alguns modelos possuem regiões de memória idênticas, conforme mostrados na Figura 3.1. Logo, o uso de heranças e classes abstratas poderiam ser utilizadas com o intuito de facilitar o mapeamento das regiões de memória e promover uma interface à biblioteca *FaultInjector*, que precisava injetar falhas nas regiões de memória sem necessariamente conhecer os endereços das regiões de cada dispositivo.

Com as regiões de memória mapeadas, o segundo passo foi a implementação desse mapeamento. A classe *MemoryRegion*[3] foi utilizada para representar as regiões de memória do *mbed*. Ela contém atributos que armazenam o endereço de memória inicial, o endereço final e o tamanho (em bytes) de cada região de memória.

A classe abstrata *MemoryMap* representa o mapa de memória de um microcontrolador *mbed* e disponibiliza uma interface para *FaultInjector* injetar falhas. Esta classe é herdada por classes não-abstratas, como a classe *MemoryMap_LPC1764* e *MemoryMap_LPC1768*, que obrigatoriamente, ficam responsáveis por implementar os métodos abstratos *getUserMemoryRegion()* e *getFlashMemoryRegion()* de *MemoryMap*, descritos a seguir:

- ***getUserMemoryRegions()*** - método abstrato que retorna uma lista das regiões de memória disponíveis para o usuário.
- ***getFlashMemoryRegions()*** - método abstrato que retorna a lista de regiões endereçadas a memória *flash*.
- ***getPeripheralsMemoryRegions()*** - retorna uma lista de regiões da memória destinadas aos periféricos. Este método não é abstrato, mas pode ser sobrescrito. Entretanto, como as regiões dos periféricos do microcontrolador é comum a todos eles, não há necessidade alguma sobrescrever este método.

Pela *FaultInjector* ser capaz de identificar o modelo do microcontrolador, ela mesma é responsável por escolher qual das versões de *MemoryMap* irá utilizar (*MemoryMap_LPC1764*, *MemoryMap_LPC1765*, entre outros). Isso tornou a biblioteca flexível, sendo possível injetar falhas nas regiões de memória mostradas na Figura 3.1. A biblioteca está disponível no endereço <https://github.com/cleitonlmeida1/FaultInjector>.

¹Constante

LPC1764		
FLASH	0x0000 0000 0x0002 0000	128 kb
Memory User	0x1000 0000 0x1000 4000	16KB SRAM
	0x2007 C000 0x2000 4000	16BK AHB SRAM
LPC1766/65		
FLASH	0x0000 0000 0x0004 0000	265 KB
LPC1768		
FLASH	0x0000 0000 0x0008 0000	512 KB
LPC1768/66/65/64		
Peripherals	0x4000 0000 0x4008 0000	APB0
	0x4008 0000 0x4010 0000	APB1
	0x4200 0000 0x4400 0000	peripheral bit band alias addressing
	0x5000 0000 0x5020 0000	AHB peripherals
	0xE000 0000 0xE010 0000	private peripheral bus
LPC1768/66/65		
Memory User	0x100 04000 0x100 08000	32 KB SRAM
	0x2007 C000 0x2008 0000	16KB AHB SRAM
	0x2008 0000 0x2008 4000	16 kB AHB SRAM1

Figura 3.1: Mapeamento das regiões de memória dos modelos LPC1768/66/65/64.

3.1.2 Injeção de Falhas na Memória Flash

Os microcontroladores *mbed* possuem memória *flash*, na qual o código do *firmware* é armazenado [49]. Logo, a alteração de um único bit em um endereço de memória que contenha partes de uma instrução pode resultar em um falha irreversível. No entanto os dados coletados por um sistema também podem ser armazenados na memória *flash*. Considerando-se o sistema embarcado de uma estação meteorológica que precisa coletar os dados climáticos e armazená-los na memória não volátil. Estes dados armazenados na *flash* são enviados a um servidor remoto após um período de tempo predefinido. Se um único bit da região de memória na qual os dados coletados estão armazenados for alterado (*bit-flip*), esta alteração poderá provocar uma modificação no valor da informação armazenada que poderia ser, por exemplo, um valor da umidade do ar, que perderia a exatidão após sua alteração, ou seja, o dado coletado enviado para o servidor remoto

estaria incorreto, impactando em uma equivocada previsão do tempo.

Conforme explicado na Seção 2.6, um dos aspectos positivos na simulação de falhas por software é a segurança de não danificar o dispositivo. Falhas podem ser inseridas em bytes aleatórios na memória *SRAM* simulando o fenômeno *bit-flip*, entretanto, a mesma lógica não pode ser aplicada na memória *flash*, pois por ser protegida não é possível escrever um único endereço de memória, somente a escrita por setores que são no total 29, divididos em duas áreas. A primeira, que abriga os setores do 1 ao 15 contém blocos de 4KB, a segunda, que vai do setor 16 até o 29, é composta por blocos de 32KB para cada setor [49].

A injeção de falhas na *flash* foi realizada mediante a escrita de bytes em um setor de memória sorteado aleatoriamente. Foram copiado 256 bytes da memória *SRAM*, que é a quantidade mínima de bytes que podem ser escritos com a biblioteca retirada do repositório de bibliotecas da *mbed* [50], para a *flash* utilizando uma biblioteca que permite fazer operações na memória não volátil.

Para realizar a injeção de falhas na flash foi necessário seguir os seguintes passos:

- Sortear um setor aleatório da memória flash.
- Copiar a quantidade de bytes escolhida da memória *RAM* podendo variar entre 256, 512, 1024 ou 4096.
- Preparar o setor para escrever os bytes copiados da memória *RAM* determinando qual o número do setor inicial e qual o do setor final.
- Após a delimitação do setor inicial e final a escrita dos bytes copiados da memória serão escritos na flash.

Neste trabalho foi realizada a escrita de 256 bytes na memória *flash*, ou seja, foram injetadas 256 bytes de falhas em setores aleatórios. As regiões da *flash* armazenam dados estáticos e as instruções do programa, logo, falhas nesses setores podem causar uma falha irreversível no sistema afetando o seu correto funcionamento. Porém a injeção de falhas não foi realizada durante a execução de algum programa, pois faltou tempo hábil e o interesse era explorar a memória *flash* ou seja, descobrir se seria possível injetar falhas nessa região de memória, uma vez que quando Kruger [3] tentou injetar falhas na memória *flash*, o microcontrolador travava durante a execução do injetor.

3.2 FaultRecovery: Extensão da biblioteca

Inicialmente pensou-se em ampliar a biblioteca *FaultRecovery* utilizando a sua estrutura inicial, que implementa macros e funções de *callback* (função executada conforme a ocorrência de um evento predefinido). No entanto, o uso de macros como funções não é uma boa prática de programação [51], pois além de dificultarem o entendimento da estrutura do código, criam outros males que podem resultar em falhas inesperadas. Por exemplo, uma macro que chama uma função *f* para que se retorne o maior entre os argumentos, se chamada com um dos parâmetros sob incremento, pode gerar erros imprevistos, conforme mostram os Quadros 1 e 2.

```
1 #define CALL_WITH_MAX(a, b) f((a) > (b) ? (a) : (b))
```

Quadro 1: Macro que chama *f* como o máximo entre *a* e *b*

```
1 int a = 5, b = 0;  
2 CALL_WITH_MAX(++a, b);  
3 CALL_WITH_MAX(++a, b + 10);
```

Quadro 2: Na primeira invocação da macro a variável *a* é incrementada duas vezes e na segunda uma vez.

Na *FaultInjector* de Kruger, as macros eram utilizadas demasiadamente, por isso foram substituídas por funções *templates*. Outra modificação importante foi o emprego do padrão de projeto *State*. Esse é indicado para programas que implementam máquinas de estados e teve como objetivo facilitar o uso da biblioteca, pois trouxe uma estrutura capaz de gerenciar automaticamente as mudanças de estados.

3.2.1 Refatoração e Aperfeiçoamento: Versão 1.0

O emprego do padrão de projeto *State* na refatoração da biblioteca, tem como um de seus principais objetivos forçar o usuário a implementar seu *firmware* como uma máquina de estados. Embora todo *firmware* seja uma máquina de estados, a maioria deles não são orientados a objetos e tão pouco legíveis. Foi Implementada uma versão dessa biblioteca para *arduino* e está sendo utilizada no projeto de extensão Coxim Robótica, do Campus de Coxim - UFMS para o desenvolvimento de um carrinho seguidor de linha. Na Figura 3.2 é mostrado o diagrama de classes da biblioteca *FaultRecovery*.

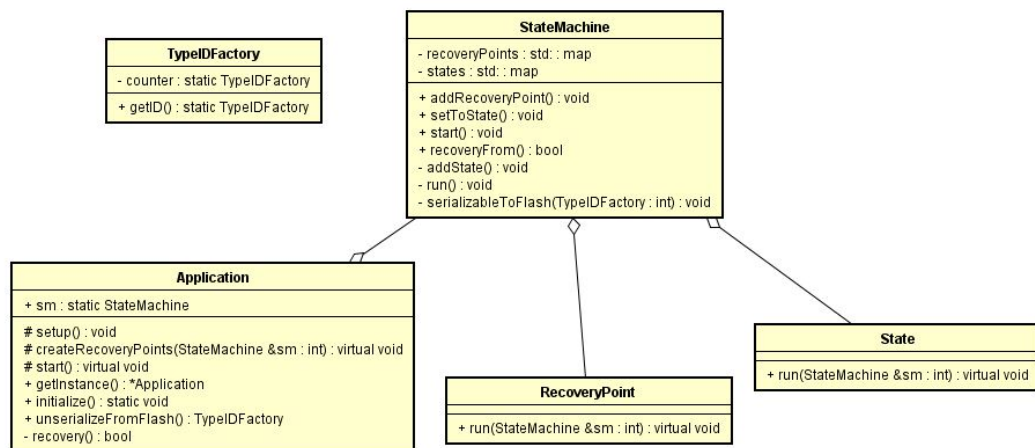


Figura 3.2: Diagrama de classes da biblioteca *FaultRecovery*.

O uso do padrão de projeto mencionado e a aplicação dos conceitos de orientação a objetos possibilitou a criação de programas modularizáveis, em outras palavras, o código do *firmware* pode ser separado por responsabilidades. Cada estado contém a sua classe de implementação, permitindo a separação do código e facilitando o entendimento da equipe. A primeira versão da biblioteca possuía três classes: *TypeIDFactory*, *State* e *StateMachine*, descritas abaixo:

- ***TypeIDFactory*** - Esta classe é responsável pela geração dos IDs de identificação dos estados. Os identificadores são únicos e auto-incrementais. Estes IDs são utilizados para indexar os estados em um *hash map*.
- ***State*** - Esta classe é responsável pela representação de um estado. Composta por um método abstrato *run*, é a classe base para todos os estados do *firmware*. No método *run* o usuário implementa as rotinas de execução do estado. Caso ao final dessa rotina haja uma transição para um novo estado, o método *setToState* deve ser invocado passando entre os sinais (< >) a classe que representa o próximo estado a ser executado, conforme demonstrado no Quadro 3. Apenas a classe é passada como parâmetro do *template*, uma vez que a própria *StateMachine* é responsável pela alocação dos objetos em seu *hash map*. Essa abordagem evita cópias desnecessárias e perda de desempenho.
- ***StateMachine*** - Esta classe representa a máquina de estados e tem a responsabilidade de controlar o despacho das funções implementadas em cada estado. Funciona como uma espécie de “engrenagem”, que controla as transições entre os diversos estados do sistema, instanciando e adicionando os novos estados em um

hashmap e selecionando-os a cada transição do programa. A escolha de um *hash map* deveu-se ao seu desempenho ser superior a outras estruturas de dados, uma vez que os estados são indexados pelos seus IDs.

Conforme mostrado na Figura 3.3 no momento em que o programa recebe um evento responsável pela mudança de estado, o método *setToState* desta classe recebe o valor da variável *current* (determina o estado atual). Se o estado não estiver sido adicionado no *map* de estados da máquina de estados, o método *addState* será chamado para incluir esse novo estado.

Para executar a máquina de estados, deve-se instanciar um objeto *StateMachine* e chamar o método *start*, passando entre (< >) o estado inicial conforme demonstrado no Quadro 3, o método *start* chamará o método *setToState* e o método *run*. Este Executará a máquina de estados por meio de um *loop*. Aquele atribuirá o id do estado inicial para a variável *current*.

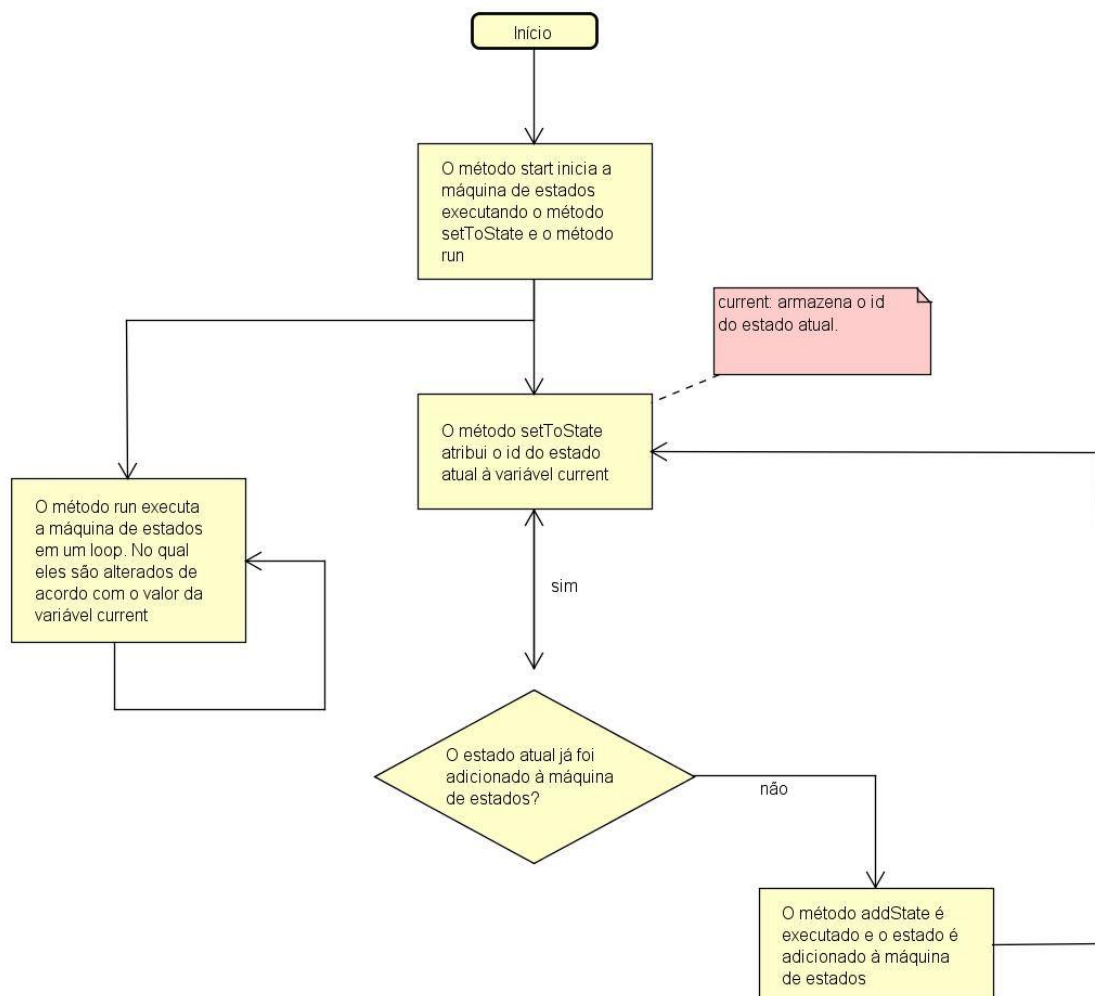


Figura 3.3: Fluxograma da biblioteca *FaultRecovery*.

```

1      class EsperandoCarroChegar: public State {
2          void run(StateMachine &sm) {
3              if (carroChegou()) {
4                  sm.setToState<EsperandoApertarBotao>();
5              }
6          }
7      };
8      int main() {
9          StateMachine sm;
10         sm.start<EsperandoCarroChegar>();
11     }

```

Quadro 3: A classe `EsperandoCarroChegar` herda da classe `State` e tem sua rotina implementada no método `run`, ao encerrar a rotina o método `setToState` é invocado, alterando o estado atual. No método `main` um objeto `StateMachine` é instanciado e o método `start` é chamado, iniciando a máquina de estados. Este quadro tem como objetivo demonstrar a utilização da biblioteca, apenas um estado será posto no quadro, os demais podem ser acessados no endereço <https://github.com/cleitonlmeida1>.

Um exemplo simples para ilustrar a utilização da nova *FaultRecovery* é o programa de estacionamento de um shopping. A entrada do estacionamento possui um emissor de *tickets*, dois sensores e uma cancela. Ao se aproximar da cancela, o motorista aperta um botão para emitir um *ticket*, após a emissão a cancela é aberta, os sensores detectam a entrada do carro ao interior do estacionamento e a cancela é fechada.

A máquina de estados do caso ilusório é composta por quatro estados, um deles é demonstrado no Quadro 3:

- **EsperandoCarroChegar** - Este é o estado inicial da máquina de estados. O sensor de presença detecta a chegada do veículo na entrada do estacionamento, o carro se aproxima da entrada, então o estado `EsperandoCarroChegar` detecta a aproximação do veículo e chama o próximo estado.
- **EsperandoApertarBotao** - Neste estado o dispositivo eletrônico emite o *ticket* de estacionamento após o motorista apertar o botão. Após isso, o estado atual é alterado para `EsperandoCarroEntrar`. Outra situação é quando o motorista decide não adentrar ao estacionamento, neste caso o estado é alterado para o anterior (`EsperandoCarroChegar`).

- **EsperandoCarroEntrar** - Neste estado os sensores estão aguardando a entrada do carro ao estacionamento. O sensor externo ao estacionamento detecta duas situações, a primeira é quando o carro se afasta da cancela e não entra no estacionamento, neste caso a máquina de estados retorna ao seu estado inicial. O segundo é quando o carro adentra ao estacionamento, neste caso o sensor externo identifica a entrada do carro, o interno ao estacionamento detecta o carro se afastando da cancela, com o seu afastamento o sistema altera o estado para **FecharCancela**.
- **FecharCancela** - Neste estado o carro se afasta da entrada do estacionamento, o sensor externo a esse detecta o afastamento do carro e a cancela é fechada. Ao fechar a cancela o estado **FecharCancela**, por ser o último estado da máquina de estados, irá chamar o estado inicial, para então reiniciar o ciclo de execução dos estados.

3.2.2 Refatoração e Aperfeiçoamento: Versão 2.0

Na Subseção 3.2.1 que trata da versão 1.0 da biblioteca, mostrou-se como implementar os estados de uma máquina de estados. Esta subseção trata da criação dos pontos de recuperação, como inicializar a máquina de estados e quais as classes adicionadas na biblioteca e suas responsabilidades. Nesta etapa de desenvolvimento, focou-se na tolerância a falhas. A versão 2.0 da biblioteca *FaultRecovery* utiliza a máquina de estados desenvolvida na versão 1.0, no entanto foi adicionada a funcionalidade que possibilita a criação de pontos de recuperação de falhas. Se o microcontrolador trava durante a execução da máquina de estados, o defeito é detectado pelo *whatchdog*, já que o temporizador tem seu valor de limiar excedido, e então o *whatchdog* reinicia o dispositivo. Ao restaurar, é possível diagnosticar o último estado seguro, pois esta informação foi gravada na *flash* durante as trocas de estado. Caso exista algum ponto de recuperação configurado para este último estado seguro, será executado.

O usuário da biblioteca deve criar os pontos de recuperação conforme seja necessário, sendo possível adicionar no máximo um ponto de recuperação por estado. Estes pontos indicam quais as rotinas deverão ser executadas caso o programa falhe durante a execução do estado especificado. Essa rotina pode executar uma determinada configuração no microcontrolador ou uma tarefa que deva ser iniciada antes da máquina de estados ser inicializada.

As classes *Application* e *RecoveryPoint* foram adicionadas à biblioteca. A primeira é responsável pela inicialização do *firmware* e gerenciamento dos pontos de recuperação,

a segunda é responsável pela implementação dos pontos de recuperação. Para utilizar a versão 2.0, deve-se instanciar um objeto *Application* e chamar o método *initialize* da seguinte forma: *Application:initialize<EstacionamentoApp>*. A classe *EstacionamentoApp* é herdada de *Application*, que possibilita a implementação de três métodos, sendo o método *start* obrigatório (pois é puramente virtual) e os métodos *creatRecoveryPoints* e *setup* opcionais (pois não são puramente virtuais). O método *initialize* é *static*, ou seja, ele pode ser chamado a partir de um código externo à classe sem a necessidade de criar uma nova instância de *Application*. Ao evocar o método *initialize*, um objeto *StateMachine* é inicializado, uma instância da classe *EstacionamentoApp* também é inicializada, essa será utilizada durante toda a execução da máquina de estados. Após o *EstacionamentoApp* ser inicializado, quatro métodos serão evocados automaticamente.

O primeiro é o método *setup* que poderá ou não ser implementado. Neste método são implementadas as configurações do microcontrolador, que podem variar dependendo do seu modelo, vale ressaltar que a utilização desta biblioteca não se limita ao microcontrolador *mbed*, pois os extensionistas do projeto Coxim Robótica do Campus de Coxim - UFMS já estão utilizando a arquitetura da biblioteca *FaultRecovery* adaptada para a plataforma *arduino*, por exemplo, no método *setup* os extensionistas programaram as configurações iniciais de um robô seguidor de linha e cada aluno ficou responsável por implementar um estado da máquina de estados do carrinho.

O segundo é o método *createRecoveryPoints*, que poderá ou não ser implementado, pois por padrão esse método não adiciona nenhum ponto de recuperação à máquina de estados. Entretanto pode ser sobrescrito, caso necessário, para que pontos de recuperação sejam adicionados à máquina de estados. Para implementar um ponto de recuperação deve-se criar uma classe que herde de *RecoveryPoints*.

Ao herdar dessa classe, o usuário obrigatoriamente deverá implementar o método abstrato *run* no qual conterá as rotinas de execução caso o programa seja restaurado daquele ponto. Para adicionar um ponto de recuperação à máquina de estados, deve-se utilizar o método *addRecoveryPoint* nativo da classe *StateMachine* e evocá-lo conforme demonstrado no quadro Quadro 4. Esse receberá entre *<>* o ponto de recuperação e o estado ao qual ele pertence, sendo assim o ponto de recuperação fica associado ao estado, em outras palavras, o id de identificação do estado no *hashmap* de estados será o mesmo do seu ponto de recuperação no *hashmap* de pontos de recuperação.

Como mencionado, na versão 2.0 o método *setToState* da classe *StateMachine* além de trocar o estado atual, também armazena o seu id na memória flash do microcontrolador *mbed*. Quando o dispositivo for reiniciado, se existir um ponto de recuperação para o id armazenado na memória *flash* a biblioteca executará o ponto de recuperação

referente ao id lido da memória *flash*.

O terceiro é o método *start* no qual é realizada a inicialização da máquina de estados conforme demonstrada no segundo parágrafo desta Subseção.

```
1      //Estacionamento App Class
2      void EstacionamentoApp::createRecoveryPoints(StateMachine &sm) {
3          sm.addRecoveryPoint<RecoveryFecharCancela, FecharCancela>();
4      }
5
6      //RecoveryFecharCancela Class
7      void RecoveryFecharCancela::run(StateMachine &sm) {
8          if (checarPortaoAberto()) {
9              fecharPortao();
10         }
11         sm.start<EsperandoCarroEntrar>();
12     }
```

Quadro 4: Este quadro demonstra a implementação do método *createRecoveryPoints* implementado no *EstacionamentoApp* que herda de *Application*. O ponto de recuperação é criado quando o método *addRecoveryPoint* é evocado, percebe-se que existem duas classes separadas por vírgula e entre *<>*, a primeira é a implementação do ponto de recuperação, ou seja, a classe que herda de *RecoveryPoints*. A segunda é a classe que representa o estado que deverá ser executado após a reinicialização do microcontrolador, ou seja, caso ele inicie neste ponto de recuperação o estado *EsperandoCarroEntrar* deverá ser escutado. No método *run* da classe *RecoveryEsperandoCarroEntrar* encontra-se o código que será executado caso o microcontrolador falhe no momento em que o carro iria entrar no estacionamento. Se a energia acabasse durante o fechamento da cancela e o programa tivesse voltado com a cancela aberta, o dono do estacionamento teria prejuízo, pois alguns carros poderiam entrar sem pagar a taxa de estacionamento enquanto a cancela estivesse aberta.

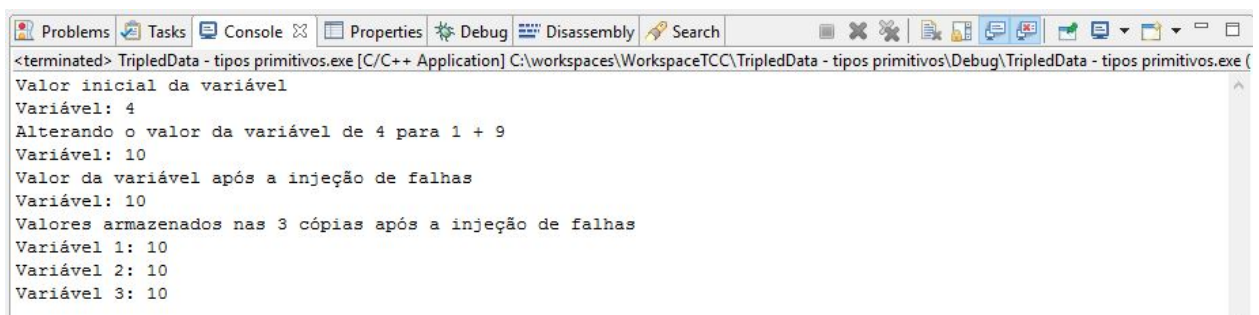
3.3 Classe de Redundância de Dados: *TData*

A classe *TData* foi implementada com o objetivo de obter-se uma redundância de dados automatizada, tanto para variáveis primitivas (int, float, long, double, char, bool), quanto para objetos de uma classe. Existem duas maneiras de se criar um objeto *TData*, a primeira é utilizando tipos primitivos e a segunda objetos.

Ao instanciar um objeto *TData*, deve-se especificar o seu tipo e passar um valor ou um objeto no construtor: *TData*<tipo_da_variável> variavel(valor). Ou o valor, ou o objeto passado como parâmetro serão copiados para três cópias de segurança, das quais serão utilizadas para manter a integridade do valor original por meio de um sistema de votação, que verifica se os valores das cópias são consistentes.

3.3.1 Classe *TData* com Tipos Primitivos

O método *setData* (pertencente a classe *TData*) será invocado automaticamente ao passar o número inteiro 4 como parâmetro no construtor da classe. Ao alterar o valor 4 para a operação $1 + 9$, o método *setData* será implicitamente chamado por meio de sobrescrita de operadores, disponível na linguagem C++, possibilitando que o valor da variável seja atualizado juntamente com as cópias de segurança. Vale ressaltar que a cada vez que o objeto *TData* for acessado, o método *getByVotting()* que implementa um esquema de votação será executado, validando todas as cópias do objeto *TData*. A injeção de falhas foi realizada por meio do método temporário chamado *injectFault*, que modificou o valor de uma das cópias do objeto *TData*, simulando o fenômeno *bit-flip*. Esse método foi utilizado durante a implementação da classe *TData*, sendo retirado dela após sua conclusão. Na Figura 3.4 é mostrado os valores das cópias após a injeção de falhas.



```
<terminated> TripledData - tipos primitivos.exe [C/C++ Application] C:\workspaces\WorkspaceTCC\TripledData - tipos primitivos\Debug\TripledData - tipos primitivos.exe (
Valor inicial da variável
Variável: 4
Alterando o valor da variável de 4 para 1 + 9
Variável: 10
Valor da variável após a injeção de falhas
Variável: 10
Valores armazenados nas 3 cópias após a injeção de falhas
Variável 1: 10
Variável 2: 10
Variável 3: 10
```

Figura 3.4: Nesta figura é mostrado que o valor das cópias foram atualizados após a operação $1 + 9$. Também é mostrado que os valores das cópias continuaram consistentes após a injeção de falhas.

3.3.2 Um Exemplo Ilusório Para Utilização da Classe *TData* com Objetos

Para demonstrar como utilizar e instanciar a classe *TData* com objetos, foi utilizado um exemplo que permitiu a utilização de objetos heterogêneos, ou seja, objetos que possuem outros objetos dentro de si, que podem ser objetos de pilha ou ponteiros para um endereço de memória. Destaca-se, um caso ilusório de um automóvel inteligente interligado a diferentes tipos de dispositivos e formas de comunicação, comunicando-se a um servidor.

As informações disponibilizadas pelo veículo são navegação, diagnósticos do funcionamento do próprio veículo, dentre outras informações. Este por sua vez sofre um acidente e o sistema se encarrega de acionar outro sistema, ou seja, o sistema embarcado instalado no carro se comunica com outro remotamente. Este, por sua vez, receberia todos os dados relativos ao paciente, inclusive sua localização para um possível deslocamento de ambulâncias [52]. Este exemplo foi implementado de maneira simples com o objetivo de demonstrar como instanciar e utilizar a classe *TData*.

Foram criadas duas classes para compor o cenário, sendo elas as classes Carro e Emergência, essas classes tiveram seus operadores de igualdade implementados pois, a classe *TData* obriga que suas implementações sejam realizadas, visto que o compilador não é capaz de definir o comportamento de uma definição de igualdade em um objeto. Cada objeto define a igualdade de um jeito diferente, a classe *string* por exemplo, define que igualdade são objetos com o mesmo conteúdo, ainda que sejam objetos distintos, ou seja, em endereços de memória diferentes.

A implementação dos operadores de igualdade para tipos primitivos não é obrigatória pois são tipos homogêneos, já os objetos podem ter outros objetos em seu escopo, como é o exemplo da classe Carro, que contém uma instância da classe Emergência em seu escopo, tornando obrigatória a implementação de seus operadores de igualdade, para que possam ser utilizados na classe *TData*.

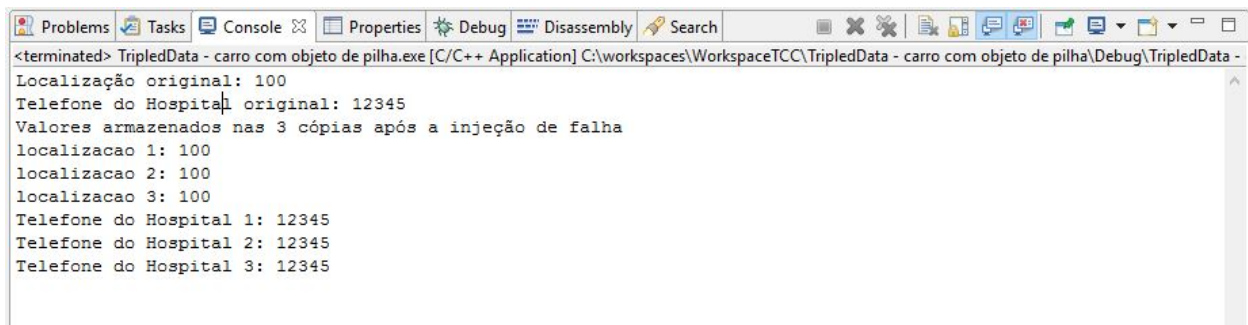
3.3.3 Classe Carro com Objeto de Pilha

Para instanciar um objeto *TData* do tipo Carro, primeiro deve-se declarar um objeto carro e setar os valores de seus atributos e consecutivamente instanciar um objeto *TData* do tipo carro passando o objeto carro recém criado como parâmetro no construtor da classe com os seus valores preenchidos.

Para este exemplo utilizou-se as classes Carro e Emergencia mencionadas nas

Seções anteriores e o método *injectFault* utilizado para simular o fenômeno *bit-flip*. A injeção de falhas foi realizada com métodos temporários implementados dentro da classe *TData* alterando os valores do telefone do hospital pertencente a classe *Emergencia* e da localização do carro pertencente a classe *Carro*.

O carro possui um sistema inteligente interligado com outros sistemas, ao sofrer o acidente automaticamente o sistema embarcado instalado no carro iria se conectar a outro sistema pedindo socorro e enviando sua localização para o hospital, essa por sua vez poderia ser alterada por alguma falha em seu endereço de memória podendo causar a morte do motorista, pois a ambulância seria enviada para outro endereço que não fosse o do carro. Após a injeção de falhas, os valores de todas as cópias podem ser visualizados na Figura 3.5 demonstrando que mesmo após a ocorrência de falhas nos endereços de memória da localização e do telefone do hospital, as cópias continuaram consistentes.

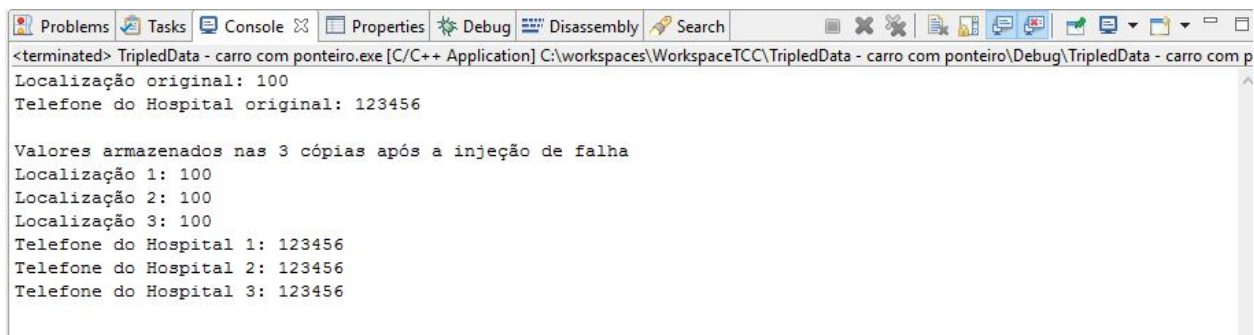


```
<terminated> TripleData - carro com objeto de pilha.exe [C/C++ Application] C:\workspaces\WorkspaceTCC\TripleData - carro com objeto de pilha\Debug\TripleData -
Localização original: 100
Telefone do Hospital original: 12345
Valores armazenados nas 3 cópias após a injeção de falha
localizacao 1: 100
localizacao 2: 100
localizacao 3: 100
Telefone do Hospital 1: 12345
Telefone do Hospital 2: 12345
Telefone do Hospital 3: 12345
```

Figura 3.5: Nesta figura é mostrado que valores das cópias continuaram consistentes após a injeção de falhas.

3.3.4 Classe Carro com Ponteiro

Este parágrafo tem por objetivo demonstrar que a classe *TData* também funciona com ponteiros. A classe *TData* também pode receber como parâmetro em seu construtor objetos que contenham referência para endereços de memória. Realizou-se uma injeção de falhas no ponteiro emergencia (armazena o telefone do hospital), alterando o seu endereço de memória por meio do método *injectFault*. Os valores do telefone e da localização após a injeção de falhas é mostrado na Figura 3.6, mostrando que mesmo após a modificação do endereço de memória do ponteiro emergência em uma das cópias, o valor do telefone do hospital continua o mesmo para todas as cópias.

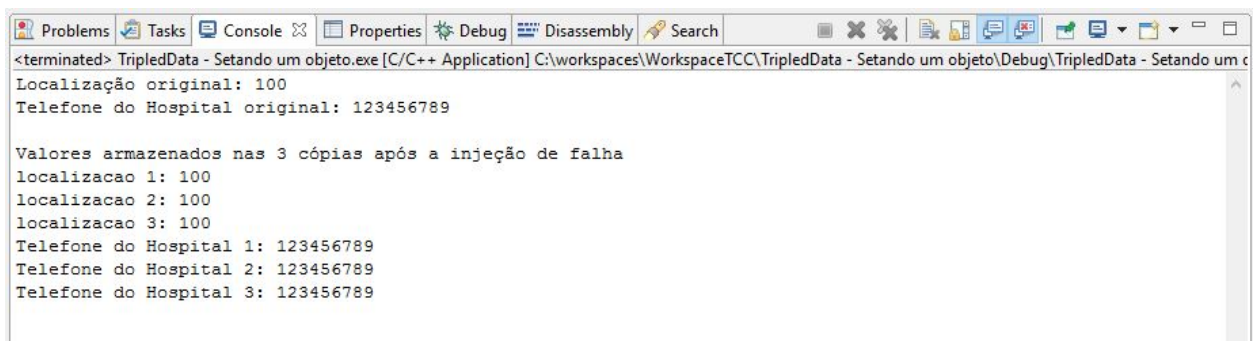


```
<terminated> TripleData - carro com ponteiro.exe [C/C++ Application] C:\workspaces\WorkspaceTCC\TripleData - carro com p
Localização original: 100
Telefone do Hospital original: 123456

Valores armazenados nas 3 cópias após a injeção de falha
Localização 1: 100
Localização 2: 100
Localização 3: 100
Telefone do Hospital 1: 123456
Telefone do Hospital 2: 123456
Telefone do Hospital 3: 123456
```

Figura 3.6: Nesta figura é mostrado que os valores das cópias continuaram consistentes após a injeção de falhas.

Além das três cópias de segurança que compõem a classe *TData* existe mais uma cópia chamada *dataObject*, utilizada para atualizar o objeto *TData* sem a necessidade de passar o objeto modificado como parâmetro no construtor da classe. A variável *dataObject* é utilizada para replicar uma alteração no objeto para as demais cópias. Para acessar o objeto *dataObject* é necessário chamar o método *getDataObject* implementado para atualizar as três cópias e executar o método *getByVotting* para manter a consistência de todas as cópias. Na Figura 3.7 é mostrada a atualização do telefone do hospital e os valores das cópias após a injeção de falhas.



```
<terminated> TripleData - Setando um objeto.exe [C/C++ Application] C:\workspaces\WorkspaceTCC\TripleData - Setando um objeto\Debug\TripleData - Setando um c
Localização original: 100
Telefone do Hospital original: 123456789

Valores armazenados nas 3 cópias após a injeção de falha
localizacao 1: 100
localizacao 2: 100
localizacao 3: 100
Telefone do Hospital 1: 123456789
Telefone do Hospital 2: 123456789
Telefone do Hospital 3: 123456789
```

Figura 3.7: Figura que apresenta a saída com os valores das cópias consistentes após a atualização do objeto *TData* do tipo carro e da injeção de falhas.

Capítulo 4

Resultados

Neste capítulo são apresentados os resultados dos testes realizados.

4.1 Desempenho da Biblioteca *FaultRecovery*

Esta Seção tem como objetivo demonstrar o desempenho da biblioteca *FaultRecovery* em termos de tempo de execução. Para realizar os testes, utilizou-se cinco algoritmos de ordenação *bubble sort*, *insertion sort*, *merge sort* e *comb sort* [53, 54], a execução dos cinco algoritmos forma um ciclo de teste. Ao todo cada ciclo foi executado cem vezes. Os algoritmos foram implementados para ordenar os elementos em forma crescente em um microcontrolador *mbed* LPC1768. Para cada ciclo de teste, utilizou-se um vetor do tipo *unsigned short* contendo 4096 elementos enumerados em ordem decrescente de n (4096) até 1, ou seja, o elemento da primeira posição recebeu o valor n , o da segunda $n - 1$, e assim sucessivamente até o elemento da última posição receber o valor 1. Essa estratégia foi utilizada para que o tempo de execução dos algoritmos fosse similar, pois se o vetor estivesse com números aleatórios ao invés de decrescente, isso poderia afetar o comportamento dos algoritmos. Por isso tentou-se manter um padrão para que fosse possível medir o tempo da biblioteca. O tamanho total do vetor totalizou 8kB de memória (2 bytes * 4096). Tentou-se aumentar a quantidade de elementos do vetor, entretanto quando se tentou alocar um espaço de memória maior que 8kB, o *firmware* teve sua execução interrompida no primeiro ciclo de teste, em outros no segundo.

O primeiro teste teve como objetivo medir o tempo de execução dos algoritmos de ordenação, simulando um *firmware* implementado por um usuário que não utilizou a biblioteca *FaultRecovery*. Todos os algoritmos compartilharam do mesmo vetor, por isso, antes de cada ordenação foi necessário desordenar o vetor para cronometrar o tempo

real de ordenação. O tempo de cada algoritmo foi cronometrado, desde o início de sua execução até seu fim, sendo desprezado o tempo de desordenação do vetor.

O segundo teste teve como objetivo medir o tempo de execução dos mesmos algoritmos de ordenação, fazendo uso da biblioteca *FaultRecovery*. O tempo de execução no segundo teste foi medido a partir do início da execução de um estado da máquina de estados até o fim dele. Tanto para o primeiro quanto para o segundo teste foi utilizada a mesma quantidade de elementos do vetor (4096), a mesma sequência de execução (*bubble, insertion, selection, merge e comb*) e o tempo de desordenação do vetor também foi desprezado. Ao final das cem execuções do primeiro e do segundo teste, as médias de tempo de execução para cada algoritmo e para cada ciclo de teste foram calculadas. O tempo de execução da biblioteca *FaultRecovery* também foi calculado, obtendo-se o resultado mostrado na Figura 4.1.

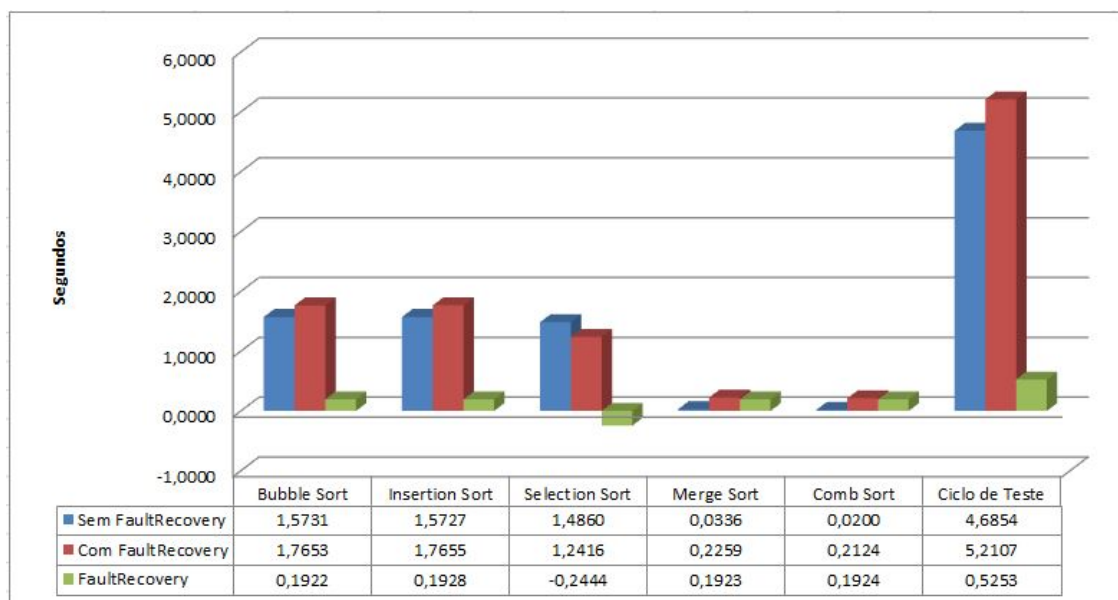


Figura 4.1: Nesta Figura são mostrados os resultado dos testes de medição do tempo de execução dos algoritmos de ordenação, sem a biblioteca e com a biblioteca. Percebe-se que com a utilização da biblioteca o tempo de execução dos algoritmos aumentou em média 0,5253 segundos ou 525,3 milisegundos.

Dentre os tempos de execução mostrados na Figura 4.1, obteve-se um resultado inesperado: o tempo de execução do algoritmo *insertion sort* foi menor com a biblioteca *FaultRecovery*, mesmo ela gerando um maior tempo de execução (devido à gerência da estrutura de sua máquina de estados). Diante disso, verificou-se a implementação dos dois testes, e não foi constatada nenhuma diferença entre os códigos que pudessem causar este resultado. Então, um terceiro teste foi executado outras cem vezes, utilizando-se um vetor de 6KB (menor que o utilizado nos testes anteriores) para verificar se o comportamento anômalo persistiria em ocorrer. O resultado foi similar, ou seja, o comportamento persistiu, mas como não havia tempo hábil para uma análise aprofundada, presumiu-se que esses casos são factíveis de ocorrer, provavelmente devido as otimizações de código feitas pelo compilador, já que o microcontrolador utilizado nos testes não possui sistema operacional e nenhuma outra aplicação rodando simultaneamente. Portanto verificou-se que quanto maior o volume de dados do vetor, o tempo de execução da *FaultRecovery* torna-se menos expressivo. Os resultados do terceiro teste são mostrados na Figura 4.2.

	Bubble Sort	Insertion Sort	Selection Sort	Merge Sort	Comb Sort
Com FaultRecovery(vetor 8KB)	0,1922	0,1928	-0,2444	0,1923	0,1924
Com FaultRecovery(vetor 6KB)	0,1966	0,1964	-0,0490	0,1970	0,1970
Com FaultRecovery(vetor 4KB)	0,2029	0,2023	0,0929	0,2020	0,2027

Figura 4.2: Nesta Figura é mostrado o tempo de execução da biblioteca *FaultRecovery* com três vetores de tamanhos diferentes. Percebe-se que o tempo de execução da biblioteca no algoritmo *insertion sort* é negativo no primeiro e no segundo teste, contudo no teste com o vetor de tamanho 4KB obteve-se um tempo de execução positivo. Portanto verificou-se que quanto maior o volume de dados do vetor, o tempo de execução da *FaultRecovery* torna-se menos expressivo.

4.2 Desempenho e Eficiência da Classe TData

Para testar a eficácia e o desempenho da classe *TData* foram realizados dois tipos de testes. O primeiro não utilizou a classe *TData* e o segundo a utilizou. Em ambos o vetor foi preenchido de forma decrescente e foram realizados cem ciclos de execuções, conforme descrito no primeiro parágrafo da Seção 4.1. No entanto houve a necessidade de reduzir o tamanho do vetor de 4096 para 1024 elementos, devido à redundância de dados da classe *TData*, pois ela copia um valor para três endereços de memória diferentes. Se fossem utilizados os 4096 elementos, a quantidade de memória alocada seria de aproximadamente 32kB, pois cada elemento que antes ocupava 2 bytes, passaria a utilizar 8 bytes. Isso impossibilitaria a realização dos testes.

O primeiro teste, que não utilizou a classe *TData*, teve como objetivo medir o tempo de execução de cada algoritmo de ordenação. Nele, a biblioteca *FaultRecovery* não foi utilizada, foi levado em conta apenas o tempo de execução de cada algoritmo. O resultado do primeiro teste foi comparado ao do segundo, obtendo-se o tempo de execução conforme mostrado na Figura 4.3. Para iniciar o segundo teste, foi necessário apenas substituir a declaração do vetor de *unsigned short vetor[n]* para *TData<unsigned short> vetor[n]*, isso mostra que a *TData* aplica redundância de dados de forma simples, reduzindo a reescrita de várias linhas de código.

Foram determinados três parâmetros de teste para deliberar se um ciclo de teste falhou ou não. Como o vetor possuía 1024 elementos ordenados em ordem decrescente de n até 1, o resultado de uma ordenação crescente é conhecido. Após a injeção de falhas no microcontrolador *mbed*, alterando um endereço de memória aleatório, estabeleceu-se que para um ciclo de teste falhar ele deveria ter entre 10%, 25% ou 50% dos 1024 elementos diferentes do resultado conhecido. Essa estratégia foi utilizada para se ter uma medida reduzida (10%), média (25%) e extrema (50%) de falhas.

Conforme exibido na Figura 4.4, o teste sem a *TData* para a quantidade de falhas acima de 10% foi de 20% para cada ciclo de testes. Com a *TData* esse número caiu para zero. Em contrapartida, notou-se uma diferença média de 0,2658 segundos a mais no tempo do ciclo com redundância de dados.

Para os resultados acima de 10%, 25% e 50% analisou-se os algoritmos de ordenação isoladamente e também cada ciclo de teste, lembrando que cada ciclo é representado pela execução dos cinco algoritmos de ordenação. Obteve-se os seguintes resultados, para os valores acima de 10%, 25% e 50% de falhas respectivamente, 44%, 25% e 20% dos ciclos de testes falharam. Percebe-se que o primeiro teste não possui redundância de dados, embora os cem ciclos de testes tenham sido executados, pode-

se notar que os valores do vetor não continuaram os mesmos após a injeção de falhas. No entanto na Figura 4.4 é possível visualizar que mesmo após as injeções de falhas, a classe *TData* se mostrou eficaz garantindo a consistência dos dados do vetor até o fim dos cem ciclos de teste.

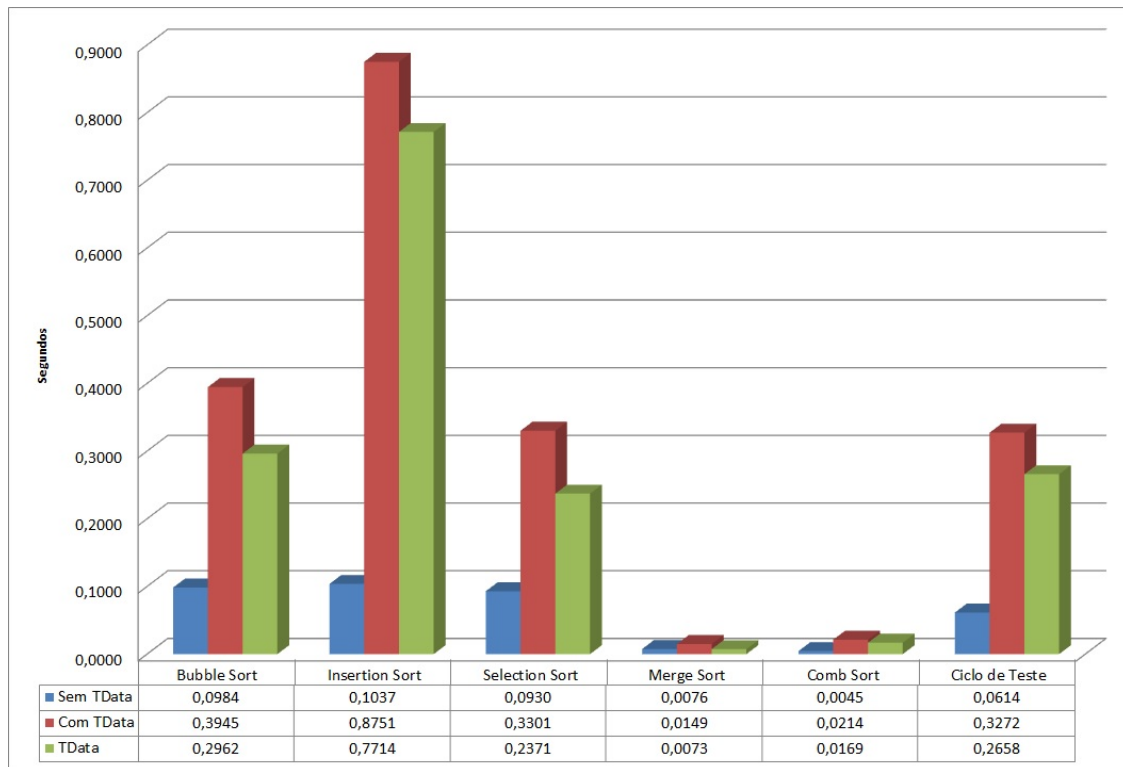


Figura 4.3: O tempo de execução dos algoritmos sem a classe *TData* foi relativamente baixo, sendo que o maior tempo atingiu 0,1037 segundos ou 103,7 milissegundos para ordenar um vetor de 1024 elementos. Já o tempo dos algoritmos com a Classe *TData* foi um pouco maior, sendo o tempo de execução do algoritmo insertion sort que atingiu 0,8751 segundos ou 875,1 milissegundos. O tempo de execução médio da classe *TData* para cada algoritmo de ordenação foi de 0,2658 segundos ou 265,8 milissegundos.

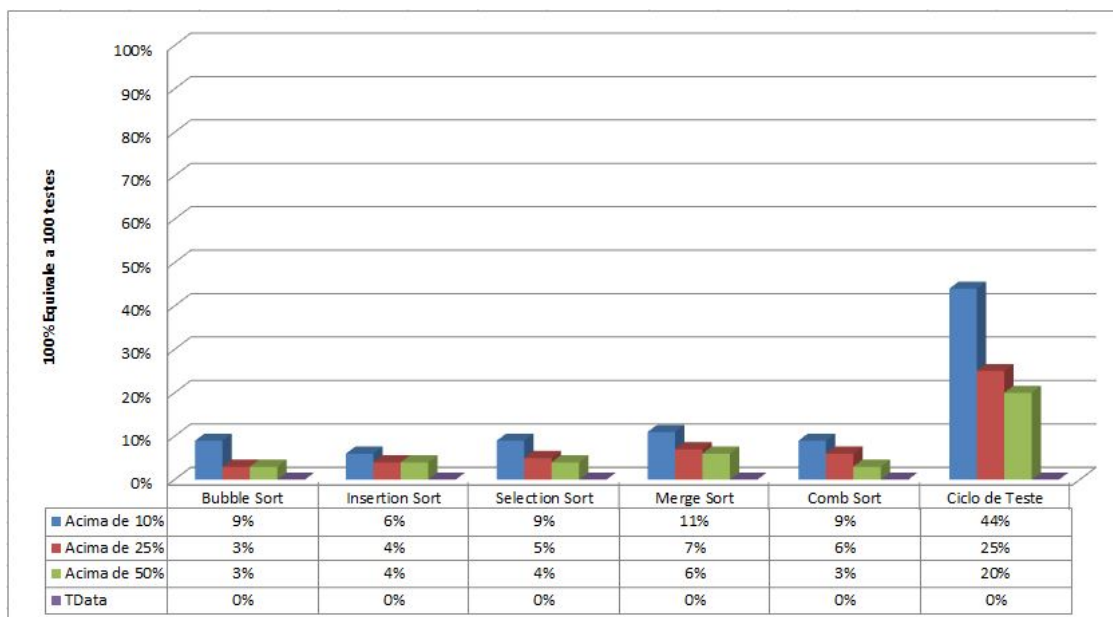


Figura 4.4: Nesta Figura, para as falhas detectadas acima de 10%, 25% e 50% são do teste que não utiliza redundância de dados, por exemplo, para as falhas acima de 10%, aproximadamente 44% dos ciclos de testes falharam. No entanto para o teste que utilizou a redundância de dados, nenhum algoritmo de ordenação ou ciclo de teste falharam. Embora o teste estivesse sendo bombardeado por falhas, a classe TData se mostrou eficaz corrigindo os valores alterados nos endereços de memória cobertos pela redundância de dados.

4.3 Recuperação de falhas da biblioteca *FaultRecovery*

Nesta Seção são mostrados os resultados dos testes realizados com a biblioteca *FaultRecovery*. O código utilizado na Seção 4.1 foi modificado para injetar falhas em endereços de memória aleatórios. Foram utilizados os mesmos algoritmos de ordenação, no entanto o tempo de execução da biblioteca foi desprezado, pois o resultado avaliado neste teste foi a capacidade de recuperação de falhas. Se as falhas registradas nos resultados ocorressem em uma situação real, os dados afetados por essas falhas poderiam ocasionar o travamento do *firmware*. Caso utilizado o mecanismo de *watchdog* para a detecção de falhas, quando o *watchdog* percebesse que o microcontrolador estivesse travado, o *mbed* seria reinicializado. No entanto, considerando-se o exemplo de uma estação meteorológica, se o travamento ocorresse no momento em que os dados coletados fossem enviados para um servidor remoto, por ser uma máquina de estados, no qual a ordem de execução de cada estado implica nos resultados obtidos, quando o *mbed* fosse reinicializado, o primeiro estado que seria executado, poderia ou não ser o estado responsável que enviaria os dados ao servidor remoto.

Para que isso não venha a ocorrer, a estação meteorológica poderia ser implementada utilizando a biblioteca *FaultRecovery* para que pontos de recuperação de falhas

pudessem ser criados, para assim que o microcontrolador reinicializasse por conta de alguma falha, algum ponto de recuperação predefinido pudesse ser executado. Entretanto utilizou-se algoritmos de ordenação para simular uma máquina de estados. Foram criados pontos de recuperação para cada algoritmo de ordenação, se em algum momento o microcontrolador vier a travar e posteriormente ser reiniciado, seja manualmente ou automaticamente, o ponto de recuperação predefinido será executado. Mostra-se na Figura 4.5 os resultados obtidos após a execução de cem ciclos de testes, cada ciclo é representado pela execução dos cinco algoritmos de ordenação conforme descrito na Seção 4.1.

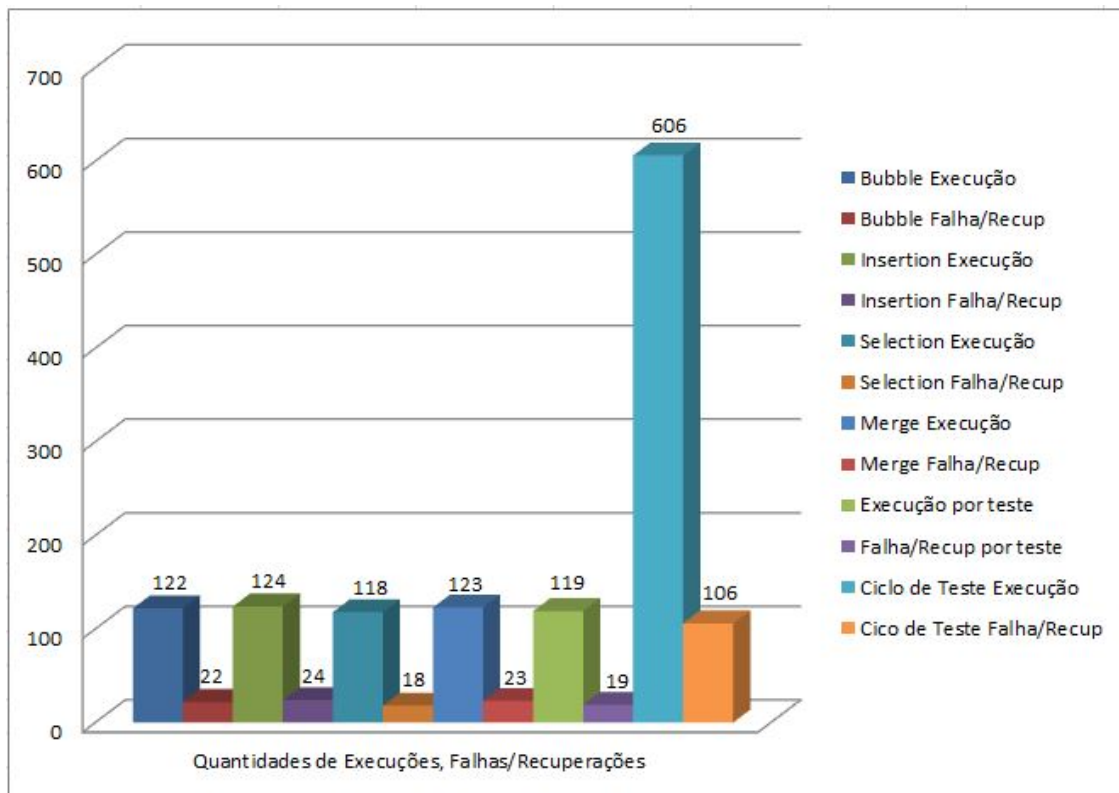


Figura 4.5: Nesta Figura é mostrada a quantidade de execuções e falhas de cada algoritmo, inclusive de cada ciclo de teste. Por exemplo, a primeira coluna, da esquerda para a direita, representa o número de execuções do algoritmo *bubble sort*, pode-se perceber que embora tenham sido executados cem ciclos de teste, o algoritmo foi executado cento e vinte e duas vezes. Em contra partida, na segunda coluna, que representa a quantidade de vezes em que um ponto de recuperação do *bubble sort* foi executado, o *firmware* reinicializou e executou o ponto de recuperação do *bubble sort* vinte e duas vezes.

4.4 Injeção de Falhas com a Biblioteca *FaultInjector*

Esta Seção mostra o resultado de injeção de falhas na memória *flash* do microcontrolador *mbed* LPC1768. A injeção de falhas na memória *flash* sorteou um setor aleatório para injetar uma quantidade predefinida de falhas, que podem variar de 256, 512, 1024 ou 4096 bytes. Na Figura 4.6 é exibido um teste em que foram injetadas 256 bytes de falhas no setor 25, que foi sorteado aleatoriamente pelo injetor de falhas. O endereço de memória do setor sorteado em hexadecimal inicia em 0x00058000 e termina em 0x0005FFFF. Pode-se perceber que os bytes dos endereços de memória do setor 25 (0x00058000 até 0x000580F0) foram alterados.

```

-----
Injetando falha no Setor
-----
25
-----
user reserved flash area: start_address=0x00058000, size=32768 bytes
-----
mempdump from 0x00058000 for 256 bytes
0x00058000 : 11111111 11111111
0x00058010 : 11111111 11111111
0x00058020 : 11111111 11111111
0x00058030 : 11111111 11111111
0x00058040 : 11111111 11111111
0x00058050 : 11111111 11111111
0x00058060 : 11111111 11111111
0x00058070 : 11111111 11111111
0x00058080 : 11111111 11111111
0x00058090 : 11111111 11111111
0x000580A0 : 11111111 11111111
0x000580B0 : 11111111 11111111
0x000580C0 : 11111111 11111111
0x000580D0 : 11111111 11111111
0x000580E0 : 11111111 11111111
0x000580F0 : 11111111 11111111
copied: SRAM(0x10007CC0)->Flash(0x00058000) for 256 bytes. memdump from 0x00058000 for 256 bytes
0x00058000 : 10011110 11111010
0x00058010 : 00001110 10100110
0x00058020 : 01010000 11010000
0x00058030 : 01100011 00000010
0x00058040 : 01111000 11100000
0x00058050 : 00000000 00001011
0x00058060 : 11010101 01101100
0x00058070 : 11010100 11000011
0x00058080 : 11010000 00100011
0x00058090 : 00000000 11010100
0x000580A0 : 11011000 00000000
0x000580B0 : 00000000 01011000
0x000580C0 : 00000001 11011110
0x000580D0 : 00110000 10110100
0x000580E0 : 00111111 00011101
0x000580F0 : 00000000 01000110

```

Figura 4.6: Nesta Figura é mostrado o setor da memória *flash* soteado pelo injetor de falhas e os endereços desse setor. Além de mostrar como os bits se encontravam antes de serem modificados, assim como também pode ser visualizada a modificação desses bits.

Capítulo 5

Conclusão

Neste trabalho a biblioteca *FaultRecovery* foi modificada para ser utilizada por usuários que queiram modularizar seu código, separando os estados de seu *firmware* por responsabilidades. A ideia e parte do código da biblioteca *FaultRecovery* está sendo utilizada pelo projeto de extensão Coxim Robótica sediado na UFMS - Campus Coxim. Os alunos estão programando um robô seguidor de linha, no qual são necessários alguns estados para que o carrinho desempenhe suas funções, como desviar de um obstáculo ou seguir em frente. Cada aluno do projeto é responsável pela implementação de um estado da máquina de estados do carrinho seguidor de linha.

Foram implementados testes com a biblioteca *FaultRecovery* e sem ela. Foi constatado que na implementação sem a biblioteca o *firmware* falhou e dessa forma não continuou a sequência de sua máquina de estados, pois não existia nenhum mecanismo de recuperação de falhas. No entanto nos testes realizados com a *FaultRecovery*, em todas as vezes que o *firmware* reiniciava, um ponto de recuperação era executado, mantendo a sequência original da máquina de estados. Existem pesquisas na área de semicondutores e na área de robótica que mostram os ruídos nos sensores como um problema comum que pode afetar a eficácia de algoritmos. Para contornar esse problema foi implementada neste trabalho a redundância de dados por meio da classe *TData*, que se mostrou eficaz em manter a integridade dos dados, protegendo as informações que por ventura venham a ser modificadas por falhas. A *TData* foi acrescentada à biblioteca *FaultRecovery*, sendo assim o usuário que utilizá-la poderá criar pontos de recuperação e separar o seu código, também poderá proteger dados importantes de seu *firmware*. Cabe observar que a biblioteca se mostrou eficaz na resolução do problema, pois em todos os testes o *firmware* tolerou 100% das falhas injetadas.

No entanto, a biblioteca adiciona um custo de desempenho no tempo de processamento, uma vez que toda a operação de leitura e escrita em uma variável, feita pela

classe *TData*, é custosa devido a execução de um esquema de votação usado para definir o valor correto permanente em todas as cópias. Porém, isso já era esperado, ainda sim o uso da biblioteca se torna mais vantajoso pelo fato de a maioria das aplicações embarcadas não terem como fator principal o tempo de execução, exceto algumas aplicações de tempo real. Mas nestes casos são utilizados microcontroladores com um maior poder de processamento.

A biblioteca *FaultInjector* também foi modificada neste trabalho, agora é possível injetar falhas na memória *flash*. No teste realizado, mostrou-se os bits armazenados em determinado setor da memória *flash* antes e depois da injeção de falhas. Com isso foi possível visualizar os bits sendo alterados. Além disso, também foi incluído um mapeamento de memória que possibilita a utilização do injetor de falhas em modelos pertencentes a família *mbed* LPC176X. Porém só foi possível testar o mapeamento de memória no modelo 1768, pois era o único microcontrolador disponível para este trabalho. O mapeamento funcionou para o modelo disponível e foi possível injetar falhas nos endereços de memória disponíveis no mapeamento. Tanto a biblioteca *FaultRecovery*, quanto a *FaultInjector* estão disponíveis no *github*, nos endereços <https://github.com/cleitonlmeida1/FaultRecovery> e <https://github.com/cleitonlmeida1/FaultInjector>

Os resultados apresentados neste trabalho se mostraram bons, devido ao bom funcionamento das bibliotecas e a eficácia em solucionar problemas ocasionados por falhas. Conforme Johnson [7], tolerância a falhas é a propriedade que permite a um sistema continuar funcionando adequadamente, mesmo que num nível reduzido, após a manifestação de falhas em alguns de seus componentes.

5.1 Contribuições deste Trabalho

Uma arquitetura de desenvolvimento de aplicações embarcadas por meio de uma máquina de estados. Essa arquitetura está sendo utilizada pelos alunos do projeto de extensão Coxim robótica sediado na UFMS - campus Coxim. Em uma breve conversa com os integrantes do projeto, que antes programavam em um mesmo computador, informaram que o desenvolvimento do programa está mais rápido pois agora eles podem trabalhar em mais de um estado ao mesmo tempo.

5.2 Dificuldades Encontradas

No início da implementação encontrou-se bastante dificuldade para se adaptar a uma linguagem de programação diferente de java. Embora elas sejam parecidas, a preparação do ambiente de desenvolvimento é totalmente diferente. Foram encontradas algumas dificuldades para configurar a IDE e o compilador c++ no windows 10. Além do tempo de estudo da linguagem que levou mais de 15 dias para adaptação. Por ser uma linguagem de ampla utilização, existem muitos fóruns de dúvidas que ajudaram durante o desenvolvimento.

5.3 Trabalhos Futuros

- Testar o mapeamento de memória incluído na biblioteca *FaultInjector* para outros modelos além do modelo *mbed* LPC1768.
- Salvar as cópias utilizadas pela classe *TData*, para se ter redundância de dados, em outras regiões de memória. Atualmente as cópias estão sendo salvas na região de memória reservada ao usuário, mas futuramente poderá ser salva na região destinada aos periféricos internos do microcontrolador.
- Aperfeiçoar a biblioteca *FaultRecovery* e a classe *TData* para diminuir o tempo de processamento em uma aplicação embarcada.
- Explorar a injeção de falhas na memória *flash*, implementando um *firmware* e injetando falhas enquanto ele está em execução.

Referências

- [1] NELSON, V. P. Fault-tolerant computing: Fundamental concepts. *Computer*, Los Alamitos, CA, USA, v. 23, p. 19–25, jul 1990.
- [2] PEREIRA, L. A.; CARVALHO, F. R.; BORTOLUCCI, T.; MORAES, M. H. Software embarcado, o crescimento e as novas tendências deste mercado. *Centro Universitário Anhanguera*, v. 6, p. 85–94, 2014.
- [3] KRUGER, K. *Programação de microcontroladores utilizando técnicas de tolerância a falhas*. 2014. Dissertação de Mestrado - UFMS, Campo Grande, 2014.
- [4] MALVINO, A. *Microcomputadores e microprocessadores*. McGraw-Hill, 1985.
- [5] PATTERSON, D. A.; HENNESSY, J. L. *Computer organization and design: The hardware/software interface*. 3rd. ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2007.
- [6] CHETAN, S.; RANGANATHAN, A.; CAMPBELL, R. Towards fault tolerance pervasive computing. *Technology and Society Magazine, IEEE*, v. 24, n. 1, p. 38–44, Spring 2005.
- [7] JOHNSON, B. Fault-tolerant microprocessor-based systems. *Micro, IEEE*, v. 4, n. 6, p. 6–21, dec 1984.
- [8] TADEU, T. G.; GALVÃO, L. E. M. Um estudo exploratório sobre sistemas operacionais embarcados. Technical Report 1, Instituto de Ciência e Tecnologia da Universidade Federal de São Paulo - UNIFESP, 2014.
- [9] REIS, E. *Previsão de doenças de plantas*. UPF EDITORA, 2004.
- [10] IAIONE, F.; LIMA, D. G.; GASSEN, F. R. Equipamento para coleta de dados e previsão de doenças na lavoura, 1999.
- [11] HSUEH, M.-C.; TSAI, T. K.; IYER, R. K. Fault injection techniques and tools. *Computer*, Los Alamitos, CA, USA, v. 30, n. 4, p. 75–82, apr 1997.

- [12] ZIEGLER, J. Terrestrial cosmic rays. *IBM Journal of Research and Development*, v. 40, n. 1, p. 19–39, Jan 1996.
- [13] PRESSMAN, R. *Software engineering: A practitioner's approach*. 6. ed. New York, NY, USA: McGraw-Hill, Inc., 2005.
- [14] TAURION, C. *Software embarcado: oportunidades e potencial de mercado*. Rio de Janeiro: Brasport, 2005.
- [15] CARVALHO, A. Grandes desafios da pesquisa em computação no brasil de 2006 - 2016. Technical report, SBC - Sociedade Brasileira de Computação, Porto Alegre: SBC, 2006.
- [16] WEBER, T. S. Um roteiro para exploração dos conceitos básicos de tolerância a falhas. Publicação online, 2002. Disponível em: <<http://www.inf.ufrgs.br/taisy/disciplinas/textos/Dependabilidade.pdf>> Acesso em: 15/11/2015.
- [17] VELAZCO, R.; FOUILLAT, P.; REIS, R. *Radiation effects on embedded systems*. Dordrecht: Springer, 2007.
- [18] LAPRIE, J. C.; ARLAT, J.; BEOUNES, C.; KANOUN, K. Definition and analysis of hardware- and software-fault-tolerant architectures. *Computer*, v. 23, n. 7, p. 39–51, July 1990.
- [19] STASSINOPOULOS, E.; RAYMOND, J. P. The space radiation environment for electronics. *Proceedings of the IEEE*, v. 76, n. 11, p. 1423–1442, Nov 1988.
- [20] BOUDENOT, J. C. *Radiation space environment*. Springer. p. 1–9.
- [21] Cinturão de van allen. Disponível em: <<http://geocities.ws/saladefisica5/leituras/vanallen.html>> Acesso em: 17/11/2015.
- [22] MANSOORI, A.; KHAN, P.; BHAWRE, P.; GWAL, A.; PUROHIT, P. Variability of tec at mid latitude with solar activity during the solar cycle 23 and 24. In: . c2013. p. 83–87.
- [23] ZIEGLER, J.; LANFORD, W. *The effect of cosmic rays on computer memories*. Science, 1979. v. 206.
- [24] MAY, T. C.; WOODS, M. H. A new physical mechanism for soft errors in dynamic memories. In: . c1978. p. 33–40.
- [25] YU, H.; FAN, X.; NICOLAIDIS, M. Design trends and challenges of logic soft errors in future nanotechnologies circuits reliability. In: . c2008. p. 651–654.

- [26] NORMAND, E. Single-event effects in avionics. *IEEE Transactions on Nuclear Science*, v. 43, n. 2, p. 461–474, Apr 1996.
- [27] ECOFFET, R.; DUZELLIER, S.; TASTET, P.; AICARDI, C.; LABRUNEE, M. Observation of heavy ion induced transients in linear circuits. In: . c1994. p. 72–77.
- [28] AVIZIENIS, A.; KELLY, J. Fault tolerance by design diversity: Concepts and experiments. *Computer*, v. 17, n. 8, p. 67–80, Aug 1984.
- [29] VON NEUMANN, J. Probabilistic logics and the synthesis of reliable organisms from unreliable components. *Automata Studies*, p. 43–98, 1956.
- [30] PEGO, M. O. *Tolerância a falhas através de escalonamento em um sistema multi-processado*. 2004. Tese de Doutorado - UFMG, Belo Horizonte, 2004.
- [31] BRILLIANT, S. S.; KNIGHT, J. C.; LEVESON, N. G. Analysis of faults in an n-version software experiment. *IEEE Trans. Softw. Eng.*, Piscataway, NJ, USA, v. 16, n. 2, p. 238–247, feb 1990.
- [32] CHEN, L.; AVIZIENIS, A. N-version programming: A fault-tolerance approach to reliability of software operation. In: . c1995. p. 113–.
- [33] PRADHAN, D. K. (Ed.). *Fault-tolerant computer system design*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1996.
- [34] SOMANI, A. K.; VAIDYA, N. H. Understanding fault tolerance and reliability. *Computer*, Los Alamitos, CA, USA, v. 30, n. 4, p. 45–50, apr 1997.
- [35] KANAWATI, G.; KANAWATI, N.; ABRAHAM, J. Ferrari: a flexible software-based fault and error injection system. *Computers, IEEE Transactions on*, v. 44, n. 2, p. 248–260, Feb 1995.
- [36] ARLAT, J.; CROUZET, Y.; KARLSSON, J.; FOLKESSON, P.; FUCHS, E.; LEBER, G. Comparison of physical and software-implemented fault injection techniques. *Computers, IEEE Transactions on*, v. 52, n. 9, p. 1115–1133, Sept 2003.
- [37] ARLAT, J.; AGUERA, M.; AMAT, L.; CROUZET, Y.; FABRE, J.-C.; LAPRIE, J.-C.; MARTINS, E.; POWELL, D. Fault injection for dependability validation: A methodology and some applications. *IEEE Transactions on Software Engineering*, Los Alamitos, CA, USA, v. 16, n. 2, p. 166–182, 1990.
- [38] GUNNEFLO, U.; KARLSSON, J.; TORIN, J. Evaluation of error detection schemes using fault injection by heavy-ion radiation. In: . c1989. p. 340–347.

- [39] SOTOMA, I. *Afids - arquitetura para injeção de falhas em sistemas distribuídos*. 1997. Dissertação de Mestrado - UFRGS, Porto Alegre, 1997.
- [40] MARTINS, E.; ARLAT, J.; CROUZET, Y.; FABRE, J.; POWELL, D. Testing multipeer protocols in the presence of faults. *Protocol Test Systems*, p. 77–91, 1989.
- [41] DAWSON, S.; JAHANIAN, F.; MITTON, T. A software fault injection tool on real-time mach. In: . c1995. p. 130–140.
- [42] ROSENBERG, H. A.; SHIN, K. G. Software fault injection and its application in distributed systems. In: . c1993. p. 208–217.
- [43] SEGALL, Z.; VRSALOVIC, D.; SIEWIOREK, D.; YASKIN, D.; KOWNACKI, J.; BARTON, J.; DANCEY, R.; ROBINSON, A.; LIN, T. Fiat-fault injection based automated testing environment. In: . c1988. p. 102–107.
- [44] SENGER, VINICIUS, X. K. 33 design-patterns aplicados com java. Publicação online, 2009. Disponível em: <<http://pt.slideshare.net/vsenger/33-design-patterns-com-java>> Acesso em: 14-Abril-2016.
- [45] ENGHOLM JR, H. *Engenharia de software na prática*. Novatec Editora Ltda, 2010.
- [46] MBED, A. mbed lpc1768, 2016. Disponível em: <<https://developer.mbed.org/platforms/mbed-LPC1768/>> Acesso em: 23/03/2016.
- [47] MBED, A. Compilador, 2016. Disponível em: <<https://developer.mbed.org/cookbook/WatchDog-Timer>> Acesso em: 29/08/2016.
- [48] MBED, A. Compilador, 2016. Disponível em: <<https://developer.mbed.org/>> Acesso em: 23/03/2016.
- [49] MBED, A. Lpc1768/66/65/64, 2009. Disponível em: <<http://www.nxp.com/>> Acesso em: 23/03/2016.
- [50] MBED, A. lap internal flash write, 2010. Disponível em: <<https://developer.mbed.org/users/okano/code/>> Acesso em: 23/03/2016.
- [51] MEYERS, S. *C++ eficaz 55 maneiras de aprimorar seus programas e projetos*. Porto Alegre: BOOKMAN COMPANHIA EDITORA, 2011.
- [52] LEITHARDT, V. R. Q. *Modelo gerenciador de descoberta de serviços pervasivos ciente de contexto*. 2008. Dissertação de Mestrado - PUCRS, Porto Alegre, 2008.
- [53] HERNANDE, D. N. Estrutura de dados, 2011. Disponível em: <<http://www.ft.unicamp.br/liag/siteEd/>> Acesso em: 03/08/2016.

- [54] STAVISK, S. Ordenar vetor com algoritmo inserton sort, 2010. Disponível em: <<https://www.vivaolinux.com.br/script/Ordenar-vetor-com-algoritmo-Insertion-Sort/>> Acesso em: 03/08/2016.

Apêndice A

Anexos

TEXTO ANEXO.