

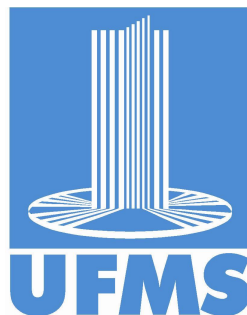
Trabalho de Conclusão de Curso

FaultRecovery: A amplificação da
biblioteca de tolerância a falhas.

Cleiton Gonçalves de Almeida

Orientação: Prof. Me. Kleber Kruger

Bacharelado em Sistemas de Informação



Sistema de Informação
Universidade Federal de Mato Grosso do Sul
10 de Maio de 2016

FaultRecovery: A ampliação da biblioteca de tolerância a falhas

Coxim, 10 de Maio de 2016.

Banca Examinadora:

- Prof. Me. Kleber Kruger (CPCX/UFMS) - Orientador

Resumo

Atualmente, o ser humano utiliza diversos aparelhos eletrônicos, tais como celulares, toca-dores 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 que abrangem uma grande quantidade dos sistemas computacionais estão cada vez mais presentes no cotidiano das pessoas. No entanto esses sistemas podem apresentar defeitos, que indicam 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 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.

Os objetivos deste trabalho foram estudar possíveis causas de falhas em sistemas embarcados, modificar as bibliotecas *FaultInjector* e *FaultRecovery*. Inclusive criar uma classe de redundância de dados no qual sua função visa garantir a integridade dos dados de um sistema embarcado. Uma das modificações visa possibilitar ao usuário desenvolver uma máquina de estados, no 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*.

Ao final, são apresentados os resultados mostrando o tempo de execução após as modificações realizadas na biblioteca *FaultRecovery*, para verificar se essas alterações impactaram no desempenho do código testado. O teste realizado com a *FaultRecovery* foi executado em 5,0883 segundos, enquanto que em média o teste sem a biblioteca foi executado em 4,4171 segundos, ou seja, a biblioteca elevou o tempo de execução do teste realizado em 0,6712 segundos. No entanto a biblioteca foi exposta a testes de recuperação de falhas, mostrando-se eficaz em todos os testes. A classe também foi exposta a testes de desempenho e redundância de dados, nos resultados que não utilizaram a TData o tempo de execução médio foi de 0,0614 segundos, enquanto que nos testes com a TData o tempo médio foi de 0,3272 segundos, a classe TData elevou o tempo de execução do teste em 0,2658 segundos. No entanto 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 carrinho seguidor de linha e continuará sendo utilizada em programas futuros. Também foi criada uma classe que possibilita a utilização de redundância de dados em um sistema embarcado, que foi inserida a 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. Thanks to expansion of ubiquitous computing, embedded systems that cover a lot of computer systems are increasingly present in daily life. However these systems can malfunction, indicating a system's inability to perform a certain task because of errors 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 study was to investigate possible causes of failures in embedded systems, modify the FaultInjector eFaultRecovery libraries. Including creating 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 would not give the user a ready development framework, it has now been modified to meet a design pattern called State.

At the end, the results are presented showing the execution time after the changes made in FaultRecovery library to see if the changes have impacted the library performance. The test with FaultRecovery was run at 5.0883, while on average the test without the library was run in 4.41 seconds, which is the library increased the test runtime performed in 0.67 seconds. However the library was exposed to disaster recovery testing, proving to be effective in all tests. The class was also exposed to the performance tests and data redundancy, which results in not used the TData the average run time was 0.0614 seconds, while the tests with TData the average time was 0.3272 seconds TData the 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 library FaultRecovery modification idea is being 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, which will be inserted in FaultRecovery library that will allow the user to set up your embedded system to recover from faults and can also protect your important data was also created.

Agradecimientos

Conteúdo

Lista de Figuras	10
Lista de Tabelas	11
Lista de Quadros	12
1 Introdução	13
1.1 Justificativa	14
1.2 Objetivos	14
1.2.1 Objetivo Geral	14
1.2.2 Objetivos Específicos	15
1.3 Organização da Proposta	16
2 Fundamentação Teórica	17
2.1 Falha, Erro e Defeito	17
2.2 Principais Fontes de Radiação e seus Efeitos nos Circuitos Eletrônicos	18
2.2.1 Cinturão de Van Allen	19
2.2.2 Atividade Solar	19
2.2.3 Raios Cósmicos	20
2.2.4 Partículas alpha	20
2.2.5 Efeitos Singulares ou <i>Single Event Effects</i> (SEE)	20
2.3 Dependabilidade	21
2.4 Tolerância a Falhas	21
2.5 Técnicas de Tolerância a Falhas	23
2.5.1 Técnicas de redundância baseadas em software	23
2.5.2 Diversidade ou Programação N-Versões	23

2.5.3	Blocos de Recuperação	24
2.5.4	Verificação de Consistência	25
2.6	Injeção de Falhas	25
2.6.1	Injeção de falhas por <i>Hardware</i>	25
2.6.2	Injeção de falhas por <i>Software</i>	26
2.7	<i>Design Patterns</i>	26
2.7.1	Padrão GoF (Padrões Fundamentais Originais)	27
3	Metodologia	28
3.1	Injetor de Falhas	29
3.1.1	Mapeamento de Memória	29
3.1.2	Injeção de Falhas na Memória Flash	33
3.2	FaultRecovery: Extensão da biblioteca	34
3.2.1	Refatoração e Aperfeiçoamento: Versão 1.0	34
3.2.2	Refatoração e Aperfeiçoamento: Versão 2.0	37
3.3	<i>Classe de Redundância de Dados: TData</i>	38
3.3.1	Classe TData com Tipos Primitivos	39
3.3.2	Um Exemplo Ilusório Para Utilização da Classe TData com Objetos	40
3.3.3	Classe Carro com Objeto de Pilha	42
3.3.4	Classe Carro com Ponteiro	44
4	Resultados	49
4.1	Desempenho da Biblioteca <i>FaultRecovery</i>	49
4.2	Desempenho e Eficiência da Classe TData	50
4.3	Recuperação de falhas da biblioteca <i>FaultRecovery</i>	52
4.4	Injeção de Falhas com a Biblioteca <i>FaultInjector</i>	53
5	Conclusão	55
5.1	Contribuições deste Trabalho	56
5.2	Dificuldades Encontradas	56
5.3	Trabalhos Futuros	56
	Referências Bibliográficas	58

Apêndices	61
A Anexos	62

Lista de Figuras

2.1	Modelo de três universos	18
2.2	Cinturão de Van Allen	19
2.3	Características das Falhas	22
2.4	Programação N-Versões	24
2.5	Blocos de Recuperação	24
3.1	Mapas das Regiões de Memória dos Modelos LPC1768/66/65/64	30
3.2	Figura que apresenta a saída com os valores das cópias consistentes após a injeção de falhas do Quadro 9.	40
3.3	Figura que apresenta a saída com os valores das cópias consistentes após a injeção de falhas do Quadro 13.	44
3.4	Figura que apresenta a saída com os valores das cópias consistentes após a injeção de falhas do Quadro 16.	46
3.5	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 do Quadro 19.	48
4.1	Tempo de execução da biblioteca <i>FaultRecovery</i>	50
4.2	Tempo de execução da Classe TData	51
4.3	Teste de redundância de dados da classe TData	52
4.4	Resultados Obtidos da Biblioteca <i>FaultRecovery</i>	53
4.5	Injeção de Falhas na Memória Flash	54

Lista de Tabelas

2.1	Resumo dos Atributos de Dependabilidade	21
-----	---	----

Lista de Quadros

1	Super Classe MemoryMap.	31
2	Mapa da Região de memória destinada aos Periféricos	32
3	Uso de macro como função	34
4	Macro sendo chamada no código	34
5	Exemplo de criação de um estado	36
6	Métodos <i>createRecoveryPoints</i> e <i>run</i>	38
7	Classe que automatiza a redundância de dados.	39
8	Exemplo de utilização da classe TData para variáveis de tipo primitivo. . . .	40
9	Método utilizado para injeção de falhas em uma das cópias da classe TData.	40
10	Classe Emergencia.	41
11	[Classe Carro sem ponteiro.	42
12	Método que demonstra integridade dos dados de localização do veículo após a injeção de falhas.	43
13	Método utilizado para injetar falhas em uma das cópias da classe TData . .	44
14	Classe Carro com ponteiro.	44
15	Método que implementada a utilização de um objeto TData do tipo Carro com um ponteiro para um enredado de memória em seu escopo.	45
16	Método utilizado para simular uma falha na localização do carro.	46
17	Objetos que representam a redundância de dados da classe TData	46
18	Método responsável pela redundância de dados.	47
19	Teste realizado verificar a integridade dos dados de localização do carro . . .	47

Capítulo 1

Introdução

Atualmente, o ser humano utiliza diversos aparelhos eletrônicos, tais como celulares, toca-dores 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). Estes, 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 deve saber ajustar a potência correta, selecionar e medir o tempo em que o forno deve ficar acionado e emitir um sinal quando a tarefa for concluída.

Por ser uma classe de computadores que executa tarefas de propósito específico, os sistemas embarcados possuem requisitos exclusivos que geralmente combinam bom desempenho com rigorosas limitações de custo e consumo de energia [3]. Aplicações embarcadas de propósito específico consomem menos bateria do que aplicações de propósito geral, em razão disso os sistemas embarcados utilizam microcontroladores ao invés de microprocessadores. Segundo 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, pois consequências de falhas podem causar danos que variam de transtorno 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 móvel *web*. Os índices são utilizados pelos agricultores para definir o momento certo para a aplicação dos defensivos [8, 9]. Se o sistema de coleta de dados climáticos falhar, o agricultor não saberá o momento correto para aplicação dos defensivos, podendo a plantação ser infectada e destruída pela doença.

Fenômenos da natureza (exploração adaptativa), interferência eletromagnética, desgaste dos componentes de hardware [10] são grandes causadores de falhas em semicondutores, principalmente quando estes dispositivos não possuem blindagem contra ruídos ou pulsos transitórios causados por prótons, íons pesadores 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 [11].

Tolerar falhas é um requisito de um sistema embarcado [12]. Considerando-se um caixa eletrônico com falhas na contagem do dinheiro quando os clientes realizam um saque, ou um sistema de falhas de controle aéreo com falhas no cálculo de suas rotas, os problemas seriam gravíssimos [5]. Por estes motivos aumentar a confiabilidade de um sistema, seja aplicando técnicas de tolerância a falhas de hardware ou software, é imprescindível [1].

1.2 Objetivos

1.2.1 Objetivo Geral

A injeção de falhas é um processo importante para validar e verificar a confiabilidade de um sistema [13]. Foram objetivos deste trabalho, ampliar o injetor de falhas (*FaultInjector*) e a biblioteca de recuperação de falhas (*FaultRecovery*) desenvolvidos por Kruger [3] em sua dissertação de mestrado. Para isso serão reutilizados o *hardware* e o *firmware* utilizados por Kruger. O injetor de falhas não injetava falhas na memória flash e também seu mapeamento de memória era específico para o microcontrolador *mbed* modelo 1768, a inserção de falhas na memória flash e a ampliação para vários modelos da plataforma *mbed* também foram objetivos deste trabalho.

A biblioteca *FaultRecovery* permitia a recuperação de falhas e a implementação de uma máquina de estados. Porém o usuário deveria criar sua própria estrutura para iniciar o desenvolvimento do seu *firmware*. Outro objetivo é a modificação da *FaultRecovery*, que visa entregar ao usuário uma estrutura de desenvolvimento pronta, na qual ele deverá apenas estender algumas classes, sendo possível criar estados independentes, configurar o seu microcontrolador e criar pontos de recuperação de falhas. A redundância de dados em um sistema

embarcado possibilita manter a integridade dos dados, para isso a criação da classe *TData* tem como objetivo realizar um esquema de votação a cada vez que uma variável do tipo *TData* é lida, possibilitando manter a integridade do valor armazenado.

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.
- Incluir a classe *TData* a ampliação da biblioteca, que proporcionará uma maior confiabilidade ao sistema que utilizar a *FaultRecovery*.
- Realizar testes de performance e eficiência após as modificações da biblioteca *FaultRecovery*.
- Realizar testes de performance 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

Neste capítulo serão 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.

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 através de 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 será apresentado 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.

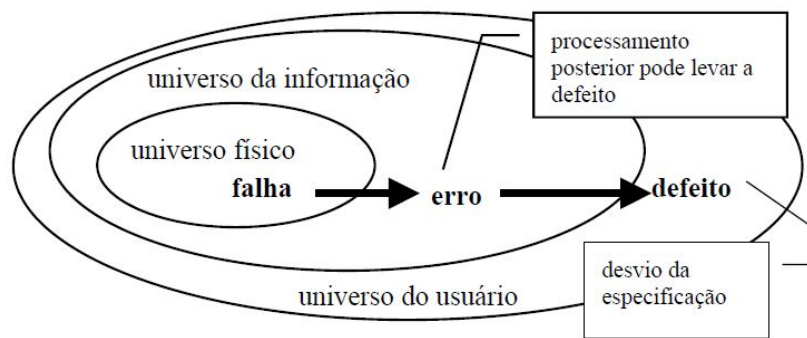


Figura 2.1: Modelo de 3 universos: falha, erro e defeito. Retirado de Weber [14].

Exemplo de uma falha no universo físico é um chip de memória com uma falha do tipo grudado-em-zero (*stack-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 [15]. 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 [16].

A radiação está presente tanto no espaço quanto na atmosfera, podendo alterar a resposta ou danificar componentes eletrônicos expostos a íons pesados (partículas 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 [17] e a atividade solar [18]. 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.

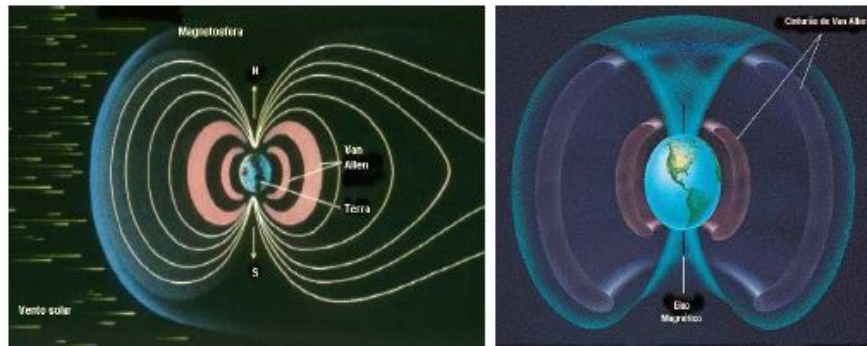


Figura 2.2: Cinturão de Van Allen [19].

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 [17]. Um terceiro cinturão de elétrons foi observado após uma tempestade magnética em 24 de março de 1991 [15].

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 [20]. O período de 11 anos correspondente ao ciclo solar está dividido em aproximadamente 7 anos de alta atividade e 4 anos de baixa atividade [18].

Durante a baixa atividade solar o sol emite rajadas de partículas energéticas no espaço. Estas 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 [17].

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 [15].

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 [17] 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 [18], 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. Estes 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 [21].

2.2.4 Partículas alfa

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 [22]. Estas partículas estão presentes nos materiais utilizados para o encapsulamento de circuitos integrados, onde 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 [15].

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 com a mudança de estados ou erros transientes num dispositivo causado pela indução de partículas energéticas ou radiação cósmica [23]. 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, neste trabalho será citado apenas SEU e SET.

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

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 [14].

armazenamento no flip-flop, ou seja, um bit-flip.

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 [24].

2.3 Dependabilidade

Segundo Laprie, 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*.

Segundo Laprie e Weber [14] 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. Os principais atributos de dependabilidade são confiabilidade, diponibilidade, segurança de funcionamento (*safety*), segurança (*security*), manutenibilidade, testabilidade e comprometimento do desempenho (*performability*). Um resumo dos principais atributos é mostrado na Tabela 2.1.

2.4 Tolerância a Falhas

Segundo Avizienis [25] 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 [26]. 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 [14].

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 mostra-se

Pêgo [27] descreve os conjuntos de atributos utilizados para caracterizar uma falha:

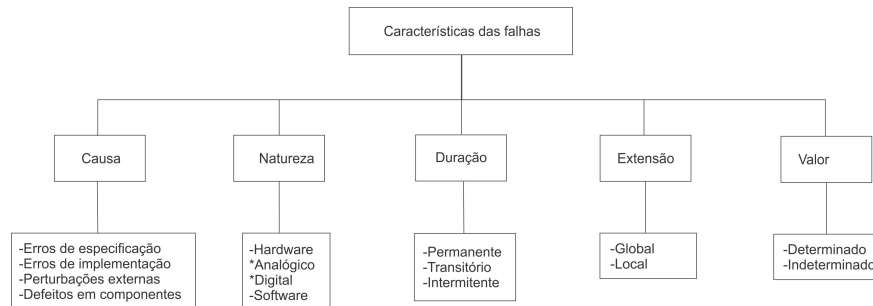


Figura 2.3: Atributos das características das falhas. Retirado de Pêgo [27].

- **Causa** - Uma falha pode ter origem em problemas de especificação, problemas de execução, defeitos em componentes do sistema operacional (que não são incomuns para dispositivos eletrônicos), ou em fatores externos, tais como, tempestades, 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 amplificadores, 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, logo em seguida, desaparece. Esse tipo de falha é geralmente provocada por causas externas. Relâmpago, por exemplo, podem provocar um erro súbito em uma linha de transmissão, mas após o relâmpago, a linha voltará ao seu funcionamento normal. Finalmente, há falhas, que são chamadas intermitentes; estas ocorrem em períodos curtos de tempo e desaparecem, mas depois elas voltam novamente. É possível que este processo se repete indefinidamente. Uma falha intermitente é a ocorrência de repetição de falhas transitórias. Este último tipo pode ser causado por especificações de projeto que permite um certo componente de trabalhar em certas condições críticas, tais como a temperatura, por exemplo.
- **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 uma avaria determinada. 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, no qual envolvem alguma forma de redundância. Existem técnicas baseadas em software e hardware, neste trabalho será citado apenas as técnicas baseadas em software, pois no desenvolvimento da aplicação proposta não serão utilizadas as técnicas baseadas em hardware, pois esta técnica demanda da utilização de um equipamento que faz bombardeamento de íons sobre o circuito eletrônico podendo danificar o hardware [14].

2.5.1 Técnicas de redundância baseadas em software

Segundo Pêgo [27] 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 [27].
- **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 *basic blocks*, construção de grafos e a checagem em tempo de execução sobre a correta transição entre os vértices deste grafo [27].

Weber [14] 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 diferente. Erros de programas idênticos vão apresentar, com grande probabilidade, de forma idêntica para os mesmos dados de entrada. Segundo Brilliant *et al.* 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 [28] A. *apud* Fischler et. al., os esforços de programação são realizados por N indivíduos. Aonde 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 [28].

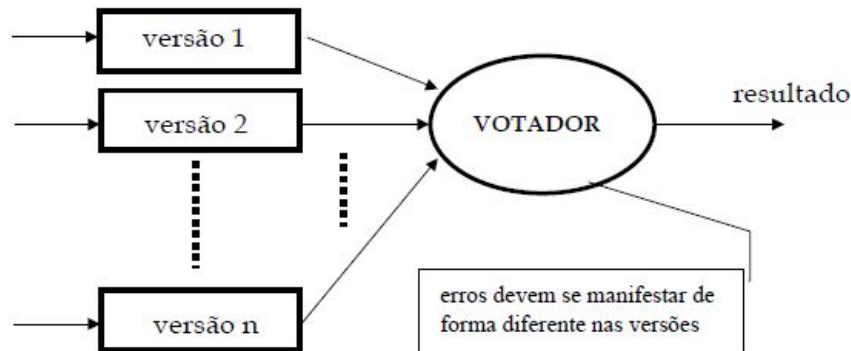


Figura 2.4: Diversidade ou Programação N-Versões. Retirado de Weber [14].

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 [29].

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, 14, 30].

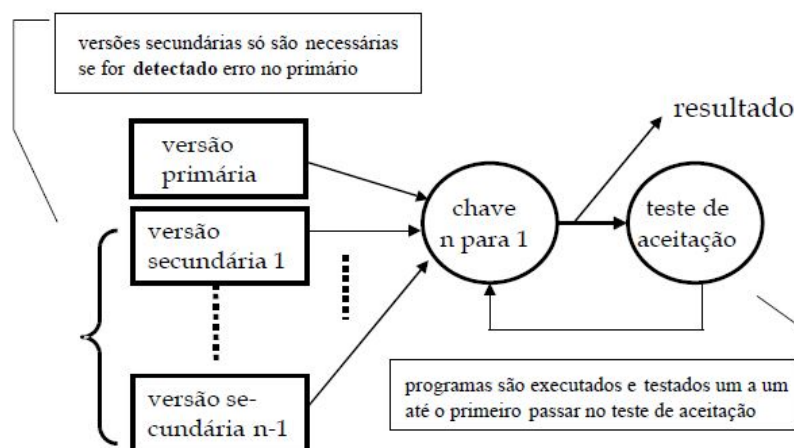


Figura 2.5: Blocos de Recuperação. Retirado de Weber [14].

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 corretude da informação. Os *softwares* das equipes são executados paralelamente e as saídas são submetidas à 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* [13] ou a nível de pinos (*Pin-level Injection*) injetando falhas diretamente no hardware [31]. Além da injeção de falhas por *software* e por *hardware* existe um terceiro tipo chamado injeção de falhas por Simulação. Arlat [31] 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.* [32] 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, no qual os mecanismos de manipulação de falhas são avaliados em sua eficiência na estimativa de medidas como cobertura e latência [32].

O auxílio ao projeto ocorre durante a fase de desenvolvimento do projeto buscando melhorar a eficiência dos mecanismos de teste e o protocolo de tolerância a falhas por meio da injeção de falhas [32].

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 realista do sistema [31].

2.6.1 Injeção de falhas por *Hardware*

Esta técnica necessita de um *hardware* especial, no qual as falhas seriam originadas. Arlat [31] 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 in-

jetor de falhas [32, 31]. Outras técnicas de injeção de falhas por *hardware* são interferências eletromagnéticas e irradiação de íons pesados [33, 32, 31], estas por sua vez podem danificar o componente sob teste [34]. O objetivo dessas técnicas visa principalmente estudar o comportamento dos mecanismos de tolerância a falhas implementados por *hardware*, porém também pode-se testar os mecanismos de falhas por meio de *softwares* que não danificam os componentes sob teste [35].

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 ao à capacidade de alterar o estado de registradores e memória [13]. Um exemplo de injeção de falhas por *software* é a ferramenta *Ferrari* apresentada por Kanawati *et. al.* [13] que permite 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 [13], PFI [36], SFI [37], FIAT [38].

2.7 *Design Patterns*

Neste trabalho identificou-se a necessidade da utilização de um padrão de projeto ou em inglês design pattern. A ideia básica de padrões de projeto é a de utilizar soluções conhecidas para problemas conhecidos. Problemas são recorrentes tanto no meio acadêmico quanto no mercado de trabalho, principalmente no desenvolvimento de software, este está sujeito a atrasos por conta desses problemas. No entanto cada padrão de projeto descreve aqueles problemas e suas soluções, permitindo que esta possa ser utilizada sem ter a necessidade de descobri-la sozinho, economizando tempo, gastos e ter a plena confiança de que ela está sendo testada e empregada tanto no meio acadêmico quanto no mercado de trabalho (referenciar).Existem inúmeros padrões de projetos para resolução de vários problemas, o conceito de máquina de estados será utilizado neste trabalho [39]para o desenvolvimento do framework Fault Recovery que será descrito na seção ?? . Design Patterns possuem alguns elementos que facilitam no desenvolvimento da sua aplicação, 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)

Os padrões GoF são divididos em três grupos: padrões de comportamento, de criação e estruturais. Estes descrevem como os objetos são colocados juntos, esses fornecem maneiras robustas de se criar objetos e aqueles descrevem como os objetos interagem, distribuindo responsabilidades [39]. Neste trabalho foi-se utilizado o padrão *State*, indicado em programas que podem ser representados por uma máquina de estados, que é o caso de todos os *firmwares*. 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. Sabemos que todo objeto possui atributos, que representam seu estado, e também métodos, que representam seu comportamento [40]. Baseado neste conceito o *framework FaultRecovery* será desenvolvido nos próximos meses.

Capítulo 3

Metodologia

Neste trabalho foi utilizado um microcontrolador de prototipagem rápida *mbed* modelo NXP LPC1768 [41] para testar a ampliação da biblioteca *faultRecovery*, que permite a programação de sistemas embarcados por meio de máquinas de estados, a injeção de falhas na memória *flash* da biblioteca *faultInjector* e a classe TData que é responsável pela redundância de dados. A arquitetura da biblioteca também pode ser utilizada por sistemas de outras plataformas, por exemplo, os sistemas embarcados implementados para *arduino*.

Existem outros modelos pertencentes a família *mbed* NXP LPC17X, são eles: LPC1764, LPC1765, LPC1766 e LPC1768 [42]. O módulo NXP LPC1768 [41] é um microcontrolador projetado para prototipagem de diversos dispositivos, especialmente aqueles que necessitam de internet, USB, e a flexibilidade de lotes de interfaces de periféricos e memória *flash*. Ele possui um núcleo ARM Cortex-M3 32bits rodando a 96 MHz, inclui 512 KB de memória *flash*, 32 KB de memória RAM, incluindo *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 microcontrolador possui um temporizador *watchdog* que reinicia o dispositivo, que é incrementado enquanto o microcontrolador estiver em funcionamento, no entanto quando o dispositivo trava, o temporizador deixa de ser incrementado, com isso é possível utilizar o temporizador para reiniciar o dispositivo em caso de travamento. Para novos desenvolvedores de sistemas embarcados para microcontroladores de 32 bits, a *mbed* fornece em seu *site* uma solução de prototipagem acessível a todos que estiverem cadastrados para obter projetos com o apoio de bibliotecas compartilhadas pela comunidade de desenvolvedores. Além das bibliotecas, o *site* disponibiliza um fórum para retirada de dúvidas. A *mbed* também fornece um compilador online, no qual após a compilação do código, um arquivo binário é gerado para ser executado no microcontrolador, contudo o compilador online tem a capacidade de compilar códigos para diversos modelos do *mbed* [43].

O Compilador online foi utilizado neste trabalho para criar a arquitetura de desenvolvimento deste trabalho. Com o primeiro contato com o microcontrolador *mbed* identificou-se a necessidade de estudar suas funcionalidades. Com a utilização do compilador online foi possível verificar que seria inviável adotá-lo como um editor de código para a expansão do injetor de falhas e da biblioteca *faultRecovery*. A solução encontrada foi a utilização do am-

biente de desenvolvimento integrado (IDE) LPCXpresso, pois o compilador online permite exportar a arquitetura criada para a IDE. Essa é baseada na IDE eclipse e possui versões gratuitas e pagas, neste trabalho utilizou-se a versão gratuita para o modelo NXP LPC1768.

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 de injeção de falhas chamada *FaultInjector* foi desenvolvida para a dissertação de mestrado de Kruger [3]. Kruger afirma que sua biblioteca injeta falhas em diversas regiões da memória do microcontrolador *mbed* LPC1768 e também menciona que ela não injeta falhas em uma região específica, que é a região de memória *flash*. Na seção que versa sobre trabalhos futuros, Kruger descreveu que teria a intenção de injetar falhas para essa região de memória.

Entre as propostas deste trabalho está a implementação da funcionalidade de injeção de falhas na memória *flash*, ampliando a biblioteca de injeção de falhas de Kruger [3] e o mapeamento de memória para todas as versões *mbed* para os modelos LPC176x. Por meio de uma pesquisa foi possível encontrar uma biblioteca que torna possível escrever na memória *flash* em um curto período de tempo, poupando horas de desenvolvimento que seriam utilizadas na criação dessa biblioteca, esta pertence a Ocano, um desenvolvedor participante da comunidade *mbed* [44]. Foram realizados alguns testes de escrita na memória *flash*, até se chegar a um resultado em que todos os endereços de memória de um setor da memória *flash* fosse modificado.

Quando os estudos da biblioteca *FaultInjector* foram iniciados identificou-se que as injeções de falhas somente poderiam ser realizadas em um único modelo de microcontrolador *mbed* (LPC1768), pois para este trabalho foi disponibilizado apenas dois microcontroladores do mesmo modelo. Para melhorar e aperfeiçoar a biblioteca fez-se uma modificação no código para permitir que a própria biblioteca identificasse o modelo do microcontrolador, permitindo assim que a injeção de falhas fosse feita em todos os modelos *mbed* LPC176X.

3.1.1 Mapeamento de Memória

Como mencionado na seção 3.1 a biblioteca *FaultInjector* de Kruger [3] injetava falhas somente em algumas regiões de memória, excluindo a região interna, sendo que o mapeamento dessas regiões eram específicas para o microcontrolador *mbed* LPC1768. Foi proposto pelo orientador deste trabalho que a biblioteca identificasse o modelo do microcontrolador da família LPC176X e disponibilizasse o mapeamento de todas as regiões de memória daquele modelo identificado. O primeiro passo para mapear as regiões de memória de cada modelo foi estudar o mapa de memória no manual da família *mbed* LPC176X [42], o estudo do mapa resultou no mapeamento demonstrado na figura 3.1, na qual podemos visualizar que alguns modelos possuem regiões de memória em comum, tornando possível o reaproveitamento de código. O conceito de herança foi utilizado com o intuito de padronizar o mapeamento das regiões de memória de cada modelo do microcontrolador, além de reduzir a quantidade de

código, por exemplo, todos os modelos LPC176X utilizam a mesma região de memória para os periféricos, esta pode ser observada na figura 3.1.

Com as regiões de memória mapeadas, o segundo passo foi a implementação desse mapeamento. Foi criada uma superclasse chamada *MemoryMap* demonstrada no Quadro 1 com três métodos, *getUserMemoryRegion()*, *getFlashMemoryRegion()*, *getPeripheralsMemoryRegion()*, sendo obrigatória a implementação de dois deles para as classes filhas, são eles: *getUserMemoryRegion()*, *getFlashMemoryRegion()*.

Outra classe chamada *MemoryMapPeripherals* demonstrada no Quadro 2 também foi criada, no entanto não com o objetivo de ser uma superclasse, mas sim, uma classe que detém o mapeamento da região de memória dos periféricos do microcontrolador, comum a todos os modelos. Esta classe possui apenas um método chamado *getPeripheralsMemoryRegion()* que será utilizado pela classe *MemoryMap* para retornar um *ArrayList* com as regiões de memória destinadas aos periféricos.

E por último outra classe chamada *MemoryRegion* [3], que será de fato a região de memória, contendo os atributos responsáveis por guardar o endereço de memória inicial, final e o seu tamanho.

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.

```
1 class MemoryMap {
2 public:
3 MemoryMap() {
4 for (unsigned int i = 0; i < map.getPeripheralsMemoryRegion().size();
5 ++i) {
6 peripheralsMemoryRegion.add(
7 map.getPeripheralsMemoryRegion().getForIndex(i));
8 }
9 }
10
11 virtual ArrayList<MemoryRegion*> getUserMemoryRegion() = 0;
12 virtual ArrayList<MemoryRegion*> getFlashMemoryRegion() = 0;
13 virtual ArrayList<MemoryRegion*> getPeripheralsMemoryRegion() {
14 return peripheralsMemoryRegion;
15 }
16
17 virtual ~MemoryMap() {
18
19 }
20
21 private:
22 MemoryMapPeripherals map;
23 ArrayList<MemoryRegion*> peripheralsMemoryRegion;
24 };
```

Quadro 1: Super Classe MemoryMap.

- ***getUserMemoryRegion()*** - retorna um *ArrayList* de regiões de memória disponíveis para o usuário, estas podem ser visualizadas na figura 3.1.
- ***getFlashMemoryRegion()*** - retorna um *ArrayList* de regiões da memória interna, estas podem ser visualizadas na figura 3.1.
- ***getPeripheralsMemoryRegion()*** - retorna um *ArrayList* de regiões da memória destinadas aos periféricos, estas podem ser visualizadas na figura 3.1.

```

1 class MemoryMapPeripherals{
2     public:
3     MemoryMapPeripherals() :
4     papb0(0x40000000, 0x40080000 - 1, MemoryRegion::PERIPHERALS,
5     "PeripheralsAPB0"),
6     papb1(0x40080000, 0x40100000 - 1,
7     MemoryRegion::PERIPHERALS, "PeripheralsAPB1"),
8     pbba(0x42000000, 0x44000000 - 1, MemoryRegion::PERIPHERALS,
9     "PeripheralsBitBandAlias"),
10    pahb(0x50000000, 0x50200000 - 1, MemoryRegion::PERIPHERALS,
11    "AHBPeripherals"),
12    ppb(0xE0000000, 0xE0100000 - 1, MemoryRegion::PERIPHERALS, "
    PrivatePeripheralsBus") {
13
14        memoryMap.add(&papb0);
15        memoryMap.add(&papb1);
16        memoryMap.add(&pbba);
17        memoryMap.add(&pahb);
18        memoryMap.add(&ppb);
19    }
20
21    ArrayList<MemoryRegion*> getPeripheralsMemoryRegion() {
22        return memoryMap;
23    }
24
25    virtual ~MemoryMapPeripherals(){
26
27    }
28
29    private:
30    MemoryRegion papb0;
31    MemoryRegion papb1;
32    MemoryRegion pbba;
33    MemoryRegion pahb;
34    MemoryRegion ppb;
35    ArrayList<MemoryRegion*> memoryMap;
36
37 };

```

Quadro 2: Classe que representa o mapeamento da região de memória destinada aos periféricos e poderá ser utilizada por todos os microcontroladores da família LPC176X.

3.1.2 Injeção de Falhas na Memória Flash

O LPC1768 possui 512KB de memória flash disponível para programação, no qual o código do *firmware* implementado pelo desenvolvedor mbed será armazenado (referenciar UM10360). Sabendo que o código do sistema embarcado será armazenado na memória flash, percebe-se que a alteração de um único bit em um endereço de memória que contenha os códigos de instrução do sistema poderá resultar em uma falha irreversível para esta instrução. No entanto os dados coletados por um sistema também podem ser armazenados na memória flash, por exemplo, um sistema embarcado de uma estação meteorológica precisa coletar os dados sobre umidade do ar, o índice pluviométrico, entre outras informações e armazená-los na memória flash. Estes dados armazenados na flash serão enviados a um servidor remoto após um período de tempo predefinido. Se um único bit da região de memória no 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 do seu valor após sua alteração, ou seja, o dado coletado que será enviado para o servidor remoto será falso, impactando em uma equivocada previsão do tempo.

Um dos objetivos deste trabalho é a ampliação da biblioteca de injeção de falhas que anteriormente injetava falhas apenas na memória reservada ao usuário que poderá ser visualizada na seção 3.1. Para simular o fenômeno *bit-flip* a injeção de falhas será realizada por software pois a injeção de falhas por hardware pode danificar o microcontrolador mbed, conforme explicado na seção 2.6 e para este trabalho disponibiliza-se de apenas dois microcontroladores, sendo um utilizado como estação meteorológica. A memória flash é dividida em 29 setores que são divididos em duas áreas, sendo a primeira que vai do setor 1 até o 15 e a capacidade de armazenamento de cada setor é de 4KB e a segunda vai do setor 16 até o 29 e a capacidade de armazenamento é de 32KB para cada setor (referenciar UM10360).

A injeção de falhas na flash foi realizada por meio da escrita de bytes em um setor de memória sorteado aleatoriamente copiando 256 bytes da memória RAM para a flash utilizando uma biblioteca que permite escrever na flash. Esta foi retirada do repositório de bibliotecas do site www.mbed.com, sendo possível escrever uma quantidade fixa de bytes que podem ser 256, 512, 1024 ou 4096 [44].

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 simulado a escrita de 256 bytes na memória flash, ou seja, foram injetadas 256 bytes de falha em setores aleatórios que poderiam, no momento da injeção,

estar armazenando os dados coletados pelo sistema embarcado ou as instruções dele, ou seja, o valor das informações coletadas foi alterado podendo causar uma falha irreversível no sistema afetando o seu correto funcionamento.

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 identificou-se a necessidade de ampliar a biblioteca com o emprego do padrão de projeto *State*, pois basicamente todos os *firmwares* desenvolvidos para a plataforma mbed são implementados como máquinas de estados.

As macros eram utilizadas demasiadamente, por isso algumas foram retiradas e outras substituídas por *templates*, pois além de dificultarem o entendimento da estrutura do código, existem outros males que advêm de seu uso, principalmente pela utilização de macros como funções. Por exemplo, uma macro que chama uma função *f* com o maior dos argumentos da macro:

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

Quadro 3: Macro que chama *f* como o máximo entre *a* e *b*

Veja as coisas estranhas que podem acontecer ao se utilizar a macro do Quadro 3 demonstrado no Quadro 4, aqui o número de vezes que *a* é incrementado antes de chamar *f* depende do que está sendo comparado a ele.

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

Quadro 4: Na primeira invocação da macro a variável *a* é incrementada duas vezes e na segunda uma vez.

Utilizou-se o exemplo anterior escrito por Meyers [45] para ilustrar uma das desvantagens de se implementar macros na expansão da biblioteca *faultRecovery*.

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 os *firmwares* da plataforma *mbed* sejam máquinas de estados, a maioria deles não são orientados a objetos e tão pouco legíveis.

Com a expansão da biblioteca será possível escrever o código da aplicação com orientação a objetos, em outras palavras, o código do *firmware* poderá ser separado por responsabilidades. Cada estado terá a sua classe de implementação permitindo a separação do código e

facilitando o seu entendimento. A primeira versão da biblioteca possui três classes, são elas: *TypeIDGenerator*, *State* e *StateMachine*, que são descritas abaixo:

- ***TypeIDGenerator*** - Esta classe é responsável pela criação do id de identificação de cada estado. Se um id for gerado para um determinado estado, esse será recuperado pela classe, ou seja, apenas um id será gerado para cada estado.
- ***State*** - Esta classe será responsável pela representação de cada estado. Um determinado estado deverá herdar a classe *State*, sendo que o usuário será obrigado a implementar o método *run*, este recebe como parâmetro uma máquina de estados, que é representada pela classe *StateMachine*. No método *run* o usuário implementará as rotinas de execução daquele estado e ao final de cada rotina, ele deverá chamar o método *setToState* e passar entre os sinais (< >) a classe que representa o próximo estado a ser executado, conforme demonstrado no Quadro 5.
- ***StateMachine*** - Esta classe representa a máquina de estados responsável por adicionar os estados automaticamente. Esses são armazenados em um *map*, semelhante ao *HashMap* da linguagem Java, por ser mais eficiente em termos de performance que uma lista. Por exemplo, no momento em que o usuário está finalizando uma rotina de um estado, ele deverá chamar o método *setToState*.. Neste momento, quando a máquina de estados for executada, o método *setToState* guardará o id do estado corrente na 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.

A classe *StateMachine* também possui os métodos *start* e *run*. 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 5, o método *start* chamará o método *setToState*, que atribuirá o id do estado inicial para a variável *current* e chamará o método *run* que executará a máquina de estados por meio de um *loop*.

```

1 class EsperandoCarroChegar: public State {
2     void run(StateMachine &sm) {
3         char c;
4         cout << "[C] - Carro chegou" << endl;
5         cin >> c;
6         cout << "" << endl;
7         switch (c) {
8             case 'C':
9             case 'c':
10                sm.setToState<EsperandoApertarBotao>();
11        }
12    }
13 };
14
15 int main() {
16     StateMachine sm;
17     sm.start<EsperandoCarroChegar>();
18 }

```

Quadro 5: 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 serão anexados ao trabalho.

Pensou-se em um exemplo simples para ilustrar a utilização da extensão da biblioteca. Destaca-se, um caso ilusório de um carro ao entrar em um 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 5:

- ***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 isto, 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 este 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 tratará da criação dos pontos de recuperação, como inicializar a máquina de estados e quais as classes adicionas a 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 travar no momento em que a máquina de estados esteja em execução, se algum ponto de recuperação estiver configurado, quando o microcontrolador for reinicializado pelo *whatchdog* o ponto de recuperação será executado. O usuário da biblioteca poderá criar esses pontos de recuperação, se ele identificar essa necessidade, caso algum ponto de recuperação seja inserido no código, ele deverá indicar quais as rotinas deverão ser executadas, que podem ser uma determinada configuração do microcontrolador ou uma tarefa que deva ser iniciada antes da máquina de estados ser executada.

As classes *Application* e *RecoveryPoint* foram adicionadas a 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, os métodos *creatRecoveryPoints* e *setup* são opcionais. 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 carrinho seguidor de linha.

O segundo é o método *createRecoveryPoints*, que poderá ou não ser implementado. Nesse método os pontos de recuperação são adicionados a máquina de estados. Mas para esses pontos serem incluídos na máquina de estados, antes eles devem ser implementados. 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 *run* no qual conterá as rotinas de execução, se houver necessidade, da máquina de estados antes dela ser executada. Para adicionar um ponto de recuperação a máquina de estados, deve-se utilizar o método *addRecoveryPoint* nativo da classe *StateMachine* e evocá-lo conforme demonstrado no quadro Quadro 6.

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<RecoveryEsperandoCarroEntrar, EsperandoCarroEntrar>();
4 }
5
6 //RecoveryEsperandoCarroEntrar Class
7 void RecoveryEsperandoCarroEntrar::run(StateMachine &sm) {
8 bool PORTAO_ABERTO = false;
9 if (!PORTAO_ABERTO) {
10 abrirPortao();
11 }
12 sm.start<EsperandoCarroEntrar>();
13 }

```

Quadro 6: 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. Pois se a energia acabar neste momento, quando o firmware for executado novamente, o ponto de recuperação verificará se o portão está fechado, caso sim, o portão será aberto e a máquina de estados será iniciada novamente a partir do estado EsperandoCarroEntrar, caso não, o portão continuará aberto e máquina de estados será executada novamente.

3.3 Classe de Redundância de Dados: TData

A classe *TData* foi implementada com o objetivo de se obter 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 tipo do dado redundante e passar um valor como parâmetro no construtor: `TData<tipo_da_variável> variavel(valor)`. O valor ou o objeto passado como parâmetro serão copiados para três cópias de segurança, para então se conseguir a redundância de dados e manter a integridade do valor original. Esta declaração funciona tanto em variáveis homogêneas (tipos primitivos) quanto em heterogêneas (objetos). Ao instanciar o objeto *TData* o valor passado como parâmetro será copiado para três variáveis diferentes por meio do método *setData*, no Quadro 7, o primeiro para objetos de pilha e o segundo para ponteiros.

```
1  template<typename T>
2  void TData<T>::setData(T data) {
3  d1 = data;
4  d2 = data;
5  d3 = data;
6  dataObject = data;
7  }
8  template<typename T>
9  void TData<T>::setData(T *data) {
10 d1 = *data;
11 d2 = *data;
12 d3 = *data;
13 dataObject = *data;
14 }
```

Quadro 7: Classe que automatiza a redundância de dados

3.3.1 Classe *TData* com Tipos Primitivos

Pode-se verificar no Quadro 8 um exemplo de utilização da classe *TData* com uma variável de tipo primitivo. A forma de instanciar um objeto *TData* foi demonstrada na seção ??, mas também pode ser vista no Quadro 8. O método *setData* 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 chamado automaticamente por meio de sobrescrita de operadores, disponível na linguagem C++, permitindo que o valor da variável seja alterado conforme o código de exemplo demonstrado Quadro 8 e a saída desse código pode ser visualizada na figura 3.2. Vale ressaltar que a cada vez que o objeto *TData* for acessado, o método *getByVotting* que implementa um esquema de votação será executado e validará todas as cópias do objeto *TData* mantendo a consistência delas, este método pode ser visualizado no Quadro 7.

A injeção de falhas foi realizada com um método temporário chamado *injectFault* no qual uma das cópias sofreu uma modificação em seu valor simulando o fenômeno *bit-flip*, este método pode ser visto no Quadro 9.

```

1  int main() {
2
3  TData<int> variavel(4);
4  cout << "Valor inicial de variavel" << endl;
5  cout << "variavel: " << variavel << endl;
6
7  cout <<"Alterando o valor da variável de 4 para 10" << endl;
8  variavel = 1 + 9;
9  cout << "variavel: " << variavel << endl;
10
11 //injetando falha
12 variavel.injectFault();
13 cout <<"Valor de variavel após a injeção de falha" << endl;
14 cout << "variavel: " << variavel << endl;
15
16 //imprime os dados armazenados nas 3 cópias
17 cout <<"Valores armazenados nas 3 cópias após a injeção de falha" << endl;
18 variavel.print();
19
20 }

```

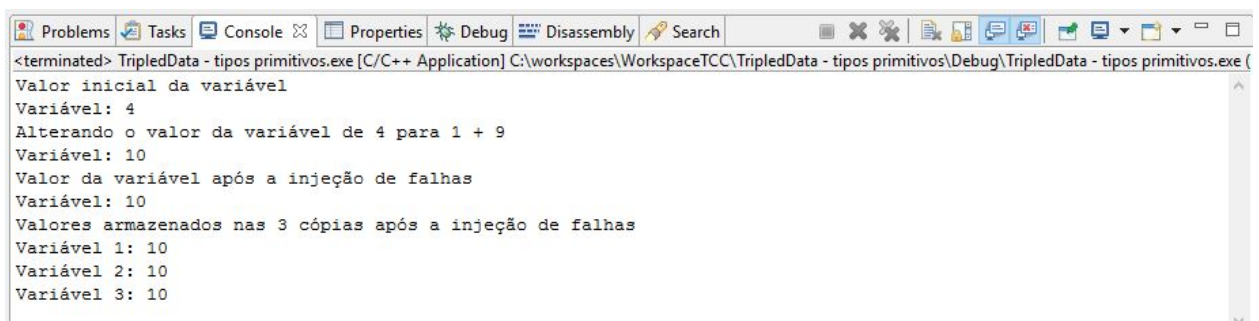
Quadro 8: Exemplo de utilização da classe TData para variáveis do tipo primitivo.

```

1  template<class T>
2  void TData<T>::injectFault() {d1 = 3000;}

```

Quadro 9: Este método pode ser implementado dentro da classe TData para simular um bit-flip.



```

<terminated> TripleData - tipos primitivos.exe [C/C++ Application] C:\workspaces\WorkspaceTCC\TripleData - tipos primitivos\Debug\TripleData - 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 3.2: Figura que apresenta a saída com os valores das cópias consistentes após a injeção de falhas do Quadro 9.

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 (referenciar). 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. Essas podem ser visualizadas no Quadro 11 e no Quadro 10 respectivamente. Mas por que os operadores de igualdade precisam ser implementados? Porque o método de votação *getByVotting* realiza algumas comparações entre as cópias da classe *TData*, cada cópia é um objeto do tipo especificado pelo usuário (*TData<tipo_especificado_pelo_usuario>*), esse método é demonstrado no Quadro 7.

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 (Quadro 11) que contém uma instância da classe Emergência (Quadro 10) em seu escopo, tornando obrigatória a implementação de seus operadores de igualdade.

```
1 class Emergencia {
2 public:
3   Emergencia() {telefoneHospital = 0;}
4   virtual ~Emergencia() {}
5
6   bool operator==(Emergencia e) {
7     if (telefoneHospital == e.getTelefoneHospital()) {return true;}return false
8     ;}
9
10  bool operator!=(Emergencia e) {if (telefoneHospital != e.getTelefoneHospital
11    ()) {return true;}return false;}
12
13  unsigned int getTelefoneHospital() {return telefoneHospital;}
14
15  void setTelefoneHospital(unsigned int telefoneHospital) {this->
16    telefoneHospital = telefoneHospital;}
17 private:
18  unsigned int telefoneHospital;
19 };
```

Quadro 10: Classe de exemplo que contém as informações dos contatos de emergência.

```
1 class Carro {
2 public:
3
4 Carro() {localizacao = 0;}
5
6 virtual ~Carro() {}
7
8 bool operator==(Carro c) {if (localizacao == c.getLocalizacao() && emergencia
    == c.getEmergencia()) {
9 return true;}return false;}
10
11 bool operator!=(Carro c) {if (localizacao != c.getLocalizacao() && emergencia
    != c.getEmergencia()) {
12 return true;}return false;}
13
14 void setLocalizacao(unsigned int localizacao) {
15 this->localizacao = localizacao;
16 }
17
18 unsigned int getLocalizacao() {return this->localizacao;}
19
20 Emergencia& getEmergencia() {return emergencia;}
21
22 void setEmergencia(Emergencia& emergencia) {this->emergencia = emergencia;}
23
24 private:
25 unsigned int localizacao;
26 Emergencia emergencia;
27 };
```

Quadro 11: Neste quadro é mostrado um exemplo da classe Carro sem a utilização de ponteiros.

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 conforme demonstrado no Quadro 12 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 dos quadros anteriores. No Quadro 12 o método *injectFault* é invocado para simular o fenômeno *bit-flip*, ou seja, se apenas um bit for alterado no endereço de memória no qual está localizada a variável localizacao demonstrada no Quadro 11, o valor da localização do carro seria alterado. 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 (Quadro 10) e da

localização do carro pertencente a classe Carro (Quadro 11), essa simulação de falha pode ser visualizada no Quadro 13.

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 3.3 demonstrando que mesmo após a ocorrência de falhas nos endereços de memória da localização e do telefone do hospital, as cópias continuaram consistentes.

```
1  #include "TData.h"
2  #include "Carro.h"
3  using namespace std;
4  int main() {
5      //objeto
6      Carro carro;
7      carro.setLocalizacao(100);
8      carro.getEmergencia().setTelefoneHospital(12345);
9
10     //objeto TData recebendo o objeto carro como parâmetro
11     TData<Carro> dataCarro(carro);
12
13     //injeção de falhas
14     dataCarro.injectFault();
15
16     cout << "Localização original: " << dataCarro.getData().getLocalizacao()
17     << endl;
18     cout << "Telefone do Hospital original: "
19     << dataCarro.getData().getEmergencia().getTelefoneHospital() << endl;
20
21     //imprime os dados armazenados nas 3 cópias
22     cout << "Valores armazenados nas 3 cópias após a injeção de falhas" <<
23     endl;
24     dataCarro.print();
25 }
```

Quadro 12: Neste quadro é mostrado o método main(), que após ser executado, mostra os dados de localização do veículo íntegros após a injeção de falhas.

```

1  template<class T>
2  void TData<T>::injectFault() {d1.getEmergencia().setTelefoneHospital
    (9876543);d1.setLocalizacao(200);}

```

Quadro 13: . Método utilizado para injetar falhas em uma das cópias da classe TData. Este método pode ser implementado dentro da classe TData para simular um bit-flip.

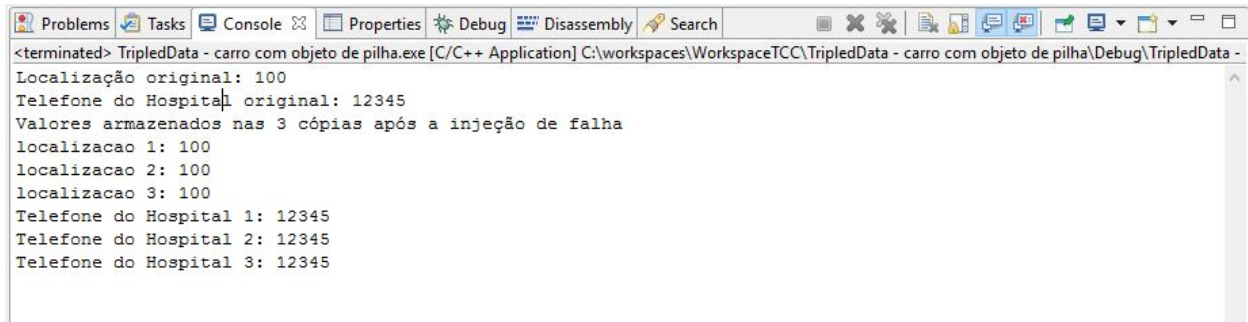


Figura 3.3: Figura que apresenta a saída com os valores das cópias consistentes após a injeção de falhas do Quadro 13.

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 alteração na variável emergencia (Quadro 11 linha 52) para que ela se tornasse um endereço de memória que pode ser visualizado na linha 52 do Quadro 14. Na linha 15 do Quadro 15 o método *injectFault* está simulando uma injeção de falhas no endereço de memória do ponteiro emergencia. Após a injeção de falhas, a saída do código demonstrado no Quadro 15 é mostrado na figura 3.4, mostrando que mesmo após a modificação do endereço de memória do ponteiro emergencia em uma das cópias, o valor do telefone do hospital continua o mesmo para todas elas.

```

1  class Carro {
2  public:
3
4  Carro() {localizacao = 0; emergencia = new Emergencia();}
5
6  virtual ~Carro() {}
7
8  bool operator==(Carro c) {if (localizacao == c.getLocalizacao() &&
    emergencia == c.getEmergencia()) {return true;}return false;}
9
10 bool operator!=(Carro c) {if (localizacao != c.getLocalizacao() &&
    emergencia!= c.getEmergencia()) {return true;}return false;}
11

```

```
12 void setLocalizacao(unsigned int localizacao) {this->localizacao =
    localizacao;}
13
14 unsigned int getLocalizacao() {return this->localizacao;}
15
16 Emergencia *getEmergencia() {return emergencia;}
17
18 void setEmergencia(Emergencia *emergencia) {this->emergencia = emergencia;}
19
20 private:
21 unsigned int localizacao;
22 Emergencia* emergencia;
23 };
```

Quadro 14: Classe de exemplo que contém os dados de localização e as informações dos contatos de emergência. Esta classe foi modificada com a intenção de testar sua eficiência com diferentes tipos de objetos, o atributo emergencia foi alterado para um ponteiro.

```
1 #include "TData.h"
2 #include "Carro.h"
3
4 using namespace std;
5
6 int main() {
7     Carro carro;
8     carro.setLocalizacao(100);
9     carro.getEmergencia()->setTelefoneHospital(123456);
10
11     //objeto TData recebendo o objeto carro como parâmetro
12     TData<Carro> dataCarro(carro);
13
14     //injeção de falhas
15     dataCarro.injectFault();
16
17     cout << "Localização original: " << dataCarro.getData().getLocalizacao()
18     << endl;
19     cout << "Telefone do Hospital original: "
20     << dataCarro.getData().getEmergencia()->getTelefoneHospital() << "\n"
21     << endl;
22
23     //imprime os dados armazenados nas 3 cópias
24     cout << "Valores armazenados nas 3 cópias após a injeção de falha" <<
    endl;
25     dataCarro.print();
26
27 }
```

Quadro 15: Método que implementada a utilização de um objeto TData do tipo Carro com um ponteiro para um enredoço de memória em seu escopo e injeta falhas para verificar a integridade dos dados de localização do veículo.

```

1  template<class T>
2  void TData<T>::injectFault() {Emergencia* e = new Emergencia(); d1.
    setEmergencia(e); d1.setLocalizacao(200);}
3  }

```

Quadro 16: Método utilizado para injeção de falhas. Este método pode ser implementado dentro da classe TData para simular um bit-flip.

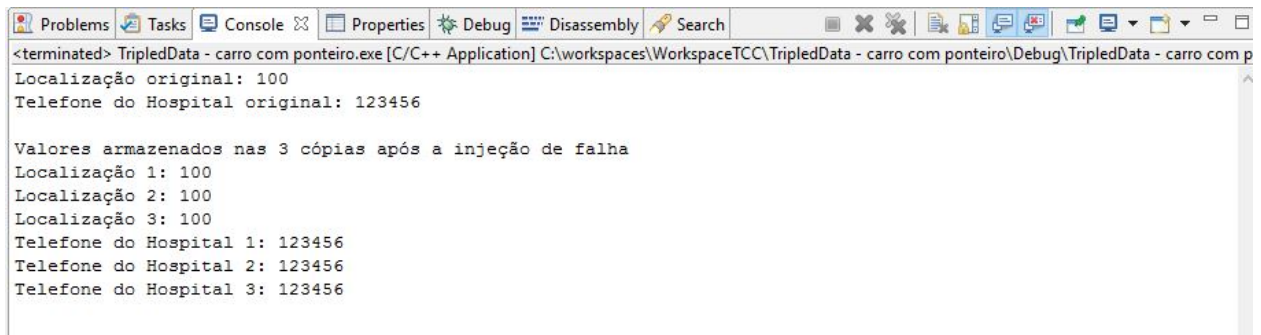


Figura 3.4: Figura que apresenta a saída com os valores das cópias consistentes após a injeção de falhas do Quadro 16.

Observa-se que no exemplo do Quadro 15 o objeto carro foi instanciado, teve os seus atributos preenchidos, para então ser passado como parâmetro no objeto *TData* e ser replicado para todas as cópias dessa classe. Mas além das três cópias de segurança que compõem a classe *TData* há mais uma cópia chamada *dataObject* (Quadro 17), que é utilizada para atualizar o objeto *TData* sem a necessidade de modificar o objeto carro e passá-lo como parâmetro para o método *setData*, ou seja, pode-se acessar uma das cópias, modificá-la e replicar esta alteração para as demais cópias conforme demonstrado no Quadro 18. 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. Um exemplo de utilização do método *getDataObject* pode ser visualizado no Quadro 19. A saída do código desse quadro é demonstrada na figura (tal).

```

1  private:
2
3  T d1;
4  T d2;
5  T d3;
6  T dataObject;

```

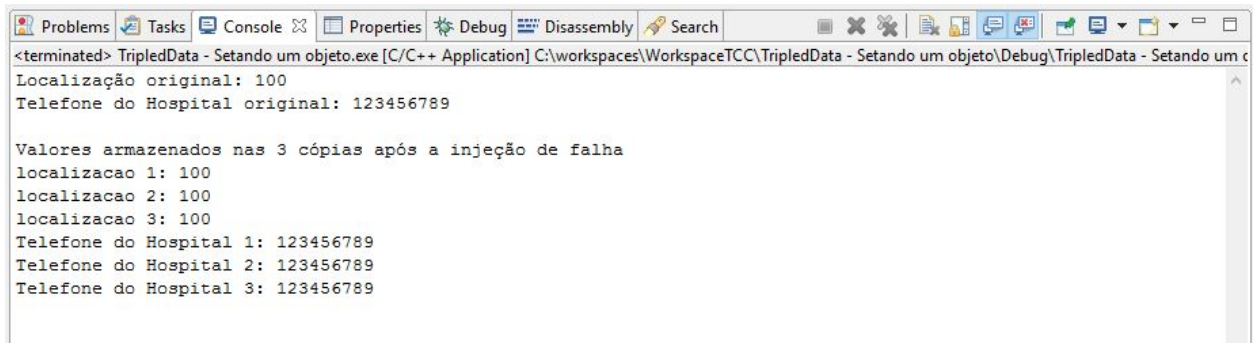
Quadro 17: Os objetos d1, d2 e d3 são utilizados para a redundância de dados e o objeto *dataObject* é utilizado para atualizar os valores de todas as cópias.

```
1  template<class T>
2  T* TData<T>::getDataObject() {
3      d1 = dataObject;
4      d2 = dataObject;
5      d3 = dataObject;
6      dataObject = getByVotting();
7      return &dataObject;
8  }
```

Quadro 18: Método responsável pela atualização de todas cópias da classe *TData*.

```
1  int main() {
2      Carro carro;
3      //carro.setLocalizacao(100);
4      //carro.getEmergencia()->setTelefoneHospital(123456);
5
6      //objeto TData recebendo o objeto carro como parâmetro
7      TData<Carro> dataCarro(carro);
8
9      //o objeto dataCarro pode ser alterado utilizando o método getDataObject
10     ()
11     dataCarro.getDataObject()->setLocalizacao(100);
12     dataCarro.getDataObject()->getEmergencia()->setTelefoneHospital
13     (123456789);
14
15     //injeção de falhas
16     dataCarro.injectFault();
17
18     cout << "Localização original: " << dataCarro.getDataObject()->
19     getLocalizacao()
20     << endl;
21     cout << "Telefone do Hospital original: "
22     << dataCarro.getDataObject()->getEmergencia()->getTelefoneHospital() <<
23     "\n"
24     << endl;
25
26     //imprime os dados armazenados nas 3 cópias
27     cout << "Valores armazenados nas 3 cópias após a injeção de falha" <<
28     endl;
29     dataCarro.print();
30 }
```

Quadro 19: O objeto carro foi instanciado com os valores iniciais declarados no seu construtor Quadro 14 e foi passado como parâmetro no objeto *TData*. Ao acessar o método *getDataObject* obtém-se como retorno uma referência para o objeto *dataObject* permitindo que se possa alterar os valores de seus atributos e atualizar as cópias com os novos dados.



```
<terminated> TripledData - Setando um objeto.exe [C/C++ Application] C:\workspaces\WorkspaceTCC\TripledData - Setando um objeto\Debug\TripledData - 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 3.5: 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 do Quadro 19.

Capítulo 4

Resultados

Neste capítulo são apresentados os resultados dos testes realizados.

4.1 Desempenho da Biblioteca *FaultRecovery*

Esta seção tem como objetivo demonstrar se a biblioteca *FaultRecovery* pode afetar o tempo de execução de um *firmware*, para isto, foram realizados dois testes, sendo que cada teste foi executado cem vezes. Para realizar os testes, utilizou-se cinco algoritmos de ordenação *bubble sort*, *insertion sort*, *merge sort* e *Comb sort* [46, 47]. Os algoritmos foram implementados para serem executados no microcontrolador *mbed* modelo 1768. Para realizar o teste utilizou-se um vetor de 4096 elementos, totalizando 4kB de memória, tentou-se aumentar a quantidade de elementos do vetor, no entanto quando se tentava alocar um espaço de memória maior que 4KB o *firmware* tinha sua execução interrompida no segundo ciclo de testes, em alguns momentos no primeiro ciclo de testes. Cada ciclo de teste é constituído pela execução dos cinco algoritmos de ordenação e todos eles utilizam o mesmo vetor.

O primeiro teste tem como objetivo medir o tempo de execução dos algoritmos de ordenação simulando um *firmware* implementado por um usuário comum, sem a utilização da biblioteca *FaultRecovery*. Todos os algoritmos compartilham do mesmo vetor, por isso, antes de cada ordenação foi necessário desordenar o vetor para cronometrar o tempo real de ordenação. Para a medição do tempo de execução de cada algoritmo desprezou-se a desordenação do vetor, o tempo foi medido por algoritmo desde o início de sua execução até o fim dela.

O segundo teste tem como objetivo medir o tempo de execução dos algoritmos de ordenação simulando um *firmware* implementado por um usuário comum, utilizando a 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 um pouco antes da execução do próximo estado, neste caso cada algoritmo de ordenação é chamado de estado dentro da máquina de estados criada para executar o teste. Tanto para o primeiro quanto para o segundo teste foram utilizados a mesma quantidade de elementos (4096), a mesma sequência de execução (*bubble*, *insertion*, *selection*, *merge* e *comb*) e o tempo de desordenação do vetor foi desprezado. Ao final das cem execuções do primeiro e do segundo teste, as médias de tempo de

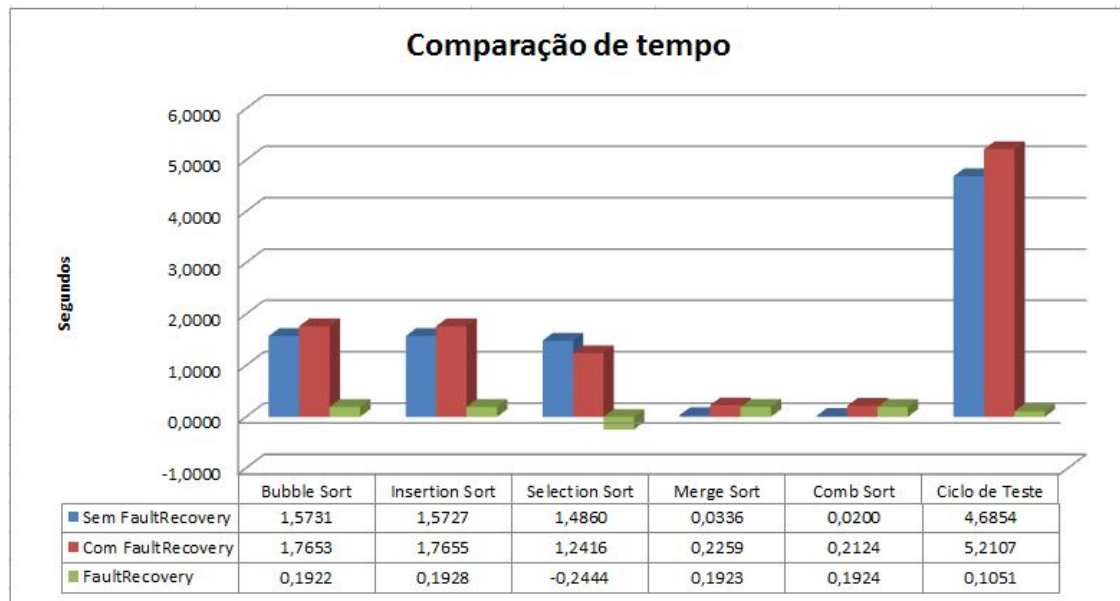


Figura 4.1: Resultado dos testes de medição do tempo de execução dos algoritmos de ordenação sem a biblioteca e com a biblioteca. Percebe-se que com a utilização da biblioteca o tempo de execução dos algoritmos aumentou em média 0,1879 segundos ou 187 milissegundos.

execução para cada algoritmo e para cada ciclo de teste foram calculadas, o tempo de execução da biblioteca *FaultRecovery* também foi calculada obtendo-se o resultado demonstrado na figura 4.1.

4.2 Desempenho e Eficiência da Classe TData

Para testar a redundância de dados da classe TData e se a sua utilização possa causar uma perda de desempenho em algum *firmware*, foram realizados dois tipos de testes. O primeiro não utilizando a classe TData e o segundo utilizando-a, em ambos os testes foram realizados cem ciclos de execução, conforme descrito no primeiro parágrafo da seção 4.1. No entanto houve a necessidade de reduzir o tamanho do vetor de 4.096 para 1.024 elementos, devido a redundância de dados da classe TData, pois ela copia um valor para três endereços de memória diferentes. Se fossem utilizados os 4096 elementos teríamos aproximadamente 16KB de memória alocada, impossibilitando a realização dos testes. O teste que não utilizava a classe TData tem como objetivo medir o tempo de execução de cada algoritmo de ordenação, sendo que a biblioteca *FaultRecovery* não foi utilizada, foram levados em conta apenas o tempo de execução de cada algoritmo. O resultado do primeiro teste foi comparado com o do segundo, obtendo-se o tempo de execução da classe TData conforme demonstrado na figura 4.2. Para iniciar o segundo teste, apenas foi necessário substituir a declaração do vetor de *unsigned short vetor[n]* para um *TData<unsigned short> vetor[n]*, com isso a redundância de dados disponível na classe TData pode ser aplicada nos dados do vetor.

Notou-se uma diferença de aproximadamente 0,26 segundos do tempo de execução de um algoritmo de ordenação sem redundância de dados para um com redundância de dados. No entanto deve-se levar em conta que o teste realizado sem a classe TData estava sujeito

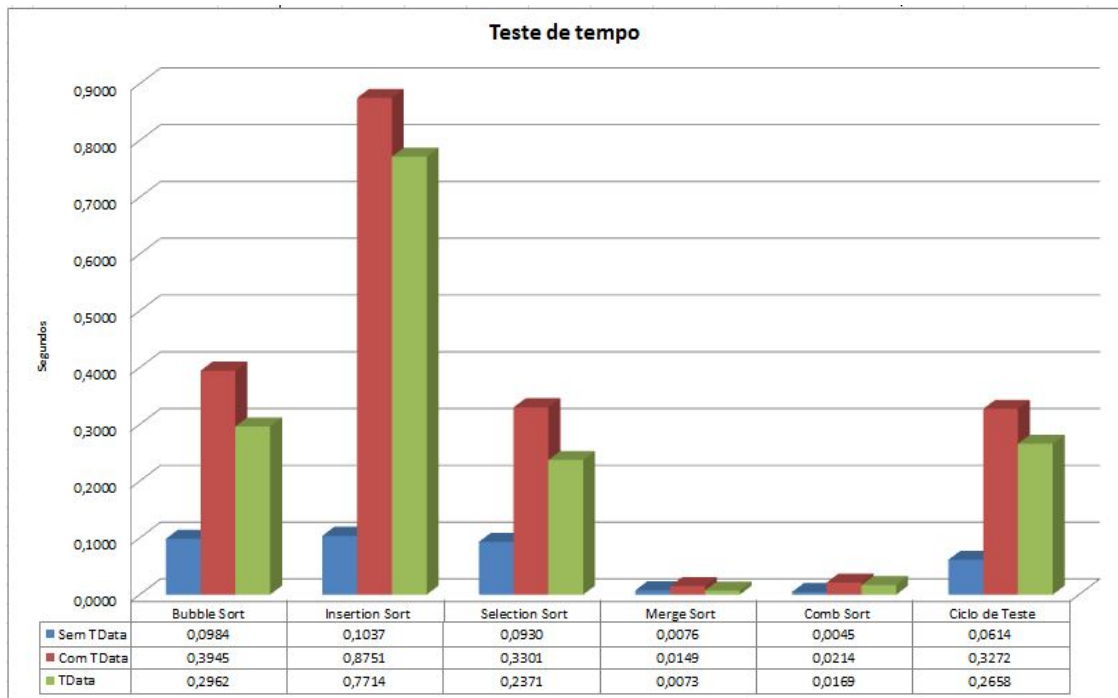


Figura 4.2: O tempo de execução dos algoritmos sem a classe TData foi relativamente baixo, não atingindo 0,15 segundos ou 150 milissegundos para ordenar um vetor de 1024 elementos. Já o tempo dos algoritmos com a Classe TData foi um pouco maior, chegando a passar de 0,32 segundos ou 320 milissegundos. O tempo de execução médio da classe TData para cada algoritmo de ordenação foi de 0,26 segundos ou 260 milissegundos.

a falhar em algum ciclo de teste, e foi o que aconteceu, na figura 4.3 têm-se o resultado da execução do primeiro e do segundo teste. Foram determinados 3 parâmetros de teste para determinar se um ciclo de teste falhou ou não. Como o vetor possui 1024 elementos e o resultado de uma ordenação que inicia em 1 até 1024 é conhecida, após a injeção de falhas no microcontrolador *mbed*, alterando um endereço de memória aleatório, estabeleceu-se que para um ciclo de teste falhar ele deve ter 10%, 25% ou 50% dos 1024 elementos diferentes do resultado conhecido.

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 4.3 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.

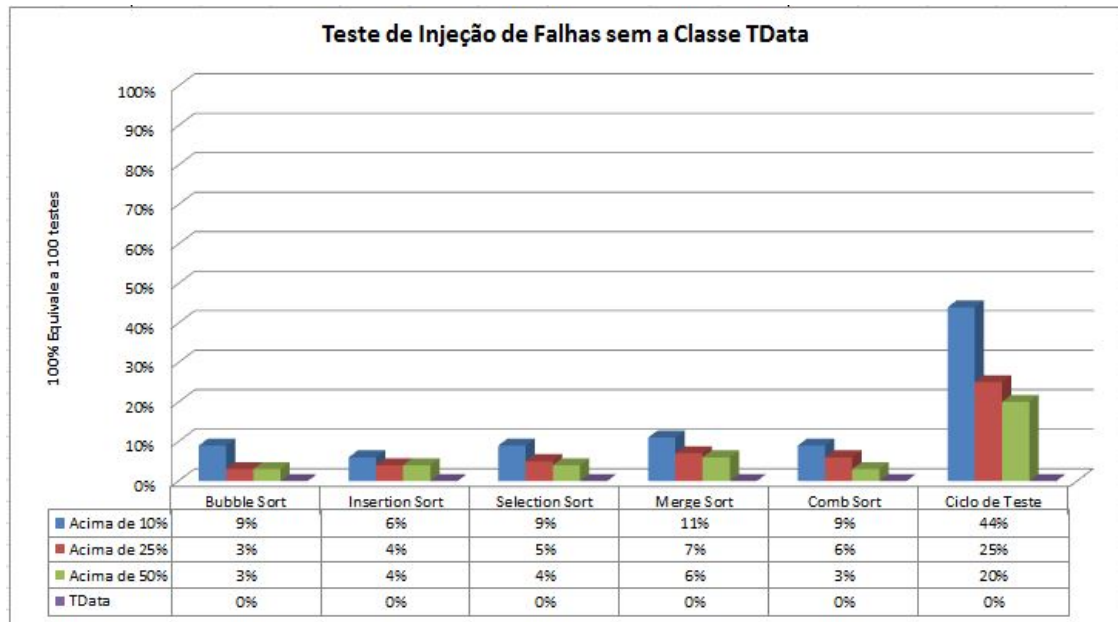


Figura 4.3: Neste figura para as falhas detectadas acima de 10%, 25% e 50% são do teste que não utiliza redundância de dados, por exemplo, para as falhas acima de 10%, aproximadamente 44% dos ciclos de testes falharam. No entanto para o teste que utilizou a redundância de dados disponibilizada pela TData nenhum algoritmo de ordenação e ciclo de teste falharam, ou seja, 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*

Esta seção mostra os resultados dos teste realizados com a biblioteca *FaultRecovery*. O código que não implementa a *FaultRecovery* utilizado para realizar os testes da seção 4.1 não utiliza a biblioteca e foi reaproveitado, ou seja, 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*. Neste caso, quando o *whatchdog* perceber que o microcontrolador está travado, o *mbed* será reinicializado. No entanto se o travamento ocorrer no momento em que os dados coletados por uma estação meteorológica seriam 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* for reiniciado o primeiro estado da máquina de estados será executado, podendo ou não ser o estado responsável por enviar os dados para o 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 reinicializa-se por causa de alguma falha, algum ponto de recuperação predefinido pudesse ser executado. Porém neste trabalho não se implementou uma estação meteorológica para simular esse acontecimento, no entanto utilizou-se algoritmos de ordenação para simular os estados. Foram criados pontos de recuperação para cada algoritmo de ordenação, se em algum momento o microcontrolador travar e reiniciar,

o ponto de recuperação predefinido será executado. Mostra-se na figura 4.4 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.

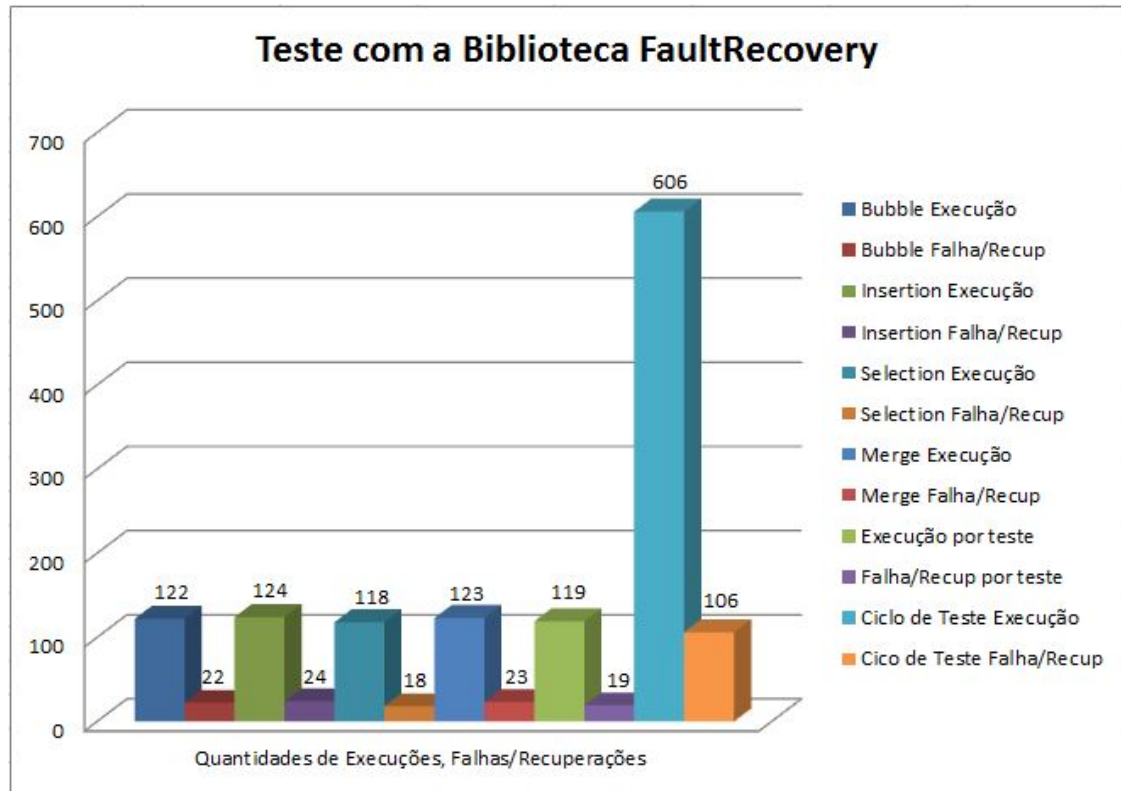


Figura 4.4: Esta figura mostra a quantidade de execuções e falhas de cada algoritmo, inclusive de cada ciclo de teste. Por exemplo, a primeira coluna, da esquerda para e direita, representa o número de execuções do algoritmo *bubble sort*, pode-se perceber que embora fossem executados cem ciclos de teste, 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* modelo 1768. A injeção de falhas na memória flash sorteia um setor aleatório para injetar uma quantidade predefinida de falhas, que podem variar de 256, 512, 1024 ou 4096 bytes. Na figura 4.5 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. Portanto agora a biblioteca *FaultInjector* pode injetar falhas na memória flash do microcontrolador *mbed* modelo 1768.

```

-----
Injetando falha no Setor
-----
25
-----
user reserved flash area: start_address=0x00058000, size=32768 bytes
-----
mendump 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. mendump 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 4.5: 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. O que não era possível antes, no qual o usuário que utilizasse a biblioteca teria que aperfeiçoá-la para que isso fosse possível. A ideia e parte do código da biblioteca *FaultRecovery*, no que diz respeito a criação de estados de uma máquina de estados, está sendo utilizada pelo projeto de extensão Coxim Robótica sediado na UFMS - Campus Coxim. Os alunos estão programando um programa para um carrinho 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* não continuou a execução da máquina de estados em sua sequência original, 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 de falhas execução predefinido 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 faz parte da biblioteca *FaultRecovery*, sendo assim o usuário que utilizar que quiser criar pontos de recuperação e separar o seu código, também poderá proteger dados importantes de seu *firmware*. Portanto a biblioteca foi eficaz na resolução do problema, pois em todos os testes o *firmware* se recuperou após a reinicialização do microcontrolador e 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 para definir qual o valor correto que deverá permanecer em todas as cópias do dado alterado por uma eventual falha. Porém isso não 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 em 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 possível testar o mapeamento de memória no modelo 1768, pois era o único microcontrolador disponível para este trabalho. No entanto 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*, no endereço <https://github.com/cleitonlmeida1>.

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 ela 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 memória de

usuário, mas futuramente poderá ser salva na memória *flash*.

- Aperfeiçoar a biblioteca *FaultRecovery* e a classe TData para diminuir o tempo de processamento em uma aplicação embarcada.

Bibliografia

- [1] NELSON, V. P. Fault-tolerant computing: Fundamental concepts. *Computer*, Los Alamitos, CA, USA, v. 23, p. 19–25, jul 1990.
- [2] CUNHA, A. O que são sistemas embarcados, 2007.
- [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] REIS, E. *Previsão de doenças de plantas*. UPF EDITORA, 2004.
- [9] IAIONE, F.; LIMA, D. G.; GASSEN, F. R. Equipamento para coleta de dados e previsão de doenças na lavoura, 1999.
- [10] 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.
- [11] ZIEGLER, J. Terrestrial cosmic rays. *IBM Journal of Research and Development*, v. 40, n. 1, p. 19–39, Jan 1996.
- [12] 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.
- [13] 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.

- [14] 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.
- [15] VELAZCO, R.; FOUILLAT, P.; REIS, R. *Radiation effects on embedded systems*. Dordrecht: Springer, 2007.
- [16] 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.
- [17] STASSINOPOULOS, E.; RAYMOND, J. P. The space radiation environment for electronics. *Proceedings of the IEEE*, v. 76, n. 11, p. 1423–1442, Nov 1988.
- [18] BOUDENOT, J. C. *Radiation space environment*. Springer. p. 1–9.
- [19] Cinturão de van allen. Disponível em: <<http://geocities.ws/saladefisica5/leituras/vanallen.html>> Acesso em: 17/11/2015.
- [20] 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.
- [21] ZIEGLER, J.; LANFORD, W. *The effect of cosmic rays on computer memories*. Science, 1979. v. 206.
- [22] MAY, T. C.; WOODS, M. H. A new physical mechanism for soft errors in dynamic memories. In: . c1978. p. 33–40.
- [23] YU, H.; FAN, X.; NICOLAIDIS, M. Design trends and challenges of logic soft errors in future nanotechnologies circuits reliability. In: . c2008. p. 651–654.
- [24] ECOFFET, R.; DUZELLIER, S.; TASTET, P.; AICARDI, C.; LABRUNEE, M. Observation of heavy ion induced transients in linear circuits. In: . c1994. p. 72–77.
- [25] AVIZIENIS, A.; KELLY, J. Fault tolerance by design diversity: Concepts and experiments. *Computer*, v. 17, n. 8, p. 67–80, Aug 1984.
- [26] VON NEUMANN, J. Probabilistic logics and the synthesis of reliable organisms from unreliable components. *Automata Studies*, p. 43–98, 1956.
- [27] PEGO, M. O. *Tolerância a falhas através de escalonamento em um sistema multiprocessado*. 2004. Tese de Doutorado - UFMG, Belo Horizonte, 2004.
- [28] CHEN, L.; AVIZIENIS, A. N-version programming: A fault-tolerance approach to reliability of software operation. In: . c1995. p. 113–.
- [29] PRADHAN, D. K. (Ed.). *Fault-tolerant computer system design*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1996.
- [30] 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.

- [31] 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.
- [32] 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.
- [33] GUNNEFLO, U.; KARLSSON, J.; TORIN, J. Evaluation of error detection schemes using fault injection by heavy-ion radiation. In: . c1989. p. 340–347.
- [34] SOTOMA, I. *Afids - arquitetura para injeção de falhas em sistemas distribuídos*. 1997. Dissertação de Mestrado - UFRGS, Porto Alegre, 1997.
- [35] MARTINS, E.; ARLAT, J.; CROUZET, Y.; FABRE, J.; POWELL, D. Testing multi-peer protocols in the presence of faults. *Protocol Test Systems*, p. 77–91, 1989.
- [36] DAWSON, S.; JAHANIAN, F.; MITTON, T. A software fault injection tool on real-time mach. In: . c1995. p. 130–140.
- [37] ROSENBERG, H. A.; SHIN, K. G. Software fault injection and its application in distributed systems. In: . c1993. p. 208–217.
- [38] 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.
- [39] ENGHOLM JR, H. *Engenharia de software na prática*. Novatec Editora Ltda, 2010.
- [40] SENGHER, 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.
- [41] MBED, A. mbed lpc1768, 2016. Disponível em: <<https://developer.mbed.org/platforms/mbed-LPC1768/>> Acesso em: 23/03/2016.
- [42] MBED, A. Lpc1768/66/65/64, 2009. Disponível em: <<http://www.nxp.com/>> Acesso em: 23/03/2016.
- [43] MBED, A. Compilador, 2016. Disponível em: <<https://developer.mbed.org/>> Acesso em: 23/03/2016.
- [44] MBED, A. Iap internal flash write, 2010. Disponível em: <<https://developer.mbed.org/users/okano/code/>> Acesso em: 23/03/2016.
- [45] MEYERS, S. *C++ eficaz 55 maneiras de aprimorar seus programas e projetos*. Porto Alegre: BOOKMAN COMPANHIA EDITORA, 2011.
- [46] HERNANDE, D. N. Estrutura de dados, 2011. Disponível em: <<http://www.ft.unicamp.br/liag/siteEd/>> Acesso em: 03/08/2016.

- [47] 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

As tabelas deste apêndice mostram os resultados individuais dos testes sem a biblioteca *FaultRecovery*, com a biblioteca *FaultRecovery*, sem a classe TData e com a classe TData.

Desempenho do teste sem a biblioteca FaultRecovery

Teste	Bubble Sort	Insertion Sort	Selection Sort	Merge Sort	Comb Sort	Ciclo de Teste
1	1,5731	1,5727	1,2140	0,0336	0,0200	4,4134
2	1,5731	1,5727	1,2137	0,0337	0,0200	4,4133
3	1,5731	1,5727	1,2143	0,0337	0,0200	4,4139
4	1,5731	1,5727	1,2139	0,0337	0,0200	4,4135
5	1,5731	1,5727	1,2139	0,0337	0,0200	4,4135
6	1,5731	1,5727	1,2115	0,0337	0,0200	4,4110
7	1,5731	1,5727	1,2138	0,0337	0,0200	4,4134
8	1,5731	1,5727	1,2137	0,0337	0,0200	4,4133
9	1,5731	1,5727	1,2147	0,0337	0,0200	4,4143
10	1,5731	1,5727	1,2160	0,0337	0,0200	4,4156
11	1,5731	1,5727	1,2157	0,0337	0,0200	4,4153
12	1,5731	1,5727	1,2146	0,0337	0,0200	4,4142
13	1,5731	1,5727	1,2135	0,0337	0,0200	4,4131
14	1,5731	1,5727	1,2154	0,0337	0,0200	4,4150
15	1,5731	1,5727	1,2179	0,0337	0,0200	4,4175
16	1,5731	1,5727	1,2163	0,0337	0,0200	4,4159
17	1,5731	1,5727	1,2131	0,0337	0,0200	4,4127
18	1,5731	1,5727	1,2149	0,0337	0,0200	4,4145
19	1,5731	1,5727	1,2143	0,0337	0,0200	4,4139
20	1,5731	1,5727	1,2149	0,0337	0,0200	4,4145
21	1,5731	1,5727	1,2117	0,0337	0,0200	4,4113
22	1,5731	1,5727	1,2122	0,0337	0,0200	4,4118
23	1,5731	1,5727	1,2151	0,0337	0,0200	4,4147
24	1,5731	1,5727	1,2160	0,0337	0,0200	4,4156
25	1,5731	1,5727	1,2149	0,0337	0,0200	4,4145
26	1,5731	1,5727	1,2187	0,0337	0,0200	4,4183
27	1,5731	1,5727	1,2172	0,0337	0,0200	4,4168
28	1,5731	1,5727	1,2144	0,0337	0,0200	4,4140
29	1,5731	1,5727	1,2138	0,0337	0,0200	4,4134
30	1,5731	1,5727	1,2161	0,0337	0,0200	4,4157
31	1,5731	1,5727	1,2173	0,0337	0,0200	4,4169
32	1,5731	1,5727	1,2229	0,0337	0,0200	4,4225
33	1,5731	1,5727	1,2178	0,0337	0,0200	4,4174
34	1,5731	1,5727	1,2170	0,0337	0,0200	4,4166
35	1,5731	1,5727	1,2197	0,0337	0,0200	4,4193
36	1,5731	1,5727	1,2209	0,0337	0,0200	4,4205
37	1,5731	1,5727	1,2179	0,0337	0,0200	4,4175
38	1,5731	1,5727	1,2453	0,0337	0,0200	4,4449
39	1,5731	1,5727	1,2213	0,0337	0,0200	4,4209
40	1,5731	1,5727	1,2218	0,0337	0,0200	4,4214
41	1,5731	1,5727	1,2251	0,0337	0,0200	4,4247
42	1,5731	1,5727	1,2373	0,0337	0,0200	4,4369
43	1,5731	1,5727	1,2285	0,0337	0,0200	4,4281
44	1,5731	1,5727	1,2271	0,0337	0,0200	4,4267
45	1,5731	1,5727	1,2333	0,0337	0,0200	4,4329

46	1,5731	1,5727	1,2190	0,0337	0,0200	4,4186
47	1,5731	1,5727	1,2288	0,0337	0,0200	4,4284
48	1,5731	1,5727	1,2194	0,0337	0,0200	4,4190
49	1,5731	1,5727	1,2189	0,0337	0,0200	4,4185
50	1,5731	1,5727	1,2222	0,0337	0,0200	4,4218
51	1,5731	1,5727	1,2193	0,0337	0,0200	4,4189
52	1,5731	1,5727	1,2184	0,0337	0,0200	4,4180
53	1,5731	1,5727	1,2202	0,0337	0,0200	4,4197
54	1,5731	1,5727	1,2170	0,0337	0,0200	4,4165
55	1,5731	1,5727	1,2168	0,0337	0,0200	4,4164
56	1,5731	1,5727	1,2123	0,0337	0,0200	4,4119
57	1,5731	1,5727	1,2188	0,0337	0,0200	4,4183
58	1,5731	1,5727	1,2202	0,0337	0,0200	4,4198
59	1,5731	1,5727	1,2181	0,0337	0,0200	4,4176
60	1,5731	1,5727	1,2197	0,0337	0,0200	4,4193
61	1,5731	1,5727	1,2204	0,0337	0,0200	4,4200
62	1,5731	1,5727	1,2163	0,0337	0,0200	4,4159
63	1,5731	1,5727	1,2157	0,0337	0,0200	4,4153
64	1,5731	1,5727	1,2151	0,0337	0,0200	4,4147
65	1,5731	1,5727	1,2170	0,0337	0,0200	4,4166
66	1,5731	1,5727	1,2127	0,0337	0,0200	4,4123
67	1,5731	1,5727	1,2217	0,0337	0,0200	4,4213
68	1,5731	1,5727	1,2171	0,0337	0,0200	4,4167
69	1,5731	1,5727	1,2150	0,0337	0,0200	4,4146
70	1,5731	1,5727	1,2134	0,0337	0,0200	4,4130
71	1,5731	1,5727	1,2164	0,0337	0,0200	4,4160
72	1,5731	1,5727	1,2142	0,0337	0,0200	4,4138
73	1,5731	1,5727	1,2174	0,0337	0,0200	4,4170
74	1,5731	1,5727	1,2166	0,0337	0,0200	4,4162
75	1,5731	1,5727	1,2134	0,0337	0,0200	4,4130
76	1,5731	1,5727	1,2177	0,0337	0,0200	4,4172
77	1,5731	1,5727	1,2137	0,0337	0,0200	4,4133
78	1,5731	1,5727	1,2134	0,0337	0,0200	4,4130
79	1,5731	1,5727	1,2182	0,0337	0,0200	4,4178
80	1,5731	1,5727	1,2130	0,0337	0,0200	4,4125
81	1,5731	1,5727	1,2150	0,0337	0,0200	4,4146
82	1,5731	1,5727	1,2176	0,0337	0,0200	4,4172
83	1,5731	1,5727	1,2148	0,0337	0,0200	4,4143
84	1,5731	1,5727	1,2140	0,0337	0,0200	4,4136
85	1,5731	1,5727	1,2049	0,0337	0,0200	4,4045
86	1,5731	1,5727	1,2129	0,0337	0,0200	4,4125
87	1,5731	1,5727	1,2191	0,0337	0,0200	4,4186
88	1,5731	1,5727	1,2166	0,0337	0,0200	4,4162
89	1,5731	1,5727	1,2190	0,0337	0,0200	4,4186
90	1,5731	1,5727	1,2195	0,0337	0,0200	4,4191
91	1,5731	1,5727	1,2189	0,0337	0,0200	4,4185
92	1,5731	1,5727	1,2173	0,0337	0,0200	4,4169
93	1,5731	1,5727	1,2176	0,0337	0,0200	4,4171
94	1,5731	1,5727	1,2173	0,0337	0,0200	4,4169
95	1,5731	1,5727	1,2179	0,0337	0,0200	4,4175

96	1,5731	1,5727	1,2145	0,0337	0,0200	4,4141
97	1,5731	1,5727	1,2168	0,0337	0,0200	4,4164
98	1,5731	1,5727	1,2164	0,0337	0,0200	4,4160
99	1,5731	1,5727	1,2240	0,0337	0,0200	4,4236
100	1,5731	1,5727	1,2209	0,0337	0,0200	4,4204
Média	1,5731	1,5727	1,2176	0,0337	0,0200	4,4171

Desempenho do teste com a biblioteca FaultRecovery

Teste	BubbleSort	InsertionSort	SelectionSort	MergeSort	CombSort	Média do ciclo de teste
1	1,7392	1,7468	1,4860	0,1979	0,1841	5,3540
2	1,7371	1,7385	1,4860	0,1995	0,1859	5,3469
3	1,7374	1,7392	1,4860	0,1981	0,1858	5,3465
4	1,7373	1,7372	1,4860	0,1988	0,1858	5,3451
5	1,7382	1,7401	1,4860	0,1994	0,1849	5,3486
6	1,7377	1,7396	1,4860	0,1976	0,1834	5,3443
7	1,7376	1,7394	1,4860	0,1977	0,1835	5,3442
8	1,7369	1,7366	1,4860	0,1989	0,1842	5,3425
9	1,7351	1,7374	1,4860	0,2001	0,1867	5,3453
10	1,7383	1,7391	1,4860	0,1970	0,1842	5,3445
11	1,7360	1,7343	1,4860	0,2002	0,1848	5,3413
12	1,7401	1,7391	1,4860	0,1986	0,1853	5,3491
13	1,7388	1,7363	1,4860	0,2016	0,1873	5,3499
14	1,7397	1,7386	1,4860	0,1971	0,1865	5,3478
15	1,7454	1,7393	1,4860	0,1996	0,1878	5,3581
16	1,7386	1,7388	1,4860	0,2005	0,1901	5,3540
17	1,7403	1,7359	1,4860	0,1988	0,1843	5,3452
18	1,7362	1,7338	1,4860	0,1972	0,1847	5,3379
19	1,7398	1,7394	1,4860	0,1976	0,1852	5,3478
20	1,7378	1,7379	1,4860	0,2011	0,1892	5,3520
21	1,7393	1,7395	1,4860	0,1972	0,1842	5,3462
22	1,7376	1,7363	1,4860	0,1985	0,1856	5,3439
23	1,7365	1,7384	1,4860	0,1983	0,1836	5,3428
24	1,7390	1,7381	1,4860	0,1953	0,1838	5,3421
25	1,7371	1,7381	1,4860	0,1994	0,1863	5,3469
26	1,7479	1,7389	1,4860	0,2001	0,1862	5,3591
27	1,7418	1,7432	1,4860	0,2020	0,1864	5,3594
28	1,7412	1,7397	1,4860	0,2010	0,1864	5,3543
29	1,7393	1,7371	1,4860	0,1984	0,1862	5,3470
30	1,7382	1,7397	1,4860	0,2004	0,1874	5,3517
31	1,7401	1,7401	1,4860	0,2010	0,1850	5,3521
32	1,7394	1,7407	1,4860	0,2015	0,1894	5,3570
33	1,7369	1,7379	1,4860	0,2005	0,1881	5,3493
34	1,7418	1,7430	1,4860	0,2017	0,1912	5,3637
35	1,7411	1,7409	1,4860	0,2040	0,1912	5,3632
36	1,7464	1,7424	1,4860	0,2048	0,1929	5,3725
37	1,7443	1,7427	1,4860	0,2048	0,1923	5,3700
38	1,7455	1,7442	1,4860	0,2049	0,1926	5,3732
39	1,7475	1,7485	1,4860	0,2035	0,1984	5,3839
40	1,7439	1,7432	1,4860	0,2041	0,1923	5,3696
41	1,7455	1,7463	1,4860	0,2153	0,2018	5,3949
42	1,7501	1,7563	1,4860	0,2167	0,2004	5,4095
43	1,7580	1,7548	1,4860	0,2102	0,1928	5,4018
44	1,7498	1,7477	1,4860	0,2123	0,1988	5,3945
45	1,7562	1,7502	1,4860	0,2105	0,1945	5,3974

46	1,7440	1,7472	1,4860	0,2009	0,1888	5,3669
47	1,7422	1,7420	1,4860	0,2016	0,1868	5,3586
48	1,7412	1,7426	1,4860	0,2020	0,1884	5,3601
49	1,7430	1,7421	1,4860	0,2033	0,1918	5,3662
50	1,7409	1,7425	1,4860	0,2009	0,1884	5,3587
51	1,7430	1,7432	1,4860	0,2037	0,1915	5,3675
52	1,7443	1,7408	1,4860	0,2038	0,1920	5,3669
53	1,7405	1,7429	1,4860	0,2019	0,1922	5,3635
54	1,7436	1,7406	1,4860	0,2120	0,1878	5,3699
55	1,7427	1,7408	1,4860	0,2002	0,1842	5,3539
56	1,7362	1,7386	1,4860	0,1987	0,1827	5,3422
57	1,7390	1,7399	1,4860	0,1996	0,1849	5,3493
58	1,7367	1,7386	1,4860	0,2036	0,1888	5,3536
59	1,7406	1,7398	1,4860	0,2043	0,1911	5,3618
60	1,7443	1,7448	1,4860	0,2051	0,1900	5,3702
61	1,7398	1,7420	1,4860	0,2015	0,1887	5,3580
62	1,7492	1,7414	1,4860	0,2010	0,1867	5,3642
63	1,7396	1,7418	1,4860	0,2012	0,1886	5,3572
64	1,7430	1,7412	1,4860	0,2042	0,1874	5,3617
65	1,7386	1,7398	1,4860	0,2008	0,1852	5,3505
66	1,7433	1,7404	1,4860	0,1993	0,1869	5,3558
67	1,7402	1,7426	1,4860	0,2017	0,1889	5,3594
68	1,7431	1,7423	1,4860	0,1988	0,1873	5,3575
69	1,7422	1,7393	1,4860	0,2001	0,1855	5,3531
70	1,7373	1,7359	1,4860	0,1978	0,1846	5,3417
71	1,7402	1,7381	1,4860	0,1987	0,1878	5,3508
72	1,7387	1,7372	1,4860	0,1998	0,1854	5,3471
73	1,7391	1,7394	1,4860	0,2004	0,1860	5,3509
74	1,7380	1,7464	1,4860	0,2019	0,1841	5,3564
75	1,7376	1,7389	1,4860	0,1980	0,1854	5,3459
76	1,7371	1,7398	1,4860	0,2022	0,1876	5,3527
77	1,7421	1,7411	1,4860	0,1986	0,1839	5,3516
78	1,7371	1,7383	1,4860	0,1974	0,1835	5,3424
79	1,7362	1,7351	1,4860	0,1983	0,1849	5,3405
80	1,7400	1,7380	1,4860	0,1999	0,1859	5,3497
81	1,7368	1,7396	1,4860	0,2000	0,1853	5,3476
82	1,7380	1,7383	1,4860	0,1997	0,1869	5,3488
83	1,7388	1,7378	1,4860	0,1963	0,1847	5,3437
84	1,7377	1,7412	1,4860	0,1974	0,1857	5,3479
85	1,7378	1,7408	1,4860	0,1986	0,1849	5,3481
86	1,7354	1,7365	1,4860	0,1978	0,1853	5,3410
87	1,7378	1,7381	1,4860	0,2008	0,1867	5,3495
88	1,7403	1,7390	1,4860	0,2008	0,1866	5,3527
89	1,7401	1,7418	1,4860	0,2006	0,1869	5,3554
90	1,7404	1,7416	1,4860	0,2001	0,1893	5,3575
91	1,7404	1,7418	1,4860	0,2028	0,1888	5,3597
92	1,7409	1,7466	1,4860	0,2042	0,1874	5,3651
93	1,7420	1,7435	1,4860	0,2045	0,1891	5,3651
94	1,7440	1,7413	1,4860	0,2019	0,1908	5,3640
95	1,7427	1,7408	1,4860	0,2024	0,1857	5,3575

96	1,7383	1,7419	1,4860	0,1986	0,1853	5,3501
97	1,7390	1,7391	1,4860	0,1980	0,1857	5,3478
98	1,7395	1,7381	1,4860	0,2030	0,1912	5,3577
99	1,7627	1,7433	1,4860	0,2042	0,1943	5,3905
100	1,7448	1,7432	1,4860	0,2067	0,1893	5,3700
Média	1,7409	1,7408	1,4860	0,2013	0,1877	5,3567

Teste de recuperação de falhas da biblioteca FaultRecovery

Teste	Bubble Sort		Insertion Sort		Selection Sort	
	Execução	Falha/Recup	Execução	Falha/Recup	Execução	Falha/Recup
1	2	1	1	0	1	0
2	1	0	1	0	1	0
3	2	1	1	0	1	0
4	2	1	1	0	2	1
5	1	0	1	0	2	1
6	1	0	1	0	1	0
7	1	0	1	0	1	0
8	1	0	1	0	1	0
9	1	0	1	0	2	1
10	1	0	1	0	1	0
11	1	0	1	0	1	0
12	1	0	1	0	1	0
13	1	0	1	0	1	0
14	1	0	1	0	1	0
15	1	0	1	0	1	0
16	1	0	2	1	1	0
17	1	0	1	0	2	1
18	1	0	1	0	1	0
19	1	0	1	0	1	0
20	1	0	1	0	1	0
21	1	0	1	0	1	0
22	1	0	1	0	1	0
23	1	0	1	0	1	0
24	1	0	1	0	1	0
25	1	0	1	0	1	0
26	1	0	1	0	1	0
27	1	0	1	0	2	1
28	1	0	3	2	1	0
29	1	0	1	0	1	0
30	1	0	1	0	1	0
31	1	0	1	0	1	0
32	2	1	1	0	1	0
33	1	0	2	1	1	0
34	1	0	1	0	1	0
35	1	0	4	3	1	0
36	1	0	1	0	2	1
37	1	0	1	0	1	0
38	1	0	1	0	1	0
39	1	0	1	0	3	2
40	1	0	2	1	1	0
41	2	1	1	0	1	0
42	1	0	2	1	1	0
43	1	0	1	0	1	0
44	1	0	1	0	1	0

45	2	1	2	1	1	0
46	2	1	1	0	2	1
47	3	2	1	0	1	0
48	1	0	1	0	1	0
49	1	0	1	0	3	2
50	1	0	1	0	1	0
51	1	0	1	0	1	0
52	2	1	1	0	2	1
53	1	0	1	0	1	0
54	1	0	1	0	2	1
55	1	0	1	0	1	0
56	1	0	1	0	1	0
57	1	0	1	0	1	0
58	1	0	1	0	1	0
59	2	1	1	0	1	0
60	2	1	1	0	3	2
61	1	0	1	0	1	0
62	1	0	2	1	1	0
63	2	1	1	0	1	0
64	1	0	1	0	1	0
65	1	0	1	0	1	0
66	1	0	1	0	1	0
67	1	0	2	1	2	1
68	1	0	1	0	1	0
68	1	0	1	0	1	0
70	1	0	2	1	1	0
71	1	0	1	0	1	0
72	2	1	1	0	1	0
73	1	0	2	1	3	2
74	1	0	1	0	1	0
75	1	0	1	0	1	0
76	1	0	2	1	1	0
77	2	1	1	0	1	0
78	1	0	1	0	1	0
79	1	0	2	1	1	0
80	3	2	1	0	1	0
81	1	0	1	0	1	0
82	1	0	3	2	1	0
83	1	0	1	0	1	0
84	2	1	1	0	1	0
85	2	1	1	0	1	0
86	1	0	1	0	1	0
87	1	0	2	1	1	0
88	2	1	1	0	1	0
89	1	0	2	1	1	0
90	1	0	1	0	1	0
91	1	0	2	1	1	0
92	1	0	1	0	1	0
93	2	1	1	0	1	0
94	2	1	3	2	1	0

95	1	0	1	0	2	1
96	1	0	1	0	1	0
97	1	0	1	0	1	0
98	1	0	1	0	1	0
99	1	0	2	1	1	0
100	1	0	1	0	2	1
Total	122	22	124	24	120	20

Teste de recuperação de falhas da biblioteca FaultRecovery						
Teste	Merge Sort		Comb Sort		Ciclo de Teste	
	Execução	Falha/Recup	Execução	Falha/Recup	Excecução	Falha
1	1	0	1	0	6	1
2	1	0	1	0	5	0
3	1	0	1	0	6	1
4	1	0	1	0	7	2
5	1	0	1	0	6	1
6	1	0	1	0	5	0
7	2	1	1	0	6	1
8	1	0	1	0	5	0
9	1	0	3	2	8	3
10	1	0	1	0	5	0
11	1	0	1	0	5	0
12	1	0	1	0	5	0
13	1	0	1	0	5	0
14	1	0	1	0	5	0
15	1	0	1	0	5	0
16	2	1	3	2	9	4
17	1	0	1	0	6	1
18	1	0	1	0	5	0
19	1	0	1	0	5	0
20	1	0	1	0	5	0
21	1	0	1	0	5	0
22	1	0	1	0	5	0
23	1	0	1	0	5	0
24	3	2	1	0	7	2
25	2	1	1	0	6	1
26	1	0	1	0	5	0
27	1	0	2	1	7	2
28	1	0	1	0	7	2
29	1	0	1	0	5	0
30	1	0	1	0	5	0
31	1	0	1	0	5	0
32	2	1	1	0	7	2
33	1	0	1	0	6	1
34	1	0	1	0	5	0
35	2	1	1	0	9	4
36	1	0	3	2	8	3
37	2	1	1	0	6	1

38	1	0	1	0	5	0
39	1	0	1	0	7	2
40	3	2	1	0	8	3
41	2	1	1	0	7	2
42	1	0	3	2	8	3
43	1	0	1	0	5	0
44	1	0	1	0	5	0
45	1	0	1	0	7	2
46	1	0	1	0	7	2
47	1	0	2	1	8	3
48	2	1	1	0	6	1
49	1	0	1	0	7	2
50	1	0	1	0	5	0
51	2	1	1	0	6	1
52	1	0	1	0	7	2
53	2	1	1	0	6	1
54	1	0	1	0	6	1
55	1	0	1	0	5	0
56	1	0	1	0	5	0
57	2	1	1	0	6	1
58	1	0	1	0	5	0
59	1	0	2	1	7	2
60	1	0	2	1	9	4
61	1	0	1	0	5	0
62	1	0	2	1	7	2
63	1	0	1	0	6	1
64	1	0	1	0	5	0
65	1	0	2	1	6	1
66	2	1	1	0	6	1
67	1	0	1	0	7	2
68	1	0	1	0	5	0
68	1	0	1	0	5	0
70	2	1	1	0	7	2
71	1	0	1	0	5	0
72	2	1	1	0	7	2
73	1	0	1	0	8	3
74	1	0	1	0	5	0
75	1	0	2	1	6	1
76	1	0	1	0	6	1
77	3	2	1	0	8	3
78	1	0	1	0	5	0
79	1	0	1	0	6	1
80	2	1	1	0	8	3
81	1	0	1	0	5	0
82	1	0	1	0	7	2
83	1	0	1	0	5	0
84	1	0	1	0	6	1
85	1	0	1	0	6	1
86	1	0	1	0	5	0
87	1	0	2	1	7	2

88	1	0	2	1	7	2
89	1	0	1	0	6	1
90	1	0	1	0	5	0
91	1	0	1	0	6	1
92	1	0	2	1	6	1
93	1	0	1	0	6	1
94	1	0	2	1	9	4
95	1	0	1	0	6	1
96	2	1	1	0	6	1
97	1	0	1	0	5	0
98	1	0	1	0	5	0
99	2	1	1	0	7	2
100	1	0	1	0	6	1
Total	123	23	119	19	608	108

Desempenho sem a classe Tdata

Teste	Bubble Sort		Insertion Sort		Selction Sort		Merge Sort		Comb Sort	
	Tempo	Falha %	Tempo	Falha %	Tempo	Falha %	Tempo	Falha %	Tempo	Falha %
1	0,0984	0,00	0,1037	0,00	0,0930	0,00	0,0076	0,00	0,0045	32,62
2	0,0984	0,00	0,1037	0,00	0,0930	0,00	0,0076	0,00	0,0045	0,00
3	0,0984	0,00	0,1037	0,00	0,0930	0,00	0,0076	0,00	0,0045	0,00
4	0,0984	0,00	0,1035	79,69	0,0930	79,69	0,0076	79,69	0,0045	79,69
5	0,0984	0,00	0,1037	0,00	0,0930	0,00	0,0076	0,00	0,0045	0,00
6	0,0984	0,00	0,1037	0,00	0,0930	0,00	0,0076	0,00	0,0045	0,00
7	0,0984	0,00	0,1037	0,00	0,0930	0,00	0,0076	7,42	0,0045	7,42
8	0,0984	0,00	0,1037	0,00	0,0930	0,00	0,0076	0,00	0,0045	0,00
9	0,0984	0,00	0,1037	0,00	0,0930	0,00	0,0076	0,00	0,0045	0,00
10	0,0984	0,00	0,1037	0,00	0,0930	0,00	0,0076	0,00	0,0045	0,00
11	0,0984	13,77	0,1037	13,77	0,0930	13,77	0,0076	13,77	0,0045	13,77
12	0,0984	0,00	0,1037	0,00	0,0930	0,00	0,0076	0,00	0,0045	0,00
13	0,0984	4,79	0,1037	4,79	0,0930	4,79	0,0076	4,79	0,0045	4,79
14	0,0984	15,92	0,1037	15,92	0,0930	15,92	0,0076	15,92	0,0045	15,92
15	0,0984	0,00	0,1037	0,00	0,0930	0,00	0,0076	11,72	0,0045	11,72
16	0,0984	0,00	0,1037	0,00	0,0930	0,00	0,0076	0,00	0,0045	0,00
17	0,0984	0,00	0,1037	0,00	0,0930	0,00	0,0076	8,20	0,0045	8,20
18	0,0984	0,00	0,1037	2,34	0,0930	2,34	0,0076	2,34	0,0045	2,34
19	0,0984	0,00	0,1037	0,00	0,0930	0,00	0,0076	24,51	0,0045	94,92
20	0,0984	0,00	0,1037	0,00	0,0930	0,00	0,0076	0,00	0,0045	0,00
21	0,0984	0,00	0,1037	0,00	0,0930	0,00	0,0076	0,00	0,0045	4,30
22	0,0984	0,00	0,1037	0,00	0,0930	0,00	0,0076	0,00	0,0045	0,00
23	0,0984	0,00	0,1037	0,00	0,0930	0,00	0,0076	0,00	0,0045	0,00
24	0,0984	0,00	0,1037	0,00	0,0930	0,00	0,0076	21,39	0,0045	21,39
25	0,0984	0,00	0,1037	0,00	0,0930	52,73	0,0076	52,73	0,0045	52,73
26	0,0984	0,00	0,1037	0,00	0,0930	21,19	0,0076	21,19	0,0045	21,19
27	0,0984	0,00	0,1037	0,00	0,0930	0,00	0,0076	0,00	0,0045	0,00
28	0,0984	0,00	0,1037	0,00	0,0930	0,00	0,0076	0,00	0,0045	0,00
29	0,0984	0,00	0,1037	0,00	0,0930	0,00	0,0076	0,00	0,0045	0,00
30	0,0984	0,00	0,1037	0,00	0,0930	0,00	0,0076	0,00	0,0045	6,05
31	0,0984	0,00	0,1037	0,00	0,0930	0,00	0,0076	0,00	0,0045	0,00
32	0,0984	0,00	0,1037	0,00	0,0930	0,00	0,0076	0,00	0,0045	0,00
33	0,0984	11,33	0,1037	11,82	0,0930	11,82	0,0076	11,82	0,0045	11,82
34	0,0984	0,00	0,1037	0,00	0,0930	0,00	0,0076	0,00	0,0045	0,00
35	0,0984	0,00	0,1035	80,96	0,0930	80,96	0,0076	80,96	0,0045	80,96
36	0,0984	0,00	0,1037	0,00	0,0930	0,00	0,0076	60,06	0,0045	60,06
37	0,0984	0,00	0,1037	0,00	0,0930	0,00	0,0076	0,00	0,0045	0,00
38	0,0984	0,00	0,1037	0,00	0,0930	0,00	0,0076	0,00	0,0045	0,00
39	0,0984	0,00	0,1037	0,00	0,0930	0,00	0,0076	0,00	0,0045	0,00
40	0,0984	13,28	0,1037	13,28	0,0930	13,28	0,0076	13,28	0,0045	13,28
41	0,0984	0,00	0,1037	0,00	0,0930	0,00	0,0076	0,00	0,0045	22,95
42	0,0984	0,00	0,1037	0,00	0,0930	0,00	0,0076	0,00	0,0045	0,00
43	0,0984	0,00	0,1037	0,00	0,0930	0,00	0,0076	0,00	0,0045	0,00
44	0,0984	0,00	0,1037	0,00	0,0930	0,00	0,0076	0,00	0,0045	0,00

45	0,0984	0,20	0,1037	0,20	0,0930	0,20	0,0076	0,20	0,0045	0,20
46	0,0984	0,00	0,1037	0,00	0,0930	0,00	0,0076	0,00	0,0045	0,00
47	0,0984	0,00	0,1037	0,00	0,0930	0,00	0,0076	0,00	0,0045	23,93
48	0,0984	0,00	0,1037	0,00	0,0930	0,00	0,0076	0,00	0,0045	0,00
49	0,0984	0,00	0,1037	1,86	0,0930	1,86	0,0076	1,86	0,0045	4,49
50	0,0984	5,96	0,1037	16,50	0,0930	16,50	0,0076	16,50	0,0045	16,50
51	0,0984	0,00	0,1037	0,00	0,0930	0,29	0,0076	0,29	0,0045	0,29
52	0,0984	0,00	0,1037	17,77	0,0930	17,77	0,0076	78,61	0,0045	78,61
53	0,0984	0,00	0,1037	0,00	0,0930	92,38	0,0076	92,38	0,0045	92,38
54	0,0984	0,00	0,1037	0,00	0,0930	15,04	0,0076	15,04	0,0045	15,04
55	0,0984	13,96	0,1037	13,96	0,0930	13,96	0,0076	13,96	0,0045	13,96
56	0,0984	0,00	0,1037	0,00	0,0930	0,00	0,0076	66,11	0,0045	66,11
57	0,0984	0,00	0,1037	0,00	0,0930	0,00	0,0076	0,00	0,0045	21,97
58	0,0984	2,93	0,1037	2,93	0,0930	2,93	0,0076	2,93	0,0045	2,93
59	0,0984	0,00	0,1037	0,00	0,0930	0,00	0,0076	0,00	0,0045	0,00
60	0,0984	0,00	0,1037	0,00	0,0930	0,00	0,0076	0,00	0,0045	0,00
61	0,0984	0,00	0,1037	0,00	0,0930	0,00	0,0076	16,99	0,0045	16,99
62	0,0984	58,30	0,1037	58,30	0,0930	58,30	0,0076	58,30	0,0045	58,30
63	0,0984	0,00	0,1037	0,00	0,0930	0,00	0,0076	0,00	0,0045	0,00
64	0,0984	0,00	0,1037	0,00	0,0930	97,95	0,0076	97,95	0,0045	97,95
65	0,0984	0,00	0,1037	0,00	0,0930	0,00	0,0076	0,00	0,0045	92,68
66	0,0984	0,00	0,1037	0,00	0,0930	0,00	0,0076	0,00	0,0045	0,00
67	0,0984	1,46	0,1037	1,46	0,0930	1,46	0,0076	1,46	0,0045	1,46
68	0,0984	0,00	0,1037	0,00	0,0930	0,00	0,0076	0,00	0,0045	0,00
69	0,0984	0,00	0,1036	69,34	0,0930	69,34	0,0076	69,34	0,0045	69,34
70	0,0984	0,00	0,1037	0,00	0,0930	0,00	0,0076	0,00	0,0045	0,00
71	0,0984	5,57	0,1036	51,56	0,0930	51,56	0,0076	51,56	0,0045	51,56
72	0,0984	0,00	0,1037	0,00	0,0930	0,00	0,0076	0,00	0,0045	0,00
73	0,0984	0,00	0,1037	0,00	0,0930	0,00	0,0076	0,00	0,0045	0,00
74	0,0984	0,00	0,1037	0,00	0,0930	0,00	0,0076	0,00	0,0045	50,00
75	0,0984	0,00	0,1037	0,00	0,0930	0,00	0,0076	0,00	0,0045	0,00
76	0,0984	0,00	0,1037	0,00	0,0930	0,00	0,0076	0,00	0,0045	0,00
77	0,0984	5,76	0,1037	5,76	0,0930	48,14	0,0076	55,08	0,0045	55,08
78	0,0984	0,00	0,1037	0,00	0,0930	0,00	0,0076	0,00	0,0045	0,00
79	0,0984	0,00	0,1037	0,00	0,0930	0,00	0,0076	0,00	0,0045	0,00
80	0,0984	0,00	0,1037	0,00	0,0930	0,00	0,0076	0,00	0,0045	0,00
81	0,0984	0,00	0,1037	0,00	0,0930	0,00	0,0076	0,00	0,0045	91,99
82	0,0984	0,00	0,1037	0,00	0,0930	0,00	0,0076	66,80	0,0045	66,80
83	0,0984	0,00	0,1037	0,00	0,0930	0,00	0,0076	0,00	0,0045	0,00
84	0,0984	0,00	0,1037	0,00	0,0930	0,00	0,0076	0,00	0,0045	0,00
85	0,0984	0,00	0,1037	4,10	0,0930	24,61	0,0076	24,61	0,0045	24,61
86	0,0984	0,00	0,1037	3,13	0,0930	3,13	0,0076	3,13	0,0045	3,13
87	0,0984	0,00	0,1037	0,00	0,0930	0,00	0,0076	35,94	0,0045	28,81
88	0,0984	0,00	0,1037	0,00	0,0930	0,00	0,0076	0,00	0,0045	0,00
89	0,0984	11,82	0,1037	11,82	0,0930	11,82	0,0076	11,82	0,0045	11,82
90	0,0984	0,00	0,1037	0,00	0,0930	0,00	0,0076	0,00	0,0045	12,50
91	0,0984	51,56	0,1037	51,56	0,0930	51,56	0,0076	51,56	0,0045	62,50
92	0,0984	0,00	0,1037	0,00	0,0930	0,00	0,0076	18,46	0,0045	18,46
93	0,0984	0,00	0,1037	0,00	0,0930	0,00	0,0076	0,00	0,0045	0,00
94	0,0984	0,00	0,1037	0,00	0,0930	0,00	0,0076	38,67	0,0045	38,67

95	0,0984	55,37	0,1037	55,37	0,0930	55,37	0,0076	55,37	0,0045	55,37
96	0,0984	0,00	0,1037	1,56	0,0930	14,65	0,0076	14,65	0,0045	14,65
97	0,0984	0,00	0,1037	0,00	0,0930	51,27	0,0076	87,70	0,0045	87,70
98	0,0984	0,00	0,1037	0,00	0,0930	0,00	0,0076	59,57	0,0045	59,57
99	0,0984	0,00	0,1037	0,00	0,0930	0,00	0,0076	5,08	0,0045	5,08
100	0,0984	0,00	0,1037	0,00	0,0930	0,00	0,0076	0,00	0,0045	40,63
Total	0,0984		0,1037		0,0930		0,0076		0,0045	0,0614

Desempenho com a classe Tdata

Teste	BubbleSort		InsertionSort		SelctionSort		MergeSort		CombSort	
	Tempo	Falha %	Tempo	Falha %	Tempo	Falha %	Tempo	Falha %	Tempo	Falha %
1	0,3932	0	0,8840	0	0,3339	0	0,0151	0	0,0216	0,0
2	0,3932	0	0,8840	0	0,3339	0	0,0151	0	0,0216	0,0
3	0,3932	0	0,8840	0	0,3339	0	0,0151	0	0,0216	0,0
4	0,3932	0	0,8840	0	0,3339	0	0,0151	0	0,0216	0,0
5	0,3932	0	0,8840	0	0,3339	0	0,0151	0	0,0216	0,0
6	0,3932	0	0,8840	0	0,3339	0	0,0151	0	0,0216	0,0
7	0,3932	0	0,8840	0	0,3339	0	0,0151	0	0,0216	0,0
8	0,3932	0	0,8840	0	0,3339	0	0,0151	0	0,0216	0,0
9	0,3932	0	0,8840	0	0,3339	0	0,0151	0	0,0216	0,0
10	0,3932	0	0,8840	0	0,3339	0	0,0151	0	0,0216	0,0
11	0,3932	0	0,8840	0	0,3339	0	0,0151	0	0,0216	0,0
12	0,3932	0	0,8840	0	0,3339	0	0,0151	0	0,0216	0,0
13	0,3932	0	0,8840	0	0,3339	0	0,0151	0	0,0216	0,0
14	0,3932	0	0,8840	0	0,3339	0	0,0151	0	0,0216	0,0
15	0,3932	0	0,8840	0	0,3339	0	0,0151	0	0,0216	0,0
16	0,3932	0	0,8840	0	0,3339	0	0,0151	0	0,0216	0,0
17	0,3932	0	0,8840	0	0,3339	0	0,0151	0	0,0216	0,0
18	0,3932	0	0,8840	0	0,3339	0	0,0151	0	0,0216	0,0
19	0,3932	0	0,8840	0	0,3339	0	0,0151	0	0,0216	0,0
20	0,3932	0	0,8840	0	0,3339	0	0,0151	0	0,0216	0,0
21	0,3932	0	0,8840	0	0,3339	0	0,0151	0	0,0216	0,0
22	0,3932	0	0,8840	0	0,3339	0	0,0151	0	0,0216	0,0
23	0,3932	0	0,8840	0	0,3339	0	0,0151	0	0,0216	0,0
24	0,3932	0	0,8840	0	0,3339	0	0,0151	0	0,0216	0,0
25	0,3932	0	0,8840	0	0,3339	0	0,0151	0	0,0216	0,0
26	0,3932	0	0,8840	0	0,3339	0	0,0151	0	0,0216	0,0
27	0,3932	0	0,8840	0	0,3339	0	0,0151	0	0,0216	0,0
28	0,3932	0	0,8840	0	0,3339	0	0,0151	0	0,0216	0,0
29	0,3932	0	0,8840	0	0,3339	0	0,0151	0	0,0216	0,0
30	0,3932	0	0,8840	0	0,3339	0	0,0151	0	0,0216	0,0
31	0,3932	0	0,8840	0	0,3339	0	0,0151	0	0,0216	0,0
32	0,3932	0	0,8840	0	0,3339	0	0,0151	0	0,0216	0,0
33	0,3932	0	0,8840	0	0,3339	0	0,0151	0	0,0216	0,0
34	0,3932	0	0,8840	0	0,3339	0	0,0151	0	0,0216	0,0
35	0,3932	0	0,8840	0	0,3339	0	0,0151	0	0,0216	0,0
36	0,3932	0	0,8840	0	0,3339	0	0,0151	0	0,0216	0,0
37	0,3932	0	0,8840	0	0,3339	0	0,0151	0	0,0216	0,0
38	0,3932	0	0,8840	0	0,3339	0	0,0151	0	0,0216	0,0
39	0,3932	0	0,8840	0	0,3339	0	0,0151	0	0,0216	0,0
40	0,3932	0	0,8840	0	0,3339	0	0,0151	0	0,0216	0,0
41	0,3932	0	0,8840	0	0,3339	0	0,0151	0	0,0216	0,0
42	0,3932	0	0,8840	0	0,3339	0	0,0151	0	0,0216	0,0
43	0,3932	0	0,8840	0	0,3339	0	0,0151	0	0,0216	0,0
44	0,3932	0	0,8840	0	0,3339	0	0,0151	0	0,0216	0,0
45	0,3932	0	0,8840	0	0,3339	0	0,0151	0	0,0216	0,0

[illegible]

96	0,4585	0	0,8840	0	0,3283	0	0,0151	0	0,0216	0,0
97	0,4585	0	0,8840	0	0,3283	0	0,0151	0	0,0216	0,0
98	0,4585	0	0,8840	0	0,3283	0	0,0151	0	0,0216	0,0
99	0,4585	0	0,8840	0	0,3283	0	0,0151	0	0,0216	0,0
100	0,4585	0	0,8840	0	0,3283	0	0,0151	0	0,0216	0,0
Total	0,3945	0	0,8751	0	0,3301	0	0,0149	0	0,0214	0,0