

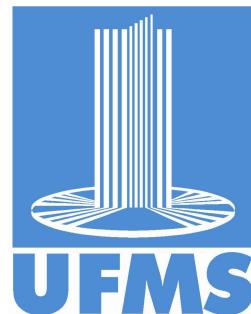
Trabalho de Conclusão de Curso

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 5,2107 segundos, enquanto que em média o teste sem a biblioteca foi executado em 4,6854 segundos, ou seja, a biblioteca elevou o tempo de execução do teste realizado em 0,5253 segundos. No entanto a biblioteca foi exposta a testes de recuperação de falhas, mostrando-se eficaz em todos eles. 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.

Agradecimentos

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	31
3.2	FaultRecovery: Extensão da biblioteca	32
3.2.1	Refatoração e Aperfeiçoamento: Versão 1.0	32
3.2.2	Refatoração e Aperfeiçoamento: Versão 2.0	36
3.3	<i>Classe de Redundância de Dados: TData</i>	37
3.3.1	Classe TData com Tipos Primitivos	38
3.3.2	Um Exemplo Ilusório Para Utilização da Classe TData com Objetos .	38
3.3.3	Classe Carro com Objeto de Pilha	39
3.3.4	Classe Carro com Ponteiro	40
4	Resultados	42
4.1	Desempenho da Biblioteca <i>FaultRecovery</i>	42
4.2	Desempenho e Eficiência da Classe TData	44
4.3	Recuperação de falhas da biblioteca <i>FaultRecovery</i>	45
4.4	Injeção de Falhas com a Biblioteca <i>FaultInjector</i>	46
5	Conclusão	49
5.1	Contribuições deste Trabalho	50
5.2	Dificuldades Encontradas	50
5.3	Trabalhos Futuros	50
Referências Bibliográficas		52

Apêndices	55
------------------	-----------

A Anexos	56
-----------------	-----------

Listas 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	Diagrama de classes da biblioteca <i>FaultRecovery</i>	33
3.3	Fluxograma da biblioteca <i>FaultRecovery</i>	34
3.4	Figura que apresenta a saída com os valores das cópias consistentes para tipos primitivos após a injeção de falhas.	38
3.5	Valores das cópias consistentes após a injeção de falhas com objeto de pilha.	40
3.6	Valores das cópias consistentes após a injeção de falhas com ponteiro.	40
3.7	Figura que apresenta a saída com os valores das cópias consistentes após a atualização do objeto <i>TData</i> do tipo carro e da injeção de falhas.	41
4.1	Tempo de execução da biblioteca <i>FaultRecovery</i>	43
4.2	Tempo de execução da biblioteca <i>FaultRecovery</i> com três tamanhos de vetores diferentes.	43
4.3	Tempo de execução da Classe <i>TData</i>	45
4.4	Teste de redundância de dados da classe <i>TData</i>	46
4.5	Resultados Obtidos da Biblioteca <i>FaultRecovery</i>	47
4.6	Injeção de Falhas na Memória Flash	48

Lista de Tabelas

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

Listas de Quadros

1	Uso de macro como função	32
2	Macro sendo chamada no código	32
3	Exemplo de criação de um estado	35
4	Métodos <i>createRecoveryPoints</i> e <i>run</i>	37

Capítulo 1

Introdução

Atualmente, o ser humano utiliza diversos aparelhos eletrônicos, tais como celulares, tocadores de MP3, televisores, *tablets*, e outros dispositivos usados no auxílio das atividades diárias e na melhoria da qualidade de vida. Em comum, esses aparelhos possuem equipamentos computacionais acoplados e por isso são denominados sistemas embarcados (ou, computadores embutidos). 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 é responsável por ajustar a potência correta, selecionar e medir o tempo de acionamento do forno, e emitir um sinal quando a tarefa é concluída.

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

Segundo Patterson e Hennessy [5], os sistemas embarcados correspondem a maior classe de computadores, pois devido a sua natureza especialista, podem ser encontrados em inúmeras aplicações. Chetan [6] afirma que a quantidade de sistemas embarcados tem crescido nos últimos anos, pois com a expansão da computação ubíqua alguns equipamentos anteriores construídos com pouco ou nenhum recurso computacional tornaram-se mais sofisticados, incorporando algum tipo de sistema embarcado [3]. Entretanto, falhas nesses sistemas podem causar transtornos e prejuízos. Por esse motivo, equipamentos utilizados na indústria aeroespacial e militar [1], dentre outros, são desenvolvidos com estratégias de tolerância a falhas para torná-los sistemas confiáveis. Conforme Johnson [7], tolerância a falhas é a propriedade que permite a um sistema continuar funcionando adequadamente, mesmo que num nível reduzido, após a manifestação de falhas em alguns de seus componentes.

1.1 Justificativa

Embora a maioria dos sistemas embarcados tenha como requisito o baixo custo, a adoção de estratégias para a tolerância a falhas é necessária [8], pois consequências de falhas podem causar danos que variam de 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 [9, 10]. Se o sistema de coleta de dados climáticos falha, o agricultor não fica ciente do momento correto para aplicação dos defensivos, podendo a plantação ser infectada e destruída pela doença.

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

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

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

1.2 Objetivos

1.2.1 Objetivo Geral

Este trabalho teve como objetivo ampliar o injetor de falhas (*FaultInjector*) e a biblioteca de recuperação de falhas (*FaultRecovery*) desenvolvidos por Kruger [3] em sua dissertação de mestrado. O injetor de falhas não injetava falhas na memória *flash* e seu mapeamento de memória era específico para o microcontrolador *mbed* modelo NXP 1768. Nesta nova

versão, a proposta foi ampliar o injetor para funcionar em qualquer modelo da família *mbed* e remover a limitação que existia de inserir falhas apenas na memória SRAM.

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

A classe TData permitiu automatizar a redundância de dados.

1.2.2 Objetivos Específicos

- Desenvolver um mapeamento das regiões de memória, para estender a utilização da biblioteca *FaultInjector* aos demais modelos da família de microcontroladores *mbed* LPC176X;
- Pesquisar as técnicas de tolerância a falhas presentes na literatura, identificar quais podem ser adicionadas a biblioteca de tolerância a falhas atual.
- Melhorar o sistema injetor de falhas para que ele possa injetar falhas em outras regiões de memória ainda não exploradas (*flash*).
- Ampliar a biblioteca de recuperação de falhas *FaultRecovery*, para que seja possível desenvolver uma máquina de estados, na qual cada estado seja implementado independentemente para facilitar a modularização.
- Implementar uma classe (TData) capaz de fazer redundância automática dos dados;
- Realizar testes de desempenho e eficiência após as modificações da biblioteca *FaultRecovery*.
- Realizar testes de desempenho e eficiência após a criação da classe TData.
- Após a modificação do injetor de falhas, verificar se falhas estão sendo injetadas na memória *flash*.

1.3 Organização da Proposta

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

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

2.1 Falha, Erro e Defeito

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

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

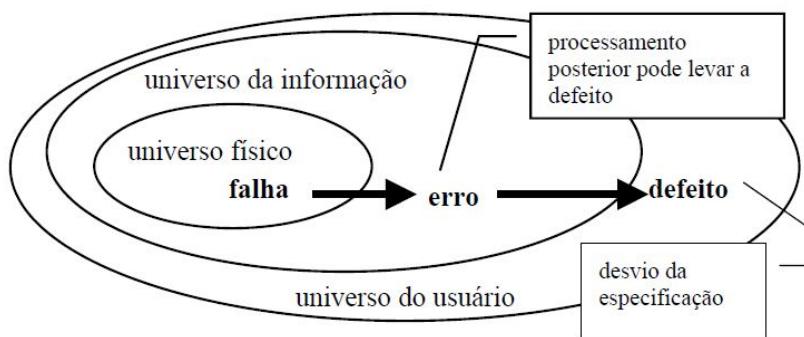


Figura 2.1: Modelo de 3 universos: falha, erro e defeito. Retirado de Weber [13].

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 [14]. 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 [15].

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 [16] e a atividade solar [17]. 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.

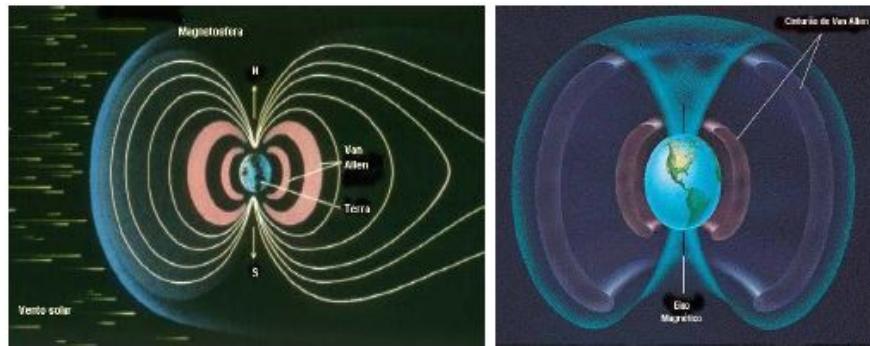


Figura 2.2: Cinturão de Van Allen [18].

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 [16]. Um terceiro cinturão de elétrons foi observado após uma tempestade magnética em 24 de março de 1991 [14].

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 [19]. O período de 11 anos correspondente ao ciclo solar está dividido em aproximadamente 7 anos de alta atividade e 4 anos de baixa atividade [17].

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 [16].

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 [14].

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 [16] 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 Boudonot [17], 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 [20].

2.2.4 Partículas alpha

São compostas por dois nêutrons e dois prótons provenientes de um átomo de hélio duplamente ionizado a partir do decaimento nuclear de isótopos instáveis. No final da década de 70, as partículas alfa emitidas de materiais com traços de Urânia (U) e Tório (Th) foram mostradas como a causa dominante de um SEU numa memória RAM na superfície terrestre [21]. Estas partículas estão presentes nos materiais utilizados para o encapsulamento de circuitos integrados, onde é necessária uma baixa concentração de Urânia (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 cumbo (Pb) e estanho (Sn), os quais são extraídos de minérios que podem conter traços de Urânia (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 [14].

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 [22]. 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 sistemas ou pessoas que dele dependam
Segurança (<i>security</i>)	proteção contra falhas maliciosas, visando privacidade, autenticidade, integridade e irreversibilidade dos dados

Tabela 2.1: Resumo dos atributos de dependabilidade Retirado de Weber [13].

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 [23].

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 [13] 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, disponibilidade, segurança de funcionamento (*safety*), segurança (*security*), mantinabilidade, testabilidade e comprometimento do desempenho (*performability*). Um resumo dos principais atributos é mostrado na Tabela 2.1.

2.4 Tolerância a Falhas

Segundo Avizienis [24] 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 [25]. 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 [13].

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

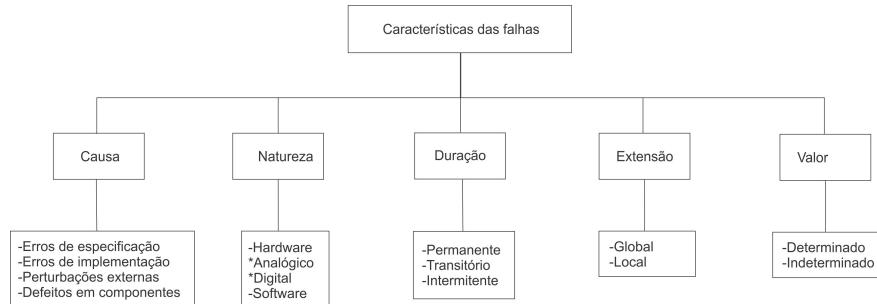


Figura 2.3: Atributos das características das falhas. Retirado de Pego [26].

- **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 [13].

2.5.1 Técnicas de redundância baseadas em software

Segundo Pêgo [26] 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 intruções 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 [26].
- **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 [26].

Weber [13] afirma que a simples replicação de componentes idênticos é uma estratégia de detecção e mascaramento de erros inútil em software. Componentes idênticos de software vão apresentar erros idênticos. Assim não basta copiar um programa e executá-lo em paralelo ou executar o mesmo programa duas vezes em tempos diferentes. Erros de programas idênticos vão apresentar, com grande probabilidade, de forma idêntica para os mesmos dados de entrada. Segundo Brilliant *et al.* 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 [27] 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 [27].

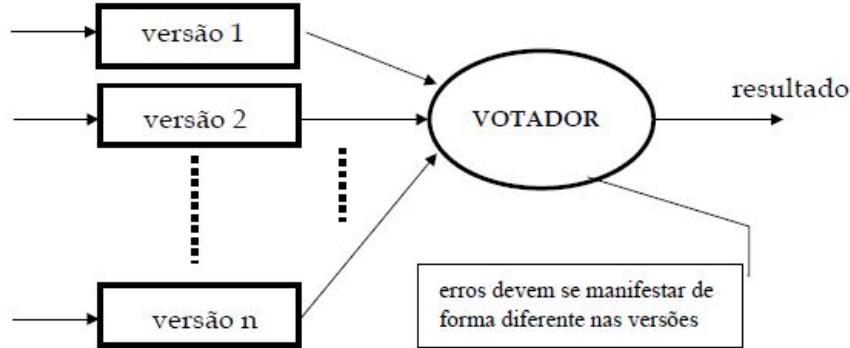


Figura 2.4: Diversidade ou Programação N-Versões. Retirado de Weber [13].

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 [28].

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, 13, 29].

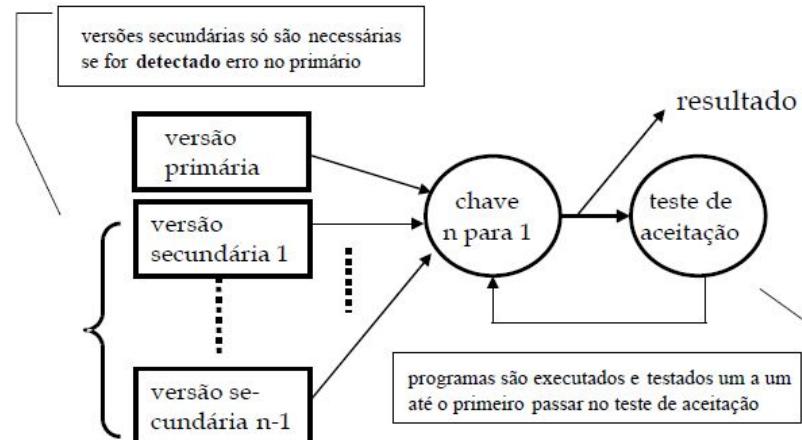


Figura 2.5: Blocos de Recuperação. Retirado de Weber [13].

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* [30] 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 viesssem 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 [30]. Um exemplo de injeção de falhas por *software* é a ferramenta *Ferrari* apresentada por Kanawati *et. al.* [30] 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 [30], 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. 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 foi utilizado neste trabalho [39] para o desenvolvimento da biblioteca *faultRecovery*. *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

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

Foram utilizados exemplos variados para exemplificar as modificações realizadas na *FaultRecovery*, pois se pensou em criar uma biblioteca genérica que possa ser utilizada em diversos casos, que vão desde um sistema embarcado para um microcontrolador *mbed* até um programa para um carrinho seguidor de linha implementado em um *arduino*.

Para a ampliação das bibliotecas citadas, utilizou-se um microcontrolador de prototipagem rápida *mbed*, modelo NXP LPC1768 [41] (o mesmo utilizado por Kruger). Este é projetado para a prototipagem de diversos equipamentos, especialmente aqueles que necessitam de conexão com a internet, portas USB e interfaces variadas para periféricos. Ele possui um núcleo ARM Cortex-M3 32 bits com 96 MHz de *clock*, 64 KB de memória RAM (32 KB disponíveis ao usuário e 32 KB reservados aos controladores internos do dispositivo), uma memória *flash* de 512 KB, portas *built-in Ethernet*, USB Host, CAN (*Controller Area Network*), SPI (*Serial Peripheral Interface*), I2C (*Inter-Integrated Circuit*), ADC (*Analog-to-Digital Converter*), DAC (*Digital-to-Analog Converter*), PWM (*Pulse-Width Modulation*) e outras interfaces de entrada e saída.

O *mbed* também possui um temporizador *watchdog* [42], que é um hardware que reinicia o dispositivo automaticamente em caso de falhas de *hardware* ou de *software*. O temporizador é carregado com um valor inicial, que é decrementado a cada vez que o *watchdog* é executado, sendo que o programa principal executa um *loop* que atravessa a execução de várias funções, que repõe o temporizador a cada vez que passa pelo circuito principal, se por conta de uma falha a reposição não ocorrer e o temporizador zerar, o dispositivo será reiniciado [43].

No sítio oficial da plataforma *mbed* são fornecidas soluções para auxiliar o desenvolvimento. Além da possibilidade de baixar bibliotecas fornecidas pela comunidade, o sítio disponibiliza um *forum* para retirada de dúvidas. A *mbed* também fornece um compilador online, no qual após a compilação do código, um arquivo binário é gerado para ser execu-

tado no microcontrolador. Este compilador online tem a capacidade de compilar códigos para diversos modelos do *mbed* [44].

Além do modelo LPC1768, existem outros pertencentes à família *mbed* NXP LPC17X, são eles: LPC1764, LPC1765, LPC1766 [42]. Embora apresentem arquiteturas compatíveis, o tamanho e o mapa de memória *flash* e *SRAM* varia conforme cada um, distinguindo-se por exemplo, nos modelos LPC1768/66/65 que possuem endereços de regiões de memória *SRAM* idênticos, mas diferentes ao modelo LPC1764, conforme mostra-se na Figura 3.1.

3.1 Injetor de Falhas

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

3.1.1 Mapeamento de Memória

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

Neste primeiro passo, observando o mapa de memória da família *mbed* LPC176X [42] foi possível constatar que alguns modelos possuem regiões de memória idênticas, conforme mostrados na Figura 3.1. Logo, técnicas de orientação a objetos, tais como heranças e classes abstratas poderiam ser utilizadas com o intuito de facilitar o mapeamento das regiões de memória e promover uma interface à biblioteca *FaultInjector*, que precisava injetar falhas nas regiões de memória sem necessariamente conhecer os endereços das regiões de cada dispositivo.

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

A classe abstrata *MemoryMap* representa o mapa de memória de um microcontrolador *mbed* e disponibiliza uma interface para *FaultInjector* injetar falhas. Esta classe é herdada por classes não-abstratas, como a classe *MemoryMap_LPC1764* e *MemoryMap_LPC1768*,

que obrigatoriamente, ficam responsáveis por implementar os métodos abstratos *getUserMemoryRegion()* e *getFlashMemoryRegion()* de *MemoryMap*, descritos a seguir:

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

Por *FaultInjector* ser capaz de identificar o modelo do processador, ela mesmo é responsável por escolher qual das versões de *MemoryMap* irá utilizar (*MemoryMap_LPC1764*, *MemoryMap_LPC1765*, entre outros). Isso tornou a biblioteca flexível, sendo possível injetar falhas nos nas regiões de memória mostradas na Figura 3.1.

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

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

3.1.2 Injeção de Falhas na Memória Flash

Os microcontroladores *mbed* possuem memória *flash*, na qual o código do *firmware* é armazenado [42]. Logo, a alteração de um único bit em um endereço de memória que contenha partes de uma instrução pode resultar em um falha irreversível. No entanto os dados coletados por um sistema também podem ser armazenados na memória *flash*. Considerando-se o sistema embarcado de uma estação meteorológica, esse precisa coletar os dados climáticos e armazená-los na memória não volátil. Estes dados armazenados na *flash* são enviados a um servidor remoto após um período de tempo predefinido. Se um único bit da região de memória na qual os dados coletados estão armazenados for alterado (*bit-flip*), esta alteração poderá provocar uma modificação no valor da informação armazenada que poderia ser, por exemplo, um valor da umidade do ar, que perderia a exatidão após sua alteração, ou seja, o dado coletado enviado para o servidor remoto estaria incorreto, impactando em uma equivocada previsão do tempo.

Conforme explicado na Seção 2.6, um dos aspectos positivos na simulação de falhas por software é a segurança de não danificar o dispositivo. Falhas podem ser inseridas em *bytes* aleatórios na memória *SRAM* simulando o fenômeno *bit-flip*, entretanto, a mesma lógica não pode ser aplicada na memória *flash*, pois por ser protegida não é possível escrever um único endereço de memória, somente a escrita por setores que são no total 29, divididos em duas áreas. A primeira, que abriga os setores do 1 ao 15 contém blocos de 4KB, a segunda, que vai do setor 16 até o 29, é composta por blocos de 32KB para cada setor [42].

A injeção de falhas na *flash* foi realizada mediante a escrita de *bytes* em um setor de memória sorteado aleatoriamente. Foi copiado 256 bytes da memória *SRAM*, que é a quantidade mínima de bytes que podem ser escritos com a biblioteca retirada do repositório de bibliotecas da *mbed* [45], para a *flash* utilizando uma biblioteca que permite fazer operações na memória não volátil.

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

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

Neste trabalho foi realizada a escrita de 256 bytes na memória *flash*, ou seja, foram injetadas 256 bytes de 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, o uso de macros como funções não é uma boa prática de programação [46], pois além de dificultarem o entendimento da estrutura do código, criam outros males que podem resultar em falhas inesperadas. Por exemplo, uma macro que chama uma função f para que se retorne o maior entre os argumentos, se chamada com um dos parâmetros sob incremento, pode gerar erros imprevistos, conforme mostram os Quadros 1 e 2.

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

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

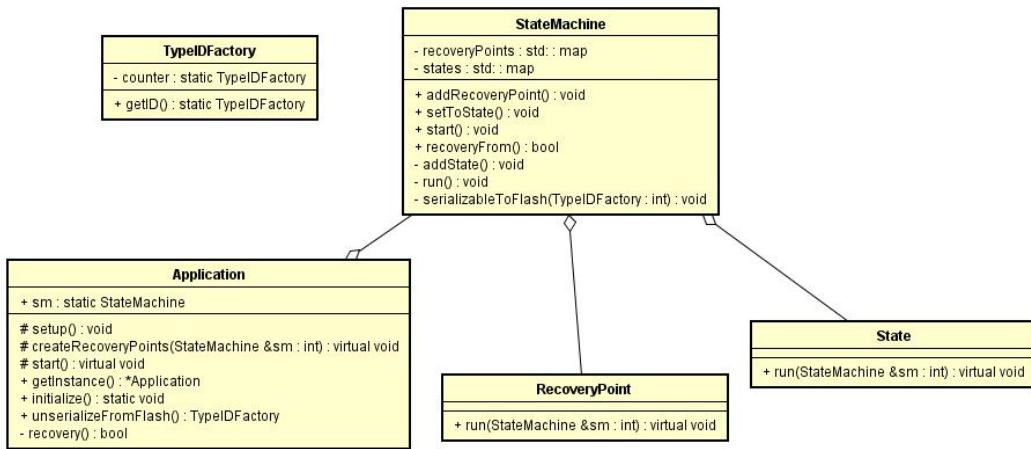
```
1 int a = 5, b = 0;
2 CALL_WITH_MAX(++a, b);
3 CALL_WITH_MAX(++a, b + 10);
```

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

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

3.2.1 Refatoração e Aperfeiçoamento: Versão 1.0

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

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

O uso do padrão de projeto mencionado e a aplicação dos conceitos de orientação a objetos possibilitou a criação de programas modularizáveis, em outras palavras, o código do *firmware* pode ser separado por responsabilidades. Cada estado 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: *TypeIDFactory*, *State* e *StateMachine*, que são descritas abaixo:

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

Conforme mostrado na Figura 3.3 no momento em que o programa recebe um evento responsável pela mudança de estado, o método *setToState* desta classe recebe o valor da variável *current* (determina o estado atual). Se o estado não estiver sido adicionado

no *map* de estados da máquina de estados, o método *addState* será chamado para incluir esse novo estado.

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

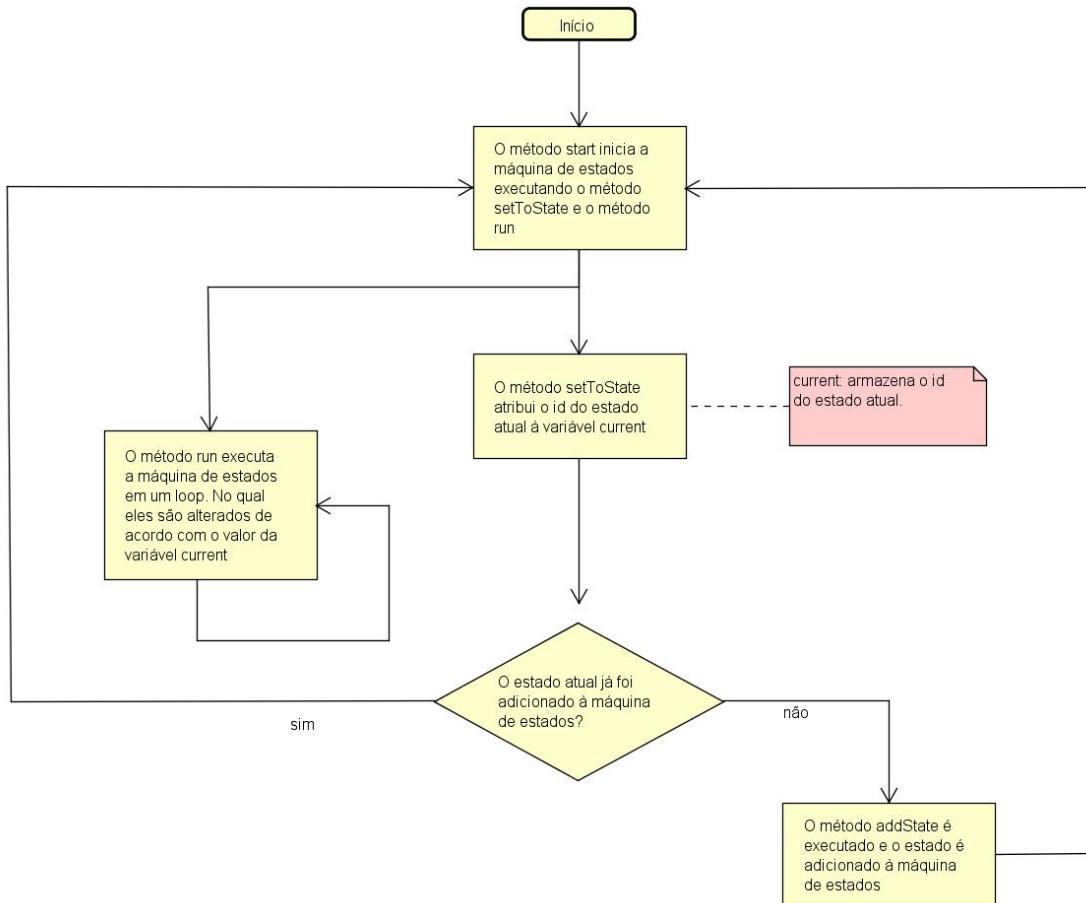


Figura 3.3: Fluxograma da biblioteca *FaultRecovery*.

```

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

Quadro 3: A classe *EsperandoCarroChegar* herda da classe *State* e tem sua rotina implementada no método *run*, ao encerrar a rotina o método *setToState* é invocado, alterando o estado atual. No método *main* um objeto *StateMachine* é instanciado e o método *start* é chamado, iniciando a máquina de estados. Este quadro tem como objetivo demonstrar a utilização da biblioteca, apenas um estado será posto no quadro, os demais serão anexados ao trabalho.

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

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

- **EsperandoCarroChegar** - Este é o estado inicial da máquina de estados. O sensor de presença detecta a chegada do veículo na entrada do estacionamento, o carro se aproxima da entrada, então o estado *EsperandoCarroChegar* detecta a aproximação do veículo e chama o próximo estado.
- **EsperandoApertarBotao** - Neste estado o dispositivo eletrônico emite o *ticket* de estacionamento após o motorista apertar o botão. Após isso, o estado atual é alterado para *EsperandoCarroEntrar*. Outra situação é quando o motorista decide não adentrar ao estacionamento, neste caso o estado é alterado para o anterior (*EsperandoCarroChegar*).
- **EsperandoCarroEntrar** - Neste estado os sensores estão aguardando a entrada do carro ao estacionamento. O sensor externo ao estacionamento detecta duas situações, a primeira é quando o carro se afasta da cancela e não entra no estacionamento, neste caso a máquina de estados retorna ao seu estado inicial. O segundo é quando o carro adentra ao estacionamento, neste caso o sensor externo identifica a entrada do carro, o interno ao estacionamento detecta o carro se afastando da cancela, com o seu afastamento o sistema altera o estado para *FeharCancela*.
- **FeharCancela** - Neste estado o carro se afasta da entrada do estacionamento, o sensor externo a esse detecta o afastamento do carro e a cancela é fechada. Ao fechar

a cancela o estado FecharCancela, por ser o último estado da máquina de estados, irá chamar o estado inicial, para então reiniciar o ciclo de execução dos estados.

3.2.2 Refatoração e Aperfeiçoamento: Versão 2.0

Na subseção 3.2.1 que trata da versão 1.0 da biblioteca, mostrou-se como implementar os estados de uma máquina de estados. Esta subseção tratará da criação dos pontos de recuperação, como inicializar a máquina de estados e quais as classes adicionadas na biblioteca e suas responsabilidades. Nesta etapa de desenvolvimento, focou-se na tolerância a falhas. A versão 2.0 da biblioteca *FaultRecovery* utiliza a máquina de estados desenvolvida na versão 1.0, no entanto foi adicionada a funcionalidade que possibilita a criação de pontos de recuperação de falhas. Se o microcontrolador 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 *watchdog* 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, sendo que os métodos *createRecoveryPoints* 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 robô seguidor de linha e cada aluno ficou responsável por implementar um estado da máquina de estados do carrinho.

O segundo é o método *createRecoveryPoints*, que poderá ou não ser implementado, pois por padrão esse método não adiciona um ponto de recuperação à máquina de estados, sendo assim se usuário necessitar que o seu *firmware* tenha um ponto de recuperação ele deverá implementar o método *createRecoveryPoints*. Nesse método os pontos de recuperação são adicionados à máquina de estados. Para implementar um ponto de recuperação deve-se criar uma classe que herde de *RecoveryPoints*.

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

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

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

```

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

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

3.3 Classe de Redundância de Dados: *TData*

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

é utilizando tipos primitivos e a segunda objetos.

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

3.3.1 Classe *TData* com Tipos Primitivos

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

```

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

```

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

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

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

As informações disponibilizadas pelo veículo são navegação, diagnósticos do funcionamento do próprio veículo, dentre outras informações. Este por sua vez sofre um acidente e o sistema se encarrega de acionar outro sistema, ou seja, o sistema embarcado instalado

no carro se comunica com outro remotamente. Este, por sua vez, receberia todos os dados relativos ao paciente, inclusive sua localização para um possível deslocamento de ambulâncias [47]. 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. Mas por que os operadores de igualdade precisam ser implementados? Porque o compilador não é capaz de definir o comportamento de uma definição de igualdade em um objeto. Cada objeto define a igualdade de um jeito diferente, a classe *string* por exemplo, define que igualdade são objetos com o mesmo conteúdo, ainda que sejam objetos distintos, ou seja, em endereços de memória diferentes.

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

3.3.3 Classe Carro com Objeto de Pilha

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

Para este exemplo utilizou-se as classes Carro e Emergencia mencionadas nas seções anteriores e o método *injectFault* utilizado para simular o fenômeno *bit-flip*. A injeção de falhas foi realizada com métodos temporários implementados dentro da classe *TData* alterando os valores do telefone do hospital pertencente a classe Emergencia e da localização do carro pertencente a classe Carro.

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

```

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

```

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

3.3.4 Classe Carro com Ponteiro

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

```

Problems Tasks Console Properties Debug Disassembly Search
<terminated> TripledData - carro com ponteiro.exe [C/C++ Application] C:\workspaces\WorkspaceTCC\TripledData - carro com ponteiro\Debug\TripledData - carro com ponteiro -
Localização original: 100
Telefone do Hospital original: 123456
Valores armazenados nas 3 cópias após a injeção de falha
Localização 1: 100
Localização 2: 100
Localização 3: 100
Telefone do Hospital 1: 123456
Telefone do Hospital 2: 123456
Telefone do Hospital 3: 123456

```

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

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

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

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

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

Capítulo 4

Resultados

Neste capítulo são apresentados os resultados dos testes realizados.

4.1 Desempenho da Biblioteca *FaultRecovery*

Esta seção tem como objetivo demonstrar o quanto a biblioteca *FaultRecovery* pode afetar no tempo de execução de um *firmware*. Para realizar os testes, utilizou-se cinco algoritmos de ordenação *bubble sort*, *insertion sort*, *merge sort* e *comb sort* [48, 49], a execução dos cinco algoritmos forma um ciclo de teste. Ao todo cada ciclo foi executado cem vezes. Os algoritmos foram implementados para serem executados no microcontrolador *mbed* modelo 1768. Para cada ciclo de teste, utilizou-se um vetor de 4096 elementos, totalizando 4kB de memória. Tentou-se aumentar a quantidade de elementos do vetor, entretanto quando se tentou alocar um espaço de memória maior que 4KB, o *firmware* teve sua execução interrompida no segundo ciclo de teste, em outros no primeiro ciclo.

O primeiro teste tem como objetivo medir o tempo de execução de cada algoritmo de ordenação, simulando um *firmware* implementado por um usuário comum, que não 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. O tempo de cada algoritmo foi cronometrado, desde o início de sua execução até o fim dela, sendo que o tempo de desordenação do vetor foi desprezado.

O segundo teste tem como objetivo medir o tempo de execução dos algoritmos de ordenação, simulando um usuário que utilizou 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 o fim dele. Tanto para o primeiro quanto para o segundo teste foi utilizada a mesma quantidade de elementos do vetor (4096), a mesma sequência de execução (*bubble*, *insertion*, *selection*, *merge* e *comb*) e o tempo de desordenação do vetor foi desprezado. Ao final das cem execuções do primeiro e do segundo teste, as médias de tempo de execução para cada algoritmo e para cada ciclo de teste foram calculadas. O tempo de execução da biblioteca *FaultRecovery* também foi calculado, obtendo-se o resultado mostrado na figura 4.1.

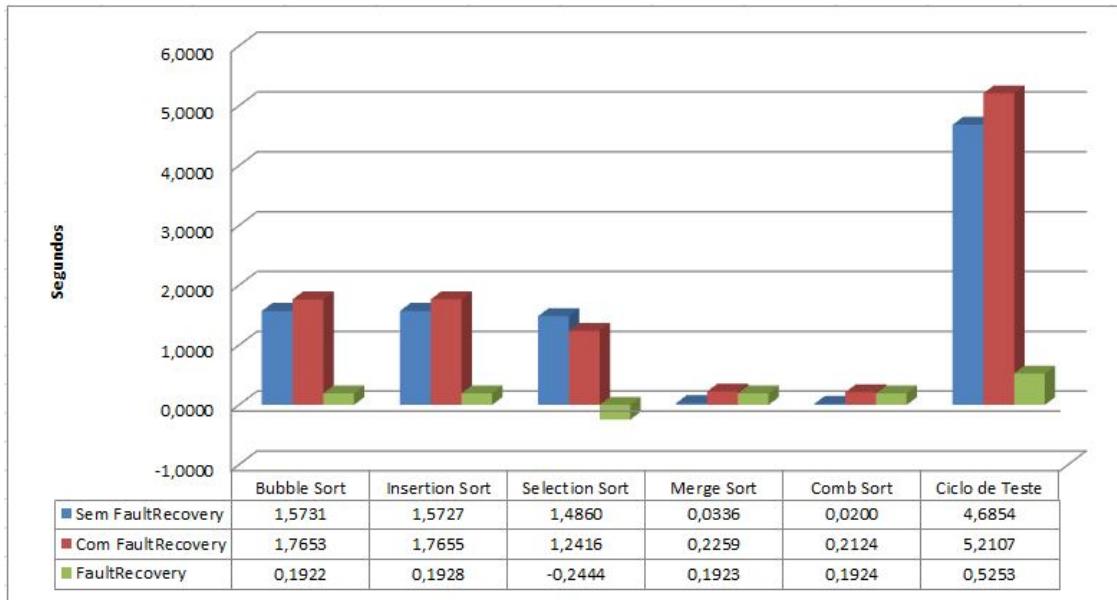


Figura 4.1: Nesta figura são mostrados os resultado dos testes de medição do tempo de execução dos algoritmos de ordenação, sem a biblioteca e com a biblioteca. Percebe-se que com a utilização da biblioteca o tempo de execução dos algoritmos aumentou em média 0,5253 segundos ou 525,3 milisegundos.

Dentre os tempos de execução mostrados na figura 4.1, obteve-se um resultado inesperado, o esperado seria que o tempo de execução dos algoritmos com a biblioteca *FaultRecovery* fosse maior em relação ao primeiro teste, entretanto o tempo de execução do *insertion sort* foi menor com a biblioteca. Diante disso, verificou-se a implementação dos dois testes, sendo que não foi constatada alguma diferença entre os códigos do primeiro e do segundo teste que pudessem causar este resultado. Após a análise realizou-se um terceiro teste que foi executado cem vezes, utilizou-se um vetor de 3KB, menor que o utilizado nos testes anteriores, para verificar se o comportamento anômalo persistiria. Verificou-se que o comportamento persistiu, sendo que não se disponibiliza de tempo hábil para uma análise e pesquisa aprofundada sobre esse comportamento. Portanto se presume que esses casos são factíveis de ocorrer, provavelmente se deve ao resultado do código gerado pelo compilador, já que o microcontrolador utilizado nos testes não possui sistema operacional e nenhuma outra aplicação rodando simultaneamente. Os resultados do terceiro teste são mostrados na figura 4.2.

	Bubble Sort	Insertion Sort	Selection Sort	Merge Sort	Comb Sort
Com FaultRecovery(vetor 4KB)	0,1922	0,1928	-0,2444	0,1923	0,1924
Com FaultRecovery(vetor 3KB)	0,1966	0,1964	-0,0490	0,1970	0,1970
Com FaultRecovery(vetor 2KB)	0,2029	0,2023	0,0929	0,2020	0,2027

Figura 4.2: Nesta figura é mostrado o tempo de execução da biblioteca *FaultRecovery* com três vetores de tamanhos diferentes. Percebe-se que o tempo de execução da biblioteca no algoritmo *insertion sort* é negativo no primeiro e segundo teste, contudo no teste com o vetor de tamanho 2KB, obteve-se um tempo de execução positivo. Portanto, pode-se concluir que quando a quantidade dados a serem processadas for grande, provavelmente o compilador optimiza o código compilado.

4.2 Desempenho e Eficiência da Classe TData

Para testar a redundância de dados da classe TData e se a sua utilização pode causar alguma 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 4096 para 1024 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 a quantidade de memória alocada seria de aproximadamente 16KB, impossibilitando a realização dos testes. O teste que não utilizada 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 4.3. 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 média 0,2658 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 a falhar em algum ciclo de teste e foi o que aconteceu, na figura 4.4 têm-se o resultado da execução do primeiro e do segundo teste. Foram determinados três 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 4.4 é possível visualizar que mesmo após as injeções de falhas, a classe TData se mostrou eficaz garantindo a consistência dos dados do vetor até o fim dos cem ciclos de teste.

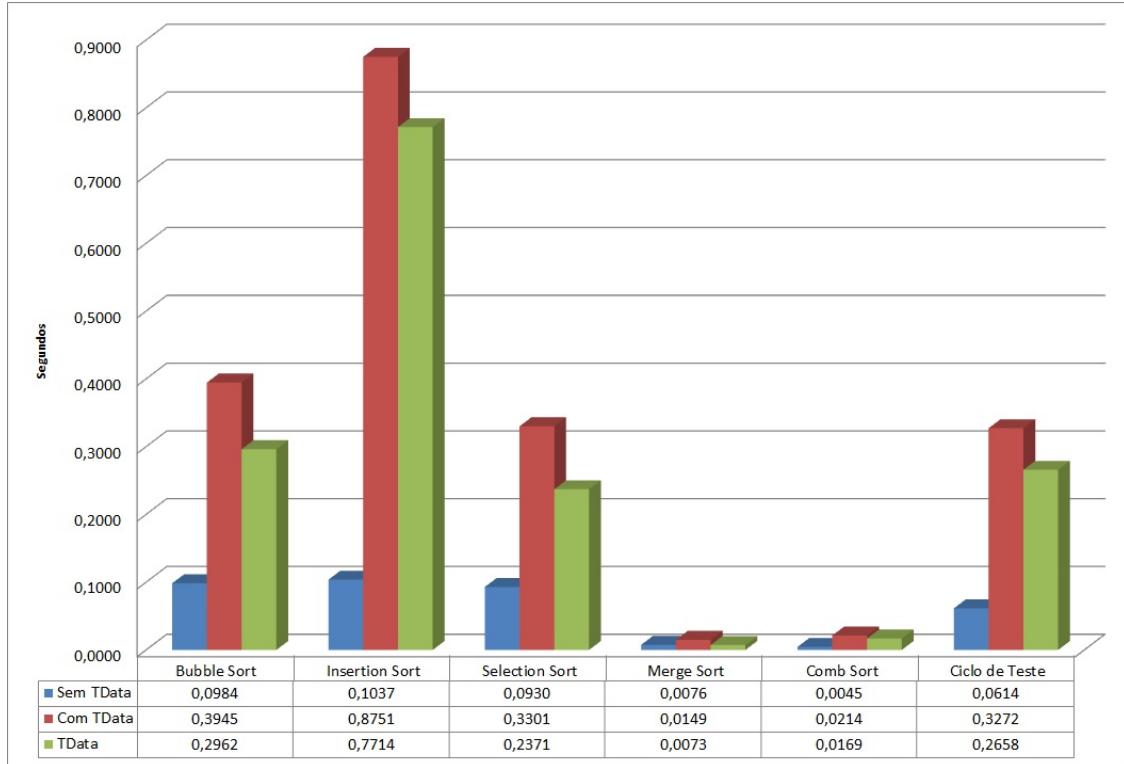


Figura 4.3: O tempo de execução dos algoritmos sem a classe *TData* foi relativamente baixo, sendo que o maior tempo atingiu 0,1037 segundos ou 103,7 milisegundos para ordenar um vetor de 1024 elementos. Já o tempo dos algoritmos com a Classe *TData* foi um pouco maior, sendo o tempo de execução do algoritmo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

4.3 Recuperação de falhas da biblioteca *FaultRecovery*

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

Para que isso não venha a ocorrer, a estação meteorológica poderia ser implementada utilizando a biblioteca *FaultRecovery* para que pontos de recuperação de falhas pudessem ser criados, para assim que o microcontrolador reinicializa-se por conta de alguma falha, algum ponto de recuperação predefinido pudesse ser executado. Porém neste trabalho não se implementou uma estação meteorológica para simular esse acontecimento. Entretanto

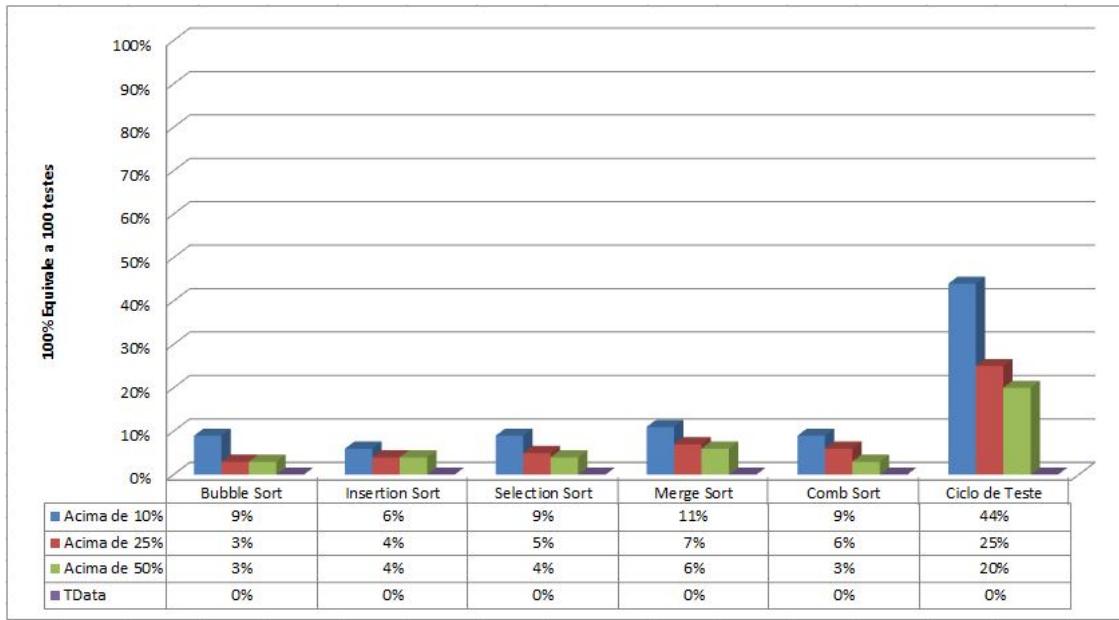


Figura 4.4: Nesta figura para as falhas detectadas acima de 10%, 25% e 50% são do teste que não utiliza redundância de dados, por exemplo, para as falhas acima de 10%, aproximadamente 44% dos ciclos de testes falharam. No entanto para o teste que utilizou a redundância de dados disponibilizada pela TData 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.

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

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 4.6 é exibido um teste em que foram injetadas 256 bytes de falhas no setor 25, que foi sorteado aleatoriamente pelo injetor de falhas. O endereço de memória do setor sorteado em hexadecimal inicia em 0x00058000 e termina em 0x0005FFFF. Pode-se perceber que os bytes dos endereços de memória do setor 25 (0x00058000 até 0x000580F0) foram alterados. Portanto agora a biblioteca *FaultInjector* pode injetar falhas na memória flash do microcontrolador *mbed* modelo 1768.

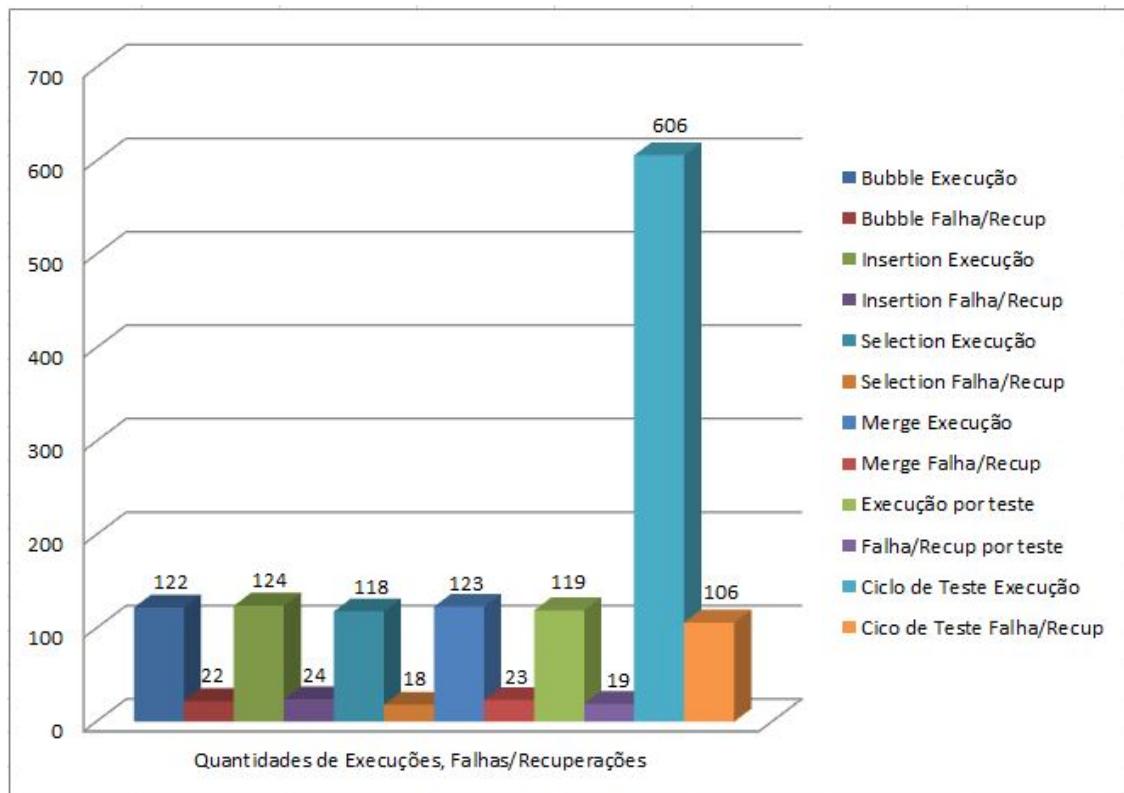


Figura 4.5: Esta figura mostra a quantidade de execuções e falhas de cada algoritmo, inclusive de cada ciclo de teste. Por exemplo, a primeira coluna, da esquerda para a direita, representa o número de execuções do algoritmo *bubble sort*, pode-se perceber que embora tenham sido executados cem ciclos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

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

Capítulo 5

Conclusão

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

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

No entanto, a biblioteca adiciona um custo de desempenho no tempo de processamento, uma vez que toda a operação de leitura e escrita em uma variável, feita pela classe TData, é custosa devido a execução de um esquema de votação usado para definir o valor correto permanecente em todas as cópias. Porém, isso já era esperado, ainda sim o uso da biblioteca se torna mais vantajoso pelo fato de a maioria das aplicações embarcadas não terem como fator principal o tempo de execução, exceto algumas aplicações de tempo real. Mas nestes casos são utilizados microcontroladores com um maior poder de processamento.

A biblioteca *FaultInjector* também foi modificada neste trabalho, agora é possível injetar falhas na memória flash. No teste realizado, mostrou-se os bits armazenados em determinado

setor da memória flash antes e depois da injeção de falhas. Com isso foi possível visualizar os bits sendo alterados. Além disso, também foi incluído um mapeamento de memória que possibilita a utilização do injetor de falhas em modelos pertencentes a família *mbed* LPC176X. Porém só foi possível testar o mapeamento de memória no modelo 1768, pois era o único microcontrolador disponível para este trabalho. O mapeamento funcionou para o modelo disponível e foi possível injetar falhas nos endereços de memória disponíveis no mapeamento. Tanto a biblioteca *FaultRecovery*, quanto a *FaultInjector* estão disponíveis no *github*, no endereço <https://github.com/cleitonalmeida1>.

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 Contribuiões 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*. [ERRADO!!! ELAS ESTÃO SENDO SALVAS NA REGIÃO X (RESERVADA AO USUÁRIO), E FUTURAMENTE PODERÁ SER SALVA NA REGIÃO Y, QUE É DESTINADA AOS

PERIFÉRICOS INTERNOS DO MICROCONTROLADOR]

- 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] TADEU, T. G.; GALVÃO, L. E. M. Um estudo exploratório sobre sistemas operacionais embarcados. Technical Report 1, Instituto de Ciência e Tecnologia da Universidade Federal de São Paulo - UNIFESP, 2014.
- [9] REIS, E. *Previsão de doenças de plantas*. UPF EDITORA, 2004.
- [10] IAIONE, F.; LIMA, D. G.; GASSEN, F. R. Equipamento para coleta de dados e previsão de doenças na lavoura, 1999.
- [11] HSUEH, M.-C.; TSAI, T. K.; IYER, R. K. Fault injection techniques and tools. *Computer*, Los Alamitos, CA, USA, v. 30, n. 4, p. 75–82, apr 1997.
- [12] ZIEGLER, J. Terrestrial cosmic rays. *IBM Journal of Research and Development*, v. 40, n. 1, p. 19–39, Jan 1996.
- [13] 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.

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

- [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] SENGER, VINICIUS, X. K. 33 design-patterns aplicados com java. Publicação online, 2009. Disponível em: <<http://pt.slideshare.net/vsenger/33-design-patterns-com-java>> Acesso em: 14-Abril-2016.
- [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/cookbook/WatchDog-Timer>> Acesso em: 29/08/2016.
- [44]MBED, A. Compilador, 2016. Disponível em: <<https://developer.mbed.org/>> Acesso em: 23/03/2016.
- [45]MBED, A. Iap internal flash write, 2010. Disponível em: <<https://developer.mbed.org/users/okano/code/>> Acesso em: 23/03/2016.
- [46]MEYERS, S. *C++ efcaz 55 maneiras de aprimorar seus programas e projetos*. Porto Alegre: BOOKMAN COMPANHIA EDITORA, 2011.

- [47] LEITHARDT, V. R. Q. *Modelo gerenciador de descoberta de serviços pervasivos ciente de contexto.* 2008. Dissertação de Mestrado - PUCRS, Porto Alegre, 2008.
- [48] HERNANDE, D. N. Estrutura de dados, 2011. Disponível em: <<http://www.ft.unicamp.br/liag/siteEd/>> Acesso em: 03/08/2016.
- [49] 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 com um vetor de 4KB

Teste	Bubble Sort	Insertion Sort	Selection Sort	Merge Sort	Comb Sort	Ciclo de Teste
1	1,5731	1,5727	1,4860	0,0336	0,0200	4,6854
2	1,5731	1,5727	1,4860	0,0336	0,0200	4,6854
3	1,5731	1,5727	1,4860	0,0336	0,0200	4,6854
4	1,5731	1,5727	1,4860	0,0336	0,0200	4,6854
5	1,5731	1,5727	1,4860	0,0336	0,0200	4,6854
6	1,5731	1,5727	1,4860	0,0336	0,0200	4,6854
7	1,5731	1,5727	1,4860	0,0336	0,0200	4,6854
8	1,5731	1,5727	1,4860	0,0336	0,0200	4,6854
9	1,5731	1,5727	1,4860	0,0336	0,0200	4,6854
10	1,5731	1,5727	1,4860	0,0336	0,0200	4,6854
11	1,5731	1,5727	1,4860	0,0336	0,0200	4,6854
12	1,5731	1,5727	1,4860	0,0336	0,0200	4,6854
13	1,5731	1,5727	1,4860	0,0336	0,0200	4,6854
14	1,5731	1,5727	1,4860	0,0336	0,0200	4,6854
15	1,5731	1,5727	1,4860	0,0336	0,0200	4,6854
16	1,5731	1,5727	1,4860	0,0336	0,0200	4,6854
17	1,5731	1,5727	1,4860	0,0336	0,0200	4,6854
18	1,5731	1,5727	1,4860	0,0336	0,0200	4,6854
19	1,5731	1,5727	1,4860	0,0336	0,0200	4,6854
20	1,5731	1,5727	1,4860	0,0336	0,0200	4,6854
21	1,5731	1,5727	1,4860	0,0336	0,0200	4,6854
22	1,5731	1,5727	1,4860	0,0336	0,0200	4,6854
23	1,5731	1,5727	1,4860	0,0336	0,0200	4,6854
24	1,5731	1,5727	1,4860	0,0336	0,0200	4,6854
25	1,5731	1,5727	1,4860	0,0336	0,0200	4,6854
26	1,5731	1,5727	1,4860	0,0336	0,0200	4,6854
27	1,5731	1,5727	1,4860	0,0336	0,0200	4,6854
28	1,5731	1,5727	1,4860	0,0336	0,0200	4,6854
29	1,5731	1,5727	1,4860	0,0336	0,0200	4,6854
30	1,5731	1,5727	1,4860	0,0336	0,0200	4,6854
31	1,5731	1,5727	1,4860	0,0336	0,0200	4,6854
32	1,5731	1,5727	1,4860	0,0336	0,0200	4,6854
33	1,5731	1,5727	1,4860	0,0336	0,0200	4,6854
34	1,5731	1,5727	1,4860	0,0336	0,0200	4,6854
35	1,5731	1,5727	1,4860	0,0336	0,0200	4,6854
36	1,5731	1,5727	1,4860	0,0336	0,0200	4,6854
37	1,5731	1,5727	1,4860	0,0336	0,0200	4,6854
38	1,5731	1,5727	1,4860	0,0336	0,0200	4,6854
39	1,5731	1,5727	1,4860	0,0336	0,0200	4,6854
40	1,5731	1,5727	1,4860	0,0336	0,0200	4,6854
41	1,5731	1,5727	1,4860	0,0336	0,0200	4,6854
42	1,5731	1,5727	1,4860	0,0336	0,0200	4,6854
43	1,5731	1,5727	1,4860	0,0336	0,0200	4,6854
44	1,5731	1,5727	1,4860	0,0336	0,0200	4,6854
45	1,5731	1,5727	1,4860	0,0336	0,0200	4,6854

96	1,5731	1,5727	1,4860	0,0336	0,0200	4,6854
97	1,5731	1,5727	1,4860	0,0336	0,0200	4,6854
98	1,5731	1,5727	1,4860	0,0336	0,0200	4,6854
99	1,5731	1,5727	1,4860	0,0336	0,0200	4,6854
100	1,5731	1,5727	1,4860	0,0336	0,0200	4,6854
Média	1,5731	1,5727	1,4860	0,0336	0,0200	4,6854

Desempenho do teste com a biblioteca FaultRecovery e com um vetor de 4KB

Teste	BubbleSort	InsertionSort	SelectionSort	MergeSort	CombSort	Média do ciclo de teste
1	1,7739	1,7752	1,2435	0,2315	0,2107	5,2348
2	1,7669	1,7639	1,2401	0,2215	0,2091	5,2015
3	1,7590	1,7620	1,2418	0,2253	0,2126	5,2007
4	1,7606	1,7615	1,2363	0,2185	0,2095	5,1865
5	1,7667	1,7614	1,2417	0,2273	0,2161	5,2133
6	1,7701	1,7682	1,2449	0,2302	0,2179	5,2312
7	1,7757	1,7760	1,2556	0,2363	0,2236	5,2672
8	1,7737	1,7864	1,2518	0,2363	0,2287	5,2769
9	1,7762	1,7778	1,2535	0,2397	0,2326	5,2798
10	1,7820	1,7825	1,2618	0,2398	0,2265	5,2926
11	1,7814	1,7773	1,2662	0,2480	0,2328	5,3057
12	1,7791	1,7839	1,2629	0,2488	0,2308	5,3055
13	1,7814	1,7868	1,2650	0,2370	0,2245	5,2947
14	1,7755	1,7805	1,2534	0,2395	0,2288	5,2776
15	1,7745	1,7724	1,2535	0,2353	0,2181	5,2539
16	1,7811	1,7824	1,2575	0,2332	0,2137	5,2679
17	1,7765	1,7719	1,2560	0,2316	0,2200	5,2560
18	1,7677	1,7685	1,2423	0,2350	0,2142	5,2277
19	1,7677	1,7643	1,2424	0,2337	0,2132	5,2213
20	1,7678	1,7680	1,2452	0,2316	0,2223	5,2349
21	1,7663	1,7671	1,2467	0,2281	0,2161	5,2243
22	1,7606	1,7601	1,2347	0,2230	0,2064	5,1848
23	1,7574	1,7589	1,2365	0,2196	0,2034	5,1759
24	1,7585	1,7662	1,2368	0,2202	0,2085	5,1902
25	1,7635	1,7584	1,2371	0,2204	0,2061	5,1855
26	1,7594	1,7604	1,2371	0,2202	0,2112	5,1884
27	1,7616	1,7595	1,2386	0,2274	0,2105	5,1975
28	1,7679	1,7632	1,2412	0,2267	0,2106	5,2096
29	1,7627	1,7648	1,2403	0,2220	0,2137	5,2035
30	1,7752	1,7641	1,2288	0,2230	0,2238	5,2149
31	1,7682	1,7671	1,2502	0,2289	0,2194	5,2338
32	1,7676	1,7644	1,2444	0,2317	0,2131	5,2212
33	1,7646	1,7755	1,2462	0,2277	0,2089	5,2229
34	1,7683	1,7629	1,2401	0,2253	0,2163	5,2129
35	1,7633	1,7674	1,2505	0,2228	0,2120	5,2160
36	1,7636	1,7649	1,2368	0,2256	0,2140	5,2049
37	1,7668	1,7644	1,2396	0,2225	0,2085	5,2018
38	1,7604	1,7637	1,2391	0,2259	0,2122	5,2013
39	1,7667	1,7651	1,2393	0,2291	0,2075	5,2078
40	1,7604	1,7570	1,2348	0,2213	0,2088	5,1822
41	1,7635	1,7665	1,2404	0,2256	0,2083	5,2043
42	1,7613	1,7633	1,2416	0,2237	0,2122	5,2022
43	1,7599	1,7650	1,2421	0,2273	0,2216	5,2160
44	1,7673	1,7665	1,2446	0,2277	0,2118	5,2178
45	1,7665	1,7625	1,2402	0,2245	0,2083	5,2019

46	1,7621	1,7657	1,2370	0,2222	0,2074	5,1943
47	1,7618	1,7630	1,2385	0,2269	0,2084	5,1985
48	1,7735	1,7675	1,2388	0,2239	0,2106	5,2144
49	1,7669	1,7684	1,2435	0,2285	0,2132	5,2205
50	1,7615	1,7625	1,2415	0,2237	0,2084	5,1977
51	1,7579	1,7650	1,2321	0,2207	0,2082	5,1839
52	1,7622	1,7573	1,2331	0,2228	0,2056	5,1811
53	1,7628	1,7616	1,2326	0,2165	0,2051	5,1787
54	1,7608	1,7598	1,2373	0,2209	0,2086	5,1874
55	1,7619	1,7605	1,2362	0,2205	0,2082	5,1873
56	1,7672	1,7671	1,2459	0,2297	0,2154	5,2253
57	1,7739	1,7701	1,2480	0,2299	0,2180	5,2400
58	1,7697	1,7652	1,2411	0,2286	0,2099	5,2144
59	1,7616	1,7620	1,2409	0,2261	0,2205	5,2111
60	1,7667	1,7724	1,2427	0,2261	0,2158	5,2236
61	1,7651	1,7642	1,2433	0,2253	0,2230	5,2209
62	1,7684	1,7711	1,2364	0,2254	0,2092	5,2105
63	1,7634	1,7606	1,2371	0,2291	0,2162	5,2063
64	1,7669	1,7633	1,2484	0,2298	0,2113	5,2197
65	1,7613	1,7714	1,2392	0,2224	0,2069	5,2011
66	1,7616	1,7734	1,2358	0,2194	0,2072	5,1974
67	1,7615	1,7587	1,2397	0,2214	0,2100	5,1914
68	1,7686	1,7601	1,2377	0,2268	0,2095	5,2028
69	1,7614	1,7628	1,2391	0,2261	0,2120	5,2015
70	1,7660	1,7736	1,2445	0,2211	0,2104	5,2156
71	1,7652	1,7666	1,2392	0,2225	0,2074	5,2010
72	1,7648	1,7668	1,2424	0,2201	0,2083	5,2023
73	1,7622	1,7644	1,2418	0,2216	0,2066	5,1965
74	1,7691	1,7652	1,2419	0,2305	0,2112	5,2180
75	1,7633	1,7685	1,2423	0,2296	0,2139	5,2177
76	1,7692	1,7679	1,2426	0,2318	0,2119	5,2234
77	1,7624	1,7644	1,2377	0,2211	0,2184	5,2039
78	1,7669	1,7698	1,2394	0,2255	0,2087	5,2105
79	1,7625	1,7584	1,2374	0,2176	0,2038	5,1798
80	1,7607	1,7579	1,2347	0,2166	0,2030	5,1727
81	1,7555	1,7549	1,2337	0,2206	0,2031	5,1678
82	1,7591	1,7576	1,2341	0,2186	0,2031	5,1725
83	1,7577	1,7579	1,2342	0,2177	0,2046	5,1720
84	1,7589	1,7589	1,2332	0,2193	0,2099	5,1801
85	1,7558	1,7575	1,2385	0,2217	0,2052	5,1787
86	1,7626	1,7604	1,2386	0,2233	0,2143	5,1992
87	1,7605	1,7593	1,2400	0,2204	0,2102	5,1903
88	1,7602	1,7591	1,2379	0,2244	0,2093	5,1910
89	1,7609	1,7619	1,2380	0,2254	0,2082	5,1943
90	1,7604	1,7592	1,2363	0,2229	0,2100	5,1888
91	1,7629	1,7651	1,2370	0,2190	0,2071	5,1911
92	1,7613	1,7617	1,2381	0,2244	0,2112	5,1966
93	1,7629	1,7593	1,2373	0,2235	0,2067	5,1898
94	1,7582	1,7622	1,2460	0,2248	0,2096	5,2009
95	1,7589	1,7601	1,2346	0,2188	0,2144	5,1867

96	1,7665	1,7625	1,2372	0,2202	0,2076	5,1939
97	1,7617	1,7599	1,2401	0,2192	0,2083	5,1891
98	1,7649	1,7646	1,2430	0,2303	0,2110	5,2137
99	1,7594	1,7569	1,2341	0,2200	0,2061	5,1765
100	1,7599	1,7639	1,2370	0,2235	0,2114	5,1958
Média	1,7653	1,7655	1,2416	0,2259	0,2124	5,2107

**Teste de recuperação de falhas com a biblioteca FaultRecovery e com
um vetor de 4KB**

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 com a biblioteca FaultRecovery e com um vetor de 4KB

Teste	Merge Sort		Comb Sort		Ciclo de Teste	
	Execução	Falha/Recup	Execução	Falha/Recup	Execução	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 com um vetor de 1024 KB

Teste	Bubble Sort		Insertion Sort		Selection 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 e com um vetor de 1024 KB

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

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

```

1 #include "StateMachine.h"
2
3 TypeIDTypeIDFactory::counter = 0;
4
5 Application *Application::instance = NULL;

```

```

1 #ifndef STATEMACHINELIB_STATEMACHINE_H
2 #define STATEMACHINELIB_STATEMACHINE_H
3
4 #include "mbed.h"
5 #include "ConfigFile.h"
6 #include "Logger.h"
7 #include <map>
8 #include <fstream>
9 #include <iostream>
10
11 using namespace std;
12
13 /*
-----*/
14 * Class Declarations
15 *
-----*/
16 classTypeIDFactory;
17 class State;
18 class RecoveryPoint;
19 class StateMachine;
20 class Application;
21
22 /*
-----*/
23 * Type Definitions
24 *
-----*/
25 typedef intTypeID;
26
27 static const char *RECOVERY_FILE = "/local/state.txt";
28
29 /*
-----*/
30 * Class Definitions

```

```
31 *-----*
32 classTypeIDFactory {
33 private:
34 staticTypeID counter;
35
36TypeIDFactory() {
37 }
38
39~TypeIDFactory() {
40 }
41
42public:
43
44template<typename T>
45staticTypeID getID() {
46 staticTypeID id = counter++;
47 return id;
48 }
49
50};
51
52class State {
53protected:
54
55State() {
56 }
57
58State(State &orig) {
59 }
60
61virtual ~State() {
62 }
63
64public:
65
66virtual double getTimeLimit() {
67 return 0.0;
68 }
69
70virtual void run(StateMachine &sm) = 0;
71};
72
73class RecoveryPoint {
74protected:
75}
```

```
76 RecoveryPoint() {
77 }
78
79 RecoveryPoint(RecoveryPoint &orig) {
80 }
81
82 virtual ~RecoveryPoint() {
83 }
84
85 public:
86
87 virtual void run(StateMachine &sm) = 0;
88 };
89
90 class StateMachine {
91 public:
92
93 StateMachine() :
94 configFile(RECOVERY_FILE), logger("/local/teste.txt") {
95 }
96
97 StateMachine(StateMachine &orig) :
98 recoveryPoints(orig.recoveryPoints), states(orig.states), current(
99 orig.current), configFile(RECOVERY_FILE), logger("/local/teste.txt") {
100 }
101
102 virtual ~StateMachine() {
103 }
104
105 template<class R, class S>
106 void addRecoveryPoint() {
107
108 TypeID id =TypeIDFactory::getID<S>();
109 recoveryPoints[id] = new R;
110 }
111
112 template<class S>
113 void setToState() {
114 StateMachine::current =TypeIDFactory::getID<S>();
115 serializeToFlash(current);
116 if (states[current] == NULL) {
117 addState<S>();
118 }
119 }
120
121 template<class S>
122 void start() {
```

```
123 setToState<S>();
124 run();
125 }
126
127 bool recoveryFrom(TypeID id) {
128 RecoveryPoint *recoveryPoint = recoveryPoints[id];
129 cout << "idRecovery = " << endl;
130 if (recoveryPoint == NULL) {
131 cout << "NULL" << endl;
132 return false;
133 }
134 recoveryPoints[id]->run(*this);
135 return true;
136 }
137
138 private:
139
140 std::map<TypeID, RecoveryPoint *> recoveryPoints;
141 std::map<TypeID, State *> states;
142 TypeID current;
143 ConfigFile configFile;
144 Logger logger;
145 Timer t;
146 template<class S>
147 void addState() {
148 TypeID id = TypeIDFactory::getID<S>();
149 states[id] = new S;
150 }
151
152 void run() {
153 static bool RUNNING = true;
154 while (RUNNING) {
155 t.reset();
156 t.start();
157 states[current]->run(*this);
158 logger.log("%f", t.read());
159 t.stop();
160 }
161 }
162
163 void serializeToFlash(TypeID id) {
164 configFile.setInt("estado", id);
165 configFile.save();
166 }
167 };
168
169 class Application {
```

```

170 protected:
171
172 Application() :
173 localFileSystem("local"), configFile(RECOVERY_FILE) {
174 }
175
176 //Application(Application &orig) : localFileSystem(orig.localFileSystem) { }
177
178 virtual ~Application() {
179 }
180
181 public:
182
183 template<class Application>
184 static Application* getInstance() {
185 return (Application *) instance;
186 }
187
188
189 template<class App>
190 static void initialize() {
191 cout << "initialize" << endl;
192 if (instance != NULL) {
193 cout << "Application has already been initialized";
194 }
195
196 static StateMachine sm;
197
198 instance = new App;
199 instance->setup();
200 instance->createRecoveryPoints(sm);
201
202 static bool RESET_BYFAULT = true;
203 bool started = false;
204 cout << "RESET_BYFAULT" << endl;
205 if (RESET_BYFAULT && instance->recovery(sm)) {
206 cout << "RECOVERY" << endl;
207 started = true;
208 }
209 if (!started) {
210 instance->start(sm);
211 }
212 }
213
214 TypeID unserializeFromFlash() {
215 int a = atoi(configFile.get("estado"));
216 cout << "lendo = " << a << endl;

```

```

217 return a;
218 }
219
220 protected:
221
222 virtual void setup() {
223 }
224
225 virtual void createRecoveryPoints(StateMachine &sm) {
226 }
227
228 virtual void start(StateMachine &sm) = 0;
229
230 private:
231
232 static Application *instance;
233
234 LocalFileSystem localFileSystem;
235
236 ConfigFile configFile;
237
238 bool recovery(StateMachine &sm) {
239 if (configFile.load()) {
240 TypeID id = unserializeFromFlash();
241 cout << "id = " << id << endl;
242
243 return sm.recoveryFrom(id);
244 }
245 return false;
246 }
247 };
248
249 #endif //STATEMACHINELIB_STATEMACHINE_H

```

```

1 #include <iostream>
2 using namespace std;
3
4 template<class T>
5 class TData {
6
7     public:
8
9     TData(){
10
11 }
12
13     TData(T t) {

```

```
14         setData(t);
15     }
16
17     TData(T *t) {
18         setData(t);
19     }
20
21     T getData();
22
23     T* getDataObject();
24
25     void injectFault();
26     void injectFaultVetor();
27
28     void setData(T data);
29     void setData(T *data);
30
31     TData& operator=(T data) {
32         setData(data);
33         return *this;
34     }
35     TData& operator=(TData& obj) {
36         setData(obj.getData());
37         return *this;
38     }
39
40     operator T() {
41         return getData();
42     }
43
44     private:
45
46     T d1;
47     T d2;
48     T d3;
49     T dataObject;
50
51     T getByVotting();
52 };
53
54 template<class T>
55 T TData<T>::getByVotting() {
56
57     if (d1 == d2) {
58         if (d1 != d3) {
59             d3 = d1;
60         }
61     }
62 }
```

```
61     } else if (d1 == d3)
62     d2 = d1;
63     else if (d2 == d3)
64     d1 = d2;
65     else {
66         cout << "throws exception" << endl;
67     }
68
69     return d1;
70 }
71 template<typename T>
72 T TData<T>::getData() {
73     return getByVotting();
74 }
75
76 template<class T>
77 T* TData<T>::getDataObject() {
78     d1 = dataObject;
79     d2 = dataObject;
80     d3 = dataObject;
81     dataObject = getByVotting();
82     return &dataObject;
83 }
84
85 template<typename T>
86 void TData<T>::setData(T data) {
87     d1 = data;
88     d2 = data;
89     d3 = data;
90     dataObject = data;
91 }
92 template<typename T>
93 void TData<T>::setData(T *data) {
94     d1 = *data;
95     d2 = *data;
96     d3 = *data;
97     dataObject = *data;
98 }
99
100
101 template<class T>
102 void TData<T>::injectFaultVetor() {
103     d1 = 9;
104 }
```