UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM COMPUTAÇÃO

LUCIÉLI TOLFO BEQUE

Avaliação dos Requisitos para Teste de um Sistema Operacional Embarcado

Dissertação apresentada como requisito parcial para a obtenção do grau de Mestre em Ciência da Computação.

Profa. Dra. Érika Fernandes Cota
Orientadora

Porto Alegre, Agosto de 2009.

CIP – CATALOGAÇÃO NA PUBLICAÇÃO

Beque, Luciéli Tolfo

Avaliação dos Requisitos para Teste de um Sistema Operacional Embarcado / Luciéli Tolfo Beque – Porto Alegre: Programa de Pós-Graduação em Computação, 2009.

59 p.:il.

Dissertação (mestrado) – Universidade Federal do Rio Grande do Sul. Programa de Pós-Graduação em Computação. Porto Alegre, BR – RS, 2009. Orientador: Érika Fernandes Cota.

1. software embarcado 2. teste de software embarcado 3. teste de Sistema Operacional Embarcado (SOE) 4. tratamento de exceção 5. teste da rotina de tratamento de exceção. I. Cota, Érika Fernandes. II Título.

UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
Reitor: Prof. Carlos Alexandre Netto
Vice-Reitor: Prof. Rui Vicente Oppermann
Pró-Reitor de Pós-Graduação: Prof. Aldo Bolten Lucion
Diretor do Instituto de Informática: Prof. Flávio Rech Wagner
Coordenadora do PPGC: Profª Luciana Porcher Nedel
Bibliotecária-Chefe do Instituto de Informática: Beatriz Regina Bastos Haro

AGRADECIMENTOS

Agradeço:

- A Deus;
- A minha orientadora: Profª. Drª. Érika Fernandes Cota;
- Ao meu noivo: Giovani de Freitas Guerra;
- A minha família: José Luiz Carvalho Beque (pai), Lizabete Maria Tolfo Beque (mãe), Camila Tolfo Beque (irmã);
- Aos meus colegas e amigos de laboratório (laboratório 205, prédio 67). Em especial: Jefferson Luiz Bosa e Paulo Roberto Miranda Meirelles.

SUMÁRIO

LISTA DE ABREVIATURAS E SIGLAS 6

LISTA DE FIGURAS 8

LISTA DE TABELAS 9

RESUMO 10

ABSTRACT 11

1 INTRODUÇÃO 12

2 TESTE DE SOFTWARE 14

2.1 Teste de Software 14

2.2 Verificação e Validação 15

2.3 Estratégias de Teste 15
- 2.3.1 Teste de Unidade 16
- 2.3.2 Teste de Integração 16
- 2.3.3 Teste de Validação/Aceitação 17
- 2.3.4 Teste de Sistema 17

2.4 Técnicas de Teste 18
- 2.4.1 Teste de Caixa Branca 18
- 2.4.2 Teste de Caixa Preta 19
- 2.4.3 Teste de Caixa Cinza 20

2.5 Casos de Teste 20

2.6 Resumo e Conclusão 21

3 TESTE DE SOFTWARE EMBARCADO 22

3.1 Sistemas Embarcados 22

3.2 Software Embarcado 24

3.3 Software Embarcado – teste e seus desafios **25**

3.4 Estado da arte – Teste de Software Embarcado **27**

3.5 Sistema Operacional Embarcado (SOE) – características e desafios para o teste **29**

3.6 Resumo e Conclusão **30**

4 ESTUDO DE CASO: eCos – SIMICS - APLICAÇÃO EMBARCADA .. **32**

4.1 eCos **32**

4.1.1 Tratamento de exceção do eCos 38

4.2 Simics **41**

4.3 Aplicação Embarcada **42**

4.4 Resumo e Conclusão **44**

5 PLANEJAMENTO E AVALIAÇÃO DE TESTES PARA A ROTINA DE TRATAMENTO DE EXCEÇÃO DO SISTEMA OPERACIONAL eCos **46**

5.1 Definição do Modelo de Falhas **46**

5.2 Resultado da Injeção de Falhas **47**

5.3 Definição dos Casos de Teste **49**

5.4 Resultados dos testes **52**

5.5 Resumo e Conclusão **53**

6 CONCLUSÃO **54**

7 REFERÊNCIAS **57**

LISTA DE ABREVIATURAS E SIGLAS

ACDATE *Actor, Condition, Data, Action, Time, Event*

AD Conversor Analógico-Digital

AdS Camada de software dependente da aplicação

API *Application Programming Interface*

ARM *Advanced RISC Machines*

ASIP *Application Specific Instruction Set Processor*

CD-ROM *Compact Disc Read-Only Memory*

CPU *Central Processor Unit*

DA Conversor Digital-Analógico

DSP Processador Digital de Sinais

eCos *Embedded Configurable Operating System*

EmITM *Embedded System's Interface Test Model*

GPF *Global Protection Fault*

HAL *Hardware Abstration Layer*

HdS Camada do software dependente de hardware

ISO *International Organization for Standardization*

ISR *Interrupt Service Routine*

MB Megabyte

MHz Megahertz

MIPS *Microprocessor without Interlocked Pipeline Stages*

OdS	Camada de software dependente de serviços do SO
PC	*Program Counter*
PowerPC	*Power Optimization With Enhanced RISC - Performance Computing*.
RAM	*Random Access Memory*
RedBoot	*Red Hat Embedded Debug and Bootstrap*
RISC	*Reduced Instruction Set Computer*
ROM	*Read Only Memory*
RTOS	*Real-Time Operating System*
SICS	*Swedish Institute Computer Science*
SO	Sistema Operacional
SOE	Sistema Operacional Embarcado
VLIW	*Very Long Instruction Word*
VSR	*Vector Service Routine*

LISTA DE FIGURAS

Figura 3.1: Retorno Financeiro e janelas de tempo (CARRO e WAGNER, 2003). 23
Figura 3.2: Esquema genérico de um sistema embarcado (BROEKMAN e NOTENBOOM, 2003). 24
Figura 3.3: Camadas de um sistema embarcado (SUNG et al., 2007). 27
Figura 3.4: Interface de hardware e interface do SO (SUNG et al., 2007).................... 28
Figura 4.1: Um exemplo da camada dos pacotes (MASSA, 2002)............................... 34
Figura 4.2: Tela da inicial da ferramenta de configuração... 36
Figura 4.3: Janela para a escolha do template e pacotes. 36
Figura 4.4: Janela para seleção de novos pacotes... 37
Figura 4.5: Janela do Resolve Conflicts.. 37
Figura 4.6: Fluxo do Tratamento de Exceção do eCos (MASSA, 2002)...................... 39
Figura 4.7: Fluxo do tratamento de exceção implementado (BEQUE et al, 2009)........ 40
Figura 4.8: Tela de inicialização do Simics.. 41
Figura 4.9: Janela de comandos do Simics.. 42
Figura 4.10: Console. 42
Figura 4.11: Código da aplicação loop usando a API do Kernel para instanciar o tratamento de exceção. 43
Figura 4.12: Código da aplicação bubble sort usando a API do Kernel para instanciar o tratamento de exceção. 44
Figura 4.13: Plataforma definida para o estudo de caso.. 45
Figura 5.1: Fluxo do tratamento de exceção implementado com as falhas injetadas..... 47
Figura 5.2: Resultado da injeção de falhas.. 48
Figura 5.3: Fluxo do tratamento de exceção implementado com os casos de teste. 49

LISTA DE TABELAS

Tabela 3.1: Quadro comparativo entre um software tradicional e um software embarcado.. 26
Tabela 3.2 : Quadro comparativo entre as pesquisas analisadas. 29
Tabela 5.1: Relação entre cada caso de teste e o tipo de falhas detectadas.................... 48
Tabela 5.2: Relação entre cada caso de teste e o tipo de falhas detectadas.................... 52
Tabela 5.3: Resumo dos casos de teste definidos.. 52

RESUMO

A sociedade está cada vez mais dependente de sistemas embarcados, sendo que na grande maioria das vezes eles operam de maneira invisível aos seus usuários. Essa dependência torna esses usuários vulneráveis a riscos, devido às falhas que podem ocorrer. Essas falhas podem provocar perdas de vidas ou sérios danos materiais e financeiros. Devido a estes fatos, a qualidade destes produtos torna-se um ponto essencial para se ter um sistema estável, livre de erros e com todas as suas funcionalidades sendo executadas. De encontro a isso, a etapa de teste apresenta-se como indispensável e de relevada importância para a obtenção de um produto com uma boa qualidade. Devido ao alto custo de produção e energia gasto com testes, surge a necessidade de novos estudos, sobre diversificados métodos, para se testar um sistema embarcado. Neste contexto, este trabalho tem como objetivo apresentar os estudos iniciais do teste de um Sistema Operacional Embarcado (SOE), através de um estudo de caso focado na rotina de tratamento de exceção do eCos (*Embedded Configurable Operating System)*, pois ela apresenta uma forte interação entre software e hardware, sendo que esta interação é um dos principais desafios encontrados no teste de um software embarcado. Com isso, este trabalho pretende dar o passo inicial para pesquisas relacionadas aos testes de um Sistema Operacional Embarcado. Após a análise dos experimentos, pôde-se notar que a principal característica do Sistema Operacional Embarcado eCos, a configurabilidade, é um ponto de dificuldade extra para a realização dos testes, pois exige um estudo detalhado do código do SOE, o qual é totalmente genérico, antes do planejamento dos testes, podendo ser gasto muito tempo nessa atividade. Outro ponto é que o teste torna-se totalmente dependente do hardware. Entretanto, os resultados experimentais apresentados para o estudo de caso do presente trabalho foram satisfatórios.

Palavras-Chave: software embarcado, teste de software embarcado, teste de Sistema Operacional Embarcado (SOE), tratamento de exceção, teste da rotina de tratamento de exceção.

Testing Requirements for an Embedded Operating System

ABSTRACT

Society is increasingly dependent on embedded systems, which in most cases operate in an invisible manner to its users. This dependence makes the user vulnerable to risks due to failures that may occur. These failures can cause loss of lives or serious property and financial damage. Because of these facts, the quality of these products becomes a key point to have a stable system, free of errors and with all the features running. This testing is of essential importance to obtain a product with good quality. Due to the high cost of production and energy spent on tests, there is a need for further studies on different methods, to test an embedded system. In this context, this work aims at presenting the initial studies as the testing of the Embedded Operating System. The case study was focused on the exception handling routine of the eCos (Embedded Configurable Operating System), because it has a strong interaction between software and hardware, and this interaction is one of the main challenges encountered in testing embedded software. Therefore, this work aims at taking the first steps towards research related to testing an Embedded Operating System. After analyzing the experiments, it was noted that the main feature of the Embedded Operating System, eCos, the configurability, is an extra point of difficulty for the tests. It requires a detailed study of the code eCos, which is completely general, before the planning of tests, and could be spent much time in this activity. Another point is that the test becomes totally dependent on hardware. However, the experimental results presented for the case study of this study showed satisfactory.

Keywords: embedded system, embedded system test, embedded operating system test, exception handling, exception handling test

1 INTRODUÇÃO

Um sistema embarcado é uma combinação de software e hardware que é projetada para desempenhar uma tarefa específica. Ou seja, consiste em um sistema micro processado que suporta uma determinada aplicação (BARR, 1999), (CARRO e WAGNER, 2003). Esse fato o diferencia de um computador pessoal, o qual é projetado para inúmeros tipos de aplicações.

Os sistemas computacionais embarcados estão inseridos em inúmeros dispositivos eletrônicos e aplicações. Com os baixos custos tecnológicos atuais, eles estão tornando-se cada vez mais presentes no cotidiano das pessoas (CARRO e WAGNER, 2003), (WONG, 2006). Ainda, diversas são as funcionalidades executadas por esses dispositivos, tornando-os cada vez mais complexos. Devido a estes fatores, deve-se ter uma preocupação especial com a etapa de testes destes sistemas, pois ela é um fator importante para obter uma boa qualidade do produto final.

O projeto de um sistema embarcado é bastante complexo. Eles possuem inúmeras características, as quais tornam o desenvolvimento destes sistemas uma área inovadora da computação (WOLF, 2001). Pesquisas referentes aos testes ainda estão em evolução. O maior objetivo consiste em encontrar um meio de testar o sistema de maneira que o produto final desenvolvido tenha uma boa qualidade sem onerar o sistema em seus pontos críticos (consumo de energia, área ocupada, desempenho, custo de produção, entre outros), além de atender todos os requisitos especificados, sem a presença de erros. Para se obter uma qualidade aceitável de um produto, seja de software, de hardware, ou de um sistema embarcado, torna-se indispensável a realização da etapa de testes durante e após o seu desenvolvimento.

O principal objetivo da etapa de testes consiste na descoberta e eliminação de erros, falhas e/ou defeitos no software, bem como sua validação e aceitação. Esta etapa pode alcançar até 50% do custo de um sistema tanto de hardware (GIZOPOULOS et al., 2004), quanto do software (PRESSMAN, 2002). Sendo assim, os testes precisam ser bem desenvolvidos e aplicados, para tentar reduzir este elevado custo.

Testes bem estruturados e planejados, com baixo custo de tempo e recursos, têm sido desenvolvidos para garantir a um sistema embarcado uma boa qualidade (MORAES, 2006), (COTA e LUBASZEWSKI, 2004). Entretanto, o teste isolado do hardware ou do software embarcado não é suficiente para garantir o bom funcionamento do sistema final. Um sistema embarcado apresenta uma forte interação entre software e hardware, quando comparado com um sistema tradicional (não-embarcado). Sendo assim, o teste de um sistema embarcado inclui, além do teste da plataforma de hardware e do teste de componentes de software, o teste da interação entre estes dois componentes. Na verdade, o teste de interação software/hardware é o maior desafio para a verificação do sistema final, pois não existe ainda um método para testar essa interação (TAURION, 2005). Os testes de interação são usualmente baseados

em estratégias *ad-hoc*, as quais podem envolver um processo demorado e com um custo elevado.

Com a complexidade atual encontrada nos sistemas embarcados, a presença de um Sistema Operacional Embarcado (SOE), para gerenciar os componentes da aplicação e a interação entre o software/hardware, torna-se de relevada importância. Neste contexto, surge a motivação para o estudo de uma metodologia de teste voltada para componentes do SOE que possuam uma alta interação entre software e hardware. Sendo assim, este trabalho tem como objetivo principal apresentar os experimentos iniciais referentes ao teste da rotina de tratamento de exceção do Sistema Operacional Embarcado (SOE) eCos. A rotina de tratamento de exceção foi a selecionada, pois possui uma forte interação do software com o hardware.

Esse trabalho apresenta o passo inicial em direção a uma solução para resolver o desafio do teste de interação entre o hardware e software. A idéia é aplicar os testes, métodos e técnicas, que são voltados para os softwares tradicionais (não-embarcados), em um componente do SOE que possui uma alta interação software e hardware (rotina de tratamento de exceção), partindo do princípio que a metodologia de teste desenvolvida e aplicada em um SOE, provavelmente, possa abordar os problemas e características dos testes de interação para inúmeros sistemas. Para este estudo inicialmente foi preciso definir uma plataforma (hardware + software) para a realização dos experimentos. Após a definição da plataforma, os testes foram realizados e o conjunto de critérios (requisitos) para testar um Sistema Operacional Embarcado foi definido.

O presente trabalho está organizado da seguinte maneira. A primeira parte (Capítulo 2) consiste no estudo sobre os conceitos, critérios e técnicas de teste de um software não-embarcado. Na segunda parte (Capítulo 3) são abordados conceitos de sistemas embarcados, software embarcado, estado da arte, bem como os desafios e problemas encontrados no teste de um software embarcado. A terceira parte (Capítulo 4) apresenta o estudo de caso referente à definição da plataforma (eCos, Simics e aplicações embarcadas). O Capítulo 5 apresenta o planejamento e avaliação de testes para a rotina de tratamento de exceção do sistema operacional eCos. Por fim, no Capítulo 6, as conclusões e os trabalhos futuros são descritos.

2 TESTE DE SOFTWARE

Este Capítulo tem como objetivo apresentar os conceitos relacionados ao teste de software tradicional, bem como as técnicas e estratégias usadas e citadas pela Engenharia de Software.

2.1 Teste de Software

Engenharia de Software é uma disciplina que se ocupa com todos os aspectos da produção de software, desde o início da especificação dos requisitos, até a fase de manutenção desse sistema (SOMMERVILLE, 2003). A Engenharia de Software contém várias etapas, cada uma delas abrange métodos, ferramentas e procedimentos para o desenvolvimento do software. Dentre estas etapas, segundo Pressman (2002), encontra-se o ciclo de vida de um software, que consiste em atividades realizadas durante todo o desenvolvimento do sistema. Dentre estas atividades, está a atividade de teste.

A qualidade de produtos de software está fortemente relacionada à qualidade do processo de software, ou seja, para se obter um produto final com uma boa qualidade, é necessário garantir esta qualidade desde a primeira atividade do ciclo de vida, até a fase de manutenção do sistema (ROCHA, 2001). Qualidade de software, segundo Rezende (2002), é a conformidade com os requisitos, ou a adaptabilidade ao uso, adequação ao cliente e/ou usuário. É ainda, atendimento perfeito de forma confiável (sem defeitos), acessível (baixo custo), segura e no tempo certo, às necessidades do cliente. Para Rocha (2001), qualidade de software pode ser definida como um conjunto de características que devem ser alcançadas em um determinado grau para que o produto atenda às necessidades de seus usuários.

A qualidade do software está fortemente ligada à qualidade dos testes. De todo o esforço gasto para a realização de um projeto, 50% é gasto nesta atividade (PRESSMAN, 2006). Por esse motivo, a preocupação com os testes deve ocorrer ainda nas fases iniciais do projeto, partindo de um bom planejamento e controle, a fim de evitar perdas de recursos e atrasos no cronograma (DIAS NETO e TRAVASSOS, 2005). Além disso, o teste é uma fonte importante de *feedback* e fornece uma base para a interação com os participantes de todo o projeto do software.

Teste, segundo Pressman (2006), é um conjunto de atividades que podem ser antecipadamente planejadas e conduzidas sistematicamente, com o objetivo de encontrar erros, defeitos e/ou falhas no sistema. É importante diferenciar os conceitos de defeito, erro e falha (MALDONADO, 2004).

- **Defeito:** incapacidade de algum componente em realizar a função à qual foi projetado.

- **Falha:** é a incapacidade de o sistema executar suas funções requeridas dentro das exigências especificadas. Não existe falha se o programa não tem defeito**.**

- **Erro:** é a manifestação de uma falha no sistema, causando diferenças das respostas apresentadas com as esperadas. Nem todas as falhas causarão erros no programa.

Os testes de software são planejados e executados durante todo o desenvolvimento do software. Sendo assim, os componentes individuais do sistema devem ser testados, bem como a sua integração. Após executar esses testes, é preciso realizar testes de sistema e aceitação. As estratégias de teste (Seção 2.3) são o passo a passo da atividade de teste. Para cada estratégia de teste, podem ser empregadas diversas técnicas (Seção 2.4).

A essência da atividade de teste consiste em projetar casos de testes onde sejam descobertos diferentes tipos de erros em pouco tempo e esforço. Um caso de teste é composto por um conjunto de entradas possíveis em um programa que gera um conjunto esperado de saídas. A Seção 2.5 apresenta alguns conceitos referentes a casos de teste.

A etapa dos testes ainda engloba dois processos importantes que são a verificação e validação, que serão apresentadas nos próximos parágrafos (Seção 2.2).

2.2 Verificação e Validação

A verificação e validação constituem um processo que começa com as revisões de requisitos e continua com as revisões de projeto e as inspeções de código, até então, chegar aos testes (SOMMERVILLE, 2003).

A verificação consiste na análise do sistema em relação ao seu funcionamento, garantindo que as funções planejadas estejam sendo executadas corretamente. São testes de baixo nível que verificam o código-fonte.

A validação é a análise que compara o pedido pelo cliente com o que foi desenvolvido, ou seja, são testes de alto nível que verificam se as funções do sistema estão de acordo com os requisitos do cliente (PRESSMAN, 2002).

2.3 Estratégias de Teste

A estratégia de teste integra as técnicas de projeto de casos de teste de forma bem definida, para possibilitar ao testador manter um padrão que servirá como uma guia na execução do teste. De acordo com Pressman (2002), um esqueleto (*template*[1]) de teste de software deve ser definido para todo o processo de Engenharia de Software. Este documento serve para se obter um melhor resultado no processo de desenvolvimento de um software.

Uma estratégia de teste deve conter tanto testes de verificação como de validação. As estratégias de testes, que serão analisadas a seguir, variam de acordo com a

---

[1] *Template:* conjunto de passos no qual podem ser alocadas técnicas de projeto de casos de teste.

abrangência do teste, e possuem a seguinte classificação: teste de unidade, teste de integração, teste de validação/aceitação e teste de sistema.

2.3.1 Teste de Unidade

O teste de unidade é o teste que verifica o módulo, ou seja, a menor unidade de projeto de software. A vantagem deste teste é a possibilidade de uma análise mais profunda e específica de uma função independente do resto do código (LEWIS, 2000). Os caminhos de controle são testados, possibilitando descobrir erros nos limites dos módulos. Esta estratégia tem como base o teste de caixa branca.

As unidades testadas no sistema compreendem desde a interface, a estrutura de dados, as condições limites, os caminhos básicos e finalmente os caminhos de tratamento.

As atividades de testes de unidade são realizadas após ser desenvolvido, revisado e verificado o código-fonte, com o objetivo de identificar erros e analisar a sintaxe mais adequada para o sistema.

No teste de unidade, os casos de testes devem ser projetados para descobrir erros devido a computações errôneas, comparações incorretas ou fluxo de controle impróprio (PRESSMAN, 2002).

2.3.2 Teste de Integração

O teste de integração é uma técnica sistemática para a construção da estrutura do programa e, ao mesmo tempo, para descobrir erros associados à interface. O objetivo é construir a estrutura do programa determinada no projeto com os módulos que foram testados nos testes de unidade (PRESSMAN, 2002). Este teste foca no teste da união das unidades do sistema. O objetivo consiste em unir uma unidade à outra, somente depois da verificação da última unidade testada (LEWIS, 2000).

O teste de integração pode ser divido em teste de regressão, integração *top-down* e integração *bottom-up.*

2.3.2.1 Teste de Regressão

O software pode sofrer alterações quando um de seus módulos é adicionado na integração, e essas modificações podem causar problemas em suas funções. O teste de regressão é o teste que reexecuta automaticamente alguns testes em um software sempre que uma mudança ocorre (PETERS e PEDRYCZ, 2001).

2.3.2.2 Integração Top-Down

A integração *top-down* é uma abordagem incremental à construção da estrutura de programa. Os módulos são integrados movimentando-se de cima para baixo através da hierarquia de controle, começando do módulo do programa principal estendendo-se aos módulos subordinados (PRESSMAN, 2002). Os componentes de alto nível de um sistema são integrados e testados antes que seus projetos e sua implementação tenham terminado (SOMMERVILLE, 2003). Nessa estratégia é necessária a presença de *drivers* e *stubs.* O *driver* é utilizado como módulo de controle principal, e os módulos reais são substituídos por *stubs.* Os *stubs* são rotinas que não possuem código executável. Eles servem para marcar o local de uma rotina ainda não implementada. À medida que os testes vão sendo realizados os *stubs* são substituídos pelos módulos reais, um de cada vez.

2.3.2.3 Integração Bottom-Up

O teste de integração *bottom-up* inicia a construção e os testes com os módulos localizados nos níveis mais baixos da estrutura de programa, ou seja, os módulos são integrados de baixo para cima. Sendo assim, o processamento exigido para módulos subordinados está sempre disponível, eliminando a necessidade de *stubs* (PRESSMAN, 2002). Os componentes de nível inferior são integrados e testados antes que os componentes de nível mais elevado tenham sido desenvolvidos (SOMMERVILLE, 2003). Para cada combinação de módulos é criado um *driver* que coordena a entrada e a saída dos casos de teste. Quando um determinado módulo é testado, o *driver* é substituído pela combinação de módulos correspondentes, que passam a interagir com os módulos do nível superior.

2.3.3 Teste de Validação/Aceitação

Os testes de validação iniciam-se logo após os testes de integração, onde o sistema já está montado como um pacote e os erros de interfaces solucionados. Os testes de validação são realizados através de uma série de testes de caixa preta, no qual é verificado se o software está de acordo com a análise de requisitos, ou seja, com as exigências dos clientes.

Os testes de aceitação visam especificamente capacitar clientes a validar seus requisitos. Ele é realizado quase que exclusivamente pelo usuário final.

2.3.4 Teste de Sistema

Esta estratégia consiste na verificação do software, quando é incorporado a outros elementos do sistema, ou seja, se ele permanece sem a presença de falhas após a integração (LEWIS, 2000).

O teste de sistema consiste em uma série de diferentes testes, cujo objetivo principal é pôr completamente à prova o sistema baseado em computador. Todo o trabalho deve verificar se todos os elementos do sistema foram adequadamente integrados e realizam suas funções corretamente (PRESSMAN, 2002).

Nos próximos parágrafos os quatro tipos de testes de sistema são apresentados.

2.3.4.1 Teste de Recuperação

O software precisa recuperar-se de falhas e retornar o processamento dentro de um tempo previamente especificado. O teste de recuperação força o software a falhar de diversas maneiras e verifica se a recuperação é adequadamente executada (PRESSMAN, 2002).

2.3.4.2 Teste de Segurança

Todo o sistema baseado em computador que gerencie informações importantes ou provoque ações que possam prejudicar alguém, abrange um perigo em relação à sua segurança. Pessoas com má índole podem tentar acessar de forma imprópria ou ilegal o sistema, se este não tiver mecanismos de segurança.

O teste de segurança testa se todos os mecanismos de proteção embutidos em um sistema o protegerão, de fato, de acessos indevidos (PRESSMAN, 2002).

O testador tenta penetrar no sistema de todas as formas possíveis, levando em conta o tempo e recursos suficientes. Um bom teste de segurança é aquele em que o testador

consegue entrar no sistema, e ainda esse acesso deve custar mais caro que a informação obtida.

2.3.4.3 Teste de Estresse

O teste de estresse é realizado para confrontar os programas com situações anormais. O testador executa o sistema de forma a exigir recursos de quantidade, freqüência ou volume anormais, para obter os limites do software, e assim verificar o seu desempenho. Esse teste tem a função de verificar o comportamento do sistema em caso de falhas causadas pela sobrecarga, e verificar se ela não causou a corrupção de dados ou perda inesperada de serviços do usuário (SOMMERVILLE, 2003).

2.3.4.4 Teste de Desempenho

Os testes de desempenho são realizados para verificar os requisitos de desempenho definidos na especificação de requisitos. Ele deve ocorrer ao longo de todo o processo de teste, desde os testes de unidade, porém somente quando todos os elementos de um sistema estão totalmente integrados é que o desempenho real do sistema pode ser avaliado.

O teste de desempenho é geralmente combinado com teste de estresse e freqüentemente exige instrumentação de hardware e software, ou seja, é necessário analisar a utilização dos recursos rigorosamente (PRESSMAN, 2002).

2.4 Técnicas de Teste

As técnicas de testes servem como base para gerar casos de testes. Elas dividem-se em técnicas de teste baseado no conhecimento da estrutura lógica do programa (caixa branca), técnicas de teste onde apenas precisa-se ter o conhecimento das entradas e saídas possíveis do sistema (caixa preta), e ainda técnicas de caixa cinza. Nos próximos parágrafos serão descritas essas técnicas.

2.4.1 Teste de Caixa Branca

Testes de caixa branca ou testes estruturais consistem em uma abordagem de testes que são derivados do conhecimento da estrutura e da implementação do software (SOMMERVILLE, 2003). O testador busca testar e conhecer todo o código do sistema, examinando o caminho lógico para verificar seu funcionamento. Nesta técnica não há preocupação com os requisitos do sistema, ou seja, se o software está de acordo com os requisitos do cliente, mas sim com o seu funcionamento (LEWIS, 2000).

A seguir serão apresentados os principais testes de caixa branca, que são: teste de caminho básico, teste de condição, teste de fluxo de dados e teste de laços (PRESSMAN, 2002).

2.4.1.1 Teste de caminho básico

O teste de caminho básico tem a finalidade de definir um conjunto básico de caminhos para a execução de testes. Os casos de teste criados para testar o conjunto básico de caminhos precisam garantir a execução de cada instrução do programa pelo menos uma vez durante a atividade de teste (PRESSMAN, 2002).

2.4.1.2 Teste de condição

O teste de condição tem por finalidade testar todas as condições lógicas contidas em um determinado código de programa. Existem dois tipos de condições, as simples, que possuem somente um operador lógico, e a composta, a qual tem mais de um operador lógico. De acordo com Pressman (2002), os tipos de erros em uma condição são: erro de operador booleano, onde há existência de operadores booleanos incorretos, faltando ou extras; erro de variável booleana; erro de parênteses booleanos; erro de operador relacional; e erro de expressão aritmética.

2.4.1.3 Teste de fluxo de dados

O método de teste de fluxo de dados seleciona caminhos de teste de um programa de acordo com o fluxo dos dados, ou seja, as localizações das definições e usos de variáveis no programa (PRESSMAN, 2002). A seleção de um caminho de teste é complexa neste caso, e por isso o uso desta técnica deve ser avaliado.

2.4.1.4 Teste de laços

Consiste na realização de testes nos laços dos programas. O projetista realiza casos de testes de forma que seja feita a passagem por todos os passos do laço, validando cada construção de determinado laço.

2.4.2 Teste de Caixa Preta

Testes de caixa preta ou testes funcionais são métodos de testes realizados na interface do programa. Um teste de caixa preta examina alguns aspectos de um sistema sem se preocupar muito com a estrutura interna do software (PRESSMAN, 2002). Esta técnica de caixa preta verifica a funcionalidade do software, preocupando-se em executar todas as funções exigidas pelo usuário corretamente, sem se preocupar muito com o que foi implementado (LEWIS, 2000).

Os principais testes de caixa preta, como particionamento de equivalência, análise de valor limite e técnicas de grafo de causa-efeito, serão abordados a seguir (PRESSMAN, 2002).

2.4.2.1 Particionamento de equivalência

A entrada de dados no programa é dividida em classes que serão testadas a partir de um caso de uso específico. O objetivo dessa técnica é eliminar os casos de testes redundantes, fazendo com que sejam descobertos classes de erros (PRESSMAN, 2002).

2.4.2.2 Análise de valor limite

A análise do valor limite consiste na escolha de casos de testes que ultrapassem os limites do software, pois de acordo com Lewis (2000), é nos limites que se encontram a maior parte dos erros. O testador utiliza casos que extrapolem os valores máximos e mínimos suportados pelo sistema, com intuito de encontrar um erro.

2.4.2.3 Técnicas de grafo de causa-efeito

O grafo de causa-efeito é uma técnica para a elaboração de casos de teste. Esta técnica oferece uma representação concisa das condições lógicas e das ações correspondentes. A técnica segue quatro passos (PRESSMAN, 2002):

1. Causas (condições de entrada) e efeitos (ações) são relacionados para um módulo e um identificador é atribuído para cada um.
2. Um grafo de causa efeito é desenvolvido.
3. O grafo é convertido numa tabela de decisão.
4. As regras da tabela de decisão são convertidas em casos de teste.

2.4.3 Teste de Caixa Cinza

Os testes de caixa cinza consistem na combinação entre os testes de caixa preta e os testes de caixa branca. Esta técnica analisa a parte lógica mais a funcionalidade do sistema, fazendo uma comparação do que foi especificado com o que está sendo realizado. Usando esse método, o testador comunica-se com o desenvolvedor para entender melhor o sistema e otimizar os casos de testes que serão realizados (LEWIS, 2000).

Os casos de testes são gerados baseados nas técnicas de testes, mas para a elaboração de um bom caso de teste é necessário aplicar uma estratégia de teste para a execução da técnica. Até aqui foram apresentadas as principais técnicas e estratégias de testes, na seqüência serão abordados alguns conceitos referentes a casos de teste.

2.5 Casos de Teste

Os casos de teste são gerados com base em métodos e/ou técnicas de testes que têm a finalidade de detectar e expor os defeitos para uma futura correção. Um bom caso de teste é aquele que descobre o maior número de erros em menos tempo e com o menor esforço.

Um caso de teste é configurado da seguinte maneira (adaptado de CARTAXO, 2006).

Entradas

Condição Inicial: assegura a condição inicial para que o caso de teste possa ser executado;

Passos: passos a serem executados, identificados pelos métodos de teste.

Saídas

Resultados esperados: respostas esperadas do sistema, para a entrada atual.

O processo do teste de software deve ser concorrente ao ciclo de vida da Engenharia de Software. Juntamente a essa idéia, a geração de casos de teste deve acontecer desde a etapa de especificação dos requisitos.

A geração dos casos de testes consiste em um terço do esforço gasto na atividade de teste. A execução dos testes e a avaliação completam esses custos. Devido a este fator, a especificação dos casos de teste torna-se importante para o sucesso final da etapa de teste.

Um ponto a ser ressaltado é que nem todos os casos de teste desenvolvidos podem ser executados, devido aos custos e tempo demandados. Para resolver esse problema,

estratégias para seleção dos casos de testes mais eficientes e eficazes vêm sendo pesquisadas. A idéia principal para diminuir os custos que envolvem a geração, seleção e execução dos casos de teste consiste na automação desses serviços.

2.6 Resumo e Conclusão

Neste Capítulo foi apresentada uma visão geral de testes para softwares tradicionais. Foram mostradas as técnicas e estratégias para teste que são usadas para gerar os casos de testes. Os casos de testes são desenvolvidos com a finalidade de detectar e expor os defeitos para uma futura correção. O caso de teste ideal consiste naquele que detecta o maior número de erros em menos tempo e esforço.

Até o momento, foram apresentados conceitos, técnicas e estratégias de testes relacionados a um software tradicional. No próximo Capítulo será abordado o conceito de sistemas embarcados, bem como os desafios para o teste de um software embarcado e de um Sistema Operacional Embarcado (SOE).

3 TESTE DE SOFTWARE EMBARCADO

Este Capítulo tem como objetivo apresentar os conceitos e características relacionados a sistemas embarcados e a softwares embarcados; os desafios para o teste de um software embarcado e de um Sistema Operacional Embarcado (SOE); bem como o estado da arte relacionado ao teste de software embarcado.

3.1 Sistemas Embarcados

Um sistema embarcado é uma combinação de software e hardware que é projetada para desempenhar uma tarefa específica. Ou seja, consiste em um sistema micro processado que suporta uma determinada aplicação (BARR, 1999), (CARRO e WAGNER, 2003). Esse fato o diferencia de um computador pessoal, o qual é projetado para inúmeros tipos de aplicações.

Os sistemas computacionais embarcados estão inseridos em inúmeros dispositivos eletrônicos e aplicações, por exemplo, telefones celulares com câmera fotográfica e agenda, sistema de controle dos carros e ônibus, máquina de lavar roupa, entre outros. Devido ao baixo custo tecnológico atual, eles estão ficando cada vez mais presentes no cotidiano das pessoas (CARRO e WAGNER, 2003).

Atualmente ainda é bastante complexo o projeto desde tipo de sistema. Eles possuem inúmeras características, as quais tornam o desenvolvimento de sistemas computacionais embarcados uma área inovadora da computação (WOLF, 2001). A grande pressão do mercado no mundo globalizado, juntamente com a contínua evolução tecnológica, impõem às empresas a necessidade de projetarem novos sistemas embarcados em um curto espaço de tempo (CARRO e WAGNER, 2003). A Figura 3.1 apresenta o gráfico do retorno financeiro em relação ao tempo; pode-se observar que o atraso de poucas semanas no lançamento do produto, pode comprometer seriamente os ganhos de um novo produto no mercado. De encontro a estes fatos, a preocupação com a etapa de testes destes sistemas torna-se um fator importante e relevante para se obter uma boa qualidade do produto final, sem perda de tempo no projeto e no desenvolvimento.

Um sistema embarcado consiste basicamente na combinação de software e hardware (plataforma de hardware), que são projetados para suportar uma determinada aplicação. O hardware de um sistema embarcado é composto, basicamente, pelos seguintes componentes: processador, memórias e periféricos. Segundo Barr (1999), o termo processador, no ambiente de sistemas embarcados, refere-se a microprocessadores, microcontroladores ou DSPs (processador digital de sinais). Os processadores embarcados podem ser de diversos tipos, dependendo da aplicação, por exemplo, RISC, VLIW, DSP, até ASIPs. Microprocessadores, em geral, possuem uma CPU, mais poderosa que a dos microcontroladores, e não são projetados para um uso específico.

Por outro lado, os microcontroladores, são projetados para uso específico em sistemas embarcados e, normalmente, incluem uma CPU, memória (RAM, ROM ou ambas) e outros periféricos integrados no circuito. Os DSPs possuem uma CPU especializada em processamento de sinais.

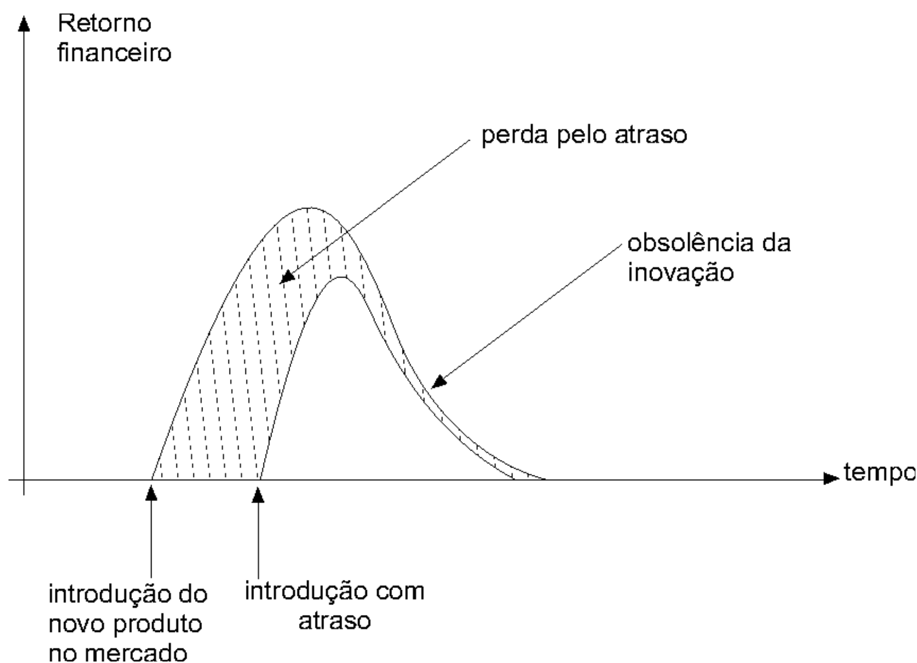

Figura 3.1: Retorno Financeiro e janelas de tempo (CARRO e WAGNER, 2003).

No caso de sistemas embarcados contendo componentes programáveis, o software de aplicação pode ser composto por múltiplos processos, distribuídos entre diferentes processadores e comunicando-se através de mecanismos variados. Um Sistema Operacional Embarcado (SOE) (BURNS e WELLINGS, 1997), oferecendo serviços como comunicação e escalonamento de processos, pode ser necessário (CARRO e WAGNER, 2003). Não é obrigatório ter um sistema operacional para executar uma aplicação no sistema embarcado, pois todas as suas funções podem ser implementadas diretamente na aplicação. Porém, à medida que a complexidade das aplicações aumenta a utilidade de um Sistema Operacional, para gerenciar o acesso da aplicação ao hardware, torna-se de elevada importância.

Através da Figura 3.2 podem-se observar os diversos componentes que podem fazer parte de um sistema embarcado. Um sistema embarcado interage com outros sistemas (embarcados ou não), através de interfaces específicas, recebendo sinais através de sensores, e enviando sinais através de atuadores os quais manipulam o ambiente em questão.

Geralmente, o software embarcado é armazenado na memória ROM do sistema embarcado, mas pode ser armazenado em cartão de memória, disco rígido ou CD-ROM e ser carregado através de uma rede ou satélite. O software embarcado é compilado para um determinado processador/plataforma (unidade de processamento), o qual requer certa quantidade de memória RAM para operar. Conversores, digital-analógico (DA) e analógico-digital (AD) são necessários quando a unidade de processamento processar somente dados digitais, enquanto o ambiente distribui sinais analógicos. A camada de Entrada/Saída é determinada para controlar a entrada e saída dos sinais. Outro componente comum em um sistema embarcado consiste na fonte de energia, a qual

geralmente possui alimentação própria e dedicada (baterias) (BROEKMAN e NOTENBOOM, 2003).

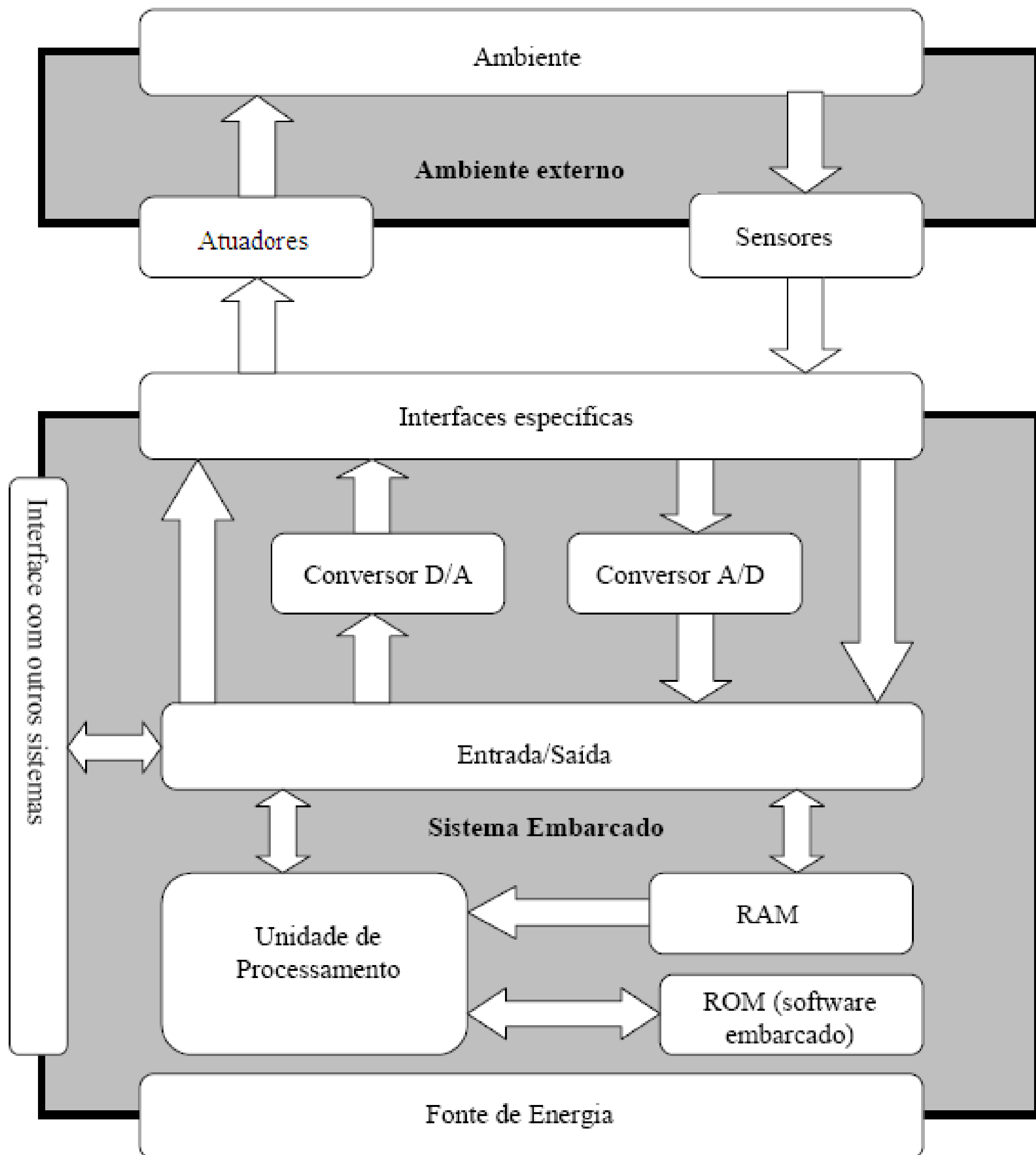

Figura 3.2: Esquema genérico de um sistema embarcado (BROEKMAN e NOTENBOOM, 2003).

3.2 Software Embarcado

Software Embarcado é um software especial que possui como foco principal a interação com o meio físico em que está inserido. Por esse motivo, ele tem por necessidade adquirir algumas propriedades desse meio, como exemplo, tempo de execução, consumo de energia, desempenho, entre outros (LEE, 2001). Geralmente está associado a um nível mais baixo do sistema, justamente por causa dessa interação.

A complexidade e o tamanho de aplicações embarcadas tendem a crescer rapidamente, e conseqüentemente as suas características tendem a ser mais realçadas. Dentre as características que demarcam a complexidade de um software embarcado, encontra-se o *timeliness (*resposta em tempo-real)*, liveness* (um software embarcado jamais poderá entrar em *deadlock* (impasse entre processos)), concorrência de processos, reatividade (interatividade entre os sistemas), e a heterogeneidade. Estas características precisam estar presentes em um software embarcado, mesmo que de maneira intrínseca (LEE, 2001). A confiabilidade é outra característica chave de um software embarcado, pois a medida que o software torna-se mais complexo e mais dependente do hardware, maior o grau de exigência em relação à sua segurança (KOOPMAN, 2007).

O software escrito para sistemas embarcados é muitas vezes chamado de *firmware*, e é armazenado em uma memória ROM ou memória flash ao invés de um disco rígido. Por vezes, o software embarcado também é executado com recursos computacionais limitados: sem teclado, sem tela e com pouca memória. Sendo assim, o tamanho do arquivo executável torna-se outra característica importante em um software embarcado. O tamanho do código compilado deverá ser relativo à memória física disponível.

A disponibilidade, ou seja, o poder de auto-recuperação caso ocorra um erro, é outra característica que precisa ser levada em consideração no desenvolvimento de um software embarcado. Sistemas embarcados geralmente residem em máquinas que precisam trabalhar continuamente por anos sem erros, e a disponibilidade é uma característica crucial para alcançar esse objetivo.

O desenvolvimento de um software embarcado é diferente do desenvolvimento de um software tradicional. O desenvolvedor precisa preocupar-se com os recursos oferecidos pelo sistema embarcado (por exemplo, memória), diferentemente de um software tradicional, onde o sistema tem inúmeros recursos computacionais (TAURION, 2005).

O software embarcado possui inúmeros aspectos agregados ao seu funcionamento que o diferenciam de um software tradicional, entre eles, limite de memória disponível, limite de desempenho do processador, mais tempo gasto com projeto do software e maior custo (LEE, 2001). Essas características são dependentes entre si. Por exemplo, a quantidade de memória disponível tem impacto na potência do sistema. Memórias mais rápidas gastam mais energia. Em contrapartida, um processador com freqüência mais baixa, gasta menos energia, entretanto pode não atingir o desempenho exigido pela aplicação embarcada. A Tabela 3.1 apresenta um quadro comparativo entre um software tradicional e um software embarcado. Através da tabela podem-se observar as diferentes características entre ambos os softwares.

3.3 Software Embarcado – teste e seus desafios

Apesar de existirem diversas pesquisas voltadas para teste de software embarcado (Seção 3.4), esse assunto ainda é inovador e com diversos desafios a serem alcançados. Um deles é o teste de interação software/hardware. Poucas pesquisas preocupam-se com esse grande e importante desafio. Outro ponto a ser analisado, que devido às características específicas apresentadas na Seção 3.2 (especialmente a disponibilidade e confiabilidade, juntamente com a complexidade, que em geral softwares tradicionais não apresentam), o software embarcado exige métodos de teste mais aprimorados do que os propostos pela Engenharia de Software tradicional, pois esta não possui

preocupação com uso de memória, desempenho, respostas de tempo real, entre outras questões.

Tabela 3.1: Quadro comparativo entre um software tradicional e um software embarcado.

	Software Tradicional – (Foco em Sistema Informação)	***Software Embarcado***
Tempo para lançamento no mercado (*time-to-market*)	Pode ser longo	Precisa ser curto (Figura 3.1)
Domínio de Aplicação	Única	Diversos (voltado para o hardware)
Modelo Computacional	Único	Múltiplos
Ferramentas para Automação	Maduro	Imaturo
Reuso de Código	Sem restrições	Com restrições
Gasto de energia	Não é considerado	É considerado
Tamanho da memória	Geralmente não considera	Precisa levar em consideração
Características de tempo real	Geralmente não considera	Precisa levar em consideração

Fonte: FERREIRA et al., 2009.

A adoção de métodos de testes da Engenharia de Software é essencial para sistemas cada vez mais complexos e com curtos prazos de entrega. Entretanto, de acordo com Taurion (2005), o uso de métodos não é uma prática comum no desenvolvimento de software embarcado. Em geral a estratégia para testar um software embarcado é realizada para determinar se um sistema satisfaz ou não seus critérios de aceitação. Os testes são conduzidos de maneira *ad-hoc*, ou seja, é feito para cada aplicação e não tem preocupação em ser reusado. Não são estruturados e são designados para a solução de um problema específico. Além disso, analisando que os métodos de Engenharia de Software precisam ser aprimorados, para adaptar-se a um software embarcado, ainda não existe uma metodologia genérica de teste para softwares embarcados.

Geralmente o software embarcado é verificado através de técnicas de teste funcionais em um ambiente não embarcado, porém, o teste isolado não garante o bom funcionamento do sistema final (BROEKMAN e NOTENBOOM, 2003). Um sistema embarcado possui diversas camadas (Figura 3.3), e uma delas, a HdS (camada de software dependente de hardware) precisa ser levada em consideração no momento dos testes (SUNG et al., 2007). Essa interação consiste em um dos principais desafios para o teste (TAURION, 2005). As falhas de interação software/hardware devem ser consideradas, pois o nível de qualidade e confiabilidade é cada vez mais elevado nesses casos.

À medida que a complexidade das aplicações aumenta a utilidade de um software embarcado mais especialista, como um Sistema Operacional Embarcado, para gerenciar o acesso da aplicação ao hardware, torna-se de elevada importância. Esse software embarcado especialista (SOE) funciona como um dispositivo de controle, e possui uma

alta interação entre o software e o hardware embarcado. Sendo assim, como o software embarcado, o SOE precisa ser testado. Como um software embarcado, um SOE também possui desafios relacionados ao teste, bem como características especiais, as quais precisam ser levadas em consideração nessa etapa.

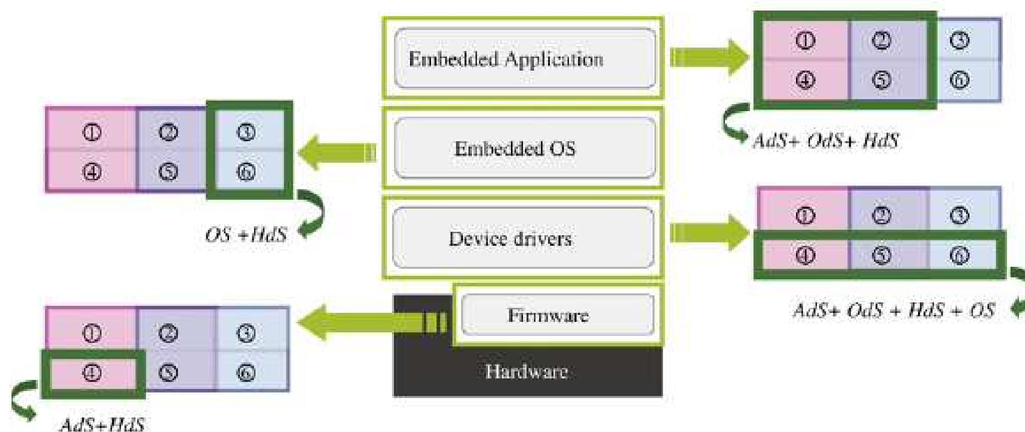

Figura 3.3: Camadas de um sistema embarcado (SUNG et al., 2007).

3.4 Estado da arte – Teste de Software Embarcado

A maioria dos trabalhos relacionados com testes de software embarcado está voltada ao teste de aplicações mais específicas, por exemplo, soluções automotivas, para celulares, aviação, entre outros. Yu (2006) apresenta a verificação de um sistema embarcado através do uso de padrões de cenários. Os componentes são modelados usando o padrão ACDATE (do inglês *Actor, Condition, Data, Action, Time, Event*). Guan et. al (2006) mostra um estudo de caso industrial, onde o software embarcado é considerado crítico do ponto de vista de segurança (*safety-critical*), ou seja, exige alta confiabilidade e segurança. Ele compara as técnicas usadas pela empresa, que são técnicas funcionais, as quais não atingem as características críticas do sistema, com técnicas estruturais, propostas para aumentar a cobertura dos testes. Pfaller et. al (2006) apresenta um modelo de teste que integra projeto e teste. Este modelo propõe o desenvolvimento de casos de testes durante todas as etapas do projeto do sistema embarcado. Nesta proposta, os requisitos extraídos dos usuários são usados como especificação para os casos de testes. Somente a interação do sistema com o usuário é levada em conta para o desenvolvimento dos casos de teste. Já Lettin et al. (2008) apresenta a verificação das propriedades temporais de um software embarcado automotivo usando um hardware verificador.

Outra linha de pesquisa analisada consiste no teste da interface do hardware e do SO, *Embedded System's Interface Test Model* (EmITM) (SUNG et al., 2007). Este modelo leva em consideração a estrutura hierárquica do sistema embarcado, que inclui a camada de software dependente de hardware, a camada do sistema operacional, e a camada de aplicação. Esta proposta consegue detectar (mas não localiza) falhas que ocorrem na interface de hardware e interface do SO (Sistema Operacional) enquanto o software embarcado é carregado e executado.

O EmITM propõe um conjunto de itens a serem testados (*checklist* - preenchido manualmente de acordo com o tipo de teste realizado) baseado na hipótese de que as falhas acontecem nas interfaces (interface do hardware e do SO) para testar um software embarcado. O *checklist* foi extraído a partir das características dos sistemas

operacionais ELCPS, POSIX1003 e KELPS (SUNG et al., 2007). A Figura 3.4 apresenta um exemplo de um sistema embarcado com SO. Nesta Figura pode-se observar a camada de software dependente da aplicação (AdS), a camada de software dependente de serviços do SO (OdS), a camada do software dependente de hardware (HdS) e por fim o *Kernel* do SO. Cada componente (exemplo: *firmware* – HdS) de um software embarcado é classificado como AdS, OdS ou HdS.

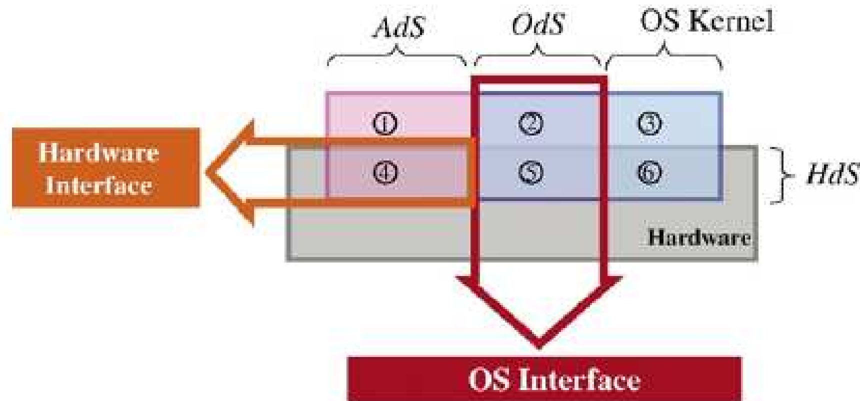

Figura 3.4: Interface de hardware e interface do SO (SUNG et al., 2007).

O EmITM basicamente apresenta o *checklist* para verificar a interface do hardware e do SO. Como exemplo dos itens de teste de interface de hardware tem-se características da memória de dados e instruções, dispositivos de I/O, temporizadores, entre outros. Características de gerenciamento de tarefas, comunicação, tempo de gerenciamento, exceções, interrupções, sistemas de arquivos, gerenciamento de memória, entre outros, são exemplos de itens relacionados à interface do SO.

Para a obtenção e análise dos resultados (teste das interfaces) e preenchimento do *checklist* foram realizados dois tipos de testes. O primeiro foi chamado de teste baseado em execução, no qual foram executados testes funcionais. O segundo foi nomeado como teste baseado na não execução. O qual consiste no teste baseado na revisão do código. Os dois tipos de testes foram realizados para testar tanto interface de hardware quando a interface do SO. O modelo de falhas adotado pela pesquisa segue de acordo como a descrição dos itens do *checklist*, e são chamadas de falhas de interface.

Para a detecção de uma falha, é realizada a análise do código para cada item. Ou seja, se o *checklist* apontar problemas no item tratamento de exceção, por exemplo, o testador saberá que há uma falha neste respectivo item. Entretanto, o testador não saberá exatamente o que causou esta falha.

Levando em consideração a possibilidade do teste de interação software/hardware, estudos relacionados ao teste do Sistema Operacional Embarcado podem ser um meio de atingir essa interação. De fato, determinadas condições para o funcionamento do SOE são extremamente dependentes do hardware, uma vez que ambos são especializados para uma dada aplicação.

Dentre as pesquisas analisadas, nenhuma trabalha especificamente com teste de um SOE. Tan et al. (2003) e Fei et al. (2004) usam o *embedded Linux OS*, e têm como principal objetivo, respectivamente, diminuir a energia consumida com as funções e serviços do SO; e analisar as características de consumo de energia do SO. Ou seja, são pesquisas que usam um SOE para atingir e melhorar características do sistema embarcado durante o projeto, não consideram a etapa de testes.

A Tabela 3.2 apresenta um quadro comparativo dos trabalhos relacionados. A Tabela apresenta o tipo de teste e o foco do trabalho. Analisando que um dos principais desafios do teste de um software embarcado é o teste de interação do software e hardware, os trabalhos (YU, 2006), (GUAN et al., 2006), (PFALLER et al., 2006) e (LETTNIN et al., 2008) não possuem essa preocupação como principal objetivo. Sung et al., (2007) leva em consideração a hierarquia do sistema embarcado para testar as interfaces, usando método funcional e análise de código para verificar o *checklist* proposto. Os trabalhos, (TAN et al., 2003) e (FEI et al., 2004), que trabalham com as características de um SOE, não focam no teste, e sim no projeto de um sistema embarcado.

Tabela 3.2 : Quadro comparativo entre as pesquisas analisadas.

	Tan et al. (2003)	*Fei et al. (2004)*	*Yu (2006)*	*Guan et. al. (2006)*	*Pfaller et. al. (2006)*	*Sung et al. (2007)*	*Lettin et al (2008)*
Tipo de Teste	-	-	Verificação	Estrutural	Verificação /Validação	Funcional/ Verificação de código	Verificação
Foco	Usam o *embedded Linux OS* - diminuir a energia consumida com as funções e serviços do SO	Usam o *embedded Linux OS* – analisar as características de consumo de energia do SO	Verificação de um Sistema Embarcado usando padrões de cenários	Caso industrial crítico de segurança	Integração do teste ao projeto	Teste das interfaces - software dependente de hardware e SOE	Verificação das propriedades temporais de um software embarcado automotivo

3.5 Sistema Operacional Embarcado (SOE) – características e desafios para o teste

Um Sistema Operacional (SO) pode ser visto como um programa de grande complexidade que é responsável por todo o funcionamento de uma máquina desde o software a todo hardware instalado na máquina (SILBERSCHATZ et al., 1999).

Um Sistema Operacional Embarcado (SOE) classifica-se como um software embarcado específico que possui uma alta interação com o hardware. Ele é composto por um conjunto de rotinas que podem ser programadas em diferentes níveis de abstração, da linguagem Assembly até código orientado a objetos, por exemplo. Rotinas de alto nível podem ser vistas e testadas como um software tradicional, através de técnicas de teste padrão (não-embarcado). Entretanto, rotinas de baixo nível requerem um procedimento de teste mais específico. Por exemplo, em algumas rotinas, parte do código é escrito em Assembly para implementar a interface com o microprocessador. Para esta parte, métodos de teste padrão podem não ser aplicados por diversas razões. Atualmente ferramentas de testes tradicionais são desenvolvidas para linguagens de mais alto nível. Além disso, embora o código em si seja bastante simples e quase

puramente seqüencial, sua análise desconectada da plataforma de hardware não tem qualquer resultado significativo. Na verdade, um código C pode acessar uma posição de memória que representa um registrador mapeado na memória. Uma ferramenta de teste padrão pode considerar este código como livre de falhas durante a simulação (na qual está desconectado do hardware) mesmo se houver um erro no índice de acesso à memória. Finalmente, a execução dessas rotinas de baixo nível, sem o hardware ou um modelo detalhado de hardware (ou seja, um simulador) não é exato e pode levar a uma menor cobertura de falhas.

Sistemas operacionais, em geral, possuem um conjunto de suítes de teste[2] que podem ser utilizados para verificar uma determinada distribuição ou configuração do sistema (LINUX TEST PROJECT, 2008), (GCOV, 2008) e (THE OPEN GROUP, 2008). Estes testes são puramente funcionais e são compostos por exemplos de aplicações as quais exercitam características específicas do sistema. Estes testes são constantemente atualizados por desenvolvedores e usuários de acordo com os problemas encontrados. O problema desta estratégia para um Sistema Operacional Embarcado é que o SO é configurado e adaptado para plataformas e aplicações de hardware específicas, e não se tem garantia de que os casos de teste disponíveis levarão a uma boa cobertura ou irão abordar as características específicas exigidas pela nova aplicação. Além disso, a execução de todos os casos testes disponíveis, aumenta a cobertura, mas pode não ser possível devido ao curto tempo disponível para o projeto, característica importante de um software embarcado, que deve ser levada em consideração. Finalmente, para os sistemas operacionais embarcados, os suítes de teste devem ser aplicados na plataforma alvo, a qual também é projetada para uma aplicação muito específica e talvez não tenha recursos suficientes para a realização desta tarefa.

Os testes do sistema operacional deverão ser considerados juntamente com os testes da aplicação embarcada para reduzir o tempo de projeto de um sistema embarcado. De fato, as estratégias de teste atuais para software embarcado são baseadas na plataforma ou em simuladores. Assim, durante a aplicação do teste, o SOE também deve estar presente. No caso de detecção de erros durante o teste, o projeto do teste precisa passar informações para localizar a falha. Falhas no SOE podem interferir na execução dos testes planejados para a aplicação causando um diagnóstico equivocado e aumentando o tempo de depuração.

3.6 Resumo e Conclusão

Neste Capítulo foi apresentado o conceito de sistemas embarcados. O conceito e características de um software embarcado, e de um software embarcado especialista (Sistema Operacional Embarcado - SOE). Essas características tornam um software embarcado diferente de um software tradicional. O estado da arte sobre testes de softwares embarcados também foi apresentado. O objetivo deste estudo foi demonstrar que ainda existem questões a serem exploradas relacionadas ao teste de software embarcado. O grande desafio, o teste de interação software/hardware, ainda está em aberto.

Em relação ao teste de software embarcado e seus desafios, não existe ainda uma metodologia genérica para a realização dos testes, normalmente eles são realizados sem planejamento e sem documentação. Em geral a estratégia de teste utilizada é conduzida para determinar se um sistema satisfaz ou não seus critérios de aceitação. Outro desafio

[2] Suítes de teste: O conjunto de casos de teste definido para uma aplicação alvo.

importante referente ao teste de um software embarcado consiste no teste de interação do software e hardware, visto que o teste isolado do hardware ou do *software* embarcado não é o suficiente para garantir o bom funcionamento do sistema final.

Analisando o fato de que um SO é um software especialista que possui uma alta dependência do hardware, e ainda o fato de que o teste de um SO embarcado poder ser uma maneira de atingir essa interação software/hardware, nenhum trabalho relacionado trabalha com teste de um SOE.

Partindo do princípio que a maioria dos trabalhos relacionados a testes de software embarcado são em geral voltados para aplicações específicas, e também o fato de que o teste isolado do hardware e do software embarcado não é o suficiente para garantir a qualidade do sistema final (desafios no teste de interação software/hardware), surge a motivação para o estudo dirigido de testes relacionados a sistemas operacionais embarcados. SOE são softwares embarcados específicos, que possuem uma alta interação do software com o hardware. Pode-se imaginar que o teste realizado em um ambiente não embarcado (sem restrições de memória, por exemplo) não corresponda exatamente ao ambiente de funcionamento do software embarcado mais especializado, como um SOE, por exemplo.

De encontro ao fato que a etapa de testes é uma fase importante para garantir a qualidade de um software, e ainda, os inúmeros desafios, juntamente com a motivação do teste de um SO embarcado para atingir a interação do hardware e software, esse trabalho apresenta nos próximos Capítulos, os estudos e resultados iniciais dos testes de um SOE.

4 ESTUDO DE CASO: eCos – SIMICS - APLICAÇÃO EMBARCADA

O uso de um Sistema Operacional Embarcado está tornando-se bastante comum em um sistema embarcado. Devido a isto, a preocupação com a garantia da qualidade e da confiabilidade aumenta. Entretanto, igualmente a um software embarcado, que possui diferenças em relação a um software tradicional (diferenças de requisitos e restrições para os testes), um SOE também possui diferenças em relação a um software embarcado em geral. Sendo assim, o teste deste software embarcado especialista ainda é um assunto inovador na literatura. Em busca de uma pesquisa inicial nesta direção, o primeiro passo, antes do planejamento e definição dos testes para um SOE, consiste na definição de um ambiente de simulação (plataforma software/hardware) para a execução dos experimentos. Sendo assim, este trabalho apresenta os estudos referentes ao teste da rotina de tratamento de exceção (Subseção 4.1.1) do SOE eCos (Seção 4.1).

A escolha do eCos para o estudo de caso ocorreu devido à sua configurabilidade. A rotina de tratamento de exceção foi selecionada por possuir uma forte interação entre os componentes do software e do hardware. Linguagens modernas orientadas a objetos usualmente implementam em sua estrutura mecanismos para tratamento de exceção. Em códigos para softwares tradicionais, exceções são menos suscetíveis a acontecer, e o mau funcionamento do tratamento da exceção possui um impacto menor, quando comparado a um código de um software embarcado. Sendo assim testes funcionais ou da aplicação podem não assegurar o bom comportamento durante a operação real de um software embarcado.

A Seção 4.2 apresenta a plataforma de simulação Simics. Na Seção 4.3 os códigos embarcados desenvolvidos e utilizados para a realização dos testes serão abordados.

4.1 eCos

O eCos é um sistema operacional de tempo real (RTOS – do inglês *Real-Time Operating System*) desenvolvido pelos programadores da *Cygnus Solutions*, atualmente incorporado à *Red Hat*. Ele foi projetado para os mais variados tipos de projetos de sistemas embarcados, desde o mais simples até o mais complexo (eCos, 2008) e (MASSA, 2002). O eCos, além de ser um RTOS gratuito, ainda possui todos os componentes usados no desenvolvimento do sistema, como exemplo, o *cross-compiler* (LOHMANN et.al., 2006). O *cross-compiler* consiste em um compilador que produz códigos para diferentes tipos de arquiteturas, ou seja, o compilador roda na arquitetura 386 e gera código para a arquitetura ARM, por exemplo.

A principal característica do eCos é sua configurabilidade. Muitos sistemas operacionais embarcados contêm códigos que não serão utilizados por uma determinada aplicação, tornando o software maior e mais complexo. O Sistema Operacional Configurável Embarcado (eCos) possibilita ao desenvolvedor o controle total do sistema, podendo escolher somente as funcionalidades que serão usadas pela aplicação, deixando o SO embarcado específico. Esta característica possibilita uma economia no uso de memória (MASSA, 2002) e (LOHMANN, et.al., 2006).

O sistema operacional eCos foi projetado para ser portado para inúmeras arquiteturas e plataformas, incluindo arquiteturas de 16, 32 e 64 bits, microprocessadores, microcontroladores e DSPs . Atualmente o eCos suporta as seguintes arquiteturas, ARM, Hitachi H8300, Intel x86, MIPS, Matsushita AM3x, Motorola 68k, PowerPC, SuperH, SPARC e NEC V8xx, incluindo muitas de suas variações e evoluções (eCos, 2008).

A idéia inicial deste trabalho era portar o eCos para a arquitetura MIPS, porém foram encontradas diversas dificuldades ao tentar simular esta arquitetura no simulador Simics (selecionado devido ao fato de simular diversas arquiteturas usando um arquivo binário para a inicialização, e também por ser utilizado pelo grupo de pesquisa). Dentre os problemas, o principal é que o simulador não possui algumas instruções geradas pelo eCos para a arquitetura MIPS. Também, foram realizados testes para outras plataformas, ARM e PowerPC, porém os mesmos problemas foram encontrados. Diante desses testes e experimentos mal sucedidos, usando as arquiteturas MIPS, ARM e PowerPC, e ainda pela falta de um simulador com as características apresentadas pelo Simics, a arquitetura selecionada para o presente trabalho foi a i386.

A classificação e a relação dos principais componentes do eCos, são citadas a seguir (MASSA, 2002).

- Camada de Abstração de Hardware (HAL – do inglês *Hardware Abstration Layer*): provê uma camada de software a qual permite um acesso geral ao hardware.

- *Kernel*: inclui tratamento de interrupção e exceção, *threads*, mecanismos de sincronização, escalonador, temporizadores, contadores e alarmes.

- Bibliotecas C e Matemática: permitem compatibilidade padrão com chamadas de funções.

- *Drivers* de Dispositivos: incluindo padronização Serial, Ethernet, Flash, entre outras.

- Suporte ao depurador da GNU (GBD - GNU *Debugger*): provê suporte de comunicação entre o *target* (hardware onde a aplicação irá executar) e o *host* permitindo que aplicações sejam depuradas.

A Camada de Abstração de Hardware é a responsável por permitir que os outros componentes da estrutura do eCos consigam realizar operações específicas de hardware, independente da arquitetura de hardware utilizada (MASSA, 2002). O HAL pode ser classificado em três módulos: Arquitetura, Plataforma e Variante. O módulo Arquitetura abstrai a arquitetura básica da CPU. O módulo Variante é o processador específico de uma família de processadores. Por fim, o módulo Plataforma é o hardware existente, ou seja, o sistema que contém um processador descrito nos níveis de Arquitetura e Variante.

A Camada do *Kernel* consiste no núcleo do sistema. No *Kernel* encontram-se as funcionalidades esperadas em um sistema de tempo real, por exemplo, tratamento de exceções e interrupções, *threads*, mecanismos de sincronização, escalonador, contadores e alarmes. O desenvolvimento do *Kernel* seguiu alguns critérios, com o intuito de cumprir os objetivos de um sistema embarcado de tempo real (MASSA, 2002).

- *Interrupt Latency* – O tempo necessário para responder a uma interrupção e iniciar a execução de uma ISR (*Interrupt Service Routine*) é mantido baixo e determinístico.
- *Dispatch Latency* – O tempo de quando uma *thread* já está pronta para ser executada e o ponto onde ela começa a execução é mantido baixo e determinístico.
- *Memory footprint* – A memória necessária para o código e os dados é mantida mínima e determinística. A alocação da memória é dinâmica para assegurar que o sistema embarcado não fique sem memória.
- *Deterministic Kernel Primitives* – a execução das operações do *Kernel* é previsível, permitindo que um sistema embarcado satisfaça os requisitos de tempo real.

A Figura 4.1 apresenta um exemplo de componentes disponíveis no eCos, que podem ser usados para configuração de um determinada aplicação.

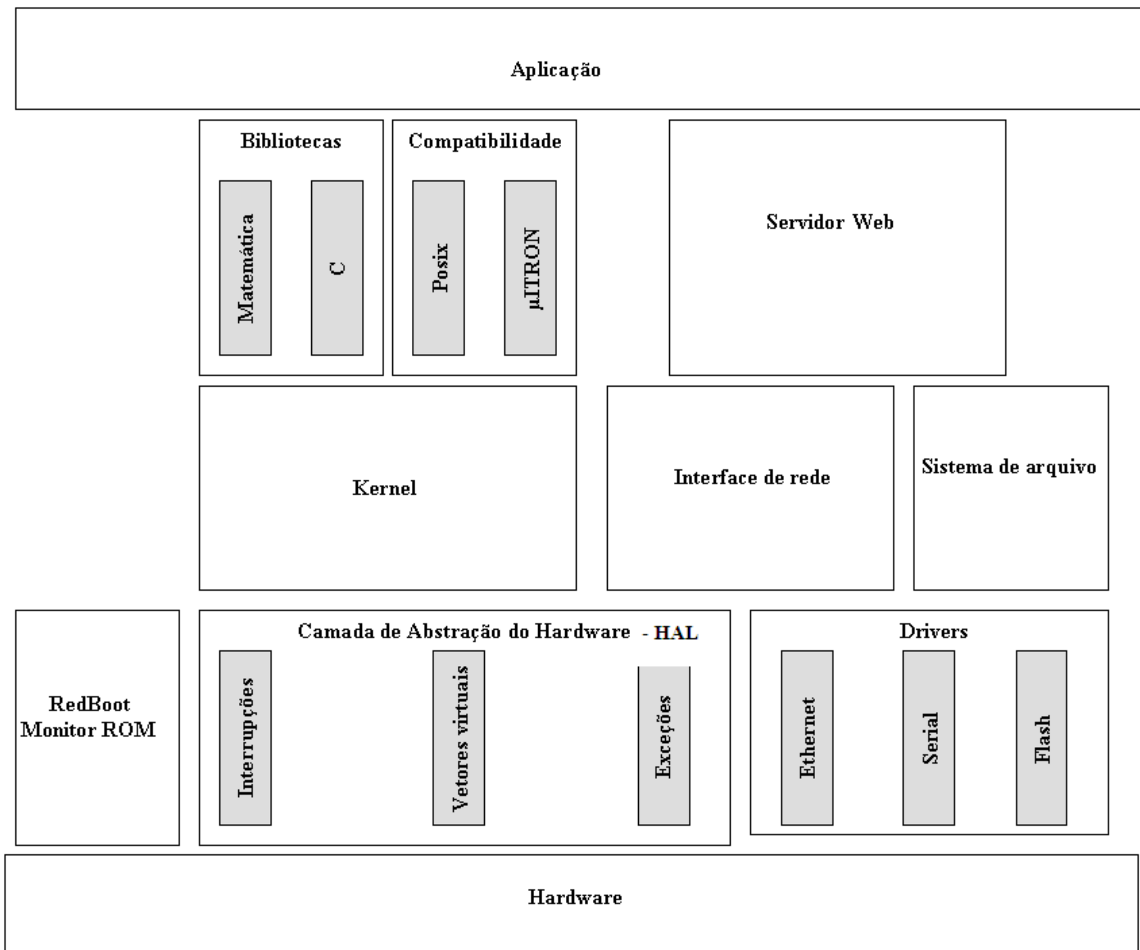

Figura 4.1: Um exemplo da camada dos pacotes (MASSA, 2002).

O *RedBoot* (*Red Hat Embedded Debug and Bootstrap*) é um programa baseado no eCos e projetado para inicializar o sistema embarcado. Fornece funcionalidades básicas como acesso à memória e Entrada/Saída. Através dele é possível carregar arquivos binários para o sistema embarcado na memória RAM ou na memória flash, para que sejam executados. O *RedBoot* funciona também como um depurador em sistemas embarcados.

A seqüência de inicialização do eCos está programada em linguagem *Assembly* no arquivo *vectors.S*. Esse código é o primeiro a ser executado pelo processador quando o processo é carregado. Neste arquivo encontra-se a inicialização da *cache* e pilha, chaveamento das CPUs, escalonador, interrupção, *clock*, entre outros. A inicialização da execução da aplicação ocorre quando a função *cyg_start* é chamada.

Para permitir a configuração dos diversos componentes de software do sistema, o eCos disponibiliza a ferramenta gráfica de configuração (*eCos Configuration Tool*). Através desta ferramenta é realizada a seleção dos pacotes utilizados para a construção da biblioteca do eCos, a qual será ligada com a aplicação para executar o Sistema Operacional Embarcado em uma determinada plataforma de hardware (LOHMANN, et.al., 2006). O termo pacote refere-se a um componente de software que inclua arquivos fonte e de configuração, prontos para uso no eCos. A Figura 4.2 apresenta a interface da ferramenta de configuração. A versão utilizada do eCos foi a 2.0.

Para trabalhar na ferramenta é preciso definir o caminho do repositório, ou seja, o endereço onde se encontram instalados todos os arquivos referentes ao eCos-2.0. No menu "*Build – Repository*" o seguinte caminho foi definido: "/opt/ecos/ecos-2.0". Através da *eCos Configuration Tool* é realizada a configuração do *RedBoot* e de outros componentes do eCos, para então criar um arquivo binário, o qual será executado na plataforma.

A escolha da plataforma é realizada no menu "*Build – Templates"* (Figura 4.3). A plataforma selecionada foi i386 *PC target,* uma versão de baixo custo, específico para o mercado de embarcados. Ao ser selecionada uma plataforma são carregados diversos pacotes correspondentes a componentes do hardware. Ainda nessa janela, tem-se a escolha dos pacotes correspondentes aos componentes de hardware presentes no mesmo. O pacote selecionado para o presente trabalho foi o *minimal,* o qual possui componentes do HAL e da infra-estrutura.

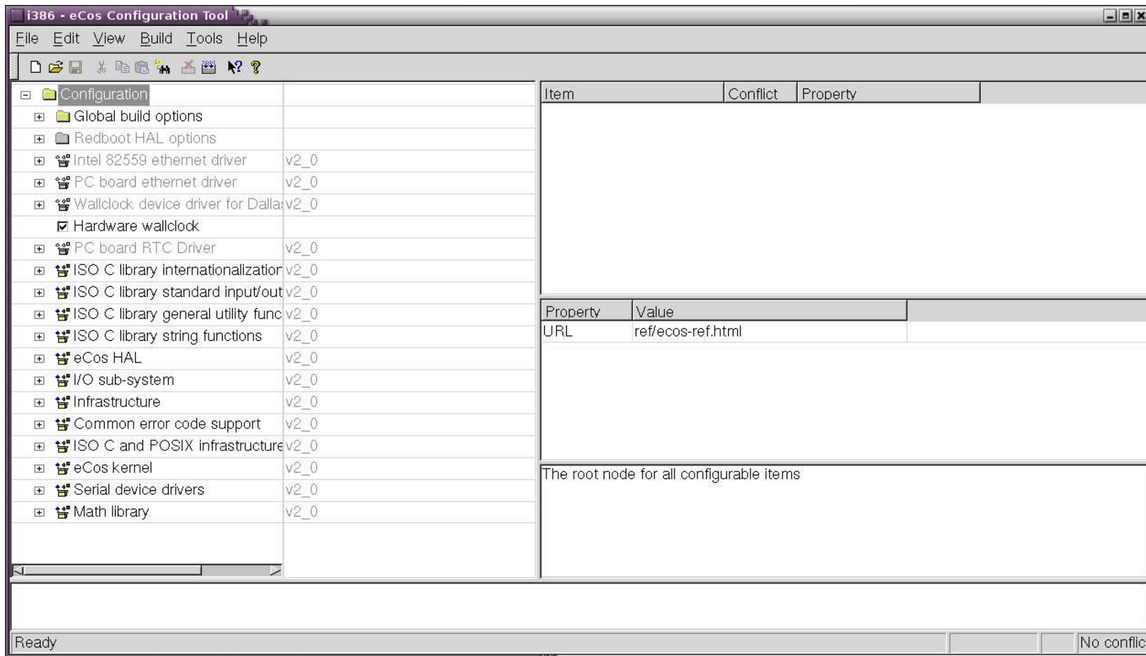

Figura 4.2: Tela da inicial da ferramenta de configuração.

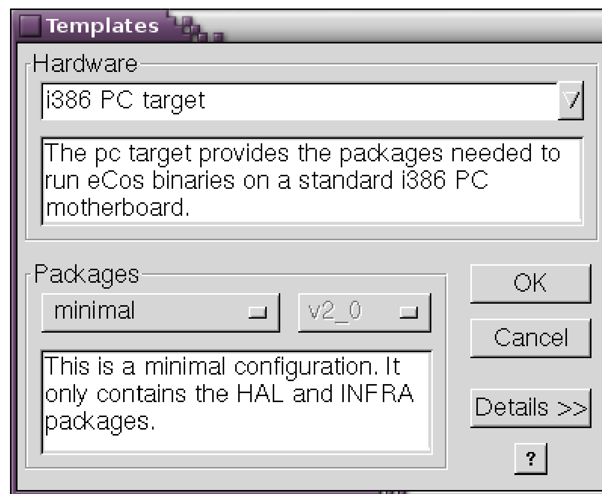

Figura 4.3: Janela para a escolha do *template* e pacotes.

No menu "*Build – Package*" (Figura 4.4) é possível adicionar ou remover quaisquer componentes necessários à aplicação. Para o estudo de caso, além do pacote *minimal* foram adicionados na configuração: *Kernel, ISO C standard I/O functions, ISO C Internationalization, ISO C General utility functions, ISO C string functions, serial device drivers* e *Math library.*

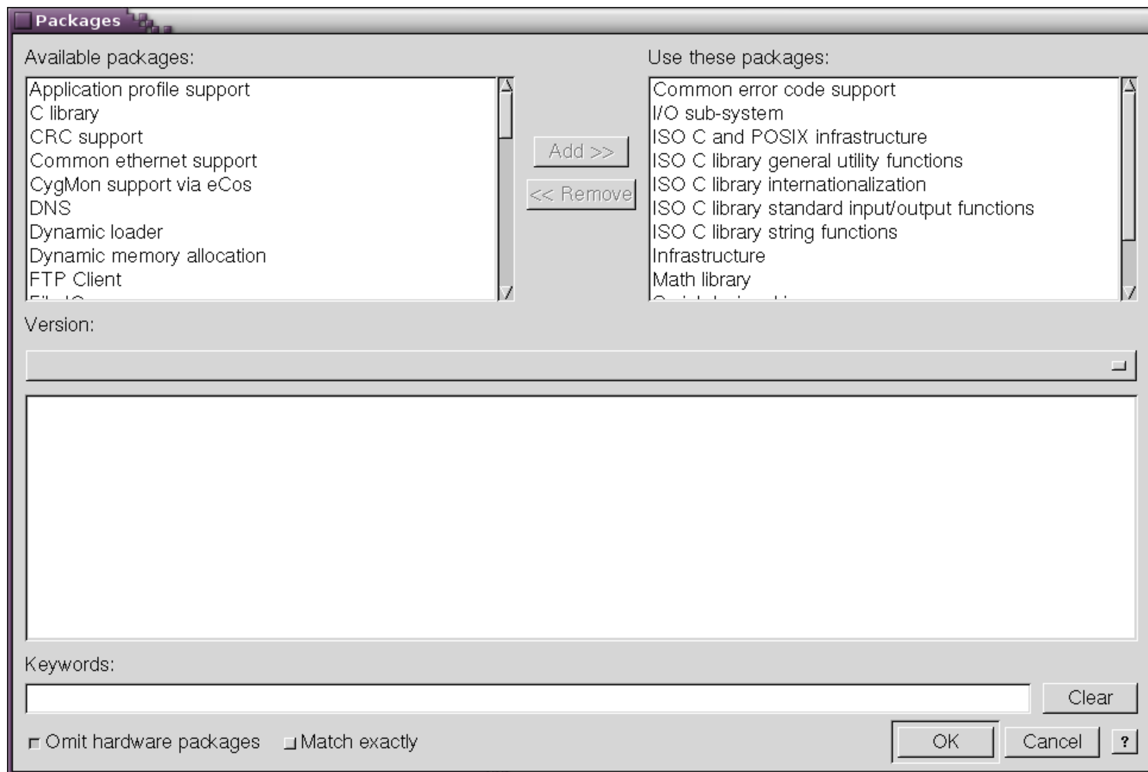

Figura 4.4: Janela para seleção de novos pacotes.

Durante a seleção dos pacotes que farão parte da plataforma, problemas de conflitos entre os componentes podem ser encontrados. Um exemplo de conflito é quando ocorre a falta de uma biblioteca para atender as necessidades de um determinado pacote (exemplo prático: o Item CYGPKG_LIBC_STDIO requer a biblioteca stdio.h, a qual não foi selecionada através da configuração). O *eCos Configuration Tool* possui uma ferramenta que detecta e propõe soluções para esses conflitos. O menu "*Tools – Resolve Conflicts*" (Figura 4.5) mostra o item que está em conflito e a proposta para a sua solução (continuando o exemplo prático: a proposta é ativar a biblioteca stdio.h).

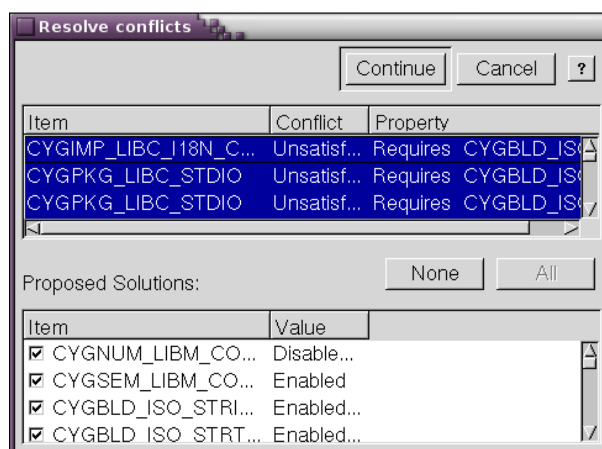

Figura 4.5: Janela do *Resolve Conflicts*.

Finalmente, depois de configurados a plataforma, os pacotes, os endereços (caminhos para encontrar os pacotes), e ainda resolvidos todos os conflitos, deve ser criada a biblioteca do eCos. O menu "*Build – Library*" aciona o processo que monta a árvore com os códigos e gera a biblioteca (libtarget.a) para a plataforma configurada.

A aplicação implementada deverá ser compilada com as ferramentas disponibilizadas pelo eCos, e ligada à biblioteca e à árvore com os códigos configurados e gerados pela ferramenta de configuração. Através da compilação, um arquivo binário com extensão "*.out*" é gerado. Esse arquivo é carregado no simulador (Simics, Seção 4.2) através do *boot loader* gerado, também, pela ferramenta de configuração do SO.

4.1.1 Tratamento de exceção do eCos

Uma exceção é um evento síncrono, que ocorre durante a execução de uma *thread*, e muda o fluxo normal das instruções do programa. Se a exceção não for devidamente processada durante a execução do programa, diversas conseqüências, como falhas, por exemplo, podem ocorrer (MASSA, 2002). O tratamento da exceção é extremamente importante, especialmente em sistemas embarcados, para evitar estas falhas, e proporcionar robustez ao software.

Em um sistema, exceções podem ser ocasionadas por hardware (por exemplo, erro de acesso à memória) ou por software (por exemplo, divisão por zero). A exceção causa uma interrupção, pois o processador vai para um endereço definido (ou um vetor de exceção) e começa a executar as instruções que ali estão localizadas. O endereço para o contador de programa é desviado contém o tratamento da exceção para processar o erro (MASSA, 2002).

Para sistemas embarcados, o método mais simples e flexível para tratamento de exceção é a chamada de funções, as quais precisam de um contexto ou uma área para trabalhar.

No eCos há duas formas de se implementar o tratamento de exceção. A primeira é a combinação do HAL e do tratamento de exceção do *Kernel* (Figura 4.6- caminho cinza). A segunda forma (Figura 4.6- caminho branco) consiste no tratamento da exceção pela aplicação, o qual permite que a aplicação tenha o total controle sobre as exceções (MASSA, 2002). A Figura 4.6 apresenta o fluxo do tratamento de exceção do eCos. O que determina o caminho que será percorrido para o tratamento de exceção é a configuração do eCos. Para o estudo de caso foi considerado o método de combinação do HAL e do tratamento de exceção do *Kernel*. É importante ressaltar que, para esse tipo de tratamento, cada exceção suportada pelo processador deve ser instanciada. Se um tipo de tratamento não for instanciado para uma exceção, por exemplo, divisão por zero, e essa exceção vier a ocorrer durante a execução do programa, o processador saltará para um endereço para começar a executar o tratamento, mas, no entanto, não irá encontrar nenhum código para executar. Esse fato pode causar um comportamento errôneo, que poderá levar a ocorrência de falhas, e ainda ser muito difícil de depurar.

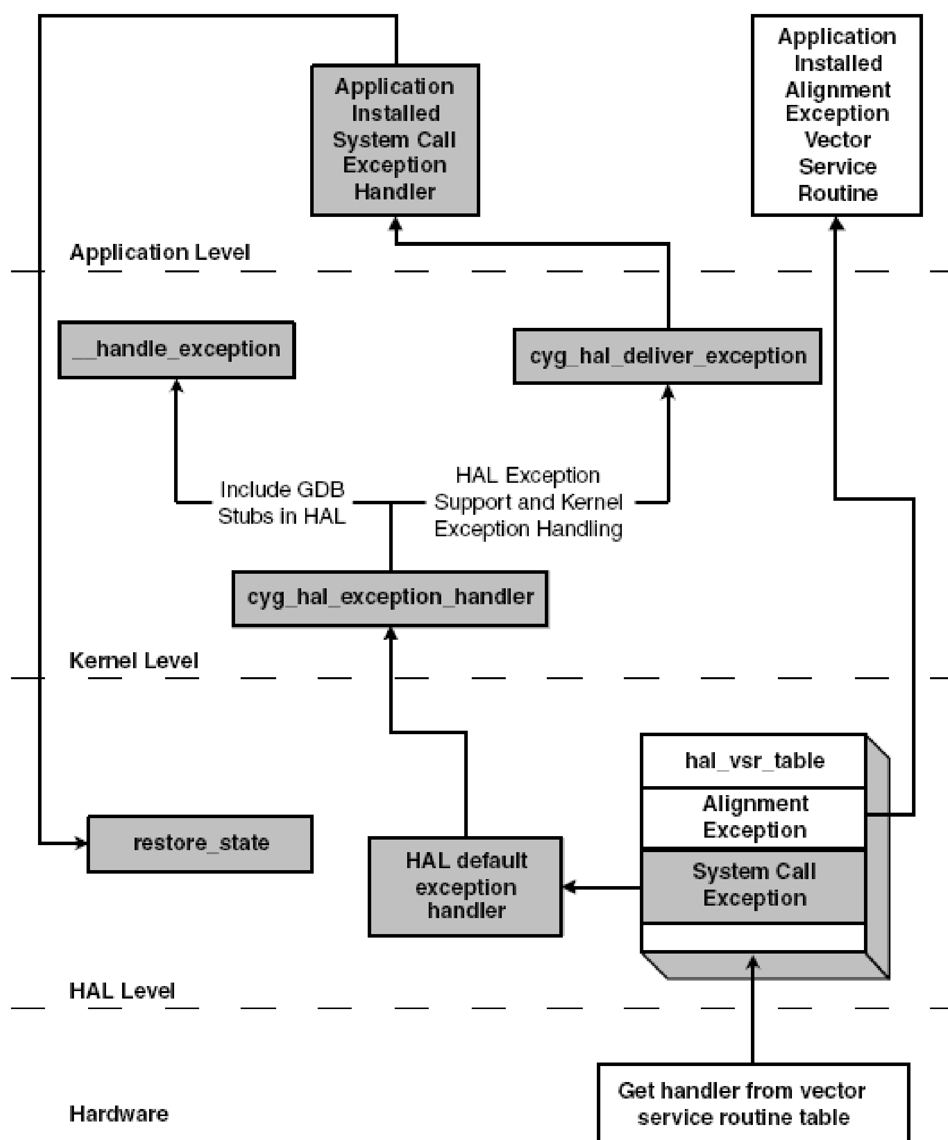

Figura 4.6: Fluxo do Tratamento de Exceção do eCos (MASSA, 2002).

A Figura 4.7 apresenta o diagrama do fluxo do tratamento de exceção implementado por este trabalho, seguindo a configuração definida para o sistema operacional eCos (plataforma i386 *PC target* i386, *template minimal,* mais os pacotes*: Kernel, ISO C standard I/O functions, ISO C Internationalization, ISO C General utility functions, ISO C string functions, serial device drivers* e *Math library.*). A Figura 4.7 está dividida em camadas; a camada da aplicação, a camada do hardware, a camada do HAL (vectors.S e Hal_misc.c) e a camada do *Kernel* (except.cxx).

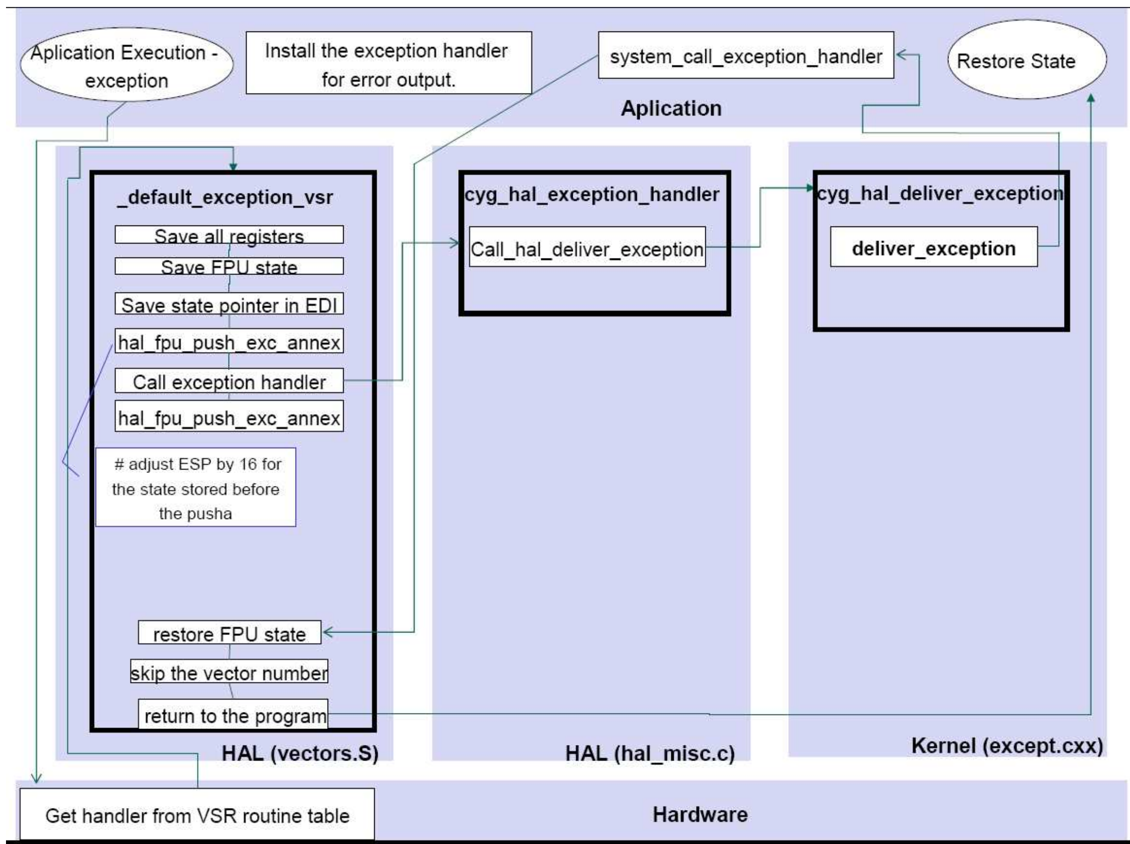

Figura 4.7: Fluxo do tratamento de exceção implementado (BEQUE et al, 2009).

O HAL do eCos utiliza uma tabela, a *Vector Service Routine* (VSR) (Figura 4.7 - camada do hardware). Essa tabela consiste em um *array* de ponteiros para o tratamento de determinadas exceções. A tabela VSR é o lugar onde o processador (hardware) procura para determinar o endereço da rotina que contém o tratamento da exceção. O tamanho e o endereço base da VSR é específico para cada arquitetura.

O HAL do eCos garante que o HAL *default exception* VSR (Figura 4.7 – camada do HAL (vectors.S)) seja instanciado durante a inicialização do sistema para cada exceção suportada pela arquitetura. A tarefa do *default exception* VSR é realizar um tratamento comum a todas as exceções, salvar o estado do processador, fazer chamadas para o nível do *Kernel*, restaurar o estado do processador, e voltar à execução normal do programa.

Após o HAL concluir o controle do processamento comum da exceção, o controle é passado para o *Kernel*. A rotina *cyg_hal_exception_handler* (Figura 4.7 – camada do HAL (Hal_misc.c)) é que faz essa chamada para a função *cyg_hal_deliver_exception* do *Kernel* (Figura 4.7 – camada do *Kernel* (except.cxx)). Para controlar as rotinas, para cada exceção, o *Kernel* do eCos define APIs (do inglês *Application Programming Interface).* Essas APIs serão apresentadas na Seção 4.3 deste Capítulo. A função *deliver_exception* do *Kernel* busca na aplicação a função *system_call_exception_handler*, que é a função chave para o tratamento da determinada exceção instanciada.

Após o tratamento da exceção, o fluxo retorna para o HAL *default exception* VSR (Figura 4.7 – camada do HAL(vectors.S)), restaura o estado do processador, e retorna para a execução normal do programa.

4.2 Simics

Simics é um ambiente de simulação de alta performance, originalmente desenvolvido pelo *Swedish Institute Computer Science* (SICS), e posteriormente, em 1998, passou a ser uma ferramenta comercial da Virtutech (VIRTUTECH, 2008). A Virtutech possui uma licença acadêmica para fins educacionais e de pesquisa. Este ambiente foi projetado para simular sistemas como AMD64, ARM, EM64T, IA-64, MIPS, PowerPC, SPARC-V8 e V9, e CPUs x86. O Simics manipula arquivos de imagem do disco da arquitetura no qual se deseja simular.

Esta ferramenta é bastante usada por analistas com o objetivo de testar determinadas plataformas antes de adquiri-las. Ela pode ser utilizada para analisar sistemas de software através de simulação, a qual se aproxima do ambiente real de execução. De Acordo com Virtutech, (2008), ela possibilita a injeção de falhas arbitrárias no sistema, através de um mecanismo de *checkpoint*.

A simulação de várias arquiteturas e de computadores de vários fabricantes, e ainda possibilidade da alteração de módulos, são características que tornam vantajoso o uso dessa ferramenta. Em cada uma das máquinas simuladas, diversas configurações podem ser alteradas, entre elas, o número de processadores, a freqüência do *clock* (MHz), tamanho da memória (MB), acesso à rede, dispositivo de *boot*, número de *floppys*.

A simulação é realizada em nível de instruções. Através da simulação pode-se ter acesso a algumas informações como o número de instruções por unidade de processamento, número de operações por segundo, número de operações por segundo em cada unidade de processamento, entre outros.

Depois de instalado e configurado pode-se inicializar a simulação da imagem da arquitetura alvo. A execução do Simics é bastante simples e direta, bastando executar o aplicativo dentro do subdiretório onde se encontra a máquina a ser simulada. A Figura 4.8 apresenta a tela de inicialização do Simics, carregando a imagem da arquitetura x86. Para o estudo de caso, o binário do *RedBoot ROM Monitor,* gerado na *eCos Configuration Tool* do eCos, é carregado através da simulação de um *floppy* (Figura 4.9).

O gerenciamento e controle do ambiente simulado são realizados através de uma janela de comandos (Figura 4.9). Em uma segunda janela (Figura 4.10), chamada console, é onde será executado o ambiente simulado (*RedBoot* – eCos).

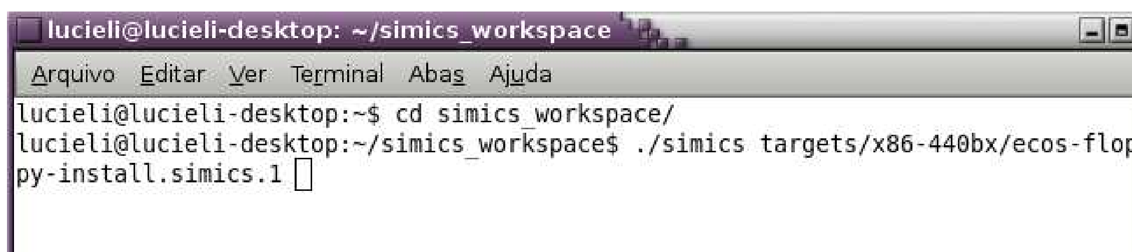

Figura 4.8: Tela de inicialização do Simics.

```
lucieli@lucieli-desktop: ~/simics_workspace
Arquivo  Editar  Ver  Terminal  Abas  Ajuda
lucieli@lucieli-desktop:~$ cd simics_workspace/
lucieli@lucieli-desktop:~/simics_workspace$ ./simics targets/x86-440bx/ecos-flop
py-install.simics.1
Checking out a license... done: academic license.

  +----------------+    Copyright 1998-2007 by Virtutech, All Rights Reserved
  |   Virtutech    |    Version: Simics 3.0.30
  |     Simics     |    Build: 1403  Host: x86-linux
  +----------------+
    www.simics.com      "Virtutech" and "Simics" are trademarks of Virtutech AB

Use of this software is subject to appropriate license.
Type 'copyright' for details on copyright.
Type 'help help' for info on the on-line documentation.


################### OS installation from floppy

Now insert floppy using:

simics> flp0.insert-floppy A <flp-file>

Where <flp-file> is the name of the floppy image.

###################

Floppy inserted in drive 'A:'. (File /opt/virtutech/simics-3.0.30/targets/x86-44
0bx/images/redboot.bin).
simics>
```

Figura 4.9: Janela de comandos do Simics.

```
Simics Console: con0

RedBoot(tm) bootstrap and debug environment [FLOPPY]
Non-certified release, version v2_0 - built 15:16:17, Sep 26 2008

Platform: PC (I386)
Copyright (C) 2000, 2001, 2002, Red Hat, Inc.

RAM: 0x00000000-0x000a0000, 0x0007bed0-0x000a0000 available
RedBoot> _
```

Figura 4.10: Console.

A seguir serão apresentadas as aplicações embarcadas usadas para a realização dos testes.

4.3 Aplicação Embarcada

Para a realização dos testes, as aplicações usadas foram o *loop* e o *bubble sort*. A aplicação *loop* consiste em um programa simples, o qual gera valores aleatórios dentro de um *loop*. O usuário entra com a quantidade de valores aleatórios que o programa irá gerar. Os valores gerados variam de 0 a 5. A aplicação *bubble sort* ordena de forma crescente um vetor de entrada com os valores {5, 4, 3, 2, 1, -5, -4, -3, -2, -1}. Com o intuito de testar a rotina de tratamento de exceção, as duas aplicações simulam exceções de divisão por zero, durante a execução.

Em ambas as implementações, foram usadas API’s do *Kernel* do eCos para instanciar o tratamento de exceção. As Figuras, 4.11 e 4.12, apresentam as funções API do *Kernel* para o tratamento de exceção inseridos nas aplicações *loop* e *bubble sort*, respectivamente.

```
//Loop

#include <cyg/kernel/kapi.h>
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <time.h>
#include <unistd.h>

int numero = 0;

//Tratamento da exceção
void system_call_exception_handler(
    cyg_addrword_t data, //dados passados para o tratamento de exceção
    cyg_code_t number, //número da exceção ("0" - divisão por zero
    cyg_addrword_t info) //ponteiro para o registrador que salva o estado da máquina no momento da exceção
{
    diag_printf( "Exception! Division by zero!\n" );
    numero = 1;

}

//Função loop
void loop () {

  float a;
  int i = 1;

    while (i<=10) {
            numero = rand()%5;
                printf("Numero%d \n", numero);
            //Tenta Gerar exceção por ZERO.
            a = 100/numero;
            i++;
        }
}

//Função Main
  void cyg_user_start(
void)
{
    cyg_exception_handler_t *old_handler;
    cyg_addrword_t old_data;
    //
    // Instancia o tratamento de exceção.
    //
    cyg_exception_set_handler(
            0, //Divisão por Zero - número da exceção
            &system_call_exception_handler, //endereço para o tratamento de exceção
            0, // dados passados para o tratamento da exceção
            &old_handler, //endereço para o tratamento de exceção anterior
            &old_data); // dados do tratamento de exceção anterior

    HAL_VSR_SET_TO_ECOS_HANDLER( CYGNUM_HAL_EXCEPTION_DIV_BY_ZERO, NULL );
        loop ();


}
```

Figura 4.11: Código da aplicação *loop* usando a API do *Kernel* para instanciar o tratamento de exceção.

```
1  //Bubble sort
2
3  #include <cyg/kernel/kapi.h>
4  #include <stdio.h>
5  #include <stdlib.h>
6  #include <math.h>
7  #include <time.h>
8  #include <unistd.h>
9
10 int numero = 0;
11
12 //Tratamento da exceção
13 void system_call_exception_handler(
14     cyg_addrword_t data,
15     cyg_code_t number,
16     cyg_addrword_t info)
17 {
18     diag_printf( "Exception! Division by zero!\n" );
19     numero = 1;
20
21 }
22
23
24 void bubble( float v[], int qtd )
25 {
26    int i;
27    int j;
28    int aux;
29    int k = qtd - 1 ;
30
31
32    for(i = 0; i < qtd; i++)
33    {
34       for(j = 0; j < k; j++)
35       {
36          if(v[j] > v[j+1])
37          {
38              aux = v[j];
39              v[j] = v[j+1];
40              v[j+1]=aux;
41          }
42       }
43       k--;
44    }
45 }
46
```

```
47 //Main
48
49 void cyg_user_start(
50         void)
51 {
52   int i=0;
53   float a;
54   float vetor[10] = {5, 4, 3, 2, 1, -5, -4, -3, -2, -1};
55   cyg_exception_handler_t *old_handler;
56   cyg_addrword_t old_data;
57
58     //
59     // Install the exception handler for error output.
60     //
61     cyg_exception_set_handler(
62             0,
63             &system_call_exception_handler,
64             0,
65             &old_handler,
66             &old_data);
67
68
69     HAL_VSR_SET_TO_ECOS_HANDLER( CYGNUM_HAL_EXCEPTION_DIV_BY_ZERO, NULL );
70
71     printf("Vetor Desordenado \n\n");
72
73     for (i = 0; i < 10; i++) {
74          printf("Posicao: %d \n", i);
75          printf("Valor: %.1f \n", vetor[i]);
76
77     }
78     i = 0;
79
80     bubble(vetor, 10);
81
82     printf("Vetor Ordenado\n\n");
83
84     for (i = 0; i < 10; i++)  {
85          printf("Posicao: %d \n", i);
86          printf("Valor: %.1f \n", vetor[i]);
87          a = 100/numero;
88     }
89
90
91 }
```

Figura 4.12: Código da aplicação *bubble sort* usando a API do *Kernel* para instanciar o tratamento de exceção.

Analisando a Figura 4.11, pode-se observar na linha 13, a rotina para o tratamento da exceção *system_call_exception_handler*. O parâmetro *cyg_addrword_t data* (linha 14) são os dados passados para o tratamento de exceção. *Cyg_code_t number* (linha 15) é o número da exceção (no estudo de caso o valor passado é “0”, que indica a exceção de divisão por zero). *Cyg_addrword_t info* (linha 16) é o ponteiro para o registrador que salva o estado da máquina. Na linha 47, a função *cyg_exception_set_handler* instancia o tratamento de exceção. Seus parâmetros são: 0 (linha 48), número da exceção (0); *&system_call_exception_handler* (linha 49), é o endereço para o tratamento de exceção; 0 (linha 50), dados passados para o tratamento da exceção (no estudo de caso, nenhum dado está sendo passado); *&old_handler* (linha 51), endereço para o tratamento de exceção anterior; *&old_data* (linha 52), dados do tratamento de exceção anterior.

4.4 Resumo e Conclusão

Este Capítulo apresentou a definição da plataforma (Figura 4.13), que consiste na primeira etapa para a realização dos testes em um SOE.

A escolha do SO eCos, ocorreu devido à sua configurabilidade. Para se obter um maior controle, e o gasto de tempo reduzido na análise do código, a configuração mínima do eCos foi a selecionada. A arquitetura selecionada no eCos foi a i386, pois ao decorrer do trabalho, foram encontrados diversos problemas ao tentar simular arquiteturas como o MIPS, ARM e PowerPC no simulador selecionado. Dentre os problemas encontrados, o principal foi a falta de instruções, no Simics, geradas pelo eCos para as arquiteturas.

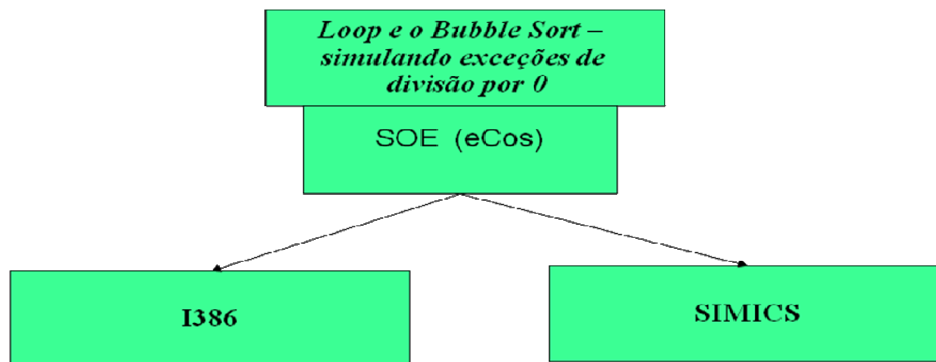

Figura 4.13: Plataforma definida para o estudo de caso.

O Simics foi o simulador usado devido às características apresentadas, como exemplo, o *checkpoint*. Devido ao fato de que a rotina de tratamento de exceção do SO possui uma grande interação com o hardware, as aplicações selecionadas foram desenvolvidas usando a API do *Kernel* para o tratamento de exceção de divisão por zero.

Após o estudo bibliográfico juntamente com a análise dos trabalhos relacionados, e também, com a etapa de definição da plataforma, chegaram-se as primeiras considerações referentes ao teste de um SOE:

- ✓ Dificuldade de montar uma plataforma de simulação, para a realização dos testes, antes da produção.
- ✓ Testes puramente funcionais em um SO não são suficientes. Geralmente eles apresentam a mesma mensagem de erro, dificultando a localização da falha, gastando um tempo maior em depuração, conseqüentemente, afetando algumas características básicas de um sistema embarcado, como exemplo, o tempo de lançamento do produto no mercado;
- ✓ Análise estrutural de Código - devido a configurabilidade do SOE eCos foi preciso, antes de planejar os teste, realizar uma análise, uma "limpeza" do código a ser testado (Figura 4.7). A configurabilidade do eCos, que para o projeto aparece como uma característica importante, para os testes é uma dificuldade extra.
- ✓ A análise estrutural, "limpeza" do código de acordo com a configuração, o código do eCos volta a ser seqüencial. O código interno é basicamente seqüencial, sendo que a verificação acontece em determinados trechos do código.

Seguindo na linha de pesquisa do teste de SOE como uma maneira de atingir a interação entre hardware e software, e ainda o fato de que um SO possui rotinas de baixo nível, as quais requerem um procedimento de teste mais específico, este trabalho apresenta no próximo Capítulo, o planejamento e avaliação de testes para o fluxo de tratamento de exceção configurado, do sistema operacional eCos.

5 PLANEJAMENTO E AVALIAÇÃO DE TESTES PARA A ROTINA DE TRATAMENTO DE EXCEÇÃO DO SISTEMA OPERACIONAL eCos

Este Capítulo tem o objetivo de apresentar o planejamento e a avaliação de testes executados para o fluxo do tratamento de exceção do SOE eCos.

A partir da configuração do SOE eCos, incluindo características para o tratamento de exceção através da combinação do HAL e do *Kernel*, e da análise e desenvolvimento do fluxo do tratamento de exceção de divisão por zero, os testes foram planejados. A partir da análise estrutural do fluxo percorrido para o tratamento da exceção de divisão por zero, as falhas para serem injetadas foram planejadas e definidas, com o intuito de expor o código a erros (Seção 5.1). A Seção 5.2 apresenta os resultados da injeção das falhas. A partir da análise das falhas e dos resultados obtidos, casos de testes foram gerados (Seção 5.3), com base em técnicas e estratégias citadas pela Engenharia de Software para o teste de softwares tradicionais. Os casos de testes foram planejados com o intuito de detectar as falhas injetadas.

5.1 Definição do Modelo de Falhas

Após a definição do fluxo do tratamento de exceção configurado do eCos (Figura 4.7), o conjunto de falhas foi definido. Como apresentado na Seção 4.1, o SOE eCos é descrito em diversas camadas, e esse fato foi levado em consideração ao ser definido o modelo de falhas. As falhas foram injetadas em duas camadas do SOE: na camada do HAL (arquivos: vectors.S, hal_misc.c) e na camada do *Kernel* (except.cxx).

Quatro tipos de falhas foram definidos considerando as diferentes camadas do sistema (localizações das falhas): em nível de código (Assembly ou código de alto nível(C, C++)), falhas de interação (hardware-software e software-hardware), e também falhas de hardware (*stuck-at*). Os quatro tipos de falhas são citados a seguir:

- Falhas de código – mudanças no código do programa (Assembly ou código alto-nível). Incluindo, por exemplo, uma falha no código (troca na operação ou operador) ou omissão de código.
- Falhas de PC (do inglês *Program Counter*) – Falhas no contador de programa. Este tipo de falha pode ser uma falha de hardware (*stuck-at*, modificação de um bit no registrador) ou uma falha de software (erro na execução de alguma tarefa, por exemplo, quando no código um endereço específico é alterado. O valor colocado como falha é sempre um valor válido entre os endereços alocados para o programa).
- Falhas de registradores – mudança de registradores. Novamente, pode ser uma falha de software ou hardware.

- Falhas de parâmetros – alteração ou exclusão de algum argumento na chamada de alguma rotina.

A Figura 5.1, apresenta a localização, o tipo e o número de falhas injetadas. O processo de injeção de falhas foi realizado de maneira manual, sendo injetada uma falha por vez. Falhas foram inseridas na aplicação, no HAL do eCos e ainda no *Kernel* do eCos. A cada falha injetada, foi necessário recompilar o eCos e a aplicação

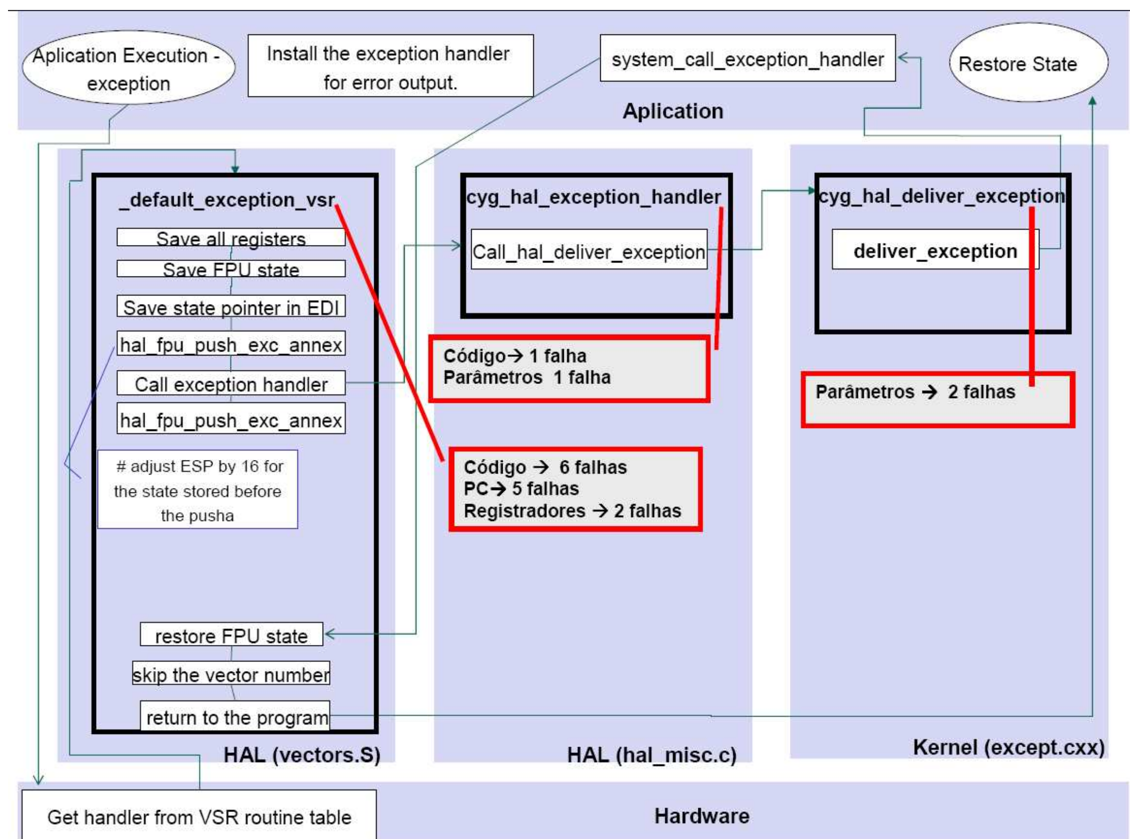

Figura 5.1: Fluxo do tratamento de exceção implementado com as falhas injetadas.

5.2 Resultado da Injeção de Falhas

A Figura 5.2 apresenta os resultados da injeção de falhas. Ao total foram 17 falhas injetadas. Essas falhas foram injetadas ao longo do código de tratamento de exceção, nas camadas do HAL e do *Kernel*. Os testes foram realizados executando as duas aplicações citadas na Seção 4.3. Os resultados obtidos foram os mesmos para as duas aplicações embarcadas que geram a exceção de divisão por zero. Através dos resultados da injeção de falhas foram observadas duas possíveis saídas para uma falha: falhas com efeito catastrófico (onde o programa pára a execução depois de acontecer o tratamento da exceção) e falhas com efeito não catastrófico (onde a exceção é tratada e o programa volta para seu fluxo normal).

Em geral, falhas com efeito catastrófico conduzem a uma GPF (*Global Protection Fault*) com uma mensagem genérica do SOE. Falhas com efeitos não catastróficos não afetam o comportamento do sistema, ou seja, o fluxo do tratamento de exceção é executado corretamente, o programa volta à execução normal sem a presença de erros, com exceção de duas falhas do tipo PC.

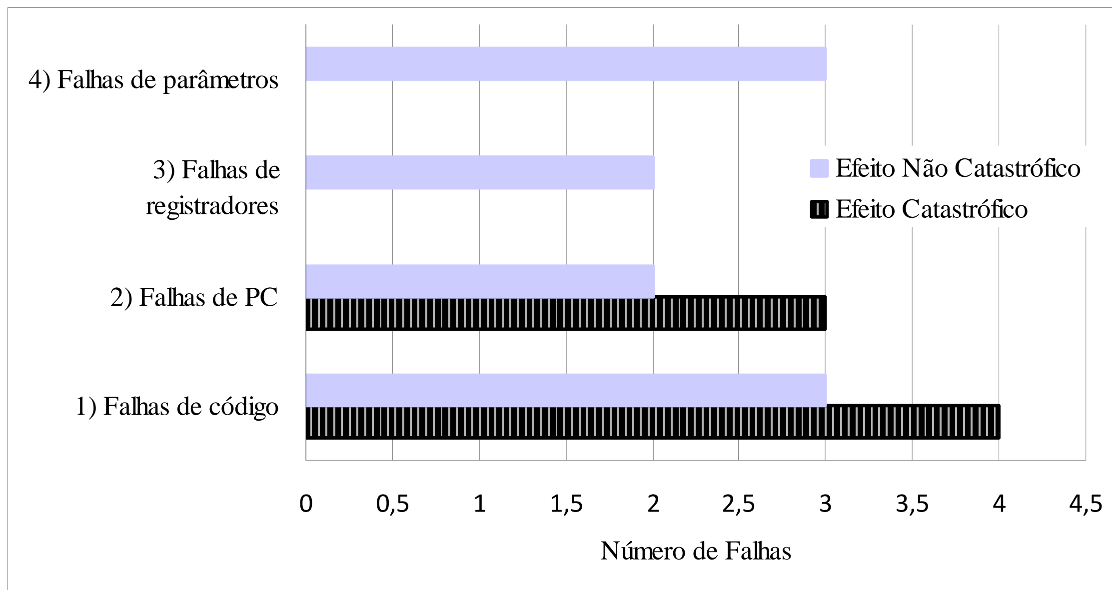

Figura 5.2: Resultado da injeção de falhas.

Na Figura 5.2 pode-se observar que 7 de 17 falhas (41%) possuem efeito catastrófico. Estas falhas envolvem ou afetam registradores especiais que armazenam o estado do sistema, e são geralmente relacionadas a violações de memória.

Dentre as falhas de efeito não catastrófico, somente duas falhas de PC levaram a um comportamento incorreto do sistema. O valor do contador de programa apontava para o endereço do início da aplicação, ou para a primeira instrução depois do reset. Nesses casos o sistema continua operando, porém a aplicação é executada em mais ciclos e os resultados apresentam saídas extras.

É importante salientar que a maioria das falhas com efeito catastrófico conduz para a mesma mensagem de erro (*Global Protection Fault*). Esse fato torna difícil a localização da falha.

Como se pode observar através da Figura 5.1, a maioria das falhas foram injetadas na camada HAL (vectors.S), totalizando 13 falhas (59%). Este número é explicado pelo fato que esta é a camada de inicialização do SOE eCos, é o primeiro código executado pelo processador. Ainda, é a camada que possui mais código implementado.

A Tabela 5.1 apresenta um resumo do resultado da injeção de falhas, relacionando aos efeitos gerados.

Tabela 5.1: Relação entre cada caso de teste e o tipo de falhas detectadas.

41%	Efeito catastrófico	a execução do programa pára
12%	Efeito não catastrófico + programa com resultado incorreto	a execução do programa não pára, porém o resultado é errado
47%	Efeito não catastrófico programa não pára + o resultado é correto	a execução do programa não pára, e o resultado é correto

Através da ferramenta de configuração do eCos, pode-se criar suítes de teste (testes puramente funcionais) para a configuração selecionada, no caso a mínima. Os testes

gerados e relacionados ao teste do tratamento de exceção do tipo combinação do HAL e do tratamento de exceção do *Kernel*, foram executados no simulador Simics, e apresentaram os mesmos efeitos analisados pela execução das duas aplicações embarcadas deste trabalho. Um ponto a ressaltar sobre estes suítes de teste gerados pela ferramenta de configuração, é que eles não detectam e nem diagnosticam nenhuma das falhas injetadas.

Após a injeção das falhas, para as falhas do efeito catastróficas acredita-se que o hardware sobrescreve o valor correto mesmo após a injeção da falha no software do SOE. Devido a isso, e ao fato que testes puramente funcional não são suficientes para testar um SOE, o próximo passo foi à inspeção do software, com a inserção de casos de teste ao decorrer do fluxo implementado (instrumentação de código).

5.3 Definição dos Casos de Teste

A partir da injeção de falhas e da análise dos resultados, quatro casos de testes foram inseridos no fluxo de tratamento de exceção configurado do SOE eCos, para ajudar na detecção e diagnóstico das falhas. A Figura 5.3 apresenta a localização dos casos de teste no fluxo de tratamento de exceção implementado.

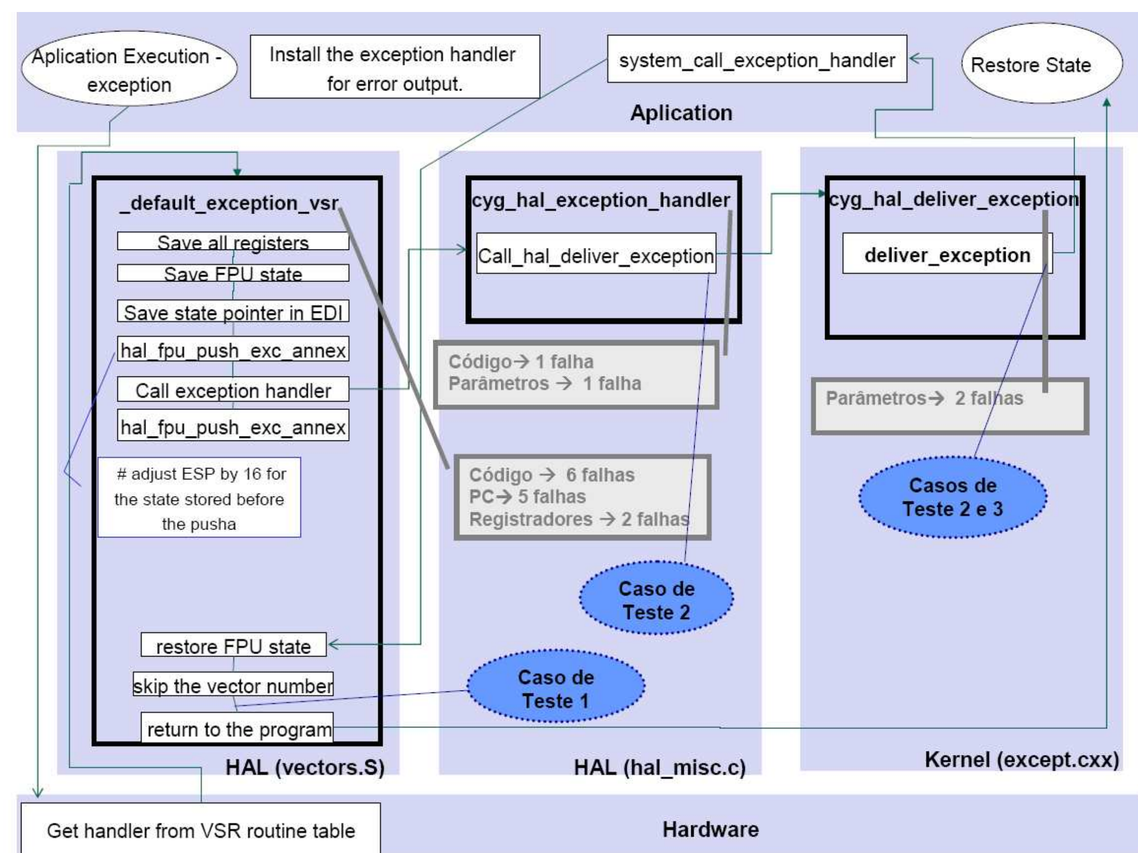

Figura 5.3: Fluxo do tratamento de exceção implementado com os casos de teste.

Em geral, os casos de testes foram definidos para detectar e localizar as falhas injetadas que obtiveram efeitos catastróficos. Para o desenvolvimento dos casos de testes, técnicas de teste estruturais foram utilizadas. Para exercitar os casos de teste a estratégia utilizada para a análise foi o teste de verificação. O uso desta estratégia visa verificar a eficiência e eficácia dos casos de teste.

Um ponto a ressaltar, é o fato de que nenhuma ferramenta de teste tradicional fornece suporte para detectar e diagnosticar falhas no SOE, pois para simular é preciso, também, ter a simulação do hardware, pois o teste é dependente do mesmo.

A seguir será apresentada a configuração dos casos de teste.

Caso de Teste 1:

Entradas

Condição Inicial:

- Aplicação gerar uma exceção de divisão por zero

Passos:

- Técnica de teste de software padrão usada – teste de caixa branca ou estrutural. Caso de teste foi criado e inserido após o conhecimento da estrutura lógica do trecho de código para o tratamento da exceção.
- Este caso de teste verifica se contador de programa é válido, ou seja, se o valor está entre o primeiro e último endereço alocados para a aplicação. Ele é executado antes de chamar a rotina de tratamento de exceção, nas camadas seguintes.
- O caso de teste 1 está localizado no ponto inicial para todos os pacotes do HAL, na camada HAL (vectors.S).

Saídas

Resultados esperados:

- Implementado na linguagem Assembly, foram inseridos para detectar as falhas com efeito catastrófico na camada HAL (vectors.S), as quais causam um GPF**.**

Caso de Teste 2:

Entradas

Condição Inicial:

- Aplicação gerar uma exceção de divisão por zero

Passos:

- Técnica de teste de software padrão usada – teste de caixa branca ou estrutural. Caso de teste foi criado e inserido após o conhecimento da estrutura lógica do trecho de código para o tratamento da exceção.
- O ponteiro para a estrutura *HAL_SavedRegisters,* a qual contêm o estado da máquina, é o ponto chave para este teste. A estrutura *HAL_SavedRegisters* sempre é salva para um endereço de memória posterior ao endereço final do programa. O caso de teste verifica se o ponteiro possui um valor válido.
- O caso de teste 2 está localizado na camada HAL (hal_misc.c).

Saídas

Resultados esperados:

- Implementado na linguagem C, o resultado esperado é detectar as falhas de parâmetros passados para as funções do tratamento de exceção no nível do HAL (hal_misc.c).

Caso de Teste 3:

Entradas

Condição Inicial:

- Aplicação gerar uma exceção de divisão por zero

Passos:

- Técnica de teste de software padrão usada – teste de caixa branca ou estrutural. Caso de teste foi criado e inserido após o conhecimento da estrutura lógica do trecho de código para o tratamento da exceção.
- O ponteiro para a estrutura *HAL_SavedRegisters,* a qual contêm o estado da máquina, é o ponto chave para este teste. A estrutura *HAL_SavedRegisters* sempre é salva para um endereço de memória posterior ao endereço final do programa. O caso de teste verifica se o ponteiro possui um valor válido.
- O caso de teste 3 está localizado na camada *Kernel* (except.cxx).

Saídas

Resultados esperados:

- Implementado na linguagem C++, o resultado esperado é detectar as falhas de parâmetros passados para as funções do tratamento de exceção no nível do *Kernel* (except.cxx).

Caso de Teste 4:

Entradas

Condição Inicial:

- Aplicação gerar uma exceção de divisão por zero

Passos:

- Técnica de teste de software padrão usada – teste de caixa branca ou estrutural. Caso de teste foi criado e inserido após o conhecimento da estrutura lógica do trecho de código para o tratamento da exceção.
- Visa as falhas que possam ocorrer no parâmetro que contém o número da exceção ocorrida.
- O caso de teste 3 está localizado na camada *Kernel* (except.cxx).

Saídas

Resultados esperados:

- Implementado na linguagem C++, o resultado esperado é o "0" (zero). (que indica a exceção de divisão por zero).

A Tabela 5.2 mostra a relação entre cada caso de teste e o tipo de falhas detectadas. A Tabela 5.3 apresenta um resumo dos casos de teste definidos.

Tabela 5.2: Relação entre cada caso de teste e o tipo de falhas detectadas.

	Falhas de código	**Falhas de PC**	**Falhas de parâmetros**
Caso de teste 1	X	X	
Caso de teste 2			X
Caso de teste 3			X
Caso de teste 4			X

Tabela 5.3: Resumo dos casos de teste definidos.

Caso de tese	**Camada**	**Linguagem**	**Foco**
1	HAL (vectors.S)	Assembly	Violação de memória
2	HAL (hal_misc.c)	C	Ponteiro para a estrutura HAL_SavedRegisters
3	*Kernel* (except.cxx)	C++	Ponteiro para a estrutura HAL_SavedRegisters
4	*Kernel* (except.cxx)	C++	Número da exceção

Através da execução e análise dos resultados dos casos de teste, pode-se afirmar que o caso de teste 1 poderá ser reusado para outras exceções, e não somente para a exceção de divisão por zero, como também para o fluxo de tratamento de interrupções.

O principal foco dos casos de teste foi para falhas inseridas no SOE. Para falhas inseridas na camada da aplicação ou na rotina de tratamento de exceção instanciado pela aplicação deverão ser detectadas usando casos de teste desenvolvidos para a aplicação.

5.4 Resultados dos testes

Para a obtenção dos resultados, as aplicações foram re-executadas com os casos de testes inseridos.

Com a execução dos casos de testes, das 17 falhas injetadas, os casos de testes conseguiram detectar 13 falhas (76%). Dentre as 4 falhas não detectadas, 2 foram as falhas de PC, aquelas que levaram a um comportamento incorreto do sistema. E as outras 2, foram as falhas de registradores. As falhas de registradores, além de não serem detectadas, sofreram um efeito não catastrófico, isto ocorreu, pois o hardware

sobrescreve o valor correto, mesmo após a injeção da falha no software do SOE. Justamente por esse motivo, não foi desenvolvido um caso de teste para detectar esse tipo de falha.

Um ponto a ressaltar, que os testes estruturais ajudam na localização das falhas. A aplicação pára sua execução, porém garante o diagnóstico.

5.5 Resumo e Conclusão

Este Capítulo apresentou os experimentos realizados para o fluxo do tratamento de exceção divisão por zero do SOE eCos. Falhas foram injetadas nas camadas HAL (vectors.S), HAL (hal_misc.c) e *Kernel* (except.cxx). A partir da análise das falhas, casos de testes foram desenvolvidos usando a técnica de teste estrutural, partindo do princípio que os testes funcionais disponíveis pelo eCos não detectaram as falhas. Para a execução e análise dos resultados dos casos de testes, a estratégia de teste de verificação foi utilizada.

6 CONCLUSÃO

Nesta Dissertação foram apresentados os passos iniciais em busca da solução para resolver o desafio do teste de interação entre o hardware e software. De encontro à complexidade atual encontrada nos sistemas embarcados, e ao uso de um Sistema Operacional Embarcado (SOE) para gerenciar os componentes da aplicação e a interação software/hardware, este trabalho propôs um método para testar a rotina de tratamento de exceção, que possui uma alta interação software/hardware, do SOE eCos.

O estudo apresentado sobre testes de software tradicional (Capítulo 2) foi de relevada importância para o planejamento e execução dos testes na rotina de tratamento de exceção do eCos.

Em relação ao teste do software embarcado, normalmente eles são realizados sem planejamento e sem documentação. O grande desafio está no teste de interação software/hardware, sendo que as falhas de interação precisam ser consideradas na etapa de teste. Metodologias existentes, e propostas pela Engenharia de Software para softwares tradicionais, não se preocupam com características especiais, como disponibilidade e confiabilidade, de um software embarcado. Sendo assim, atualmente ainda não existe uma metodologia genérica para testar um software embarcado como um todo.

Para este estudo inicialmente foi preciso definir uma plataforma (hardware + software) para a realização dos experimentos. Após a definição da plataforma, os testes foram realizados e o conjunto de critérios (requisitos) para testar um Sistema Operacional Embarcado foi definido.

Partindo do princípio que ainda existe o desafio (teste de interação) relacionado ao teste de um software embarcado, este trabalho apresentou os passos iniciais para testar a rotina de tratamento de exceção de um SOE (eCos). O método de teste planejado e executado para a rotina de tratamento de exceção pode ser utilizado como base para o desenvolvimento de uma metodologia genérica para o teste de um software embarcado, pois essa rotina apresenta um alto grau de interação software/hardware.

A partir da definição da plataforma e análise estrutural do fluxo percorrido para o tratamento da exceção de divisão por zero, as falhas para serem injetadas foram planejadas e definidas, com o intuito de expor o código a erros. Foram definidos quatro tipos de falhas considerando as diferentes localizações das falhas (camadas do sistema), em nível de código (Assembly ou código de alto nível(C, C++)), falhas de interação (hardware-software e software-hardware), e também falhas de hardware (*stuck-at*). Casos de testes foram definidos para detectar e localizar as falhas injetadas que obtiveram efeitos catastróficos. Para o desenvolvimento dos casos de testes, técnicas estruturais foram utilizadas.

Através de todo o percurso do trabalho, pôde-se notar que a configurabilidade do eCos, que inclusive foi a principal característica para a escolha desse SOE para o estudo de caso, pode apresentar bons resultados para o projeto de um sistema embarcado, mas para os testes, ela representa uma dificuldade extra. Os casos de testes precisam ser específicos, e a técnica de instrumentação de código se faz necessária. De fato, o código disponível do eCos é totalmente genérico com várias diretivas de compilação e macros. Sendo assim, é preciso extrair o código que será realmente compilado antes do planejamento dos testes, e esta tarefa pode consumir muito tempo, devido a hierarquia de código. Além do tempo, isto torna a execução dos testes totalmente dependente do hardware e da configuração selecionada. Importante salientar que a execução dos testes são dependentes do hardware e da plataforma a ser utilizada, pois é ele quem dispara a exceção, tornando inviável o uso de ferramentas para teste de software tradicional.

Para o estudo de caso particular apresentado neste trabalho, ou seja, teste do fluxo de tratamento da exceção de divisão por zero, usando a configuração da plataforma i386 PC target i386, *template minimal* e mais os pacotes *Kernel*, ISO C *standard* I/O *functions*, ISO C *Internationalization*, ISO C *General utility functions*, ISO C *string functions, serial device drivers e Math library*, os resultados foram satisfatórios, pois com a execução dos casos de testes 76% falhas injetadas, foram detectadas.

Com esse estudo inicial, apresentado por este trabalho, podem-se elencar os seguintes requisitos fundamentais para a realização de testes em um SOE, visando uma metodologia genérica que tenha a preocupação com a interação software/hardware:

- Necessidade da presença do hardware (plataforma) para o planejamento e execução dos testes em um SOE.
  - Dificuldade em montar um ambiente de simulação para a realização dos testes.
- Necessidade de visualizar o código configurado, para o início do planejamento e execução dos testes.
  - Configurabilidade do SOE apresentou uma dificuldade extra para os testes.
- Necessidade de instrumentação de código (casos de teste) para a detecção e diagnósticos das falhas.
  - Dificuldade em encontrar as falhas, executando testes puramente funcionais em SO.

Após o estudo de caso definido e estudado por este trabalho, juntamente com a definição e injeção de falhas, foram definidos os seguintes passos, genericamente, para o teste de um SOE:

- Definição e implementação da Plataforma de Simulação + Processador
- Definição de funcionalidades do SO passíveis de serem testadas isoladas
- Determinação e estudo do fluxo, caminho do código a ser percorrido (análise estrutural e “limpeza” do código)
- Definição de um mecanismo de injeção de falhas
- Planejamento e execução dos testes

A idéia central do trabalho foi o estudo e apresentação dos passos iniciais em direção a uma solução para resolver o desafio do teste de interação entre o software e o hardware. Seguindo essa idéia, como trabalhos futuros têm-se:

- Definir previamente um modelo de hardware que poderá ser usado.
- Descobrir alguma forma de configurar o SOE para gerar um código intermediário, a fim de que se torne o código padrão para o teste (mediação do teste).
- Estender os testes e o método realizado neste trabalho, para outras rotinas do SOE, além da rotina de tratamento de exceção.
- Realização da definição de um modelo de falhas mais completo, e seus correspondentes casos de teste.
- Buscar um meio de sistematizar o processo
    - Ferramentas injeção de falhas
    - Gerar código intermediário
- Desenvolvedor do SO, prever e inserir no decorrer da implementação (desenvolvimento) trechos de códigos para a inspeção.

7 REFERÊNCIAS

BARR, G. P. **Programming Embedded Systems in C and C++.** Sebastopol, USA: O'Reilly & Associates, 1999.

BEQUE, L. T.; DAI PRA, T.; COTA, E. Testing Requirements for an Embedded Operating System: the Exception Handling Case Study. In: IEEE LATIN-AMERICAN TESTWORKSHOP, 10, 2009, LATW 2009, Buzios, Brasil. **Proceedings…** [S.l.: s.n.], 2009.

BROEKMAN, B.; NOTENBOOM, E. **Testing Embedded Software**. London: Addison-Wesley, 2003.

BURNS, A.; WELLINGS, A. **Real-Time Systems and Programming Languages.** London: Addison-Wesley, 1997.

CARRO, L.; WAGNER, F. Sistemas Computacionais Embarcados. In: **Jornadas de Atualização em Informática**. Campinas, Brasil: Sociedade Brasileira de Computação, 2003. p.45-94.

CARTAXO, E. G. **Geração de Casos de Teste Funcional para Aplicações de Celulares.** Dissertação (Ciência da Computação) – Universidade Federal de Campina Grande, Campina Grande – Paraíba, 2006.

COTA, E.; LUBASZEWSKI, M. Reuse-Based Test Planning for Core-Based Systems-on-Chip. In: WORKSHOP DE TESES E DISSERTAÇÕES, 2004, Salvador, BRA. **Proceedings**.., [S.l.: s.n.]. v. 1. p. 64-64, 2004.

DIAS NETO, A. C.; TRAVASSOS, G. H. Towards a Computerized Infrastructure for Software Testing Planning and Control. In: LATW - LATIN AMERICAN TEST WORKSHOP, 6, 2005, LATW 2005. Salvador, Brasil. **Proceedings…** [S.l.: s.n.], 2005.

eCos. Disponível em: http://ecos.sourceware.org/ Abril. Acesso em dezembro de 2008.

FEI, Y.; RAVI, S.; RAGHUNATHAN A.; JHA, N. K. Energy-Optimizing Source Code Transformations for OS-driven Embedded Software. In: INTERNATIONAL CONFERENCE ON VLSI DESIGN, 17, 2004, VLSID'04. Mumbai, India. **Proceedings…** [S.l.: s.n.], 2004.

FERREIRA, R.; BRISOLARA, L.; SPECHT, E.; MATTOS, J. C. B. de; COTA, E.; CARRO, L. Engineering Embedded Software: from application modeling to software synthesis. In: GOMES, L.; FERNANDES, J. M. (Org.). **Behavioral Modeling for Embedded Systems and Technologies:** applications for design and implementation. Hershey, USA: IGI Global, 2009, Cap. 10. Accepted for publication.

GCOV. Disponível em: http://gcc.gnu.org/onlinedocs/gcc/Gcov.Html. Acesso em novembro de 2008.

GIZOPOULOS, D.; PASCHALIS, A.; ZORIAN, Y. **Embedded Processor-Based Self-Test.** Dordrecht, NDL: Kluwer Academic Publishers, 2004.

GUAN, J.; OFFUTT, J. ; AMMANN, P. An Industrial Case Study of Structural Testing Applied to Safety-critical Embedded Software. In: INTERNATIONAL SYMPOSIUM ON EMPIRICAL SOFTWARE ENGINEERING, ISESE'06, Rio de Janeiro, Brazil, 2006. **Proceedings…** [S.l.: s.n.], 2006.

KOOPMAN, P. Reliability, Safety, and Security in Everyday Embedded Systems. In: LATIN-AMERICAN SYMPOSIUM ON DEPENDABLE COMPUTING, 3, Morella, Mexico, 2007. **Proceedings**...[S.l.: s.n.], Vol. 4746, 2007.

LEE, Edward A. (2001) **Embedded Software.** To appear in Advances in Computers (M. Zelkowitz, editor), Vol. 56, Academic Press, London, 2002. University of California, Berkeley, USA November 1, 2001.

LETTNIN, D.; NALLA, P. K.; RUF, J., KROPF, T.; ROSENSTIEL, W.. Verification of Temporal Properties in Automotive Embedded Software. In: DESIGN, AUTOMATION AND TEST IN EUROPE CONFERENCE, 2008. DATE, 2008 Munich, Germany. **Proceedings…** [S.l.: s.n.], 2008.

LEWIS, W. E. **Software Testing Continuous Quality Improvement.** Boca Raton, USA: Auerbach, 2000**.**

LINUX TEST PROJECT. Disponível em : http://ltp.sourceforge.net/ Acesso em novembro 2008.

LOHMANN, Daniel; et al. A Quantitative Analysis of Aspects in the eCos Kernel. ACM SIGOPS Operating Systems Review. In: EUROSYS CONFERENCE. Bélgica, 2006. **Proceedings**… [S.l.: s.n.] SESSION: System design methodologies. p. 191 - 204 , 2006.

MALDONADO, J. C. et al. **Introdução ao Teste de Software**. Technical. Notas Didáticas do ICMC, Instituto de Ciências Matemáticas e de Computação – ICMC/USP, São Carlos, Abril 2004.

MASSA, A. J. **Embedded Software Development with eCos**. Upper Sadle River, USA: Prentice Hall PTR, 2002**.**

MORAES, M. S. **STEP**: **Planejamento, Geração e Seleção de Auto-Teste On-Line para Processadores Embarcados.** 2006. Dissertação (Mestrado em Ciência da Computação) — Instituto de Informática - Universidade Federal do Rio Grande do Sul, Porto Alegre, Brasil. 2006.

PETERS, J. F.; PEDRYCZ, W. **Engenharia de Software:** teoria e prática. Rio de Janeiro, BRA: Campus, 2001.

PFALLER, C.; FLEISCHMANN , A.; HARTMANN, J.; RAPPL, M.; RITTMANN, S.; WILD, D. On the Integration of Design and Test - A Model-Based Approach for Embedded Systems. In: AUTOMATION OF SOFTWARE TEST, 2006. AST'06, Shanghai, China, 2006. **Proceedings…** [S.l.: s.n.], 2006.

PRESSMAN, S. R. **Engenharia de Software**. São Paulo, BRA: McGraw-Hill. 5. ed., 2002.

PRESSMAN, S. R. **Engenharia de Software**. São Paulo, BRA: McGraw-Hill. 6. ed., 2006.

REZENDE, D. A. **Engenharia de Software e Sistemas de Informação**. Rio de Janeiro, BRA: Braspot Livros e Multimídia, 2002.

ROCHA, A. R. C. da;, MALDONADO, J. C.; WEBER, K. C. **Qualidade de Software:** teoria e prática. São Paulo, BRA: Person Education do Brasil, 2001.

SILBERSCHATZ, A.; GALVIN, P.; GAGNE, G. **Sistemas Operacionais:** conceitos e aplicações. [S.l.]: Campus, 1999.

SOMMERVILLE, I. **Software Engineering.** [S.l.]: Addison-Wesley. 6. ed., 2003.

SUNG, A.; CHOI, B.; SHIN, S. An Interface test model for hardware-dependent software and embedded OS API of the embedded system. **Computer Standards & Interfaces**. [S.l.: s.n.], n. 29, p. 430-443, 2007.

TAN, T. K.; Raghunathan, A. JHA, N. K. A Simulation Framework for Energy-Consumption Analysis of OS-Driven Embedded Applications. **IEEE Transactions on Computer-aided Design of Integrated Circuits and Systems.** [S.l.: s.n.], v. 22, n. 9, Sept. 2003.

TAURION, C. **Software Embarcado:** a nova onda da informática. Rio de Janeiro, BRA: Brasport, 2005.

THE OPEN GROUP. Disponível em: http://www.opengroup.org/testing/. Acesso em Novembro de 2008.

VIRTUTECH. Disponível em: www.virtutech.com. Acesso em novembro de 2008.

WOLF, W. **Computers as Components**. [S.l.]: McGraw-Hill, 2001.

WONG, W. E. [et al]. Coverage Testing Embedded Software on Symbian/OMAP. In: INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING AND KNOWLEDGE ENGINEERING, 18, 2006, SEKE, 2006, San Francisco, California. **Proceedings…** [S.l.: s.n.], 2006.

YU, R. Fiscal Cash Register Embedded System Test with Scenario Pattern. **International Journal of Computer Science and Network Security**, [S.l.: s.n.], v.6 n.5A, May 2006.