UNIVERSIDADE TECNOLÓGICA FEDERAL DO PARANÁ
CURSO SUPERIOR DE TECNOLOGIA EM SISTEMAS PARA INTERNET

WILLIAM BARTKO

CRIXUS-TRUCOV: UMA FERRAMENTA PARA APOIAR O TESTE ESTRUTURAL DE SISTEMAS EMBARCADOS CRÍTICOS

TRABALHO DE CONCLUSÃO DE CURSO

CAMPO MOURÃO
2012

WILLIAM BARTKO

CRIXUS-TRUCOV: UMA FERRAMENTA PARA APOIAR O TESTE ESTRUTURAL DE SISTEMAS EMBARCADOS CRÍTICOS

Trabalho de Conclusão de Curso de graduação do Curso Superior de Tecnologia em Sistemas para Internet da Universidade Tecnológica Federal do Paraná – UTFPR, como requisito parcial para obtenção do título de Tecnólogo.

Orientador: Professor Dr. Reginaldo Ré

CAMPO MOURÃO
2012

ATA DA DEFESA DO TABALHO DE CONCLUSÃO DE CURSO

As **vinte horas** do dia **dezenove de novembro de dois mil e doze** foi realizada na sala B105 da UTFPR-CM a sessão pública da defesa do Trabalho de Conclusão do Curso Superior de Tecnologia em Sistemas para Internet do acadêmico **William Bartko** com o título **CRIXUS-TRUCOV: UMA FERRAMENTA PARA APOIAR O TESTE ESTRUTURAL DE SISTEMAS EMBARCADOS CRÍTICOS**. Estavam presentes, além do acadêmico, os membros da banca examinadora composta pelo professor **Dr. Reginaldo Ré** (Orientador-Presidente), pelo professor **Me. Rafael Liberato Roberto** e pelo professor **Me. Gabriel Costa Silva**. Inicialmente, o aluno fez a apresentação do seu trabalho, sendo, em seguida, arguido pela banca examinadora. Após as arguições, sem a presença do acadêmico, a banca examinadora o considerou **APROVADO** na disciplina de Trabalho de Conclusão de Curso e atribuiu, em consenso, a nota _____ (_____). Este resultado foi comunicado ao acadêmico e aos presentes na sessão pública. A banca examinadora também comunicou ao acadêmico que este resultado fica condicionado à entrega da versão final dentro dos padrões e da documentação exigida pela UTFPR ao professor Responsável do TCC no prazo de **quatro dias**. Em seguida foi encerrada a sessão e, para constar, foi lavrada a presente Ata que segue assinada pelos membros da banca examinadora, após lida e considerada conforme.

Observações:

_____

_____

_____

Campo Mourão, 19 de novembro de 2012.

Prof. Me. Rafael Liberato Roberto Membro	Prof. Me. Gabriel Costa Silva Membro	Prof. Dr. Reginaldo Ré Orientador

RESUMO

BARTKO, William. Crixus-Trucov: uma ferramenta para apoiar o teste estrutural de Sistemas Embarcados Críticos. 62 f. Trabalho de Conclusão de Curso – Curso Superior de Tecnologia em Sistemas para Internet, Universidade Tecnológica Federal do Paraná – UTFPR. Campo Mourão, 2012.

Os sistemas embarcados críticos são dispositivos programáveis projetados para trabalharem em conjunto com situações críticas, no qual falhas podem gerar resultados catastróficos. O desenvolvimento desses sistemas necessita de atenção no hardware e software para que essas situações sejam previnidas. Porém, muitas falhas ocorrem por erros no programa, pois estão mal testados e propensos a problemas inesperados, por não seguirem critérios de teste que cobririam todos os pontos do programa, e muitas vezes pela ausência de ferramentas com esse objetivo. Esse trabalho tem como propósito o desenvolvimento de uma ferramenta de teste para microcontroladores Pic, da Microchip, que mostre de forma visual os pontos cobertos do programa facilitando assim a atividade de teste nestes dispositivos.

Palavras-chaves: Teste de Software. Teste de Cobertura. Teste de Sistemas Embarcados Críticos. Sistemas microcontrolados.

ABSTRACT

BARTKO, William. Crixus-Trucov: a toll to support the structural testing in critical embedded systems. 62 f. Trabalho de Conclusão de Curso – Curso Superior de Tecnologia em Sistemas para Internet, Universidade Tecnológica Federal do Paraná – UTFPR. Campo Mourão, 2012.

Safety-Critical systems are programmable devices designed to be functional, even when failures can produce catastrophic results. It is necessary attention during the development of this systems in order to avoid this situations. However, the lack of specific tools to support appropriated testing criteria and to support testing coverage of safety critical embedded systems results in error-prone software. In this work, we propose a coverage testing tool to support the test of microcontroller systems implemented to PIC microcontrollers of Microchip. The main objective of our tool is to aid testing activities during the development process of safety critical embedded software.

Keywords: Software testing. Test coverage. Safety-Critical Embedded Software Testing. Microcontroller systems.

LISTA DE FIGURAS

SUMÁRIO

1 INTRODUÇÃO

1.1 CONTEXTUALIZAÇÃO

Cada vez mais os softwares embarcados estão presentes na vida das pessoas, nas mais diversas aplicações, como celulares, aviões, equipamentos médicos, entre outros. Muitos desses softwares compõem sistemas denominados sistemas críticos (do inglês*, safety critical systems*). Eles são sistemas técnicos ou sociotécnicos dos quais as pessoas ou negócios dependem. Se esses sistemas falharem ao desempenharem os seus serviços conforme o esperado, podem causar sérios problemas e prejuízos significativos (SOMMERVILLE, 2008). A variedade das novas tecnologias utilizadas nessa categoria de sistemas, juntamente com sua própria criticalidade, tornam o desenvolvimento do software que está embarcado mais complexo. Software de sistemas embarcados críticos que não funcionam corretamente podem levar a muitos problemas e não inspiram confiança aos usuários (BSTQB, 2007). Além disso, caso apresentem problemas pós entrega, podem acarretar em custos altos de manutenção, causando má reputação a empresa desenvolvedora, e até mesmo colocar em risco a vida de seres humanos (LEVESON; TURNER, 1993; WONG et al., 2010). Um exemplo é o caso do Therac-25, um dos casos mais conhecidos na comunidade acadêmica, um sistema de computação de tempo real controlando um equipamento de terapia radioativa que causou diversos acidentes que resultaram na morte de pacientes (LEVESON; TURNER, 1993).

Desta maneira, no processo de desenvolvimento e manutenção de sistemas embarcados críticos, devem ser utilizadas técnicas e ferramentas que busquem prevenir a ocorrência de acidentes e uma vez que quanto maiores ou mais graves forem as perdas ou prejuízos decorrentes de falhas do sistema, mais se justificam esforços e recursos investidos na prevenção desses acidentes (SAEED et al., 1991). Dentre estes esforços, o teste de software pode evitar os problemas citados uma vez que reduz do risco da ocorrência de defeitos do software, contribuindo para melhorar a qualidade do produto final (MYERS, 1979).

Recentemente, muitas ferramentas automatizadas e aperfeiçoamentos de técnicas para geração e aplicação de testes de software têm sido desenvolvidos (JORGE, 2010). Um dos principais motivos para o desenvolvimento dessas ferramentas é a confiabilidade que seu uso proporciona. Isso porque quando a atividade de teste é aplicada manualmente, aumentam-se as chances de erros humanos (JORGE, 2010).

1.2 MOTIVAÇÃO

A Saubern Produtos Médicos[1] é uma empresa que atua na área de produtos nefrológicos e desenvolve equipamentos que aperfeiçoam a hemodiálise de pacientes com insuficiência renal. Dentre os principais produtos, destacam-se a reprocessadora de filtros Quality One, responsável pela limpeza de filtros de hemodiálise, e a Hemodialisadora, responsável pela realização de diálise.

Atualmente, na Saubern Produtos Médicos, são usadas normas para validação do hardware, como as normas de validação de sistemas eletromédicos NBR IEC 60601-1-1(ASSOCIAÇÃO..., 2007) e incompatibilidade magnética, *CEM* NBR IEC 60601-1-2(ASSOCIAÇÃO..., 2009). Essas normas são exigidas pelos órgãos responsáveis pela homologação de produtos e equipamentos médicos (INMETRO[2] e ANVISA[3]). No entanto, no projeto e desenvolvimento da Hemodialisadora não foram utilizadas técnicas nem critérios estruturais de teste de unidade. Isso porque a empresa utiliza o teste de sistema, que significa verificar se as placas executam a função a elas designada, como teste de aceitação final para o software embarcado nas placas. Outro fator negativo a essa prática, é a dificuldade em encontrar ferramentas com esse objetivo. A falta de ferramentas específicas para testes estruturais é uma das motivações para a não utilização dessa atividade.

Nesse contexto, a falta de rigor no desenvolvimento do software é algo problemático, pois quanto se desenvolve sistemas críticos, o teste de software

[1] www.saubern.com.br
[2] www.inmetro.gov.br
[3] www.anvisa.gov.br

juntamente com o teste de hardware são fundamentais para a segurança do projeto (HAGAR, 1998). Os resultados dos testes podem ser usados para validar a integridade e a segurança do sistema, além de fornecer evidências de que o sistema está adequadamente testado para evitar consequências catastróficas ou críticas (HAGAR, 1998; BSTQB, 2007). Além disso, investir na atividade de teste pode contribuir para a melhoria da qualidade do produto final, reduzindo custos de manutenção pós-entrega.

Atualmente, na Saubern, é utilizado para o teste de software, uma ferramenta chamada MPLAB SIM (MICROCHIP, 2012) que possui recursos limitados no que diz respeito à análise de cobertura referente a teste de software, por não ser projetada para esse fim. A saída apresentada por esse recurso apenas exibe um arquivo textual apresentando as linhas que foram cobertas, não mostrando a contagem de vezes que a linha foi executada, muito menos um relatório gráfico de execução contendo os arcos e blocos que constituem o programa, dificultando a atividade de teste.

1.3 OBJETIVOS

O objetivo desse trabalho é apoiar o teste estrutural na Saubern por meio do desenvolvimento de uma ferramenta de teste estrutural para microcontroladores PIC (MICROCHIP, 2012) que seja simples, ao mesmo tempo flexível, podendo ser utilizada na aplicação de diferentes critérios, mostrando a cobertura da execução do teste realizado, facilitando assim a execução da atividade de testes em sistemas embarcados críticos.

Atualmente, os sistemas baseados nesses microcontroladores não possuem nenhuma ferramenta que faça o teste de cobertura. Sem a utilização de uma ferramenta para esse fim, é difícil aplicar a atividade de teste de maneira confiável e assim prevenir erros e falhas nos produtos finais. É indispensável a existência de uma ferramenta de testes de cobertura para a realização da atividade de teste.

2 REFERENCIAL TEÓRICO

2.1 Teste de software

O processo de desenvolvimento de software envolve uma série de atividades nas quais, apesar do uso de técnicas, métodos e ferramentas empregados, erros no produto ainda podem persistir. Atividades agregadas sob o nome de Garantia de Qualidade de Software têm sido utilizadas ao longo de todo o processo de desenvolvimento, entre elas atividades de V V& T – Verificação, Validação e Teste, com o objetivo de minimizar a ocorrência de erros e riscos associados. Dentre as técnicas de verificação e validação, a atividade de teste é uma das mais utilizadas, constituindo-se em um dos elementos para fornecer evidências da confiabilidade do software em complemento a outras atividades, como por exemplo, o uso de revisões e de técnicas formais e rigorosas de especificação e de verificação (MALDONADO et al., 1998).

Segundo Myers (1979), o principal objetivo do teste de software é revelar a presença de erros no produto. Portanto, o teste bem sucedido é aquele que consegue determinar casos de teste para os quais o programa em teste falhe. Tem-se observado que a própria atividade de projeto de casos de teste é bastante efetiva em evidenciar a presença de defeitos de software. O ideal seria que todo software fosse exercitado com todos os valores de domínio de entrada. Porém, na maioria das vezes isso é impraticável, levando em conta o custo e tempo dedicado para realizá-los. Por isso, é necessário determinar quais casos de teste utilizar, de forma que descubram a maioria de erros existentes, justificando sua execução (MALDONADO et al., 1998). Os casos de teste geralmente consistem de uma referência a um identificador ou requisito de uma especificação, pré-condições, eventos, série de passos a se seguir, entrada de dados, saída de dados, resultado esperado e resultado obtido. São elaborados para identificar defeitos na estrutura interna, por meio de situações que exercitem adequadamente todas as estruturas utilizadas em um programa; ou ainda, garantir que os requisitos do software sejam plenamente atendidos. Numa situação ideal, ao desenvolver casos de teste, espera-

se encontrar o subconjunto com a maior probabilidade de encontrar a maioria dos erros (MYERS, 1979). Para sua criação, são utilizados critérios de teste, que servem para selecionar e avaliar os casos de teste de forma a aumentar as possibilidades de provocar falhas ou, quando isso não ocorre, estabelecer um nível elevado de confiança na correção do produto (ROCHA et al., 2001). Os critérios de teste não comprovam necessariamente que um software está atendendo aos requisitos documentados. Mas se executados sistematicamente e criteriosamente, contribuem para aumentar a confiança de que o programa está executando o desejado (VINCENZI et al., 2001).

De forma geral, os problemas de software podem ser divididos em três categorias: defeito (*fault*), engano (*mistake*) e erro (*error*). O padrão IEEE número 610.12-1990 (IEEE, 1990) diferencia os termos: defeito (*fault*) – passo, processo ou definição de dados incorretos, como por exemplo, uma instrução ou comando incorreto; engano (*mistake*) – ação humana que produz um resultado incorreto, como por exemplo, uma ação incorreta tomada pelo programador; erro (*error*) – diferença entre o valor obtido e o valor esperado, ou seja, qualquer estado intermediário incorreto ou resultado inesperado na execução do programa constitui um erro; e falha (*failure*) – produção de uma saída incorreta com relação à especificação. A Figura 1 exemplifica esquematicamente a diferença entre esses termos. Apesar da diferenciação dada pelo padrão IEEE, neste trabalho, os termos engano, defeito e erro serão referenciados como erro (causa) e o termo falha (consequência) a um comportamento incorreto do programa.

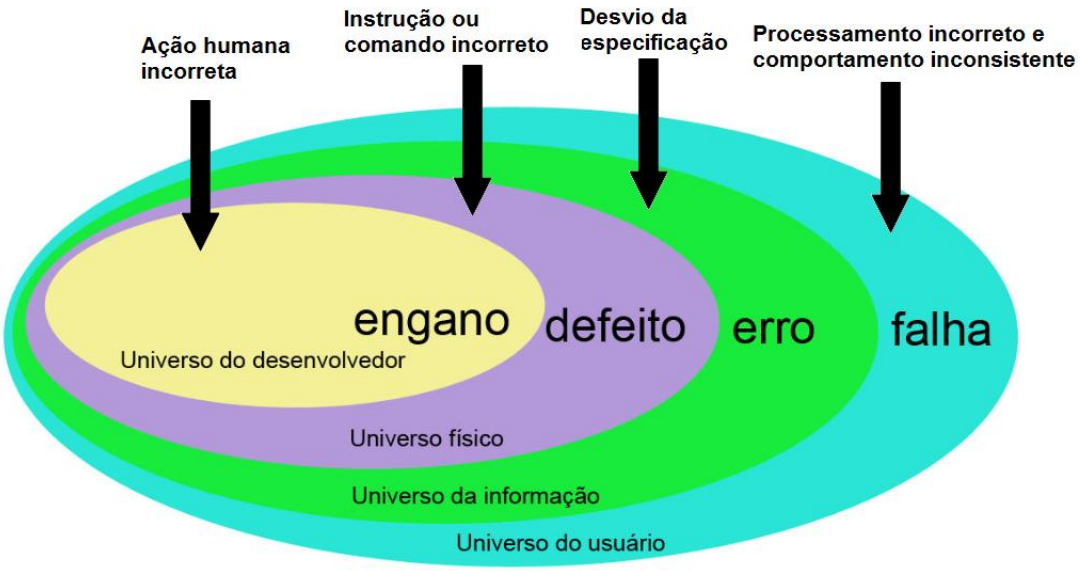

Figura 1 - Engano x defeito x erro x falha
Fonte: Adaptado de Barbosa et al., 2000.

O teste de produtos de software envolve basicamente quatro etapas: planejamento de testes, o projeto de casos de teste, execução e avaliação dos resultados dos testes (MYERS, 1979). Essas atividades devem ser desenvolvidas ao longo do próprio processo de desenvolvimento do software, e em geral, concretizam-se em fases de teste: de unidade, de integração e de sistema, que são a divisão do programa em partes menores, assim facilitando a condução da atividade de teste.

2.1.1 Fases de teste de software

A atividade de teste pode ser considerada como uma atividade incremental realizada em três fases: teste de unidade, integração e sistema (PRESSMAN, 2000).

Com a divisão da atividade de teste em varias fases, o testador pode se concentrar em aspectos diferentes do software e em diferentes tipos de erros e utilizar diferentes estratégias de seleção de dados de teste e medidas de cobertura (porções de código a ser analisado) em cada uma delas (LINNENKUGEL, 1990).

Os testes de unidade e de integração concentram-se na verificação funcional de um módulo e na incorporação de módulos na estrutura de um programa,

respectivamente. O teste de sistema valida o software assim que ele é incorporado a um sistema maior. A seguir é apresentando uma descrição de cada uma dessas fases (PRESSMAN, 2000).

1. **Teste de unidade:** Concentra-se na menor unidade do projeto de software: o módulo. Um módulo pode ser definido como uma divisão de funcionalidades num software, geralmente especificado como uma função, procedimento ou método. Usando a descrição do projeto como guia, caminhos de controle importantes são testados para descobrir erros dentro das fronteiras dos módulos.
2. **Teste de integração:** O objetivo é, a partir dos módulos testados no nível de unidade, verificar a estrutura do programa determinada pelo projeto. Existem vários tipos, destacando-se dois: a **não-incremental** e a **incremental**. Na incremental, o teste é construído e testado em pequenos segmentos, onde os erros são mais fáceis de serem isolados e corrigidos. Na técnica não incremental, o teste é feito usando todo o código, sem dividir em blocos menores.
3. **Teste de sistema:** Depois que o software foi integrado, e o sistema funciona como um todo são realizados os testes de sistema. O objetivo é certificar que o programa e os demais elementos que compõem a estrutura do produto, tais como hardware e banco de dados, desempenhem o papel desejado.

2.1.2 Técnicas e critérios de teste de software

Ortogonalmente às fases de teste, o teste de software pode ser dividido em técnicas e critérios de teste. As técnicas e critérios de teste fornecem ao desenvolvedor uma abordagem sistemática e teoricamente fundamentada para a condução e avaliação do teste de software. Destacam-se, dentre as varias técnicas, a técnica de teste **funcional, estrutural** e **baseado em erros.**

Segundo Howden (1987) o teste pode ser classificado de duas maneiras: **teste baseado em especificação** e **teste baseado em programa**. De acordo com tal classificação, os critérios da técnica funcional são baseados em especificação e os critérios técnica estrutural e baseada em erros são considerados critérios baseados em programa. A seguir, é apresentada uma descrição de cada uma

dessas técnicas.

2.1.2.1 Técnica Funcional

O teste funcional, também conhecido como teste de caixa preta, tem esse nome pelo fato de tratar o software como uma caixa preta, no qual o conteúdo é desconhecido e é possível visualizar apenas o lado externo (PRESSMAN, 2000). Dessa maneira, o teste utiliza apenas a especificação funcional do programa para derivar os casos de testes que serão utilizados, sem se preocupar com a implementação (BEIZER, 1991). Portanto, uma especificação correta e de acordo com os requisitos funcionais do sistema é essencial para esse tipo de teste (VINCENZI, 1998).

O teste funcional procura revelar erros nas categorias: funções incorretas ou ausentes; erros de interface; erros nas estruturas de dados ou no acesso a banco de dados externo; erros de desempenho; e erros de inicialização e término. Alguns exemplos de critérios de teste funcional são (PRESSMAN, 2000): particionamento em classes de equivalência, análise de valor limite e grafo de causa-efeito.

2.1.2.2 Técnica Estrutural

O teste estrutural ou caixa branca é uma técnica de projeto de casos de teste que usa a estrutura de controle e de fluxo de dados de um programa para derivar os requisitos de teste (PRESSMAN, 2000). Na Figura 2, pode-se observar um trecho de código e um exemplo de grafo defluxo de programa. Como pode ser observado na figura, um grafo de fluxo, ou grafo de programa é um grafo dirigido, com um único nó de entrada e um único nó de saída, no qual cada arco representa um possível desvio de um bloco para outro. Cada bloco possui as seguintes características: uma vez que o primeiro comando do bloco é executado, todos os demais são executados

sequencialmente; e não existe desvio da execução para nenhum comando dentro do bloco.

```
/*01*/ {
/*01*/    char  achar;
/*01*/    int   length, valid_id;
/*01*/    length = 0;
/*01*/    printf ("Digite um possível identificador\n");
/*01*/    printf ("seguido por <ENTER>: ");
/*01*/    achar = fgetc (stdin);
/*01*/    valid_id = valid_starter (achar);
/*01*/    if (valid_id)
/*02*/       length = 1;
/*03*/    achar = fgetc (stdin);
/*04*/    while (achar != '\n')
/*05*/       {
/*05*/        if (!(valid_follower (achar)))
/*06*/           valid_id = 0;
/*07*/           length++;
/*07*/           achar = fgetc (stdin);
/*07*/       }
/*08*/    if (valid_id&& (length >= 1) && (length < 6))
/*09*/        printf ("Valido\n");
/*10*/    else
/*10*/       printf ("Invalido\n");
/*11*/ }
```

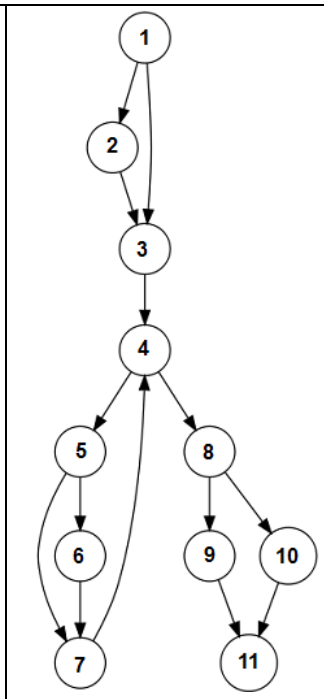

Figura 2 - Exemplo de fluxo (à direita) gerado a partir da execução do trecho de código apresentado (à esquerda)
Fonte: BARBOSA et al., 2000.

Os critérios baseados no fluxo de controle utilizam apenas características de controle da execução do programa, como comandos ou desvios, para derivar os requisitos de testes necessários. Os critérios mais conhecidos desta classe são:

1. **Todos - Nós:** Exige que a execução do programa passe, ao menos uma vez em cada nó do grafo de fluxo. Cada comando do programa deve se executado ao menos uma vez;
2. **Todos - Arcos:** Todos os arcos do grafo, ou seja, cada desvio do programa precisa ser exercitado ao menos uma vez; e
3. **Todos - Caminhos:** Requer que todos os caminhos possíveis do programa sejam executados.

Os casos de testes obtidos durante a aplicação dos critérios funcionais podem ser utilizados como uma base inicial para os testes estruturais. Como, em geral,

essa base inicial não é suficiente para satisfazer totalmente um critério estrutural, novos casos de teste são gerados e adicionados ao conjunto até que se atinja o grau de satisfação desejado, explorando-se, desse modo, os aspectos complementares das duas técnicas (SOUZA, 1996).

2.1.2.3 Técnica Baseada em Erros

O teste baseado em erros utiliza informações sobre os erros mais frequentes cometidos no processo de desenvolvimento de software e sobre os tipos específicos de erros que se devem revelar (DEMILLO, 1987). Existem dois critérios de teste baseados em erros, a semeadura de erros, e a análise de mutantes.

2.2 Teste de Software Embarcado Crítico

Software embarcado é aquele usado como código-fonte em sistemas embarcados. Eles geralmente são parte de um produto na qual o usuário final não interage diretamente ou controla (TIAN et al, 2009). São dispositivos utilizados para controlar, monitorar e apoiar funcionamento de equipamentos, máquinas ou instalações (EBERT e SALECKER, 2009). Esses sistemas interagem com o mundo físico real, controlando um hardware específico. Interagem recebendo sinais através de sensores e enviando sinais de saída para atores que de alguma forma manipulam o ambiente. A Figura 3 apresenta um esquema genérico de um sistema embarcado.

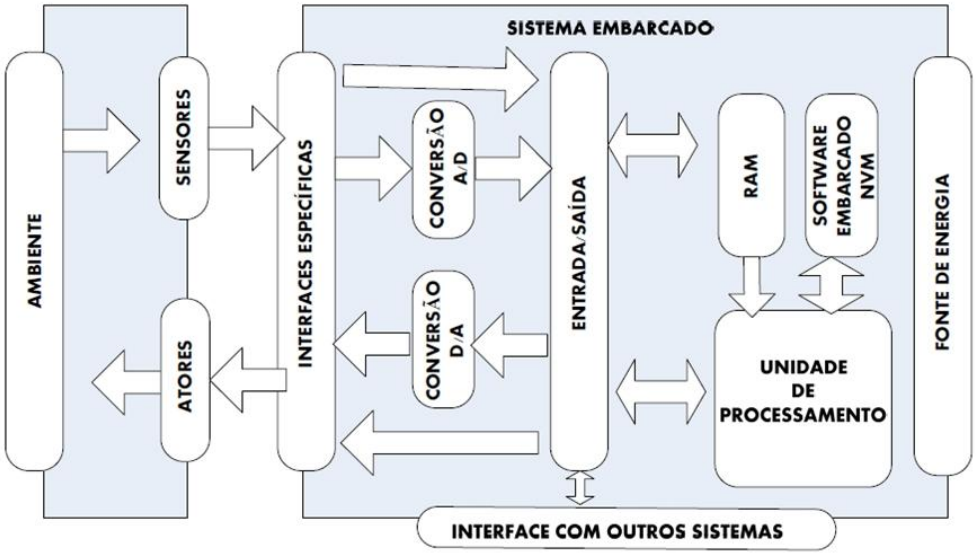

Figura 3: Esquema genérico de um sistema embarcado
Fonte: BROEKMAN; NOTENBOOM, 2002.

De acordo com o esquema, o software embarcado é armazenado em qualquer tipo de memória não volátil (NVM), muitas vezes na ROM, embora ele também possa ser armazenado em cartões de memória flash, no disco rígido, etc. O software embarcado é compilado para um processador particular, a unidade de processamento, que normalmente requer certa quantidade de memória RAM para funcionar. Como a unidade de processamento só pode processar sinais digitais, enquanto o ambiente eventualmente lida com sinais analógicos, deve ser feita a conversão digital-analógico e analógico-digital. A unidade de processamento processa todas as entradas e saídas (E/S) de sinais através de uma camada de E/S dedicada. O sistema integrado interage com os próprios componentes e, possivelmente com outros sistemas (embarcados) usando protocolos de comunicação, como por exemplo, o protocolo Can (BOSCH, 1991), ou por meio de interfaces específicas. Sistemas embarcados podem extrair energia a partir de uma fonte genérica ou podem ter sua própria fonte de alimentação dedicada, como baterias (BROEKMAN; NOTENBOOM, 2002).

Segundo Berger (2001), os sistemas embarcados possuem, dentre várias outras, as seguintes características que diferem dos sistemas tradicionais:

1. São dedicados a tarefas específicas;
2. São geralmente de alto custo;
3. Tem restrições de tempo real;
4. As implicações da falha de software, em geral, são muito mais graves;

5. Exigem ferramentas e métodos especializados para serem projetados de forma eficaz.

Os sistemas embarcados apresentam características próprias que separam o teste de sistema embarcado de um software comum (BERGER, 2001). Como exemplo, pode-se citar:

1. Devem executar de maneira confiável por longos períodos de tempo;
2. São utilizados com frequência em aplicações onde a vida humana está em risco;
3. Devem com frequência compensar falhas no hardware embarcado; e
4. Eventos no mundo real são normalmente assíncronos e não determinísticos, fazendo com que testes de simulação sejam difíceis e não confiáveis.

Como há vários tipos de sistemas embarcados e todos eles possuem características que podem diferir enormemente entre um e outro, uma questão que deve ser levada em conta é o sistema que está sendo testado (BERGER, 2001). Por exemplo, o teste de celulares é significativamente diferente do teste de conversores de vídeo ou do sistema de controle de velocidade em automóveis. Dessa forma, não há uma abordagem única de teste para sistemas embarcados. Por outro lado, sistemas embarcados têm características únicas, ou seja, há muitos problemas semelhantes e que tem soluções semelhantes. Os princípios básicos de teste devem ser aplicados a todos os projetos de teste de sistemas embarcados, mas devem ser diferenciados, de alguma forma, para resolver seus problemas específicos (BROEKMAN; NOTENBOOM, 2002).

Outra questão referente ao teste de software embarcado é que ele deve ser feito sobre uma plataforma de hardware relevante. A maioria dos softwares embarcados tem um ambiente de desenvolvimento que é diferente do ambiente destino. O ambiente de desenvolvimento muitas vezes tem simuladores e outros meios para verificar que o software funcione conforme o pretendido. Quando o software é finalizado, ele é carregado para o ambiente destino e, normalmente, ele não pode ser depurado ou alterado nesse ambiente. As mudanças necessárias são realizadas no ambiente de desenvolvimento, e uma versão compilada é novamente carregada para o ambiente de destino (TIAN et al, 2009).

Devido à estreita associação do software embarcado com sistemas de segurança crítica e muitas vezes com riscos que ameaçam a vida, o software embarcado crítico enfrenta exigências de qualidade elevada. Sistemas de segurança crítica (do inglês, *safety critical*) são aqueles que, se suas operações são perdidas ou degradadas (por exemplo, como resultado de operações incorretas ou inadvertidas), podem resultar em consequências catastróficas ou críticas. Exemplos de sistemas de segurança crítica incluem sistemas de controle de voo de aeronaves, sistemas de comércio automático, sistemas centrais de regulagem de usinas nucleares e, principalmente sistemas médicos (BSTQB, 2007).

Portanto, atividades de teste devem fornecer evidências de que o sistema de segurança crítica está adequadamente testado para evitar consequências desastrosas ou de risco (BSTQB, 2007).Deve-se assegurar que os sistemas e seus softwares executem da forma pretendida. Quanto se desenvolve sistemas críticos, o teste de software, juntamente com o teste de hardware são fundamentais para a qualidade do projeto. O resultado do teste pode ser usado para validar a integridade e a segurança do sistema (HAGAR, 1998).

Isso é relevante para toda a engenharia de software, mas é fundamental para software embarcado crítico. Se o software embarcado crítico tem qualidade insuficiente, prejuízos graves poderão ocorrer, com ferimentos, mortes, ou catástrofes (EBERT; SALECKER, 2009).

2.3 Ferramentas deTeste de Software

Os avanços na tecnologia de fabricação de circuitos integrados têm permitido a fabricação de microcontroladores com cada vez mais recursos de hardware e a custos cada vez menores. Esta disponibilidade de recursos tem sido traduzida em um crescente incremento da complexidade do software embarcado. De fato, o software embarcado está extremamente sofisticado, muitas vezes substituindo o hardware, e vem se tornando a maior parte do sistema embarcado (SEO et al., 2007).

Entretanto, devido à dependência do hardware, o software embarcado, por

muito tempo, tem sido desenvolvido após a construção de um protótipo do equipamento ou com o auxílio de kits de desenvolvimento. Nas etapas iniciais, anteriores ao primeiro protótipo, o hardware é avaliado através de uma plataforma de hardware padrão, fornecida pelo fabricante do microcontrolador utilizado. O software é então desenvolvido com um pacote de ferramentas (compiladores, simuladores e debuggers) bastante específico para a plataforma alvo. Testes de software são executados durante as etapas de desenvolvimento, onde programadores testam a funcionalidade de cada módulo individualmente (testes de unidade) e, com o auxílio de protótipos ou hardware de avaliação, verificam a correta execução do software na plataforma alvo, onde é executada a grande maioria dos testes de integração e de sistema. Conforme (SEO et al., 2007), muitas vezes tais testes são executados em uma abordagem do tipo *big-bang*, onde diferentes partes da aplicação são unidas ainda em um estágio bastante instável. A validação do sistema embarcado como um todo é deixada para os testes funcionais na plataforma alvo, onde o atendimento aos requisitos do projeto é verificado e o equipamento é sujeito a condições extremas de uso.

Esta abordagem de desenvolvimento, dada a complexidade dos sistemas embarcados atuais, não é capaz de garantir a confiabilidade do equipamento, uma vez que o software embarcado atual é composto por uma intrincada combinação de milhares de linhas de código que não podem ter sua funcionalidade verificada sem que se faça uso de técnicas modernas de teste de software (GOMES, 2010). Na contramão deste processo, as ferramentas de desenvolvimento de sistemas embarcados baseados em microcontroladores não estão evoluindo na mesma velocidade (GOMES, 2010). Métodos de geração de casos de teste, análise de cobertura de teste, simuladores que sejam capazes de simular o hardware como um todo (e não apenas o microprocessador) e eficientes analisadores estáticos de código não são comumente encontrados em tais ambientes de desenvolvimento, embora existam algumas (CODE COMPOSER, 2012; IAR, 2012). Infelizmente, devido às particularidades do software embarcado, ferramentas de teste de software geral, facilmente encontradas no mercado, não podem ser diretamente aplicadas a este, pela simples razão de que os acessos ao hardware não serão compreendidos pela ferramenta que, na maioria das vezes, não será sequer capaz de executar o código embarcado.

Neste contexto, visando-se à eliminação de algumas das restrições impostas ao software embarcado que o impedem de serem completamente testado, principalmente com relação ao seu acoplamento com o hardware, algumas abordagens têm sido propostas, como em (ENGBLOM; GIRARD; VERNER, 2006) onde são apresentados todos os benefícios do teste de sistemas embarcados em um ambiente totalmente simulado chamado Simics, ou em (KARLESKY; WILLIAMS, 2007) onde a proposta é a substituição de módulos complexos, ou ainda não existentes, por funções que eles chamaram de Mocks, que possuem a mesma interface das funções substituídas e que podem ser utilizadas de forma a tornar possível o ambiente de testes.

Assim, de maneira geral, não existem ferramentas *Open Source* disponíveis para plataformas específicas, especialmente, no caso deste trabalho, para a família de microcontroladores PIC (MICROCHIP, 2012; YANG et al., 2006). Existem, apenas, ferramentas pagas que são caras e funcionam com limitações que exigem hardwares específicos para seu funcionamento.

3 A FERRAMENTA CRIXUS-TRUCOV

O software Crixus-Trucov é uma extensão da ferramenta Trucov (TRUCOV, 2012) capaz de gerar uma saída visual contendo o fluxo de execução das funções de um arquivo fonte, escrito em linguagem C, compilado pelo compilador C30 (MICROCHIP, 2012). É uma ferramenta open source e seu código-fonte pode ser acessado em http://sourceforge.net/projects/crixustrucov/. Seu funcionamento pode ser observado na Figura 4:

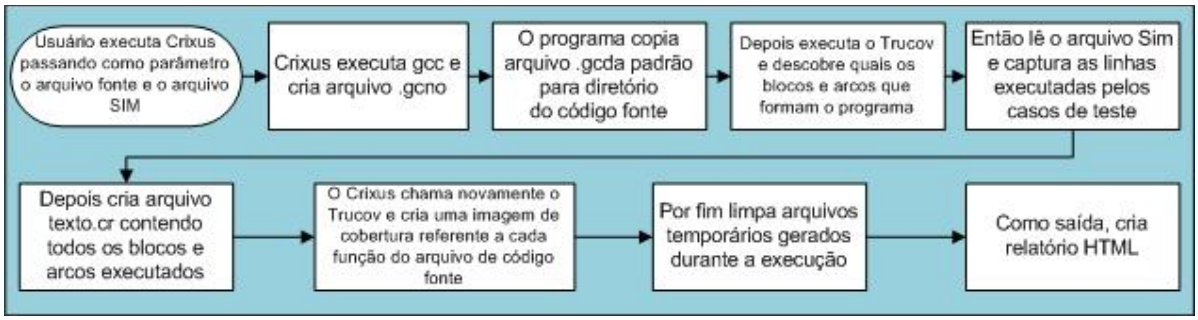

Figura 4 - Fluxo de funcionamento da ferramenta Crixus-Trucov
Fonte: Autoria própria.

A ferramenta foi desenvolvida em Java e recebe como parâmetro dois arquivos: o código fonte, onde está escrito o código em que se deseja realizar a cobertura e o arquivo SIM, que é um arquivo gerado pelo MPLAB SIM e contém o fluxo de execução do programa. A Figura 4 ilustra a sequência das operações que ocorrem no programa. Pode-se verificar que ocorrem vários eventos, desde a escrita de arquivos temporários com scripts para geração de arquivos binários até a criação do relatório final, contendo informações sobre os pontos cobertos pelo programa.

3.1 FERRAMENTA TRUCOV

Para a implementação de uma ferramenta de apoio ao teste de sistemas embarcados críticos, era necessário utilizar dos recursos de teste que o compilador

GCC possuía, pois o compilador C30, utilizado no projeto é baseado nele. Porém, o C30 não da suporte ao recursos de testes. Por isso, foi necessário fazer a pesquisa de uma ferramenta de código aberto que pudesse dar a cobertura do código testado e que pudesse ser adaptado à estrutura do compilador C30. A melhor ferramenta para esse caso foi o Trucov.

O Trucov (TRUCOV, 2012) é uma ferramenta de código aberto que funciona em conjunto com o compilador GCC para mostrar o fluxo de execução e as informações de cobertura. Ele facilita a visualização da cobertura e ajuda o usuário a verificar se os casos de testes feitos são suficientes para a qualidade final do software. O Trucov simplifica a identificação do código não testado de forma visual. É uma ferramenta feita para GNU Linux, sendo compilada em C++. Funciona por linha de comando, sendo necessário que o código fonte seja compilado com o compilador GCC, contendo as diretivas `-fprofile-arcs -ftest-coverage`, resultando na criação de dois arquivos necessários para a criação de seus relatórios: o arquivo `.gcda` e `.gcno`. Possui as opções de relatório gráfico e textual, aceitando também diretivas de debug.

3.1.1Relatórios do Trucov

Na Figura 6 é apresentado um exemplo dos relatórios que podem ser gerados pelo Trucov, usando como referência o código apresentado na Figura 5 e utilizando as diretivas para geração dos arquivos .gcno e .gcda:

```
gcc -fprofile-arcs -ftest-coverage teste.c.
```

Como pode ser analisado, existe um arco no código onde uma decisão será tomada, executando ou o bloco das linhas 8 a 10 ou o bloco das linhas 12 a 14.

```
1 int main(void)
2 {
3       unsigned int a = 10;
4       unsigned int b = 12;
5       unsigned int c = 0;
6
7       if(a>10)
8       {
9               c = b+a;
10      }
11      else
12      {
13              c = b-a;
14      }
15 }
```

Figura 5 - Código fonte teste.c

Fonte: Autoria própria.

3.1.1.1 Relatório gráfico

Utilizando a sintaxe `trucov graph teste. c`, é gerada uma saída gráfica, como pode ser observado na Figura 6:

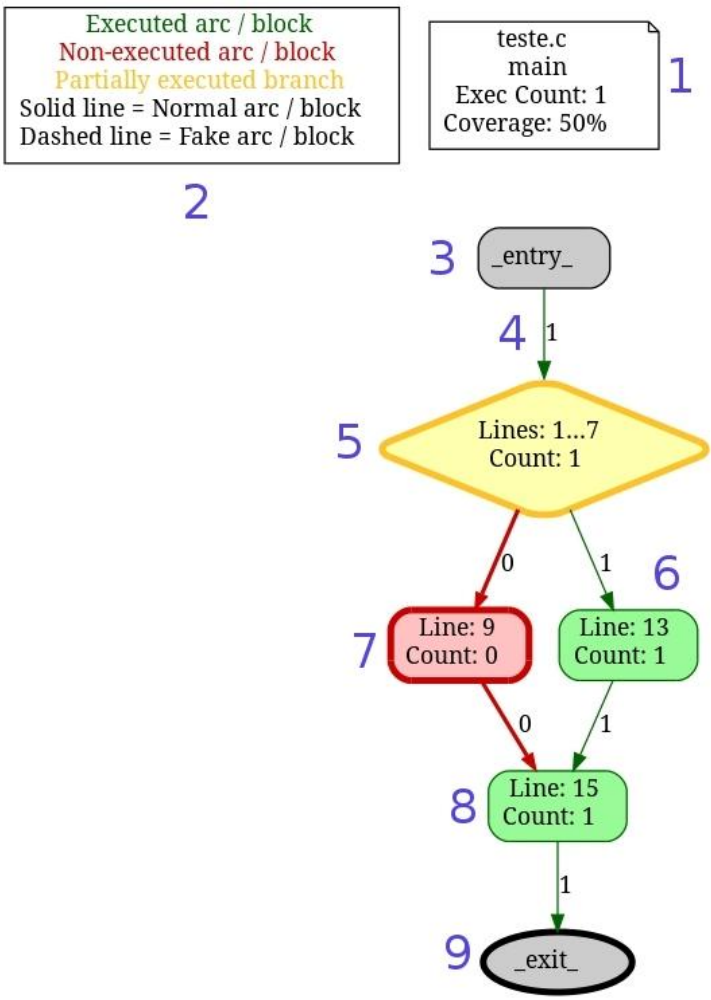

Figura 6 - Relatório gráfico do fluxo de execução do programa `teste.c`

Fonte: Autoria própria.

A imagem possui várias informações, sendo:

1. Esse trecho indica o nome do arquivo do código fonte, o nome da função executada, o número de vezes que foi executada e a cobertura total na da função. No caso, o programa passou uma vez na função e houve um total de 50% de cobertura total dos arcos;
2. Esse campo é uma legenda, indicando o significado das cores presentes na figura. A cor verde (Executed arc/block) mostra o caminho por onde passou o programa; A cor vermelha (Non-executed arc/block) indica os blocos e arcos que não foram executados pelo programa; A cor amarela (Partially executed branch) representa os arcos que tiveram apenas um lado executado (no caso, a condição das linhas 1 a 7 possui a possibilidade de passar pelo resultado positivo do teste “a>10”, seguindo

as linhas 9 e 10, ou negativo, seguindo as linhas 13 e 14, que significa o else). Se o caso de teste cobrir as duas possibilidades, esse arco passaria para a cor verde (Executed arc/block); As linhas sólidas (Solid line) representam os arcos que foram gerados pelas condições do código fonte; e as linhas sólidas são os arcos gerados pelo compilador na execução do programa. São opcionais e o usuário pode ativar/desativar essa opção pelo parâmetro `--show-fake/ --hide-fake`;

3. Esse bloco (entry) indica o ponto onde inicia o programa;
4. As flechas indicam os próximos blocos/arcos por onde o programa passará. O número encima indicam a quantidade de vezes que a execução do software passou por aquele ponto;
5. A figura em losango indica que naquele ponto há um arco. Terá sempre mais de uma flecha, indicando os pontos que seguirá conforme o que for executado no software;
6. Os pontos 6, 7 e 8 indicam blocos. Os que estão em cor verde (6 e 8) indicam que as linhas de códigos desses blocos (linha 13, no bloco 6 e linha 15 no bloco 8) foram executadas pelo programa. O bloco 7 está em cor vermelha, pois seu conteúdo não foi executado.
7. Vide número 6;
8. Vide número 7; e
9. Este circulo indica o fim do programa.

3.1.1.2 Relatório textual

Utilizando a sintaxe `trucov report teste.c` é gerado um arquivo texto com o nome `teste.c.trucov`. Seu conteúdo pode ser observado na Figura 7:

```
1 50% testeTrucov/teste.c
2 50% main (1/2) branches
3               teste.c:7: 1/2 branches: if(a>10)
4               teste.c:9: destination: c = b+a;
```

Figura 7 - Relatório textual do fluxo de execução do programa teste.c
Fonte: Autoria própria.

No relatório textual, as informações são agrupadas, sendo mostrado:

Linha *1*: Nessa linha é mostrado o arquivo fonte cujas funções foram executadas e a quantidade de cobertura obtida nas funções desse arquivo. Como é analisada uma função de cada vez, o total de cobertura do arquivo fonte será o mesmo da função analisada, que no caso é função main.

Linha *2*: É mostrada a função que foi analisada, bem como a porcentagem de execução de seus arcos e quantidade de branches que possui a função. No caso, foram executados 50% dos arcos dos dois branches que possui.

Linha *3*: Essa linha em diante conterá os trechos do código. É organizada da seguinte maneira: <*nome do arquivo, número da linha, função, código*>. No caso, a linha analisada é um arco, situado na linha 7, tendo duas possibilidades e o código presente é `if(a>10)`.

Linha *4*: Aqui tem-se o bloco alcançado na execução do arco da linha 3. Está presente na linha 9, e o código dessa linha foi `c=b+a`.

Pode-se observar que são mostradas as informações de cobertura da função executada, bem como o ponto onde é encontrado um arco e qual o rumo que o programa tomou, no caso descrito foi na linha 9, cujo destino é `c=b+a`.

Ambos os relatórios são de grande ajuda ao programador, pois permitem a visualização dos caminhos por onde o programa passou, baseado na sua execução. A primeira alternativa (visual) permite a visualização de maneira mais elegante e clara, ficando visíveis os pontos que não foram testados e quais casos de teste precisam ser implementados para que haja uma cobertura total do software. A segunda maneira (textual) pode ser utilizada como esqueleto para aplicações

personalizadas em que o testador possa manipular e trabalhar com os dados de maneira mais flexível, possibilitando a criação de dados estatísticos e comparativos.

3.1.2 Arquivos .gcno e .gcda

Os softwares de cobertura para a linguagem C utilizam dois arquivos gerados pelo compilador GCC (GCC, 2012) para a criação dos relatórios de cobertura. São os arquivos com extensão `.gcno` e `.gcda`.

O arquivo `.gcno` é gerado quando um arquivo fonte é compilado pelo compilador GCC utilizando as opções `-fprofile-arcs` e `-ftest-coverage`. Esse arquivo contém uma lista dos arcos de cada função do código fonte e das funções dos arquivos incluídos. Contém também informações que permitem que softwares de análise de cobertura possam reconstruir as chamadas de função e as linhas que compõe cada bloco/arco (HAGEN, 2006).

O arquivo `.gcda` é gerado quando um programa escrito em linguagem C é compilado com a opção `-fprofile-arcs`. Esse arquivo contém informações sobre chamadas de função e fluxo de decisão. Toda vez que a aplicação é executada, o conteúdo do arquivo é incrementado, adicionando novos fluxos conforme decisões tomadas no programa (HAGEN, 2006).

3.2 Mplab SIM

O compilador C30 (MICROCHIP, 2012) é baseado no GCC, porém sua estrutura é diferente. Ao invés de gerar um executável que pode ser executado em qualquer terminal pela linha de comando, a saída final é um arquivo com a extensão .hex. Por isso, não é possível utilizar as ferramentas do GCC. Esse arquivo é importado por um gravador, e assim seus dados são transferidos diretamente para o microcontrolador. O conteúdo desse arquivo é caracterizado por uma estrutura binária, com as instruções e registradores responsáveis pela tomada de decisões do

microcontrolador. Por conta disso, a diretiva `-fprofile-arcs -ftest-coverage` aplicada ao C30 não cria o arquivo `.gcda`, apenas o arquivo `.gcno`, de maneira que o Trucov não pode ser executado, assim não podendo ser gerado os relatórios de cobertura descritos anteriormente.

Porém, existe na IDE Mplab, um plugin chamado Mplab SIM, que tem como função a simulação do hardware do microcontrolador. É possível verificar os pontos por onde o programa passou, e há um arquivo de log que pode ser lido, contendo as linhas executadas na simulação. A Figura 8 mostra como seria a visualização do arquivo teste.c se estivesse anexado num projeto no Mplab e estivesse sido executado pelo Mplab Sim:

Figura 8 - Cobertura do Mplab Sim da execução do programa teste.c

Fonte: Autoria própria.

O relatório do Mplab Sim contém as informações sobre a execução das linhas do código, sendo mostrado:

Bloco *1*: Mostra a data de criação do arquivo de cobertura SIM;

Bloco *2*: Mostra o projeto do Mplab que contém o arquivo fonte executado;

Bloco *3*: Contém as informações sobre o código coberto. A sequência de informações sempre será: Nome do arquivo fonte; nome da função; e trecho coberto, inicio (fromLine) e termino (toLine). A expressão “Address” indica o endereço de memória no microcontrolador que contém aquela linha; e

Bloco *4*: Indica as linhas de código que não foram cobertas na execução. Sua estrutura é similar ao bloco *3*.

3.3 Funcionamento da ferramenta Crixus-Trucov

Nessa seção será explicado o funcionamento da ferramenta Crixus-Trucov. Para usá-la, é necessário que o usuário tenha feito o download no site do projeto http://sourceforge.net/projects/crixustrucov/ e compilado o código-fonte. É necessário também que tenha feito um projeto no Mplab e ativado a simulação pelo Mplab Sim, habilitando a saída da cobertura para um arquivo texto.

O Crixus-Trucov é feito para ser executado na plataforma Linux, sendo necessário que haja dois arquivos: O arquivo fonte `(.c)` e o arquivo do Mplab Sim. Deve ser executado sobre um terminal. Sua sintaxe é:

```
java -jar Crixus.jar <nome do arquivo fonte.c> <nome arquivo .sim>
```

O Crixus-Trucov lê o arquivo fonte descobrindo quais são os arcos e funções que o compõe, depois lê o arquivo de log do Mplab Sim para descobrir quais os blocos e funções que foram cobertos.

É feito na saída, um relatório contendo um arquivo HTML com as funções executadas, bem como o relatório gráfico da função. A Figura 9 mostra o relatório feito com o arquivo teste.c:

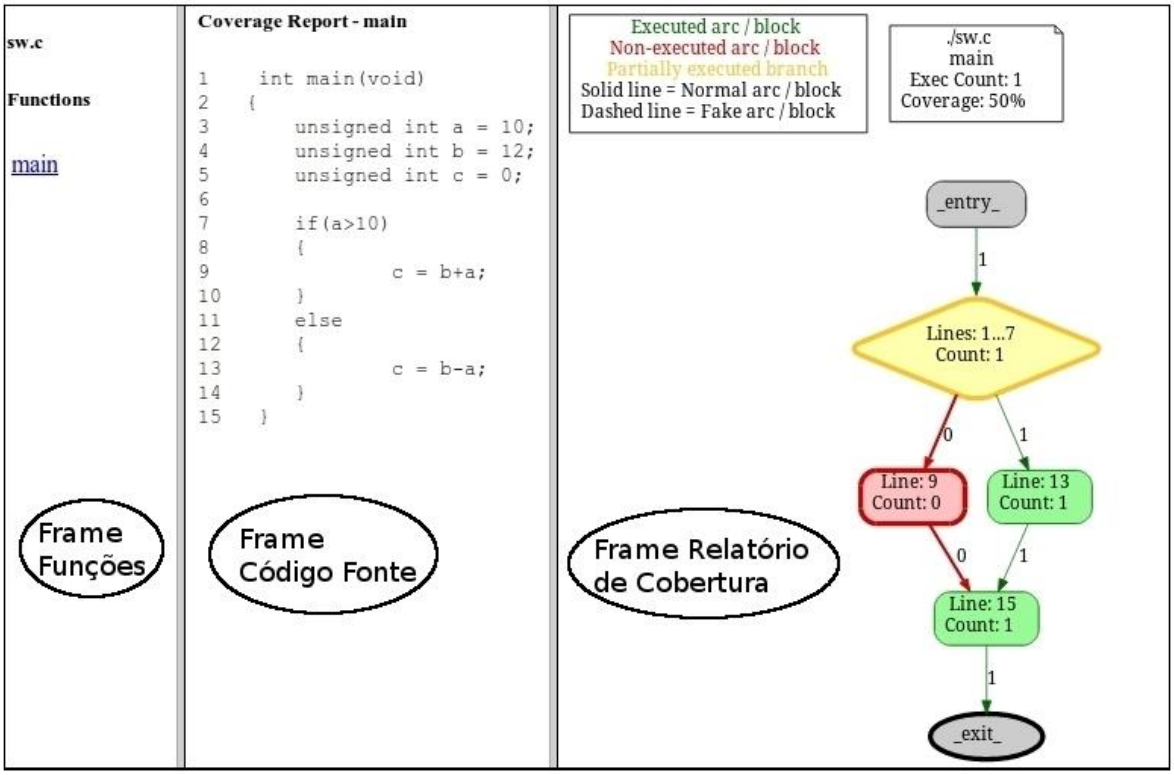

Figura 9 - Relatório de cobertura do arquivo teste.c feito pela ferramenta Crixus-Trucov
Fonte: Autoria própria.

Como pode ser observado na Figura 9, o relatório consiste de uma página HTML contendo três frames: O frame de funções, que apresenta todas as funções que compõem o arquivo fonte, o frame de código fonte, que apresenta o código fonte da função selecionada e o frame de relatório de cobertura, que apresenta a imagem gerada com a execução do programa. O programa ainda possui um parâmetro que pode ser passado para ver a execução em debug. Esse parâmetro é `-d`, quando inserido logo a frente do arquivo sim, permite visualizar passo a passo no terminal os passos de leitura e criação do arquivo de relatório.

Esse relatório é de grande ajuda ao desenvolvedor, pois com ele pode ser verificado a eficiência dos casos de teste, quais pontos cada um está cobrindo, além de servir como documentação do histórico de testes realizados no programa. Pode também servir para verificar se a documentação original do projeto está de acordo com o que foi implementando, representando todos os caminhos que o software pode executar.

3.4 Limitações da ferramenta

Como o programa ainda está na versão de experimentação, existem alguns bugs que precisam ser corrigidos. Segue a lista desses problemas:

- O compilador C30 acrescenta em sua execução o caminho do arquivo header que contém as informações do microcontrolador em uso. Esse arquivo se encontra no diretório “C:\Program Files\Microchip\MPLAB C30\support”. Por isso, ao tentar compilar o projeto, é necessário copiar o arquivo header do microcontrolador no diretório padrão do projeto e incluí-lo, ao invés de incluir o arquivo padrão do microcontrolador;
- Nesse mesmo arquivo, é necessário fazer uma alteração. A linha #error "Include file does not match processor setting" deve ser comentada, pois ela existe para impedir que o usuário escolha um modelo de PIC use o header do outro. Porém, se isso estiver ativo no programa, o GCC não conseguirá gerar o arquivo .gcno, impossibilitando a geração dos relatórios finais; e
- A ferramenta não é capaz de mostrar o número de vezes que a execução passou por um determinado bloco/arco, pois o log do Mplab Sim não indica fornece essa informação.

3.5 Exemplo de uso da ferramenta

Nesta sessão é mostrado o uso da ferramenta Crixus-Trucov aplicado em softwares reais. Para o exemplo, será utilizada uma função responsável por controlar o nível de água utilizada em um reservatório chamada `le_nivel(void)`, apresentando na Figura 10. Ela faz parte do módulo abastecimento, do projeto Hemodialisadora. É responsável por verificar se o reservatório de água presente na máquina está vazio. Conforme a leitura dos sensores (dados.snbra, dados.snmra e dados.snara) ela indicará se a máquina pode ou não operar, devido à existência ou não de água. Se ficar muito tempo abaixo do nível baixo, será indicado que o reservatório está vazio. Como essa função roda num laço principal, significa que

todo instante ela é chamada. Para que seja atualizada em intervalos de tempos específicos, é utilizada a flag `FS_BASE_ATUALIZA_NIVEL`, que é zerada logo que a função inicia, sendo definida a cada 2s, numa interrupção externa do software. A função foi escrita utilizando o Mplab, tendo sua execução registrada no arquivo sim.txt, gerado pelo Mplab SIM. Logo após, é utilizado o Crixux-Trucov para a geração dos relatórios.

```
83   inline void le_nivel( void )
84   {
85       union unsigned_char aux;
86
87       if( FS_BASE_ATUALIZA_NIVEL )
88       {
89           FS_BASE_ATUALIZA_NIVEL = 0;
90
91           tempo_nivel_baixo = TOUT_NIVEL_BAIXO;
92
93           // Atualiza o nível baixo (SNBRA) de acordo com
94           // a tendência do nível médio (SNMRA)
```

```
95            if( dados.snmra == 1 )
96            {
97                dados.snbra = NIVEL_DETECTADO;
98
99                tempo_nivel_baixo = TOUT_NIVEL_BAIXO;
100           }
101           else
102           {
103               if( tempo_nivel_parado )
104               {
105                       tempo_nivel_parado--;
106               }
107               else
108               {
109                       dados.snbra = NIVEL_NAO_DETECTADO;
110               }
111           }
112           // Concatena estado dos sensores individuais de nível
113           aux.value = 0;
114           aux.bit0 = dados.snbra;
115           aux.bit1 = dados.snmra;
116           aux.bit2 = dados.snara;
117
118           // Verifica o estado completo dos sensores de nível
119           switch( aux.value )
120           {
121           case 0b000:
122               nivel_reservatorio = NIVEL_VAZIO;
123               break;
124
125           case 0b001:
126               nivel_reservatorio = NIVEL_BAIXO;
127               break;
128
129           case 0b011:
130               nivel_reservatorio = NIVEL_MEDIO;
131               break;
132
133           case 0b111:
134               nivel_reservatorio = NIVEL_ALTO;
135               break;
136
137           default:
138               nivel_reservatorio = NIVEL_DESCONHECIDO;
139               break;
140           }
141
142           // Verifica se há falha nos sensores de nível
143           if( nivel_reservatorio == NIVEL_DESCONHECIDO )
144           {
145               dados.falha_sensor_nivel = 1;
146           }
147           else
148           {
149               dados.falha_sensor_nivel = 0;
150           }
151       }
152       else
153       {
154           tempo_nivel_baixo =  TOUT_NIVEL_CONFIGURADO;
155       }
157   }
```

Figura 10: Código fonte da função `le_nivel`
Fonte: Autoria própria.

Foram realizados seis casos de teste para atingir a cobertura total do código. A seguir é apresentada a implementação desses casos de teste e a seguir a imagem gerada após a execução do Crixus-Trucov:

```
1   int testa_nivel_reservatorio_igual_NIVEL_VAZIO(void)
2   {
3       FS_BASE_ATUALIZA_NIVEL = 1;
4
5       dados.snbra = 0;
6       dados.snmra = 0;
7       dados.snara = 0;
8
9       ler_nivel();
10
11      mu_assert(0, nivel_reservatorio == NIVEL_VAZIO;
12      return 1;
13  }
```

Figura 11 - Código fonte da função `testa_nivel_reservatorio_igual_NIVEL_VAZIO`

Fonte: Autoria própria.

No caso de teste da Figura 11, é feito o teste que indica que o nível de água não foi detectado, pois todos os sensores de água (baixo, médio e alto) não alcançaram o valor1. Ao executar esse caso de teste, deseja-se ter como resultado nivel_reservatorio igual a constante NIVEL_VAZIO. Se esse valor não foi alcançado, significa que houve alterações no código ou foram modificadas as regras de negócio.

Executando esse teste com o Crixus-Trucov, obteve-se cobertura de 38% do fluxo do programa, sendo mostrada na Figura 12:

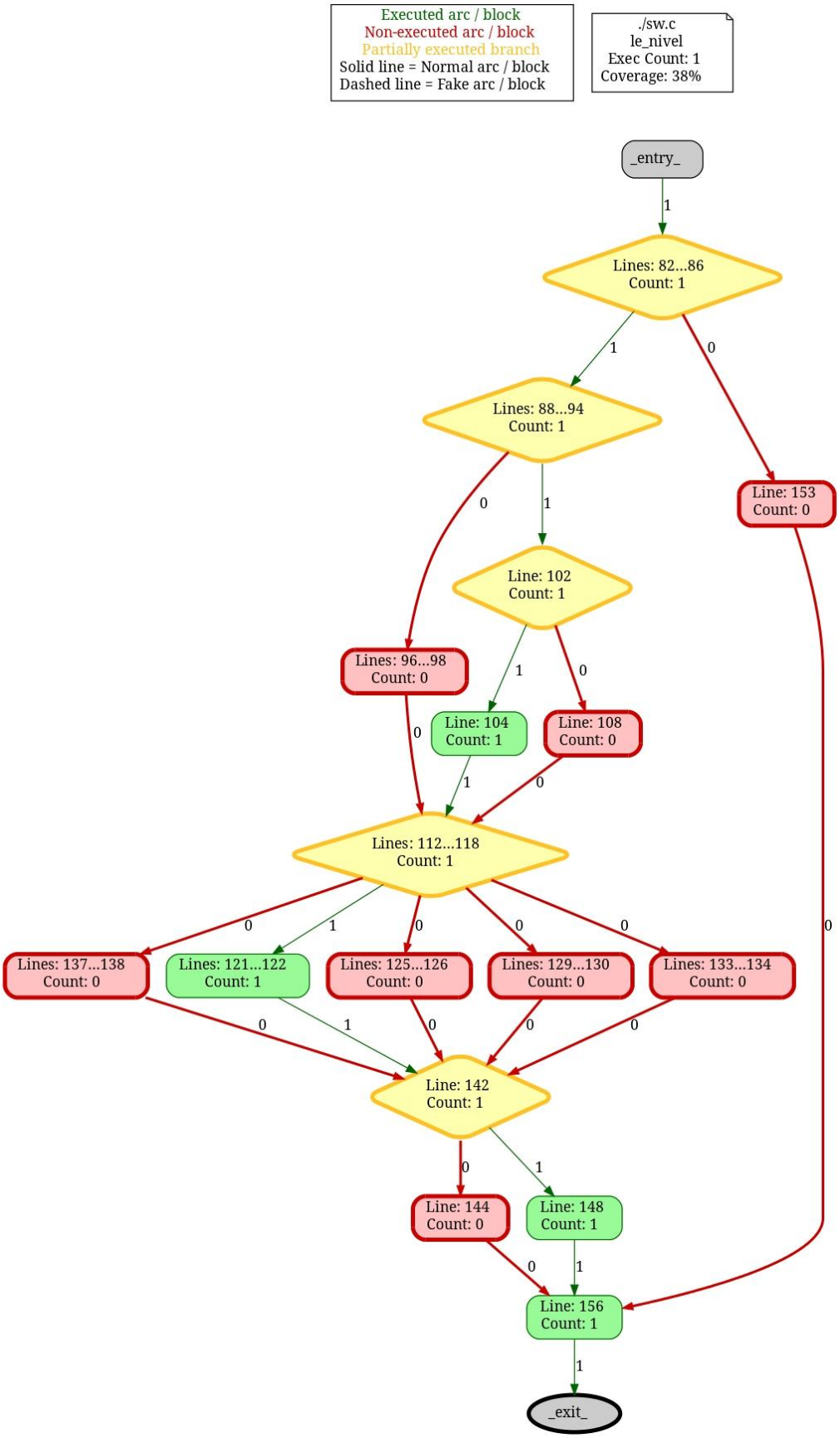

Figura 12 - Cobertura realizada em le_nivel pela função `testa_nivel_reservatorio_igual_NIVEL_VAZIO`

Fonte: Autoria própria.

A Figura 13 mostra o segundo caso de teste e a cobertura alcançada por ele. O objetivo é testar a função quando os dados dos sensores de água estavam com valores diferentes dos corretos (0 e 1). A função então deve retornar na variável `nivel_reservatorio` o valor `NIVEL_NAO_DETECTADO`. Além de retornar na variável `dados.falha_sensor_nivel` o valor 1. O total coberto foi 38% também, porém passando por outros blocos e arcos:

```
15  int testa_nivel_reservatorio_igual_NIVEL_NAO_DETECTADO (void)
16  {
17   FS_BASE_ATUALIZA_NIVEL = 1;
18   dados.snmra = 0;
19   tempo_nivel_parado = 0;
20
21   dados.snbra = 0xff;
22   dados.snara = 0xff;
23
24   le_nivel();
25   mu_assert(0, nivel_reservatorio == NIVEL_NAO_DETECTADO);
26   mu_assert(0, dados.falha_sensor_nivel == 1);
27   return 1;
28  }
```

Figura 13 - Código fonte da função `testa_nivel_reservatorio_igual_NIVEL_NAO_DETECTADO`

Fonte: Autoria própria.

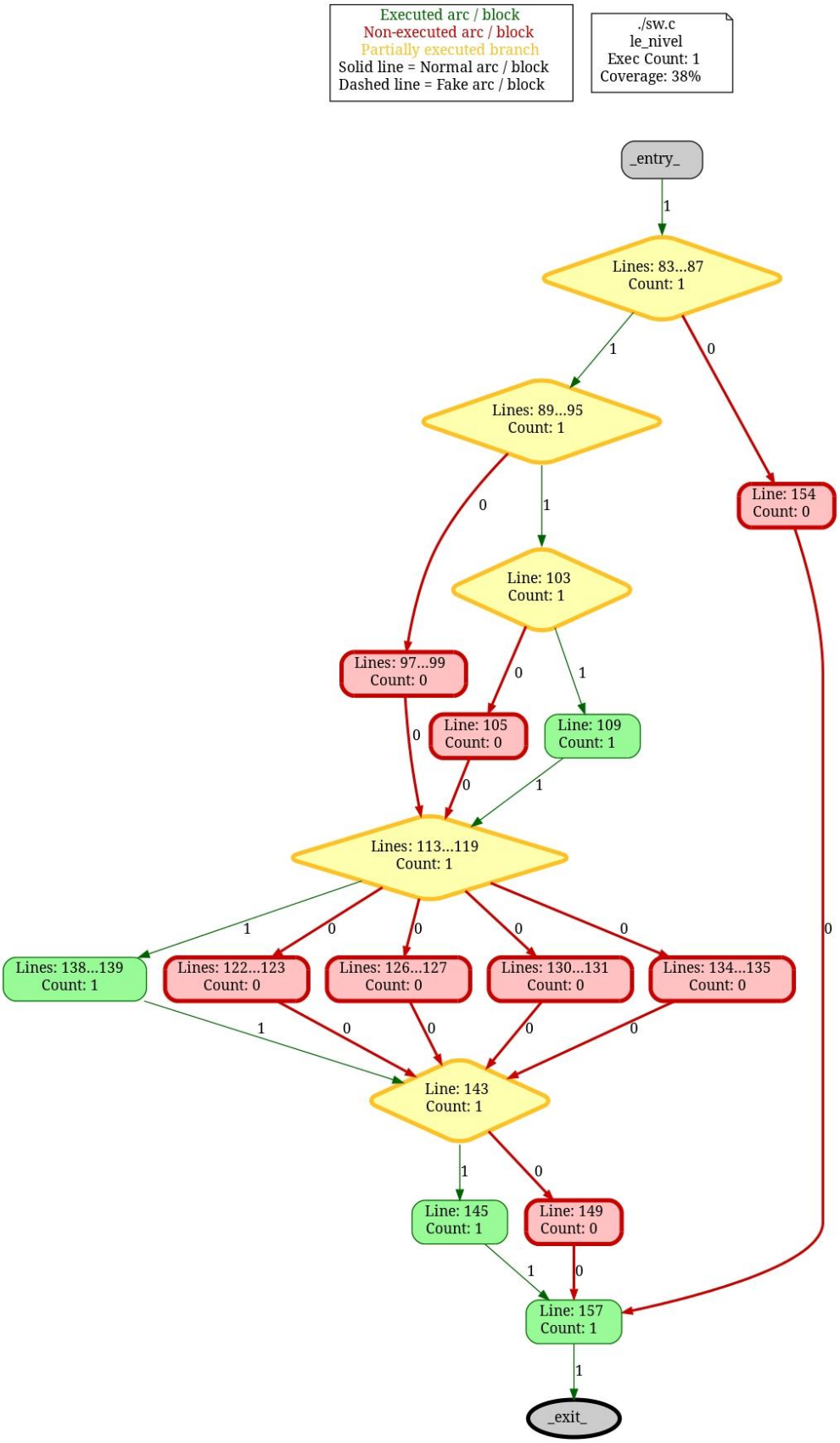

Figura 14 - Cobertura realizada em le_nivel pela função `testa_nivel_reservatorio_igual_NIVEL_NAO_DETECTADO`.

Fonte: Autoria própria.

No próximo caso de teste (Figura 15), procurou-se contornar a expressão. `if( dados.snmra == 1 )` na linha 95 , fazendo com que fosse verdadeira, desviando dos testes que acontecem quando essa expressão é falsa(ocorre um timeout quando é falsa, e consequentemente outros testes), diminuindo assim os pontos alcançados. No final, o resultado dessa expressão é que a variável `nivel_reservatorio` deve ter seu valor igual a constante `NIVEL_MEDIO`. Quando executada, atinge uma cobertura para 31%. A Figura 16 mostra o relatório gerado.

```
29  int testa_nivel_reservatorio_igual_NIVEL_MEDIO (void)
30  {
31   FS_BASE_ATUALIZA_NIVEL = 1;
32   dados.snmra = 1;
33   dados.snmra = 1;
34   le_nivel();
35   mu_assert(0, nivel_reservatorio == NIVEL_MEDIO);
36   return 1;
37  }
```

Figura 15 - Código fonte da função `testa_nivel_reservatorio_igual_NIVEL_MEDIO`

Fonte: Autoria própria.

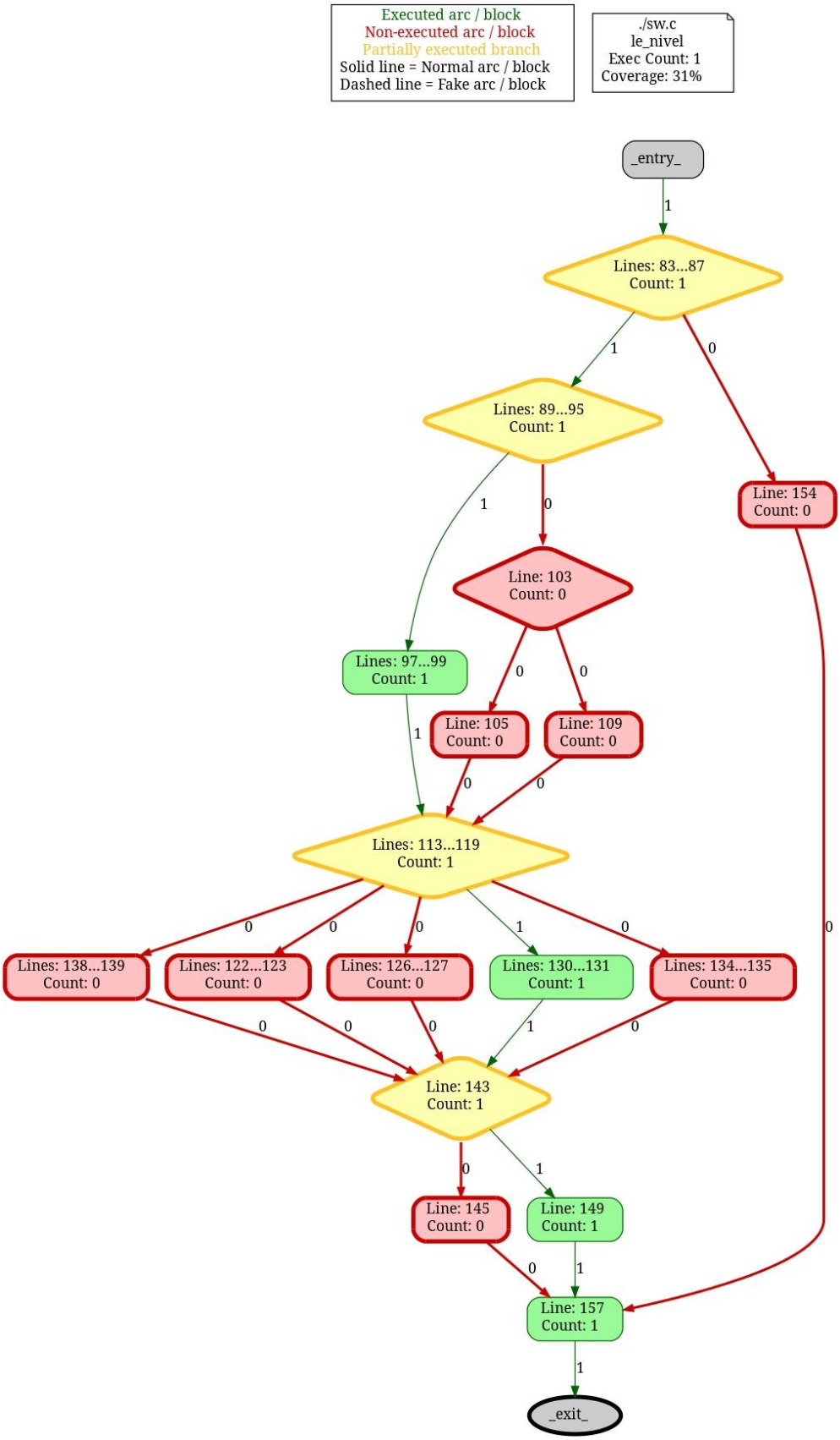

Figura 16 - Cobertura realizada em le_nivel pela função `testa_nivel_reservatorio_igual_NIVEL_MEDIO`

Fonte: Autoria própria.

No próximo caso de teste, foi feito um teste para verificar o nível do reservatório quando apenas o sensor de nível baixo (dados.snbra) está setado.
O resultado é que a variável `nivel_reservatorio` deve retornar o valor da constante `NIVEL_BAIXO`. A cobertura pode ser observada na Figura 18:

```
39  int testa_nivel_reservatorio_igual_NIVEL_BAIXO (void)
40  {
41   FS_BASE_ATUALIZA_NIVEL = 1;
42   tempo_nivel_parado = 10;
43   dados.snbra = 1;
44   le_nivel();
45   mu_assert(0, nivel_reservatorio = NIVEL_BAIXO);
46   return 1;
47  }
```

Figura 17 - Código fonte da função `testa_nivel_reservatorio_igual_NIVEL_BAIXO`

Fonte: Autoria própria.

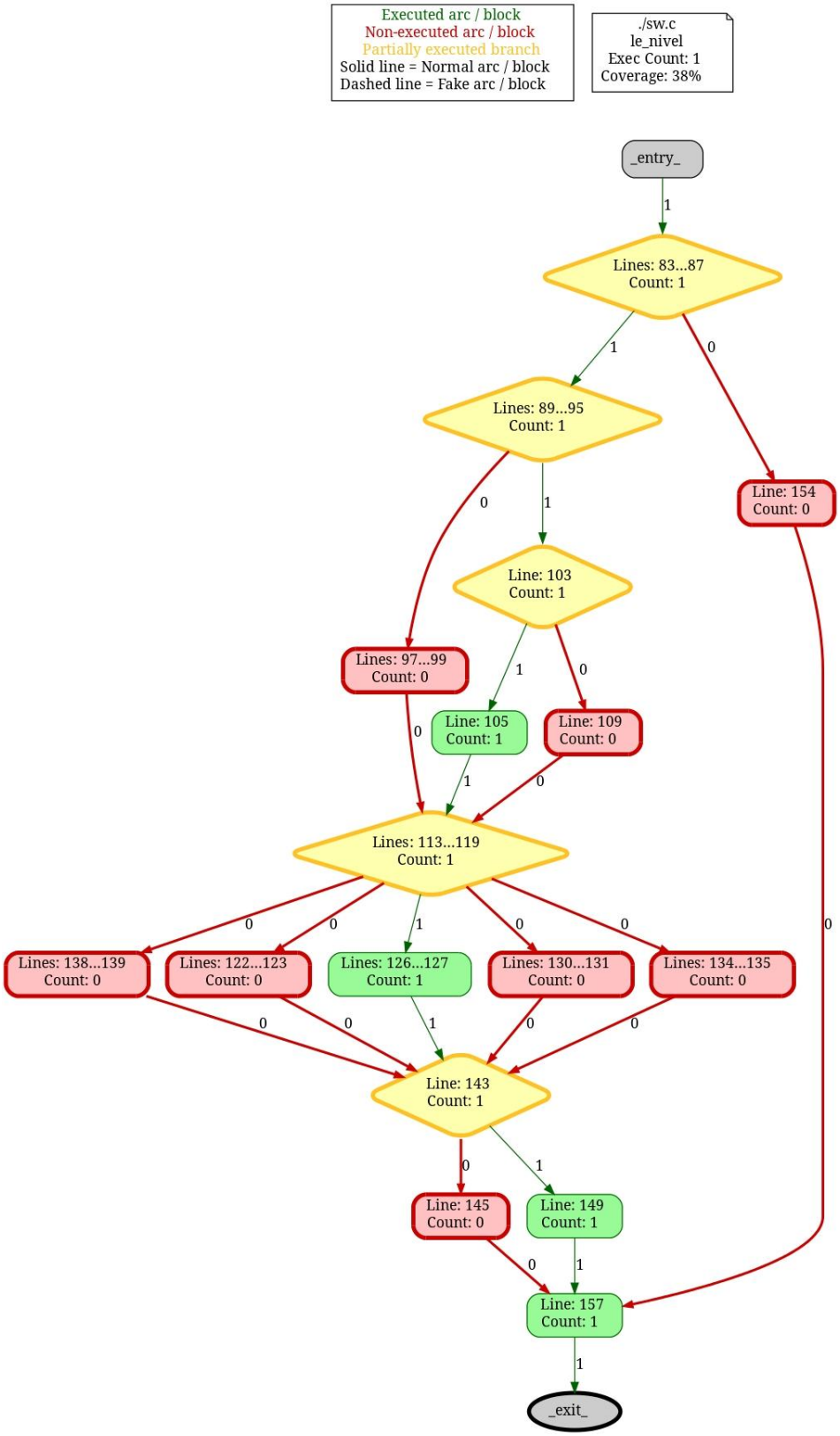

Figura 18 - Cobertura realizada em le_nivel pela função `testa_nivel_reservatorio_igual_NIVEL_BAIXO`

Fonte: Autoria própria.

No quinto caso de teste da Figura 19, verificou-se a situação onde o reservatório de água está completamente cheio, quando todos os sensores de água tem o valor 1 lido. Nessa condição, a variável `nivel_reservatorio` deve apresentar o valor igual ao da constante `NIVEL_ALTO`. Sua cobertura pode ser visualizada na Figura 20:

```
49  int testa_nivel_reservatorio_igual_NIVEL_ALTO (void)
50  {
51   FS_BASE_ATUALIZA_NIVEL = 1;
52   dados.snara = 1;
53   dados.snmra = 1;
54   dados.snbra = 1;
55   le_nivel();
56   mu_assert(0, nivel_reservatorio == NIVEL_ALTO);
57   return 1;
58  }
```

Figura 19 - Código fonte da função `testa_nivel_reservatorio_igual_NIVEL_ALTO`.

Fonte: Autoria própria.

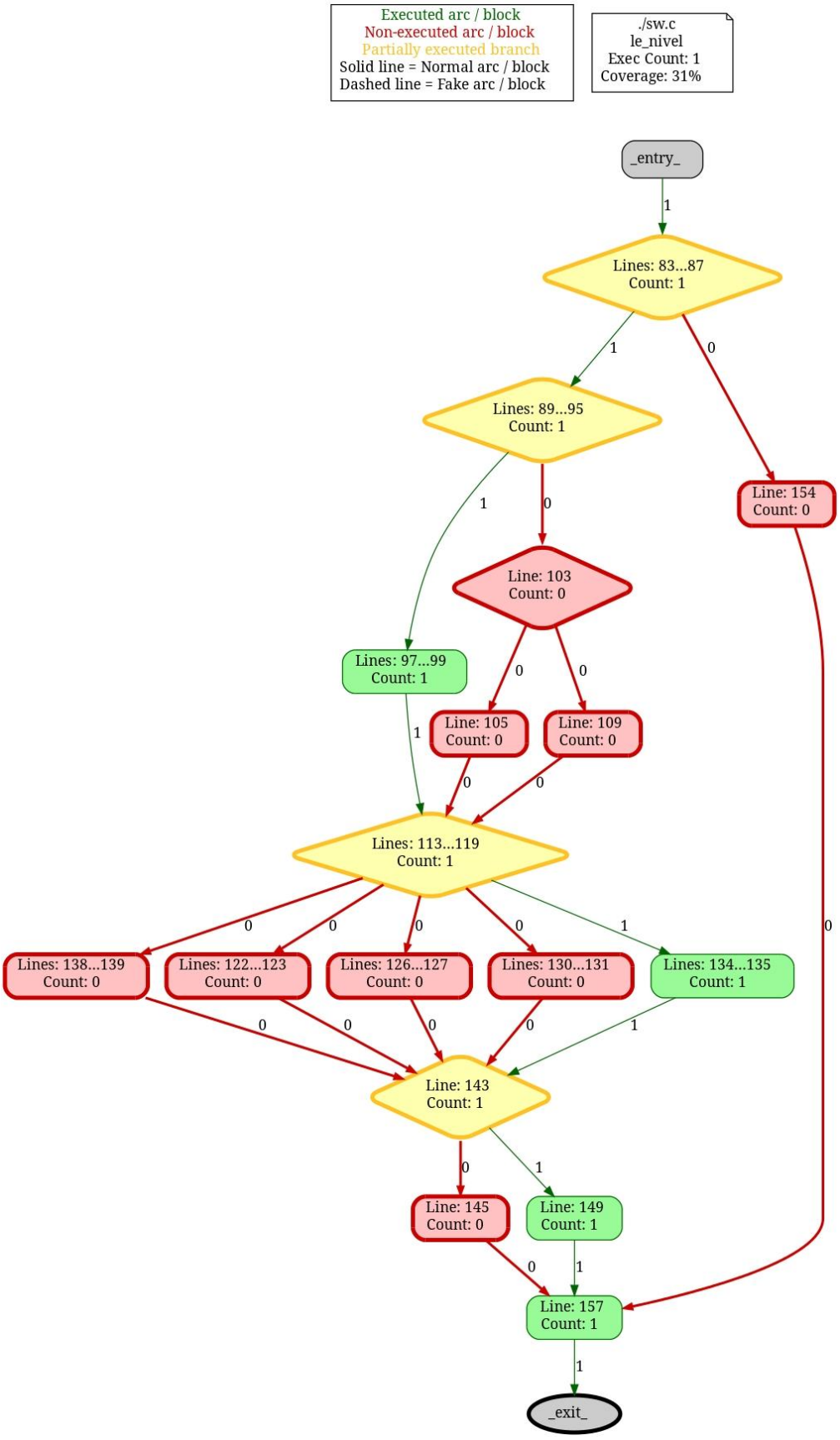

Figura 20 - Cobertura realizada em le_nivel pela função `testa_nivel_reservatorio_igual_NIVEL_ALTO`.

Fonte: Autoria própria.

O último caso de teste (Figura 21) foi utilizado para verificar a execução do programa quanto o flag inicial `FS_BASE_ATUALIZA_NIVEL` está zerado, desviando a condição do programa para o último bloco, linha 154. Seu código pode ser observado na Figura 21, e sua cobertura na Figura 22:

```
60  int testa_tempo_nivel_baixo_igual_TOUT_NIVEL_CONFIGURADO (void)
61  {
62          FS_BASE_ATUALIZA_NIVEL = 0;
63          le_nivel();
64          mu_assert(0, tempo_nivel_baixo ==  TOUT_NIVEL_CONFIGURADO);
65          return 1;
66  }
```

Figura 21 - Código fonte da função `testa_tempo_nivel_baixo_igual_TOUT_NIVEL_CONFIGURADO`

Fonte: Autoria própria.

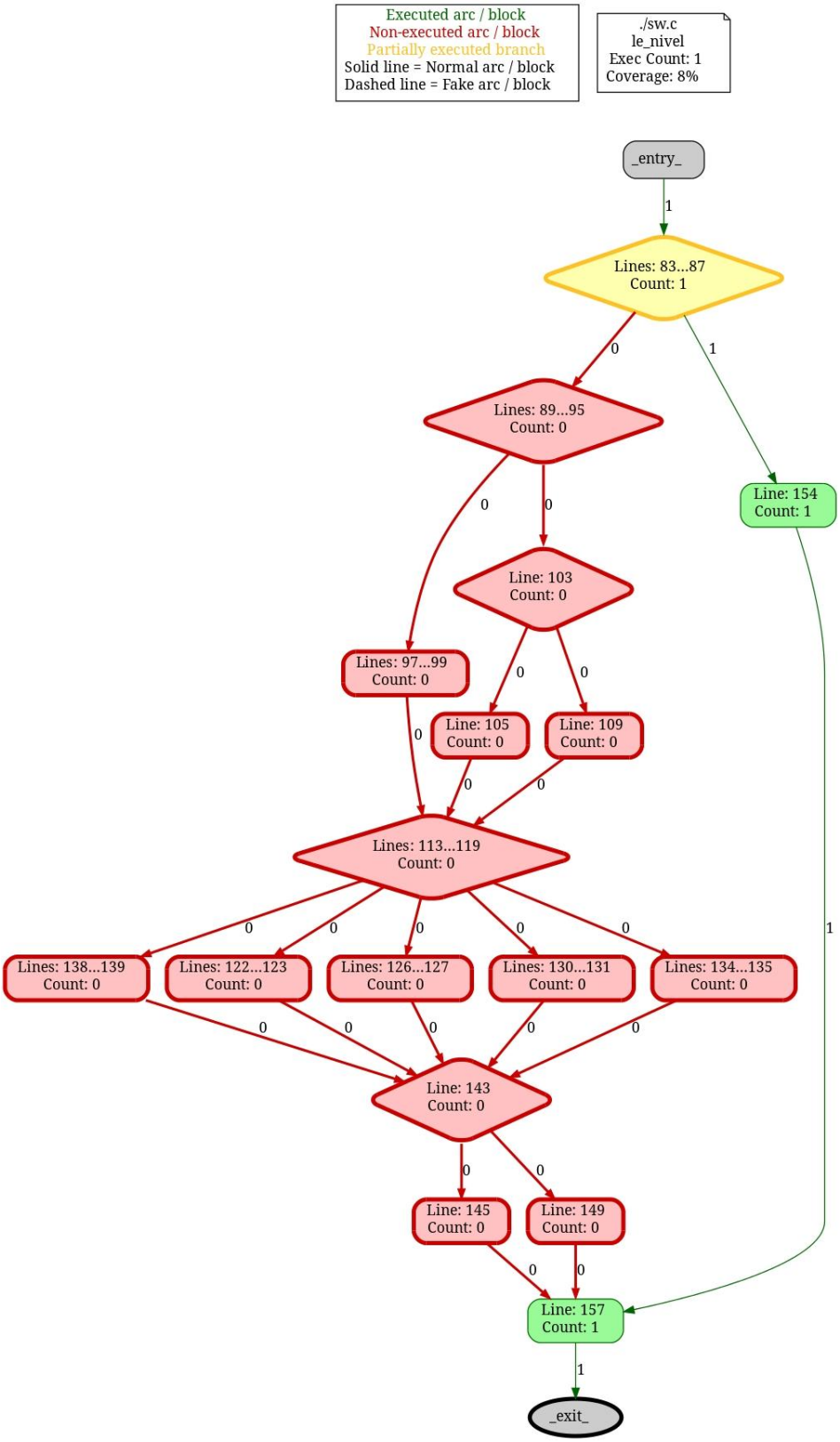

Figura 22 - Cobertura realizada em le_nivel pela função `testa_tempo_nivel_baixo_igual_TOUT_NIVEL_CONFIGURADO`

Fonte: Autoria própria.

Para se ter uma cobertura total do programa, foram realizados todos os casos de teste de uma só vez, alcançando uma cobertura total de 100%. A figura 23 mostra o relatório gerado:

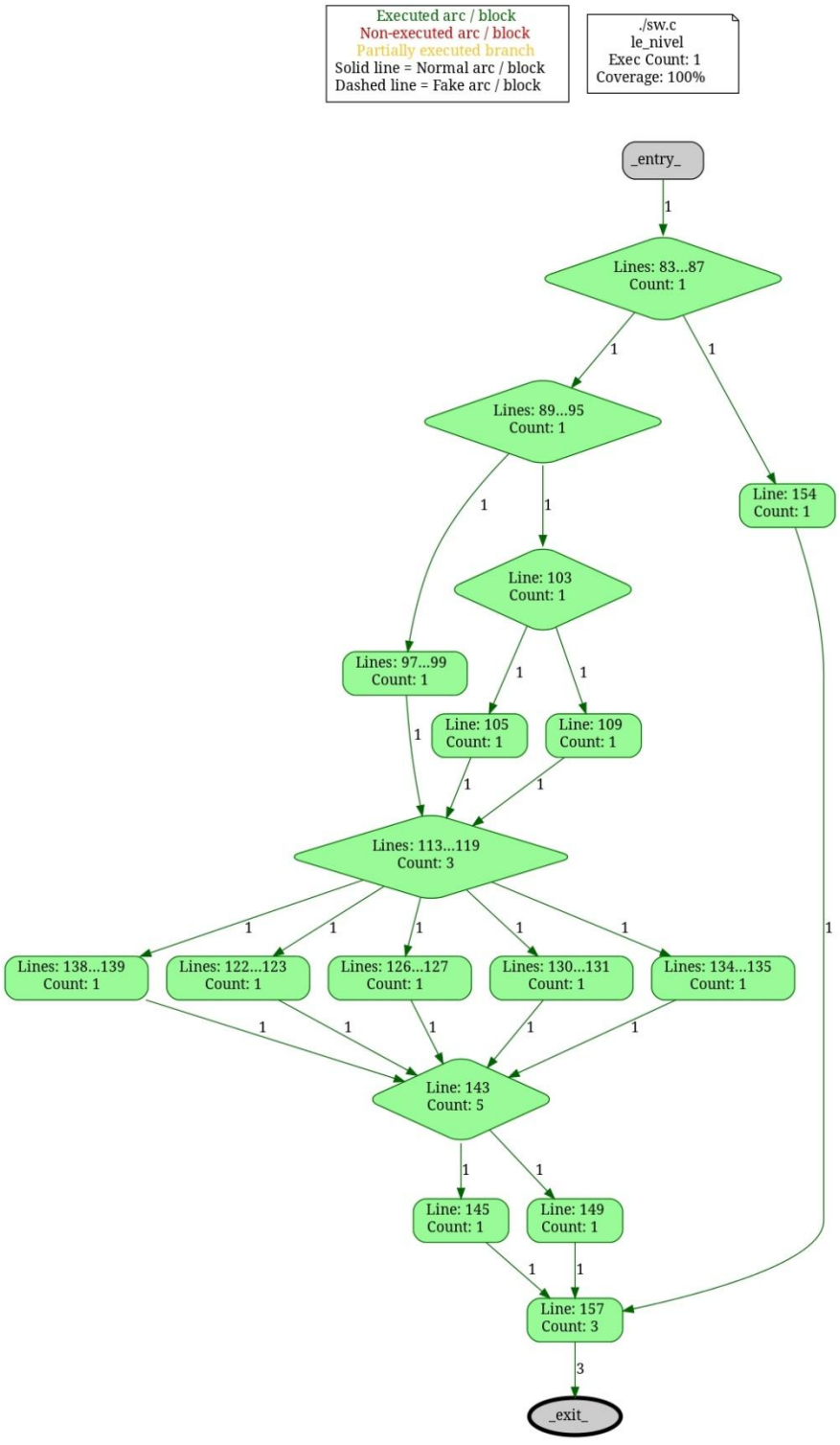

Figura 23 - Cobertura total da função `le_nivel` pela execução de todos os casos de teste apresentados

Fonte: Autoria própria.

Na Figura 24 é apresentado o relatório final, exibido em formato HTML:

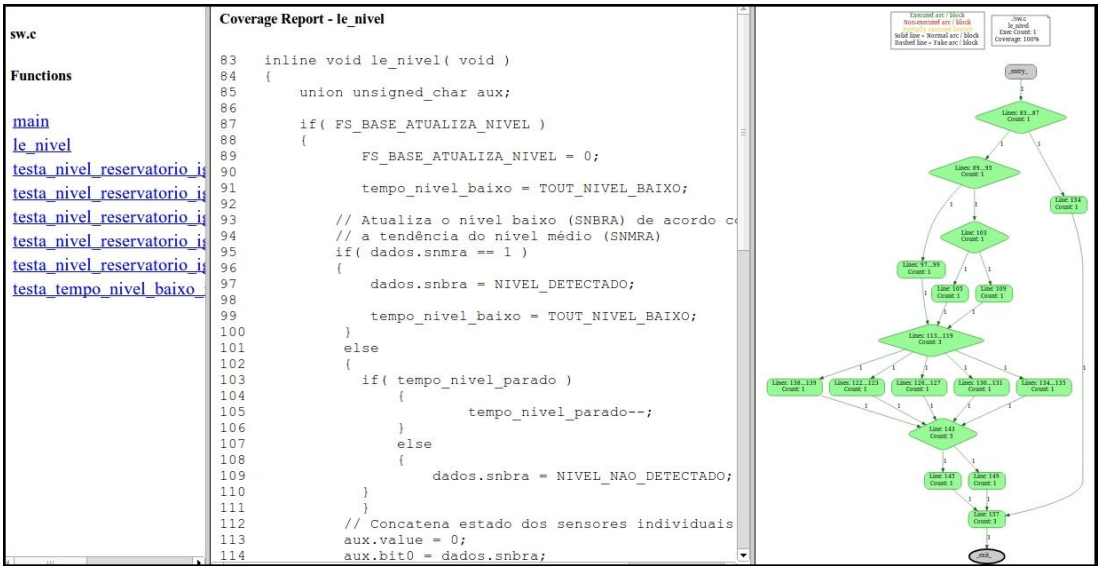

Figura 24 - Relatório final da função le_nivel feito pela ferramenta Crixus-Trucov

Fonte: Autoria própria.

4 CONCLUSÃO

4.1 Considerações Finais

O trabalho de testes de sistemas é algo complexo. Exige um projeto do sistema baseado nessa área, de maneira que os testes precisam ser projetados antes da codificação final do programa. Exige também que haja uma cultura entre os programadores e a empresa, para que exista uma equipe apenas para os testes e outra para a programação.

Esse ambiente ideal é algo complicado de implementar em sistemas computacionais, pois exige toda uma cultura voltada ao desenvolvimento baseado em testes, no ambiente embarcado seu desenvolvimento é muito pior. A ausência de ferramentas livres e funcionais contribui para isso. Existem as proprietárias, mas elas se referem a famílias específicas de microcontroladores que exigem um simulador específico do hardware, limitando as opções do software.

No MPLAB, a IDE utilizada nesse trabalho, existe um recurso que indica os pontos cobertos pelo programa, o MPLAB SIM, porém é um recurso textual pobre, que não indica quantas vezes uma função ou uma linha do programa foi executado. Não permite utilizar o recurso `-fprofile-arcs` do GCC, que é usado para simular o funcionamento do software com o Gcov e gerar um relatório textual contendo as informações de cobertura.

Portanto, para a realização desse trabalho houve a necessidade de fazer uma mescla, emulando o arquivo .gcda através do relatório gerado pelo MPLAB SIM e modificando o código fonte do programa Trucov, conseguindo dessa maneira fazer um relatório visual a partir de um código feito para microcontroladores da família Pic que utilizam a IDE MPLAB.

4.2 Contribuições do trabalho

A ferramenta Crixus-Trucov é importante para a verificação de cobertura do

software. Juntamente com ela e aplicada à testes de unidade pode-se garantir os pontos que foram cobertos no programa. Como é uma ferramenta open source, seu código é aberto e qualquer programador pode efetuar o download e modificá-la. Ainda possui algumas limitações, mas essas não impedem de realizar a cobertura quando executada.

Se a empresa/equipe optar por trabalhar na codificação visando realizar testes finais no software, com a ferramenta Crixus-Trucov se torna mais simples a verificação dos blocos/arcos cobertos. Verificando assim, quais são os testes que precisam ainda ser realizados.

4.3 Trabalhos Futuros

Como a ferramenta Crixus-Trucov possui algumas limitações ainda, existem alguns trabalhos a serem feitos no futuro:

1) O arquivo header do microcontrolador que vai anexado no projeto do MPLAB envia um erro quando o compilador é chamado sem enviar a diretiva `-mcpu=<modelo pic>-x c -c`. Para resolver esse problema, foi necessário copiar o header do modelo do microcontrolador para a pasta do projeto (o header se localiza no diretório padrão do compilador C30) e alterar a linha que lança o erro: “#error "Include file does not match processor setting"”. É necessário estudar melhor as opções do gcc e procurar por uma opção que envie algo similar a “mcpu”;

2) Atualmente, a ferramenta funciona apenas para o sistema operacional Linux. É necessário recompilar o programa para que também venha a funcionar na plataforma Windows.

3) Como atualmente está sendo cada vez mais usado o Mplabx (nova IDE, com a finalidade de substituir o Mplab), é interessante criar um plugin que incorpore o Crixus-Trucov.

4) Implementação de um relatório de forma textual, ao invés do relatório visual.

5) Verificar a eficácia de testes estruturais com a ferramenta num ambiente onde não é aplicada a atividade de testes e compilar os resultados.

5 REFERÊNCIAS BIBLIOGRÁFICAS

ALMEIDA J., Jorge. R. **Segurança em sistemas críticos e em sistemas de Informação: um estudo comparativo**. 2003. 191 f. Tese de Livre Docência. Escola Politécnica da USP, São Paulo: São Paulo. Disponível em: <www.cos.ufrj.br/~taba> Acesso em: 08 de nov. 2011.

ASSOCIAÇÃO BRASILEIRA DE NORMAS TÉCNICAS. **NBR IEC 60601-1-1**: informação e documentação: referências: elaboração. Rio de Janeiro, 2007.

ASSOCIAÇÃO BRASILEIRA DE NORMAS TÉCNICAS. **NBR IEC 60601-1-2**: informação e documentação: referências: elaboração. Rio de Janeiro, 2009.

BARBOSA, E.; MALDONADO, J.C.; VINCENZI, A.M.R.; DELAMARO, M.E; SOUZA, S.R.S. e JINO, M.. "**Introdução ao Teste de Software**. XIV Simpósio Brasileiro de Engenharia de Software", 2000.

BEIZER, B. **Software testing techniques**. 2nd ed. New York: Van Nostrand Reinhold Company, 1990.

BERGER, A.**Embedded Systems Design: An Introduction to Processes, Tools and Techniques**, CMP Books, 2001.

BOSCH, ROBERT **CAN Specification 2.0**. 1991. 73f. RelatórioTécnico. Chuck Powers, Motorola MCTG Multiplex Applications. Stuttgart, Germany, 1991.

BROEKMAN, B; NOTRNBOOM, E. **Testing Embedded Software**. Addison-Wesley Professional, 2002.

BSTQB, Brazilian Software Testing Qualifications Board - **Certificação em Teste Advanced Level Syllabus**, Norma Brasileira para certificação de testes em sistemas críticos, Versão 2007br.

CODE COMPOSER, Homepage. Disponível em <http://www.ti.com/tool/ccstudio>>. Acesso em: 12 out. 2012, 16:17.

DEMILLO, R. A. **Software testing and evaluation**. The Benjamin/Cummings Publishing Company, Inc., 1987.

EBERT, C.; SALECKER, J. **Introduction: Embedded Software Technologies and Trends**, Software, IEEE, vol.26, no. 3, pp.14-18, May-June 2009.

ENGBLOM, J.; GIRARD, G.; WERNER, B. **Testing Embedded Software Using Simulated Hardware**. In: 3RD EMBEDDED REAL-TIME SOFTWARE CONFERENCE, ERTS, 2006.

GCC, Disponível em: <http://gcc.gnu.org/> Acesso em: 14 ago. 2012, 21:24.

GOMES, V, H. **Metodologia de Projeto de Software Embarcado Voltada ao Teste.** Universidade Federal do Mato Grosso Do Sul, 2010.

HAGAR, D. **Testing Critical Software: Practical Experiences**. Department of Computer Science, University of Texas at Dallas, 1998.

HAGEN, V. W. **The Definitive Guide to GCC**. Apress, New York, 2006.

HOWDEN, W. E. **Software engineering and technology: Functional program testing and analysis**. New York: McGrall-Hill, 1987.Acesso em: 12 out. 2011, 14:14.

IAR SYSTEMS, Homepage. Disponível em <http://www.iar.com/>Acesso em: 12 out. 2012, 16:19.

IEEE Standard Glossary of Software Engineering Terminology.**IEEE Std610.121990**: informação e documentação: referências: elaboração. New York, 1990.

JORGE, F, R. **Teste de Softwares Embarcados**, CED-UFMS, Mato Grosso do Sul - MS, 2010 – Disponível em: <http://www.ic.unicamp.br/~rodolfo/Cursos/.../099173-A.pdf > Acesso em: 10 de nov. 2012.

KARLESKY, M.; BEREZA, W.; ERICKSON, C. **Effective Test Driven Development for Embedded Software**. Atomic Object LLC, USA, 2007.

LEVESON, N.; TURNER, J.**An investigation on the Therac-25 accidents.** IEEE Computer, vol. 26, no.7, pp. 18-41, 1993.

LINNENKUGEL, U.; MÜLLERBURG, M. **Test data selection criteria for integration testing**. In: First International Conference on Systems Integration, Morristown, NJ, 1990, p.709–717.

MALDONADO, J. C. **Critérios potenciais usos: Uma contribuição ao teste estrutural de software**. Tese de Doutoramento, DCA/FEE/UNICAMP, Campinas, SP, 1991.

MALDONADO, J. C.; VINCENZI, A. M. R.; BARBOSA, E. F.; SOUZA, SM. E. **Aspectos teóricos e empíricos de teste de cobertura de software**. 1998. 28f. Relatório Técnico. Instituto de Ciências Matemáticas e de Computação ICMC-USP. São Carlos, 1998.

MICROCHIP Homepage. Disponível em: <http://www.microchip.com/> Acesso em: 12 set. 2012, 16:04.

MYERS, G. J. **The art of software testing**.Wiley, New York, 1979.

PRESSMAN, R. S. **Software engineering – a practitioner's approach**.5 ed. McGraw-Hill, 2000.

QIAN, YANG, J., JENNY, LI and DAVID WEISS. 2006. **A survey of coverage based testing tools**. In Proceedings of the 2006 international workshop on Automation of *software test* (AST '06). ACM, New York, NY, USA, 99-103.

ROCHA, A. R. C., MALDONADO, J. C., WEBER, K. C. et al., **Qualidade de software – Teoria e prática**, Prentice Hall, São Paulo, 2001.

SAEED, A.; LEMOS, R.; ANDERSON, T. **The role of formal methods in the requirements analysis of safety-critical systems: a train set example**. In: IEEE INTERNATIONAL SYMPOSIUM ON FAULT TOLERANT COMPUTING, 2, Montreal, Canadá, 1991, p.478-85.

SAUBERN Homepage. Disponível em: <http://www.saubern.com.br/> Acesso em: 12 set. 2012, 15:53.

SEO, J.; SUNG, A.; CHOI, B.; KANG, S. **Automating Embedded Software Testing on an Emulated Target Board**. In: SECOND INTERNATIONAL WORKSHOP ON AUTOMATION OF SOFTWARE TEST, AST. Proceedings Washington: IEEE Computer Society, 2007, p. 1-7.

SOMMERVILLE, I.**Engenharia de Software** 8ªedição, São Paulo, Pearson Edison Wesley, 2008.

SOUZA, S. R. S. **Avaliação do custo e eficácia do critério análise de mutantes na atividade de teste de programas**. Dissertação de Mestrado, ICMC-USP, São Carlos – SP, 1996.

TEXAS INSTRUMENTS, Homepage. Disponível em <http://www.ti.com/ >. Acesso em: 12 out. 2012, 16:23.

TIAN, P.; WANG, J; LENG, H; QIANG, K. **Construction of Distributed Embedded Software Testing Environment**, International Conference on Intelligent, 2009.

TRUCOV, Homepage. Disponível em: <http://code.google.com/p/trucov> Acesso em: 14 ago. 2012, 21:23.

VINCENZI, A. M. R. **Subsídios para o estabelecimento de estratégias de teste b, a técnica de mutação**. Dissertação de Mestrado, ICMC-USP, São Carlos – SP, 1998.

VINCENZI, A. M. R.; MALDONADO, J. C.; BARBOSA, E. F.; DELAMARO, M. E. **Unit and integration testing strategies for C programs using mutation-based criteria**. Software Testing, Verification and Reliability, v. 11, n. 4, 2001.

WONG, W. W.; DEBROY, V.; SURAMPUDI, A. HYOEONJEONG, K.**Recent Catastrophic Accidents: Investigating How Software Was Responsible**. Department of Computer Science.University of Texas at Dallas, 2010.

ANEXO A – Documento de trabalho – Diário de trabalho

O trabalho inicial consistia em verificar a eficácia de testes de estruturais num ambiente onde são feitos apenas testes de caixa preta. Para isso, utilizaria-se um sistema médico embarcado, e após executar os testes estruturais, seria feita essa comparação. Porém, ao procurar por ferramentas, verificou-se que no mercado não existiam ferramentas gratuitas para a plataforma PIC, utilizada nesse ambiente. Por conta disso, foi necessário criar uma ferramenta que fizesse isso. Abaixo segue a sequência de atividades feitas no trabalho:

1) Procurar por ferramentas: O primeiro ponto foi procurar ferramentas que pudessem executar os casos de teste e dessem a cobertura. Ao mesmo tempo, pesquisar por ferramentas de teste de unidade. A primeira alternativa foi utilizar um plugin do eclipse, chamado PIC C Builder for Eclipse trabalhando em conjunto com a ferramenta de cobertura Gcov. Porém, não foi possível executar o código, o plugin apresenta muitos problemas e está descontinuado, impossibilitando inclusive suporte;
2) A alternativa foi utilizar uma nova ide, fornecida pela Microchip, o MplabX. Essa ferramenta é baseada na Ide Netbeans, possuindo todos os artifícios de auxílio ao código e debug. Mas ainda não é possível integrar os plugins do Netbeans de teste, portanto, não houve progresso nos testes utilizando o MplabX;
3) Sobre os testes de unidade. Para a realização dos testes de unidade, a primeira alternativa foi utilizar a ferramenta Cunit, mas o que foi observado é que sua utilização funciona apenas para código em linguagem C. Para os microcontroladores PIC, fica inviável, pois seus recursos de comparação exigem muito processamento, já que seus testes são todos feitos com strings e o microcontrolador possui pouca memória de programa, impossibilitando o seu uso.
4) Após várias pesquisas, foi encontrada uma ferramenta que pudesse fazer o teste de unidade. É o Minunit, um framework muito simples que pode ser utilizando para a criação de testes. Funcionou bem, sendo possível testar várias funções e variáveis, utilizando o Mplab Sim como partida no programa.

5) O problema agora era fazer algo que mostrasse a cobertura do software. As ferramentas mais conhecidas da comunidade open source eram o Gcov e o Lcov. Gastou-se algum tempo pesquisando-as, e acabou por descobrir que era necessária inicialmente uma compilação feita com o GCC, utilizando os parâmetros `-fprofile-arcs -ftest-coverage`. A saída seria dois arquivos, o `.gcno` e o `.gcda`. O primeiro indica a estrutura do software, contendo os blocos e arcos que o compõe. O segundo contém o fluxo de funcionamento da última execução do programa. Os programas de cobertura utilizam desses dois arquivos para gerar uma imagem ou um arquivo de texto contendo os dados de execução do programa.

6) O compilador C30 é baseado no GCC. Partindo dessa ideia, imaginou-se que poderia ser executado utilizando desses parâmetros para a criação dos dois arquivos. Mas ai que começaram os problemas. Apenas o arquivo `.gcno` era criado, impossibilitando a criação do relatório final. Tentamos ainda a utilização de outros parâmetros, encontrados na documentação do GCC[4]: `-fdump-tree-original-raw`, `-fdump-translation-unit`, `-fdump-tree-switch`, `-fdump-tree-cfg`, `-fdump-tree-vcg`, `-o`, `-dv`. Porém, nenhum deles gerou resultados que pudesse ser aproveitado.

7) Teve-se então a ideia de criar um simulador de arquivos `.gcda`. Depois de muita procura, foi encontrada uma ferramenta de código aberto chamada Trucov. É um software de código aberto, escrito em linguagem C++, semelhante ao Gcov. Capaz de gerar relatórios textuais e gráficos sobre cobertura de código escrito em linguagem C.

8) A ideia então foi estudar o código, e assim, verificar qual ponto do código do Trucov é lido o arquivo .gcda, e assim realizada a cobertura. Foi aberto o projeto inicialmente no Netbeans, mas sem muito sucesso na depuração. Obteve-se uma leitura melhor dos dados utilizando a IDE QtCreator. Foi descoberto que a leitura é feita na função `assign_arc_counts()`, do arquivo "Parser.cpp". Essa função lê o arquivo `.gcda` e joga os dados num vetor, assinalando a quantidade de vezes que a execução do programa passou em determinado bloco e arco. Todos os pares bloco/arco começam com a contagem zerada, sendo setado a quantidade de

[4] http://gcc.gnu.org/onlinedocs/gcc-4.7.1/gcc.pdf

vezes conforme o que diz o arquivo `.gcda`.

9) Depois de encontrado esse ponto, foi implementado nesse arquivo uma função para ler um arquivo chamado “texto.cr”, que deve existir no diretório que contém o arquivo fonte que se está sendo chamado. No arquivo deve conter o número da função que deseja ser executado (um arquivo fonte pode conter varias funções), o numero total de blocos/arcos e os pares blocos e arcos por onde o programa passou.

10) Nesse ponto, chegou a hora de implementar um programa executável que fosse capaz de ler o arquivo de relatório do Mplab SIM e criar um relatório gráfico na saída, criando para isso o arquivo “texto.cr”, chamando o Trucov modificado e criando o gráfico na saída.

11) A maior dificuldade estava agora em saber como criar a numeração referente ao bloco/arco em que o programa passou. Tentou-se verificar qual o padrão havia em vários relatórios aleatórios criados, mas sem sucesso. Também verificar por meio da diretiva “report” o resultado na saída textual do Trucov, mas também sem sucesso. Depois de explorar todos os parâmetros disponíveis na ferramenta, havia um que surtia o resultado esperado. É o parâmetro `-d`, utilizado como debug. Através dele, é gerado um arquivo de saída, com a extensão `.dump`. Nele, há a descrição dos arcos e blocos que compõe o código analisado, através da leitura do arquivo .gcno.

12) Agora, era necessário juntar tudo o que foi feito separado e criar um executável capaz de ler o código fonte e o arquivo de SIM, gerando em sua saída um relatório HTML. A sequência de execução do programa ficou:

12.1) O programa deve sempre verificar se o usuário passou os dois arquivos como parâmetro. Caso contrário, fica impossível a geração dos relatórios;

12.3) O programa então cria um script que gera um executável com o nome “criaGcno.sh”. Seu conteúdo é:

`#!/bin/bash \n gcc -x c -c./arquivoFonte.c -w -fprofile-arcs -ftest-coverage`. Depois de executado, um arquivo `.gcno` é criado.

12.4) Depois, o programa copia um arquivo `.gcda` padrão vazio para o diretório do arquivo fonte e o renomeia para o nome do arquivo. É

necessário ter esse arquivo para a criação do arquivo `.dump`, executado pelo Trucov com o parâmetro `-d`. Sem ele, é dada uma mensagem de erro.

12.5) Com os dois arquivos recém-criados no diretório, é então executado o Trucov com a diretiva `-d`. Um arquivo `.dump` é gerado contendo as informações dos blocos e dos arcos que formam o programa.

12.6) O Crixus-Trucov agora lê o arquivo `.dump` e verifica quais são os blocos e arcos que compõem o programa, bem como o nome de todas as funções que o compõem, separando as linhas que formam cada bloco e a linhas que fazem parte dos arcos. Captura os dados e joga numa lista dinâmica.

12.7) O Crixus-Trucov verifica as linhas que foram executadas no Mplab SIM, através do arquivo SIM passado como parâmetro, separando-os e colocando numa lista dinâmica.

12.8) O Crixus-Trucov então verifica em qual bloco e arco pertence a linhas executadas e joga numa nova lista dinâmica.

12.9) Criação do arquivo `texto.cr`. Depois de ter os dados separados, o Crixus-Trucov escreve no arquivo quais são todos os arcos que formam a função. Logo, escreve quais são os blocos e arcos que devem ser setados como aqueles em que o programa passou.

12.11) O programa então verifica se há um diretório chamado `crixus_files` dentro do diretório do arquivo fonte. Se não houver, um novo será criado. Esse diretório é onde ficarão todos os arquivos criados após a execução do Crixus-Trucov.

12.10) É então criado um script, chamado `scriptCriaImagem.sh`, contendo em seu conteúdo as instruções:

`#!/bin/bash \n trucov graph + nomeFuncao`. É nessa hora que o Trucov irá ler o arquivo `texto.cr` e criar um gráfico chamado “coverage.svg”. O Crixus-Trucov renomeará essa imagem para o nome da função cuja cobertura foi feita e colará no diretório crixus_files.

12.13) O programa faz a sequência das operações 12.7 a 12.10 para cada função do arquivo fonte passado como parâmetro. No final, fica um

arquivo gráfico para cada função.

12.14) Exclusão dos arquivos temporários: Depois do relatório feito da última função, os arquivos temporários são todos deletados. Isso inclui o `.gcda, o .gcno`, os scripts e os arquivos do Trucov.

12.15. Contendo todas as funções, é então gerado um relatório HTML no qual o usuário pode visualizar todas as funções do arquivo fonte, sua implementação e o relatório gráfico de sua execução, obtida pelo arquivo SIM passado como parâmetro.