MEF e TDD para Sistemas Embarcados: Uma Abordagem Básica e Ferramental Integrado

Rogerio Atem de Carvalho, Hudson Silva Ferreira, Rafael Ferreira Toledo, Milena Silveira de Azevedo

Núcleo de Pesquisa em Computação Científica (NC^2), Centro de Referência em Sistemas Embarcados e Aeroespaciais (CRSEA), Instituto Federal Fluminense (IFF) Campos dos Goytacazes, Brasil

ratem@iff.edu.br, silvaferreira.hsf@gmail.com, rafaelt91@gmail.com, milena.slvr@gmail.com

Abstract. *The evolution of information technology and electronics in general has been consistently increasing the use of embedded systems. While hardware development for these systems is already consistent, software development for embedded systems still lacks a consolidated methodology. This short paper describes a process and toolset for Embedded Systems Modeling and Verification using FSM (Finite State Machines) and TDD (Test-Driven Development).*

Resumo. *A evolução da tecnologia da informação e da eletrônica tem de forma geral aumentado consistentemente o uso de sistemas embarcados. Enquanto o desenvolvimento de hardware neste tipo de ambiente já se encontra bastante consistente, o desenvolvimento de software para sistemas embarcados ainda necessita de uma metodologia consolidada. Este trabalho descreve brevemente um processo e um conjunto de ferramentas para Modelagem e Verificação de Sistemas Embarcados usando MEF (Máquina de Estados Finitos) e TDD (Test Driven Development).*

1. Introduction

A evolução da tecnologia da informação e da eletrônica tem de forma geral aumentado consistentemente o uso de sistemas embarcados. Esses sistemas podem ser definidos como dispositivos computacionais de propósito específico, usualmente integrados a um sistema externo que desempenha alguma função específica. Atualmente, os sistemas embarcados possuem várias aplicações, tais como, gestão de tráfego, navegação de automóveis, controle de processos industriais etc. Assim, um sistema embarcado apresenta as seguintes características [Grenning 2011]:

- Suas funcionalidades e desempenhos são profundamente dependentes da integração entre elementos mecânicos, eletrônicos e a tecnologia de hardware e de software.
- Usualmente tem recursos de hardware limitados, como tamanho de memória e capacidade de processamento.

- Deve ser robusto, ou seja, seu comportamento deve ser rigidamente controlado mesmo durante falhas no sistema.

A especificação de requisitos sistêmicos é uma etapa complexa e deve ser verificada diversas vezes durante o tempo de vida do projeto. A fim de facilitar essa etapa, os requisitos são obtidos através de notações usadas para representar processos de negócios.

Nesse sentido, as Máquinas de Estados Finitas (MEF) representam uma notação amplamente escolhida para modelagem de sistemas embarcados.
A MEF modela um processo identificando em quais estados o sistema pode estar, quais entradas ou eventos disparam a transição de estados e como o sistema irá se comportar em cada um desses estados. Nesse modelo, a execução do software é vista como uma sequência de transições que desloca o sistema por seus vários estados [Wright 2005].

O objetivo deste documento é apresentar um ferramental e um método para facilitar o desenvolvimento da parte de software de sistemas embarcados. Esse ferramental concerne tanto a codificação como a validação e modelagem sistêmica. A fim de alcançar esse objetivo, este documento apresentou uma introdução a sistemas embarcados e o objetivo do presente trabalho (seção 1), descreve brevemente um processo proposto (seção 2) e o ferramental desenvolvido (seção 3), bem como apresenta algumas conclusões e direções atuais e futuras de pesquisas no desenvolvimento de sistemas embarcados (seção 4).

2. Processo Proposto

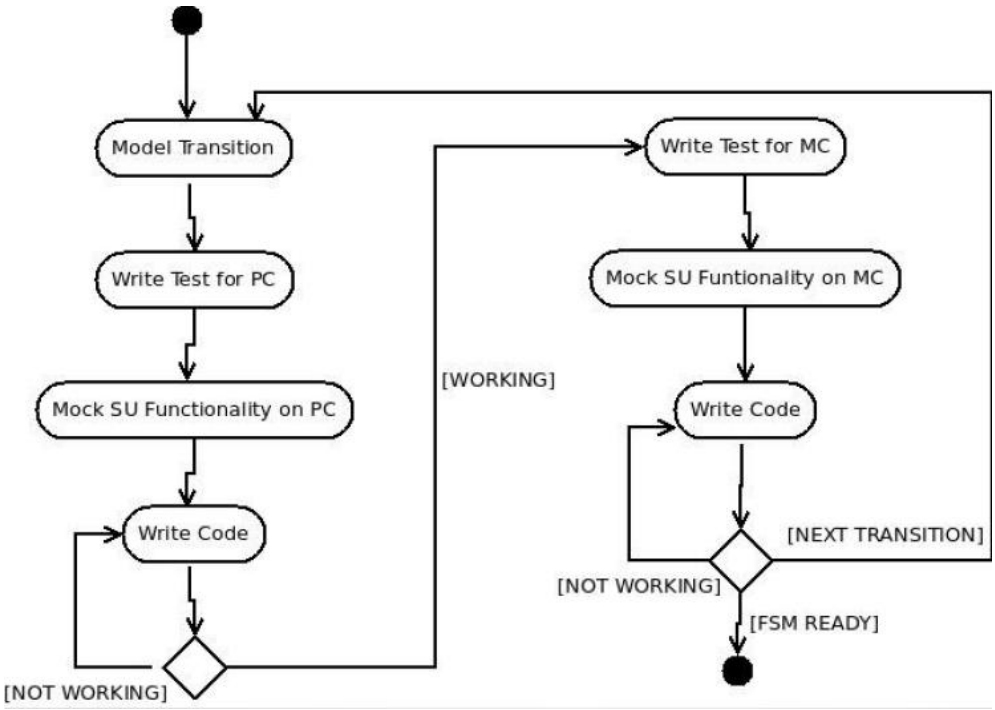

Figura 1. Processo de Desenvolvimento Proposto

A figura 1 apresenta o processo proposto para o desenvolvimento de sistemas embarcados utilizando TDD. Primeiramente, o fluxo de desenvolvimento começa com a modelagem de uma única transição da MEF que representa o sistema embarcado, assim, um único teste é criado para essa transição, usando uma arquitetura de computador para executá-lo. Ainda com o hardware de um computador padrão, o próximo passo é simular o comportamento do hardware (SU[1]) através do software. Finalmente, o comportamento do hardware é simulado em um microcontrolador, com o objetivo de garantir uma simulação mais fiel do mesmo. Desse modo, é possível desenvolver o software e o hardware de forma paralela, ou desenvolver o primeiro antes do segundo.

2.1 Simulação

Além da modelagem, Yakindu também possibilita a simulação do comportamento do sistema, utilizando a notação MEF. A figura 2 mostra a simulação de um modelo executado por Yakindu. No exemplo apresentado, o modelo é composto por três estados (State1, State2 e State3). Além disso, para cada estado é esperado o recebimento de informações. Se as informações de uma dada entrada satisfizerem as condições de transição, esse evento de transição será disparado. O próximo estado a ser ativado é indicado pelo sentido das setas. Na figura abaixo, o estado atual é o State2, evidenciado pela cor vermelha.

A simulação, sem a necessidade de código, ajuda o usuário a identificar falhas na especificação dos requisitos sistêmicos, uma vez que a ideia principal do modelo pode ser validada e simulada através de recursos visuais.

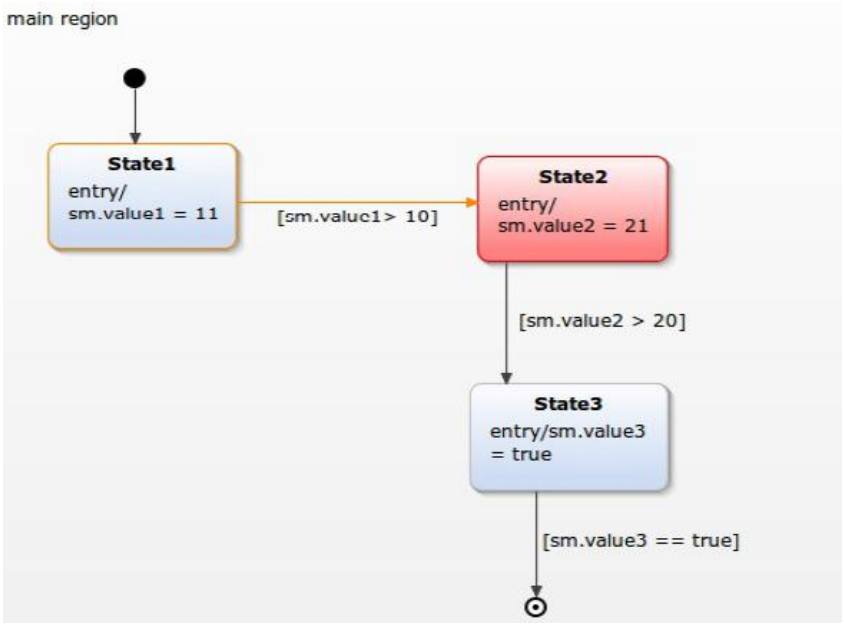

Figura 2. Exemplo de Simulação no Yakindu

2.2 Geração de Código Fonte

Com a informação fornecida pelo usuário, o módulo Test Suite gera o código responsável por executar a simulação da MEF e por verificar as assertivas do comportamento esperado. A figura 3 apresenta um exemplo de arquivo com os testes automáticos gerados.

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include "src-gen/sc_types.h"
4  #include "src-gen/Sm.h"
5  #include "testinglib/gtest/gtest.h"
6
7  class TestStatemachine: public ::testing::Test {
8      protected:
9      Sm handle;
10     SmStates sm_main_region_State1;
11     SmStates sm_main_region_State2;
12     SmStates sm_main_region_State3;
13     SmStates sm_main_region__final_;
14 };
15
16 TEST_F(TestStatemachine, testsm) {
17     sm_init(&handle);
18     sm_enter(&handle);
19
20     EXPECT_TRUE(sm_isActive(&handle, sm_main_region_State1));
21
22     smIfaceSm_set_value1(&handle, 13);
23     EXPECT_TRUE(sm_isActive(&handle, sm_main_region_State2));
24
25     smIfaceSm_set_value2(&handle, 54);
26     EXPECT_TRUE(sm_isActive(&handle, sm_main_region_State3));
27
28     smIfaceSm_set_value3(&handle, true);
29     EXPECT_TRUE(sm_isActive(&handle, sm_main_region__final_));
30 }
```

Figura 3. ArquivoTestStateMachine.cpp.

A assertiva EXPECT_TRUE verifica se o parâmetro de entrada da função é igual a *true*. O conjunto de comandos contidos no arquivo consiste no repetitivo processo de verificação do estado de origem, de execução das condições de transição e de verificação do estado de destino até que os dados de entrada do usuário tenham terminados.

3. Ferramental

O ferramental é baseado em ferramentas de software livre, desenvolvidas por comunidades, como Yakindu [Itemis 2015] e Google C++ Testing Framework [Google 2015] e outras desenvolvidas pelos autores, como Clover e Test Suite.

3.1. Yakindu

O Yakindu Statechart Tools (SCT) consiste em um ambiente de modelagem integrado para especificação e desenvolvimento de sistemas reativos, guiado por eventos, usando o conceito de Máquina de Estados Finitos (MEF). Essa ferramenta de fácil utilização inclui edição gráfica sofisticada, validação e simulação, assim como geração de código para Java, C e C++ [Itemis 2015]. Yakindu foi utilizado como um plugin do Eclipse (uma plataforma de desenvolvimento de software baseada em Java) para geração de esqueletos de testes, integração com a biblioteca de testes e execução de testes durante a simulação.

3.2 Clover

Uma das técnicas básicas do TDD é o uso de Objetos Dublês (Test Doubles). Quando um teste é escrito e o uso de sua real dependência não é possível, ou não é desejada, o mesmo pode ser substituído por um dublê [Freeman 2004]. Contudo, os sistemas embarcados são usualmente implementados em linguagem C, a qual não é do tipo orientada a objetos, portanto, nesse caso, dublês são usados não para substituir objetos, mas chamadas de funções. Com este propósito, os autores desenvolveram uma biblioteca chamada de C Libraries Overloading (Clover). Clover é uma ferramenta poderosa com as seguintes capacidades:

- Qualquer função C pode ser sobrecarregada, desde que uma implementação do dublê seja fornecida para isso.

- Clover é uma Biblioteca Dinâmica (Dynamic Lybrary), portanto, não é necessário introduzir nenhuma mudança no código de produção (Unidade Sob Teste - UST), e essa é ligada em conjunto ao UST somente durante os testes.

- Chamadas de funções podem ser duplicadas em várias situações: durante a execução de um dado fragmento de código, durante todo o tempo de execução, ou em uma dada chamada específica. Desse modo, chamada de funções que são usadas para implementar outras funções de biblioteca, como malloc() ou fprintf(), não sofrem interferência e funcionam normalmente.

A Figura 4 mostra um exemplo do uso de Clover testando malloc(), set_status() é usada para interceptar as chamadas de malloc() e reorientá-las para o comportamento duplicado.

```
void it_overrides_malloc(void){
    void* pt;

    pt = allocate(10);
    assert( pt != NULL );
    free(pt);
    set_status(_malloc, 1, 0, NULL);
    pt = allocate(10);
    assert( pt == NULL );
    //Checks deactivation
    pt = allocate(10);
    assert( pt != NULL );
    free(pt);
    //Always activated
    set_status(_malloc, -1, 0, NULL);
    pt = allocate(10);
    assert( pt == NULL );
    set_status(_malloc, 0, 0, NULL);
}
```

Figura 4. Teste de malloc() usando Clover

Clover é um elemento vital do módulo Test Suite proposto, uma vez que ele pode simular, de forma muito barata e automatizada, falhas como falta de memória, erro na leitura ou escrita de arquivo, ou qualquer outro tipo de erro de comportamento de qualquer função da biblioteca C, cobrindo todos os possíveis erros que uma chamada de função de biblioteca pode retornar, o que é altamente válido para sistemas críticos.

3.3 Plugin Test Suite

Após modelagem da MEF, ou de partes, os esqueletos de testes automatizados são gerados com o módulo Test Suite, o qual também é um plugin do Eclipse. Esse módulo pede três tipos de informação ao usuário, como indicado na Figura 5.

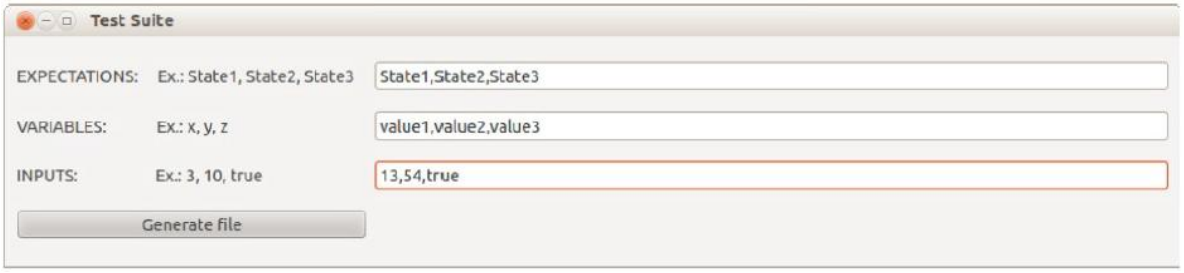

Figura 5. Test Suite GUI

O campo *Expectations* deve conter todos os estados de destino esperados de acordo com o modelo da MEF. *Variables* são responsáveis por definir a sequência de variáveis verificadas durante as transições dos eventos. Finalmente, o campo *Inputs* recebe a sequência de valores atribuídos às variáveis que permitirão o disparo das referentes transições de estado.

Atualmente, a biblioteca de testes utilizada é a Google C++ Testing Framework, a qual oferece uma fácil integração com Eclipse IDE. Google Test suporta a descoberta automática de testes, um rico acervo de assertivas, assertivas baseadas na definição do usuário, testes parametrizados por tipo e valor, entre outras características [Google 2015]. No entanto, o uso de *Test High C* (THC) é previsto por ser uma biblioteca de testes leve e de fácil extensão.

4. Conclusões

Esse artigo apresentou brevemente uma metodologia e um ferramental de suporte para o desenvolvimento de software de sistemas embarcados com o uso de TDD e MEF. A ferramenta de modelagem com MEF e de simulação Yakindu é usada para validação de requisitos, enquanto as ferramentas desenvolvidas e integradas pelos autores são usadas para atender a parte de verificação do processo.

Os recursos aqui apresentados estão sendo empregados no desenvolvimento do nanosatélite brasileiro 14-BISat. Especificamente no desenvolvimento do software embarcado, o qual é responsável por controlar sua carga útil. Esse satélite faz parte da missão internacional QB50 e realizará a função de servidor para uma rede de satélites.

A principal contribuição deste trabalho em relação ao que se encontra hoje na área é relacionada à integração da notação de uso comum (MEF) a uma técnica comprovadamente eficiente na melhoria de qualidade de software (TDD), provendo assim uma melhor integração das atividades de Validação e Verificação respectivamente.

Atualmente, uma série de melhorias já está sendo aplicada ao ferramental aqui apresentado: (i) uso de uma Domain Specific Language (DSL) para definir construções de testes específicas, como as relativas às plataformas de hardware; (ii) geração e execução dos testes em plataforma de microcontrolador; (iii) incorporação de Integração Contínua; (iv) incorporação de *Automated Deployment* em microcontrolador e (v) melhoria dos dublês de testes para microcontroladores. Todas estas melhorias estão sendo validades em plataformas MSP430 e Arduíno de maneira a serem definitivamente incorporadas e disponibilizadas.

Referências

Freeman, Steve et al. (2004) “Mock Roles, Not Objects”. In Proceeding of the Object-Oriented Programming, Systems, Languages, and Applications Conference (OOPSLA ’04), New York, USA, pages 236-246.

Google. Google C++ Testing Framework. Disponível em <http://code.google.com/p/googletest/>. Acesso em Julho, 2015.

Grenning, J. W. (2011), Test Driven Development for Embedded C, Pragmatic Bookshelf, 1st edition.

Itemis. Yakindu Statechart Tool. Disponível em < http://statecharts.org/documentation.html> Acesso em Julho, 2015.

Wright, D. R. (2005) “Finite State Machines”. Disponível em <http://www4.ncsu.edu/~drwrigh3/docs/courses/csc216/fsm-notes.pdf>. Acesso em Julho, 2015.