InRob-UML: uma Abordagem para Testes de Interoperabilidade e Robustez baseados em Modelos

Anderson C. Weller[1], Eliane Martins[1], Fátima Mattiello-Francisco[2]

[1]Instituto de Computação – Universidade Estadual de Campinas (Unicamp)
Av. Albert Einstein 1251 – 13081-970 – Campinas – SP – Brasil

[2]Instituto Nacional de Pesquisas Espaciais (INPE)
São José dos Campos – SP – Brasil

acweller@gmail.com, eliane@ic.unicamp.br, fatima.mattiello@inpe.br

Resumo. *Neste artigo apresentamos InRob-UML, um método para a geração automática de casos de testes de interoperabilidade e robustez a partir de modelos UML (Unified Modeling Language). O objetivo dos testes é determinar se duas implementações em teste são capazes de interoperar em presença de falhas temporais e de comunicação. O método proposto foi utilizado em um estudo de caso, um sistema de controle de passagem de nível em uma ferrovia, amplamente utilizado na literatura. Com o uso de um gerador de casos de teste baseado em meta-heurística, guiada por propósitos de teste, o InRob-UML evita problemas de explosão combinatória na geração de casos de teste, procurando a sequência de teste mais completa para um dado propósito de teste.*

Abstract. *This article presents InRob-UML, a method for automatic test case generation for interoperability and robustness testing from UML (Unified Modeling Language) models. The goal is to determine whether two implementations are able to interoperate in the presence of temporal and communication faults. The proposed method is applied in a case study, the Generalised Railroad Crossing problem, largely used in the literature. Using a metaheuristic based test case generator, guided by test purposes, the InRob-UML avoid the combinatorial explosion problems in test cases generation searching the most complete test sequence for given test purpose.*

1. Introdução

Contexto. Os testes de interoperabilidade visam determinar [Desmoulin and Viho 2008]: se duas ou mais implementações interagem corretamente e se estas implementações fornecem os serviços especificados durante a interação. É preciso verificar se elas interoperam na presença de falhas, portanto, utilizamos testes de robustez para determinar o grau em que esse sistema pode funcionar corretamente em presença de entradas inválidas ou sob condições ambientais estressantes [IEEE 1990]. A técnica de testes baseados em modelos é utilizada para a geração automática dos casos de teste.

Motivação prática: O tempo cada vez mais curto para colocar o *software* no mercado e a evolução dos requisitos do cliente ao longo do desenvolvimento são alguns dos fatores que têm levado a mudanças na forma como o *software* é elaborado. O reúso de componentes e o desenvolvimento incremental fazem com que os testes de interoperabilidade sejam cada vez mais necessários para determinar se os novos elementos são

capazes de interagir adequadamente com os demais módulos existentes. Mesmo que um componente funcione em um contexto, ele pode falhar em outro devido a problemas no ambiente ou em outros elementos, tornando necessária a realização de testes de robustez. Em trabalho prévio foi proposta a estratégia *InRob* (*INteroperability and ROBustness testing*) [Mattiello-Francisco et al. 2012], utilizada nos testes de interoperabilidade de aplicações espaciais. *InRob* utiliza TIOA (*Timed Input and Output Automata*) para a modelagem do comportamento das implementações em teste (IUT - *Implementation Under Test*), enquanto que nosso interesse atual é utilizar modelos de estado da UML, que são amplamente utilizados na academia e na indústria.

Trabalhos exitentes: Combinar testes de interoperabilidade e de robustez apresenta alguns desafios para a modelagem e para a geração automática de casos de teste. Na modelagem dos testes de interoperabilidade, o modelo deve representar a interação entre duas ou mais IUTs. [Desmoulin and Viho 2008]. Para os testes de robustez, devemos levar em conta as falhas na interação com o ambiente, quer dizer, precisamos de um *workload*, que são entradas que ativam as funcionalidades do sistema, e de um *faultload*, contendo as falhas do ambiente. Em geral, os dois são definidos de forma independente e as falhas são inseridas aleatoriamente, sem preocupação com o estado do sistema. Isso pode levar a situações onde diferentes falhas são injetadas num mesmo cenário de execução, ou nem sejam ativadas [Tsai et al. 1997, Cotroneo et al. 2013]. Já os testes de robustez baseados em modelos (MBRT - *Model-Based Robustness Testing*) [Fernandez et al. 2005, Ali et al. 2012] visam obter casos de teste a partir de um modelo do comportamento da IUT em presença de falhas. Uma forma de representar o comportamento robusto consiste em aumentar o modelo da IUT com as falhas a serem controladas durante os testes [Khorchef et al. 2010, Batth et al. 2007, Mattiello-Francisco et al. 2012]; porém, há a dificuldade em representar as falhas no modelo. Pode-se criar mutações do modelo original da IUT [Fernandez et al. 2005, Cotroneo et al. 2013]; mas, a dificuldade está em gerar casos de teste correspondendo a entradas inoportunas, i.e., entradas que existem no modelo, mas que não são esperados em um dado estado. Outra possibilidade consiste em aplicar mutações aos casos de teste gerados [Rollet 2003]; mas a dificuldade está em determinar a conformidade em relação ao comportamento modelado. Pode-se utilizar modelagem baseada em aspectos para representar o comportamento robusto do sistema [Ali et al. 2012]. Neste caso, o modelo original não precisa ser aumentado, e o comportamento robusto pode ser reutilizado. O modelo de robustez pode ser criado por um especialista, porém, é necessário ordenar os modelos aspectuais para alinhavá-los ao modelo original, para construir um modelo completo e correto a partir do qual os casos de teste sejam gerados. Além da dificuldade na ordenação, também há o risco de explosão de estados caso todos os aspectos sejam combinados de uma só vez.

Objetivo: Esse trabalho propõe uma estratégia para testar a interoperabilidade entre componentes de um sistema em presença de falhas, com o intuito de determinar a robustez na interação entre os componentes. Para atingir esse objetivo, será dada ênfase a problemas de interoperabilidade em presença de falhas de comunicação e de sincronização (restrições de tempo real) entre as IUTs. Outro objetivo é o uso de modelos de estados da UML para representar os componentes em teste. A partir desse modelo serão gerados o *workload* e *faultload* que serão utilizados durante os testes.

Solução: Na *InRob*, as falhas são atribuídas às IUTs interoperantes e introduzidas

no modelo de cada IUT. Apesar do canal de comunicação ser utilizado para mimicar os desvios de tempo na execução dos testes, assume-se que o ambiente não contenha falhas, e que estas são causadas por mau funcionamento de cada IUT. São consideradas apenas as falhas temporais, nas quais o tempo de ocorrência ou a duração da informação fornecida pelas interfaces desvia do que foi especificado [Avizienis et al. 2004]. Na *InRob-UML*, o injetor é modelado como parte do ambiente de comunicação entre os sistemas em teste, o que permite também representar um ambiente com falhas. O injetor é capaz de emular não só falhas temporais, mas também, falhas de conteúdo. Tanto o injetor quanto as IUTs são descritos por modelos independentes de plataforma. Além disso, é possível criar casos de teste para entradas inoportunas, sem que para isso seja necessário criar um modelo completo. Com essa estratégia é possível desenvolver o modelo funcional do injetor que reflita o comportamento de falha do ambiente de forma independente do modelo funcional das IUTs interoperantes, o que possibilita sua reutilização nos testes de outras aplicações, sem apresentar a dificuldade do uso de aspectos. E, como consideramos que as falhas são introduzidas por ambiente hostil, é possível ter um modelo de falhas mais completo.

Resultados: O método é aplicado a um estudo de caso, um sistema de controle de passagem de nível, utilizado na literatura para representar modelagem de sistemas de tempo real [Lavazza et al. 2001, Knapp et al. 2002, Hänsel et al. 2004]. O estudo de caso permitiu analisar se os casos de testes gerados correspondem às expectativas, e permitiu também determinar o que o gerador de casos de teste utilizado pode oferecer aos usuários.

Outline: Seção 2, O estudo de caso para ilustrar o método; Seção 3, O método *InRob-UML*; Seção 4, Geração de casos de teste; Seção 5, Conclusões e trabalhos futuros.

2. O estudo de caso

O GRC (*Generalized Railroad Crossing*) é um sistema que gerencia uma cancela de controle de acesso a um cruzamento de uma estrada de ferro. Os trens seguem em uma única direção, e devem passar por esse trecho com uma velocidade padrão. O controle é feito a partir de informações enviadas por dois sensores, instalados na estrada de ferro, que informam a chegada (*Arriving*) e a saída (*Leaving*) dos trens na região crítica.

Quando um trem passa pelo primeiro sensor, este envia uma mensagem para o sistema de controle (*Crossing*) informando o fato. Em resposta, o *Crossing* envia um comando para a cancela (*Gate*) solicitando o fechamento do cruzamento, e aguarda uma resposta em até $g + \Delta$ segundos, onde g é o tempo máximo para abertura ou fechamento da cancela e Δ é a tolerância máxima para possíveis *delays* na comunicação.

Quando o *Gate* recebe um comando para mudar seu estado atual, ele retorna uma mensagem de confirmação ao final do procedimento de abrir ou fechar (em g segundos). Caso ele já esteja no estado solicitado, a resposta é retornada sem aguardar tempo. Se o *Crossing* não receber uma confirmação de fechamento do *Gate* dentro do prazo máximo, então ele muda seu estado para *Failure* sinalizando que deve ser realizada uma intervenção externa ao sistema, dando tempo suficiente para o trem (t_p) parar antes do cruzamento (C), evitando potenciais perigos e possibilitando o reparo dos equipamentos e a reinicialização do sistema, conforme apresentado por [Hänsel et al. 2004].

Quando o trem sai da região crítica, detectado pelo segundo sensor, o *Crossing* envia comando de abrir para o *Gate*. A espera pela confirmação é temporizada, e após duas tentativas, o *Crossing* sinaliza a falha (*failure*) para a central de operações.

3. *InRob-UML*

Os passos do método *InRob-UML* são descritos brevemente a seguir:

1. Modelagem do serviço e do ambiente com falha: o primeiro é baseado na especificação, e o segundo no modelo de falhas do sistema.
2. Validação do modelo: o modelo gerado é transformado em um modelo executável, independente de plataforma, e sua execução permite validar o modelo e determinar se o sistema reage da maneira esperada aos parâmetros fornecidos.
3. Geração de CTA (Casos de Teste Abstratos): após ser considerado adequado, o modelo é utilizado na geração automática de CTA, descritos em formato independente da implementação. Usamos propósitos de teste (TP - *test purposes*), que descrevem propriedades ou cenários que se deseja testar [Grabowski et al. 1993], para evitar o risco de explosão combinatória. Os propósitos são especificados na forma de diagramas de sequência e a escolha é feita em conjunto com o cliente.
4. Concretização dos CTA: Visa transformar os CTA em *scripts* de teste em formato adequado para a realização na plataforma de execução. Também é feita a implementação do injetor de falhas, com base no modelo do ambiente com falha.
5. Execução dos testes e análise dos resultados: Os *scripts* de teste são aplicados à implementação do serviço em teste, e os resultados observados são comparados aos valores esperados (indicados pelo modelo).

3.1. Modelagem estática

O diagrama de classes na Figura 3.1 representa a estrutura estática, mostrando cada componente e cada tipo de dado utilizado na arquitetura de testes. O componente FEM (*Failure Emulator Mechanism*) emula tanto falhas de comunicação quanto falhas no *Gate*. Ele é parte do ambiente, e representa um canal de comunicação falho entre os subsistemas. As falhas a serem aplicadas (baseadas em [Han et al. 1995]) são: corrupção, duplicação ou perda de pacotes, bem como falhas temporais, atraso e avanço na entrega de pacotes.

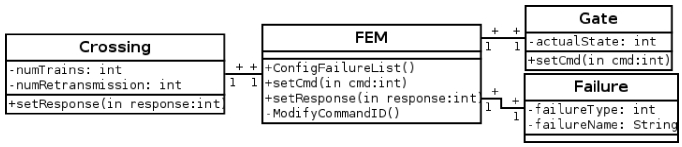


Figura 3.1. Diagrama de classes da arquitetura de testes para o estudo de caso.

3.2. Modelagem dinâmica

Cada componente representado no diagrama de classes tem um diagrama de estados associado, especificando o seu comportamento. Os atributos são variáveis utilizadas em ações e condições de guarda do modelo de estados. Já as operações são codificadas na linguagem Java e representam ações ou condições de guarda.

Uma transição inclui os seguintes elementos: i) nome do evento e seus eventuais parâmetros; ii) condição de guarda, uma expressão lógica que deve ser satisfeita para a transição ser disparada; iii) ações, realizadas quando é disparada uma transição. Além disso, ações podem ser disparadas na entrada ou na saída de um estado.

Para evitar problemas de explosão combinatória, representamos a comunicação (interoperabilidade) entre apenas duas implementações em teste (*one-to-one context*), que

no nosso caso é a interação entre o *Crossing* e o *Gate*. Para efeitos do teste de robustez, a comunicação entre esses dois componentes se dá através do FEM, como mostra a Figura 3.1, que representa a ação do injetor de falhas nessa interação.

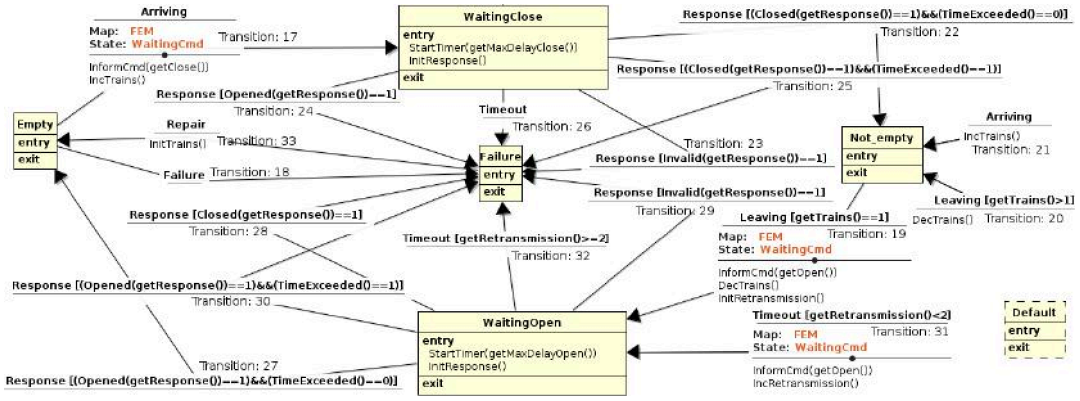


Figura 3.2. Diagrama de estados do componente *Crossing*.

O modelo é criado a partir das especificações do compilador de modelos de estado SMC (*State Machine Compiler*) [Rapp 2015], usando a ferramenta StateMutest (em desenvolvimento pelo grupo), onde cada modelo de estado é chamado de mapa. Apesar de não ser possível representar concorrência, pode-se realizar troca de mensagens entre os mapas através de transições *push*, que passam o controle da execução para outro mapa, e *pop*, que retornam a execução para o mapa chamador.

A Figura 3.2 mostra o diagrama de estados da classe *Crossing*, que inicia no estado *Empty*. Assim que ocorre um *Arriving* é disparado um *push* (transição 17), que envia o comando *Close* para *Gate* - via canal de comunicação (FEM) - e altera seu estado para *WaitingClose*. Assim que o mapa chamado terminar o processamento, ele retorna o fluxo de controle para *Crossing* com uma transição *pop* e dispara um evento neste (transição 8, Figura 3.4). Com isso, dependendo da mensagem recebida e do tempo decorrido, o estado é alterado para *Not_empty* ou *Failure*.

Na implementação há um objeto Relógio para auxiliar as restrições temporais. Ele é iniciado sempre que uma transição *push* é disparada no modelo inicial (ver *StartTimer*, Figura 3.2), e incrementado com o tempo previsto para cada mudança de estado (*IncreaseClock*, Figura 3.3); esses valores são definidos a partir da documentação do sistema. Com isso, é possível executar vários casos de teste sem ficar limitado a um relógio real.

A UML2 define *after(período)* para modelar a ocorrência de um evento após um certo período de tempo [OMG 2011]. Devido a limitações do gerador de casos de teste, os eventos precisam ter nomes distintos para podermos utilizar tempos diferentes em cada um deles. Portanto, definimos que o nome desses eventos devem iniciar com o prefixo "after_", seguido por um complemento para identificar sua função (ver transições 34 e 35, Figura 3.3).

O diagrama do *Gate*, Figura 3.3, especifica que a cancela sempre aguarda a chegada de uma solicitação (estado *WaintingCmd*), e retorna o controle da execução através de um *pop(Response)*, após enviar mensagem de confirmação através de um *InfResponse*. Dessa forma, é necessário armazenar a informação do estado final da última execução para que as condições de guarda selecionem as transições de estado corretas.

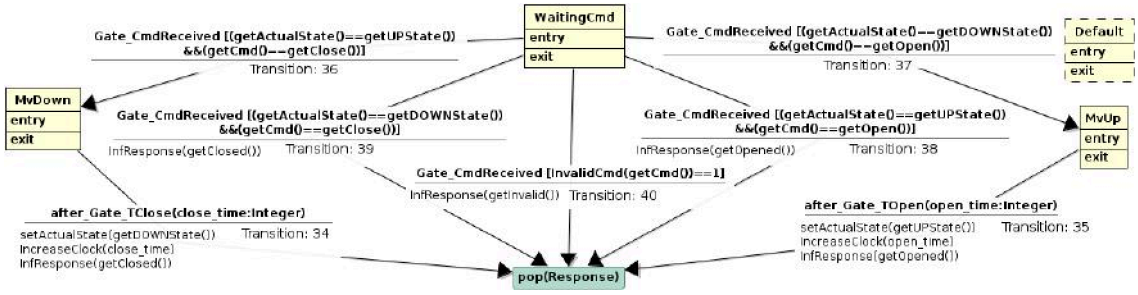


Figura 3.3. Diagrama de estados do *Gate*.

O diagrama do FEM, Figura 3.4, é sempre o ponto inicial da execução do modelo. A transição *push FEM_Setup* configura a sequência de falhas que será aplicada durante a execução do modelo (pode ser uma lista fixa ou escolhida pelo gerador), altera seu estado para *WaitingCmd* (para aguardar as mensagens enviadas pelo canal de comunicação) e direciona o controle da execução para o mapa cliente (*Crossing*).

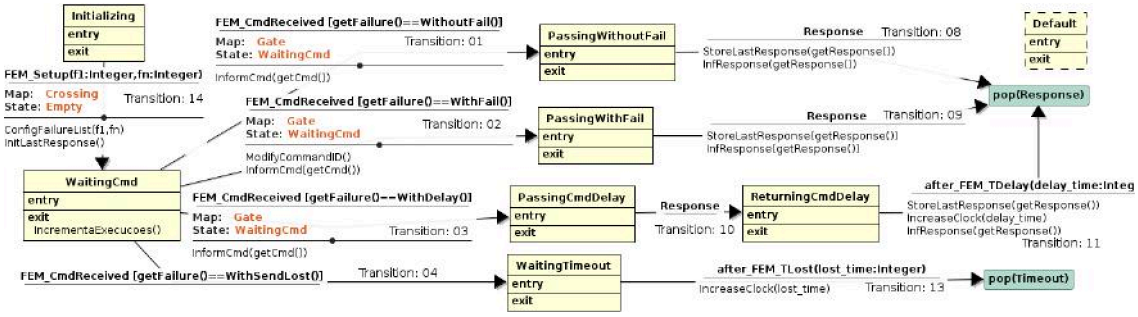


Figura 3.4. Diagrama de estados simplificado do FEM.

A Figura 3.4 apresenta apenas 4 das 7 possibilidades de falhas de comunicação modelada no FEM: na transição 1, a mensagem é repassada sem interferências; na 2 é alterado o conteúdo da mensagem enviada; na 3, a resposta de retorno é retida por tempo suficiente para a ocorrência de *Timeout*; e na 4, a mensagem enviada para o *Gate* é retida, ocasionando um *Timeout* no sistema solicitante.

3.3. Validação do Modelo

A StateMutest permite a animação do modelo. Sendo assim, para validar o modelo construído, criamos manualmente alguns casos de teste (contendo sequências de eventos) e os executamos no modelo dinâmico. A validação é feita através da conferência das saídas produzidas, bem como os estados visitados durante o processo.

4. Geração de casos de teste

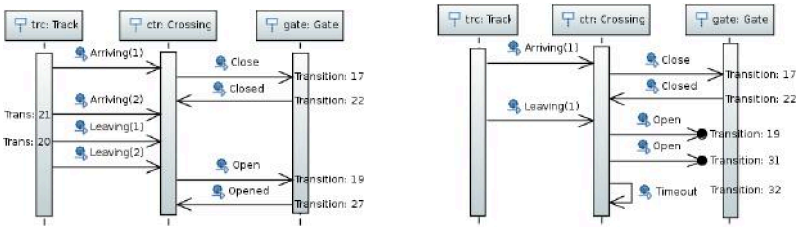
Em um trabalho prévio foi proposto o método MOST (*Multi-Objective Search-based Testing approach from EFSM*), para gerar casos de teste a partir de Máquina Finita de Estados Estendida (MEFE) [Yano et al. 2011]. A MOST usa algoritmo de otimização multiobjetivo para a busca por sequências de entradas que cubram um dado propósito de teste, portanto, não é baseada na enumeração exaustiva de caminhos, mas busca por alguns que atendam a dois objetivos: devem cobrir o propósito de teste e não devem ser muito longos.

Os caminhos são obtidos através da execução do modelo e selecionados através desses objetivos. Os dados são gerados durante a execução, para reduzir a geração de caminhos infactíveis, i.e, sintaticamente possíveis, mas semanticamente inviáveis, devido a conflitos nas condições de guarda. É possível obter vários caminhos que atendam a um propósito de teste, mas, ao contrário dos métodos exaustivos, que buscam todos os caminhos possíveis, MOST seleciona aqueles que melhor atendem aos objetivos da busca.

Para a execução são necessários: i) o modelo executável; ii) domínio dos parâmetros de entrada dos eventos; iii) propósito de teste (uma transição alvo e de um conjunto de cobertura de transições); iv) tamanho máximo da sequência de eventos nos casos de teste; v) número máximo de iterações durante a busca por uma solução; vi) parâmetro de ajuste do algoritmo multiobjetivo ($\tau \geq 0$), que define o grau de determinismo da busca, variando de uma caminhada aleatória ($\tau = 0$), a uma busca determinística local ($\tau \to \infty$).

4.1. Definição dos *test purposes*

Selecionamos dois TPs em nosso estudo de caso. O primeiro, Figura 4.1(a) representa um comportamento seguro do sistema, baseado em [Knapp et al. 2002], e especifica que, quando *Crossing* recebe o sinal de chegada do primeiro trem, *Gate* recebe comando para fechar (*Close*) a cancela. No entanto, o comando para abertura (*Open*) só é enviado após a confirmação de saída do último trem (*Leaving*). O segundo, Figura 4.1(b), representa um comportamento robusto em caso de falha de comportamento da cancela: se *Gate* não retornar confirmação a duas solicitações de *Crossing* para levantar a cancela (*Open*), então *Crossing* deve informar que há uma falha na passagem de nível e entrar em modo de falha.



(a) Cenário Normal (b) Cenário de Robustez

Figura 4.1. Diagramas de sequência representando dois possíveis TPs.

Dado que MOST aceita como propósitos de teste apenas identificadores de transições, os cenários devem ser mapeados através de uma transição alvo (t_{alvo}) e um conjunto de transições cobertura (t_{cob}). Como exemplo para os cenários especificados na Figura 4.1 temos: para (a) $t_{alvo} = 27$ e $t_{cob} = \{20\}$; e para (b) $t_{alvo} = 32$ e $t_{cob} = \{19, 31\}$.

4.2. Obtenção dos casos de teste

A MOST produz sequências de eventos contendo total ou parcialmente o TP (dependendo se cobrem todas ou apenas algumas transições do conjunto de cobertura). Caso não produza nenhuma sequência, isso pode ser uma indicação de que o TP é infactível.

MOST gera dois tipos de sequência: nominal (SN), que contém somente eventos especificados para os estados, e furtiva (SF), que contém eventos não especificados para os estados. A SF é gerada quando a máquina não é completa em relação às entradas (*input-complete*). A suposição de completude adotada pela UML é de que, ao receber um evento

não especificado para o estado, a máquina permanece inalterada [OMG 2011]. Dessa forma, é possível estender o modelo de falhas considerado para os testes de robustez, de modo a considerar também o comportamento em presença de entradas inoportunas.

Para obtermos um caso de teste é necessário determinar ainda as saídas esperadas e o veredito. Para obter as saídas esperadas, as sequências de entrada são aplicadas ao modelo executável, e as saídas produzidas são armazenadas. Se o caso de teste atende ou não ao propósito de teste, podemos definir os vereditos [Grabowski et al. 1993]:

- Passou: o caso de teste contém integralmente o TP, as saídas produzidas pelo IUT são iguais às saídas esperadas e as restrições temporais são satisfeitas;
- Inconclusivo: o caso de teste contém parcialmente o TP, as saídas produzidas pelo IUT são iguais às saídas esperadas e as restrições temporais são satisfeitas;
- Falhou: corresponde à situação em que as saídas produzidas pelo IUT não são iguais às saídas esperadas ou as restrições temporais não são satisfeitas.

Na Tabela 4.1 é apresentado o caminho de transições da SN (*path*) de dois casos de teste com veredito "Passou" - gerados a partir dos parâmetros informados na Seção 4.1 - seguida do seu tamanho (*pathSize*) e do número de iterações necessárias para obter o caso de teste (*numEval*). A lista contém o código das transições que compõem caso de teste (ver Figura 3.2, 3.3 e 3.4). Os números em negrito destacam t_{alvo} e t_{cob}.

Tabela 4.1. Caminho de transições para os TP's da Figura 4.1.

Cenário Normal	path 14 18 33 17 1 36 34 8 22 21 **20** 21 **20** 19 1 37 35 8 **27** 17 1 pathSize 21 numEval 1382688
Cenário Robustez	path 14 18 33 17 1 36 34 8 22 21 20 **19** 5 37 35 15 13 **31** 4 13 **31** 4 13 **32** pathSize 24 numEval 92140

Escolhemos os seguintes parâmetros para a busca dos casos de teste: Número máximo de avaliações = 5.000.000; Número de execuções independentes = 100; tamanho máximo da sequência de eventos = 500; τ = 3.75. Os demais parâmetros para configuração dos tempos foram: tempo de atraso = 61s, tempo de avanço = 1s e tempo de perda = 61s no FEM; tempo para fechar = 15s e tempo para abrir = 10s no *Gate*. No cenário normal utilizou-se apenas transmissão sem falha (tipo 1), e no cenário de robustez a escolha das falhas ficou a cargo da MOST (podendo ser da falha 1 ao 7).

4.3. Resultados

Realizamos buscas nos dois cenários escolhidos, e incluímos variações no número de transições de cobertura (T_{Cob}) para verificar se haveriam diferenças nos resultados obtidos. A Tabela 4.2 apresenta um sumário desses valores, e inclui as seguintes informações: Número de vezes que o mesmo cenário foi executado (Exc); Somatório total de casos de teste obtidos ao final das execuções (NCT); Média de casos de teste obtidos por execução do cenário (CTE); Total de casos de teste Válidos/Passou (CTV); Total de Casos de teste Inconclusivos (NCI); e a média de avaliações/iterações por caso de teste válido (MDA).

Utilizamos a ferramenta StateMutest em um Notebook HP Pavilion dv5 com Debian 7.7 (64bits), Intel core i3 M350 (2,27Mhz x 4) com 7,6GiB de memória RAM, e HD SATA de 465 GiB (com 30% de espaço livre), e o tempo médio de execução de cada um dos experimentos foi de 1h 35m 08s e a média geral para a obtenção de um caso de teste que atenda ao propósito de teste foi de 2.324.500 avaliações.

Tabela 4.2. Resultados dos Cenários Normal e de Robustez

		Cenário Normal					Cenário de Robustez				
Exc	T_{Cob}	NCT	CTE	CTV	NCI	MDA	NCT	CTE	CTV	NCI	MDA
8	1	12	1,50	8	4	1.836.867	5	0,63	5	0	1.692.522
8	2	16	2,00	8	8	2.702.582	2	0,25	2	0	1.405.316
8	3	14	1,75	8	6	1.908.298	2	0,25	2	0	2.053.126
8	Todos	16	2,00	8	8	3.008.021	2	0,25	2	0	4.463.931

Após os resultados, não percebemos diferença que justifique a utilização de um conjunto de cobertura maior na busca dos casos de testes nesses dois cenários analisados. Além disso, observamos que o cenário de robustez, por requerer caminhos mais complexos no modelo, obteve menos casos de teste do que o cenário normal.

5. Conclusões e Trabalhos Futuros

O presente trabalho demonstra a estratégia *InRob-UML*, baseada na *InRob*, que utiliza modelos UML para a geração de casos de teste de interoperabilidade e robustez entre dois subsistemas na presença de falhas. Outra diferença é a inclusão do modelo de um injetor de falhas (FEM) no modelo das IUTs interoperantes, elaborado para facilitar a sua reutilização em outras aplicações, com isso diminuindo o tempo de criação de novos projetos. A modelagem do injetor também oferece flexibilidade para modelar falhas temporais e de comunicação, além de permitir sincronizar o *faultload* e *workload* nos testes de robustez.

Os resultados obtidos indicam que o método é viável, mas foram necessárias algumas adaptações ao padrão da UML, devido a características das ferramentas utilizadas na geração dinâmica dos casos de teste.

Alguns parâmetros de entrada da ferramenta precisam ser avaliados de acordo com o modelo e o propósito de teste, portanto é necessário realizar experimentos que permitam calibrar os parâmetros de configuração para otimizar o processo de busca e aumentar a média de soluções por execução. Outro ponto que merece atenção futura é a inclusão de uma solução inicial aos parâmetros da ferramenta utilizada, possibilitando ao algoritmo melhorar uma solução previamente obtida. Também será feito um estudo comparando o uso das sequências nominais e uso das sequências inoportunas para revelar defeitos.

Referências

Ali, S., Briand, L., and Hemmati, H. (2012). Modeling robustness behavior using aspect-oriented modeling to support robustness testing of industrial systems. *Software & Systems Modeling*, 11(4):633–670.

Avizienis, A., Laprie, J.-C., Randell, B., and Landwehr, C. (2004). Basic concepts and taxonomy of dependable and secure computing. *Dependable and Secure Computing, IEEE Transactions on*, 1(1):11–33.

Batth, S. S., Vieira, E. R., Cavalli, A., and Uyar, M. Ü. (2007). Specification of timed efsm fault models in sdl. In *Formal Techniques for Networked and Distributed Systems–FORTE 2007*, pages 50–65. Springer.

Cotroneo, D., Di Leo, D., Fucci, F., and Natella, R. (2013). Sabrine: State-based robustness testing of operating systems. In *Automated Software Engineering (ASE), 2013 IEEEACM 28th International Conference on*, pages 125–135. IEEE.

Desmoulin, A. and Viho, C. (2008). Automatic interoperability test case generation based on formal definitions. In *Formal Methods for Industrial Critical Systems*, volume 4916 of *Lecture Notes in Computer Science*, pages 234–250. Springer Berlin Heidelberg.

Fernandez, J.-C., Mounier, L., and Pachon, C. (2005). A model-based approach for robustness testing. In *Testing of Communicating Systems*, volume 3502 of *Lecture Notes in Computer Science*, pages 333–348. Springer Berlin Heidelberg.

Grabowski, J., Hogrefe, D., and Nahm, R. (1993). Test case generation with test purpose specification by mscs. *SDL*, 93:253–266.

Han, S., Rosenberg, H. A., and Shin, K. G. (1995). Doctor: an integrated software fault injection environment for distributed real-time systems. In *Computer Performance and Dependability Symposium, 1995. Proceedings., International*, pages 204–213.

Hänsel, F., Poliak, J., Slovák, R., and Schnieder, E. (2004). Reference case study "traffic control systems". In *Integration of Software Specification Techniques for Applications in Engineering*, pages 96–118. Springer Berlin Heidelberg.

IEEE (1990). IEEE standard glossary of software engineering terminology. *IEEE Std 610.12-1990*, pages 1–84.

Khorchef, F. S., Berrada, I., Rollet, A., and Castanet, R. (2010). Automated robustness testing for reactive systems: application to communicating protocols. In *Gesellschaft für Informatik (GI)*, page 409. Citeseer.

Knapp, A., Merz, S., and Rauh, C. (2002). Model checking timed uml state machines and collaborations. In *Formal Techniques in Real-Time and Fault-Tolerant Systems*, volume 2469 of *Lecture Notes in Computer Science*, pages 395–414. Springer.

Lavazza, L., Quaroni, G., and Venturelli, M. (2001). Combining uml and formal notations for modelling real-time systems. In *Proceedings of the 8th European Software Engineering Conference*, ESEC/FSE-9, pages 196–206, New York, NY, USA. ACM.

Mattiello-Francisco, F., Martins, E., Cavalli, A. R., and Yano, E. T. (2012). Inrob: An approach for testing interoperability and robustness of real-time embedded software. *Journal of Systems and Software*, 85(1):3–15. Dynamic Analysis and Testing of Embedded Software.

OMG (2011). OMG Unified Modeling Language (OMG UML), Superstructure, V2.4.1. Technical report, Object Management Group.

Rapp, C. W. (2015). The state machine compiler. `http://smc.sourceforge.net`.

Rollet, A. (2003). Testing robustness of real-time embedded systems. In *Proceedings of Workshop On Testing Real-Time and Embedded Systems (WTRTES), Satellite Workshop of Formal Methods (FM) 2003 Symposium*. Citeseer.

Tsai, T. K., Zhao, H., Hsueh, M.-C., and Iyer, R. K. (1997). Path-based fault injection. In *Proc. 3rd ISSAT Conf. on R&Q in Design*, volume 51, pages 121–125.

Yano, T., Martins, E., and de Sousa, F. (2011). Most: A multi-objective search-based testing from efsm. In *Software Testing, Verification and Validation Workshops (ICSTW), 2011 IEEE Fourth International Conference on*, pages 164–173.