Uma avaliação do potencial de detecção de defeitos dos casos de teste de robustez de acordo com a análise de mutantes

Wallace Felipe Francisco Cardoso[1] **e Eliane Martins**[1]

[1]Instituto de Computação – Universidade Estadual de Campinas (UNICAMP)
Campinas – SP – Brazil

wallace.cardoso@ic.unicamp.br, eliane@ic.unicamp.br

Abstract. *Testing whether a system satisfies the specified functionalities is necessary, but not enough. It is also important to determine how a system behave in presence of invalid or unexpected inputs. This is the goal of robustness testing. If robustness faults go undetected during development, they can occur when the system is in operation, when faults are harder to diagnose and fix, which can lead to total interruption of system operations. In this paper we investigate the fault detection potential of nominal and robustness test sequences. Nominal sequences consider only specified inputs, whereas the robustness ones consider also innoportune inputs. We considered sequences generated from UML state diagrams which models the functional behavior of a system. Model-based mutation analysis was used to evaluate the fault detection potential of the generated sequences.*

Resumo. *Testar se o sistema satisfaz as funcionalidades especificadas é uma etapa necessária, mas não é suficiente. Importante é também determinar como o sistema se comporta em presença de entradas inválidas ou não esperadas (inoportunas). Este é o objetivo dos testes de robustez. Caso não sejam detectadas durante a fase de testes, as falhas de robustez (robustness faults) podem ocorrer durante a operação do sistema, quando são mais difíceis de diagnosticar e corrigir, e podem levar à interrupção das operações ou ao mau funcionamento do sistema. Neste trabalho analisamos o potencial de detecção de defeitos (faults) de sequências de teste nominais e sequências de teste de robustez. As sequências nominais contêm somente entradas especificadas, enquanto as sequências de robustez contêm também entradas inoportunas. As sequências são geradas a partir de modelos de estado da UML representando o comportamento funcional do sistema. Análise de mutantes foi utilizada para avaliar o potencial de detecção de falhas das sequências geradas.*

1. Introdução

Com o avanço tecnológico, os sistemas computacionais permeiam cada vez mais o nosso dia a dia. Sistemas embarcados em dispositivos diminutos são capazes de se comunicar através da internet com diversos outros sistemas. Dessa forma, nos tornamos cada vez mais dependentes destes sistemas tanto para tarefas corriqueiras, como por exemplo, trocar mensagens rápidas com outros, quanto para tarefas essenciais, como controle de tráfego aéreo, aviões e dispositivos médicos. Nestes casos, negócios ou mesmo vidas, dependem do bom funcionamento dos sistemas computacionais. Uma forma de garantir

que os sistemas se comportam conforme o que foi especificado é através da realização de testes funcionais. No entanto, determinar se o sistema realiza o que foi especificado não é mais suficiente; também interessa-nos saber se o sistema se comporta de maneira adequada em presença de falhas que podem advir das mais diversas fontes: operadores, outros sistemas, dispositivos de hardware com os quais o software interage, ou até mesmo, ataques. Este é o objetivo dos testes de robustez, que é determinar o quanto o sistema é capaz de operar adequadamente em presença de entradas inválidas ou inesperadas, ou ainda, em condições ambientais estressantes [IEEE Std 24765:2010]. Com "operar adequadamente" entende-se que o sistema deve ter mecanismos para tratar estas entradas anômalas e com isso evitar consequências catastróficas.

Os testes de robustez consistem em executar um sistema em presença de entradas anômalas que são introduzidas deliberadamente durantes os testes. Uma estratégia comumente usada nos testes de robustez consiste em introduzir parâmetros inválidos na interface do sistema com o seu ambiente [Micskei et al. 2012]. No entanto, esta estratégia não leva em conta o estado do sistema, e com isso, a controlabilidade dos testes é baixa. Nossa proposta é o uso de testes baseados em modelos tanto para gerar os testes funcionais quanto os testes de robustez.

Nos testes baseados em modelos (ou TBM), a ideia é que, a partir de um modelo formal ou semiformal do comportamento do sistema ou do seu ambiente, casos de teste completos (contendo entradas e saídas esperadas) sejam gerados [Prenninger and Pretschner 2005]. TBM é muito utilizada nos testes funcionais, e muitas propostas foram feitas, as quais utilizam o mesmo arcabouço para os testes de robustez.

O estudo apresentado neste trabalho visa responder à seguinte questão: *os casos de teste de robustez, para eventos inoportunos, detectam mais defeitos (faults) do que os casos de teste para situações normais?*

O restante deste trabalho é organizado como a seguir. A Seção 2 apresenta a fundamentação teórica. A Seção 3 apresenta alguns trabalhos relacionados, mostrando como o nosso trabalho se situa com relação ao estado da arte. A Seção 4 apresenta uma breve descrição da ferramenta StateMutest. A Seção 5 apresenta os estudos realizados visando responder à questão acima e finalmente as conclusões e perspectivas para trabalhos futuros estão contidos na Seção 6.

2. Fundamentação Téorica

Os conceitos básicos para o entendimento deste trabalho estão contidos nesta seção. Primeiro é descrita a abordagem de teste baseada em modelos, e depois uma técnica para avaliação de casos de teste, baseado no potencial de detecção de determinados defeitos típicos e específicos, conhecido como análise de mutantes.

2.1. Teste Baseado em Modelo

O objetivo da atividade de teste é descobrir a maior quantidade de defeitos presentes em um sistema. O domínio de entrada de um sistema mesmo com poucas linhas de código é enorme, o que torna a tarefa de testar sob todas as combinações possíveis de entrada algo impraticável. Por essa razão, deve-se escolher um subconjunto do domínio de entrada, a partir de informações contidas seja na própria implementação, seja

na especificação ou ainda, em um modelo do sistema. [Prenninger and Pretschner 2005, DeMillo and Offutt 1991].

Um modelo é a abstração do comportamento explícito de uma implementação chamada sistema sob teste. O modelo não deve ser tão detalhado quanto sua implementação e nem tão pobre de detalhes inviabilizando sua compreensão, mas suficiente para descrever como o sistema deverá se comportar. A etapa de criação de modelos do sistema, em uma metodologia de desenvolvimento bem planejada, antecede a etapa de criação do código do sistema. Os programadores frequentemente utilizam-se de modelos para desenvolver seu código. Assim, muitos dos erros que poderiam ocorrer futuramente podem ser evitados já durante a etapa de criação dos modelos através de uma análise, de uma simulação ou animação do modelo. Modelos também podem ser úteis na geração de casos de teste.

Existem várias propostas para a geração automática de casos de teste a partir de modelos formais (*i.e.* possuem semântica e sintaxe bem definida e formalizada). Destaca-se o uso de Máquina de Estados Finita (MEF) [Martins et al. 1999], Máquina de Estados Finita Estendida (MEFE) [Zhang et al. 2012], e Statecharts [Shirole et al. 2011, Santiago et al. 2006, Santiago et al. 2006].

Os casos de teste gerados tendo como base o modelo do sistema sob teste são abstratos (tanto quanto seu modelo). Especificamente para determinados tipos de modelos, existem alguns critérios que estabelecem requisitos mínimos de cobertura de determinados elementos, os quais podem ser combinados entre si, como por exemplo exercitar todas as transições (cobertura de transições) de uma máquina de estados como também passar por todos os estados (cobertura de estados), visando um conjunto de casos de teste potencialmente forte em detectar defeitos [Kim et al. 1999]. Os casos de teste abstratos podem ser reusados em diferentes implementações, pois independem de plataforma de execução.

Além de permitir a geração de casos de teste desde cedo no ciclo de desenvolvimento, o uso de TBM também pode fornecer uma solução para o problema do oráculo. O problema do oráculo diz respeito a como determinar a saída esperada para uma dada entrada. Tal tarefa é custosa e propensa a erros, quando realizada manualmente [Samuel et al. 2008]. O modelo, que representa o comportamento esperado do sistema, pode ser uma fonte para a produção de saídas esperadas.

Casos de teste gerados a partir de modelos são abstratos. O conjunto abstrato de casos de teste, para ser aplicado ao sistema sob teste, deve ser concretizado. É através da execução da suíte de teste concretizada que o testador conhecerá a quantidade de falhas ocorridas, e poderá tomar as medidas necessárias para a eliminação dos defeitos que as ocasionaram.

2.2. Análise de Mutantes

Análise de mutantes é uma técnica para avaliação de conjuntos de casos de teste, baseado em defeitos acidentais, introduzidos por desenvolvedores. A ideia é criar, a partir de um programa original, versões levemente alteradas sintaticamente, porém semanticamente diferentes e incorretas. Estas versões são conhecidas como *mutantes*, e são geradas através de *operadores de mutação*, os quais possuem regras do como e onde as mutações devem ser realizadas (ponto de mutação) [Jia and Harman 2011].

Pode-se observar no diagrama da Figura 1 que todos os mutantes (EQ, M1, M2,

e M3) são sintaticamente diferentes do original (OR), entretanto nem todos são semanticamente diferentes, como é o caso do mutante equivalente EQ. O mutante equivalente EQ, para qualquer entrada fornecida, sempre retornará a mesma saída do original OR. Os mutantes M1, M2 e M3 são de um único operador de mutação o qual troca um operador aritmético por outro. Neste exemplo, o mutante equivalente EQ é gerado através de um operador de mutação que troca a ordem dos parâmetros de uma função ou método.

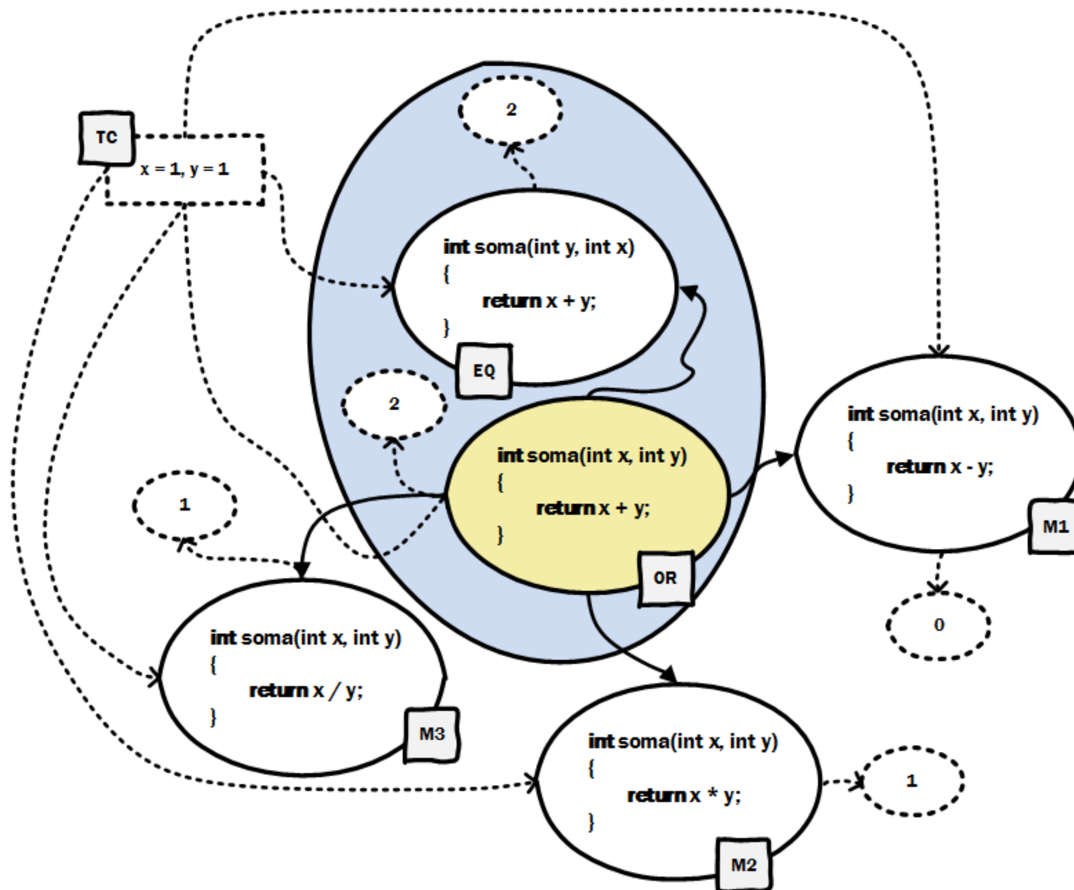

Figura 1. Programa Original (OR) e sua vizinhança (EQ, M1, M2, e M3).

Utilizando-se do conjunto de mutantes do diagrama da Figura 1, através do caso de teste TC ($x = 1, y = 1$), espera-se que a função *soma* retorne $x + y = 20$. Com TC foi possível identificar 3 possíveis versões defeituosas (M1, M2, e M3). Quando a saída do mutante difere da saída do original, o mutante é considerado *morto*. Quando a saída do mutante não difere da saída do original, para uma determinada entrada, o mutante é considerado como *vivo*. Um mutante vivo é dito *equivalente*, caso não existam entradas capazes de mostrar a diferença semântica entre ambos. Caso contrário, o conjunto de casos de teste deve ser melhorado, adicionando-se mais entradas, com o objetivo de que todos os mutantes não equivalentes sejam mortos [DeMillo and Offutt 1991]. O potencial de detecção de defeitos pode ser avaliado pelo escore de mutação, mostrado na Equação 1: onde TS é um conjunto de casos de teste, M é um conjunto de mutantes, DM é uma função que retorna a quantidade de mutantes mortos por TS, e EM é uma função que retorna a quantidade de mutantes equivalentes em M.

$$MS(TS, M) = \frac{DM(TS, M)}{|M| - EM(M)} \tag{1}$$

Claramente, o escore de mutação estabelece uma relação entre mutantes mortos e mutantes não equivalentes gerados. O ideal é ter um conjunto de teste que mate 100% dos mutantes, ou seja, com $MS = 1$.

Existem 3 condições que devem ser satisfeitas para um mutante ser morto: alcançabilidade, necessidade, e suficiência [Papadakis and Malevris 2010, Harman et al. 2011]. Alcançabilidade significa que um caso de teste deve passar pelo ponto de mutação. Necessidade significa que logo após passar pelo ponto de mutação, o mutante deve apresentar um estado interno diferente do original (infecção). Suficiência significa que a infecção interna deve se propagar até produzir uma saída observável.

Existem dois grandes problemas práticos a serem tratados ao utilizar da análise de mutantes. O primeiro é o problema da enorme quantidade de mutantes, pois existem inúmeras possibilidades de versões defeituosas mesmo para um trecho muito pequeno de código. O segundo é o problema de determinar a equivalência entre dois programas.

Duas das hipóteses propostas para a análise de mutantes restringem a geração de mutantes a apenas um subconjunto destes, conhecidas como a Hipótese do Programador Competente (HPC) e a Hipótese do Efeito Acoplamento (HEA). De acordo com a HPC, os programadores criam seus programas corretos ou muito próximo aos corretos. Como consequência da HEA, um conjunto de casos de teste que é capaz de matar mutantes formados por apenas uma única alteração sintática (mutante de primeira ordem), também é capaz de matar mutantes de várias alterações sintáticas (mutante de alta ordem) [Fabbri et al. 1994].

Mesmo utilizando-se apenas mutantes de primeira ordem, ainda assim o conjunto de mutantes é muito grande. Existem duas técnicas de redução de custos muito conhecidas que lidam com este problema: a mutação aleatória [Acree Jr 1980], e a mutação seletiva [Offutt et al. 1993]. A mutação aleatória é a seleção aleatória dos mutantes baseada em um percentual de seleção, por exemplo, selecionando-se apenas 10, 20, ou 30% dos mutantes. A mutação seletiva é a utilização criteriosa de apenas um subconjunto de operadores de mutação. Estas técnicas não são mutuamente exclusivas, e podem ser combinadas.

O problema de determinar a equivalência entre dois programas é indecidível. Existem apenas heurísticas que auxiliam o testador a determinar manualmente a equivalência entre um mutante e o original [Jia and Harman 2011].

3. Trabalhos Relacionados

Nesta seção serão apresentados alguns trabalhos que propõem ferramentas que auxiliam a atividade de teste, tanto para geração de casos de teste quanto para avaliação de casos de teste, relacionados à análise de mutantes e ao teste baseado em modelo, assim como trabalhos sobre teste de robustez e sua geração de casos de teste.

No contexto de teste de robustez e TBM, foram propostas estratégias e abordagens para geração de testes, as quais diferem na forma em que os eventos inválidos são introduzidos no modelo.

Uma estratégia consiste em completar o modelo com entradas inválidas (errôneas ou inoportunas) [Saad-Khorchef et al. 2007], bem como as saídas esperadas para estas entradas. A limitação desta estratégia é que o modelo pode se tornar muito grande para ser tratado por ferramentas de geração automática de casos de teste.

Para evitar o crescimento de modelos e com isso, a explosão combinatória do número de casos de teste, a estratégia CoFI (*Conformance and Fault Injection testing*) [Ambrosio et al. 2007] representa comportamento diante de entradas válidas e inválidas em diferentes modelos. A limitação dessa abordagem está no número de modelos, que pode crescer com a complexidade do sistema. Há ainda propostas de se modificar casos de teste, criando "mutantes" dos mesmos, para evitar a inclusão de entradas inválidas no modelo [Rollet and Salva 2009]. Neste caso, o problema é que alguns casos de teste podem se tornar infactíveis.

A proposta SABRINE (*StAte-Based Robustness testIng of operatiNg systEms*) [Cotroneo et al. 2013] extrai o modelo de estados a partir do *trace* de execução do sistema alvo, e gera os casos de teste de robustez a partir desse modelo. A limitação neste caso está no fato de que o modelo reflete o que foi implementado, e portanto, falhas de omissão de comportamento, por exemplo, não são reveladas.

Em trabalho prévio [Yano et al. 2010, Yano et al. 2011b] foi proposto um método de teste baseado em modelo de estados da UML para derivar casos de teste de forma dinâmica, *i.e.* utiliza-se um modelo executável e um algoritmo baseado em meta-heurística multiobjetivo (MOST, do inglês *Multi-Objective Search-based Testing*) o qual gera casos de teste visando cobrir um dado propósito de teste (*test purpose*), e que não sejam muito longos. Este método permite derivar casos de teste de robustez [Yano et al. 2011a] baseado na suposição de completude da UML, ou seja, por *default*, se uma entrada é recebida quando o sistema está em um estado em que essa entrada não é esperada, o sistema a ignora e permanece no mesmo estado [Rumbaugh et al. 2004].

No contexto de ferramentas de TBM, foram propostas ferramentas para a análise de mutantes e para o teste de mutação, *i.e.* avaliam ou geram conjuntos de casos de teste adequados à mutação.

A primeira ferramenta de teste para análise de mutantes baseada em modelo (*Model-Based Mutation Analysis*), de acordo com o nosso conhecimento, foi a Proteum/FSM [Fabbri et al. 1999a] . Inicialmente, os autores aplicaram a análise de mutantes manualmente em conjuntos de sequências de teste geradas a partir de MEF [Fabbri et al. 1994], propuseram novos operadores de mutação, e argumentaram como mandatório a implementação de uma ferramenta.

Em continuidade, foi proposta uma ferramenta de teste baseada em Statecharts e na análise de mutantes, chamada de Proteum/ST [Sugeta 1999] . Ambas as ferramentas Proteum/FSM e Proteum/ST implementam as técnicas de redução de custos mutação aleatória e mutação seletiva.

Momut::UML foi a primeira ferramenta em teste baseado em mutação de modelo (*Model-Based Mutation Testing*) [Aichernig et al. 2015]. Os autores propuseram uma ferramenta de geração de casos de teste, a partir dos diagramas da UML, e no contexto dos testes de mutação, no qual os diagramas utilizados são o de estados, o de instanciação, e o de classes. A saber, o conjunto de teste gerado através da Momut::UML tem o propósito

de matar mutantes, enquanto que outras ferramentas baseadas em análise de mutantes somente são capazes de avaliar os conjuntos de casos de teste de acordo com sua capacidade de matar mutantes.

Neste trabalho utilizamos a ferramenta StateMutest [Cardoso 2015], desenvolvida pelo grupo. A StateMutest oferece algumas melhorias ou diferenças significativas em relação a abordagens anteriores. O testador, através da StateMutest, não precisa completar o modelo com entradas adicionais, e também não precisa lidar com vários modelos para aspectos de robustez. Com o objetivo de reduzir a geração de casos de teste infactíveis, a ferramenta usa um método dinâmico, em que caminhos no modelo são selecionados de acordo com a execução do modelo. Outra característica da geração de casos de teste na StateMutest é a possibilidade de se obter tanto sequências nominais quanto inoportunas.

4. Sobre a ferramenta StateMutest

A StateMutest é uma ferramenta de apoio ao teste baseado em modelo [Cardoso 2015]. A StateMutest não somente oferece a geração de casos de teste a partir de modelos, mas também a avaliação do potencial de detecção de defeitos de um conjunto de teste usando a análise de mutantes. Atualmente, a StateMutest aceita apenas um subconjunto dos elementos do modelo de estados da UML. Fazem parte da ferramenta: editor gráfico e textual de MEFE, animação do modelo, assistente de importação de casos de teste (casos de teste gerados por outras ferramentas), análise de mutantes, geração de casos de teste, e possibilidade de extensões.

A geração de casos de teste de uma MEFE considera tanto o fluxo de controle quanto o de dados, o que significa considerar parâmetros, variáveis, predicados, e ações. Para tratar de um problema complexo como é o da geração de casos de teste de uma MEFE, a StateMutest utiliza-se de uma meta-heurística. A meta-heurística é multiobjetivo (MOST) [Yano et al. 2011b], a qual busca pela melhor solução baseando-se em dois objetivos. O primeiro objetivo é o de encontrar uma sequência de eventos e parâmetros que exercitem um determinado caminho do modelo, que cubra o critério dado. Por ora, somente o critério baseado em propósito de testes (*test purposes*) está disponível. O testador deve fornecer uma transição alvo (transição final a ser atingida), bem como um conjunto de cobertura contendo as demais transições a serem visitadas pelo propósito de teste. A saída são sequências de entradas que atinjam a transição alvo e que cubram uma ou mais transições do conjunto de cobertura. O segundo objetivo é o de encontrar uma solução adequada e que seja a menor possível. A razão para ter a redução do tamanho do caso de teste é por que casos de teste mais longos levam mais tempo para serem executados, dado que a ferramenta gera casos de teste de tamanhos variados, o usuário pode escolher se prefere casos de teste mais longos ou mais curtos.

A StateMutest oferece 11 operadores de mutação propostos para Statecharts [Fabbri et al. 1999b], os quais estão descritos na Tabela 1. Foram implementadas também as técnicas de redução de custos: mutação aleatória e mutação seletiva (citado na Seção 2.2). Várias tarefas simultâneas (paralelismo) são criadas durante a execução dos mutantes, já que a execução de um mutante não interfere na de outro, reduzindo assim o tempo total de execução.

Usuários experientes podem preferir criar *scripts* para automatizar os passos necessários para o uso da ferramenta. São comandos específicos da ferramenta que exe-

Tabela 1. Operadores de mutação utilizados pela StateMutest

Identificação	**Nome**	**Descrição**
TraIniStaAlt	*Transition – Initial State Alteration*	Altera o estado inicial
TraArcDel	*Transition – Arc Deletion*	Remoção de uma transição
TraEveDel	*Transition – Event Deletion*	Remoção de um evento de uma transição
TraDesStaAlt	*Transition – Destination State Alteration*	Altera o estado de destino de uma transição
OutDel	*Output Deletion*	Remove todas as ações de uma transição
StaDel	*State Deletion*	Remove um estado
CondTraConsAltCons	*Condition/Transition – Constant Alteration by Constant*	Substitui uma constante por outra constante, apenas em condições de guarda
CondTraLogOperLog	*Condition/Transition – Logical Operator by Logical Operator*	Substitui um operador lógico por outro operador lógico, apenas em condições de guarda
TraParamAltParam	*Transition – Parameter Alteration by Parameter*	Substitui um parâmetro em um transição por outro parâmetro da mesma transição
CondTraRelOperRel	*Condition/Transition – Relational Operator by Relational Operator*	Substitui um operador relacional por outro operador relacional, apenas em condições de guarda
CondTraVarAltCons	*Condition/Transition – Variable Alteration By Constant*	Substitui uma variável por uma constante, apenas em condições de guarda

cutam a geração, execução e a análise de mutantes, assim como existem os comandos específicos para geração automática de casos de teste. A criação de *scripts* é muito útil principalmente na realização de experimentos, onde várias execuções, sob as mesmas condições, são necessárias de serem realizadas sequencialmente. Usuários com conhecimento avançado de programação podem inclusive criar extensões para a ferramenta.

5. Realização dos Experimentos

Os experimentos desta seção buscam responder à questão levantada anteriormente: *os casos de teste de robustez, para eventos inoportunos, detectam mais defeitos do que os casos de teste para situações normais?* Para responder a esta pergunta, utilizamos a análise de mutantes. As sequências de entrada nominais e inoportunas são aplicadas a mutantes do modelo original e o escore de mutação é obtido. Todo o experimento foi realizado em um computador (notebook) ASUS, modelo X55C, 12 GB de memória SO-DIMM DDR3 1333Mhz, disco rígido de 7200RPM 2.5" SATA, e processador Intel Core i3-3110M 2.40Ghz. O sistema operacional utilizado foi o sistema Windows 7 Professio-

nal, e a versão da StateMutest utilizada foi a versão 3.1.30. Utilizamos o *teste-t pareado* [Walpole et al. 2011, p. 246] para as análises estatísticas, dado que o tamanho das amostras são iguais e podemos assumir que possuem a mesma variância.

Primeiramente, 11 modelos foram selecionados, os quais são considerados como *benchmark* por estarem presentes em diversas pesquisas sobre teste baseado em modelo [Kalaji et al. 2009, Yano et al. 2010]. Todos os 11 modelos (EFSM), bem como suas características, estão na Tabela 2.

Tabela 2. Os 11 modelos (EFSM) utilizados, e seus respectivos elementos

Nome do Modelo	Estados	Transições	Eventos	Guardas	Ações
ATM	10	24	13	11	51
Cashier	13	22	16	10	50
CruiseControl	6	18	10	11	110
FuelPump	14	26	17	7	112
Inres	9	19	10	5	49
Lift	7	13	12	3	12
VendingMachine	8	29	12	15	70
InFlight	5	33	15	31	313
Class2Protocol	7	22	14	11	106
ATM2	11	31	14	22	44
Inres2	6	17	8	8	78

O modelo *ATM* representa o comportamento (simplificado) de um sistema de caixa eletrônico. *Cashier* representa o comportamento de um sistema de caixa de uma loja. *CruiseControl* representa o comportamento do controlador automático de velocidade de um veículo. O modelo *FuelPump* representa o comportamento de um sistema de posto de combustível. O modelo *Inres* representa o comportamento de um protocolo de redes (simplificado). *Lift* representa o comportamento de um controlador de um elevador. *VendingMachine* representa o comportamento de um sistema de uma máquina automática de venda de produtos. *InFlight* representa o comportamento de um controlador de voo. O modelo *Class2Protocol* representa o comportamento do protocolo da camada de transporte, classe 2. Por fim, os modelos *ATM2* e *Inres2* são melhorias dos modelos *ATM* e *Inres2*, respectivamente.

Para os experimentos, foram consideradas as sequências de teste usadas em trabalhos prévios do grupo [Yano et al. 2010, Yano et al. 2011b], no qual este trabalho está inserido também. Para a obtenção dos casos de teste abstratos, aplicam-se as entradas geradas ao modelo original para obter as saídas esperadas. O conjunto de testes final foi a união dos conjuntos de testes gerados para diferentes propósitos de teste, de forma que, ao final, temos um conjunto que cobre todas as transições dos modelos utilizados. A Tabela 3 apresenta um resumo dos conjuntos de casos de teste, um para cada modelo. O menor caso de teste, de todos os modelos, tem tamanho de sequência de entrada igual a 1, estes são os casos de teste que exercitam uma das transições de saída do estado inicial. Existem também casos de teste de 4 a 5 vezes maiores do que a quantidade de transições de seus respectivos modelos, e são os que mais têm eventos inoportunos.

Após a geração dos casos de teste, utilizando a StateMutest, foi realizada a análise

Tabela 3. Resumo da quantidade de casos de teste utilizada

Nome do Modelo	Casos de Teste	Maior	Menor
ATM	70	44	1
Cashier	153	100	1
CruiseControl	94	19	1
FuelPump	71	61	1
Inres	261	103	1
Lift	52	37	1
VendingMachine	454	490	1
InFlight	326	494	1
Class2Protocol	322	482	1
ATM2	202	255	1
Inres2	188	105	1

de mutantes em dois subconjuntos de casos de teste de cada modelo. O primeiro subconjunto contém todos os casos de teste (SEI), o que significa que neste há tanto entradas nominais quanto inoportunas. O segundo é semelhante ao primeiro, exceto pela remoção de todos os eventos inoportunos (SEN). É claro que o conjunto SEI demora mais do que o conjunto SEN para executar, entretanto, a questão é qual deles é mais efetivo em detectar defeitos.

Na Tabela 4 é apresentado o resultado da análise de mutantes para o conjunto SEI e o conjunto SEN, onde NR é o conjunto de casos de teste que não alcançam o ponto de mutação do modelo (não satisfazem a propriedade de alcançabilidade), e EQ é o conjunto de mutantes equivalentes identificados manualmente. Em média, de acordo a Tabela 4, o conjunto SEI tem um potencial de detecção de defeitos maior do que o conjunto SEN ($t_{value}(11) = 6.0430 > t_{0.01} = 3.106$).

Tabela 4. Escore de mutação para o conjunto SEI e o conjunto SEN

	Escore de Mutação			
Nome do Modelo	**SEN (NR)**	**SEI (NR)**	**Casos de Teste**	**Mutantes (EQ)**
ATM	0,9626 (13)	1 (0)	70	348 (2)
Cashier	0,7197 (105)	0,8071 (74)	153	389 (3)
CruiseControl	0,8269 (9)	0,8615 (0)	94	260 (5)
FuelPump	0,9030 (17)	0,9229 (0)	71	588 (5)
Inres	0,9314 (12)	0,9838 (0)	261	248 (13)
Lift	0,9338 (10)	0,9779 (0)	52	136 (1)
VendingMachine	0,9191 (26)	0,9570 (0)	454	396 (5)
InFlight	0,7449 (136)	0,8708 (59)	326	938 (45)
Class2Protocol	0,7379 (70)	0,7964 (48)	322	393 (27)
ATM2	0,8954 (14)	0,9291 (0)	202	593 (7)
Inres2	0,8468 (9)	0,8899 (0)	188	209 (10)
Média	**0,8556**	**0,9100**		
Variância	**0,0075**	**0,0048**		

A ocorrência de eventos inoportunos não acarreta em mudança de estados. No entanto, observa-se que as sequências inoportunas alcançam melhor os pontos de mutação do que as sequências nominais.

Foi realizada a medição do tempo de execução de cada modelo considerado durante a análise de mutantes, visando descobrir se a diferença de custo entre ambos os conjuntos é significativa. O tempo de execução é apresentado na Tabela 5, onde se pode ver que o tempo de execução do conjunto SEI e do conjunto SEN, em média, não é significativamente diferente ($t_{value}(11) = 1.6831 < t_{0.05} = 2.201$).

Tabela 5. Tempo de execução dos conjuntos SEI e SEN na análise de mutantes

Nome do Modelo	Tempo	
	SEN	**SEI**
ATM	00:04:34	00:04:31
Cashier	00:07:43	00:08:26
CruiseControl	00:03:26	00:04:48
FuelPump	00:08:33	00:10:01
Inres	00:07:57	00:07:52
Lift	00:01:51	00:01:37
VendingMachine	00:28:05	00:27:40
InFlight	00:53:50	01:03:26
Class2Protocol	00:15:21	00:17:57
ATM2	00:18:35	00:19:37
Inres2	00:05:38	00:05:35
Média	**00:14:08**	**00:15:35**

Nosso experimento demonstra que o conjunto de casos de teste com eventos inoportunos têm um potencial de detecção de defeitos maior do que o conjunto de casos de teste que não tem eventos inoportunos, sendo o custo-benefício do primeiro melhor do que o do segundo. Embora o conjunto SEN, na maioria dos testes, execute um pouco mais rápido do que o conjunto SEI, a diferença de tempo de execução de ambos não é significativa.

6. Conclusões

Os testes de robustez, os quais testam o sistema sob condições e entradas anômalas, são uma das formas de garantir que um determinado sistema é capaz de se comportar adequadamente tanto para as condições previstas quanto para as anomalias que podem ocorrer.

Neste trabalho, através de um experimento, investigamos o potencial de detecção de defeitos dos conjuntos de casos de teste com eventos inoportunos (condições anômalas) em relação aos conjuntos de casos de teste que não os consideram (condições normais). Primeiro, geramos as sequências de entrada (nominais e inoportunas), em seguida, aplicamos as sequências geradas ao modelo, para obter as saídas esperadas para os modelos originais. Essas saídas são comparadas com as saídas produzidas quando os casos de testes gerados são aplicados aos modelos mutantes. Comparou-se então as saídas originais àquelas produzidas pelos mutantes. Depois, a análise de mutantes foi realizada visando avaliar o potencial de detecção de defeitos da suíte de teste, e responder à questão: *os*

casos de teste de robustez, para eventos inoportunos, detectam mais defeitos do que os casos de teste para situações normais?

Os resultados mostram que a capacidade de revelar defeitos dos conjuntos de eventos inoportunos foi em média superior aos testes utilizando-se apenas os eventos nominais, apresentando também um melhor custo-benefício, *i.e.* mais defeitos encontrados gastando-se pouco tempo (efetividade). Com isso, os testadores podem acrescentar sequências anômalas ou inoportunas em seus conjuntos de casos de teste, ao invés de retirá-las visando um possível ganho em desempenho, pois elas melhoram a qualidade do conjunto de testes em termos de potencial de detecção de defeitos.

Nosso experimento foi realizado com uma quantidade pequena de modelos, o que pode ser uma possível ameaça à validade. Os modelos utilizados, embora sejam parte de um *benchmark* da comunidade de teste, não são modelos reais, de sistemas reais. Os tempos mensurados também podem não ser exatos, com prejuízos, devido a possibilidade de execução de programas do sistema operacional. Portanto, não é possível generalizar os resultados desta pesquisa, contudo, conseguimos resultados motivadores em relação aos testes de robustez mesmo para um conjunto com poucos modelos disponíveis.

Futuramente, pretendemos estender a avaliação da análise de mutantes, de acordo com entradas anômalas e normais, para um conjunto que tenha uma quantidade representativa de modelos, com a presença de modelos de sistemas reais. Também pretendemos verificar se estes resultados são semelhantes utilizando-se outros critérios de teste.

7. Agradecimentos

Agradecimentos ao CNPq pelo suporte financeiro durante a realização desta pesquisa, e à Professora Doutora Sandra Camargo Pinto Ferraz Fabbri, professora do Departamento de Computação da UFSCar, por nos ajudar através de discussões sobre análise de mutantes aplicada a modelos.

Referências

Acree Jr, A. T. (1980). *On Mutation.* PhD thesis, Georgia Institute of Technology, Atlanta School of Information and Computer Science.

Aichernig, B., Brandl, H., Jobstl, E., Krenn, W., Schlick, R., and Tiran, S. (2015). Momut:: Uml model-based mutation testing for uml. In *Software Testing, Verification and Validation (ICST), 2015 IEEE 8th International Conference on*, pages 1–8. IEEE.

Ambrosio, A. M., Mattiello-Francisco, F., Santiago Jr, V. A., Silva, W. P., and Martins, E. (2007). Designing fault injection experiments using state-based model to test a space software. In *Dependable computing*, pages 170–178. Springer.

Cardoso, W. F. F. (2015). Statemutest: uma ferramenta de apoio ao teste baseado em modelos de estado estendidos. Master's thesis, Universidade Estadual de Campinas.

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 IEEE/ACM 28th International Conference on*, pages 125–135. IEEE.

DeMillo, R. and Offutt, A. J. (1991). Constraint-based automatic test data generation. *Software Engineering, IEEE Transactions on*, 17(9):900–910.

Fabbri, S. C., Delamaro, M. E., Maldonado, J. C., and Masiero, P. C. (1994). Mutation analysis testing for finite state machines. In *Software Reliability Engineering, 1994. Proceedings., 5th International Symposium on*, pages 220–229. IEEE.

Fabbri, S. C., Maldonado, J. C., and Delamaro, M. E. (1999a). Proteum/fsm: a tool to support finite state machine validation based on mutation testing. In *Computer Science Society, 1999. Proceedings. SCCC'99. XIX International Conference of the Chilean*, pages 96–104. IEEE.

Fabbri, S. C. P. F., Maldonado, J. C., Sugeta, T., and Masiero, P. C. (1999b). Mutation testing applied to validate specifications based on statecharts. In *Software Reliability Engineering, 1999. Proceedings. 10th International Symposium on*, pages 210–219. IEEE.

Harman, M., Jia, Y., and Langdon, W. B. (2011). Strong higher order mutation-based test data generation. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, pages 212–222. ACM.

Jia, Y. and Harman, M. (2011). An analysis and survey of the development of mutation testing. *Software Engineering, IEEE Transactions on*, 37(5):649–678.

Kalaji, A., Hierons, R. M., and Swift, S. (2009). Generating feasible transition paths for testing from an extended finite state machine (efsm). In *Software Testing Verification and Validation, 2009. ICST'09. International Conference on*, pages 230–239. IEEE.

Kim, Y. G., Hong, H. S., Bae, D.-H., and Cha, S. D. (1999). Test cases generation from uml state diagrams. In *Software, IEE Proceedings-*, volume 146, pages 187–192. IET.

Martins, E., Sabião, S. B., and Ambrosio, A. M. (1999). Condata: a tool for automating specification-based test case generation for communication systems. *Software Quality Journal*, 8(4):303–320.

Micskei, Z., Madeira, H., Avritzer, A., Majzik, I., Vieira, M., and Antunes, N. (2012). Robustness testing techniques and tools. In *Resilience Assessment and Evaluation of Computing Systems*, pages 323–339. Springer.

Offutt, A. J., Rothermel, G., and Zapf, C. (1993). An experimental evaluation of selective mutation. In *Proceedings of the 15th international conference on Software Engineering*, pages 100–107. IEEE Computer Society Press.

Papadakis, M. and Malevris, N. (2010). Automatic mutation test case generation via dynamic symbolic execution. In *Software reliability engineering (ISSRE), 2010 IEEE 21st international symposium on*, pages 121–130. IEEE.

Prenninger, W. and Pretschner, A. (2005). Abstractions for model-based testing. *Electronic Notes in Theoretical Computer Science*, 116:59–71.

Rollet, A. and Salva, S. (2009). Testing robustness of communicating systems using ioco-based approach. In *Computers and Communications, 2009. ISCC 2009. IEEE Symposium on*, pages 67–72. IEEE.

Rumbaugh, J., Jacobson, I., and Booch, G. (2004). *Unified Modeling Language Reference Manual, The*. Pearson Higher Education.

Saad-Khorchef, F., Rollet, A., and Castanet, R. (2007). A framework and a tool for robustness testing of communicating software. In *Proceedings of the 2007 ACM symposium on Applied computing*, pages 1461–1466. ACM.

Samuel, P., Mall, R., and Bothra, A. K. (2008). Automatic test case generation using unified modeling language (uml) state diagrams. *Software, IET*, 2(2):79–93.

Santiago, V., Amaral, A. S. M. d., Vijaykumar, N. L., Mattiello-Francisco, M. F., Martins, E., and Lopes, O. C. (2006). A practical approach for automated test case generation using statecharts. In *Computer Software and Applications Conference, 2006. COMPSAC'06. 30th Annual International*, volume 2, pages 183–188. IEEE.

Shirole, M., Suthar, A., and Kumar, R. (2011). Generation of improved test cases from uml state diagram using genetic algorithm. In *Proceedings of the 4th India Software Engineering Conference*, pages 125–134. ACM.

Sugeta, T. (1999). *PROTEUM-RS/ST: uma ferramenta para apoiar a validação de especificações statecharts baseada na análise de mutantes.* PhD thesis, Universidade de São Paulo.

Walpole, R. E., Myers, R. H., Myers, S. L., and Ye, K. (2011). *Probability and statistics for engineers and scientists*, volume 9. Prentice Hall.

Yano, T., Martins, E., and De Sousa, F. L. (2010). Generating feasible test paths from an executable model using a multi-objective approach. In *Software Testing, Verification, and Validation Workshops (ICSTW), 2010 Third International Conference on*, pages 236–239. IEEE.

Yano, T., Martins, E., and De Sousa, F. L. (2011a). A model-based approach for robustness test generation. In *Dependable Computing Workshops (LADCW), 2011 Fifth Latin-American Symposium on*, pages 33–34. IEEE.

Yano, T., Martins, E., and De Sousa, F. L. (2011b). 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. IEEE.

Zhang, J., Yang, R., Chen, Z., Zhao, Z., and Xu, B. (2012). Automated efsm-based test case generation with scatter search. In *Proceedings of the 7th International Workshop on Automation of Software Test*, pages 76–82. IEEE Press.