Um Método para Geração de Casos de Teste a partir da Validação de Modelos UML/OCL utilizando Satisfatibilidade

Marcos V. F. A. Dias[1], **Eber Assis Schmitz**[1], **Priscila M. V. Lima**[1]

[1]Programa de Pós-graduação em Informática - PPGI
Universidade Federal do Rio de Janeiro (UFRJ)

mscfurriel@gmail.com, eber@nce.ufrj.br, priscilamvl@gmail.com

Abstract. *This paper proposes the generation of generic test cases from UML models annotated with OCL constraints. The proposal applies a Boolean satisfiability-based UML/OCL model validation technique. The application of the process of validation of the UML/OCL model is performed through the USE tool and its extension called ModelValidator. Two quasi-experiments were carried out in order to test the viability of the approach. Initial results indicate the usefulness of test case generation approach to structure and range of values of a system under test, considering class models of little complexity.*

Resumo. *Este artigo propõe a geração de casos de teste genéricos a partir de modelos UML anotados com restrições OCL. A proposta aplica a técnica de validação de modelos UML/OCL baseada em satisfatibilidade booleana. A aplicação do processo de validação do modelo UML/OCL é realizada por intermédio da ferramenta USE e de sua extensão denominada ModelValidator. Dois quasi-experimentos foram realizados a fim de testar a viabilidade da abordagem. Os resultados iniciais demonstram que há indícios de que a abordagem de geração de casos de teste para estrutura e faixa de valores de um sistema sob teste seja útil considerando modelos de classes pouco complexos ou parciais.*

1. Introdução

À medida que a tecnologia da informação evolui, novos desafios são descobertos durante a produção de um sistema de informação. Tais desafios acabam por aumentar a complexidade dos produtos de software desenvolvidos. A necessidade de criar sistemas que se inter-relacionam com tecnologias de naturezas diversas, sejam elas novas ou legadas, aumenta o risco de que o produto não apresente os níveis de qualidade esperados pelos usuários. É nesse sentido que o teste de software tem papel fundamental na produção de um sistema de informação, pois é o processo de teste que irá garantir que o produto desenvolvido atenda às expectativas dos usuários e, consequentemente, apresente um nível de qualidade satisfatório [Brambilla et al. 2012].

Entretanto, os desafios relacionados à produção do software também atingem o processo de teste, visto que, quanto mais complexo for o sistema, mais complicado será testá-lo [Silva-De-Souza et al. 2014]. Devido a isso, novas abordagens de teste ganharam destaque nas últimas décadas, como é o caso dos Testes Baseados em Modelos (TBM). Na abordagem TBM os testes são gerados a partir de modelos abstratos que representam a estrutura e o comportamento do sistema sob teste (SST) [Utting and Legeard 2007].

Em uma abordagem TBM, os modelos abstratos são caracterizados por serem descritos por uma linguagem independente de plataforma, isto é, livre das dificuldades inerentes às tecnologias de uma plataforma específica. No contexto de TBM, a Linguagem de Modelagem Unificada (do inglês, UML - *Unified Modeling Language*) em conjunto com a Linguagem para Especificação de Restrições em Objetos (do inglês, OCL - *Object Constraint Language*) apresentam relevância, uma vez que os modelos UML fornecem, em conjunto, não só a estrutura como também o comportamento de um sistema de informação [Utting and Legeard 2007].

Os modelos UML podem ser verificados e validados, fornecendo uma garantia de que eles estejam de acordo com os requisitos do negócio. Dessa forma, um modelo UML consistente, que represente de maneira abstrata um SST, permite que diversos testes possam ser gerados. A consistência de um modelo UML pode ser obtida por intermédio de testes sobre os modelos utilizando satisfatibilidade (SAT). Nesse caso, um modelo UML é transformado em um modelo de lógica proposicional e aplicado em um Solucionador SAT (do inglês, SAT *Solver*) [Kuhlmann and Gogolla 2012].

A geração de testes utilizando SAT apresentou alguns trabalhos recentes como [Przigoda et al. 2018] e [Desai and Gogolla 2018]. O trabalho de [Przigoda et al. 2018] emprega a utilização de Teorias do Módulo de Satisfatibilidade (do inglês, SMT - *Satisfiability Modulo Theory*) para a validação de modelos UML/OCL. Em [Desai and Gogolla 2018], os testes são gerados a partir de um esquema de teste previamente informado pelo usuário. Contudo, ambas as abordagens são empregadas com o intuíto apenas de validar o modelo UML/OCL.

A aplicação das técnicas de validação de modelos UML/OCL para geração de testes para SST representa uma oportunidade de pesquisa. O trabalho apresentado em [Dias et al. 2017] demonstra uma abordagem *lightweight* baseada na validação de modelos UML para geração de casos de teste abstratos utilizando diagramas de classes anotados com restrições OCL. O método descrito em [Dias et al. 2017] necessita que os projetistas de requisitos tivessem conhecimento avançado na linguagem ASSL.

2. Arcabouço Conceitual

2.1. USE: UML-based Specification Environment

A ferramenta USE é utilizada para modelagem, verificação e validação de diagramas de classe da UML e restrições OCL. A ferramenta permite avaliar a consistência de um diagrama de classes em relação as suas regras de negócio. A verificação e validação do modelo UML é realizada de três maneiras: manual, semi-automática utilizando uma linguagem denominada "A Snapshot Sequence Language"(ASSL) e semi-automática por intermédio da extensão ModelValidator, que utiliza SAT.

De acordo com [Soeken et al. 2010], a abordagem utilizando SAT é mais eficiente que a abordagem utilizando ASSL. A extensão ModelValidator transforma o modelo UML da ferramenta USE em um modelo relacional. Esse modelo relacional é aplicado a uma API denominada Kodkod, que será responsável por traduzi-la para a Forma Normal Conjuntiva (FNC) para então ser solucionada por um SAT *Solver*.

2.2. Satisfatibilidade (SAT)

O problema SAT é um problema NP-completo. Um problema SAT possui soluções, ou é considerado satisfazível, se houver pelo menos uma atribuição de valores-verdade às suas variáveis que tornem a sua fórmula verdadeira, no caso proposicional [Przigoda et al. 2018]. No caso de primeira ordem requer também a atribuição de constantes às variáveis presentes na fórmula. Para evitar a explosão combinatória na busca desses valores, algumas estratégias são empregadas, a fim de reduzir o espaço de busca, às custas de uma possível solução incompleta.

Os SAT *Solvers* permitem que um grande espaço de busca seja coberto de forma eficiente [Przigoda et al. 2018, Soeken et al. 2010]. A aplicação de SAT no contexto da verificação e validação de modelos UML está diretamente ligado ao paradigma conhecido como Bounded Model Checking (BMC). Através do BMC é possível testar as propriedades do modelo, permitindo inclusive a geração de contra-exemplos.

3. Método para geração dos casos de teste

O método proposto é fortemente baseado na abordagem presente em [Dias et al. 2017]. O método consiste em gerar um conjunto de casos de teste de acordo com a combinação das invariantes que serão testadas. O Algoritmo 1 resume a proposta.

Algoritmo 1: MÉTODO PARA GERAÇÃO DE CASOS DE TESTE

Entrada: $dc, cInv$
Saída: Casos de Teste

```
1  início
2      use ← Carregar(dc, cInv)
3      pModel ← Transformar(use.model)
4      cCombInvs ← Combinar(cInv)
5      para cada item ∈ cCombInvs faça
6          cConfig ← GerarPropriedades(pModel)
7      fim
8      para cada config ∈ cConfig faça
9          solucao ← ModelValidator.scrollingAll(config)
10         resultado ← config.invariantes
11         ct ← Armazenar(solucao, resultado)
12     fim
13 fim
14 retorna ct
```

Um diagrama de classes (dc) e um conjunto de invariantes OCL (cInv) são as entradas iniciais do processo. Esses artefatos são carregados na ferramenta USE. Em seguida, o modelo USE é transformado em um modelo simplificado de Propriedades (pModel). A partir de pModel e do conjunto de invariantes combinadas (cCombInvs) é possível gerar um conjunto de arquivos de configuração (cConfig). Cada configuração é validada pela extensão ModelValidator sequencialmente através do comando *scrollingAll*. As soluções encontradas são armazenadas juntamente com o resultado esperado para formar um caso de teste (ct). Para este trabalho, um caso de teste é uma dupla contendo (instância, resultado esperado). A figura 1 demonstra a estrutura simplificada de

configurações das propriedades.

```
[config0]
    classeA_min = 1
    classeA_max = 2

    relacionamentoAB_min = 0
    relacionamentoAB_max = 4

    classeB_min = 1
    classeB_max = 2

    aggregationcyclefreeness = off
    forbiddensharing = off
```

```
[config2]
    classeA_min = 1
    classeA_max = 2

    relacionamentoAB_min = 0
    relacionamentoAB_max = 4

    classeB_min = 1
    classeB_max = 2

    invarianteB = negate

    aggregationcyclefreeness = off
    forbiddensharing = off
```

```
[config1]
    classeA_min = 1
    classeA_max = 2

    relacionamentoAB_min = 0
    relacionamentoAB_max = 4

    classeB_min = 1
    classeB_max = 2

    invarianteA = negate

    aggregationcyclefreeness = off
    forbiddensharing = off
```

```
[config3]
    classeA_min = 1
    classeA_max = 2

    relacionamentoAB_min = 0
    relacionamentoAB_max = 4

    classeB_min = 1
    classeB_max = 2

    invarianteA = negate
    invarianteB = negate

    aggregationcyclefreeness = off
    forbiddensharing = off
```

Figura 1. Exemplo simplificado de configurações

Cada configuração será aplicada à extensão ModelValidator, que irá buscar por todas as instâncias que satisfaçam as condições.

4. Quasi-Experimentos

Os quasi-experimentos se diferenciam de experimentos pelos fatos de não haver grupo de controle e de suas amostras terem sido obtidas por conveniência [Wohlin et al. 2012]. Dessa maneira, foram realizados dois quasi-experimentos com a intenção de avaliar a viabilidade da abordagem proposta. O primeiro quasi-experimento gerou testes com foco na estrutura de um sistema, apresentando casos de teste para verificação de integridade referencial. Já o segundo quasi-experimento gera testes a partir de um modelo que contém restrições sobre valores de atributos.

Os dois quasi-experimentos foram realizados em um computador com processador i5 4440 de 3,3Ghz, memória RAM de 16GB, SSD, Sistema Operacional Windows 10, USE versão 5.0.2, Java JDK 8.

4.1. Integridade Referencial

No contexto deste trabalho, os testes de integridade referencial dizem respeito às restrições impostas como multiplicidades entre as classes. Por conveniência, foi criado

um modelo de classes genérico denominado ABCDEF com restrições de um para um sobre todas as classes. A figura 2 apresenta o diagrama de classes criado.

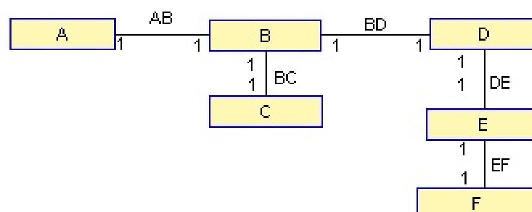

Figura 2. Diagrama de classes genérico

O diagrama de classes ABCDEF foi divido em 5 diagramas parciais: AB, ABC, ABCD, ABCDE e ABCDEF. A abordagem proposta neste trabalho foi aplicada a cada um dos diagramas parciais. Cada diagrama de classes apresenta um arquivo contendo as invariantes OCL relacionadas às multiplicidades do diagrama de classes que serão carregadas dinamicamente na ferramenta USE. Como cada relacionamento é restrito e bidirecional, cada relacionamento se transforma em duas invariantes OCL.

Tabela 1. Resultado do quasi-experimento com até dois objetos de cada tipo

Classes	Total de invariantes	Iterações	Tempo para tradução (ms)	Tempo para solucionar (ms)	Tempo total (ms)
AB	2	13	99	11	110
ABC	4	91	337	34	371
ABCD	6	613	5893	358	6251
ABCDE	8	3855	185035	3811	188864
ABCDEF	10	24501	6557879	44796	6602675

Tabela 2. Resultado do quasi-experimento com até três objetos de cada tipo

Classes	Total de invariantes	Iterações	Tempo para tradução (ms)	Tempo para solucionar (ms)	Tempo total (ms)
AB	2	42	1167	48	1215
ABC	4	939	4081952	5885	4087537
ABCD	6	*	*	*	>10h

Cada iteração representa a busca por uma solução e consequentemente um caso de teste. A iteração gera uma fórmula relacional diferente. A coluna *Tempo para tradução* representa o somatório do tempo gasto para traduzir a fórmula relacional em FNC para todas as iterações. Já a coluna *Tempo para solucionar* diz respeito ao tempo total para solucionar todas as fórmulas FNC geradas nas iterações.

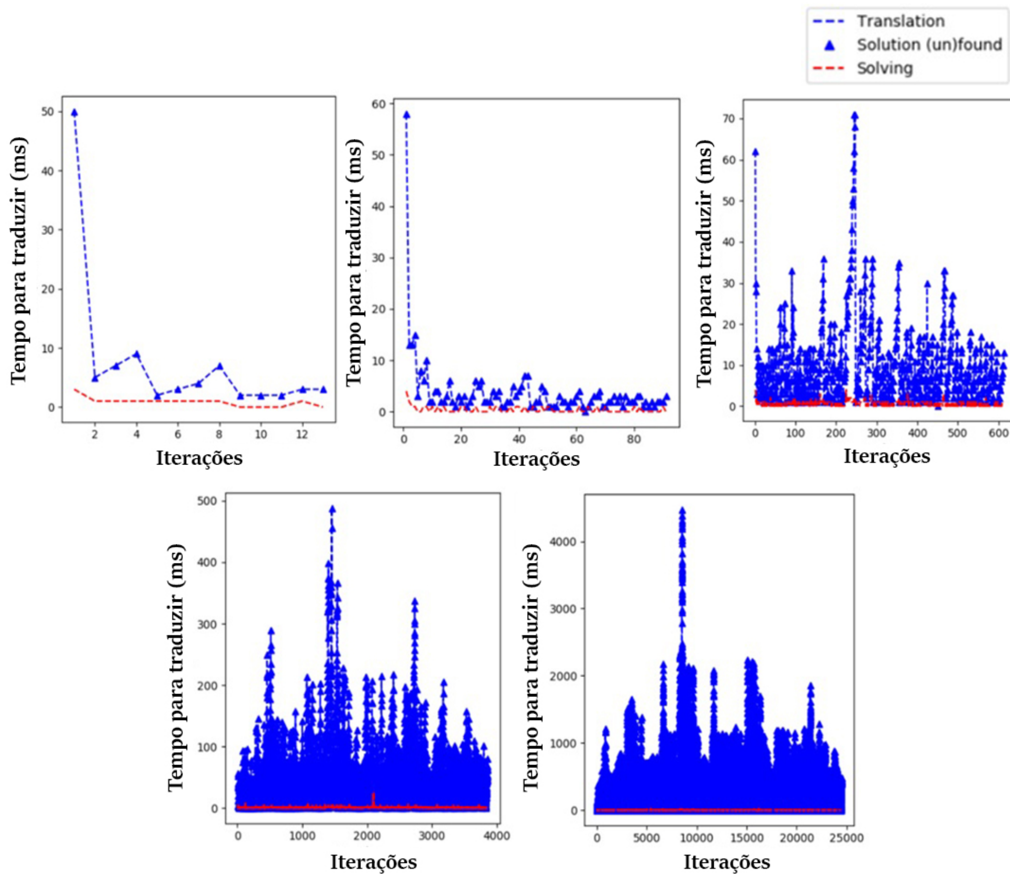

Figura 3. O tempo em relação às iterações para até 2 objetos de cada tipo de classe (da esquerda para a direita: AB, ABC, ABCD, ABCDE e ABCDEF)

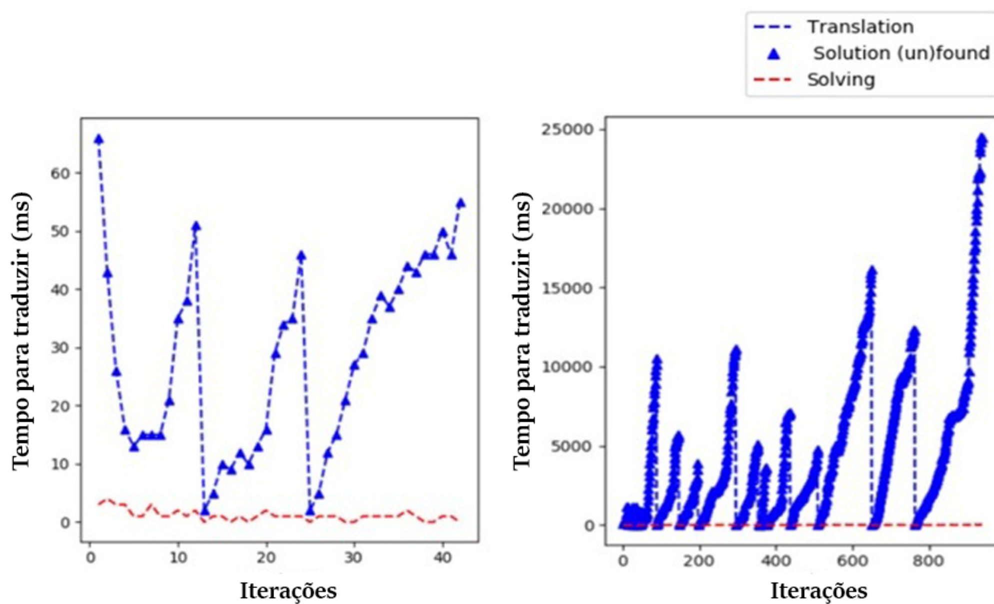

Figura 4. O tempo em relação às iterações para até 3 objetos de cada tipo de classe (da esquerda para a direita: AB e ABC)

4.2. Faixa de Valores

No contexto deste trabalho, os testes relacionados às faixas de valores representam os casos de teste que envolvem restrições sobre os valores dos atributos das classes. Para este quasi-experimento foi utilizado o modelo Flight adaptado de [Warmer and Kleppe 2003].

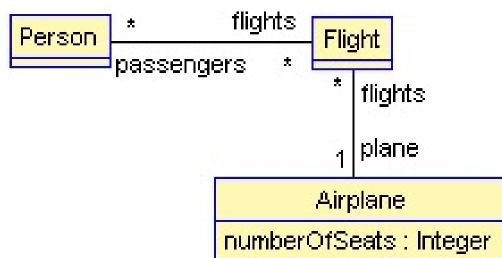

Figura 5. Modelo Flight

Foram escolhidas duas restrições sobre o modelo *Flight*. A primeira restrição diz que um determinado voo só pode estar relacionado a um avião específico. Já a segunda restrição informa que o número de passageiros em um voo deve ser igual ou menor que o número de assentos do avião. A figura 6 apresenta as restrições do modelo *Flight* especificadas como invariantes OCL. A tabela 3 apresenta o resultado após a execução do método proposto.

```
context f:Flight
    inv OneFlightOneAirplane:
        f.plane -> size() = 1

    inv numberOfSeats:
        Airplane.allInstances ->
            forAll(a | a.numberOfSeats
                >= a.flights.passengers -> size)
```

Figura 6. Restrições especificadas em OCL

Tabela 3. Resultado da execução do método sobre o modelo UML Flight

Total de invariantes	Iterações	Tempo para tradução (ms)	Tempo para solucionar (ms)	Tempo total (ms)
2	84	1631	100	1731

Os resultados obtidos a partir dos quasi-experimentos exibem indícios da utilidade da abordagem para geração de casos de teste, considerando diagramas de classes não complexos ou apenas partes deles. Por outro lado, a abordagem mostra-se inviável para modelos complexos contendo muitas classes, relacionamentos ou atributos. O tempo gasto para traduzir o modelo Relacional para FNC representa o grande problema da abordagem. O tempo para fazer essa tradução tende a aumentar consideravelmente à medida que as fórmulas relacionais ficam mais complexas.

5. Considerações Finais

A principal contribuição deste trabalho é apresentar um método para geração de casos de teste para um sistema em produção utilizando uma técnica de validação de modelos UML que utiliza SAT. Trabalhos futuros podem buscar soluções para o problema do tempo elevado para tradução das fórmulas relacionais em FNC. Uma possibilidade está ligada ao processamento paralelo de cada configuração gerada para a extensão Model-Validator. Outra possibilidade está relacionada ao teste aleatório, no qual, um conjunto de configurações é gerado aleatoriamente seguindo alguma estratégia específica. Os casos de teste gerados pelo método apresentado são considerados abstratos, podendo haver futuramente geração para casos concretos contendo procedimentos de teste.

Referências

Brambilla, M., Cabot, J., and Wimmer, M. (2012). Model-driven software engineering in practice. *Synthesis Lectures on Software Engineering*, 1(1):1–182.

Desai, N. and Gogolla, M. (2018). Generating OCL Constraints from Test Case Schemas for Testing Model Behavior. In Medina-Bulo, I., Merayo, M. G., and Hierons, R., editors, *Proc. 30th Int. IFIP WG 6.1 Conf. Testing Software and Systems (ICTSS'2018)*. Springer, LNCS 11146.

Dias, M. V. F., Schmitz, E. A., Silva, M. F., and Lima, P. M. V. (2017). Geração de casos de teste independentes de plataforma utilizando diagramas de classes da uml anotados com restrições ocl. In *Anais da IV Escola Regional de Sistemas de Informação do Rio de Janeiro*, Porto Alegre, Brasil. Sociedade Brasileira de Computação - SBC.

Kuhlmann, M. and Gogolla, M. (2012). From uml and ocl to relational logic and back. In *Proceedings of the 15th International Conference on Model Driven Engineering Languages and Systems*, MODELS'12, pages 415–431, Berlin, Heidelberg. Springer-Verlag.

Przigoda, N., Wille, R., Przigoda, J., and Drechsler, R. (2018). *Automated Validation & Verification of UML/OCL Models Using Satisfiability Solvers*. Springer Publishing Company, Incorporated, 1st edition.

Silva-De-Souza, T., Ribeiro, V. V., and Travassos, G. H. (2014). Estimativa de esforço em teste de software: modelos: fatores e incertezas. *XX Congreso Argentino de Ciencias de la Computación*.

Soeken, M., Wille, R., Kuhlmann, M., Gogolla, M., and Drechsler, R. (2010). Verifying uml/ocl models using boolean satisfiability. In *Proceedings of the Conference on Design, Automation and Test in Europe*, DATE '10, pages 1341–1344, 3001 Leuven, Belgium, Belgium. European Design and Automation Association.

Utting, M. and Legeard, B. (2007). *Practical Model-Based Testing: A Tools Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

Warmer, J. and Kleppe, A. (2003). *Object Constraint Language, The: Getting Your Models Ready for MDA*. Addison Wesley.

Wohlin, C., Runeson, P., Hst, M., Ohlsson, M. C., Regnell, B., and Wessln, A. (2012). *Experimentation in Software Engineering*. Springer Publishing Company, Incorporated.