Geração automática de testes em um processo de *Model Driven Development*: um exemplo de utilização da ferramenta Sikuli

Romulo de Almeida Neves [1]
Willian Massami Watanabe [1]

Resumo: Contexto: *Model Driven Development* (MDD) eleva a importância de modelos dentro do ciclo de vida do software, incorporando-os como parte integrante do produto final por meio de técnicas de modelagem e geração de código, com isso, parte da complexidade do software fica escondida dentro de geradores. Problema: Mesmo com a utilização do processo MDD, o custo associado à elaboração de casos de teste ainda é alto. Objetivo: Automatizar a geração de casos de teste incorporando o *Model Driven Testing* (MDT) em um processo MDD já existente a fim de gerar automaticamente casos de teste de aceitação para um sistema legado. Justificativa: Garantir a qualidade do software gerado através de um processo de geração de código e reduzir o tempo geral do ciclo de vida do software. Método: Foi realizado uma prova de conceito como forma de validar o objetivo deste trabalho e avaliar a efetividade da abordagem de teste utilizando métricas de *Code Coverage* dos casos de testes gerados automaticamente. Resultados: Para este estudo foram selecionados cinco *Graphical User Interface* (GUIs), na qual a GUI Seguradoras obteve 98% de cobertura de código pelos casos de teste, Moeda obteve 94,39%, Fornecedor com 95,04%, Tipo do item com 95,95% e por fim a GUI Classificação Contábil obteve 95,91% de cobertura dos casos de teste.

***Palavras-chave*:** *Code Coverage*, Prova de Conceito , *Model Driven Development*, *Model Driven Testing*

Abstract: *Context: Model Driven Development (MDD) elevates the importance of models within the software lifecycle, incorporating them as an integral part of the final product through modeling and code generation techniques, with part of the complexity of the software is hidden inside generators. Problem: Even with the use of the MDD process, the cost associated with case test is still high. Objective: Automate the generation of test cases by incorporating Model Driven Testing (MDT) into an existing MDD process in order to automatically generate acceptance test cases for a legacy system. Justification: Ensure the quality of software generated through a code generation process and reduce overall lifecycle time the software. Method: A proof of concept was carried out to validate the objective of this work and evaluate the effectiveness of the test approach using the Code Coverage metric of the automatically generated test cases. Results: For this study five Graphical User Interface (GUIs) were selected, in which the GUI Insurance Company obtained 98% of code coverage by the test cases, Coin obtained 94.39%, Provider with 95.04%, Item type With 95.95% and finally the GUI Accounting Classification obtained 95.91% coverage of the test cases.*

***Keywords*:** *Proof of concept, Code Coverage, Model Driven Development, Model Driven Testing*

[1]Programa de Pós-Graduação em Informática (PPGI), UTFPR, Campus Cornélio Procópio (PR) - Brasil
{romulo.neves@gmail.com, wwatanabe@utfpr.edu.br}

http://dx.doi.org/10.5335/rbca.v9i4.6735

1 INTRODUÇÃO

O desenvolvimento de software baseado em modelos, *Model Driven Development* (MDD), consiste em reconhecer a importância dos modelos no processo de software e a sua principal proposta é fazer com que o engenheiro ou arquiteto de software não precise interagir com o código fonte diretamente, preocupando-se principalmente com modelos de alto nível [1]. Esse processo traz vantagens como aumento de produtividade, confiabilidade, manutenibilidade, interoperabilidade das aplicações desenvolvidas, melhorias na manutenção, reutilização e documentação do software [2] [3] [4].

No processo MDD, os diagramas conceituais não são apenas utilizados como referência para a codificação, mas também são utilizados como artefatos ativos no processo e servem como entrada para ferramentas de geração de código, reduzindo o esforço dos desenvolvedores [3]. As transformações entre modelos geram sequências de transformações que permitem a implementação completa de um sistema partindo dos seus requisitos [5].

No entanto, apesar das vantagens desse processo de desenvolvimento, um processo MDD, não se isenta de falhas. Dependendo de projeto para projeto, falhas humanas podem ocorrer no processo de desenvolvimento. Com isso a existência de um processo de testes de softwares bem estruturado e organizado torna-se requisito básico para se atingir uma melhor qualidade do software produzido e, consequentemente, sendo uma das alternativas para obter um produto com um número reduzido de defeitos. Sendo assim, o *Model Driven Testing* (MDT) surge como uma forma de abordagem para automatização do Teste de Software (TS), a fim de reduzir os esforços na criação dos casos de teste.

Baker *et al.* [6] definem o MDT como um processo para a geração de artefatos de teste em diferentes níveis de abstração aplicando regras de transformação e contribuindo para aumentar a confiabilidade e produtividade em processos de teste, sendo composto por: casos de teste, dados de teste e oráculos. Lima *et al.* [7] demonstram dois benefícios ao adotar a abordagem MDT: a redução do custo (após implantada) e a automação no processo de geração de casos de teste. Além disso, os modelos de teste podem auxiliar na identificação de erros antes da transformação dos modelos em código [8]. Para isso, a fim de obter benefícios na geração e execução de testes, é necessário que a abordagem MDT refine-se a três tarefas: (i) a geração de casos de teste obtidos através de modelos de acordo com o critério de cobertura; (ii) a geração de oráculos de testes que determinam os resultados esperados de um teste; e (iii) a execução de testes em ambientes de teste obtidos através de modelos. Pode-se observar que a primeira e segunda tarefa são independentes de plataforma, no entanto a terceira deve ser executada em uma plataforma [9].

Neste contexto a empresa Exactus Software desenvolveu um processo MDD para que sejam gerados códigos para o desenvolvimento de seus sistemas legados ou seja, sistemas que foram desenvolvidos utilizando tecnologias ultrapassadas [10]. No entanto, os testes sobre os produtos desenvolvidos pelo processo MDD na Exactus Software são realizados de forma manual sobre os produtos gerados sem nenhum processo de automatização.

Com base no que foi exposto, o objetivo é automatizar a geração de casos de teste incorporando o MDT em um processo MDD, utilizando a ferramenta Sikuli através dos scripts gerados automaticamente pelo processo MDT. O processo de automatização dos testes será implementado devido a necessidade de garantir a qualidade do software gerado através de um processo MDD, reduzir o tempo de execução dos testes, reduzir o tempo geral de desenvolvimento de software e por fim melhorar a comunição entre os analistas de negócio e os desenvolvedores.

O restante deste artigo está organizado na seguinte forma: na Seção 2 é apresentado a fundamentação teórica. Na Seção 3 são apresentados o MDT e os trabalhos relacionados. Na Seção 4 é apresentada a proposta deste trabalho, em conjunto com o método de pesquisa, a prova de conceito e os resultados obtidos. Por fim são apresentadas as considerações finais e trabalhos futuros.

2 MODEL DRIVEN DEVELOPMENT

Na Engenharia de Software diferentes paradigmas de desenvolvimento vêm surgindo no intuito de oferecer uma maior produtividade sem perda de qualidade. Um desses paradigmas é o MDD cujo o objetivo é colocar os modelos como artefato central do processo de desenvolvimento, ao invés do código-fonte [11].

Deursen *et al.* [12] relatam que a principal diferença entre o processo tradicional de desenvolvimento de software e o MDD são os artefatos gerados ao longo das fases do MDD, os quais são modelos passíveis de transformação especificados através de linguagens específicas de domínio, ou seja as *Domain-Specific Language*(DSL) o que sugere altos índices de automação entre as fases. Uma DSL é uma linguagem pequena, normalmente declarativa focada em um domínio de um problema particular [12] [13].

Lucredio [14] relata que, para possibilitar a criação de modelos, é necessária uma ferramenta de modelagem para a construção dos modelos de domínio na qual os modelos precisam ser semanticamente completos e corretos, pois necessitam também serem compreendidos por um computador. Neste contexto, os modelos servem como entrada para as transformações que irão gerar outros modelos. Para definir as transformações é necessária uma ferramenta que permita que o desenvolvedor defina regras de mapeamento de modelos para modelos, ou de modelo para código.

No MDD os modelos fornecem a base para a compreensão do comportamento do sistema e a geração das implementações. Com isso, o objetivo do MDD é reduzir a distância que existe entre o domínio do problema e o domínio tecnológico da solução, utilizando modelos abstratos que protegem os desenvolvedores das complexidades inerentes da plataforma de implementação [1]. Para isso, o MDD assim como as abordagens orientadas a modelos, utiliza-se de modelos abstratos e transformações (semi-) automática de modelos ou de modelos para código [15].

Além dos modelos no MDD, o processo de geração é composto por *templates* que segundo Czarnecki *et al.* [13], são arquivos textos quaisquer instrumentados com condições de seleção e expansão de código. Essas construções são responsáveis por realizar consultas de uma entrada que pode ser um programa, uma especificação textual ou diagramas [16]. O resultado desta consulta é utilizada como parâmetro para produzir código fonte.

Tolvanen *et al.* [2], Kleppe *et al.* [3], Bittar *et al.* [4], destacam os benefícios para o processo de desenvolvimento utilizando um processo de MDD como:

- **Produtividade:** o tempo do desenvolvimento será melhor aproveitado, pois será gasto na produção de modelos de mais alto nível;
- **Melhorias na manutenção**: no desenvolvimento convencional, a urgência inerente às atividades de manutenção faz com que os desenvolvedores insiram modificações diretamente no código, fazendo com que a documentação logo fique desatualizada;
- **Portabilidade**: um mesmo modelo pode ser transformado em código para diferentes plataformas;
- **Interoperabilidade**: cada parte do modelo pode ser transformada em código para uma plataforma diferente, resultando em um software que executa em um ambiente heterogênio, porém mantendo a funcionalidade global; e
- **Reutilização:** a reutilização de artefatos de alto nível proporciona maiores benefícios do que a reutilização de código fonte;

Selic [17] afirma que o foco do MDD são os modelos, ao invés dos programas de computador. Com isso os modelos são criados utilizando conceitos menos ligados à tecnologia de implementação subjacente e são mais próximos do domínio do problema em relação às linguagens de programação.

No MDD, a principal idéia é realizar a transformação de modelos de maiores níveis de abstração (domínio do problema) em modelos mais concretos (domínio solução) até se obter o sistema (o código gerado), fazendo com que as futuras modificações do sistema produzido sejam realizadas apenas no modelo mais abstrato [18].

2.1 Celera

A fim de promover a utilização do MDD através de especificações, a empresa Exactus Software desenvolveu um processo MDD denominado Celera. Esse processo tem como característica permitir que através de uma entrada de especificações, sejam gerados códigos nas linguagens *Visual Basic* (V.B.) e Cobol.

A geração de código realizada pela Celera é baseada na especificação dos modelos inseridos no processo MDD em uma estrutura *tree* (árvore), de acordo com os *templates* de geração desenvolvidos de acordo com as necessidades de implantações de novos casos forem surgindo.

Cada modelo possui suas propriedades, nelas são especificadas o que serão utilizadas na geração de código, como por exemplo:

- A visibilidade dos componentes;
- Habilitação / Desabilitação do componente;
- Posição na qual o componente será inserido na *Graphical user interface* (GUI);
- Obrigatoriedade do componente;
- Regra de validação do componente a ser gerado;
- Exibição de rótulo (*label*);
- Molduras (posição, largura, comprimemento);
- Definição de filtros (componentes, *labels*, botões); e
- Tipo de componente (botão, caixa de texto, *check*);

Toda a geração de código é realizada, de acordo com as propriedades especificadas na ferramenta Celera, por exemplo: tamanho do campo, tipo do componente (botão, texto, *check*), obrigatoriedade, visibilidade, posição a ser alocada na tela.

Com a utilização da Celera pela Exactus Software, destacam-se as seguintes vantagens:

- **Produtividade**: o desenvolvimento do software é constitído de projeto de alto nível e codificação. Os modelos são produzidos antes da codificação, servindo de auxílio às tarefas de desenvolvimento e manutenção;
- **Manutenibilidade**: A manutenção restringe-se principalmente nos templates, e não no código gerado;
- **Corretude**: Além do gerador não introduzir erros acidentais, como erros de digitação, o gerador permite que a identificação de erros conceituais, sejam verificados em um nível mais alto de abstração; e
- **Comunicação**: com a utilização da Celera, diversos profissionais possuem meios mais efetivos para a comunicação, uma vez que os modelos são mais abstratos do que os códigos. Neste contexto, especialistas de domínio tem um papel mais ativo no processo, podendo utilizar diretamente os modelos para identificar os conceitos do negócio.

No entanto destacam-se desvantagens como:

- O custo para se projetar, implementar e manter um processo MDD.
- A dificuldade de se balancear entre especificidade do domínio e linguagens de programação genéricas;
- A complexidade do processo MDD com as transformações e geradores de código, pois tratam-se de artefatos inerentemente mais difíceis de construir e manter;
- O processo MDD exige profissionais com habilidades na construção de linguagens, na Celera, nas transformações e nos geradores de código; e
- O desenvolvimento dos artefatos específicos do processo MDD exige profissionais com habilidades nas transformações e nos geradores de código.

3 MODEL DRIVEN TESTING

Uma das características presentes nos testes tradicionais é que, os casos de testes geralmente são escritos manualmente por meio de análise de requisitos [19]. Uma das dificuldades de se elaborar os casos de testes de forma manual é a necessidade da utilização de mais recursos e tempo [19]. No entanto, quando se utiliza a abordagem MDT, essas necessidades (recurso e tempo) são reduzidas, visto que, o MDT automatiza a geração de casos de teste.

Para isso, inicialmente é desenvolvido um modelo de teste para descrever o comportamento esperado do *System Under Test* (SUT), isto é, o sistema que deverá ser testado. Uma vez finalizada a geração de modelo de teste, os casos de teste são projetados (automaticamente) a partir de modelos baseados em critérios de cobertura que foram selecionados [20]. Sendo assim, [6] definem o MDT como uma abordagem para a geração de artefatos de teste, composto por casos de teste, dados de teste e oráculos, através de modelos de desenvolvimento obtidos pela regra de transformação entre modelos e a geração automática.

Neste contexto, Jiao *et al.* [21] relatam que o MDT tornou-se um dos fatores para geração e *design* de testes, isso possibilita a geração da base de conhecimento para ánalise automática de defeitos e localização do problema. Para isso, Jiao *et al.* [21] relatam que o MDT deve ser composto pelas seguintes etapas:

- Criação do modelo do sistema de teste SUT alinhado com o modelo de design;
- Geração de caso de teste e oráculo; e
- Execução de casos de teste em modelo de desing executável e/ou sistema em execução, comparando o resultado de teste com o oráculo.

Uma das características presentes na abordagem MDT é que essa abordagem é baseada em modelos, ou seja um artefato utilizado para a construção do sistema e na comunicação das decisões de design [17]. Lima *et al*. [7] demonstram dois benefícios ao adotar a abordagem MDT: a redução do custo (após implantada) e a automação no processo de geração de casos de teste. Além disso, os modelos de teste podem auxiliar na identificação de erros antes da transformação dos modelos em código [8].

Baker *et al.* [6] e Kashyap [22] também relatam benefícios de se utilizar o MDT como: uma melhora nas especificações aliada a uma integração dos testadores nas fases iniciais do processo de desenvolvimento, o aumento dos testes contínuos, uma melhora na qualidade no ato da entrega do produto final, a previsibilidade e redução de riscos e a melhora na eficiência e eficácia dos produtos.

Em contrapartida, a utilização do MDT pode requerer um maior custo durante o período de implementação, além da dificuldade de se balancear entre especificidade do domínio e linguagens de programação genéricas [8] [9]. Essas dificuldades podem influenciar na geração de estudo e pesquisa referente ao tema abordado.

3.1 Trabalhos relacionados

Na linha de pesquisa relacionada ao uso do MDT, Li *et al.* [23] retratam uma metodologia de MDT para uma aplicação Web, a fim de permitir que usuários definam meta-modelos oferencendo uma interface mais amigável. Este trabalho relata a realização de um experimento como um plugin para a IDE Eclipse denominado MDWATP, utilizando um motor de template e uma DSL própria. O MDWATP está em desenvolvimento e como trabalhos futuros é proposta a realização de testes do MDWATP em outras aplicações da web.

Alves *et al.* [24] discutem objetivos e questões sobre como integrar os processos MDD e MDT, apresentando uma proposta de integração a partir de um estudo de caso para sistema de empréstimo de biblioteca, utilizando o motor de *template Atlas Transformation Language* (ATL) que ser caracteriza por ser uma linguagem de transformação híbrida e bastante expressiva. Alves *et al.* utilizam o motor de *template* ATL em conjunto com diagramas da UML 2.0. Como trabalhos futuros, é proposto substituir as regras para o motor de *template Query View Transformation* (QVT) ou seja, uma linguagem para transformações entre modelos padronizada pela *Object Management Group* (OMG) com base na *Object Constraint Language* (OCL).

Lamancha *et al.* [25] apresentam um MDT para geração automática de casos de teste em um estudo de caso para desenvolvimento de jogos. O motor de *template* utilizado nesse trabalho é o QVT em conjunto com a UML 2.0 com diagramas de classe e sequência. Como trabalhos futuros é proposto incluir outros tipos de processos de geração e automatização.

Almeida e De Oliveira [11] assim como este trabalho, promovem uma integração prática do MDD com o MDT denominado Qualitas, como um modelo de processo de desenvolvimento de software orientado a modelos. O processo Qualitas foi desenvolvido no departamento de Tecnologia da Informação (TI) do Hospital Universitário (HU) da Universidade Federal de Sergipe (UFS), utilizando um motor de *template* e uma DSL própria. Como trabalhos futuros, deseja-se validar o modelo proposto em estudos e contextos reais.

Olajubu *et al.* [26] relatam tentativas preliminares para automatizar a geração de casos de teste a partir de requisitos de modelos de softwares e utilizaram um caso de uso industrial e o o motor de *template Model-to-Text* (M2T) ou seja, uma técnica de geração de artefatos textuais a partir de modelos a fim de realizar as transformações necessárias para a geração de casos de teste. Além do *template* M2T, Olajubu *et al.* também utilizam uma DSL própria. Esse trabalho relata um estudo de caso industrial em sistemas de aviação na empresa GE *Aviation Systems* como uma aplicação desktop utilizando uma técnica de modelagem textual semelhante a UML, no entanto não descreve maiores detalhes sobre a técnica. Como trabalhos futuros é proposto ampliar a geração de casos de teste a partir de novos tipos de requisitos.

Com base no cenário descrito anteriormente é possível afirmar que este trabalho difere de Li *et al.* [23], Alves *et al.*[24], Lamancha *et al.*[25] , Almeida e De Oliviera[11] e Olajubu *et al.* [26] pois é realizado um processo de geração de *scripts* a serem utilizados pela ferramenta Sikuli como forma de testar automaticamente as GUIs, geradas pelo processo MDD em sistemas legado. Com isso o MDD está sendo utilizado em conjunto com o Sikuli para automatizar e viabilizar a geração de casos de teste, pois sem esse processo haveria a necessidade de um investimento na contratação de testadores para realizar o processo de teste de todas as telas do sistema.

4 PROPOSTA

A proposta deste trabalho é incluir um processo MDT no contexto de um processo MDD já existente a fim de gerar *scripts* a serem utilizados pela ferramenta Sikuli, de acordo com os modelos especificados e com os *templates* desenvolvidos.

A Figura 1 ilustra o processo atual de desenvolvimento e teste realizado na Exactus Software, e a Figura 2 ilustra a proposta deste trabalho. No processo atual, conforme ilustrado na Figura 1 os desenvolvedores especificam os modelos, que são transformados em código V.B. e Cobol, e ao final os testadores executam os testes manualmente sobre o produto gerado. No entanto, conforme ilustrado na Figura 2 a proposta deste trabalho é automatizar o processo de testes, incluindo no processo MDD o MDT. Uma das características deste processo é que os desenvolvedores especificam os modelos que são transformados em código V.B., Cobol e *Script* Sikuli. Ao final os testadores especificam os modelos de casos de teste que serão utilizados como entrada, na qual serão transformados em *Scripts* Sikuli pela Celera a fim de realizarem os testes automaticamente sobre os produtos produzidos pela Celera.

Figura 1: Modelo de teste de desenvolvimento atual

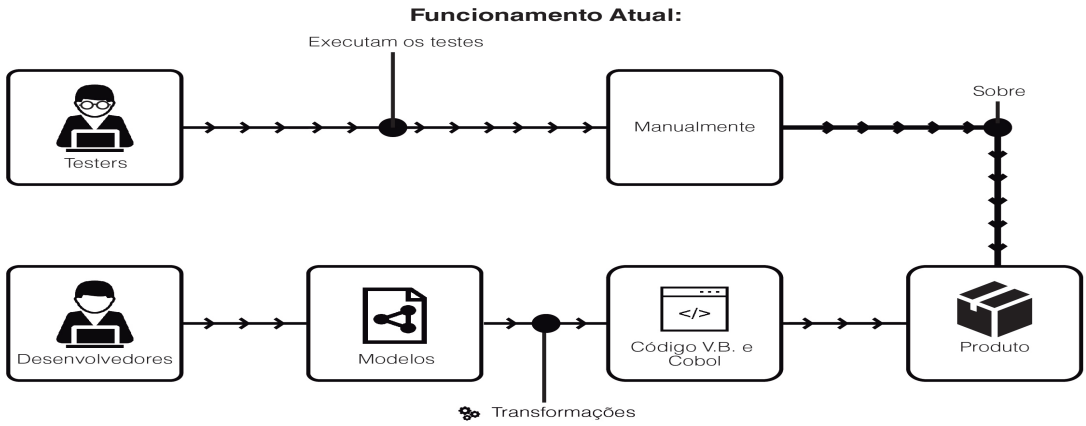
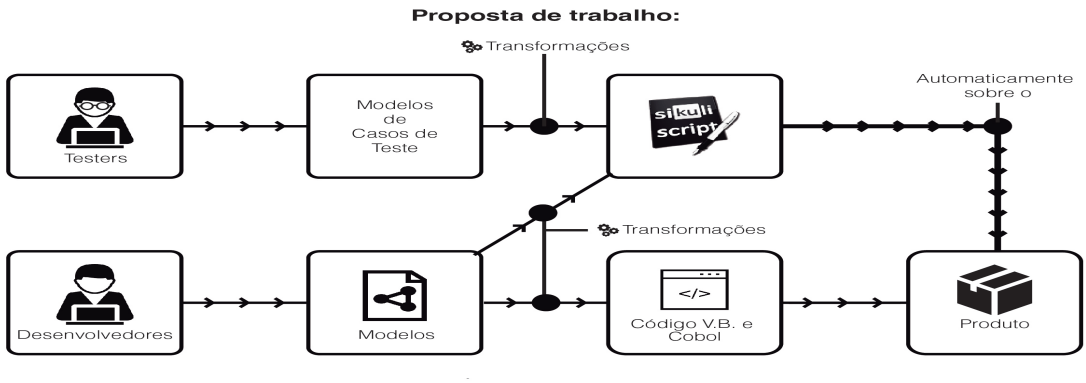


Figura 2: Proposta de modelo de teste e desenvolvimento deste trabalho



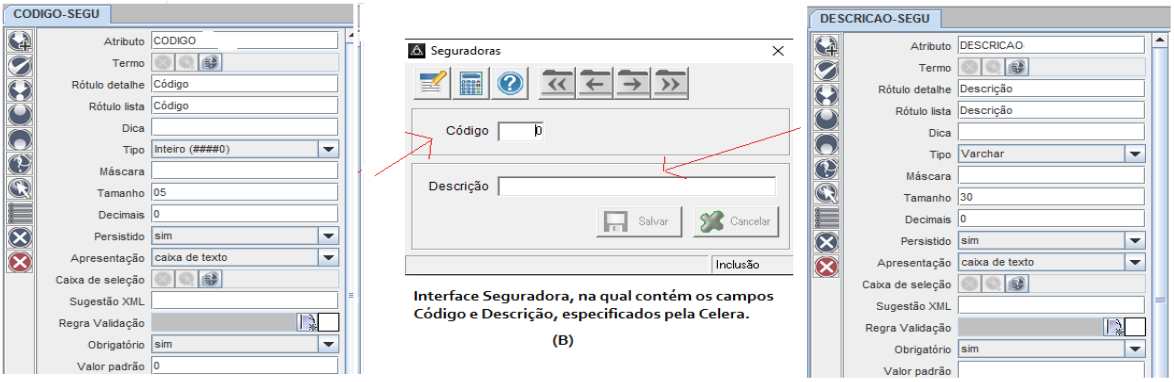
Para realizar os testes sobre o produto gerado pela Celera será utilizada a ferramenta Sikuli. O Sikuli tem como característica a utilização de *screenshots* (imagens) para gerar testes utilizando padrões de imagens. Esses testes são direcionados a eventos que podem ocorrer através de dispositivos de entrada (mouse e teclado) em uma GUI. Nesse sentido o uso do Sikuli deve contribuir com essa limitação e auxiliar atividades de manutenção desse sistema.

Com isso, a proposta de gerar os testes automaticamente no processo MDD deve-se pela necessidade de diminuir o tempo e os custos na elaboração dos testes feitos pelos desenvolvedores e *testers*. Atualmente todo o processo é realizado manualmente sem a utilização de nenhuma ferramenta. Com isso, a fim de minimizar o tempo com a execução dos testes, este trabalho tem como objetivo automatizar o processo dos testes realizados, visto que o processo de confecção dos *scripts* Sikuli se tornem automáticos através de um processo MDD. Esses *scripts* serão gerados pela Celera através das especificações inseridas nos modelos, transformações e executadas na ferramenta Sikuli.

4.1 Geração dos casos de teste no MDT

Sommerville [27] e Delmaro *et al.*[28] definem casos de teste como uma forma de estabelecer as entradas a serem informadas pelo testador (manualmente ou com apoio ferramental) e os resultados esperados a partir dessa ação.

Figura 3: Modelo de entrada especificado na Celera



Atributo de domínio Código, especificado na Celera, para produzir a interface Seguradoras

(A)

Interface Seguradora, na qual contém os campos Código e Descrição, especificados pela Celera.

(B)

Atributo de domínio Descrição, especificado na Celera para produzir a interface Seguradoras.

(C)

Os casos de teste são compostos por entradas, os passos e por fim os oráculos que segundo Sommerville [27] e Delmaro *et al.* [28] são mecanismos que possibilitam definir a saída ou comportamento esperado de uma execução.

Considerando o contexto da implementação do processo MDT na empresa Exatus Software, neste trabalho os casos de teste são obtidos através de transformações de dois tipos de modelos, os modelos de teste gerados pelos testadores em uma *Fit Table* e os modelos especificados na Celera.

As entradas são definidas pelas *Fit Tables* (arquivos texto) nos casos de teste. Devem conter as informações de valores (válidos e inválidos) para a realização dos casos de teste. As *Fit Tables* devem definir todos os casos de teste a serem executados de acordo com a GUI na qual será realizado os testes. As *Fit Tables* são definidas pelos Analistas de negócio em conjunto com os testadores da empresa Exactus Software.

Os modelos definidos pela Celera são estabelecidos pelos desenvolvedores em conjunto com os arquitetos de software de acordo com as especificações para cada caso de uso a ser desenvolvido a fim de fazer um vínculo dos elementos das *Fit Tables* com os compenentes da interface da tela. Com isso, os desenvolvedores especificam os atributos de domínio a serem utilizados em cada GUI. Os atributos de domínio utilizados serão: o tamanho do componente, o tipo (VarChar, inteiro, decimal, date, char) e o tipo de apresentação (caixa de texto, botão, check ou uma combo).

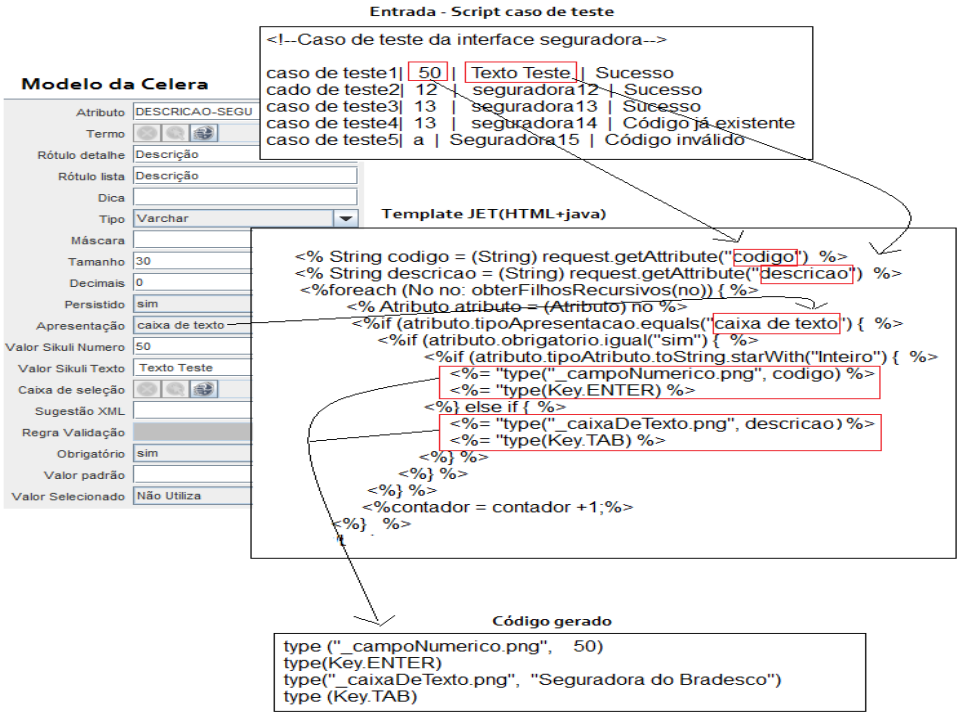
A Figura 3 ilustra o processo de especificação dos atributos de domínio produzida pela Celera para definirem uma GUI. Nesta Figura, por exemplo, são especificados os campos código e descrição, na qual Figura a 3 (A) ilustra a especificação do atributo de domínio referente ao campo código. A Figura 3 (C) demonstra a especificação do atributo de domínio descrição a serem utilizados na produção da interface seguradora, conforme demonstrado na Figura 3 (B).

Os próximos componentes de casos de teste a serem estabelecidos são os passos, que são definidos pelos modelos da Celera. Em termos de teste de software, os modelos da Celera são importantes pois definem os passos do caso de teste. Outro fator a ser destacado é que a elaboração dos modelos da Celera já fazem parte do processo de MDD empregado pela Exactus Software. Por isso, esse passo não difere do processo que já é utilizado na empresa, apenas sendo aproveitado para a geração de casos de teste.

Por fim, os oráculos serão obtidos através das especificações definidas nas *Fit Tables* pelos testadores, onde é demonstrado as saídas esperadas pelos casos de teste.

Com isso, o processo de automomatização dos testes dos sistemas legados produzidos pela Celera, ocorre através das especificações dos casos de teste e dos atributos de domínio já introduzidos pela Celera. Neste contexto, o processo Celera produzirá *scripts* Sikuli utilizando instruções definidas em *templates*. Essas instruções serão responsáveis por realizar consultas de entrada nas *Fit Tables* e nos modelos da Celera, a fim de serem transformados

Figura 4: Geração de script Sikuli baseada em *templates*



em códigos para *Scrips* Sikuli. Sendo assim todo o processo de geração de *scritps* Sikuli ocorrerá através dos modelos que serão utilizados dentro do processo MDT e dos casos de teste desenvolvidos pelos testadores.

A Figura 4 ilustra todo o processo da definição dos elementos que serão utilizados na produção do *scripts* Sikuli pelo processo Celera. Cada trecho do *template* será processado para consultar os casos de testes e os modelos definidos na Celera para produzir o código correspondente, a partir de dois modelos.

Os modelos da Celera, são utilizados para iniciar as transformações, nesses modelos os desenvolvedores especificam os atributos de domínio a serem utilizados na produção dos *scripts* Sikuli. O segundo modelo utilizado na geração de código é uma *Fit Table* (arquivo texto) de casos de teste que devem conter as informações de valores (válidos e inválidos) a serem introduzidas pelo Sikuli para a realização dos casos de teste.

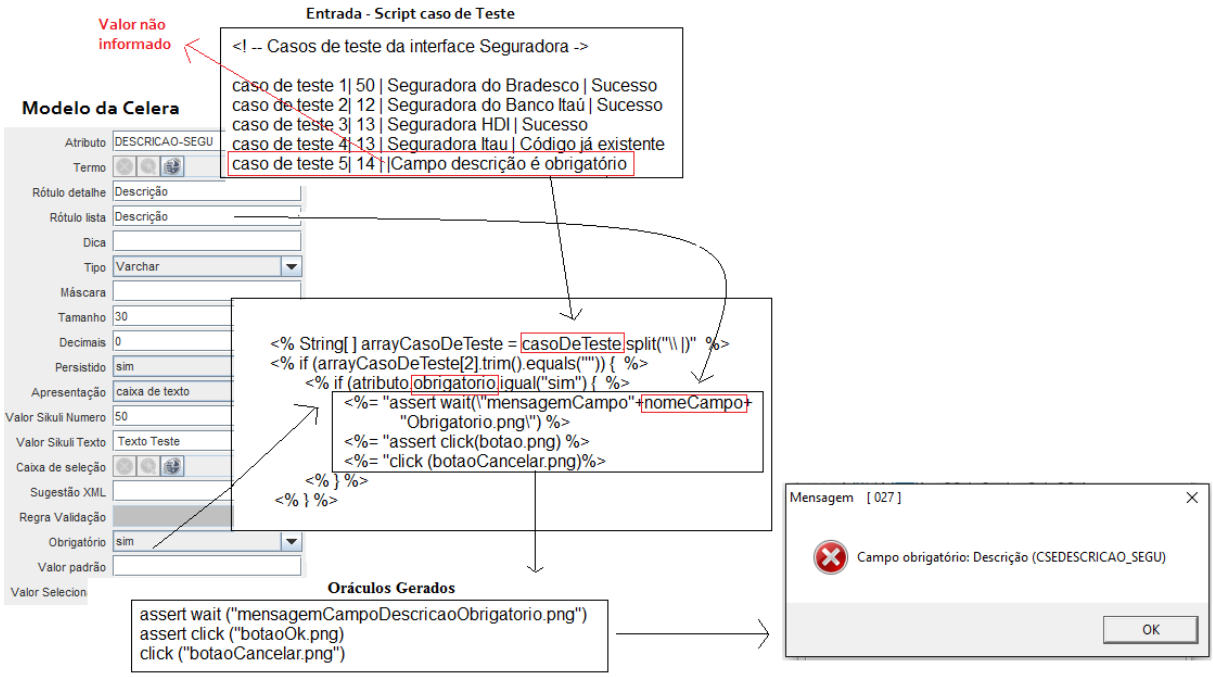
Após da definição dos modelos da Celera e das *Fit Tables*, é definido o processador de *templates* em um formato *Java Emitter Templates* (JET) com código Java, responsável por instanciar o *template* com base nas entradas. Esse elemento é encarregado por consultar as entradas e retornará o código *Script* Sikuli.

Por fim é necessário verificar os oráculos dos casos de teste conforme ilustrado na Figura 5, na qual o processo de geração é composto pelos modelos da Celera, pelas *Fit Tables* e o processador de *templates*. Os modelos da Celera são responsáveis por repassar para o processador de *template* as informações referentes a obrigatoriedade, descrição e se o campo é único ou não. As *Fit Tables* representadas pelos *scripts* dos casos de teste, são responsáveis por repassarem para o processador de *template* o caso de teste a ser utilizado na geracão dos oráculos. Por fim, o processador de *templates* é responsável por gerar os oráculos através definições dos atributos de domínio no processo MDD e dos *scripts* de entrada dos casos de teste.

4.2 Prova de conceito

No contexto deste estudo, foi realizado uma prova de conceito como forma de validar esta proposta. Para isso, foi definida a seguinte Questão de Pesquisa (QP).

Figura 5: Geração de oráculos



- **QP**: A geração de casos de teste automático do processo MDD da empresa Exactus Software melhora a efetividade da abordagem de teste da empresa?

Considerando a QP acima, foi elaborada a seguinte hipótese.

- **H1**: O uso da geração de casos de teste automático do processo MDD da Exactus Software aumenta a efetividade da abordagem de teste da empresa.

Neste trabalho, a efetividade relatada na QP será medida a fim de :

- Verificar o quanto os casos de testes gerados neste processo automático serão úteis para a aplicação obtida através do processo MDD;
- Avaliar a qualidade dos casos de teste gerados automaticamente; e
- Avaliar o resultado do processo de automatização de teste em relação ao manual.

Metodologia

Considerando a hipótese H1 descrita anteriormente, para testar a validade da hipótese foi conduzido o processo de MDT dentro da empresa Exactus Software contemplando um determinado conjunto de funcionalidades da aplicação sendo desenvolvida.

A efetividade considerando H1 vai ser medida inicialmente utilizando a métrica de *Code Coverage* dos casos de testes gerados automaticamente.

Com isso, foram implementados módulos de geração de casos de teste para as seguintes funcionalidades:

- **Seguradora**: Uma GUI que contém dois campos (Código e Descrição). A definição e a sequência dos campos na GUI é demonstrado abaixo;

  - Código : Campo deverá ser obrigatório, do tipo numérico (no máximo 4 dígitos), único e sua forma de apresentação deverá ser do tipo caixa de texto; e
  - Descrição: Campo deverá ser obrigatório, do tipo VarChar como no máximo 30 caracteres e sua forma de apresentação deverá ser do tipo caixa de texto.

- **Moeda**: Uma GUI que contém cinco campos (Código, Sigla, Descrição, Tipo e Frequência). A definição e a sequência dos campos na GUI é demonstrado abaixo;
  - Código : Campo deverá ser obrigatório, do tipo Varchar com 1 caracter apenas e sua forma de apresentação deverá ser do tipo caixa de texto;
  - Sigla : Campo deverá ser obrigatório com no máximo 8 caracteres, do tipo VarChar e sua forma de apresentação deverá ser do tipo caixa de texto;
  - Descrição: Campo deverá ser um campo obrigatório, do tipo VarChar, com no máximo 30 caracteres e sua forma de apresentação deverá ser do tipo caixa de texto;
  - Tipo: Campo deverá ter apenas um caracter, do tipo VarChar, é obrigatório e sua forma de apresentação deverá ser do tipo caixa de seleção; e
  - Frequência : Campo deverá ter apenas um caracter, do tipo VarChar, é obrigatório e sua forma de apresentação deverá ser do tipo caixa de seleção.

- **Fornecedor**: Uma GUI que contém 12 campos (Código, Nome, Endereço, Bairro, Município, UF, Cep, Fone, Fax, Contato, CNPJ/CPF e Insc. Estadual). A definição e a sequência dos campos na GUI é demonstrado abaixo;
  - Código : Campo deverá ser um valor numérico, é obrigatório, com no máximo 15 caracteres e sua forma de apresentação deverá ser do tipo caixa de texto;
  - Nome : Campo deverá ser obrigatório, do tipo VarChar com no máximo 50 caracteres e sua forma de apresentação deverá ser do tipo caixa de texto;
  - Endereço : Campo deverá ser obrigatório, do tipo VarChar com no máximo 50 caracteres e sua forma de apresentação deverá ser do tipo caixa de texto;
  - Bairro : Campo deverá ser obrigatório, do tipo VarChar com no máximo 20 caracteres e sua forma de apresentação deverá ser do tipo caixa de texto;
  - Município : Campo deverá ser obrigatório, do tipo VarChar com no máximo 30 caracteres e sua forma de apresentação deverá ser do tipo caixa de texto;
  - UF: Campo deverá ser obrigatório, do tipo VarChar com no máximo 2 caracteres e sua forma de apresentação deverá ser do tipo caixa de seleção;
  - Cep: Campo deverá ser obrigatório, do tipo Inteiro, com no máximo 8 caracteres e sua forma de apresentação deverá ser do tipo caixa de texto;
  - Fone: Campo deverá ser obrigatório, do tipo Inteiro, com no máximo 8 caracteres e sua forma de apresentação deverá ser do tipo caixa de texto;
  - Fax : Campo deverá ser obrigatório, do tipo Inteiro, com no máximo 8 caracteres e sua forma de apresentação deverá ser do tipo caixa de texto;
  - Contato : Campo deverá ser obrigatório, do tipo VarChar com no máximo 20 caracteres e sua forma de apresentação deverá ser do tipo caixa de texto;

  - CPF / CNPJ: Campo deverá ser do tipo numérico, obrigatório com no máximo 14 caracteres e sua forma de apresentação deverá ser do tipo caixa de texto; e

  - Insc. Estadual : Campo deverá ser do tipo VarChar, obrigatório com no máximo 15 caracteres e sua forma de apresentação deverá ser do tipo caixa de texto.

- **Tipo do Item**: Uma GUI que contém dois campos (Código e Descrição). A definição e a sequência dos campos na GUI é demonstrado abaixo; e

  - Código: Campo deverá ser obrigatório, do tipo numérico (no máximo 4 dígitos), único e sua forma de apresentação deverá ser do tipo caixa de texto; e

  - Descrição: Campo deverá ser obrigatório, do tipo VarChar como no máximo 30 caracteres e sua forma de apresentação deverá ser do tipo caixa de texto.

- **Classificação Contábil**: Uma GUI que contém dois campos (Código e Descrição). A definição e a sequência dos campos na GUI é demonstrado abaixo.

  - Código: Campo deverá ser obrigatório, do tipo numérico (no máximo 2 dígitos), único e sua forma de apresentação deverá ser do tipo caixa de texto; e

  - Descrição: Campo deverá ser obrigatório, do tipo VarChar como no máximo 30 caracteres e sua forma de apresentação deverá ser do tipo caixa de texto.

Resultados

Para validar a proposta deste trabalho com o uso de uma abordagem MDT, por meio da geração de casos de teste para a ferramenta Sikuli, onde dois modelos são transformados e utilizados nos sistemas, foi realizado uma prova de conceito com a avaliação da métrica de *Code Coverage*.

A métrica de *Code Coverage* foi adotada para medir a qualidade dos casos de teste produzidos do código V.B. produzido pela Celera, a fim de verificar os códigos referentes as camadas de negócio e persistência. Será descartado no entanto os códigos referentes à camada de visão e os componentes (de terceiros e os que foram desenvolvidos em outras linguagens, porém de domínio da própria Exactus Software) utilizados no sistema produzido pela Celera.

Para este estudo foram selecionados cinco GUIs (conforme demonstrado na Tabela 1). O código gerado pela GUI Seguradora tinha 78 linhas de código, sendo que os testes gerados pelo processo MDT proposto neste trabalho cobriram 77 destas linhas, totalizando 98% de cobertura. O código gerado pela GUI Moeda tinha 107 linhas de código, sendo que os testes gerados pelo processo MDT proposto neste trabalho cobriram 101 destas linhas, totalizando 94,39% de cobertura. O código gerado pela GUI Fornecedor tinha 222 linhas de código, sendo que os testes gerados pelo processo MDT proposto neste trabalho cobriram 211 destas linhas, totalizando 95,04% de cobertura. O código gerado pela GUI Tipo do Item tinha 99 linhas de código, sendo que os testes gerados pelo processo MDT proposto neste trabalho cobriram 95 destas linhas, totalizando 95,95% de cobertura. O código gerado pela GUI Classificação Contábil tinha 98 linhas de código, sendo que os testes gerados pelo processo MDT proposto neste trabalho cobriram 94 destas linhas, totalizando 95,91% de cobertura.

Tabela 1: Cobertura do código V.B. gerado pela Celera

GUI	Total de Linhas	Total de linhas Cobertas	% de cobertura
Seguradora	78	77	98%
Moeda	107	101	94,39%
Fornecedor	222	211	95,04%
Tipo do Item	99	95	95,95%
Classificação Contábil	98	94	95,91%

Considerando os resultados acima, podem ser observadas evidências que suportam a hipótese alternativa no entanto, mais estudos devem ser conduzidos.

5 DISCUSSÃO

Diferentemente do trabalho atual, os trabalhos de Li *et al.* [23], Alves *et al.* [24] e Lamancha *et al.* [25] não utilizam a métrica de *Code Coverage* como forma de avaliação dos casos de teste pois, os trabalhos estão focados em construir modelos de casos de teste utilizando diagramas de UML 2.0. Com isso, os resultados são demonstrados utilizando diagramas de sequência para os casos de testes desenvolvidos pelos testadores de acordo com cada GUI a ser realizado o teste.

Enquanto Li *et al.* [23], Alves *et al.* [24] e, Lamancha *et al.* [25] apresentam os resultados em diagramas de sequencia da UML 2.0, Almeida e De Oliveira [11] relatam que o trabalho está em andamento e que apenas as fases de revisão bibliográfica e a elaboração da proposta foram concluídas, com isso não é relatado nenhum resultado parcial ou final neste trabalho.

Diferentemente dos trabalhos citados acima cujo os resultados são demonstrados em diagramas da UML 2.0 e do trabalho atual que utiliza a métrica de *Code Coverage* em sua avaliação, Olajubu *et al.* [26] apresentam os resultados em uma tabela, indicando se os casos de testes produzidos pelo processo MDT desenvolvido obtiveram sucesso ou falha.

Com isso, é importante destacar que esse trabalho utiliza o Sikuli, para realizar os testes de acordo com cada GUI e seus casos de testes respectivos, para os sistemas legados produzidos pela Celera. O Sikuli é utilizado neste trabalho devido a uma versão antiga da ferramenta *Visual Studio* adquirida pela empresa Exactus Software, que não tem suporte de teste unitário e *framework* de teste por isso, o Sikuli se mostrou como uma alternativa. Uma dificuldade observada no uso da ferramenta Sikuli, é a instabilidade de interface, porém neste caso, o processo MDD segue um padrão bem estabelecido e definido pela empresa Exactus Software fazendo com que o ponto fraco do Sikuli seja resolvido devido ao processo MDD.

Por fim, é importante destacar que esse trabalho possui uma limitação na qual a prova de conceito conduzido não contempla todas as funcionalidades para o MDT desenvolvido.

6 CONSIDERAÇÕES FINAIS

Esse trabalho apresenta uma forma para automatizar a geração de casos de teste de modelos específicos de domínio da empresa Exactus Software, incorporando o MDT em um processo MDD. Uma transformação baseada em templates é aplicada para automatizar geração de casos de teste, através de especificações de requisitos definidas no processo Celera, em modelos de entrada.

No entanto, esse trabalho está em andamento e os benefícios esperados pela utilização deste trabalho são (i) automatizar o processo de testes da empresa Exactus Software dos sistemas legados gerado pela Celera, (ii) com o processo de automatização dos testes é esperado uma melhoria na qualidade do produto gerado pela Celera, permitindo que os testes acompanhem a evolução e desenvolvimento de um ambiente de teste completo. Com isso, a geração dos *scripts* feito automaticamente dentro do processo Celera que posteriormente será utilizado pela ferramenta Sikuli está sendo validado.

As sugestões de trabalhos futuros são (i) extender a prova de conceito e tentar viabilizar outras técnicas para tentar automatizar ainda mais o processo de geração de casos de teste, (ii) realizar o mesmo estudo com processos de MDD diferente, fora da empresa Exactus Software, sem considerar a Celera.

Referências

[1] FRANCE, R.; RUMPE, B. Model-driven development of complex software: A research roadmap. In: *Future of Software Engineering, 2007. FOSE '07*. [S.l.: s.n.], 2007. p. 37–54.

[2] TOLVANEN, J.-P.; POHJONEN, R.; KELLY, S. Advanced tooling for domain-specific modeling: Metaedit+. In: *Sprinkle, J., Gray, J., Rossi, M., Tolvanen, JP (eds.) The 7th OOPSLA Workshop on Domain-Specific Modeling, Finland*. [S.l.: s.n.], 2007.

[3] KLEPPE, A. G.; WARMER, J. B.; BAST, W. *MDA explained: the model driven architecture: practice and promise*. [S.l.]: Addison-Wesley Professional, 2003.

[4] BITTAR, T. J. et al. Web communication and interaction modeling using model-driven development. In: *Proceedings of the 27th ACM International Conference on Design of Communication*. New York, NY, USA: ACM, 2009. (SIGDOC '09), p. 193–198. ISBN 978-1-60558-559-8. Disponível em: <http://doi.acm.org/10.1145/1621995.1622033>. Acesso em: 30 nov. 2017.

[5] DISTANTE, D. et al. Model-driven development of web applications with uwa, mvc and javaserver faces. In: BARESI, L.; FRATERNALI, P.; HOUBEN, G.-J. (Ed.). *Web Engineering: 7th International Conference, ICWE 2007 Como, Italy, July 16-20, 2007 Proceedings*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007. p. 457–472.

[6] BAKER, P. et al. *Model-driven testing: Using the UML testing profile*. [S.l.]: Springer Science & Business Media, 2007.

[7] LIMA, H. S. et al. Automatic generation of platform independent built-in contract testers. In: *SBCARS*. [S.l.: s.n.], 2007. p. 47–60.

[8] FRANCE, R. B. et al. Model-driven development using uml 2.0: promises and pitfalls. *Computer*, IEEE, v. 39, n. 2, p. 59–66, 2006.

[9] HILDENBRAND, T.; KORTHAUS, A. A model-driven approach to business software engineering. In: CITESEER. *Proceedings of the 8th World Multi-Conference on Systemics, Cybernetics and Informatics (SCI 2004)*. [S.l.], 2004. v. 4, p. 18–21.

[10] PINTO, H. L. M.; BRAGA, J. L. Sistemas legados e as novas tecnologias: técnicas de integração e estudo de caso. *Informática Pública, Belo Horizonte*, v. 7, n. 1, p. 48–69, 2004.

[11] ALMEIDA, C. C. de J.; OLIVEIRA, A. A. de. Qualitas: A proposal of process model development software driven models. In: *Proceedings of the 7th Euro American Conference on Telematics and Information Systems*. New York, NY, USA: ACM, 2014. (EATIS '14), p. 28:1–28:4. ISBN 978-1-4503-2435-9. Disponível em: <http://doi.acm.org/10.1145/2590651.2590678>. Acesso em: 30 nov. 2017.

[12] DEURSEN, A. V.; KLINT, P.; VISSER, J. Domain-specific languages: An annotated bibliography. *Sigplan Notices*, v. 35, n. 6, p. 26–36, 2000.

[13] CZARNECKI, K. et al. Model-driven software product lines. In: *Companion to the 20th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*. New York, NY, USA: ACM, 2005. (OOPSLA '05), p. 126–127. ISBN 1-59593-193-7. Disponível em: <http://doi.acm.org/10.1145/1094855.1094896>. Acesso em: 30 nov. 2017.

[14] LUCRÉDIO, D. Uma abordagem orientada a modelos para reutilização de software. *INSTITUTO DE CIÊNCIAS MATEMÁTICAS E DE COMPUTAÇÃO UNIVERSIDADE DE SÃO PAULO*, p. 37, 2009.

[15] BEYDEDA, S. et al. *Model-driven software development*. [S.l.]: Springer, 2005. v. 15.

[16] CLEAVELAND, J. C. Building application generators. *IEEE Software*, IEEE Computer Society, v. 5, n. 4, p. 25, 1988.

[17] SELIC, B. The pragmatics of model-driven development. *IEEE software*, IEEE Computer Society, v. 20, n. 5, p. 19, 2003.

[18] BUARQUE, A. d. S. M. *Desenvolvimento de software dirigido por modelos: Um foco em engenharia de requisitos*. Tese (Doutorado) — Universidade Federal de Pernambuco, 2009.

[19] FERNANDES, G. F. D. Geração automática de casos de teste a partir de requisitos. 2014.

[20] ABBORS, F.; TRUSCAN, D.; LILIUS, J. Tracing requirements in a model-based testing approach. In: *Advances in System Testing and Validation Lifecycle, 2009. VALID '09. First International Conference on*. [S.l.: s.n.], 2009. p. 123–128.

[21] JIAO, Y. et al. Towards model-driven methodology: a novel testing approach for collaborative embedded system design. In: IEEE. *2006 10th International Conference on Computer Supported Cooperative Work in Design*. [S.l.], 2006. p. 1–5.

[22] KASHYAP, A.; O'REILLY, S. *Lean Approach Through Model-Based Testing*. [S.l.]: Testing Experience Magazine, 2012.

[23] LI, N. et al. A framework of model-driven web application testing. In: *30th Annual International Computer Software and Applications Conference (COMPSAC'06)*. [S.l.: s.n.], 2006. v. 2, p. 157–162. ISSN 0730-3157.

[24] ALVES, E. L.; MACHADO, P. D.; RAMALHO, F. Uma abordagem integrada para desenvolvimento e teste dirigido por modelos. In: *2nd Brazilian Workshop on Systematic and Automated Software Testing*. [S.l.: s.n.], 2008.

[25] LAMANCHA, B. P.; USAOLA, M. P.; GUZMAN, I. G. R. de. Model-driven testing in software product lines. In: *Software Maintenance, 2009. ICSM 2009. IEEE International Conference on*. [S.l.: s.n.], 2009. p. 511–514. ISSN 1063-6773.

[26] OLAJUBU, O. et al. Automated test case generation from domain specific models of high-level requirements. In: *Proceedings of the 2015 Conference on Research in Adaptive and Convergent Systems*. New York, NY, USA: ACM, 2015. (RACS), p. 505–508. ISBN 978-1-4503-3738-0. Disponível em: <http://doi.acm.org/10.1145/2811411.2811555>. Acesso em: 30 nov. 2017.

[27] SOMMERVILLE, I.; SAWYER, P. *Requirements engineering: a good practice guide*. [S.l.]: John Wiley & Sons, Inc., 1997.

[28] DELAMARO, M. E.; MALDONADO, J. C.; JINO, M. *Introdução ao teste de software*. [S.l.: s.n.], 2007.