Aquila: uma Linguagem de Domínio Específico para Teste Baseado em Modelos para Projetos Ágeis

Aline Zanin, Avelino F. Zorzo

[1]Pontifícia Universidade Católica do Rio Grande do Sul (PUCRS)

aline.zanin@acad.pucrs.br, avelino.zorzo@pucrs.br

Resumo. *A aplicação da técnica de Teste Baseado em Modelos (Model-based Testing - MBT) pode trazer diversos benefícios para a melhoria da qualidade de software. Usualmente MBT é aplicado apenas em modelos tradicionais de ciclo de vida de desenvolvimento de software e.g. Cascata, sendo que poucos trabalhos exploram a sua aplicação no contexto de equipes ágeis. Desta forma, neste trabalho apresenta-se uma Linguagem de Domínio Específico, denominada Aquila, que foi projetada para modelagem de testes funcionais em equipes ágeis. Aquila é uma extensão da Linguagem de Domínio Específico Domain-Specific Language (DSL) Gherkin, aonde novas palavras chaves, relacionadas a testes funcionais, são adicionadas para permitir a geração automatizada de scripts. Neste trabalho apresentamos uma revisão de literatura, a arquitetura da linguagem Aquila e um exemplo de uso da aplicação desta DSL.*

Abstract. *Although Model-based Testing (MBT) brings several benefits for improving software quality, usually MBT is only used in traditional software development life cyles, e.g. waterfall, and there is almost no work that shows how to use MBT in agile environments. Hence, this paper presents a Domain-dpecific Language (DSL), called Aquila, which was designed to model functional testing in agile environments. Aquila is an extension of the Ghergin DSL in which new keywords related to agile software development and functional testing are added. In this paper, we present a literature review, Aquila's architecture and a example of use that was used to show how Aquila can be applied.*

1. Introdução e Motivação

A busca crescente por aumento da qualidade nos produtos e serviços entregues aos clientes, faz com que as empresas procurem otimizar os seus processos e ferramentas para tornar possível atender a este propósito. Neste sentido, as técnicas utilizadas para realização de testes de software precisam acompanhar esta tendência e estarem sendo atualizadas e aprimoradas constantemente.

Entre estas técnicas encontra-se Teste Baseado em Modelos (*Model-Based Testing* - MBT). MBT fundamenta-se na criação de artefatos de testes, baseados na reutilização dos modelos criados pelas equipes de desenvolvimento para a realização de outras atividades, como por exemplo: análise e especificação de requisitos, análise de sistemas e programação. A técnica de MBT agrega diversos benefícios às empresas que optam pela sua utilização, estando entre os principais: facilitação da manutenção dos artefatos de testes; melhor rastreabilidade entre requisito de sistema e artefato de teste; e melhor visualização dos requisitos, de forma gráfica, através de modelos [El-Far and Whittaker 2001].

Contudo, embora os benefícios de MBT já sejam conhecidos e difundidos, poucos trabalhos apresentam a aplicação desta técnica em projetos que fazem uso de metodologias ágeis de desenvolvimento de sistemas. Isto se deve ao

fato de MBT trazer alguns desafios quando aplicado neste contexto. Estes desafios dizem respeito, principalmente a gerência do tempo e dos recursos demandados para a criação dos modelos e ao fato deles serem geralmente construídos em ferramentas separadas das ferramentas de desenvolvimento tradicionalmente utilizadas [Katara and Kervinen 2006][Jalalinasab and Ramsin 2012][Entin et al. 2012].

Estes desafios são notados especificamente no contexto de desenvolvimento ágil de software porque as equipes trabalham com ciclos curtos de entregas contínuas, o que exige que o tempo de criação dos modelos seja reduzido para que exista tempo hábil para execução dos testes antes da entrega ao cliente. Além disso, as metodologias ágeis de desenvolvimento embasam sua estrutura no manifesto ágil para o desenvolvimento de sistemas [Fowler and Highsmith 2001] e neste documento, os princípios de comunicação, interação e trabalho colaborativo, acima de documentações extensas, responsabilidades individuais e burocracia contratual, são norteadores. Desta forma torna-se inviável a dedicação de profissionais especificamente para a criação dos modelos.

Visando o cumprimento do princípio básico de colaboração e integração entre os profissionais que atuam no desenvolvimento do sistema e destes com o cliente, é comum em equipes ágeis a utilização de *user stories* para documentação de requisitos [VersionOne 2017]. O uso de *user stories* se popularizou por introduzir uma linguagem seminatural na especificação de requisitos tornando-os compreensíveis para o cliente final do produto, sendo um mecanismo eficiente de comunicação durante todo o processo de desenvolvimento, desde o contato com o cliente até a entrega final. Além disso, atualmente existem no mercado técnicas que fazem uso desta documentação para a produção de artefatos utilizados no desenvolvimento do sistema, esta atividade é chamada de *Living Documentation*. Um exemplo de técnica que explora a utilização de *Living Documentation* é a de *Behavior Driven Development - BDD*, que através da linguagem Gherkin descreve de forma detalhada os cenários que compõem cada uma das *user stories*, mantendo a linguagem natural e efetuando a geração de métodos que guiam a automação de testes.

Contudo, a técnica de BDD não prove a criação completa dos *scripts* de testes, sendo necessário que o testador crie de forma manual cada uma das ações que deseja automatizar, além disso a técnica de BDD não apresenta para o testador sequências de teste, sendo necessário que esse efetue a criação das sequências segundo o seu conhecimento do sistema. Desta forma, aliar a técnica de MBT com a técnica de BDD pode agregar vantagens para equipes ágeis. Para este fim, uma das alternativas é por meio da utilização de Linguagens de Domínio Específico (Domain-specific Languages - DSLs). DSLs são linguagens projetadas para atender as necessidades de um domínio específico e fazer uso de vocabulários que se aproximem deste domínio. Neste contexto, propomos neste trabalho a DSL Aquila, uma DSL que será utilizada para representação de cenários de teste através de linguagem seminatural com a adição de informações de teste e sistema, viabilizando a criação automatizada de modelos para execução de MBT.

Para a criação desta DSL efetuamos primeiramente uma revisão de literatura visando identificar quais desafios e dificuldades no processo de MBT foram descritos na literatura como os potenciais empecilhos da aplicação da técnica. Após percebermos que o principal problema está na criação dos modelos, buscamos identificar como as equipes ágeis documentam requisitos, e nesta busca percebemos que os requisitos são documentados em forma de *user stories* [VersionOne 2017]. Baseado nisso, e considerando o *background* de nosso grupo de pesquisa em teste baseado em modelos [Rodrigues et al. 2010], [Rodrigues et al. 2015], [Bernardino et al. 2016], propomos a arquitetura da DSL Aquila

e a avaliamos preliminarmente através de um exemplo de uso.

2. Trabalhos Relacionados

A fim de embasar nossa pesquisa, primeiramente efetuamos uma busca na literatura por trabalhos que apresentam métodos, abordagens ou processos para realização de MBT para teste funcional de software no contexto de desenvolvimento ágil. Posteriormente, efetuamos uma busca por DSLs para testes funcionais de software. E, por fim, buscamos agrupar todos estes conceitos, buscando trabalhos que apresentem DSLs para teste funcional de software baseado em modelos para o desenvolvimento ágil. A seguir apresentamos os trabalhos localizados, que possuem maior correlação com a DSL Aquila, proposta neste estudo.

Utting Utting *et al.* (2007) [Utting et al. 2007] apresentam em seu trabalho uma ferramenta para geração automatizada de *scripts* de testes a partir de modelos. Contudo, esta ferramenta não considera a utilização de Gherkin.

Entin *et al.* (2015) [Entin et al. 2015] apresentam a criação automatizada de modelo com base nos requisitos escritos na linguagem Gherkin. O que difere a DSL Aquila deste trabalho é que, neste trabalho é utilizado Gherkin em sua estrutura pura, e por este motivo os modelos gerados não são ricos em detalhes, são replicação literal das informações inseridas nos cenários Gherkin.

Li *et al.* (2016) [Li et al. 2016] apresentam uma ferramenta que a partir de modelos UML efetua a criação de cenários para a ferramenta Cucumber. Neste trabalho a partir dos modelos são gerados *scripts* de testes e a partir destes *scripts* são gerados os cenários para serem executados na ferramenta Cucumber. Este trabalho se assemelha com a linguagem Aquila porque ambos fazem uso de Gherkin, contudo, o processo utilizado pela DSL Aquila é o inverso, sendo gerados modelos a partir de Gherkin e não Gherkin a partir de modelos.

Elallaoui *et al.* (2015) [Elallaoui et al. 2015] se propõem em seu trabalho a gerar, através de *user stories*, diagramas de sequências. Embora este trabalho se assemelhe com a linguagem Aquila por efetuar a geração de modelos, a linguagem Aquila se propõe a gerar diagramas de atividades com um nível de detalhamento elevado, permitindo a aplicação de MBT. Neste trabalho os autores efetuam apenas a geração de diagramas de sequência o que não compreende dados minimalistas sobre o sistema que permitam a aplicação de MBT.

Elallaoui *et al.* (2016) [ELA 2016] propõem a geração de *scripts* de teste baseado em diagramas de sequência. O processo apresentado neste trabalho considera a criação de modelo de sequencias simples, migrado para um modelo de sequências com informações de testes, sendo que, posteriormente estes modelos completos que foram gerados são transformados em código de teste automatizado.

Dwarakanath *et al.* (2017) [Dwarakanath et al. 2017] apresentam uma DSL para geração de *scripts* de teste funcional baseado em Gherkin. Esta DSL tem propósito similar a DSL Aquila, contudo, não é destinada à aplicação de MBT, e com isso não permite aos projetos que a utilizam, beneficiar-se das vantagens de MBT.

Embora existam vários trabalhos que mencionem DSL, testes funcionais ou testes baseado em modelos aplicado em ambientes ágeis, no melhor de nosso conhecimento, não encontramos nenhum trabalho que aborde a utilização de DSL para modelagem de teste baseado em modelos para equipes ágeis. Visando minimizar esta lacuna, propomos

uma nova DSL, que através da geração de modelos, baseada em requisitos escritos em linguagem seminatural, permite a união dos conceitos supra citados e agrega os benefícios de MBT para o contexto de times ágeis de desenvolvimento de sistemas.

3. Aquila

Visando propor uma alternativa facilitada para a realização de MBT, e considerando um enfoque principal em equipes que atuam com metodologias ágeis de desenvolvimento de software, propomos neste trabalho a DSL Aquila. Esta DSL propõe estender a DSL Gherkin para tornar possível a criação automática de modelos para aplicação de MBT a partir dos cenários escritos em linguagem seminatural. Isto deverá ser feito através da agregação de palavras chaves que representam informações de sistema. Dentre essas informações se encontram referências às ações que precisam ser executadas no sistema para realizar um teste, por exemplo: preenchimento correto dos campos presentes na interface gráfica de determinada funcionalidade. Com a geração automatizada de modelos com base nos cenários, se torna possível a automação da criação de *scripts* de teste, bem como, facilita-se a manutenção destes artefatos, uma vez que a atualização de um determinado requisito de sistema pode ser replicada para o *script* de teste equivalente de forma automatizada.

A geração automatizada de *scripts* de teste a partir de modelos é um processo já existente e aplicado por diversos pesquisadores, entre eles: [Domingues et al. 2016][Bouquet et al. 2008][Lasalle et al. 2011]. Contudo, estes trabalhos consideram a criação manual dos modelos ou a geração de modelos a partir de código fonte, não considerando a geração de modelos a partir de cenários Gherkin. A criação manual de modelos, em processo de desenvolvimento ágil, costuma ser citada como um dos principais pontos que dificultam a implantação de MBT, principalmente devido aos ciclos curtos de entrega contínua aplicados nas metodologias ágeis [Entin et al. 2011]. Já a geração de *scripts* de teste a partir de código fonte apresenta desvantagens uma vez que, pode ocasionar a realização de testes tendenciosos, sendo que, um dos princípios do teste de software segundo [ISTQB] é realizar os testes com base nos requisitos do sistema e não com base no sistema implementado. Contudo, a desvantagem supra citada não se aplica nos casos de testes de regressão aonde visa-se garantir que um sistema continua em funcionamento após uma alteração de código [ISTQB].

Diferente do que propõem estes dois tipos de estratégias, a DSL Aquila permite a geração de *scripts* de teste baseado exclusivamente nos requisitos especificados, nos padrões estabelecidos e no conjunto de entradas e saídas esperadas para o sistema. Esta característica faz com que os artefatos de teste gerados representem aquilo que o sistema deve fazer e não aquilo que o sistema implantado faz, aumentando assim a eficiência dos testes. Nas próximas seções apresenta-se a definição do domínio, arquitetura e requisitos desta DSL.

3.1. Análise de Domínio

O desenvolvimento de uma DSL, segundo [Van Deursen et al. 2000] costumeiramente envolve três etapas distintas, sendo elas: Análise, Implementação e Utilização. Dentro da etapa de análise, a atividade principal é a definição do domínio do problema, ou seja a definição do contexto em que a DSL será empregada. Segundo [Neighbors 1984] a definição de domínio compreende entender as necessidade e requisitos de um conjunto de sistemas que possuem características similares, sendo o profissional responsável por isso, o analista de domínio ou *domain analyst*. No contexto da DSL Aquila definiu-se como

domínio específico **teste de software funcional para sistemas web em equipes de desenvolvimento ágil de software**.

3.2. Requisitos da Linguagem

Nesta seção são descritos os requisitos que formam esta DSL. Cada um dos requisitos presenta uma característica que esta DSL precisa ter para atingir o propósito para o qual foi criada. Estes requisitos dizem respeito ao domínio desta DSL (testes funcionais de software).

Requisito 1. A DSL deverá ser textual e deve ser estruturada baseada em linguagem seminatural.

O termo seminatural se trata de um subconjunto da Linguagem Natural e em razão de que deseja-se evitar a ambiguidade inerente das linguagens naturais, ou seja, o duplo significado de que determinadas palavras podem assumir em um mesmo contexto. A linguagem seminatural utilizada neste trabalho será proposta na linguagem estrangeira inglês para facilitar a aplicação da DSL em âmbito internacional. A escrita de requisitos no formato de linguagem seminatural, através da DSL Gherkin, é um padrão difundido em equipes de desenvolvimento ágil de software. Por este motivo a DSL Aquila deve seguir esta mesma estrutura de escrita visando a fácil utilização e aprendizado pelos profissionais das equipes ágeis que já estão adaptados a escrever requisitos neste formato.

Requisito 2. A DSL construída deve permitir a escrita de informações de testes funcionais de software.

O grande objetivo do desenvolvimento desta DSL é construir uma ligação entre a escrita de requisitos em linguagem seminatural e a modelagem funcional de testes para aplicação de MBT. Desta forma, é de extrema importância que a DSL proposta tenha agregada em sua estrutura a inserção de informações de testes como dados essenciais para a construção de modelos comportamentais que serão a saída da linguagem.

Requisito 3. A DSL construída deve permitir a adição de informações de sistema que permitam uma rastreabilidade entre os requisitos e o sistema desenvolvido.

Para que possa ser atendido o objetivo central do desenvolvimento desta DSL, no que diz respeito a criação de modelos comportamentais acrescidos de informações de testes e sistema que permitam a aplicação da técnica de MBT, se faz necessário incluir informações de sistema na estrutura da DSL proposta. Essas informações dizem respeito ao *front-end* das aplicações web que serão testadas. Isto é importante, uma vez que, para que se possa automatizar a criação de *scripts* de teste, é necessário que exista uma forma de identificar os campos que estão presentes na interface da aplicação e correlaciona-los com os modelos comportamentais.

Requisito 4. A DSL proposta deve permitir a realização de um *parser* para modelos comportamentais *e.g.* diagramas de atividades da UML.

Todas as informações referentes a ações de usuário e padrões de sistema que forem inseridas na DSL Aquila precisam ser refletidas automaticamente para os modelos comportamentais gerados. Uma vez que, este trabalho tem por objetivo criar modelos que permitam a geração de *scripts* automatizados de teste, a DSL deverá gerar diagramas de atividades da UML. A especificidade por este diagrama se dá como requisito por, entre os diagramas comportamentais, ser o que permite a inserção de maiores detalhes sobre as ações do usuário no sistema e os resultados esperados para cada uma delas.

Requisito 5. A DSL proposta deve permitir a geração de *scripts* de teste para aplicações web, independentemente da plataforma ou linguagem de programação na qual as aplicações foram programadas.

A DSL proposta deverá atender a um domínio específico, o da realização de testes funcionais, porém, não deverá se limitar a nenhuma plataforma ou linguagem de programação. A DSL terá seu domínio no âmbito de requisitos, gerando modelos que permitam a criação de *scripts* independentes do código fonte da aplicação. Desta forma os *scripts* gerados pela aplicação da técnica de MBT nos modelos gerados pela DSL serão classificados como *scripts* de teste de caixa preta.

3.3. Decisões de Projeto

Para atender os requisitos citados acima, se faz necessário estabelecer alguns padrões e definições arquiteturais para a DSL. Neste contexto algumas decisões estratégicas foram tomadas de forma a tornar possível atingir estes objetivos.

Decisão de Projeto 1: A DSL Aquila será projetada como extensão da DSL Gherkin.

A DSL Aquila foi projeta como uma extensão da DSL Gherkin, visando aproveitar os conhecimentos que os profissionais que atuam em equipes ágeis de desenvolvimento já possuem sobre esta DSL. Isto permite reduzir a curva de aprendizagem e facilitar a adaptabilidade para utilização da DSL Aquila. Além disso, optou-se por estender Gherkin, por ela ser compatível com testes funcionais uma vez que, descreve comportamentos. Aquila complementa Gherkin adicionando maiores informações e detalhamentos.

Decisão de Projeto 2: A DSL Aquila será composta por Palavras Chave.

A DSL Aquila tem como principal característica a inserção de informações de testes em cenários Gherkin. Desta forma, se faz necessário estabelecer um conjunto de palavras chaves que devem ser utilizadas para representar o comportamento do usuário no sistema e consequentemente serem refletidas em artefatos de testes. A Tabela 1 apresenta as palavras chaves definidas para a representação destes comportamentos. Na coluna da esquerda são representadas as palavras chaves e na coluna da direita o comportamento representado por cada uma elas.

Decisão de Projeto 3: A DSL Aquila deverá ser utilizada seguindo uma estrutura de arquivos no projeto de desenvolvimento.

Para a facilitação da geração dos testes e a melhor representação dos requisitos em modelos, estabeleceu-se uma estrutura para a criação dos arquivos que contém as funcionalidades escritas no formato Aquila. Para esta padronização levamos em consideração algumas convenções já definidas, sendo elas as seguintes: uma *feature* descreve uma funcionalidade em teste e pode agrupar diversos *scenários*, sendo um *scenario* a especificação mais detalhada de uma *feature*. Desta forma, na estrutura da DSL Aquila, todos os cenários referentes a uma *features* devem ser descritas em um único arquivo e cada *feature* representará um modelo.

Decisão de Projeto 4: A fim de estabelecer a rastreabilidade entre requisitos e modelo fica definido o seguinte:

1. A palavra GIVEN estabelece o início do modelo de uma *feature*;
2. A Palavra WHEN representa a criação de uma nova atividade no modelo;
3. A Palavra AND dentro da cláusula WHEN representa a criação de uma nova atividade no modelo;

Tabela 1. Palavras Chave Linguagem Aquila

Tag	Definição
<>	Marca caminho de tela
{}	marca a chamada para outro cenário
[]	marca nome de um campo específico
click	Indica o click em um botão
use-valid-data-on	Indica o preenchimento de um campo específico com valor válido
use-invalid-data-on	Indica o preenchimento de um campo específico com valor inválido
use valid data	indica que serão preenchidos valores válidos para todos os campos
save	indica que os dados devem ser salvos
show-sucess-message	indica que deve ser exibida uma mensagem de sucesso
show-error-message	indica que deve ser exibida uma mensagem de erro
input-data	indica que haverá entrada de dados
checked	indica seleção em checkbox
choose	indica seleção em radio
enables	indica que o sistema deverá habilitar determinado campo
disables	indica que o sistema deverá desabilitar determinado campo
successfully-saved	Indica que os dados submetidos foram salvos com sucesso
dont-fill-out	indica que um determinado campo não será preenchido
select-data	indica seleção em list
GIVEN	estado inicial do sistema
WHEN	ação
THEN	resultado de uma ação
AND	conectivo para mais de uma ação ou resultado

4. Quando utilizada a palavra USE-VALID-DATA na cláusula WHEN, deverá ser criada uma atividade no modelo para cada um dos campos presentes na interface da funcionalidade;
5. Quando utilizado USE-VALID-DATA e a aplicação possuir campos que permitem seleção, deverá ser criado no modelo um ponto de decisão onde sejam permitidos fluxos distintos de acordo com o selecionado. Por exemplo: se a aplicação possuir um *radio button* "rdbTipo" para representar Pessoas Físicas e Pessoas Jurídicas, deverá ser criado no modelo um *decision point* com dois fluxos, um cobrindo os campos referente a pessoa física e outro os campos referente a pessoa jurídica;
6. Campos que pertencem a um fluxo específico devem conter um padrão de nomenclatura com prefixo que indica o fluxo a qual pertencem;
7. Quando utilizada a palavra USE-INVALID-DATA-ON[nome do campo] deverá ser criado um ponto de decisão no modelo antes do campo para a criação do *scenario* negativo;
8. Quando utilizada a palavra DON'T-FILL-OUT[nome do campo]deverá ser criado um ponto de decisão no modelo antes do referido campo para a criação do *scenario* negativo. Caso o ponto de decisão já exista, deverá ser criado apenas uma atividade para a validação do fluxo sem preenchimento do campo;
9. Quando o caso de teste exigir a seleção de um valor específico no campo de *radio button* será utilizada a palavra CHOOSE. Neste caso será criado no modelo um fluxo específico para a opção descrita no CHOOSE. O mesmo acontece para a seleção de valores específicos em *checkbox*, através da palavra CHECKED e para seleção de valores específicos em listas, através da palavra SELECT-DATA;
10. As situações descritas na cláusula THEN, não serão convertidas em atividades nos modelos, e sim em resultados esperados para os testes.

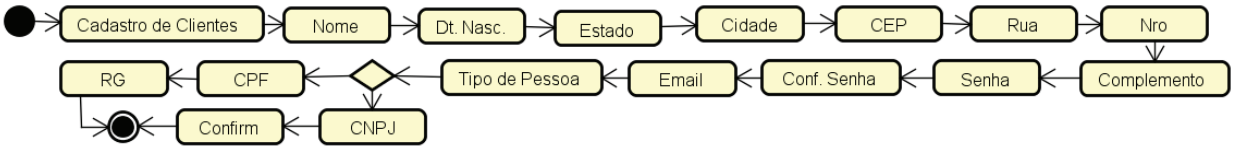

Figura 1. Fluxo Principal Cadastro de Clientes Gerado a Partir da DSL Aquila (cenário 1)

4. Exemplo de uso

A fim de realizarmos uma avaliação preliminar da DSL Aquila, realizamos a modelagem de teste funcional de um software. Este software trata-se de um cadastro de pessoas que tem um fluxo principal e oito fluxos alternativos, sendo eles; inserção de data de nascimento inválida; CEP inválido; email inválido e confirmação de senha inválida, não preenchimento de dados obrigatórios. Para exemplificação da utilização da DSL Aquila foram modelados estes fluxos supracitados e fluxo de seleção de valores em *radio button*. Na Tabela 2 descrevemos o fluxo principal, o fluxo de data de nascimento inválida e o fluxo de Salvar Dados sem Preencher Campos Obrigatórios. Os demais fluxos podem ser vistos modelados utilizando a DSL Aquila no seguinte link: https://goo.gl/Ewb1pe

Tabela 2. Cenários Aquila

1	Fluxo Principal representa o comportamento do sistema quando todos os campos são preenchidos com informações válidas.	GIVEN Customer Registration Page WHEN I **use-valid-data** AND I **click [enviar]** THEN Data is **successfully-saved**
2	Fluxo Alternativo Preenchimento inválido do Campo Data de Nascimento	GIVEN Customer Registration Page WHEN I **use-invalid-data-on[data de nascimento]** AND I **click [enviar]** THEN The data **is-not-saved**
3	Fluxo Alternativo Salvar Dados sem Preencher Campos Obrigatórios	GIVEN Customer Registration Page WHEN I **use-valid-data** AND I **choose[pessoa fisica]** AND I **dont-fill-out[CPF]** THEN The data **is-not-saved** AND The System **show-error-message**

4.1. Modelo Comportamental Gerado Após a Modelagem de Teste Utilizado a Ferramenta Aquila

O primeiro cenário escrito com a DSL Aquila, demonstra o comportamento do fluxo principal do sistema, aonde todos os campos são preenchidos com valores válidos. Este modelo é o ponto de partida para a criação de todos os demais e pode ser visualizado na Figura 1. Os demais cenários refletem fluxos alternativos, um exemplo pode ser visto na Figura 2. A Figura 3 representa o modelo geral da aplicação em teste, compreendendo todos os fluxos alternativos.

5. Conclusões e Trabalhos Futuros

Apresentamos neste trabalho a DSL Aquila e um exemplo de sua utilização. Através deste exemplo de uso podemos concluir preliminarmente que é possível efetuar a criação

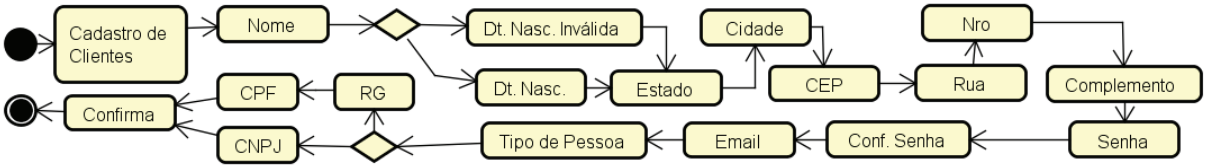

Figura 2. Fluxo Alternativo Efetuar Cadastro com Data de Nascimento Inválida (cenário e)

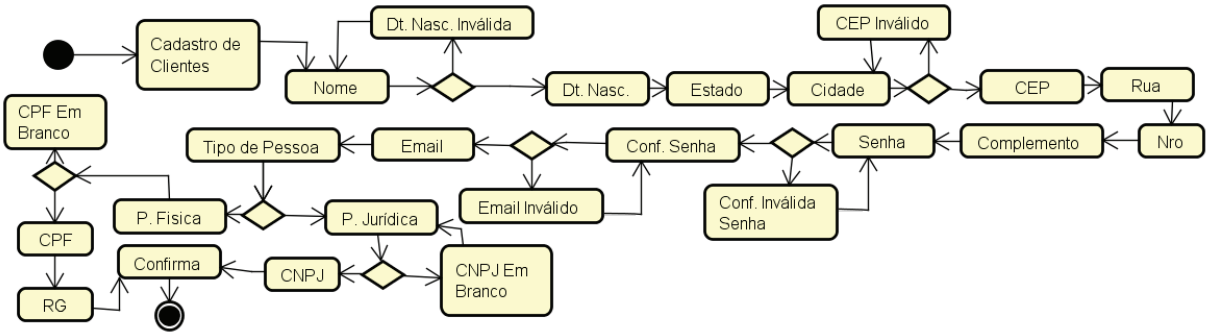

Figura 3. Modelo que Descreve a União de Todos os Cenários Contemplando Fluxo Principal e Fluxos Alternativos

de modelos para aplicação da técnica de MBT de acordo com os requisitos especificados no formato Aquila. Futuramente pretende-se expandir o exemplo de uso, efetuar a validação em um cenário real de uma empresa que trabalha com desenvolvimento de sistemas, bem como realizar reuniões com especialistas em desenvolvimento ágil de software para garantir que a DSL Aquila agrega benefícios para realização de testes em times ágeis e que a extensão realizada na linguagem Gherkin mantém as qualidades que a linguagem Gherkin possui, e não afeta a usabilidade desta linguagem.

Referências

(2016). Automated model driven testing using andromda and uml2 testing profile in scrum process. *Procedia Computer Science*, 83(Supplement C):221 – 228.

Bernardino, M., Zorzo, A. F., and Rodrigues, E. M. (2016). Canopus: A domain-specific language for modeling performance testing. In *2016 IEEE International Conference on Software Testing, Verification and Validation (ICST)*.

Bouquet, F., Grandpierre, C., Legeard, B., and Peureux, F. (2008). A test generation solution to automate software testing.

Domingues, A., Rodrigues, E. M., and Bernardino, M. (2016). Autofun: An automated model-based functional testing tool. In *Proceedings of the 1st Brazilian Symposium on Systematic and Automated Software Testing*. ACM.

Dwarakanath, A., Era, D., Priyadarshi, A., Dubash, N., and Podder, S. (2017). Accelerating test automation through a domain specific language. In *Software Testing, Verification and Validation (ICST), 2017 IEEE International Conference on*.

El-Far, I. K. and Whittaker, J. A. (2001). Model-based software testing. *Encyclopedia of Software Engineering*.

Elallaoui, M., Nafil, K., and Touahni, R. (2015). Automatic generation of uml sequence diagrams from user stories in scrum process. In *Intelligent Systems: Theories and Applications (SITA), 2015 10th International Conference on*, pages 1–6. IEEE.

Entin, V., Winder, M., Zhang, B., and Christmann, S. (2011). Combining model-based and capture-replay testing techniques of graphical user interfaces: An industrial approach. In *Proceedings of the IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops*.

Entin, V., Winder, M., Zhang, B., and Christmann, S. (2012). Introducing model-based testing in an industrial Scrum project. In *Proceedings of the 7th International Workshop on Automation of Software Test*.

Entin, V., Winder, M., Zhang, B., and Claus, A. (2015). A process to increase the model quality in the context of model-based testing. In *Software Testing, Verification and Validation Workshops (ICSTW), 2015 IEEE Eighth International Conference on*.

Fowler, M. and Highsmith, J. (2001). The agile manifesto. *Software Development*, 9(8):28–35.

ISTQB, I. S. T. Q. B. Syllabi. http://www.istqb.org/downloads/syllabi.html. acessado em 11/06/2017.

Jalalinasab, D. and Ramsin, R. (2012). Towards model-based testing patterns for enhancing agile methodologies. In *SoMeT*.

Katara, M. and Kervinen, A. (2006). Making model-based testing more agile: a use case driven approach. In *Proceedings of the Haifa Verification Conference*.

Lasalle, J., Peureux, F., and Fondement, F. (2011). Development of an automated mbt toolchain from uml/sysml models. *Innovations in Systems and Software Engineering*, 7(4).

Li, N., Escalona, A., and Kamal, T. (2016). Skyfire: Model-based testing with cucumber. In *Software Testing, Verification and Validation (ICST), 2016 IEEE International Conference on*.

Neighbors, J. M. (1984). The draco approach to constructing software from reusable components. *IEEE Transactions on Software Engineering*, SE-10(5):564–574.

Rodrigues, E. M., de Oliveira, F. M., Costa, L. T., Bernardino, M., Zorzo, A. F., Souza, S. d. R. S., and Saad, R. (2015). An empirical comparison of model-based and capture and replay approaches for performance testing. *Empirical Software Engineering*, 20(6).

Rodrigues, E. M., Viccari, L. D., Zorzo, A. F., and Gimenes, I. (2010). PLeTs - Test automation using software product lines and model based testing. In *Proceedings of the 22nd International Conference on Software Engineering and Knowledge Engineering*.

Utting, M., Perrone, G., Winchester, J., Thompson, S., Yang, R., and Douangsavanh, P. (2007). The modeljunit model-based testing tool. acessado em 11/06/2017.

Van Deursen, A., Klint, P., Visser, J., et al. (2000). Domain-specific languages: An annotated bibliography. *Sigplan Notices*, 35(6):26–36.

VersionOne (2017). State of agile report. http://stateofagile.versionone.com/. acessado em 11/06/2017.