PUC

ISSN 0103-9741

Monografias em Ciência da Computação
n° 21/10

Uma Nova Linguagem de Modelagem para Testes de Software

Andrew Diniz da Costa
Viviane Torres da Silva
Carlos José Pereira de Lucena

Departamento de Informática

PONTIFÍCIA UNIVERSIDADE CATÓLICA DO RIO DE JANEIRO
RUA MARQUÊS DE SÃO VICENTE, 225 - CEP 22451-900
RIO DE JANEIRO - BRASIL

Monografias em Ciência da Computação, No. 21/10 ISSN: 0103-9741
Editor: Prof. Carlos José Pereira de Lucena Dezembro, 2010

Uma Nova Linguagem de Modelagem para Testes de Software

Andrew Diniz da Costa, Viviane Torres da Silva[1],
Carlos José Pereira de Lucena

[1]Departamento de Informática – Universidade Federal Fluminense (UFF)

acosta@inf.puc-rio.br, viviane.silva@ic.uff.br, lucena@inf.puc-rio.br

Abstract. Nowadays, to document software tests is a big concern of projects that perform the creation and execution of tests at large-scale systems. One of the approaches used to perform such documentation is UML Testing Profile (UTP). This approach allows the modeling of several test concepts. However, from our experience coordinating test teams in different large-scale systems, we identified important test concepts that are not modeled by UTP. Based on this idea, the document presents a new test modeling language. Such language will represent these identified concepts from the extension of the UTP. Besides this, we intend to define new well-formed rules to this extension, beyond creating a transformer that will allow the generation of test cases from UTP diagrams. These tests will be executed from a framework that performs self-test in adaptive systems based on software agents.

Keywords: Modeling, Software Test, UML Testing Profile, Autonomic Computing.

Resumo. Documentar testes de software é uma das grandes preocupações em projetos que realizam a criação e execução de testes em sistemas de larga escala. Uma das abordagens utilizadas para realizar tal documentação é o uso de UML Testing Profile (UTP), que permite a modelagem de diversos conceitos de teste. No entanto, a partir da nossa experiência coordenando equipes de testes em diferentes sistemas de larga escala, identificou-se importantes conceitos de teste que ainda não são modelados pela UTP. Seguindo essa linha, o documento apresenta como proposta uma nova linguagem de modelagem de teste. Essa linguagem irá representar esses conceitos identificados a partir da extensão da UTP. Além disso, pretende-se definir novas regras de boa formação para essa extensão, além de criar um transformador que permita a partir de diagramas UTP, gerar casos de teste para um framework responsável por realizar autoteste em sistemas adaptativos baseados em agentes de software.

Palavras-chave: Modelagem, Teste de Software, UML Testing Profile, Computação Autonômica.

Responsável por publicações:

Rosane Teles Lins Castilho
Assessoria de Biblioteca, Documentação e Informação
PUC-Rio Departamento de Informática
Rua Marquês de São Vicente, 225 - Gávea
22451-900 Rio de Janeiro RJ Brasil
Tel. +55 21 3527-1516 Fax: +55 21 3527-1530
E-mail: bib-di@inf.puc-rio.br
Web site: http://bib-di.inf.puc-rio.br/techreports/

Sumário

1 Introdução 1
1.1 Proposta 1
1.2 Contribuições 2
1.3 Organização do Documento 2
2 Novos Conceitos de Teste Identificados 3
2.1 Procedimentos Empíricos Seguidos 3
2.2 Descrição dos Conceitos Identificados 4
2.3 Conceitos Propostos por Participantes 5
3 Modelando Conceitos Identificados 6
3.1 Procedimentos para Estender UML Testing Profile 6
3.2 Abordagem Inicial da Extensão da UTP 8
3.3 Definindo Restrições do Meta-Modelo 10
3.4 Definindo Permissões 11
4 Realizando Auto-Teste em Sistemas Adaptativos 12
5 Trabalhos Relacionados 13
5.1.1 Ferramentas de Gerenciamento 14
5.2 Linguagens (Modelagem/Programação) 14
5.3 Trabalhos de Auto-Teste 15
5.4 Geração de Dados 16
6 Cronograma 16
7 Considerações Finais 17
Referências 17

1 Introdução

Com o passar do tempo os sistemas desenvolvidos estão cada vez mais complexos, exigindo um aumento na qualidade dos produtos gerados. Conseqüentemente, um dos impactos dessa exigência foi a criação de novas tecnologias que ajudassem no desenvolvimento dos sistemas. (OMG UML, 2010), (RFT, 2010), (RPT, 2010), (JUnit, 2010), (Fest, 2010) e (TTCN-3, 2010) são algumas das abordagens amplamente usadas no mercado para execução e design de casos de teste, assim como novos paradigmas de desenvolvimento, tal como arquiteturas dirigidas a modelos (MDA).

Baseado na área de testes de software e no conceito de MDA, um conjunto de instituições (Ericsson, Fraunhoufer/FOKUS, IBM/Rational, Motorola, Telelogic, University of Lübeck) decidiu trabalhar em cooperação visando criar um profile para a UML, chamado de UML Testing Profile – UTP (OMG UTP, 2010). Esse profile foi criado com o intuito de modelar importantes conceitos de teste não presentes na UML 2 (OMG UML, 2010) e que auxiliasse no projeto e no desenvolvimento de testes. Após dois anos de trabalho, a especificação da UTP começou a ser adotada pela OMG e tornou-se um padrão de modelagem para a área de testes de software.

1.1 Proposta

Essa tese propõe uma nova linguagem de modelagem de testes responsável por representar importantes conceitos de teste não modelados por famosas abordagens de teste, como, por exemplo, (OMG UTP, 2010), (UTML, 2010), (Agedis, 2010) e (TTCN-3, 2010). Alguns desses conceitos já puderam ser identificados a partir de nossa experiência em coordenar, criar e executar testes em diferentes sistemas industrias de larga escala. Os resultados encontrados são apresentados em (b- Costa el al., 2010). Uma cópia desse trabalho está no Apêndice 2.

Como conseqüência, decidimos estender a UTP para representar esses novos conceitos de teste. Os motivos que nos levaram a seguir essa abordagem foram os seguintes: (i) a UTP não representa os conceitos identificados, (ii) ela já propõe a modelagem de outros conceitos importantes de teste que foram identificados por diferentes instituições com larga experiência na área de teste, (iii) a UTP é considerada um padrão da OMG, (iv) ela é amplamente aceita e utilizada pela comunidade de teste e modelos, e (v) um dos requisitos de boa parte dos projetos em que trabalhamos, é a criação de diagramas UML.

Ao realizar a extensão da UTP, pretendemos propor restrições que permitam orientar a construção adequada de modelos (regras de boa formação), assim como restrições de permissão referente a quais ações cada membro de um time de teste pode fazer (ex: executar, atualizar, criar ou deletar testes).

Outro ponto que pretendemos tratar é referente a geração de código a partir de diagramas UTP que incluem os novos conceitos de teste. Como uma das grandes deficiências no mercado é a ausência de frameworks que auxiliem na execução de auto-testes em sistemas auto-adaptativos, pretendemos criar um transformador que permita gerar testes a partir de diagramas UTP e que utilizem um novo framework de auto-teste baseado em agentes de software (Wooldridge e Jennings, 1998). (a- Costa el al, 2010) e (c- Costa el al, 2010) apresentam versões iniciais desse framework. Uma cópia desses artigos, está disponível nos Apêndices 1 e 3, respectivamente.

1.2 Contribuições

A seguir, são listadas as contribuições que esperamos alcançar com o trabalho:

- Identificação de um conjunto de conceitos de teste que devem ser documentados a partir da experiência em coordenar, criar e executar testes em diferentes sistemas industriais de larga escala.
- Avaliação de quais conceitos de teste identificados não estão modelados em famosas abordagens propostas na literatura, assim como, UTP, Agedis e TTCN-3.
- Nova linguagem de modelagem criada a partir da extensão do meta-modelo da UTP. Tal linguagem irá representar os conceitos de teste identificados. Além disso, serão definidas regras de boa formação a partir da linguagem *Object Constraint Language* – OCL (OMG OCL, 2010) com o objetivo de garantir que os modelos gerados sejam instancias válidas do meta-modelo.
- Uso da linguagem de modelagem SecureUML (Lodderstedt et al., 2002). Para o controle de acesso baseado em papéis para o domínio de testes de software.
- Criação de um novo framework que visa auto-testar aplicações auto-adaptativas que sejam baseadas em agentes de software.
- Criação de um transformador que permita a partir de diagramas da nova linguagem de modelagem realizar a geração de testes que sejam executados pelo novo framework de auto-teste.

1.3 Organização do Documento

Primeiramente (seção 2) apresentamos os principais resultados obtidos a partir de um estudo inquisitivo que permitiu identificar importantes conceitos de teste não modelados por famosas abordagens de teste. Detalhes desse estudo são descritos em (b- Costa el al., 2010). Na seção 3 descrevemos a idéia geral de como pretendemos estender a UML Testing Profile e quais restrições em OCL e SecureML pretendemos escrever. Na seção 4, a idéia geral do novo framework de auto-teste é apresentada. Na seção 5 alguns trabalhos relacionados são mencionados, enquanto que na seção 6 é apresentado o cronograma do trabalho. Finalmente na seção 7 são realizadas as considerações finais.

2 Novos Conceitos de Teste Identificados

A partir da coordenação, criação e execução de casos de teste em diversos sistemas, pudemos identificar importantes conceitos de teste não representados em linguagens de modelagem propostas na literatura. Visando validar a importância destes conceitos e identificar novos conceitos, realizou-se um estudo inquisitivo (Lethbridge et al., 2000) composto por um conjunto de entrevistas e a aplicação de um questionário.

Análises inquisitivas são recomendadas para casos que necessitem de uma investigação exploratória sem precisar de hipóteses bem definidas. Por outro lado, experimentos controlados impõe: (i) maior custo e (ii) mais tempo adicional dos participantes envolvidos no estudo. Logo, essas dificuldades nos motivaram a aplicar um estudo inquisitivo. Nessa seção inicialmente é apresentada a visão geral dos procedimentos seguidos para encontrar os conceitos de teste, seguida pela descrição dos conceitos identificados. Caso deseje ler mais detalhes o estudo realizado, veja o artigo (b- Costa et al., 2010) que também está no Apêndice 2.

2.1 Procedimentos Empíricos Seguidos

O estudo inquisitivo teve a duração de três anos e foi dividido em cinco etapas principais (ver Figura 1). A primeira etapa foi responsável por coordenar a criação e execução de testes em três projetos de larga escala. Essa etapa durou dois anos e para cada projeto foram realizados testes em sete versões (em média). Já na segunda etapa entrevistamos os líderes dos diferentes projetos. Tais entrevistas nos permitiu avaliar quais conceitos de teste os líderes achavam imprescindíveis de serem documentados e a importância de cada conceito nos projetos.

Na terceira etapa, após o levantamento dos conceitos de teste, procuramos famosas linguagens de modelagem e ferramentas de gerenciamento que conseguissem representá-los. Ao finalizar essa análise, concluímos que diversos conceitos identificados não eram representados nessas abordagens.

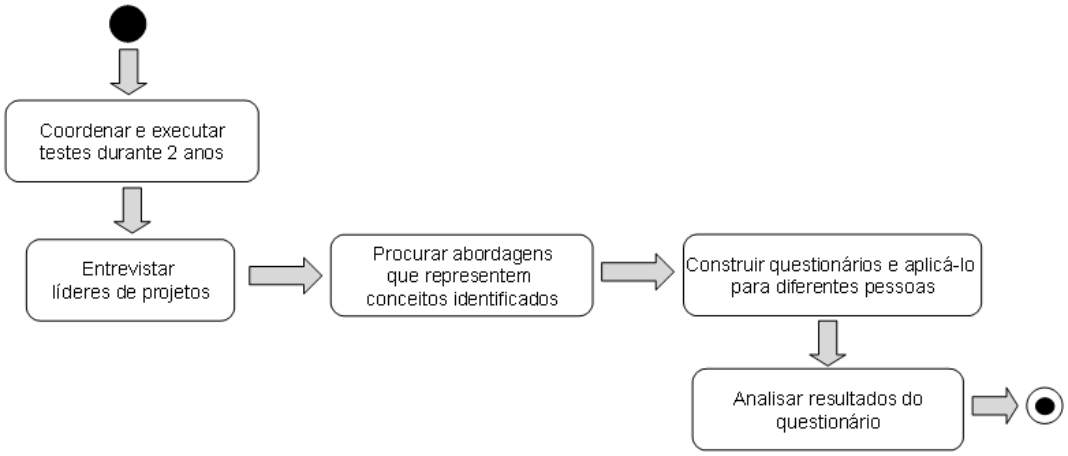

Figura 1. Etapas seguidas no estudo inquisitivo.

A quarta etapa do nosso processo visou aplicar um questionário para quatorze participantes com diferentes níveis de conhecimento e experiência na área de testes. O objetivo central do questionário foi verificar a importância dos conceitos identificados a partir da opinião de outras pessoas. Na tabela 1 é apresentado o perfil dos participantes do questionário.

Por fim, na última etapa, analisamos as respostas do questionário aplicado. Após essa análise, concluimos que, segundo os participantes, os conceitos identificados foram considerados importantes e devem ser documentados. Na seção a seguir, esses conceitos são descritos em detalhe.

Tabela 1. Perfil dos participantes que responderam o questionário aplicado.

Participantes e seus Papéis	Nível acadêmico	Anos de experiência	Descrição
(5 participantes) 1 líder de projeto 2 desenvolvedores sênior 2 desenvolvedor júnior	2 doutorandos e 1 mestre em Engenharia de Software (ES). 1 graduado e 1 aluno de graduação em Ciência da Computação (CC).	> 3 anos	Grande conhecimento sobre conceitos de teste e bibliotecas, como, por exemplo, ferramentas Rational, JUnit, DBUnit, etc.
(6 participantes) 2 admins. de BD 4 desenvolvedores sênior	1 doutorando em ES, 3 mestres em ES, 2 graduados em CC.	1 .. 3 anos	Possui experiência com testes unitários, funcionais e de performance.
(3 participantes) 2 desenvolvedores sênior 1 desenvolvedor júnior	2 alunos de graduação em CC. 1 aluno de graduação em ES.	< 1 ano	Não possui muita experiência com testes. No entanto, sabe os conceitos básicos de teste, como, por exemplo, a existência de diferentes tipos de teste.

2.2 Descrição dos Conceitos Identificados

A seguir, os conceitos de teste identificados e considerados importantes são apresentados.

Documentar a versão do sistema a qual cada caso de teste está associado: Identificar a versão do sistema em que cada caso de teste está atualizado é uma informação de extrema importância. Em algumas situações, a versão mais nova de um conjunto de casos de teste pode ser usada exclusivamente para a última versão do sistema testado. O mesmo pode acontecer para versões antigas, isto é, casos de teste atualizados para alguma versão anterior, podem ser executados somente em tal versão. Logo, documentar a versão do sistema que um caso de teste está atualizado é uma tarefa importante que sempre deve ser realizada. Além disso, identificou-se a necessidade de definir e relacionar classificações de testes com os casos de teste criados e atualizados para cada versão de um sistema. Sua representação visa ajudar a identificar a importância dos testes, assim como auxiliar no planejamento dos testes. Um exemplo de classificação seria "testes de regressão", que dependendo da versão do sistema, o grupo de casos de teste relacionado pode sofrer mudanças.

Identificar casos de teste obrigatórios e opcionais: Segundo os líderes de teste, os times possuem dificuldade em identificar quais testes são obrigatórios e opcionais para

ser testados em cada versão de um sistema. Como alguns testes não possuem muita prioridade em sua execução, eles podem ser executados somente se houver tempo hábil, isto é, são considerados testes opcionais. Já os testes obrigatórios, são aqueles que devem ser executados, independente das dificuldades a serem apresentadas.

Documentar dependências entre casos de teste: Para que um caso de teste consiga realizar sua execução sem problema, ele pode depender de outro teste. Conseqüentemente, diferentes tipos de dependência podem existir, como, por exemplo, o recebimento de algum dado gerado por outro teste, o uso de uma configuração no ambiente realizada por outro teste, etc. Portanto, entender quais dependências existem entre os testes torna-se fundamental para realizar as execuções sem problema e entender a complexidade do teste.

Representar relações conceituais entre casos de teste: Diferentes casos de teste podem estar conceitualmente relacionados, como, por exemplo, a um mesmo caso de uso ou quem sabe a um componente do sistema. A partir disso, podemos concluir que esses testes têm grandes chances de testar as mesmas partes do sistema. Assim, realizar a documentação dessas relações nos ajudaria a melhor entender a cobertura provida pelos testes criados, sem que haja duplicação na criação dos testes. A partir do estudo inquisitivo realizado, verificamos que os participantes consideraram útil conseguir visualizar essas relações para ajudar na identificação de quais testes são responsáveis por validar qual requisito ou funcionalidade.

Documentar casos de teste automatizados e manuais: Testes podem ser executados tanto manualmente como automaticamente. Assim, dependendo da quantidade de testes criados, a identificação de quais testes são manuais e automatizados torna-se uma tarefa custosa. Além disso, essa identificação guia o testador a utilizar a ferramenta adequada para executar cada caso de teste.

Documentar *suites* criadas: *Suite* é um componente responsável por executar um conjunto de casos de teste automaticamente em uma ordem específica. Dependendo do sistema testado, uma grande quantidade de *suites* pode ser definida. Assim, documentá-las ajuda a identificar quais testes são executados por *suite*, assim como a ordem de execução definida em cada uma delas.

Identificar tipos de teste: Existem diversos tipos de teste já propostos (Graham et al., 2008), como, por exemplo, testes funcionais, unitários, integração, segurança, carga, etc. Logo, identificar os tipos de teste ajuda a entender a finalidade de cada teste. Além disso, essa informação ajuda na delegação de tarefas, já que em certas equipes de teste, podem haver especialistas em específicos tipos de teste.

2.3 Conceitos Propostos por Participantes

Além de solicitar a percepção dos participantes sobre a importância dos conceitos identificados, o questionário também solicitou a identificação de outros conceitos não mencionados. A seguir, esses conceitos de teste são listados com os comentários mais significativos dos participantes:

Documentar papéis desempenhados: Dois dos participantes mencionaram que dependendo do projeto, diferentes papéis podem ser desempenhados por indivíduos em um mesmo time, como, por exemplo, especialista em testes funcionais, testador de testes de performance, coordenador, etc. Assim, documentar os papéis e as permissões de cada papel ajuda a identificar quem irá trabalhar ou quem trabalhou em algum caso de

teste. Os participantes que destacaram esse conceito possuem grande experiência na área de teste.

Identificar em mais detalhe a prioridade da execução dos testes: Esse conceito foi recomendado por duas pessoas que responderam o questionário. Segundo essas pessoas, dependendo do tempo disponível para testar um projeto, definir em mais detalhe a prioridade dos testes pode ser útil, especialmente quando o conjunto de testes obrigatórios é grande e o tempo para executá-los é pequeno. Logo, tal informação nos ajudaria a definir ordens de execução dos casos de teste.

Relacionar cada caso de teste com artefatos de teste: Três participantes informaram que documentar quais artefatos (ex: classes, componentes, etc) são testados por cada caso de teste é uma tarefa importante devido duas razões principais: (i) quando há a necessidade de identificar o artefato com problema, e (ii) quando torna-se necessário conhecer a cobertura dos testes. Perceba, que o principal foco desse conceito é permitir a rastreabilidade dos artefatos testados a partir da execução dos testes.

3 Modelando Conceitos Identificados

A idéia central dessa seção é descrever como pretendemos modelar os conceitos de teste identificados na seção 2. Inicialmente apresentamos os procedimentos que pretendemos seguir para propor e validar a nova linguagem de modelagem de teste que será representada a partir da extensão na UML Testing Profile - UTP. Em seguida, são apresentados alguns *insights* da abordagem proposta e novas regras de boa formação que serão escritas em OCL. Por fim, são apresentadas restrições escritas em SecureUML que permitam controlar permissões baseadas em papéis desempenhados por membros de diferentes equipes de teste.

3.1 Procedimentos para Estender UML Testing Profile

Como a UTP tem sido amplamente aceita pela comunidade da área de modelos e testes, decidimos avaliar se os conceitos identificados (ver seção 2) estavam representados em tal abordagem. No final da análise concluímos que boa parte dos conceitos não estavam presentes, surgindo a necessidade de incluí-los. A Figura 2 ilustra as etapas que pretendemos seguir para propor e avaliar a extensão na UTP.

Na primeira etapa (já realizada) verificou-se quais conceitos de teste não estavam representados na UTP. Os únicos conceitos que a UTP atualmente representa são os seguintes: (i) dependência entre casos de teste e (ii) suítes. Apesar disso, há limitações em tais representações. No conceito de dependência, a UTP não informa a semântica da dependência presente entre casos de teste. Tal informação é de grande importância para entender a relação entre os testes, além dela poder ser usada como base para o planejamento dos testes. Já o conceito de suítes é representado pelo estereótipo <<*TestContext*>>, responsável por definir uma classe com um ou mais casos de teste e que estão associados a um determinado contexto. Apesar disso, em equipes de teste, há a necessidade de distinguir a classe em que foi desenvolvido um ou mais testes, da entidade responsável por executar um conjunto de casos de tese em uma ordem específica. Usando o estereótipo <<*TestContext*>>, tal distinção torna-se difícil.

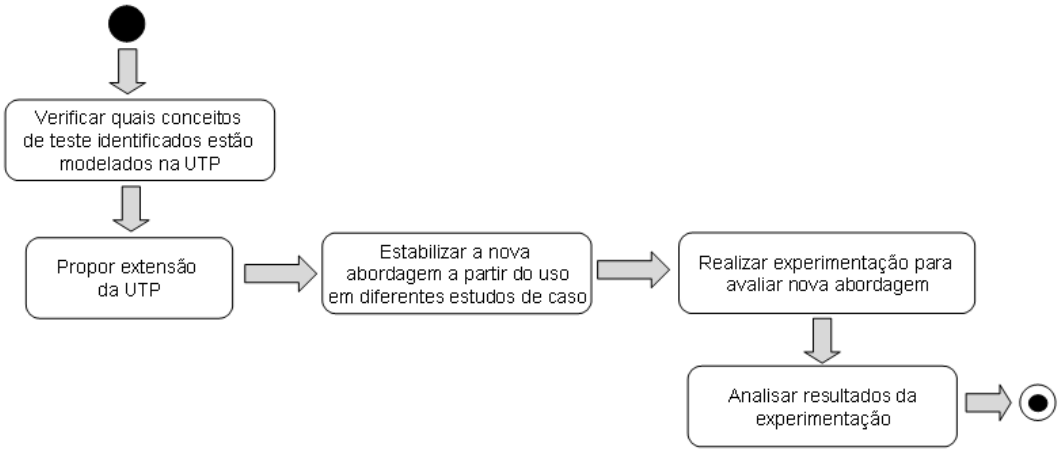

Figura 2. Etapas a serem seguidas para avaliar extensão da UTP.

Na segunda etapa, após avaliar as limitações da UTP, iremos propor uma abordagem que a estenda incluindo os novos conceitos de teste. Na seção 3.2 é apresentada uma idéia inicial de como pretendemos realizar tal extensão.

Na terceira etapa pretende-se avaliar a nova abordagem em diferentes estudos de caso, a fim de identificar novos conceitos de teste, e gerar uma versão estável do novo meta-modelo da UTP. Durante esse processo também serão realizadas discussões com especialistas da área de modelagem. Essas discussões terão como objetivo avaliar a melhor maneira de modelar os novos conceitos de teste na UTP.

Na quarta etapa será realizada uma experimentação envolvendo um sistema de larga escala e pessoas que possuem experiência na área de testes. A idéia central dessa experimentação será avaliar as seguintes hipóteses:

1. A nova abordagem melhora a comunicação entre membros das equipes de teste em diferentes projetos.
2. A curva de aprendizado da nova abordagem é menor do que o aprendizado de ferramentas de gerenciamento de teste.
3. A nova abordagem permite aos novos integrantes de equipes terem uma visão relativamente rápida da cobertura de testes provida para qualquer sistema.
4. A nova abordagem permite que novos integrantes sejam inseridos de forma mais rápida em equipes de teste em vez de utilizar famosas abordagens de modelagem e ferramentas de gerenciamento de teste.
5. A nova abordagem apresenta de forma mais clara dependências entre testes do que famosas abordagens de modelagem e ferramentas de gerenciamento de teste.
6. A nova abordagem torna mais completa a documentação dos casos de teste em vez de utilizar famosas abordagens de modelagem e ferramentas de gerenciamento de teste.

Os resultados da experimentação serão obtidos e analisados na quinta e última etapa.

3.2 Abordagem Inicial da Extensão da UTP

Quando trabalhamos com *profiles* em UML novos estereótipos e atributos podem ser definidos. Seguindo essa idéia como base, decidimos criar novos elementos que conseguissem representar os conceitos de teste, apresentados na seção 2, na UTP.

A Figura 3 ilustra um exemplo de como poderiam ser representados alguns dos novos conceitos na UTP. Como mencionado na seção 3.1, o estereótipo <<*TestContext*>> deve ser usado para representar uma classe que possui um ou mais casos de teste implementados. Visando distinguir esse conceito da semântica de entidades responsáveis por definir a ordem de execução de um conjunto de casos de teste, definimos o estereótipo <<*OrderSuite*>>, representado na classe *OrderSuite1*. Além disso, as entidades (ex: classes) que representam esse tipo de suíte, também devem ter um método *executionOrder()* responsável por definir quais casos de teste devem ser executados em qual ordem, assim como ilustrado na Figura 3.

Já a classe *TestComponent1*, também da Figura 3, possui um conjunto de novos estereótipos e atributos que representam diversos conceitos. Inicialmente, perceba que cada método considerado o ponto de partida de um caso de teste, a UTP identifica como um caso de teste representado pelo estereótipo <<*TestCase*>>. Logo, aqueles métodos que não possuem esse estereótipo são classificados como métodos auxiliares usados pelos casos de teste.

Visando informar qual a versão do sistema (SUT – *System Under Test*) que um especifico *TestContext* deverá ser atualizado para executar seus testes, definimos um atributo chamado *systemVersion*. Na classe *TestComponent1* é ilustrado um exemplo.

Um importante conceito de teste não representado na UTP é a distinção dos casos de teste obrigatórios e opcionais para uma específica versão de um sistema (representada pelo atributo *systemVersion*). Para realizar essa diferenciação, dois novos estereótipos foram criados, <<*Mandatory*>> e <<*Optional*>>. O primeiro informa que um caso de teste é obrigatório, enquanto que o segundo representa um caso de teste opcional.

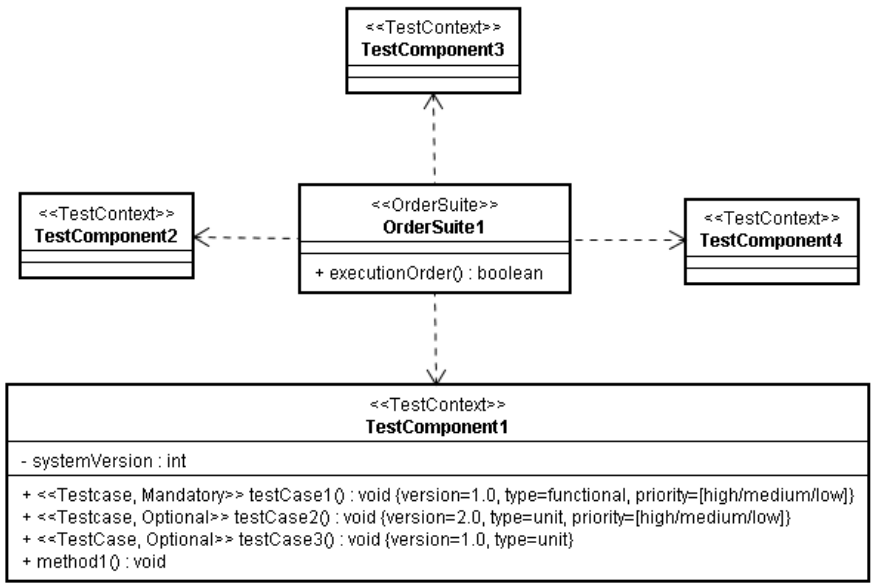

Figura 3. Exemplo de extensão da UTP

Além disso, outras três informações foram representadas para cada caso de teste: versão do sistema que ele está atualizado, o tipo de teste relacionado (ex: testes funcionais, unitários, segurança, etc), e a prioridade mais detalhada de cada caso de teste, já que uma grande quantidade de testes podem ser considerados obrigatórios ou opcionais. A proposta inicial para representar esses três conceitos é a partir do uso dos seguintes atributos: *version*, *type* e *priority*, respectivamente.

UML 2.0 provê o conceito de dependências entre classes e que é reusado pela UTP. Apesar disso, a UTP não oferece uma forma de identificar a semântica das dependências modeladas. Visando tratar essa preocupação, pretendemos propor um conjunto de estereótipos que possam ser usados em relacionamentos de dependências entre *TestContexts*, assim como ilustrado na Figura 4. A seguir, há a explicação de alguns estereótipos propostos:

- **artifact_created**: Quando um caso de teste precisa usar um artefato (ex: arquivo, componente, entidade, etc.) que deve ser criado previamente por outro teste.
- **artifact_removed**: Um teste depende da remoção de algum artefato no sistema.
- **artifact_modified**: Teste depende de alguma mudança de um artefato (ex: mudança de nome, localização, etc.).
- **set_artifact_created**: Dependência referente a um conjunto de artefatos a serem criados.
- **set_artifact_removed**: Dependência referente a um conjunto de artefatos que devem ser removidos.
- **set_artifact_modified**: Teste depende da alteração de um conjunto de artefatos.
- **environment_modification**: Caso um teste dependa de um conjunto de mudanças, como, por exemplo, criações, remoções e modificações no ambiente a ser testado, esse estereótipo pode ser usado.

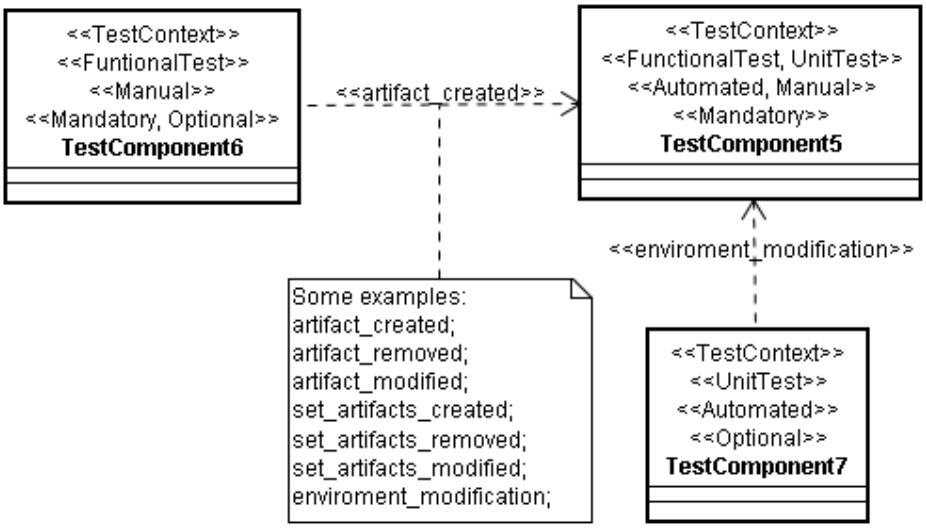

Figura 4. Representando dependências.

Além dos tipos de dependências mencionados, a Figura 4 ilustra uma modelagem em nível mais abstrato, isto é, permite aos stakeholders do sistema terem uma visão dos testes criados sem que haja a necessidade de modelar todos os conceitos ilustrados na Figura 3. Perceba que a Figura 4 apresenta três *TestContexts*. Em cada *TestContext*

são informados: (i) os tipos de teste desenvolvidos (ex: <<*FuntionalTest*>> para testes funcionais, <<*UnitTest*>> para testes unitários), (ii) se os testes são manuais ou automatizados (<<*Manual*>>, <<*Automated*>>), e (iii) se os testes são considerados obrigatórios ou opcionais (<<*Mandatory*>>, <<*Optional*>>). Na classe *TestComponent7*, apenas testes unitários, automatizados e opcionais foram desenvolvidos. Por outro lado, a classe *TestComponent5* informa que há testes obrigatórios manuais e automatizados que são do tipo funcional e unitário.

No artigo (b- Costa et al., 2010) mencionamos um exemplo de projeto em que testes precisam ser classificados, como, novos, regressão ou impactados. Os testes novos são aqueles criados devido os novos requisitos definidos em uma versão do sistema. Testes de regressão são aqueles que devem ser executados em toda versão, enquanto que testes impactados são testes já criados em alguma versão anterior e que devem ser atualizados devido os novos requisitos. Baseado nessa idéia, pretendemos usar o conceito de pacotes para agrupar classes de forma conceitual. Decidimos adotar essa abordagem, pois o meta-modelo da UML 2 (OMG UML 2.0, 2010) define pacotes como agregações de elementos de diferentes tipos. Como essa definição é genérica, ela nos permite definir estereótipos para aplicar mais semântica nessas agregações. Assim, definimos o estereótipo <<*Classification*>>, responsável por definir classificações. Essas classificações podem ser usadas tanto para classificações de teste, como para outras classificações definidas por diferentes stakeholders.

Na Figura 5 são representadas as três classificações mencionadas: novos, regressão e impactados. Além dessa abordagem, decidimos propor o estereótipo <<*Development*>> para indicar os pacotes que representam a real estrutura usada para organizar classes desenvolvidas. Em projetos desenvolvidos pelo nosso grupo, geralmente adotamos a idéia de organizar as classes de teste por caso de uso, assim como ilustrado na Figura 5.

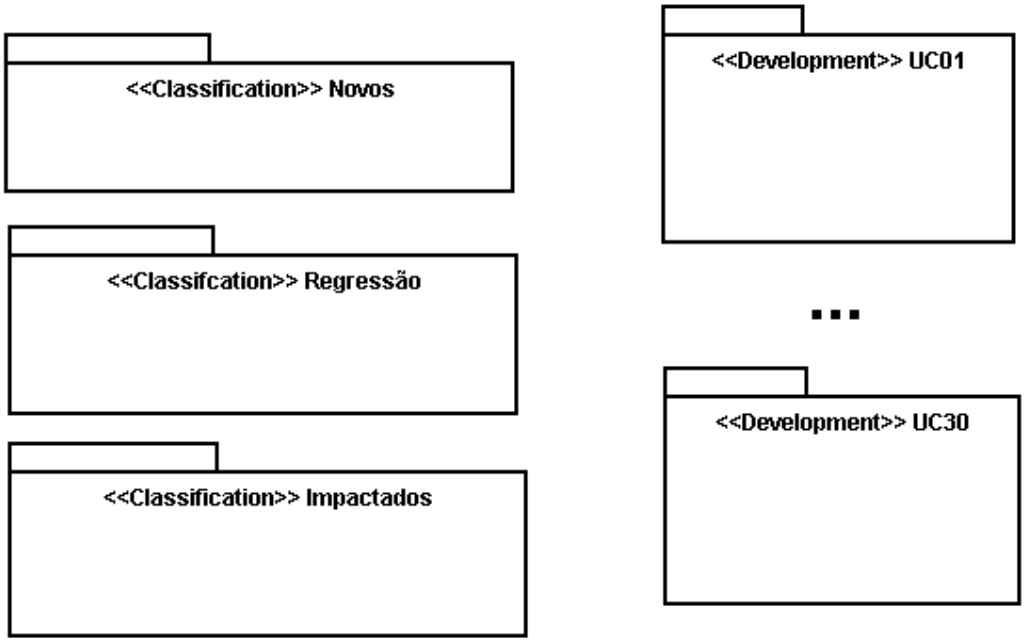

Figura 5. Modelando classificações de testes.

3.3 Definindo Restrições do Meta-Modelo

Quando abordamos a área de modelos, uma preocupação constante são as regras de boa formação, que definem regras para meta-modelos. Como estamos propondo a ex-

tensão de um *Profile*, regras também deverão ser definidas. Assim, pretendemos utilizar a linguagem *Object Constraint Language* – OCL para escrevê-las. OCL é uma linguagem declarativa criada pela IBM, responsável por descrever regras para modelos UML. Decidimos utilizá-la, pois além de ser voltada para modelos UML, ela também é considerada um padrão reconhecido pela OMG. Visando desenvolver essas restrições, pretendemos utilizar a ferramenta Eye OCL Software - EOS proposta por (Dios et al., 2008).

A seguir, há a listagem de algumas restrições que podem ser criadas a partir da extensão da UTP:

i. Não devem ser modeladas suítes sem ao menos estarem relacionadas com ao menos um *TestContext* que possua um caso de teste implementado.

ii. Os testes considerados obrigatórios para uma versão, devem estar atualizados.

iii. Estereótipos das classes devem estar consistentes com estereótipos dos métodos.

iv. Todos os testes associados a um suíte devem ser automatizados.

v. Todos os testes devem estar relacionados a uma classificação.

Além disso, abaixo há algumas consultas (*queries*) que a linguagem OCL também nos permite criar:

i. Verificar quantos *TestContext* estão relacionados a uma suíte. Tal informação torna-se importante para avaliar se a execução de um suíte poderá demorar muito.

ii. Verificar quantos testes obrigatórios e opcionais continuam desatualizados.

3.4 Definindo Permissões

Em projetos industriais que possuam grandes equipes de teste, torna-se comum definir papéis nesses times, como, por exemplo, especialista em testes funcionais, especialista em testes de performance, coordenador, etc. Conseqüentemente, diversas regras de permissão podem ser definidas e aplicadas para tais times. Baseado nessa idéia, SecureUML foi proposta. SecureUML é uma linguagem de modelagem voltada para UML para o desenvolvimento dirigido a modelos baseado em segurança e sistemas distribuídos. Tal abordagem baseia-se no controle de acesso baseado em papel, onde permissões são definidas visando representar quais ações as pessoas que desempenham papéis possuem acesso.

A seguir, são listadas algumas restrições de permissão que pretendemos definir:

i. Pessoas, que estejam desempenhando um papel de especialista de teste funcional, devem ter permissão para atualizar, deletar e executar testes funcionais.

ii. O coordenador de teste pode ter acesso completo a todos os testes desenvolvidos pela sua equipe.

iii. O cliente responsável por homologar o sistema pode ter permissão para atualizar os dados de entrada e as assertivas de saída de um conjunto de teste. Por outro lado, ele não possui permissão para atualizar o código desses testes.

Como a SecureUML não foi criada visando a área de testes, realizaremos uma análise para ver se a linguagem consegue representar restrições similares as mencionadas acima. Caso não seja possível, iremos propor extensões na linguagem.

4 Realizando Auto-Teste em Sistemas Adaptativos

O desenvolvimento de sistemas complexos, capazes de auto-adaptar seus comportamentos quando necessário, está cada vez mais comum. Relacionado a esse contexto, há o paradigma de sistemas multi-agentes (Wooldridge e Jennings, 1998), que tem sido especialmente usado quando são representadas entidades distribuídas, autônomas e pró-ativas.

Seguindo essa linha, alguns trabalhos foram propostos para ajudar na construção de sistemas auto-adaptativos. Apesar disso, poucos trabalhos, como, (Denaro et al., 2007), (Stevens et al., 2007) e (Wen et al., 2005), provem soluções que tratam uma das maiores preocupações de sistemas auto-adaptativos: garantir a confiabilidade do comportamento auto-adaptado. Assim, a principal idéia dessas abordagens é executar testes antes de efetivamente adaptar o comportamento. A partir disso, torna-se possível avaliar se a adaptação sugerida é ou não adequada.

Embora seja possível definir casos de teste que consigam testar o comportamento adaptado, essas abordagens publicadas não tratam preocupações importantes, como, por exemplo:

- Relacionar artefatos a serem testados com seus casos de teste. Torna-se importante não somente definir os testes por si só, mas também relacioná-los com os artefatos (ou comportamento) que eles são capazes de testar.
- Definir diferentes conjuntos de dados de entrada e assertivas de saída para um teste específico.
- Definir a ordem de execução dos testes usados para validar um dado artefato. Tal informação torna-se importante, pois há casos em que um teste depende de outro teste para realizar sua execução sem problemas.
- Permitir o uso da atividade que representa o auto-teste em diferentes processos de auto-adaptação, isto é, tal atividade não deve estar amarrada a somente um processo.
- Definir diferentes logs que sejam responsáveis por armazenar o resultado dos testes executados.

Baseado nos itens acima, decidimos propor um framework que permite realizar a criação de sistemas multi-agentes capazes de auto-testar o comportamento auto-adaptado. Esse framework permite a construção de agentes auto-adaptativos que podem executar diferentes processos (control-loops) de auto-adaptação compostos por atividades capazes de realizar coleta de dados, análises, decisões, etc. (a- Costa et al., 2010) e (c- Costa et al., 2010) apresentam versões iniciais desse framework. A cópia desses artigos está nos Apêndices 1 e 3, respectivamente.

Como documentar testes e artefatos testados é uma importante tarefa para sistemas auto-adaptativos, pretendemos criar um transformador que permita a partir de diagramas UTP, gerar código automático. Esse código gerado usará o framework proposto para executar seus testes. Assim, para que seja possível a criação do transformador, os passos descritos na Figura 6 serão seguidos.

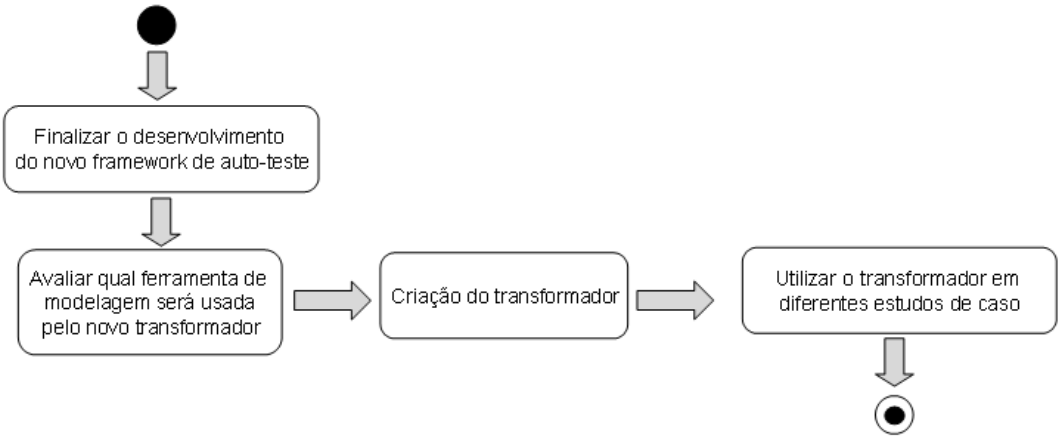

Figura 6. Procedimentos para criar o transformador de testes.

A primeira etapa está relacionada a finalização do novo framework de auto-teste. Para isso, pretendemos avaliar se há a necessidade de incluir novos conceitos de teste, assim como estabilizá-lo a partir do seu uso em diferentes domínios.

Já na segunda etapa será feita uma análise de qual ferramenta de modelagem, responsável por criar diagramas UML, poderá ser utilizada pelo transformador a ser criado. O novo transformador deverá ler um ou mais arquivos XML, gerados por alguma ferramenta, para identificar quais entidades de teste foram modeladas em diagramas UML. A partir disso, será possível realizar a geração de novos casos de teste. Rational Software Architecture (RSA, 2010) e Enterprise Architect (EA, 2010) são exemplos de ferramentas que poderiam ser utilizadas pelo transformador, já que geram arquivos XML desse tipo.

Após a escolha da ferramenta de modelagem, será realizada a criação do transformador na terceira etapa. Esse transformador irá criar testes unitários e funcionais na linguagem Java, já que o framework de auto-teste está sendo desenvolvido nessa linguagem. As bibliotecas DBUnit (DBUnit, 2010) e Fest (Fest, 2010), referente aos testes unitários e funcionais, respectivamente, devem ser usadas. Elas foram escolhidas, pois são amplamente aceitas e utilizadas na indústria.

Na última etapa iremos utilizar o transformador em diferentes domínios que seja necessário gerar testes para sistemas que realizam auto-testes. Os estudos de caso que pretendemos usar estão descritos nos artigos (a- Costa El al, 2010) e (c- Costa El al, 2010), que estão relacionados, respectivamente, ao domínio de deslizamento de terra e controle de estoque e suprimento de petróleo e derivados.

5 Trabalhos Relacionados

Nesta seção são apresentados alguns trabalhos relacionados em diferentes linhas de pesquisa. A primeira linha (subseção 5.1) refere-se a ferramentas de gerenciamento de dados utilizadas para documentar diversos conceitos de teste. Na subseção 5.2 apresentamos algumas linguagens de modelagem e programação voltadas para testes. A partir disso, avaliamos quais conceitos de teste identificados a partir do estudo inquisitivo, descrito na seção 2, são representados em cada abordagem. Na subseção 5.3 são

mencionados alguns trabalhos referente a auto-teste de sistemas adaptativos. Já na subseção 5.4 são apresentados alguns trabalhos de geração de dados, isto é, trabalhos que a partir de diagramas UML conseguem gerar casos de teste em diferentes linguagens.

5.1.1 Ferramentas de Gerenciamento

Uma da ferramentas utilizadas em nossos projetos testados é o Rational TestManager - RTM (RTM, 2010). Essa é uma ferramenta de gerenciamento, execução e documentação de casos de teste. O interessante dessa ferramenta é que ela provê visões interessantes a partir da interface e que não são oferecidas por outras abordagens, como, por exemplo, a habilidade de agrupar conceitualmente casos de teste e visualizar quais casos de teste são executados por quais suítes definidas no sistema. Por outro lado, a ferramenta não provê importantes conceitos de teste, como: (i) dependências entre testes, (ii) a identificação de quais testes são obrigatórios ou opcionais, (iii) quais testes são manuais e automatizados, (iv) identificação dos tipos de teste, e (v) quais testes estão atualizados para uma dada versão de um sistema.

Outra ferramenta interessante é o Rational Quality Manager – RQM (RQM, 2010), que possui como finalidade gerenciar, executar e documentar casos de teste e artefatos a serem testados, como, por exemplo, realizar a rastreabilidade dos requisitos testados com seus casos de teste relacionados. Um conjunto de conceitos de teste são representados, como, por exemplo, suítes, bugs, casos de teste, plano de teste, pessoas responsáveis por cada atividade, etc. Por outro lado, há conceitos identificados no estudo inquisitivo descrito na seção 2 que não são representados, como: (i) versão do sistema que um caso de teste está atualizado, (ii) tipos de dependências entre casos de teste, e (iii) informar se um caso de teste é ou não obrigatório para uma específica versão.

(Atlassian JIRA, 2010) é um sistema que realiza o gerenciamento de *issues*, como, por exemplo, casos de testes, tarefas, bugs, documentos, etc. Tal ferramenta permite que stakeholders definam rastreabilidades, cadastramento de diferentes tipos de *issues*, gerar diferentes relatórios, etc. Logo, essa ferramenta poderia ser utilizada para representar conceitos definidos no meta-modelo estendido da UTP para que os conceitos considerados pertinentes de teste possam ser documentados.

5.2 Linguagens (Modelagem/Programação)

De acordo com (UTML, 2010) os benefícios da "engenharia dirigida a modelos" (MDE) para o desenvolvimento de produtos de software têm sido demonstrado em numerosas ocasiões. Conseqüentemente, tais benefícios podem também ser alcançados em MDE para o desenvolvimento de testes de software. Dessa forma, os testes baseados em modelos é chamado de "engenharia de testes dirigida a modelos" (MDTE) ou simplesmente "testes dirigidos a modelos" (MDT). No entanto, otimizar a eficiência de MDT, boas práticas e específicos padrões para desenvolvimento de teste tem que ser tomado em conta. Baseado nessa idéia, (Feudjio, 2010) propôs uma notação projetada para MDT orientada a padrões chamada de Unified Test Modeling Language (UTML). Ela provê os meios para projetar todos os aspectos de um sistema de teste em um alto nível de abstração e que sejam independentes de qualquer infra-estrutura específica de nível mais baixo de teste. Além disso, essa abordagem também provê uma orientação para seguir padrões de projeto de teste e evitar problemas usuais de MDT. Tal abordagem oferece uma ferramenta chamada MDTester que permite modelar os conceitos

propostos pela UTML. No entanto, essa ferramenta e linguagem não representam os conceitos de teste identificados pelo nosso estudo inquisitivo (ver seção 2).

A linguagem de teste chamada "AGEDIS modeling language" – AML (Trost et al., 2010) (Agedis, 2010) é baseada no meta-modelo da UML 1.4 e permite a especificação estruturada (estática) e comportamental (dinâmica) de testes em modelos UML, assim como a UML Testing Profile. Apesar da AML representar diversos conceitos de teste, ela não representa boa parte dos conceitos mencionados na seção 2, como, por exemplo, identificação de testes obrigatórios e opcionais, casos de teste manuais e automatizados, tipos de teste, definir classificações de teste (ex: novos, regressão e impactados), etc.

Testing and Test Control Notation - TTCN-3 (TTCN-3, 2010) é uma linguagem modular que parece com uma linguagem típica de programação. Essa linguagem é amplamente aceita como um padrão para o desenvolvimento de testes nas áreas de telecomunicação e comunicação de dados. Apesar disso, ela não provê um conjunto de conceitos úteis que equipes de teste geralmente precisam, como, por exemplo, a identificação dos casos de teste obrigatórios e opcionais para uma específica versão de um sistema, tipos de dependências entre casos de teste, uma forma de visualizar casos dos testes relacionados a partir de algum conceito, etc.

5.3 Trabalhos de Auto-Teste

O foco principal do trabalho apresentado em (Wen et al., 2005) é prover um framework de auto-teste usado em microprocessadores. O framework consiste de duas etapas principais: (i) auto-testar o núcleo do processador, e (ii) testar e analisar outros componentes, usando o núcleo do processador testado como o analisador das respostas. Uma vez que os componentes do framework são amarrados a conceitos de microprocessadores, torna-se inviável seu uso em diferentes domínios. Por outro lado, nossa abordagem pretende permitir seu uso em diferentes domínios.

Em (Denaro et al., 2007) os autores apresentam uma abordagem auto-adaptativa responsável por usar um *control-loop* (processo auto-adaptativo) estruturado em três partes: mecanismos de monitoração, mecanismos de diagnósticos, e estratégias de adaptação. Mesmo considerando tal estrutura, a abordagem não considera etapas importantes definidas em nossa proposta, como, por exemplo, definir dados de entrada e assertivas de saída para testes, assim como definir a ordem de execução de um conjunto de testes para validar um artefato específico. Além disso, o processo de auto-teste adotado em [3] é fixo, isto é, não é possível criar outros *control-loops* que realizem auto-teste, assim como em nossa abordagem.

(Stevens et al., 2007) propõe um framework para testar sistemas auto-adaptativos. Esse trabalho introduz o conceito de container autonômico, que é considerado uma estrutura de dados que tem *capabilities* de auto-gerenciamento, e que também tem a habilidade implícita de realizar auto-teste. Tal abordagem usa uma estratégia que copia recursos usados em testes, enquanto que o recurso real é utilizado pelo sistema. Tal estratégia é conhecida como "replicação com validação". Apesar disso, o framework não permite o uso de atividades de auto-teste em outras arquiteturas, já que o framework é amarrado com a arquitetura em camadas da IBM para sistemas de computação autonômica (IBM, 2003). Conseqüentemente, as atividades de teste não podem ser usadas em diferentes processos de auto-adaptação, assim como nossa proposta.

5.4 Geração de Dados

Na literatura há uma grande quantidade de trabalhos que realizam geração de casos de teste, assim como geração de dados usados em testes a partir de diagramas UML. (Abdurazik, 2000), por exemplo, usa diagramas de colaboração para realizar a geração de dados para testes. No entanto, tal abordagem não realiza a geração de casos de teste, que seria viável a partir de outros diagramas UML.

(Vieira, 2006) é um exemplo de trabalho que utiliza diagramas de atividades e diagramas de classe para gerar casos de teste escritos em TSL (Siemens Test Script Language). Essa linguagem possui a idéia similar a TTCN-3, pois representa diversos conceitos de teste. No entanto, ela não está relacionada as áreas de telecomunicação e comunicação de dados, assim como TTCN-3. Nesse trabalho, cada caso de teste gerado usa como base um diagrama de atividade, onde é definido o passo a passo da sua execução, enquanto que o diagrama de classes é usado para identificar dados que possam ser usados em tal teste. Essa abordagem é uma solução interessante que talvez seja adotada para gerar testes e dados para o framework de auto-teste.

Diferentemente de (Vieira, 2006), os trabalhos (Hartmann, 2000) e (KIM, 1999) usam o diagrama de estados para gerar casos de teste. Tais abordagens consideram os estados como etapas de um ou mais testes. O trabalho (Hartmann, 2000), em especial, permite a geração de testes de integração e comunicação, como, por exemplo, para sistemas que utilizem componentes COM- e CORBA. Além disso, ele também gera casos de teste na linguagem TSL, assim como (Vieira, 2006).

Como ainda não definimos quais diagramas UML serão usados para realizar as gerações dos casos de teste, estamos avaliando a partir de diversos trabalhos propostos, quais seriam esses diagramas mais interessantes. Apesar disso, já pudemos concluir que os diagramas mais usados para geração de testes são: classes, atividade, estado e colaboração. Logo, parte ou todos esses diagramas serão usados em nosso trabalho.

6 Cronograma

Na Tabela 2 é apresentado o cronograma do trabalho. A seguir, são descritas as atividades a serem cumpridas

Tabela 2. Cronograma da tese de doutorado.

	2010.1	2010.2	2011.1	2011.2	2012.1
A1					
A2					
A3					
A4					
A5					
A6					
A7					
A8					
A9					
A10					
A11					

A1: Realizar a identificação de importantes conceitos de teste que devem ser documentados a partir da nossa experiência com testes.

A2: Avaliar quais conceitos de teste ainda não estão modelados em abordagens propostas na literatura.

A3: Criar o novo meta-modelo estendido da UTP.

A4: Definir restrições de boa formação para o novo meta-modelo a partir da linguagem OCL.

A5: Aplicar em diferentes estudos de caso o novo meta-modelo.

A6: Realizar experimentação do novo meta-modelo.

A7: Definir restrições de permissão a partir da linguagem *SecureUML*.

A8: Terminar o desenvolvimento do novo framework de auto-teste e avaliar qual ferramenta de modelagem poderá ser usado pelo novo transformador.

A9: Criar um transformador que permita gerar uma instância do framework de auto-teste a partir de diagramas UTP.

A10: Utilizar o transformador em diferentes estudos de caso.

A11: Escrever a tese de doutorado.

7 Considerações Finais

Nesse documento foi apresentada a proposta de uma nova linguagem de modelagem de teste responsável por representar importantes conceitos de teste identificados por um estudo inquisitivo realizado com sistemas de larga-escala. Além disso, o documento menciona regras de boa formação que serão criadas a partir da linguagem OCL, assim como novas restrições de permissão que serão escritas em SecureUML.

Um dos focos desse trabalho também será a criação de um framework capaz de realizar auto-teste em sistemas auto-adaptativos, já que essa é uma deficiência presente na comunidade de computação autonômica. A partir disso, pretendemos realizar a geração automática de casos de teste a partir de diagramas UTP. Consideramos esse trabalho de grande importância, pois a necessidade de confiar em adaptações realizadas por sistemas tem sido umas das grandes preocupações da indústria.

Referências

Abdurazik, A. and Offut, J. (2000), "Using UML Collaboration Diagrams for Static Checking and Test Generation", Proceedings of UML 2000 - The Unified Modeling Language. Advancing the Standard. Third International Conference, York, UK, Springer, pp. 383-395, 2000.

AGEDIS - Automated Generation and Execution of Test Suites for DIstributed Component based Software, http://www.agedis.de.

Amodt, A. and Plaza, E., Case-based reasoning: Foundational issues, methodological variations, and system approaches. In AI Communications, volume 7:1, pages 39–59. IOS Press, Março de 1994.

Atlassian JIRA, http://www.atlassian.com/software/jira/. Último acesso em Novembro de 2010.

Black, R.: Managing the Testing Process: Practical Tools and Techniques for Managing Hardware and Software Testing, Publisher: Wiley, 2nd edition, ISBN: 0471223980, 2002

Bellifemine, F., Caire, G., Trucco, T., Rimassa, G., Jade Programmer's Guide, 2007.

Bigus, J. P.; Schlosnagle, D. A., Pilgrim, J. R.; et. al..ABLE: A toolkit for building multiagent autonomic systems. IBM Syst. J. 41, 3, 350–371, 2002.

Booch, G., Rumbaugh, J., and Jacobson, I.: Unified Modeling Language User Guide, 2nd Edition, The Addison-Wesley Object Technology Series, 2005.

a) Costa, A. D., Nunes, C., Silvia, V., Fonseca, B. Lucena, C. J. P., JAAF+T: A Framework to Implement Self-Adaptive Agents that Apply Self-Test. In Proceeding of the 25th Symposium on Applied Computing (SAC 2010), Sierre, Switzerland, pp. 928-935, Março de 2010.

b) Costa, A. D., Silvia, V., Garcia, A., Lucena, C. J. P., "Improving Test Models for Large Scale Industrial Systems: An Inquisitive Study", Proceedings of the ACM/IEEE 13th International Conference on Model Driven Engineering Languages and Systems, Part I, LNCS Springer 6394, pp. 301-315, Oslo, Norway, 2010.

c) Costa, A. D., Silvia, V., Lucena, C. J. P., "A Multi-Agent System Framework to Assure the Realiability of Self-Adapted Behaviours", Monografias em Ciência da Computação nº 16/10, Departamento de Informática da Pontifícia Universidade Católica do Rio de Janeiro, 2010.

DBUnit web site, http://www.dbunit.org/. Último acesso em Novembro de 2010.

Denaro, G., Pezze, M., and Tosi, D., Designing Self-Adaptive Service-Oriented Applications. In Proceedings of the Fourth International Conference on Autonomic Computing. IEEE Computer Society, Washington, DC, 16, 2007.

Dios, M. A. G., Clavel, M., Egea., M., The Eye OCL Software (EOS), http://www.bm1software.com/eos/. Último acesso em Novembro de 2010.

Dobson, S., Denazis, S., Fernández, A., Gaiti, D., Gelenbe, E., Massacci, F., Nixon, P., Saffre, F., Schmidt, N., Zambonelli, F., A survey of autonomic communications, *ACM Transactions Autonomous Adaptive Systems (TAAS)*, 223-259, Dezembro de 2006.

EA – Enterprise Architect, http://www.sparxsystems.com.au/. Último acesso em Dezembro de 2010.

Fayad, M., Johnson, R., Building Application Frameworks: Object-Oriented Foundations of Framework Design (Hardcover), Wiley publisher, first edition, ISBN-10: 0471248754, 1999.

Feudjio, A. V.: MDTester User Guide, http://www.fokus.fraunhofer.de/distrib/motion/utml/. Último acesso em Novembro de 2010.

Fest – Fixtures for Easy Software Testing, http://fest.easytesting.org/. Último acesso em Novembro de 2010.

Fink, A.: The Survey Kit: How to ask survey questions, Volume 2, Publisher: SAGE, ISBN 0761925791, 2003.

Gamma, E., Helm, R., Johnson, R., Vlissides, J., Design Patterns: Elements of Reusable Object-Oriented Software, 1st.ed. USA: Addison-Wesley, 1995.

Graham, D., Veenendaal, E. V., Evans, I., Black, R., Foundations of Software Testing: ISTQB Certification, Intl Thomson Business Pr, ISBN: 1844809897, 2008.

Harrold, M. J.: Testing: A Roadmap. Proceedings of ICSE 2000, Future of Software Engineering, pp. 61-72, 2000.

Harrold, M. J.: Testing Evolving Software: Current Practice and Future Promise. Proceedings of ISEC 2008, pp. 3-4, 2008.

Hartmann, J.; Imoberdorf, C. & Meisinger, M., "UML-based integration testing", Proceedings of ISSTA'2000, pp. 60-70, 2000.

IBM, An architectural blueprint for autonomic computing. Technical Report., IBM, 2003.

IEEE-SA Standards Board, IEEE Standard for Software Test Documentation http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=741968&userType=inst. Último acesso em Novembro de 2010.

JUnit Web Site, http://www.junit.org/, Último acesso em Agosto de 2010.

Kaner, C., Bach, J., Pettichord, B.: Lessons Learned in Software Testing, Publisher: Wiley,1st edition, ISBN: 0471081124, 2001.

Kaner, C., Falk, J., Nguyen, H. Q.: Testing Computer Software, Publisher: Wiley, 2nd edition, ISBN: 0471358460, 1999.

Karam, K. S., Landslide Hazards Assessment and Uncertainties, Thesis: Massachusetts Institute of Technology, 2005.

Kephart, J. O. and Chess, D. M., The Vision of Autonomic Computing. *Computer* 36, 41-50, Janeiro de 2003.

Kim, Y.; HONG, H.; BAE, D. & CHA, S., "Test cases generation from UML state diagrams", Software, IEE Proceedings- [see also Software Engineering, IEEE Proceedings] 146(4), pp. 187-192, 1999.

King, T. M., Ramirez, A. E., Cruz, R., Clarke, P. J., An integrated self-testing framework for autonomic computing systems, Journal of Computers, Vol. 2, No. 9, Novembro de 2007.

Lethbridge, T., Sim, S., Singer, J.: Studying Software Engineers: Data Collection Methods for Software Field Studies, Submitted May 2000 to Empirical Software Engineering, 2000.

Lodderstedt, T., Basin, D., Doser, J., "SecureUML: A UML-Based Modeling Language for Model-Driven Security", Published in Proceeding UML '02 Proceedings of the 5th International Conference on The Unified Modeling Language, ISBN:3-540-44254-5, 2002.

OMG OCL, Documents associated with Object Constraint Language, Version 2.2, http://www.omg.org/spec/OCL/2.2/. Último acesso em Novembro de 2010.

OMG UTP, UML Testing Profile, version 1, http://www.omg.org/cgi-bin/doc?formal/05-07-07, Último acesso em Agosto de 2010.

OMG UML 2.0, OMG document number: formal/2010-05-03, http://www.omg.org/spec/UML/2.3/. Último acesso em Novembro de 2010.

Pohl, K., Bockle, G., Linden, F.: Software Product Line Engineering, Publisher: Birkhauser, ISBN 3540243720, New York, 2005.

RFT - Rational Functional Tester, http://www-01.ibm.com/software/awdtools/tester/functional/. Último acesso em Novembro de 2010.

RPT - Rational Performance Tester, http://www.acutest.co.uk/acutest/testing-rational-ibm. Último acesso em Novembro de 2010.

RQM - Rational Quality Manager, http://www-01.ibm.com/software/awdtools/rqm/. Último acesso em Novembro de 2010.

RSA – Ratinal Software Architect, http://www.ibm.com/developerworks/rational/products/rsa/. Último acesso em Dezembro de 2010.

RTM - Rational TestManager and Rational ManualTest, http://www-01.ibm.com/software/awdtools/test/manager/. Último acesso em Novembro de 2010.

Stevens, R., Parsons, B., and King, T. M., A self-testing autonomic container. In Proceedings of the 45th Annual Southeast Regional Conference (Winston-Salem, North Carolina). ACM-SE 45. ACM, New York, NY, 1-6, 2007.

Trost, J., and Cavarra, A. AGEDIS Language Specification, http://www.agedis.de/documents/d127_1/AGEDIS-ls-fpd.pdf. Último acesso em Agosto de 2010.

TTCN-3 web site, http://www.ttcn3.org/. Último acesso em Novembro de 2010.

UTML - The Unified Test Modeling Language for Pattern-Oriented Test Design, http://www.fokus.fraunhofer.de/en/motion/ueber_motion/technologien/utml/index.html21. Último acesso em Agosto de 2010.

Vieira, M.; Leduc, J.; Hasling, B.; Subramanyan, R. & Kazmeier, J., "Automation of GUI testing using a model-driven approach", Proceedings of the 2006 international workshop on Automation of software test (AST '06), ACM Press, 2006.

Wen, C., Wang, L.-C, Cheng, K.-T, Yang, K., Liu, W.-T.., "On a Software-Based Self-Test Methodology and Its Application". IEEE VLSI Test Symposium, Maio de 2005.

Wooldridge, M. and Jennings, "N. R. Pitfalls of agent-oriented development," Proceedings of the Second International Conference on Autonomous Agents (Agents'98), ACM Press, pp. 385-391, 1998.