UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM COMPUTAÇÃO

JÚLIO GERCHMAN

Integrando Injeção de Falhas ao Perfil UML 2.0 de Testes

Dissertação apresentada como requisito parcial
para a obtenção do grau de
Mestre em Ciência da Computação

Prof[a]. Dr[a]. Taisy S. Weber
Orientadora

Porto Alegre, março de 2008

CIP – CATALOGAÇÃO NA PUBLICAÇÃO

Gerchman, Júlio

Integrando Injeção de Falhas ao Perfil UML 2.0 de Testes / Júlio Gerchman. – Porto Alegre: PPGC da UFRGS, 2008.

75 f.: il.

Dissertação (mestrado) – Universidade Federal do Rio Grande do Sul. Programa de Pós-Graduação em Computação, Porto Alegre, BR–RS, 2008. Orientadora: Taisy S. Weber.

1. Tolerância a falhas. 2. Injeção de falhas. 3. Teste de software. 4. Engenharia de software. I. Weber, Taisy S. II. Título.

"Yes, features do sell products,
but only if they actually work!"
— STEVEN R. RAKITIN

AGRADECIMENTOS

Agradeço à minha orientadora, Prof[a]. Dr[a]. Taisy Silva Weber, por todo o seu apoio desde que ingressei no Grupo de Tolerância a Falhas e por seu entusiasmo sempre presente na docência e na pesquisa.

Agradeço aos meus amigos John Jochens, Rodrigo Kassick e Carolina Chiao pelas discussões técnicas sobre engenharia de *software*, filosofia, arquitetura, mecânica, televisão, trabalho, futebol, ciclismo, programação e culinária.

Agradeço acima de todos aos meus queridos irmãos e aos meus queridos pais, fontes inesgotáveis de conselhos, de alegrias, de conversas e de admiração.

SUMÁRIO

LISTA DE ABREVIATURAS E SIGLAS 9

LISTA DE FIGURAS 11

RESUMO 13

ABSTRACT 15

1 INTRODUÇÃO 17
1.1 Motivação 18
1.2 Objetivo 18
1.3 Resultados alcançados 19
1.4 Organização do trabalho 19

2 INJEÇÃO DE FALHAS 21
2.1 Técnicas de injeção de falhas 21
2.2 Modelos de falhas 24
2.3 Ambiente de injeção de falhas 24
2.4 Condução de experimentos de injeção de falhas 25
2.5 Padrões de projeto para ferramentas de injeção de falhas 25
2.6 Ferramentas de injeção de falhas 26

3 PERFIL UML 2.0 DE TESTES 31
3.1 Perfis UML 32
3.2 Objetivos do Perfil UML 2.0 de Testes 33
3.3 Grupos de conceitos 34
3.3.1 Arquitetura de teste 35
3.3.2 Comportamento de teste 36
3.3.3 Dados de teste 37
3.3.4 Temporização de teste 38
3.4 Conclusões 39

4 EXTENSÃO PARA INJEÇÃO DE FALHAS 41
4.1 Princípios de projeto 41
4.2 Objetivos de U2TP-FI 42
4.3 Questões de projeto 43
4.4 Arquitetura geral 44
4.5 Grupos de conceitos 44
4.6 Conclusões 48

5 PROVA DE CONCEITO 51
5.1 Criação e transformação de modelos 51
5.2 Metodologia 52
5.3 Exemplo 1: colapso de nodo OurGrid 53
5.4 Exemplo 2: múltiplos injetores 57
5.5 Exemplo 3: geração de relatórios 61
5.6 Conclusões 62
6 CONSIDERAÇÕES FINAIS 65
6.1 Trabalhos futuros 66
REFERÊNCIAS 69
APÊNDICE A HISTÓRICO DE DESENVOLVIMENTO 73

LISTA DE ABREVIATURAS E SIGLAS

API	Application Programming Interface
CASE	Computer Aided Software Engineering
CORBA	Common Object Request Broker Architecture
CWM	Common Warehouse Metadata
EDOC	Enterprise Distributed Object Computing
FIERCE	Fault Injector for Evaluation of Remote Communication Environments
FIONA	Fault Injector Oriented to Network Applications
FIRMAMENT	Fault Injection Relocatable Module for Advanced Manipulation and Evaluation of Network Transports
FIRMI	Fault Injector for RMI
IP	Internet Protocol
JVMTI	Java Virtual Machine Tool Interface
MDA	Model Driven Architecture
MOF	Meta Object Facility
OMG	Object Management Group
PFI	Protocol Fault Injection
QoS	Quality of Service
RMI	Remote Method Invocation
SUT	System Under Test
TCP	Transmission Control Protocol
U2TP-FI	UML 2.0 Testing Profile with Fault Injection
U2TP	UML 2.0 Testing Profile
UDP	User Datagram Protocol
UML	Unified Modelling Language
XMI	XML Metadata Interchange
XML	Extensible Markup Language

LISTA DE FIGURAS

Figura 2.1: Ambiente básico de injeção de falhas 24
Figura 2.2: Padrão de projeto para injetores de falha 26

Figura 3.1: Arquitetura de perfis OMG 32
Figura 3.2: Arquitetura de teste 35
Figura 3.3: Comportamento de teste 37
Figura 3.4: Dados de teste 38
Figura 3.5: Temporização de teste 39

Figura 4.1: Arquitetura geral da extensão 45
Figura 4.2: Exemplo de classe e instância de descrição de falhas 46
Figura 4.3: Exemplo de diagrama de comunicação com injeção de falhas na mensagem sendData 48
Figura 4.4: Exemplo de diagrama de classes com pacotes de categoria e de carga de falhas 49

Figura 5.1: Modelo de falhas criado, referente ao injetor FIRMI 55
Figura 5.2: Caso de teste para avaliação de OurGrid 55
Figura 5.3: Especificação das instâncias de falhas 56
Figura 5.4: Carga de falhas XML FIRMI para o caso de teste 57
Figura 5.5: Carga de falhas Java FIRMI para o caso de teste 57
Figura 5.6: Aplicação de teste de JGroups com falha particionamento de rede . . 58
Figura 5.7: Modelo de falhas para a aplicação de teste JGroups 59
Figura 5.8: Carga de falhas para a aplicação de teste JGroups 60
Figura 5.9: Carga de falhas para FIONA e para FIERCE 60
Figura 5.10: Relatório descrevendo a carga de falhas de um experimento 62

RESUMO

Mecanismos de tolerância a falhas são implementados em sistemas computacionais para atingir níveis de dependabilidade mais elevados. O teste desses mecanismos é essencial para validar seu funcionamento e demonstrar sua eficácia. Uma técnica de teste usada nesse caso é a injeção de falhas: uma simulação ou protótipo funcional é executado em um ambiente onde falhas são artificialmente emuladas e o sistema monitorado de forma a entender seu comportamento, bem como avaliar a eficiência da implementação dos mecanismos de tolerância.

Descrever as atividades de teste usando modelos é útil para a documentação do sistema. O Perfil UML 2.0 de Testes (U2TP) é uma linguagem padronizada para a descrição de modelos de testes, possibilitando a representação de ambientes e atividades de verificação e validação. No entanto, U2TP não oferece elementos para suportar técnicas de injeção de falhas.

Este trabalho apresenta U2TP-FI, uma extensão do Perfil UML 2.0 de Testes para a descrição de atividades de teste que usem técnicas de injeção de falhas. U2TP-FI é uma linguagem de modelagem que oferece elementos para representar as falhas a serem emuladas em um ambiente de teste, descrevendo os parâmetros que regem seu comportamento, suas condições de ativação e suas relações com os componentes do sistema. O estabelecimento dessa linguagem permite uma melhor visualização da atividade, um melhor projeto do teste e uma fácil documentação do projeto. Além disso, possibilita a criação de ferramentas para automação do processo de injeção de falhas.

Como prova de conceito para demonstrar a viabilidade da proposta, foram desenvolvidos usando U2TP-FI modelos de teste para a injeção de falhas em aplicações usando injetores existentes. Ferramentas de transformação de modelos foram aplicadas para gerar de forma automatizada artefatos a serem usados na atividade, como cargas de falhas e relatórios.

Palavras-chave: Tolerância a falhas, injeção de falhas, teste de software, engenharia de software.

Integrating Fault Injection to the UML 2.0 Testing Profile

ABSTRACT

Computer systems use fault tolerance mechanisms to reach higher dependability levels. Testing those mechanisms is essential for the validation of their proper operation and for the verification of their effectiveness. Fault injection is a technique for testing fault tolerance mechanisms: a simulation or a functional prototype of the system is executed in a testbed environment where faults are artificially emulated. Monitoring its behavior enables the validation of the implementation and the evaluation of the efficiency of the fault tolerance mechanisms.

It is useful for documenting the system to describe the test activities using models. The UML 2.0 Testing Profile is a standard language to create test models, enabling the test engineer to describe the environment, data, components, validations and other elements of the activity. However, U2TP does not offer elements that support fault injection techniques.

This work presents U2TP-FI, an extension of the UML 2.0 Testing Profile to model test activities that use fault injection techniques. U2TP-FI is a modeling language offering elements to represent the faults to be emulated on a test environment, describing the parameters which govern their behavior, the activation conditions and the relations between them and the system components. Using this language allows a better visualization of the test activity, a better test project and an easier project documentation. Besides, it enables the development of automation tools for the fault injection process.

As a proof of concept to demonstrate the viability of the proposal, U2TP-FI was used to create test models for applications using existing fault injectors. Model transformation tools were applied to automatically generate test artifacts such as faultloads and reports.

Keywords: fault tolerance, fault injection, software testing, software engineering.

1 INTRODUÇÃO

É irreal considerar que sistemas computacionais não apresentarão problemas durante o seu funcionamento. Projetos complexos e requisitos que não correspondem fielmente às necessidades e aos ambientes de execução podem levar a erros na construção dos sistemas. Mesmo se o sistema for corretamente construído, falhas externas, como falta de luz ou problemas de comunicação em rede, podem induzir a erros e levar a defeitos na computação. No entanto, cada vez mais sistemas computacionais são considerados críticos, devido ao aumento da dependência de organizações e pessoas em seu funcionamento correto. Assim, *níveis de dependabilidade* cada vez maiores são exigidos por seus clientes. Dependabilidade pode ser definida como a confiança justificada que se pode depositar em um sistema computacional e é a soma de vários atributos: confiabilidade, disponibilidade, integridade, *safety*, manutenibilidade e confidencialidade (AVIZIENIS et al., 2004).

Para atingir altos níveis de dependabilidade, são implementados em sistemas computacionais mecanismos de tolerância a falhas. No entanto, garantir o funcionamento correto desses mecanismos em sistemas complexos é um desafio. Etapa chave no processo de garantia de qualidade (FELDMAN, 2005), o teste torna-se essencial por definição nestes casos. Adiar o teste dos mecanismos de tolerância a falhas para um momento posterior à sua entrada em produção pode levar a resultados desastrosos.

Uma técnica usada para realizar o teste de mecanismos de tolerância a falhas é a *injeção de falhas* (CARREIRA; SILVA, 1998). Nesta técnica, ou um modelo ou um protótipo funcional do sistema é executado em um ambiente onde falhas são artificialmente causadas e erros são emulados. Durante a execução, o sistema sob teste é monitorado para que seu comportamento sob condição de falhas seja estudado. Além de validar a implementação dos mecanismos implementados, medidas úteis podem ser derivadas, como a cobertura e a eficiência destes.

No entanto, o uso de técnicas de injeção de falhas aumenta a complexidade dos modelos de teste usados. Todo teste depende de um modelo que o descreva, nem que exista apenas na mente dos integrantes da equipe de garantia de qualidade. O uso de injeção de falhas aumenta o número de componentes a serem descritos, como as ferramentas usadas, além de implicar associações destes com os componentes já existentes na atividade anterior de teste.

Descrever os modelos de teste usados traz vantagens como melhoria na comunicação entre as equipes de teste e de desenvolvimento. Para realizar essa descrição, uma linguagem comum deve ser adotada. A aplicação da técnica de injeção de falhas leva à necessidade de uma linguagem que descreva também os seus elementos, além dos elementos de teste já existentes.

1.1 Motivação

O teste de um sistema depende de um modelo que descreva os elementos envolvidos na atividade, como os componentes alvo, a carga de trabalho, os resultados esperados e as ferramentas usadas. Esse modelo, se não for formalizado em diagramas ou outros artefatos, existe ao menos na mente dos integrantes da equipe de testes. A formalização desses modelos torna possível sua documentação e facilita sua análise, permitindo não só a integração com os modelos do restante do sistema mas também o reuso de documentação e de ferramentas nas diversas fases do desenvolvimento.

Para realizar a descrição de modelos de teste, o *Object Management Group* (OMG), grupo responsável pela manutenção da linguagem *Unified Modeling Language* (UML), criou o Perfil UML 2.0 de Testes (*UML 2.0 Testing Profile*, U2TP). U2TP é uma extensão da linguagem UML para realizar a descrição dos componentes e atividades envolvidos (OBJECT MANAGEMENT GROUP, 2005). Através desse perfil, é possível projetar, visualizar, especificar, analisar, construir e documentar os artefatos usados, que abrange tanto aspectos estáticos (arquitetura) como dinâmicos (comportamento) de teste de sistemas.

No entanto, U2TP não tem suporte à técnica de injeção de falhas. Cada equipe pode criar sua própria notação, mas isto causa duplicação de esforço, problemas de comunicação e dificuldades na criação de ferramentas que usem esses modelos.

Existem na área de injeção de falhas esforços para a definição de metodologias padronizadas para a aplicação desta técnica. Por exemplo, Menegotto e Weber adaptaram uma metodologia conhecida de análise de desempenho, incorporando essa técnica de teste (MENEGOTTO; VACARO; WEBER, 2007). No entanto, não há uma ênfase na definição de uma notação padronizada para a descrição de modelos das atividades de teste com injeção de falhas. Existem padrões de projeto para a descrição das ferramentas de injeção usadas, como o descrito por Leme, Martins e Rubira; no entanto, seu objetivo não é descrever a atividade de teste em si (LEME; MARTINS; RUBIRA, 2001).

O Perfil UML 2.0 de Testes é uma linguagem padronizada, suportando todos os demais componentes e elementos presentes em uma atividade de teste. Estender U2TP para integrar conceitos ligados à técnica de injeção de falhas torna-se uma abordagem natural.

Este trabalho se aproveita da experiência adquirida pelo Grupo de Tolerância a Falhas da UFRGS no desenvolvimento de ferramentas de injeção de falhas para a validação de aplicações computacionais e de protocolos de comunicação. O autor participou no desenvolvimento de diversas desses injetores, seja em sua construção ou nas discussões de questões de projeto. Integrar a aplicação das técnicas e das ferramentas em modelos de teste é visto como uma continuação do trabalho que vem sendo realizado.

1.2 Objetivo

O objetivo deste trabalho é definir uma extensão do Perfil UML 2.0 de Testes para permitir a descrição em UML de atividades de teste que usem técnicas de injeção de falhas. Os elementos próprios às ferramentas de injeção, como a descrição dos tipos das falhas e a associação de sua ocorrência com eventos de interesse deve ser compreendida pela extensão.

Para desenvolver a extensão torna-se necessário um estudo de U2TP para definir quais elementos podem ser reusados e quais devem ser adicionados. Conceitos como monitoração e registro de atividades já são parte de U2TP. O estudo visa definir quais conceitos são

próprios da técnica de injeção de falhas, devendo ser representados por novos elementos UML.

Juntamente ao estudo de U2TP, os injetores existentes devem ser comparados para que se possa definir a maneira mais abrangente de descrever a atividade. Isto leva em consideração os meios, os formatos e as informações usadas pelas ferramentas para definir os cenários de falha e a execução do teste.

A viabilidade da extensão deve ser exemplificada através de provas de conceito, em que casos de teste são modelados em U2TP com a indicação dos cenários de falha usando a extensão. Esses casos de teste devem compreender diferentes tipos de falhas, como comunicação e interface, e diversas ferramentas, de forma a demonstrar a flexibilidade da proposta. Ferramentas de transformação de modelos e geração de artefatos

Seguindo a filosofia de UML, a extensão deve definir apenas uma linguagem, e não uma metodologia de uso. O usuário é livre para usar os elementos oferecidos pela extensão dentro de sua própria metodologia. Apesar disso, as etapas usadas na criação das provas de conceito poderão ser usadas como sugestão para metodologias próprias, específicas para cada uso.

1.3 Resultados alcançados

Entre os resultados alcançados por este trabalho estão:

- *Comparação das estratégias usadas para a descrição de cenários de falhas*, realizada a partir do estudo de ferramentas de injeção.
- *U2TP-FI*, a extensão do Perfil UML 2.0 de Testes para injeção de falhas, oferecendo um conjunto de elementos e restrições UML para descrever atividades de teste que usem essa técnica.
- *Prova de conceito* da aplicação da extensão, através da modelagem de casos de teste e geração automatizada de artefatos a partir dos modelos.
- *Sugestão de metodologia* para o uso da extensão de U2TP para injeção de falhas, derivada da criação da prova de conceito.

1.4 Organização do trabalho

Este trabalho está organizado da seguinte forma:

- O Capítulo 2 discute a técnica de injeção de falhas para validação de sistemas, o ambiente necessário para sua aplicação e arquiteturas usadas por ferramentas.
- O Capítulo 3 apresenta os conceitos e elementos do Perfil UML 2.0 de Testes.
- O Capítulo 4 descreve U2TP-FI, mostrando sua arquitetura geral, definindo os três grupos de conceitos que compõem a extensão.
- Provas de conceito são apresentada no Capítulo 5, através da modelagem de casos de teste e geração automatizada de cargas de falhas para injetores.
- Considerações finais são apresentadas no Capítulo 6.

2 INJEÇÃO DE FALHAS

Níveis de dependabilidade cada vez maiores são exigidos por usuários de sistemas computacionais. Dependabilidade é o grau de confiança justificada que se pode depositar em um sistema computacional, um conceito composto pela soma de vários atributos: confiabilidade, disponibilidade, integridade, *safety*, manutenibilidade e confidencialidade (AVIZIENIS et al., 2004). Mecanismos de tolerância a falhas são implementados para atingir um nível maior de dependabilidade e a técnica de *injeção de falhas* pode ser usada para avaliar e validar o sistema.

Injeção de falhas é uma técnica experimental de validação de sistemas. Falhas são introduzidas artificialmente ou emuladas no ambiente de execução de um sistema, seja durante uma simulação de seu funcionamento ou durante um teste usando um protótipo funcional. O comportamento da aplicação é analisado para verificar se está de acordo com suas especificações. Técnicas de injeção de falhas podem ser usadas para avaliar o efeito tanto de falhas de hardware como de falhas de software (VOAS, 1997).

A injeção de falhas pode ser usada para atingir diversos objetivos: identificar gargalos de dependabilidade, analisar o comportamento do sistema na presença de falhas, determinar a cobertura dos mecanismos de detecção e recuperação de erros e avaliar a eficácia dos mecanismos de tolerância a falhas, inclusive a perda de desempenho introduzida por eles (HSUEH; TSAI; IYER, 1997). Os resultados dos testes e sua análise posterior possibilitam um melhor entendimento do comportamento da aplicação em um ambiente real.

Estender o Perfil UML 2.0 de Testes para contemplar atividades usando as técnicas de tolerância a falhas requer o estudo destas, listando quais são seus tipos e requisitos, que meios são usados para a condução de testes e que ferramentas existem para apoiá-los. A Seção 2.1 discute a injeção de falhas de software e de hardware e os meios usados para realizá-la. A Seção 2.2 discute os modelos de falhas, que descrevem o comportamento dos componentes e são usados para projetar uma campanha de teste. A Seção 2.3 mostra os elementos presentes em um ambiente de injeção de falhas e que são usados na condução de um experimento, cujas etapas são descritas na Seção 2.4. A Seção 2.5 descreve um padrão de projeto para descrever a arquitetura geralmente usada por ferramentas de injeção de falhas, e a Seção 2.6 compara características de interesse presentes em injetores encontrados na literatura da área.

2.1 Técnicas de injeção de falhas

Uma *falha* é um fenômeno que pode causar um desvio de um componente de software ou de hardware em relação à sua função original (CARREIRA; SILVA, 1998). Uma *falha de hardware* é causada ou por componentes que apresentam defeitos ou por condições ambientais que impeçam seu funcionamento correto. Por exemplo, um chip pode quei-

mar, seja por problema de fabricação ou por flutuações na voltagem aplicada. Esse tipo de falha é classificado por sua duração: *permanente* quando o dano é irreversível, *transiente* quando é causada por alterações ambientais temporárias ou *intermitentes*, quando o hardware usado é instável, independentemente de alterações ambientais. As *falhas de software* são causadas por especificação, projeto ou construção incorretas de um programa, levando a erros de codificação e incompatibilidades com outros componentes ou com o ambiente de execução. Os tipos de falha estão sumarizados na Tabela 2.1.

Tabela 2.1: Tipos de falhas

Tipo de Falha	Descrição
Hardware	Componentes que apresentam defeitos por problemas de especificação, de construção ou de ocorrência por alterações no ambiente de execução, causando erros no sistema e levando a defeitos
Software	Problemas na especificação, projeto ou construção do software que levam a erros de execução. Pode ser ativada durante uma execução normal ou por um cenário não previsto, como problemas de sincronização de tarefas ou ocorrência de erros causados por falhas de hardware

As técnicas de injeção de falhas têm como foco principal a validação de mecanismos de tolerância a falhas. Essa validação é necessária antes que um sistema crítico entre em produção: adiar os testes pode ser desastroso por não existir garantias que os mecanismos funcionem de maneira correta. O teste é necessário para assegurar os níveis de dependabilidade que são impostos ao sistema, determinando a cobertura dos mecanismos e avaliando sua eficiência em um ambiente próximo ao real.

A injeção de falhas de hardware e de software podem apresentar objetivos diferentes, sumarizados na Tabela 2.2. A *injeção de falhas de hardware* emula falhas de componentes de hardware para causar erros no sistema e verificar como os aplicativos sob teste se comportam dentro desses cenários. O objetivo principal é validar a implementação dos mecanismos de tolerância a falhas em relação à especificação original, forçando a execução de casos especiais e a ocorrência de defeitos de projeto ou codificação. Essa técnica pode levar à descoberta tanto de falhas de hardware como de falhas de software. A *injeção de falhas de software* não têm como objetivo avaliar uma implementação de mecanismos de tolerância a falhas, mas sim avaliar os casos de teste, selecionando os mais capazes de distingüir comportamentos válidos de inválidos (SUGETA; MALDONADO; WONG, 2004). A técnica principal é o *teste de mutantes*: são gerados versões alteradas do programa ou da especificação original, usando operadores que os modificam de forma a emular erros comuns que ocorrem durante seu desenvolvimento. A técnica foi estendida a testes de integração na forma de *mutação de interfaces* (DELAMARO; MALDONADO; MATHUR, 1996).

Neste trabalho é considerada apenas a técnica de *injeção de falhas de hardware*, focando na avaliação de mecanismos de tolerância a falhas implementados nos componentes de um sistema computacional.

Experimentos de injeção de falhas podem se dar por *simulação* ou pelo teste de *protótipos*. Técnicas de simulação permitem a realização de experimentos antes da codificação da implementação e oferecem um controle maior sobre os parâmetros de teste, tornando-

Tabela 2.2: Objetivos principais das técnicas de injeção de falhas

Injeção de Falha	Objetivo principal
Injeção de falhas de hardware	Avaliar a implementação dos mecanismos de tolerância a falhas de um sistema computacional
Injeção de falhas de software	Selecionar os casos de teste com maior capacidade de distingüir comportamentos válidos de inválidos

as capazes de modelar sistemas complexos com alta fidelidade (BARCELLOS; WOSZEZENKI; MUNARETTI, 2005). No entanto, como o modelo do sistema representa uma simplificação da realidade, os testes podem apresentar resultados com uma precisão inferior à desejada se algum parâmetro ou comportamento não for corretamente especificado. A injeção de falhas realizada em protótipos, por usar um sistema concreto e um ambiente que se aproxima à realidade, permite a obtenção de resultados mais precisos.

A injeção de falhas em protótipos pode ser realizada através de ferramentas de hardware ou de software. *Ferramentas de injeção por hardware* atuam através da alteração física do estado dos componentes de um sistema. Para isso, usam equipamentos como sondas ou ponteiras, alterando sinais em pinos específicos de componentes eletrônicos, ou até mesmo fontes radioativas para gerar íons pesados que alterem o estado de chips (HSUEH; TSAI; IYER, 1997). *Ferramentas de injeção de falhas por software* comportam-se como uma camada de código que monitora as atividades do sistema sob teste e que altera seu fluxo normal de execução, criando comportamentos incorretos de maneira controlada (CARREIRA; SILVA, 1998). Esta técnica permite simular falhas de hardware, software ou de interface entre sistemas emulando manifestações de erros, e não inserido como falhas físicas reais. O estado interno do sistema é alterado da mesma forma que aconteceria se essa falha em nível lógico ou elétrico ocorresse. Desta forma, a latência de erro é diminuída: o tempo para que uma falha se manifeste como um erro é menor.

Os tipos de experimento de injeção de falhas estão sumarizados na Tabela 2.3.

Tabela 2.3: Tipos de experimento de injeção de falhas

Tipo de experimento	Vantagens	Desvantagens
Simulação	Grande controle sobre os parâmetros usados no teste; facilidade de repetição de testes	Como o modelo é uma simplificação da realidade, algum parâmetro importante pode ser deixado de lado
Injetores em hardware	Mais próximos à falha real; maior precisão	Alto custo; grande latência de erro; podem causar danos irreparáveis em componentes
Injetores em software	Baixo custo; menor latência de erro; grande flexibilidade	Intrusividade temporal (tempo de execução do código do injetor) e espacial (presença de código externo ao sistema)

2.2 Modelos de falhas

Um ambiente de injeção adota um modelo de falhas, coleção de atributos e um conjunto de regras que governam a interação entre componentes que falharam (SCHNEIDER, 1993). Um componente que apresenta falhas tem um comportamento consistente com o modelo de falhas assumido. A adoção deste tipo de modelo permite planejar os mecanismos de tolerância a falhas, prevendo as maneiras como os erros se manifestam dentro do sistema. Cada sistema adota um modelo de falhas relacionado ao seu objetivo; de acordo com o modelo de falhas assumido, os mecanismos são projetados para que os cenários previstos sejam cobertos. Aplicações de rede têm de considerar falhas de comunicação, como perda ou corrupção de pacotes. Sistemas embarcados em satélites têm de considerar mecanismos para tolerar falhas causadas por íons pesados e raios cósmicos que podem alterar o estado interno dos componentes eletrônicos.

Cenários de falha, explorando as possibilidades e combinações de falhas, erros e defeitos que podem ocorrer, são delineados como parte do plano de teste do sistema. O cenário de falhas é usado como uma das entradas para as atividades de validação. Além de especificar qual o nível de abstração em que as falhas devem ser injetadas, são descritos seus parâmetros, como as condições de ativação e as distribuições probabilísticas que determinam seu padrão de repetição e seu tempo de atuação.

2.3 Ambiente de injeção de falhas

Devido às várias naturezas diferentes de falhas, bem como ao grande número de ambientes de execução e requisitos, é comum encontrar ferramentas de injeção específicas para um domínio de aplicação; por exemplo, injetores de falhas de comunicação. Apesar disso, as ferramentas tipicamente apresentam os mesmos componentes (HSUEH; TSAI; IYER, 1997). Um *injetor de falhas*, guiado por uma *biblioteca de falhas*, atua sobre um sistema alvo. O sistema alvo executa ações de acordo com *gerador de carga de trabalho*, que fornece comandos para seu funcionamento durante o teste. Um *monitor* registra a execução desses comandos e ativa o *coletor de dados*, cuja saída pode ser processada posteriormente por um *analisador de dados*. Esse ambiente básico de injeção de falhas é mostrado na Figura 2.1.

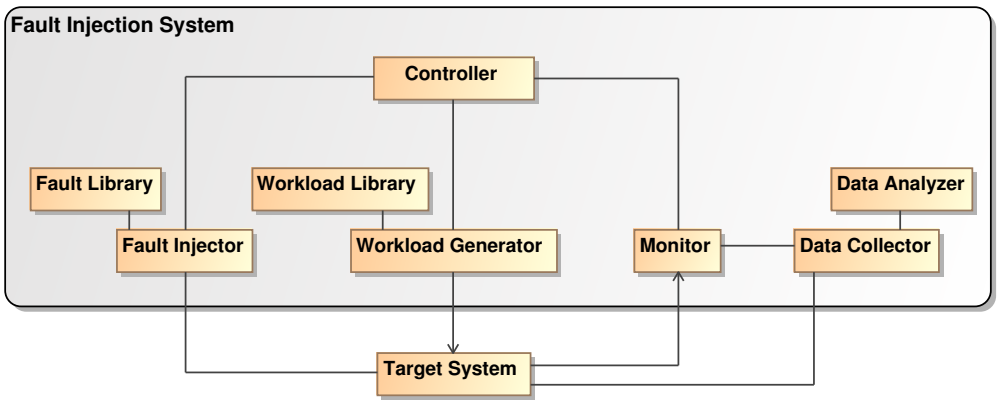

Figura 2.1: Ambiente básico de injeção de falhas, adaptado de (HSUEH; TSAI; IYER, 1997)

2.4 Condução de experimentos de injeção de falhas

A condução de um experimento de injeção de falhas é realizada em três fases distintas: definição do cenário de falhas, execução do teste e análise dos resultados.

A definição do *cenário de falhas* é realizada de acordo com a identificação dos mecanismos de tolerância a falhas implementados no sistema e o modelo de falhas adotado. Parte do plano de teste usado, o cenário é uma lista das falhas a serem injetadas durante o experimento. Para cada falha é definido seu tipo, local (componente), condições de ativação (como um momento no tempo ou um evento de interesse) e duração. Para realizar um teste amplo, é importante a criação de cenários de falhas variados, testando o sistema em diferentes situações. Quanto mais situações, maior é a segurança depositada no teste e melhor é a precisão das medidas obtidas.

A execução do teste leva em consideração a carga de trabalho (*workload*) do sistema, responsável por sua ativação e execução, bem como a carga de falhas (*faultload*), composta pelos conjuntos de falhas a serem injetadas. Durante a execução, é realizado o monitoramento das entradas, saídas e condições do teste. Esses dados são coletados e armazenados para análise posterior.

A análise dos resultados do teste é a última etapa do experimento. Além de validar o comportamento do sistema em relação aos requisitos e especificações, medidas de dependabilidade podem ser calculadas. Por exemplo, comparando os conjuntos de falhas injetados com os registros de ativação dos mecanismos de tolerância a elas, pode-se calcular a cobertura, ou seja, o percentual de falhas que os mecanismos atuam. Outra medida é a diferença de desempenho entre execuções com e sem falhas para avaliar a interferência dos mecanismos no tempo e na eficiência do sistema.

2.5 Padrões de projeto para ferramentas de injeção de falhas

Existem diversas ferramentas de injeção de falhas, cada uma adaptada a um ambiente de execução e a um tipo de aplicação específica. Devido ao grande número de combinações desses parâmetros existentes, não é prático construir uma ferramenta genérica o suficiente que injete qualquer tipo de falha. No entanto, como discutido na Seção 2.3, os ambientes usados em atividades de validação usando injeção de falhas são semelhantes; isto se reflete na arquitetura das ferramentas.

Para facilitar a construção de ferramentas, Leme, Martins e Rubira propõem a criação de um sistema de padrões para injeção de falhas (LEME; MARTINS; RUBIRA, 2001). Um sistema de padrões é um conjunto de soluções documentadas para um domínio de problemas específico, o que acelera a construção de uma aplicação e proporciona mais recursos para o desenvolvedor; é proposto um padrão de arquitetura que descreve a estrutura geral de uma ferramenta de injeção de falhas. São apresentados componentes destinados a ativar o sistema sob testes, injetar a falha, monitorar seu comportamento e expor os resultados através de uma interface.

O padrão, mostrado na Figura 2.2, define cinco subsistemas principais. Através de uma interface com o sistema alvo, *Activator* inicia sua execução, permitindo seu teste em condições normais, enquanto que *Injector* introduz falhas no ambiente onde é realizada a execução do sistema e *Monitor* verifica as condições de operação do sistema alvo. *Controller* coordena as atividades e a comunicação de todos os subsistemas, permitindo ao usuário configurar o teste e recuperar resultados através do subsistema *User interface*. Dois outros subsistemas auxiliares, *Fault manager* e *Monitored data manager*, servem

como repositórios de dados para a ferramenta.

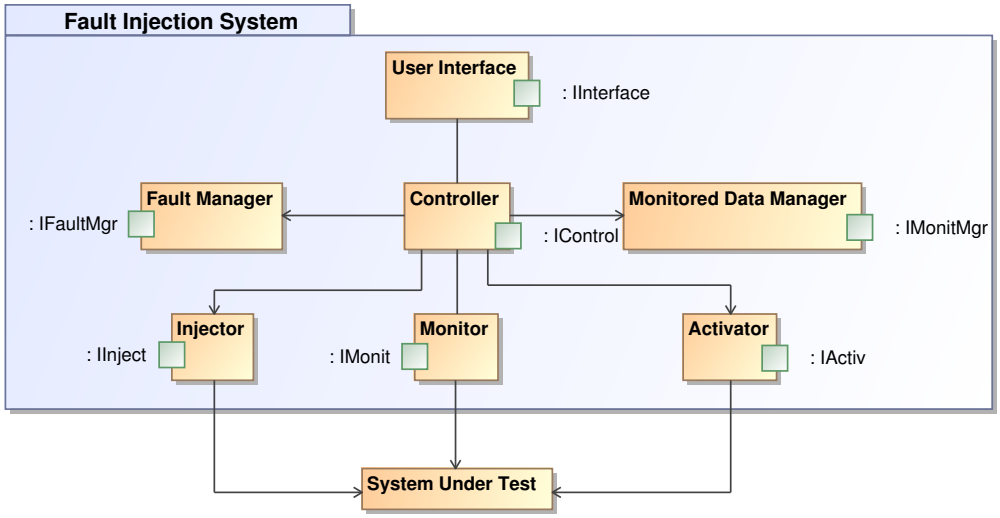

Figura 2.2: Padrão de projeto para injetores de falha, adaptado de (LEME; MARTINS; RUBIRA, 2001)

Esse padrão de arquitetura se propõe a descrever a ferramenta a ser construída; não faz parte de seus objetivos a descrição dos demais aspectos de um ambiente de injeção, como as condições de ativação de falhas. Não são oferecidos elementos que possibilitem a documentação do uso da ferramenta e de sua integração com o resto de um plano de testes. Mesmo usando o padrão para descrever o injetor de falhas, a especificação desses demais elementos usando apenas componentes encontrados em UML ou U2TP pode causar a especificação de notações diferentes para as atividades de injeção, dificultando a comunicação entre equipes e projetos distintos.

2.6 Ferramentas de injeção de falhas

Um dos pontos principais no desenvolvimento da extensão para injeção de falhas do Perfil UML 2.0 de Testes é a adição do conceito de falha a um modelo de testes. Este conceito deve representar o tipo de falha injetada durante uma atividade de validação do sistema, bem como as condições de ativação, como tempo ou evento de interesse.

Para definir os atributos e notações usados para modelar uma falha em um modelo de testes, foi realizada uma comparação entre diversas ferramentas existentes na literatura. Essa comparação busca sumarizar os meios usados para descrever as cargas de falhas (*faultloads*) usadas durante a atividade de teste.

Esta seção apresenta algumas ferramentas e sumariza as estratégias usadas por elas para descrever cargas de falhas: Jaca, FIONA/FIERCE, Simmcast, FIRMI e ORCHESTRA. As quatro primeiras ferramentas foram escolhidas pela facilidade de acesso à sua documentação e ao seu código-fonte, permitindo uma análise mais detalhada: FIONA, FIERCE e FIRMI são desenvolvidas pelo Grupo de Tolerância a Falhas da UFRGS, e Jaca e Simmcast por grupos de pesquisa respectivamente da Unicamp e da Unisinos. ORCHESTRA foi escolhida por sua significância na área de injetores falhas de comunicação, sendo uma ferramenta muito citada na literatura.

Jaca

Jaca é uma ferramenta para a validação de aplicações Java através da corrupção de atributos de classes, parâmetros de métodos e seus valores de retorno (MARTINS; RUBIRA; LEME, 2002). Essa injeção de falhas é realizada usando o *framework* Javassist, que estende os conceitos de reflexão computacional e introspecção oferecidos pela plataforma Java, permitindo alterar o *bytecode* de classes carregadas por uma aplicação, instrumentando-o conforme necessário.

A *faultload* é descrita em um arquivo de especificação de falhas, carregado pela ferramenta no início de uma campanha de testes. Cada linha de texto presente nesse arquivo especifica o tipo de falha, o alvo, os parâmetros e a condição de ativação. O tipo de falha é uma *string*, como `ReturnMethodFault`, indicando a alteração de um valor de retorno de método. O alvo são *strings* indicando o nome da classe e o nome do método. Os parâmetros dependem do tipo da falha injetada; por exemplo, uma alteração de retorno de método pode indicar uma adição de uma constante ao valor original. A condição de ativação é definida como o número de invocações do método alvo em que a falha será injetada.

FIONA e FIERCE

FIONA é um injetor de falhas de comunicação para a avaliação de aplicações Java de rede que usem o protocolo UDP para comunicação (JACQUES-SILVA et al., 2006). FIONA usa a interface de depuração JVMTI, oferecida pela plataforma Java, para instrumentar as classes de sistema responsáveis pela comunicação, o que permite a injeção de falhas como descarte, corrupção e duplicação de datagramas UDP, bem como a introdução de atrasos responsáveis por *time-outs* e reordenamento de mensagens. FIERCE é a extensão de FIONA para a injeção de falhas em aplicações Java que usem o protocolo TCP (GERCHMAN; WEBER, 2006).

FIONA é uma evolução de uma ferramenta anterior, uma extensão de Jaca para injeção de falhas em comunicação UDP. Apesar de usar mecanismos completamente diferentes de instrumentação de classes, FIONA e FIERCE mantiveram a mesma estratégia de descrição de *faultloads* usada por Jaca. Em arquivos de configuração carregados pela ferramenta no início da execução dos testes, linhas de texto indicam o tipo de falha, parâmetros como *host* e porta usados e as condições de ativação. Os parâmetros variam de acordo com a falha, podendo ser, por exemplo, percentual de datagramas descartados ou tempos superior e inferior de atraso no envio de mensagens pela rede. As condições de ativação oferecidas são o tempo decorrido desde o início do experimento ou o número de bytes transmitidos entre dois computadores.

ORCHESTRA

ORCHESTRA é uma ferramenta para a validação da implementação de protocolos de comunicação, em especial TCP, em sistemas operacionais (DAWSON; JAHANIAN; MITTON, 1997). A ferramenta se divide em uma camada denominada *Protocol Fault Injection* (PFI), carregada no núcleo do sistema operacional diretamente abaixo do módulo em teste, e um interpretador de *scripts* na linguagem Tcl. Toda mensagem enviada ou recebida pelo protocolo de comunicação é filtrada pelo PFI, que invoca um script Tcl, realizando ações como descarte ou alteração.

A definição das falhas a serem injetadas através de scripts Tcl é colocada como uma característica de flexibilidade de ORCHESTRA. O usuário da ferramenta não fica limi-

tado a falhas definidas pelos seus autores; ele pode criar comportamentos complexos, adaptados às suas necessidades.

Simmcast

Simmcast é um *framework* de simulação de eventos discretos destinado ao apoio do desenvolvimento e avaliação de protocolos de comunicação *multicast* (BARCELLOS; WOSZEZENKI; MUNARETTI, 2005). Os protocolos são definidos e simulados usando uma combinação de blocos básicos de construção, como filas e processos; o simulador provê suporte para endereços *unicast* e *multicast*. A ferramenta é baseada no *toolkit* de simulação JavaSim.

Simmcast, sendo um *framework*, oferece uma interface de programação (API) com operações para criação de objetos e simulação de operações de comunicação e temporização. O usuário, através da programação de classes Java, define o protocolo, a topologia da rede e a distribuição de agentes; parâmetros de simulação podem ser externalizados em arquivos de configuração. A injeção de falhas é especificada como parte da simulação.

FIRMI

FIRMI é um injetor de falhas de comunicação para aplicações Java que usem o protocolo RMI para comunicação entre componentes (VACARO; WEBER, 2006). A ferramenta usa as interfaces oferecidas pela plataforma Java a partir da versão 1.5 para instrumentação de classes, alterando as classes de sistema que gerenciam requisições RMI para emular os erros que ocorreriam em situações de falha. FIRMI oferece integração com o *framework* de teste JUnit e de integração Ant.

FIRMI oferece duas opções diferentes para sua configuração. Originalmente, os *faultloads* eram descritos através de classes Java que chamavam uma API fornecida pelo injetor, estabelecendo filtros para a alteração de requisições RMI de acordo com as estratégias de injeção de falhas definidas em código. Menos flexível mas de manipulação mais fácil, a ferramenta permite também a criação de arquivos de configuração em uma linguagem XML, onde são descritos os parâmetros de injeção (como *host* origem e destino) e as condições de ativação (como número de invocações a um método remoto via RMI ou tempo decorrido desde o início do experimento).

Comparação entre ferramentas

Um sumário das ferramentas apresentadas, notando o foco destas e a estratégia usada para descrição do cenário de falhas, pode ser encontrada na Tabela 2.4.

Como pode ser visto na Tabela 2.4, as ferramentas analisadas ou representam sua configuração de maneira descritiva, usando arquivos com indicações das falhas a serem injetadas, ou de maneira programática, através de scripts ou programas que acessam uma API predefinida. Estas duas formas de configuração foram a base para a estratégia de modelagem de falhas de U2TP-FI, a extensão descrita neste trabalho para injeção de falhas do o Perfil UML de Testes. U2TP-FI representa um tipo de falha injetada através de uma classe UML. No caso de injetores que representam sua configuração em arquivos, cada tipo de falha suportada pode ser mapeada em uma classe UML criada pelo projetista que modela os diagramas de casos de testes. Injetores que seguem uma maneira programática também podem ter sua configuração representada por classes UML: essa linguagem permite a associação de classes a um comportamento (*behavior*), ou seja, uma descrição de operações a serem executadas. Esse comportamento pode, por exemplo, ser modelado

Tabela 2.4: Comparação entre ferramentas de injeção de falhas

Ferramenta	Foco	Descrição do cenário
Jaca	Corrupção de valores em interfaces entre classes	Arquivos de configuração com a lista de falhas, seus parâmetros e condições de ativação
FIONA/FIERCE	Falhas de comunicação UDP e TCP em aplicações de rede	Arquivos de configuração com a lista de falhas, seus parâmetros e condições de ativação
ORCHESTRA	Avaliação de implementações de protocolos de comunicação	Scripts Tcl invocados por um módulo carregado no núcleo do sistema operacional
Simmcast	Simulação para avaliação de protocolos de comunicação	Programas Java usando APIs oferecidas pela ferramenta e pelo framework de simulação
FIRMI	Falhas de comunicação em aplicações baseadas em RMI	Arquivos de configuração XML ou classes Java usando API oferecida pela ferramenta

como um diagrama de seqüência. No caso em questão, cada script ou programa fornecido ao injetor seria representado por uma classe UML com um comportamento associado.

3 PERFIL UML 2.0 DE TESTES

O teste de um sistema sempre depende da construção de um modelo que o descreva, mesmo que este exista apenas em rascunho ou na mente do engenheiro de teste. A equipe de testes deve entender o comportamento desejado do sistema, bem como as entradas esperadas em sua execução. A existência prévia de um modelo do sistema permite aos engenheiros de teste visualizar mais rapidamente os requisitos que levaram à sua construção, permitindo investir mais recursos na geração de planos de testes mais abrangentes. Além disso, torna possível determinar a cobertura de testes e adequar os planos de teste de acordo com os recursos disponíveis (EL-FAR, 2001).

Capturar, formalizar e descrever os modelos usados permite o compartilhamento e reuso destes. A documentação pode ser usada por um novo engenheiro de teste na equipe para sua familiarização com o sistema. Problemas encontrados na especificação original podem ser discutidos entre a equipe de teste e a de desenvolvimento usando a mesma linguagem de descrição, facilitando a conversação. O reuso pode ser visto na evolução do modelo: a ele são incrementalmente adicionadas novas funcionalidades do produto, ao mesmo tempo que são expandidos os seus testes (APFELBAUM; DOYLE, 1997).

A arquitetura dirigida por modelos (MDA) da OMG busca padronizar o uso de linguagens de descrição, como UML. O desenvolvimento da versão 2 de UML, alinhado com a estratégia de MDA, teve como um de seus objetivos possibilitar a "execução" de UML, permitindo não só a geração de código, simulação e validação dos modelos, mas também a geração de testes. Esse objetivo permitiria uma melhor implementação de um processo de teste baseado em modelos.

Esse processo de teste baseado em modelos é realizado com o suporte do Perfil UML 2.0 de Teste (*UML 2.0 Testing Profile*, ou *U2TP*). U2TP define uma linguagem para projetar, visualizar, especificar, analisar, construir e documentar os artefatos de sistemas de teste (OBJECT MANAGEMENT GROUP, 2005). É uma linguagem para modelagem de testes que pode ser usada com todas as principais tecnologias de componentes e objetos e aplicada a sistemas de teste em diferentes domínios. U2TP pode ser usado isoladamente para a manipulação de artefatos de teste ou em uma maneira integrada com o resto da aplicação, manipulando juntamente artefatos de sistema e de teste.

A Seção 3.1 inicia este capítulo com a definição de um perfil UML, mecanismo adotado pela OMG que permite a especialização de um meta-modelo para uso em um determinado domínio de aplicação. Após, segue um estudo do Perfil UML 2.0 de Teste nas Seções 3.2 e 3.3, onde os objetivos e os elementos que o compõem são apresentados. Este estudo é necessário para a definição dos elementos do perfil relacionados a atividades de teste usando a técnica de injeção de falhas.

3.1 Perfis UML

A linguagem UML é mantida pelo OMG (*Object Management Group*), entidade que agrupa diversas empresas e indivíduos atuantes na área de análise e projeto de software. Além de UML, outras linguagens são desenvolvidas e mantidas pelo grupo, como CWM (*Common Warehouse Metadata*, dirigida a *data warehouses*). Essas linguagens são chamadas de *meta-modelos*: descrevem os modelos usados na análise e projeto de sistemas.

A OMG padronizou uma arquitetura de quatro camadas para organizar esses meta-modelos, exemplificada na Figura 3.1. No nível mais baixo, denominado M0, residem os objetos, as instâncias através das quais um sistema é executado. São objetos encontrados no "mundo real": código, máquinas, interfaces e outros artefatos concretos. Essas instâncias são descritas no nível M1, o nível de modelo. Essa descrição, no caso de um sistema orientado a objetos, poderia ser através de diagramas de classes UML. O modelo é uma simplificação dos artefatos concretos para documentar o sistema e facilitar sua análise e seu projeto.

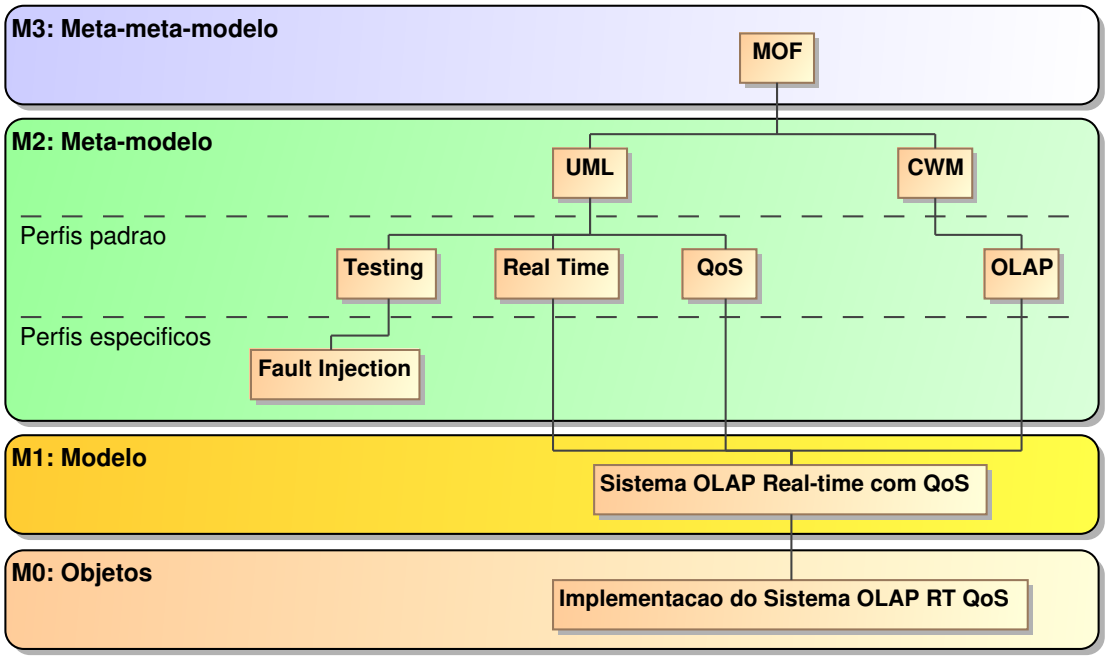

Figura 3.1: Arquitetura de perfis OMG, adaptado de (OBJECT MANAGEMENT GROUP, 1999)

A linguagem usada para descrever o modelo da aplicação é definida no nível M2, o nível de meta-modelo. Neste nível localizam-se linguagens como UML e CWM. Personalizações e adaptações desses padrões, como os perfis UML, também estão nesse mesmo nível. Cada meta-modelo cobre um domínio específico com grande número de aplicações. Essas áreas são normalmente decompostas em domínios que vão de encontro às necessidades de grupos menores de usuários, levando à criação de meta-modelos especializados, como EDOC (*Enterprise Distributed Object Computing*). Perfis UML, mesmo sendo construídos sobre essa linguagem, também estão localizados neste nível da hierarquia. Na hierarquia da OMG, todas essas linguagens (meta-modelos) são descritas por um meta-meta-modelo chamado MOF (*Meta Object Facility*), localizado no nível mais alto, M3. Segundo a documentação da organização, para evitar a proliferação de prefixos

meta, esse meta-meta-modelo é referido por *modelo MOF* (OBJECT MANAGEMENT GROUP, 1997).

É útil a um usuário de UML estender a linguagem para adaptá-la a necessidades, incorporando peculiaridades presentes no domínio de interesse. O padrão UML já incorpora mecanismos de extensão, chamados valores rotulados (*tagged values*), estereótipos (*stereotypes*) e restrições (*constraints*). Esses elementos são capazes de adaptar a semântica de UML sem alterar o meta-modelo e são referidas como mecanismos leves de extensão (*lightweight extensibility mechanisms*). É possível também estender o meta-modelo usando os meios presentes no modelo MOF, através da alteração do meta-modelo com a adaptação de metaclasses existentes ou a definição de novas.

A noção de *perfil* busca representar e estruturar essas extensões de meta-modelos. Perfil é uma especificação que especializa um ou mais meta-modelos padrão, chamados de meta-modelos de referência (OBJECT MANAGEMENT GROUP, 1999). É dedicado a um domínio específico de problema, permitindo adaptações flexíveis de um padrão "congelado". Um perfil é um agrupamento de elementos que especializam a semântica de um meta-modelo de referência. Os elementos são:

- Elementos selecionados do meta-modelo de referência que constituem o seu foco particular. Essa seleção não proíbe a inclusão de outros elementos definidos. Por exemplo, o perfil EDOC não seleciona os elementos relacionados a casos de uso, o que não exclui seu uso se necessário;
- Estereótipos definidos que representem metaclasses usadas na descrição no domínio do problema. Por exemplo, o estereótipo «beans» pode fazer parte de um perfil Java do meta-modelo de referência UML;
- Definições de *tagged values* contendo seus nomes, os tipos de valores, as descrições semânticas e suas restrições;
- Restrições à semântica de determinados elementos da referência. Por exemplo, em um perfil Java, a metaclasse Class teria uma restrição que impede a herança múltipla;
- Descrições em linguagem natural dos elementos pertencentes ao perfil;
- Extensões *heavyweight* definidas pelo modelo MOF, como novas metaclasses;
- Notação usada pelo perfil, como ícones para estereótipos;
- Regras de transformação, validação e apresentação.

3.2 Objetivos do Perfil UML 2.0 de Testes

Historicamente, o desenvolvimento da linguagem UML teve foco principalmente na definição da estrutura de sistemas e de seu comportamento. No entanto, a evolução de abordagens baseadas em modelos, como a própria arquitetura dirigida a modelos (MDA) da OMG, criou uma necessidade de integração de atividades de teste de conformidade (ZANDER et al., 2005). Essa integração deveria permitir a especificação de modelos que capturassem as informações necessárias para realizar testes do tipo caixa-preta em implementações do sistema.

Para realizar essa integração entre atividades de teste e modelos de sistema, foi desenvolvido pela OMG o Perfil UML 2.0 de Testes (*UML 2.0 Testing Profile*, U2TP). U2TP define uma linguagem para projetar, visualizar, especificar, analisar, construir e documentar os artefatos de teste de sistemas (OBJECT MANAGEMENT GROUP, 2005). É uma linguagem de modelagem de testes que pode ser usada com tecnologias de componentes e linguagens orientadas a objeto, aplicadas em diversos domínios de aplicação. Os seguintes princípios de projeto foram usados:

- *Integração com UML*: o perfil é baseado no meta-modelo UML, usando os mesmos princípios e elementos dessa linguagem;
- *Reuso*: U2TP usa, onde possível, conceitos e elementos já presentes em UML. Novos elementos só foram adicionados ao se mostrarem de relevância central para a área de teste durante o desenvolvimento do perfil.

O perfil foi projetado para apoiar o teste de implementações de sistema de maneira eficiente e tão automatizada quanto possível. Seu foco é o teste funcional caixa-preta, ou seja, verificar o comportamento de entrada e saída de um alvo cuja estrutura interna fica idealmente escondida. Esse sistema sob teste (*system under test*, SUT) não é especificado como parte dos diagramas descritos usando U2TP: os elementos são importados a partir do modelo completo. O SUT é acessado a partir de suas interfaces públicas. No entanto, se o modelo de sistema define e oferece acesso a interfaces de subsistemas e componentes internos, esses também podem ser acessados em um teste: cada componente pode ser considerado um SUT menor, de escopo mais restrito. Isto permite a descrição usando U2TP de testes em todos os níveis, não apenas de integração como também unitários.

É importante salientar que, seguindo os mesmos princípios que UML, U2TP define apenas uma linguagem, e não um método (ou seja, uma linguagem somada a um processo). Apenas é oferecida uma notação, e não instruções fixas de como usá-la. Desta forma, os usuários do perfil ficam livres para definir metodologias próprias, adaptadas às suas necessidades e domínios de aplicação.

O perfil é organizado em diferentes grupos de conceitos. Cada um desses grupos oferece elementos e relações a serem usadas na descrição de uma atividade de testes. Eles são detalhados na próxima seção.

3.3 Grupos de conceitos

A arquitetura do Perfil UML 2.0 de Testes é dividida em quatro pacotes, chamados grupos de conceitos. São eles:

- *Arquitetura de teste*, contendo elementos para a descrição da estrutura e da configuração de um teste e os relacionamentos entre eles e entre o teste e o SUT;
- *Comportamento de teste*, dirigido às observações e atividades executadas durante uma campanha de teste;
- *Dados de teste*, para definir sintaxe e semântica dos valores de entrada e saída processados;
- *Temporização de teste*, com elementos para especificar restrições de tempo durante a execução.

A Tabela 3.1, adaptada de Zander *et al* e atualizada em relação à especificação final de U2TP, sumariza os elementos de cada um dos grupos (ZANDER et al., 2005; OBJECT MANAGEMENT GROUP, 2005).

Tabela 3.1: Sumário dos conceitos de U2TP

Arquitetura	Comportamento	Dados	Temporização
SUT	Veredito	Coringa	Temporizador
Componente de Teste	Ação de Validação	Repositório de Dados	Fuso Horário
Contexto de Teste	Padrão	Partição de Dados	
Árbitro	Registro de Teste	Seletor de Dados	
Escalonador		Regras de Codificação	

Os itens da Tabela 3.1 serão detalhados a seguir.

3.3.1 Arquitetura de teste

O grupo de *arquitetura de teste* define os conceitos necessários para descrever os componentes usados em uma atividade de teste e as relações entre eles. É focado nos aspectos estáticos usados durante a verificação de um sistema, como a identificação do sistema sob testes (*system under test*, SUT), que provê operações através de interfaces públicas acessadas por componentes de teste (*test components*), agrupados em contextos de teste (*test contexts*). Define também um árbitro (*arbiter*), que determina o veredito final do teste, e um escalonador (*scheduler*), responsável por controlar a execução.

Os elementos UML deste grupo estão representadas na Figura 3.2, adaptada da especificação do perfil.

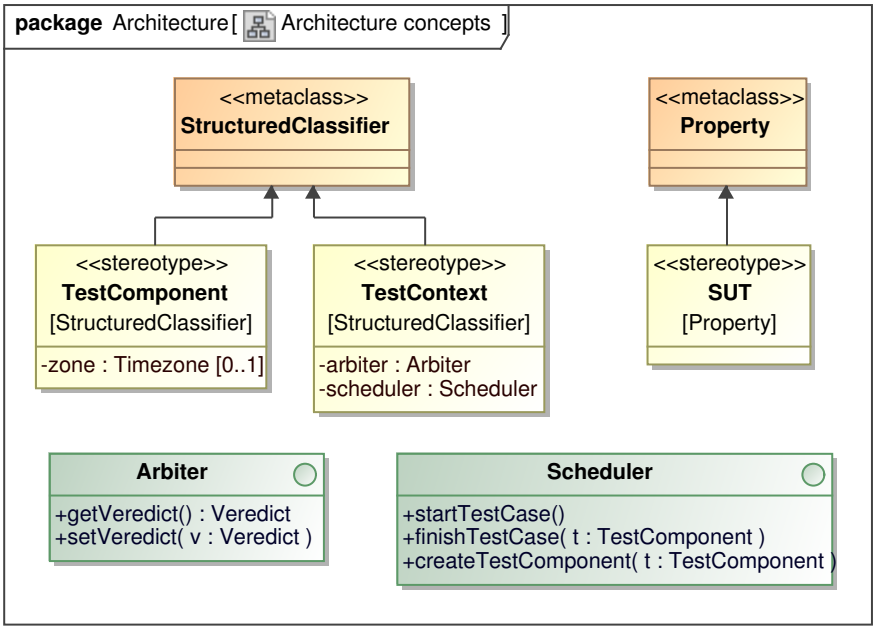

Figura 3.2: Arquitetura de teste, adaptado de (OBJECT MANAGEMENT GROUP, 2005)

***System under test* (SUT):** O sistema sob testes é ativado através de um conjunto de interfaces públicas que expõe suas funcionalidades. Essas interfaces são importadas

a partir do modelo completo do sistema. Como os componentes de teste se comunicam com o alvo apenas através delas, não existem outras maneiras de obter informação do SUT, caracterizando um teste tipo caixa-preta. Um SUT pode se apresentar através de diferentes níveis de abstração: um sistema completo, um subsistema ou um conjunto de componentes, como classes: a definição de sua granularidade define o nível de teste, como unitário ou de integração.

***Test Component*:** O componente de teste representa a implementação de um caso de teste: é normalmente uma classe que interage com o SUT ou com outros componentes. Ele executa uma seqüência de ações na forma de estímulos, ativando funcionalidades do alvo, e de observações, recuperando informações. O componente pode realizar validações, verificando os dados recebidos do SUT e atualizando o veredito do teste, bem como registrar informações em *logs*.

***Arbiter*:** O árbitro é uma interface cuja implementação tem como propósito determinar o veredito final para o teste. Cada componente de teste informa a um árbitro central os seus vereditos locais. O árbitro central determina a política para o veredito final: por exemplo, indicar o teste inteiro como falho se uma certa percentagem dos casos de teste não terminar com sucesso.

***Scheduler*:** O escalonador é uma interface cuja implementação controla a execução dos diferentes componentes de teste. Ele realiza a criação e destruição dos componentes, inicia sua execução e interage com o árbitro para informar o momento de calcular o veredito final do teste.

***Test Context*:** Um contexto de teste é uma estrutura usada para agrupar componentes de teste. Ele é uma classe associada a um conjunto de componentes de teste, a uma instância de árbitro, uma instância do SUT e uma instância de um escalonador. Sua implementação pode ser tanto código executável como *scripts*. O elemento contexto de teste foi projetado para ser gerado automaticamente por ferramentas, usando as descrições dos demais elementos.

3.3.2 Comportamento de teste

O grupo de *comportamento de teste* define os conceitos relacionados à aspectos dinâmicos dos procedimentos, como invocação de ações, determinação de vereditos e registros (*logs*). A este grupo de conceitos pertencem também elementos que associam casos de uso a casos de teste, documentando o comportamento que o SUT deve seguir.

Os elementos UML deste grupo estão representadas na Figura 3.3, adaptada da especificação do perfil.

***Veredict*:** Um veredito é uma enumeração que contém ao menos os valores *fail*, *inconclusive*, *pass* e *error*, que indicam o resultado da execução de um teste. O valor *pass* indica que o SUT se comportou como o previsto. *Fail*, por outro lado, indica que o comportamento do SUT mostrou-se fora do esperado. Se a execução do teste não pôde determinar um resultado, o valor *inconclusive* é usado. O veredito *error* é usado para indicar que o sistema de teste, e não o seu alvo (o SUT), teve problemas.

***Validation Action*:** Uma ação de validação é uma ação executada durante um teste que informa um resultado local ao árbitro central.

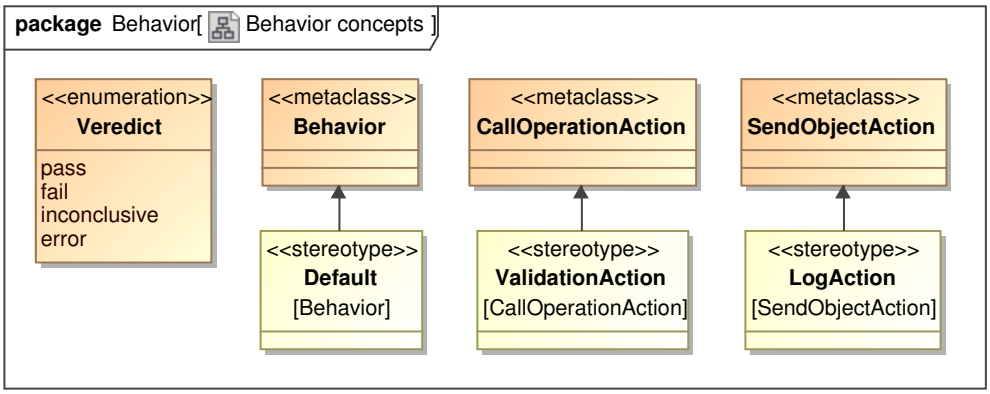

Figura 3.3: Comportamento de teste, adaptado de (OBJECT MANAGEMENT GROUP, 2005)

***Default*:** Uma especificação *default* (padrão) é um artifício usado para simplificar os modelos de teste, permitindo construir definições parciais de componentes de teste de maneira compacta. Uma especificação default, que pode ser apresentada como diagramas de seqüência ou de estado, indica qual ação o sistema de teste deve tomar quando o SUT não se comporta da maneira esperada. Como essas especificações podem ser referenciadas por diferentes componentes, os modelos se tornam mais compactos.

***Test Log*:** O registro de teste é um componente através do qual os contextos de teste podem criar rastros de execução e indicar informações de interesse. Esses registros tornam-se parte da especificação do sistema de teste.

3.3.3 Dados de teste

O grupo de *dados de teste* define a sintaxe e semântica dos dados usados como entrada e saída dos procedimentos de teste. Dados explícitos e classes de equivalência formam conjuntos de dados (*data sets*); essas classes de equivalência especificam regras para formação de dados de entrada ou saída. Valores coringas (*wildcards*), em adição aos já presentes em UML padrão, são oferecidos. Os elementos seletores de dados (*data selectors*) são usados para facilitar a criação de estratégias de teste.

Os elementos UML deste grupo estão representadas na Figura 3.4, adaptada da especificação do perfil.

***Wildcards*:** Os coringas são valores literais usados para especificar de forma relaxada um dado fornecido a um SUT ou recebido dele durante um teste. A linguagem UML 2.0 oferece o elemento *LiteralNull*, indicando a omissão de um valor. U2TP estende o conceito fornecendo os valores *LiteralAny* e *LiteralAnyOrNull*.

***Data Partition*:** Uma partição de dados estabelece uma classe de equivalência para valores a serem fornecidos a um SUT ou recebidos durante um teste. É usada como uma forma de diferenciar visivelmente os dados, atribuindo a eles nomes como, por exemplo, *NumerosCPFValidos*.

***Data Pool*:** Um repositório de dados é o mecanismo usado para associar conjuntos de dados a contextos de teste. É uma classe que contém partições de dados ou valores explícitos e uma associação a um contexto de teste.

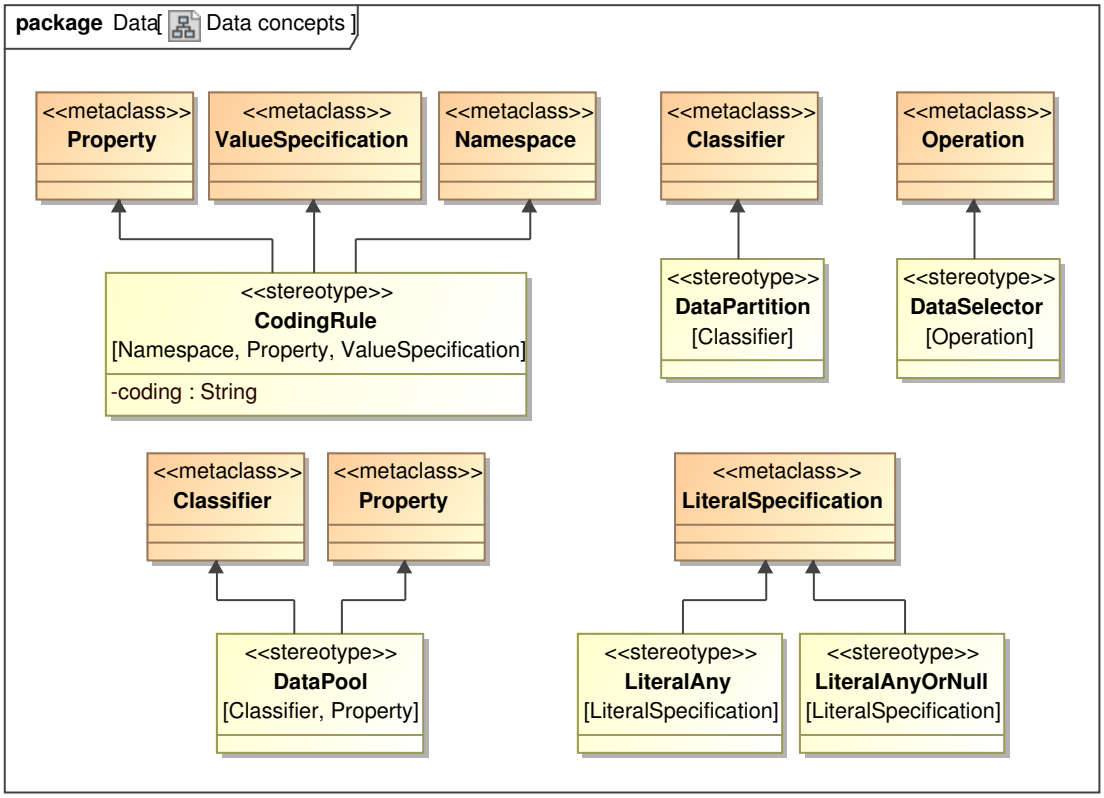

Figura 3.4: Dados de teste, adaptado de (OBJECT MANAGEMENT GROUP, 2005)

***Data Selector*:** Tipicamente, seletores de dados são operações que definem as estratégias usadas pelo sistema de teste para fornecer e verificar os dados de entrada e saída do SUT. São associados a uma partição ou repositório de dados.

***Coding Rules*:** As regras de codificação são *strings* usadas para referenciar padrões de codificação e de protocolos de transmissão de dados externos ao perfil de teste, como CORBA ou XML.

3.3.4 Temporização de teste

O grupo de *temporização de teste* define os conceitos de tempo usados para sincronização entre sistemas e para verificação de desempenho. Temporizadores (*timers*) são definidos, permitindo especificar limites de tempo para operações em um caso de teste. O conceito de fuso horário (*timezones*) agrupa componentes de teste sob uma mesma percepção de tempo.

Os elementos UML deste grupo estão representadas na Figura 3.5, adaptada da especificação do perfil.

***Timer*:** Um temporizador é uma interface cujas implementações são usadas por componentes de teste para verificar o tempo decorrido entre dois pontos de execução. Um objeto inicializa um temporizador indicando um período de tempo; quando esse período expira, o temporizador envia uma mensagem para o objeto. Uma especificação de *default* pode fazer, por exemplo, que o teste termine com o veredito *fail* se isto acontecer (*timeout*). U2TP define uma notação especial para o início e parada de temporizadores em diagramas de seqüência, na forma de uma ampulheta e de um “X”.

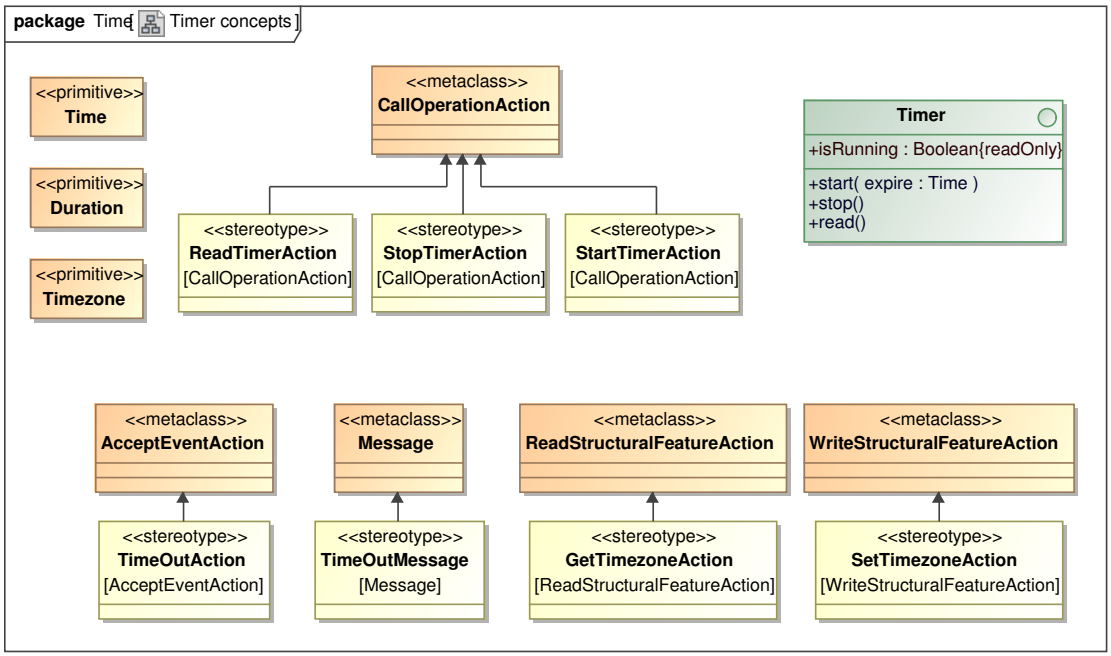

Figura 3.5: Temporização de teste, adaptado de (OBJECT MANAGEMENT GROUP, 2005)

***Timezone*:** O fuso horário é o mecanismo usado por U2TP para agrupar e sincronizar diferentes componentes de teste, fazendo com que todos tenham a mesma percepção de tempo.

3.4 Conclusões

Os conceitos e elementos oferecidos pelo Perfil UML 2.0 de Testes permitem a um engenheiro de teste projetar e descrever atividades de verificação e validação de sistemas em uma linguagem padronizada, desenvolvida com foco em metodologias MDA e em automação. U2TP estende a linguagem UML padrão com elementos específicos a essas atividades de teste.

No entanto, o perfil originalmente não compreende as técnicas de injeção de falhas, usadas em atividades de validação de sistemas com foco em mecanismos de tolerância a falhas. A falta uma linguagem padrão para descrever os componentes e conceitos usados por essas técnicas dificulta o compartilhamento de modelos e diagramas entre equipes, bem como a criação de ferramentas de automação que os processem para gerar artefatos como arquivos de configuração ou mesmo código-fonte.

Para remediar a falta desta linguagem padronizada, este trabalho propõe U2TP-FI, uma extensão de U2TP para a descrição de atividades de teste que usem técnicas de injeção de falhas. Os objetivos, conceitos e elementos de U2TP-FI são descritos no próximo capítulo.

4 EXTENSÃO PARA INJEÇÃO DE FALHAS

Apesar de a notação oferecida pelo Perfil UML 2.0 de Testes permitir o uso de uma mesma linguagem na comunicação entre diferentes equipes e projetos de conceitos de teste e diagramas que os descrevam, ela não oferece suporte a técnicas de injeção de falhas. O usuário do perfil pode criar uma notação própria para este objetivo; no entanto, a especificação de um modelo de teste que use essas técnicas não será feita de maneira padronizada, dificultando a comunicação e reintroduzindo os problemas que U2TP deveria afastar.

A abordagem natural para descrever testes que usem injeção de falhas é a extensão do perfil U2TP através da inclusão de componentes para esse fim. Como U2TP já oferece os elementos necessários para a descrição de atividades de verificação e validação de sistemas, não faz sentido criar um perfil à parte. Estender U2TP, incluindo conceitos e componentes para descrever atividades de injeção de falhas, permite um reuso tanto de modelos como de ferramentas já existentes: elementos são apenas adicionados a eles, e não modificados.

A extensão proposta neste trabalho, U2TP-FI (*UML 2.0 Testing Profile with Fault Injection*), tem como objetivo oferecer elementos que permitam a um engenheiro de teste descrever atividades de verificação e validação de sistemas usando técnicas de injeção de falhas.

Este capítulo inicia com a Seção 4.1, que discute os princípios de projeto usados para nortear o desenvolvimento de U2TP-FI. A Seção 4.2 detalha o objetivo principal que a extensão deve atingir e lista seus objetivos secundários. Questões encontradas durante o desenvolvimento são discutidas na Seção 4.3. A arquitetura geral de U2TP-FI é mostrada na Seção 4.4, e os grupos de conceitos e elementos que compõem a extensão são detalhados na Seção 4.5.

4.1 Princípios de projeto

Com o objeto de tornar a adoção de U2TP-FI o mais suave possível por uma equipe de testes já existente, com modelos e metodologias de teste já adotadas, o desenvolvimento da extensão teve como foco os princípios de *reuso* e de *integração*. U2TP-FI, por se tratar de uma extensão a um perfil já existente, não deveria desestabilizar um ambiente de análise e modelagem preexistente e baseado em U2TP no processo de sua adoção.

Os princípios de projeto adotados foram os seguintes:

- *Reuso de ferramentas.* Um ambiente de modelagem usa ferramentas CASE para a geração dos diagramas U2TP necessários no desenvolvimento. Os elementos

definidos por U2TP-FI devem usar mecanismos de extensão UML amplamente disseminados nessas ferramentas.

- *Reuso de conceitos.* Os conceitos originais usados em U2TP não devem ser modificados, evitando a alteração de metodologias, modelos e ferramentas existentes.

- *Integração com modelos de teste existentes.* Nenhum elemento original de U2TP deve ser alterado ou retirado; U2TP-FI deve apenas oferecer novos componentes para seus usuários. Modelos de teste existentes podem ser estendidos com elementos de injeção de falhas, sem necessitar de alterações mais profundas.

- *Integração com iniciativas de modelagem existentes.* Os elementos de U2TP-FI devem ser facilmente integrados com iniciativas de modelagem de injeção de falhas encontradas na literatura, como o sistema de padrões descrito por Leme, Martins e Rubira, descrito na Seção 2.5.

- *Fácil transformação de modelos.* Ferramentas de transformação de modelos devem ser facilmente adaptadas para produzir artefatos de teste, como arquivos de configuração, a partir de diagramas que usem U2TP-FI.

4.2 Objetivos de U2TP-FI

O objetivo principal da extensão para injeção de falhas do Perfil UML 2.0 de Teste é o estabelecimento de uma *linguagem de descrição de modelos para atividades de injeção de falhas*. Os elementos oferecidos por U2TP-FI devem permitir a um desenvolvedor identificar facilmente as características de injeção de falhas em um modelo UML, bem como possibilitar a transformação desses por ferramentas.

Com base nos princípios de projeto descritos na Seção 4.1, os seguintes objetivos secundários foram estabelecidos:

- *Independência de plataforma.* Para possibilitar a integração com ferramentas de injeção e de teste, deve ser possível modelar conceitos de injeção de falhas sem associar o modelo de testes a um injetor ou artefato específico.

- *Documentação de atividades de teste.* A descrição de características do teste com injeção de falhas usando a linguagem UML tem como objetivo facilitar a transmissão de conhecimento na equipe, impedindo que sejam ocultadas em artefatos e documentos paralelos ao modelo de testes.

- *Consistência entre diferentes artefatos de teste.* Um modelo usando U2TP-FI deve suportar a descrição de todos os atributos necessários para a execução da campanha de injeção de falhas. Desta forma, artefatos gerados automaticamente a partir do modelo de testes, como casos de teste e arquivos de configuração, permanecem com seus dados consistentes, sem conflitos.

- *Automação de tarefas repetitivas.* Deve ser possibilitado o reuso de elementos modelados, como classes e instâncias que descrevam as falhas a serem injetadas, para que não aconteça retrabalho na modelagem. Da mesma forma, a integração com ferramentas de automação de testes deve ser levada em conta.

4.3 Questões de projeto

A arquitetura geral de U2TP-FI foi inspirada no perfil UML para especificação de qualidade de serviço (QoS) e mecanismos de tolerância a falhas (*UML Profile for Modeling Quality of Service and Fault Tolerance Characteristics and Mechanisms*, o Perfil QoS) (OBJECT MANAGEMENT GROUP, 2006) devido às similaridades na necessidade de especificar condições de funcionamento para componentes de um sistema e restrições para mensagens trocadas entre seus objetos. O Perfil QoS tem como objetivo especificar a qualidade mínima aceita na interação entre componentes e na execução de um sistema; o perfil para injeção de falhas pode ser visto como o inverso desse objetivo, especificando a "falta de qualidade" que deve existir nos componentes usados durante a execução do sistema sob testes.

O Perfil QoS não se propõe a especificar os mecanismos usados para a garantia de qualidade de serviço, mas sim qual o nível desta que deve ser oferecido para o sistema. Este objetivo é similar ao proposto pela extensão para injeção de falhas de U2TP: não descrever a estrutura e implementação de um injetor de falhas, mas sim a falha que ele deve emular durante uma atividade de teste.

A característica mais visível do Perfil QoS é a associação de instâncias que descrevem parâmetros a mensagens trocadas entre classes em um diagrama de interação. O Perfil QoS especifica estereótipos para decorar elementos UML, indicando-os como características de qualidade de serviço. Esses elementos são associadas a classes e mensagens de um diagrama através de relações de dependência, indicando as características que os componentes que representam devem oferecer.

O mecanismo de *estereótipos* foi usado para o desenvolvimento de U2TP-FI. Estereótipos permitem criar novos elementos de modelagem, derivados de já existentes, com propriedades específicas a um domínio de problema. Esse mecanismo de extensão de UML é oferecido pela maioria das aplicações CASE encontradas no mercado, o que vai de encontro ao princípio de integração com ferramentas existentes. Seu uso não é incômodo, por ser bem suportado pelos sistemas CASE. As classes decoradas com um estereótipo são facilmente identificadas em um diagrama, o que vai de encontro ao objetivo de documentação das atividades de teste. Além disso, muitas ferramentas de transformação e de geração de artefatos, como AndroMDA, oferecem facilidades para a busca e a manipulação de elementos decorados, indo de encontro com o princípio de integração com ferramentas e com o objetivo de automação de tarefas (ANDROMDA.ORG, 2007).

O conceito central de U2TP-FI é a *descrição de uma falha*. Outros conceitos encontrados em uma campanha de injeção de falhas, como a carga de trabalho, os dados de entrada, a verificação de dados de saída, a monitoração e o registro de atividades são oferecidos pelo Perfil UML 2.0 de Testes original. Para descrever uma atividade de injeção, deve-se incluir no modelo de testes um elemento que represente uma falha e associá-lo a classes ou a mensagens, abstrações do local em que o erro deve ser emulado.

A estratégia para a descrição de falhas e associação dessas descrições a componentes de um sistema foi a mesma usada pelo Perfil QoS. As características de qualidade de serviço são representadas por classes e instâncias dessas classes decoradas com o estereótipo «QoSCharacteristic» e seus derivados, o que inspirou o estereótipo «FaultDescription». A quantificação dos parâmetros que compõem uma característica de QoS é feita por atributos anotados com «QoSDimension», o que inspirou o estereótipo «FaultParameter». A associação dessas características a um elemento do modelo é feita por uma relação de dependência anotada com um dos subtipos do estereótipo «QoSConstraint» («QoSRequired», «QoSContract» ou «QoSOffered»), inspirando em U2TP-FI a cri-

ação do estereótipo «InjectFault». Para questões de organização, elementos anotados com «QoSCharacteristic» podem ser agrupados em pacotes marcados com o estereótipo «QoSCategory»; em U2TP-FI, o papel de organização de elementos é feito com os estereótipos «FaultCategory» e «Faultload».

O Perfil QoS oferece também elementos para descrever a implementação de mecanismos de tolerância a falhas. No entanto, esses elementos são focados em implementações tolerantes a falhas de CORBA, em arquiteturas de *clustering* e em estratégias de replicação de dados. Como os elementos oferecidos eram demasiado específicos para esses objetivos, não foram levados em consideração para o desenvolvimento de U2TP-FI.

4.4 Arquitetura geral

A extensão U2TP-FI é composta por seis estereótipos, organizados em três grupos de conceitos. Esses estereótipos são usados para marcar classes, pacotes, relações de dependência, propriedades e instâncias em um modelo de testes, indicando as falhas a serem ativadas durante uma campanha de injeção.

O estereótipo central de U2TP-FI é «FaultDescription», que decora uma classe representando um tipo de falha. Essa falha é ativada por condições como tempo ou eventos de interesse; a parametrização dessas condições de ativação é representada por classes decoradas com o estereótipo «FaultActivation». Essa ativação é representada por uma associação entre os dois elementos. Uma classe «FaultDescription» tem as características que regem o comportamento da falha parametrizado por atributos decorados com o estereótipo «FaultParameter».

Uma relação de dependência decorada com o estereótipo «InjectFault» liga uma falha a um elemento de interesse, como um componente ou uma mensagem trocada no sistema. Os dois outros estereótipos, «Faultload» e «FaultCategory», são usados para organizar classes e instâncias de falhas em um modelo UML.

Os elementos da extensão serão descritos em detalhes na próxima seção. A arquitetura geral de U2TP-FI pode ser vista na Figura 4.1.

4.5 Grupos de conceitos

A extensão U2TP-FI organiza-se em três grupos de conceitos, cada um com objetivos diferentes. Os elementos descritos na Figura 4.1 são separados de forma a descrever aspectos distintos de uma atividade de validação de sistemas.

Os três grupos de conceito são *descrição de falhas*, *ativação de falhas* e *organização de falhas*. O grupo de descrição oferece conceitos para modelar aspectos estáticos, no caso os tipos e os parâmetros, de uma falha a ser injetada na atividade de teste. O grupo de ativação contém conceitos relacionados a aspectos dinâmicos, como as condições de interesse que iniciarão a injeção de uma determinada falha no ambiente de testes. No grupo de organização encontra-se estereótipos usados para agrupar e categorizar as falhas.

Descrição de falhas

Ferramentas de injeção oferecem aos seus usuários, os engenheiros de testes, diferentes opções de falhas que podem injetar em uma atividade de validação. Essas falhas são descritas por um *modelo de falhas*, um conjunto de regras que especificam a interação entre componentes que apresentam problemas.

Em U2TP-FI, os diferentes tipos de falhas que compõem esse modelo são representa-

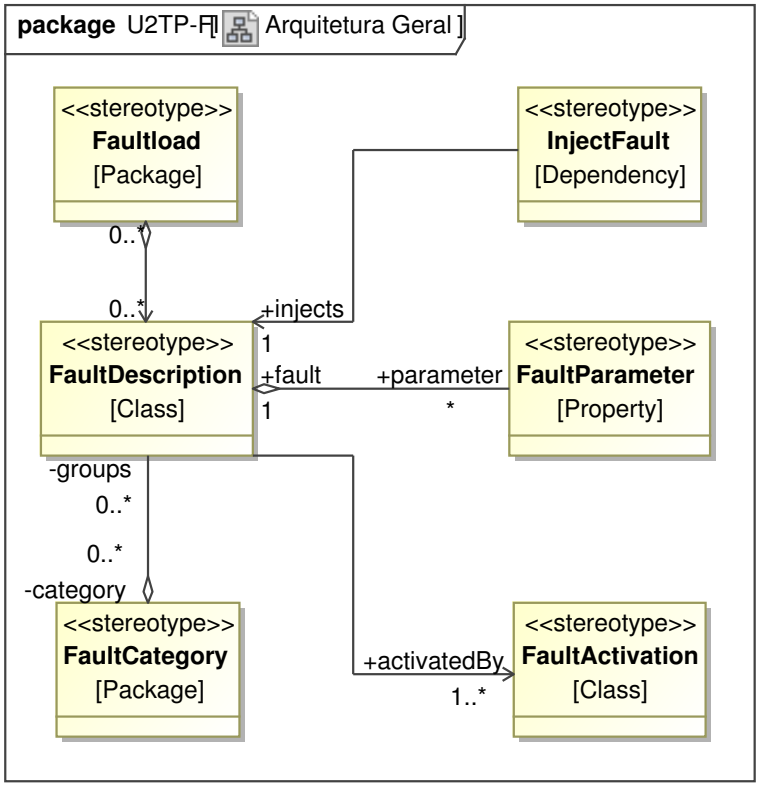

Figura 4.1: Arquitetura geral da extensão

dos por classes anotadas com o estereótipo «FaultDescription». As falhas são especificadas independentemente do elemento do sistema a que estiverem associadas, permitindo sua associação com diferentes mensagens ou objetos. Por exemplo, uma falha de memória pode ser injetada em diferentes endereços, ou mesmo em diferentes componentes (como em um *buffer* de comunicação).

Uma falha pode ter parâmetros associados, permitindo uma variação de seu comportamento. Usando o mesmo exemplo de uma falha de memória, diferentes máscaras que definem os *bit flips* podem ser aplicadas. Esses parâmetros são representados por atributos decorados com o estereótipo «FaultParameter».

Uma classe «FaultDescription» representa um tipo de falha, e não uma instância desta. Por instância de falha, entende-se um tipo de falha associado a valores de seus atributos, definindo o seu comportamento e suas características que serão emuladas em uma campanha de injeção. Em um modelo U2TP-FI, uma instância de falha é representada por uma especificação de instância (*Instance Specification*), ou seja, um objeto com valores literais preenchendo os atributos indicados na classe que determina seu tipo.

Os elementos para descrição de falhas são sumarizados na Tabela 4.1.

Um exemplo de descrição de falhas pode ser visto na Figura 4.2. A classe DatagramDroppedFault, decorada com o estereótipo «FaultDescription», representa uma falha de descarte de datagramas em uma transmissão de dados através de redes de computadores. Ela é composta por cinco atributos, parâmetros que definem seu comportamento durante o teste: *strings* que indicam *host* e porta de origem e destino do datagrama, bem como a taxa de descarte, um número em ponto flutuante. DatagramDroppedFault tem uma associação com uma classe que representa uma condição de ativação; esse aspecto é coberto na seção seguinte, *Ativação de falhas*.

Tabela 4.1: Elementos de descrição de falhas

Nome	Semântica	Restrições
«FaultDescription»	Uma classe anotada com esse estereótipo representa uma abstração de uma falha. Modela um tipo de falha que pode ser injetada em um componente do sistema.	
«FaultParameter»	Uma propriedade anotada com esse estereótipo representa um parâmetro que rege o comportamento de uma falha.	Quando uma propriedade é anotada com «FaultParameter», o elemento que inclui a propriedade deve ser anotado com «FaultDescription».

Esses elementos não podem ser usados para gerar uma carga de falhas: os atributos que definem o comportamento durante a atividade de teste não foram preenchidos com os valores necessários. Isto é feito criando instâncias das classes acima, representadas no diagrama pelos objetos unstableNetwork e sinceBeginning. Esses objetos apresentam os valores a serem usados durante a injeção de falhas.

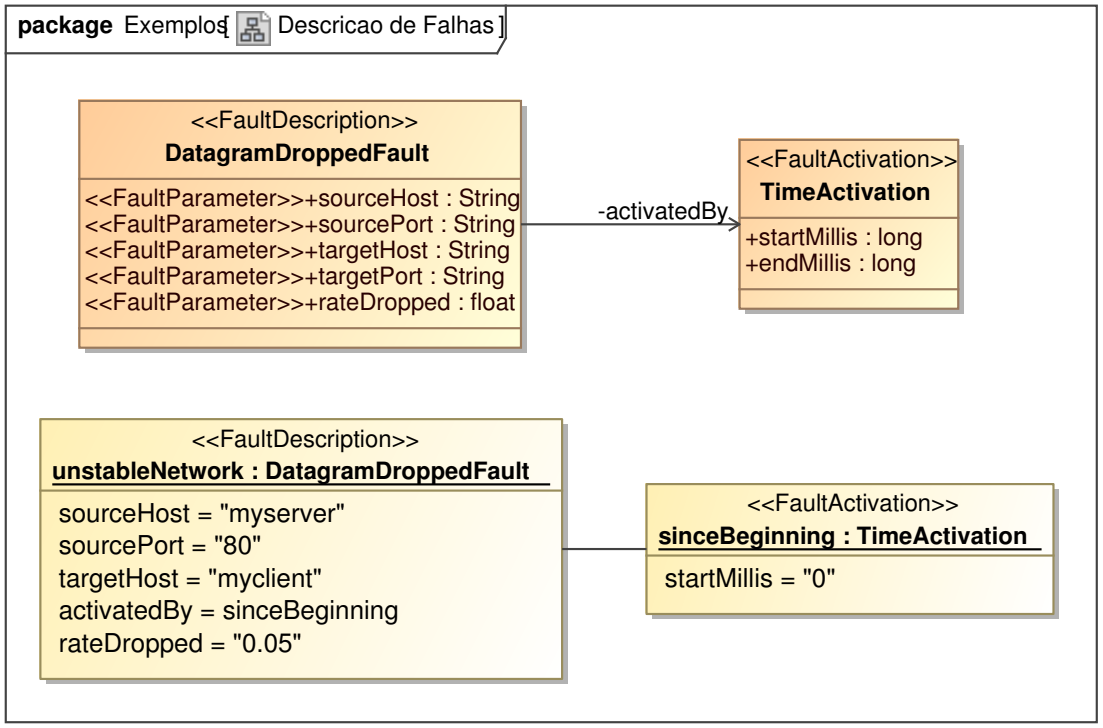

Figura 4.2: Exemplo de classe e instância de descrição de falhas

Ativação de falhas

A freqüência de ocorrência de uma falha pode não ser constante; ela pode vir a se manifestar apenas após alguma condição, como tempo decorrido, se tornar verdadeira.

De acordo com essa freqüência, podem ser classificadas como permanentes, transientes ou intermitentes (CARREIRA; SILVA, 1998).

Para representar as condições de ativação de uma falha em U2TP-FI, uma classe anotada com o estereótipo «FaultActivation» descreve as propriedades necessárias e é associada a uma descrição de falha. As condições variam de acordo com a falha descrita e com as possibilidades de configuração oferecidas pela ferramenta de injeção usada.

Uma descrição de falha, associada a uma ativação de falha, é ligada a um elemento presente no modelo do sistema através de uma relação de dependência marcada com o estereótipo «InjectFault». Esta relação indica que a falha será aplicada pela ferramenta de injeção em um determinado elemento, como um objeto ou uma mensagem enviada entre dois componentes.

Os elementos de U2TP-FI para modelar aspectos dinâmicos de injeção de falhas são mostrados na Tabela 4.2.

Tabela 4.2: Elementos de ativação de falhas

Nome	Semântica	Restrições
«FaultActivation»	Uma classe anotada com esse estereótipo representa as condições necessárias para a ativação de uma falha.	Quando um elemento é anotado com «FaultActivation», deve estar associado a outro elemento anotado com o estereótipo «FaultDescription».
«InjectFault»	Uma relação de dependência anotada com esse estereótipo indica a associação de uma falha ao comportamento de um elemento do sistema.	O elemento fornecedor (*supplier*) de uma relação de dependência anotada com «InjectFault» deve ser anotado com «FaultDescription».

Exemplos de ativação de falhas podem ser vistos nas Figuras 4.2 e 4.3. Na primeira figura, a classe TimeActivation modela uma condição de ativação por tempo decorrido desde o início do experimento de injeção de falhas, apresentando os atributos startMillis e endMillis. Em um experimento, a falha ligada a essa classe tem definido o tempo inicial a partir do qual pode ser ativada e o tempo final, depois do qual não poderá mais ser emulada no ambiente. TimeActivation é instanciada no objeto sinceBeginning que, preenchendo o atributo startMillis com o valor zero, indica que a falha associada pode ser injetada desde o momento inicial dos testes.

A segunda figura, Figura 4.3, exemplifica o uso de uma relação de dependência «InjectFault». Classes representando um servidor e um cliente de serviços de rede estão representados no diagrama de comunicação. O servidor tenta enviar dados para o cliente, evento representado pela mensagem sendData disparada. Para indicar a injeção de uma falha de comunicação nesse envio de dados, a relação de dependência «InjectFault» liga a mensagem sendData ao objeto unstableNetwork, definido na Figura 4.2.

Organização de falhas

Um cenário de falhas é a descrição de um ambiente contendo o sistema em funcionamento, sua carga de trabalho (*workload*) e uma carga de falhas (*faultload*) que atua durante sua execução. Essa carga de falhas é normalmente especificada em um arquivo de configuração fornecido à ferramenta de injeção de falhas ao ser iniciada e contém a descrição das falhas a serem ativadas, bem como seus parâmetros. Em um modelo usando

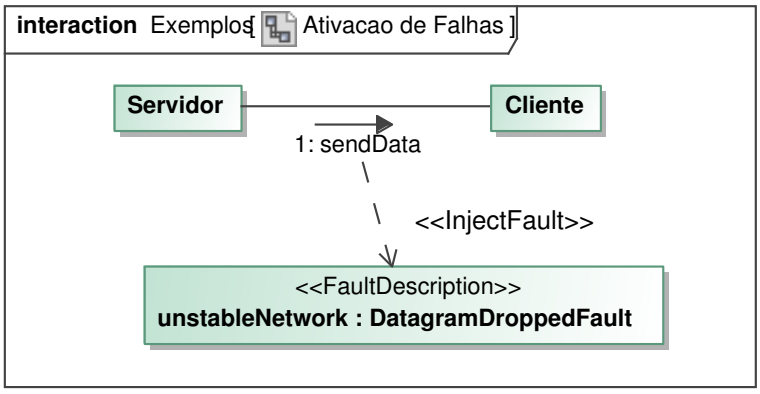

Figura 4.3: Exemplo de diagrama de comunicação com injeção de falhas na mensagem sendData

U2TP-FI, as instâncias de falhas podem ser agrupadas em um pacote anotado com o estereótipo «FaultIoad», indicando a configuração a ser fornecida à ferramenta.

Caso uma ferramenta permita a injeção de falhas de diferentes naturezas, pode ser desejável agrupá-las em categorias abrangentes para simplificar a documentação do sistema. Isto pode ser feito agrupando falhas em pacotes anotados com o estereótipo «FaultCategory», destinado à documentação de ferramentas de injeção.

Esses dois estereótipos para organização de falhas podem ser vistos na Tabela 4.3.

Tabela 4.3: Elementos de organização de falhas

Nome	Semântica	Restrições
«FaultIoad»	Representa as falhas a serem injetadas durante a operação de uma ferramenta em uma atividade de teste.	Deve estar associada a um ou mais elementos anotados com o estereótipo «FaultDescription».
«FaultCategory»	Representa os tipos de falhas oferecidas por uma ferramenta de injeção de falhas.	Deve estar associada a um ou mais elementos anotados com o estereótipo «FaultDescription».

Um exemplo de organização de falhas pode ser visto na Figura 4.4. O pacote Communication Faults, anotado com o estereótipo «FaultCategory», exibe as falhas suportadas pela ferramenta ou levadas em consideração durante o teste. No caso, ele contém apenas a classe DatagramDroppedFault, definida na Figura 4.2. O pacote My Faults agrupa as falhas que serão injetadas durante a atividade de teste. Assim, ao invés de classes, ele apresenta os objetos unstableNetwork e sinceBeginning, também definidos na Figura 4.2. Um diagrama exibindo esse pacote pode ser usado por uma equipe de testes para rapidamente observar todas as falhas a serem injetadas e definir a configuração das ferramentas a serem usadas.

4.6 Conclusões

Este capítulo apresentou U2TP-FI, uma extensão do Perfil UML 2.0 de Testes para descrição de atividades de teste que usam técnicas de injeção de falhas. U2TP-FI oferece

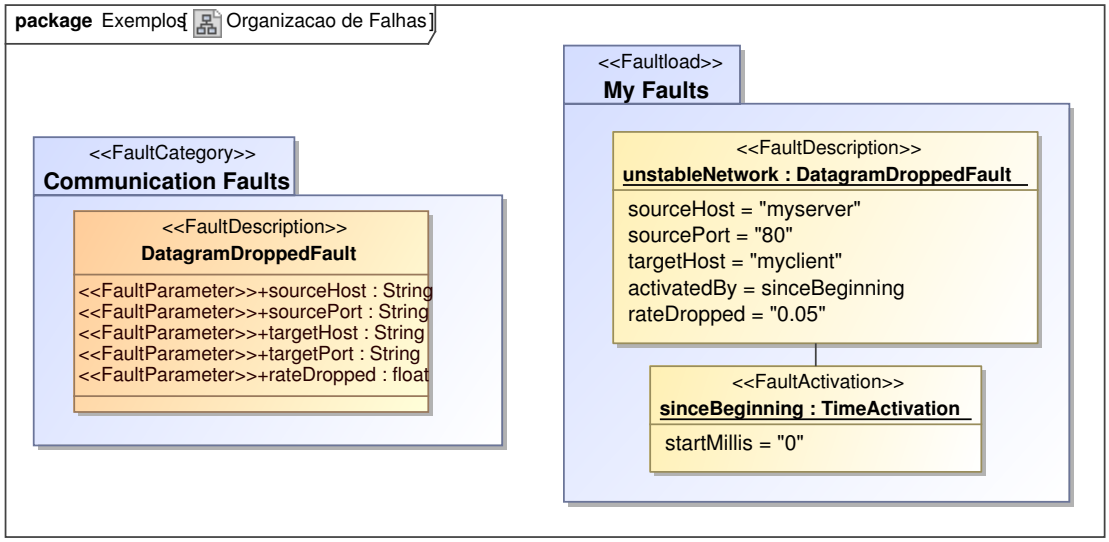

Figura 4.4: Exemplo de diagrama de classes com pacotes de categoria e de carga de falhas

conceitos e elementos para um projetista de testes na forma de uma linguagem padronizada, o que permite uma melhor comunicação entre equipes e o uso de ferramentas de automação para a geração de artefatos.

U2TP-FI é organizada em três grupos de conceitos diferentes, cada um oferecendo elementos para descrever diferentes partes de uma atividade de teste usando injetores: a descrição dos parâmetros que regem o comportamento de uma falha, a descrição das condições para sua ativação e a organização tanto do modelo de falhas adotado quanto da carga de falhas a ser injetada.

Para exemplificar o uso de U2TP-FI foram criadas provas de conceito. Sua estrutura e metodologia usada para o desenvolvimento são descritas a seguir, no Capítulo 5.

5 PROVA DE CONCEITO

Para demonstrar o uso de U2TP-FI na modelagem de testes usando técnicas de injeção de falhas, foram desenvolvidas provas de conceito. Essas provas de conceito são exemplos que envolvem a definição de modelos de teste usando a extensão proposta e a transformação destes para a geração de artefatos que podem ser usados em uma campanha de injeção de falhas.

Foram desenvolvidos três exemplos de uso de U2TP-FI, cada um destinado a demonstrar uma faceta diferente dessa extensão. Os três exemplos envolvem a transformação de modelos de teste em artefatos, como arquivos de configuração para ferramentas e relatórios para documentação.

O primeiro exemplo é a modelagem de um teste do *middleware* de computação em grade OurGrid, em que um injetor de falhas de comunicação em protocolo RMI é usado para emular o colapso de um nodo. O resultado esperado é a detecção por OurGrid dessa falha e a tomada de medidas como a interrupção do fornecimento de tarefas para este. O segundo exemplo demonstra a possibilidade de uso conjunto de múltiplas ferramentas de injeção de falhas, cada uma voltada a um protocolo de comunicação diferente, no teste de uma aplicação que implementa comunicação em grupo usando transmissão de dados via UDP e detecção de falhas via TCP. O terceiro exemplo sai do foco em geração de *faultloads* para ferramentas e demonstra o uso de U2TP-FI para a documentação de testes, através da geração de relatórios que listam as falhas injetadas em um teste e em que elementos, como classes ou mensagens, elas atuam.

U2TP-FI não define uma metodologia ou política de uso: assim como UML, é apenas uma linguagem a ser usada na representação de modelos de sistemas. No entanto, como sugestão, é descrita a metodologia usada no desenvolvimento das provas de conceito. Essa sugestão de metodologia pode ser aproveitada por usuários de U2TP-FI como base para a definição de políticas próprias para seu uso dentro de um projeto ou organização.

A Seção 5.1 discute a ferramenta usada para a criação da extensão de U2TP e dos modelos UML usados neste trabalho, bem como a ferramenta usada para a transformação desses modelos em outros artefatos de teste. A Seção 5.2 descreve a metodologia usada no desenvolvimento das três provas de conceito, estas mostradas nas Seções 5.3, 5.4 e 5.5. Este capítulo encerra com a Seção 5.6, discutindo algumas conclusões alcançadas neste trabalho.

5.1 Criação e transformação de modelos

A ferramenta usada para realizar a transformação de modelos e geração de artefatos foi AndroMDA. AndroMDA é um *framework* para a geração de artefatos a partir de modelos UML, aderindo à abordagem MDA (ANDROMDA.ORG, 2007). A ferramenta recebe

como entrada diagramas, exportados por ferramentas CASE no formato XMI (*XML Metadata Interchange*), uma linguagem XML para a troca de modelos UML padronizada pela OMG. Esses diagramas são processados por cartuchos (*cartridges*), pacotes que realizam a geração dos artefatos.

Um cartucho é um pacote contendo regras para a seleção de elementos de interesse e gabaritos (*templates*) para a geração de arquivos de saída. Para selecionar elementos UML, o principal mecanismo usado por um cartucho é a busca por estereótipos: todos os elementos do modelo que são decorados por um estereótipo específico são agrupados e processados. O processamento desses elementos de interesse é feito por gabaritos, onde atributos e propriedades são usados para preencher lacunas, resultando em arquivos de saída.

Para cada tipo de artefato a ser gerado, é criado um cartucho diferente. Por exemplo, é fornecido junto com AndroMDA um cartucho que seleciona classes marcadas com o estereótipo «ValueObject». Para cada uma dessas classes UML é gerada uma classe Java que segue o padrão de projeto *value object*: cada atributo é transformado em uma variável de instância, acessada por métodos get e set. No caso de U2TP-FI, uma metodologia sugerida para a geração de *faultloads* é a criação de um cartucho para cada injetor, compreendendo o modelo de falhas oferecido. A ferramenta AndroMDA, ao ser invocada, transformaria os modelos U2TP-FI em arquivos de configuração para o injetor específico. Assim, o cartucho poderia ser reusado entre experimentos com *faultloads* diferentes mas que usem as mesmas ferramentas de teste.

A ferramenta CASE usada para a criação de modelos UML foi MagicDraw versão 12.1, desenvolvida pela empresa No Magic, Inc. (NO MAGIC INC., 2007). MagicDraw foi escolhido por ter suporte completo à linguagem UML 2.0 e por ser usada também pela equipe de desenvolvimento de AndroMDA, o que resultou em um ótimo suporte e poucos problemas de compatibilidade. A empresa No Magic oferece uma versão gratuita da ferramenta, denominada *Community Edition*, que tem limitações em relação ao número de elementos e à quantidade de diagramas abertos em uma sessão de modelagem. Como esses limites não seriam atingidos, não mostraram-se como problemas no desenvolvimento dos exemplos de uso de U2TP-FI.

5.2 Metodologia

As provas de conceito desenvolvidas neste trabalho foram criadas usando cinco etapas distintas: criação do modelo de falhas, modelagem do caso de teste, criação do cartucho AndroMDA, geração dos artefatos de teste e execução.

Cada uma dessas etapas é ligada ou à ferramenta de injeção de falhas usada ou à campanha de teste em curso. Por exemplo, um modelo de falhas e seu cartucho AndroMDA correspondente podem ser criados apenas uma vez para cada injetor de falhas e reusado em campanhas de teste diferentes. No entanto, se o injetor a ser usado for alterado por questões quaisquer, o caso de teste permanece inalterado, mas o cartucho AndroMDA deve ser adaptado à nova ferramenta de teste.

Durante a primeira etapa, a *criação do modelo de falhas*, os objetivos do teste e a capacidade das ferramentas de injeção são analisadas e um modelo UML é criado com classes representando os tipos de falhas de interesse e os tipos de ativação suportados. O resultado dessa etapa é um pacote anotado com o estereótipo «FaultCategory». Este pacote contém as classes de falha que serão referenciadas no modelo dos casos de teste. Essas classes apresentam propriedades que são de interesse para o teste e que são suportadas

pelas ferramentas usadas.

A *modelagem dos casos de teste* é a etapa seguinte. Usando o Perfil UML 2.0 de Testes, os elementos do sistema-alvo, como classes e interfaces, são importadas para o modelo. Os casos de teste são criados como classes com comportamentos associados. Esses comportamentos são especificados por diagramas de seqüência ou de comunicação, mostrando a ordem de mensagens a serem trocadas entre os elementos, determinando as interações entre os componentes do teste. Nesses diagramas, em um primeiro momento, são usados os conceitos U2TP originais, definindo elementos e comportamentos como ações de validação, indicação de dados usados, entre os demais necessários durante uma atividade de teste.

Em um diagrama de classes à parte, são criadas instâncias de descrição e de ativação de falhas. Essas instâncias, objetos associados aos tipos definidos na etapa anterior, têm seus parâmetros definidos. Esses parâmetros definem o comportamento dessas falhas ao serem injetadas durante a campanha de teste.

Nos diagramas de seqüência que definem o comportamento e as ações dos casos de teste, são incluídos elementos que representam as falhas a serem injetadas. Usando relações de dependência decoradas com o estereótipo «InjectFault», eles são ligados aos componentes do teste que indicam os lugares em que as falhas atuarão. Por exemplo, uma falha pode ser ligada a uma classe que representa uma máquina que sofrerá colapso, ou a uma mensagem que terá seus parâmetros corrompidos.

Com os modelos UML representando as falhas e os casos de teste prontos, passa-se à próxima etapa, a *criação do cartucho AndroMDA*. Um cartucho é criado para cada injetor de teste a ser usado durante a atividade de teste. No desenvolvimento do cartucho, são criados arquivos de descrição que especificam quais são os elementos em que a ferramenta atuará. Um exemplo muito usado no desenvolvimento dos casos de teste deste trabalho é a seleção de todas os elementos UML do tipo instância que estejam anotados com o estereótipo «FaultDescription». Junto a isso são definidos arquivos de gabarito (*templates*) com esqueletos dos artefatos e marcações indicando onde as propriedades das falhas serão preenchidas.

Com o modelo de teste e a ferramenta pronta, a etapa de *geração dos artefatos* consiste na execução de AndroMDA, fornecendo como entrada os arquivos de modelos UML e os cartuchos desenvolvidos. Essa geração pode ser automatizada através de *scripts* de preparação e compilação (*build*); por exemplo, nos mesmos scripts que realizam uma compilação de casos de teste que usem o framework JUnit, a ferramenta AndroMDA pode ser invocada, gerando os arquivos de configuração do injetor.

Os arquivos gerados são usados na etapa seguinte, a *execução do teste*. Para realizá-la, é definido o ambiente experimental de execução, contendo componentes como os casos de teste, os injetores, os arquivos de configuração e as estratégias de monitoração e registro. A correta especificação desse ambiente permite uma repetição confiável dos experimentos.

Nas próximas seções, os exemplos criados neste trabalho para mostrar o uso de U2TP-FI serão descritos. Em cada um deles, as etapas da metodologia descrita nessa seção serão indicadas.

5.3 Exemplo 1: colapso de nodo OurGrid

O primeiro exemplo de aplicação de U2TP-FI é a modelagem de um caso de teste de OurGrid, proposto pelos autores do injetor de falhas FIRMI (VACARO; WEBER, 2006).

O sistema sob testes é o *middleware* de computação em grade OurGrid, que coordena a execução de aplicações paralelas *bag-of-tasks* (ANDRADE et al., 2003). OurGrid usa o protocolo RMI para comunicação entre seus componentes, tornando apropriado o uso da ferramenta de injeção de falhas FIRMI. O foco de FIRMI é o teste de aplicações Java que usam esse protocolo, permitindo a injeção de falhas de colapso, de temporização e mesmo bizantinas, além de oferecer facilidades como a integração com o framework JUnit.

Um ambiente OurGrid é formado por três componentes. As máquinas disponíveis na grade possuem um *agente*, que é o componente responsável pela execução de tarefas. Esses nodos são agrupados em domínios administrativos organizados por uma máquina denominada *peer*, que atua como porta de entrada, provendo e anunciando os agentes disponíveis para execução. Os *peers* são contatados pela máquina do usuário. Essa máquina do usuário executa o componente *MyGrid*, responsável pela coordenação e escalonamento de tarefas.

Um dos casos de teste criados pelos autores de FIRMI foi a verificação da capacidade de OurGrid de perceber a queda de um agente ao listar as máquinas disponíveis. Essa listagem é feita pelo comando `peer status`. Ao ser executado, os *peers* chamam o método remoto `ping` dos agentes conhecidos, e os *hostnames* dos agentes que respondem são retornados ao usuário.

O caso de teste de OurGrid usando FIRMI injeta uma falha de colapso entre um *peer* e um agente da grade em execução na máquina `bentley`. Essa falha só é ativada (ou seja, passa a ser injetada no ambiente) após a segunda chamada a `ping`. Isso permite verificar que o agente está disponível antes da falha e que o *peer* não reporta mais essa disponibilidade depois da ocorrência do colapso. O resultado esperado do teste é a listagem de `bentley` na primeira chamada a `peer status`; uma segunda chamada a esse comando não indicaria mais a máquina.

A seguir são mostradas as etapas percorridas na aplicação da metodologia sugerida na Seção 5.2.

Criação do modelo de falhas

O primeiro passo foi modelar classes que representam as falhas que são de interesse no teste e que são oferecidas pelas ferramentas, decoradas com o estereótipo «FaultDescription». Da mesma forma, foram modeladas classes que representam as condições de ativação, decoradas com «FaultActivation». O diagrama da Figura 5.1 mostra os elementos que foram criados, durante o desenvolvimento deste exemplo de U2TP-FI, nesta primeira etapa. Foram representadas falhas de colapso (`CrashFault`) e de temporização (`DelayFault`), ambas especializações de `CommunicationFault`, que tem atributos para indicar os *hostnames* envolvidos. Essa classe tem uma associação a `CommunicationFaultActivation`, que representa as condições que podem ser usadas por FIRMI para ativar uma falha: a partir de um tempo determinado (`TimeActivation`) ou a partir de um número de invocações a um método específico (`InvocationActivation`).

Modelagem do caso de teste

Para desenvolver a segunda etapa da metodologia sugerida neste trabalho, o caso de teste foi modelado como um diagrama de seqüência mostrando a interação entre quatro elementos. A classe `BentleyCrashedTestCase` inicia as operações, chamando duas vezes comando `peer status` de um *peer* OurGrid. O *peer* invoca o método `ping` de um agente representado pelo elemento `bentley`. Essa última mensagem é

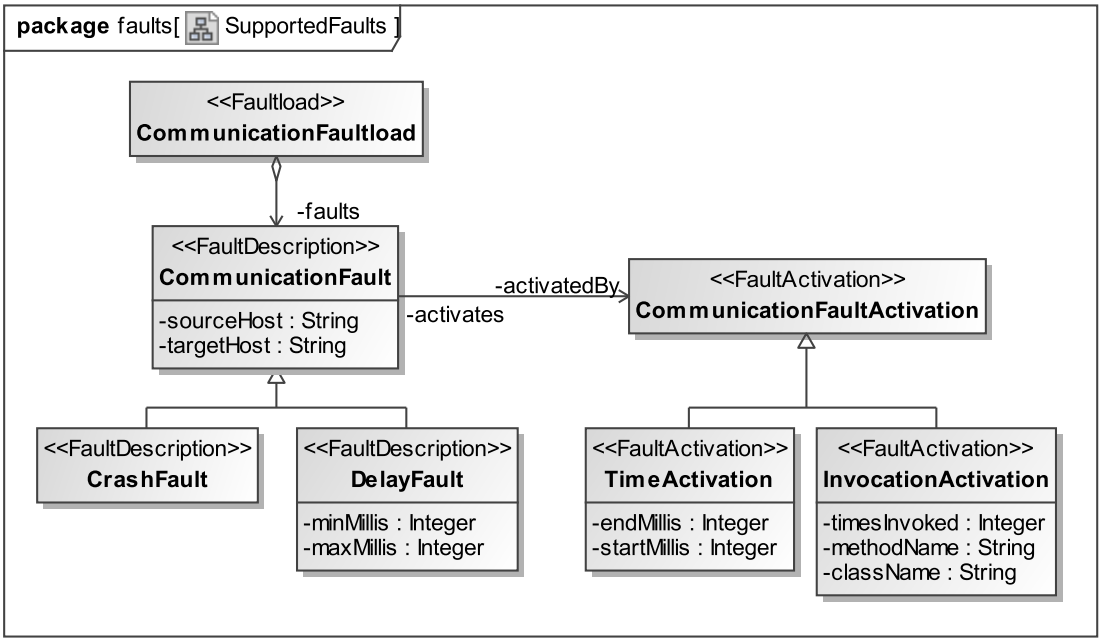

Figura 5.1: Modelo de falhas criado, referente ao injetor FIRMI

associada através de uma relação de dependência «InjectFault» a uma falha identificada por `bentleyCrashed`. O diagrama pode ser visto na Figura 5.2.

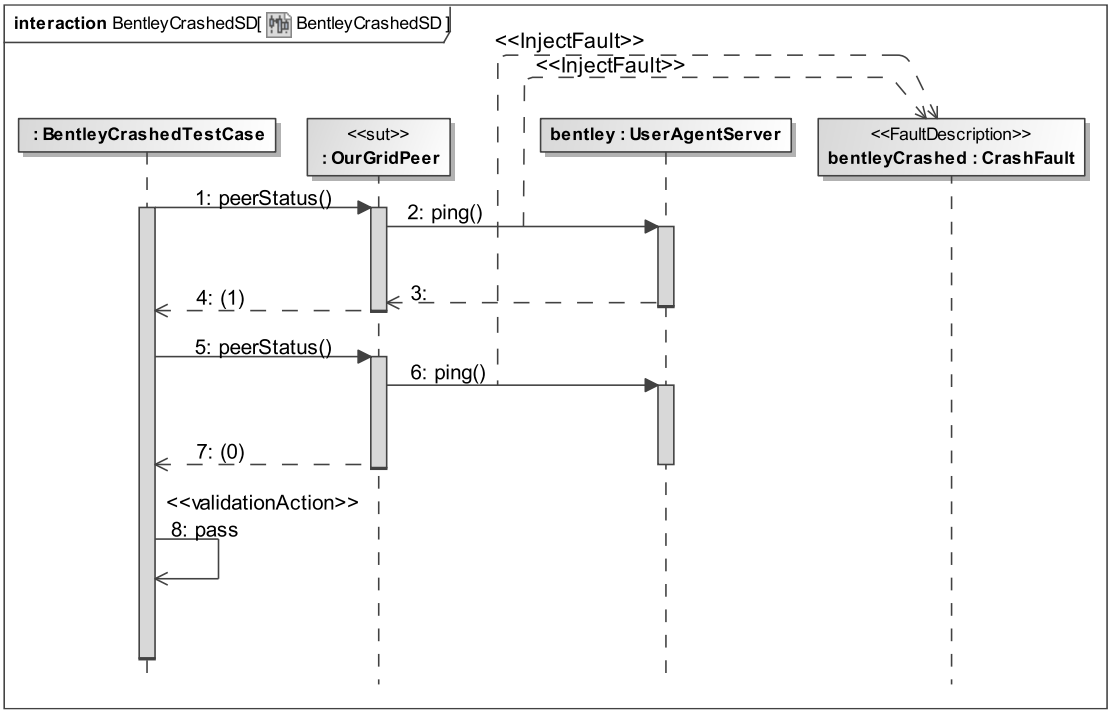

Figura 5.2: Caso de teste para avaliação de OurGrid

Um diagrama de classes auxiliar foi usado para especificar os parâmetros das falhas e das ativações usadas no caso de teste. Uma instância de `CrashFault`, chamada

`bentleyCrashed`, mostra que a falha ocorre para comunicações que envolvam a máquina de nome `bentley`. As condições de ativação são especificadas em uma instância de `InvocationActivation`, associada a ela. Os atributos indicam que a falha será injetada a partir da segunda invocação do método `ping`, pertencente à classe remota `UserAgentServer`, o agente OurGrid. O diagrama pode ser visto na Figura 5.3.

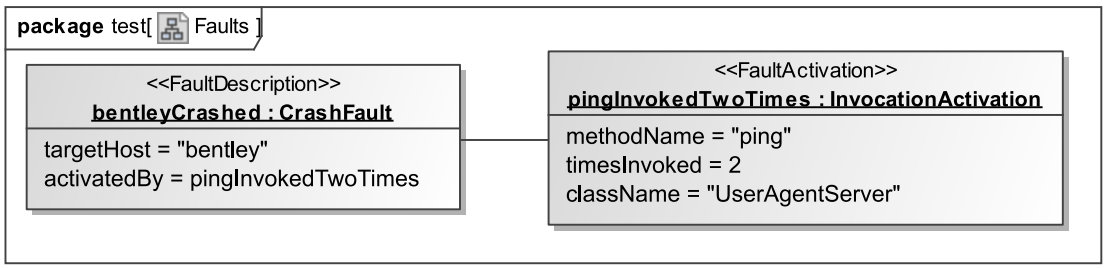

Figura 5.3: Especificação das instâncias de falhas

Criação do cartucho AndroMDA

Com o modelo pronto, o passo seguinte foi criar um cartucho AndroMDA para processar os diagramas UML e extrair a carga de falhas (*faultload*) para os testes. No caso do injetor FIRMI, a carga de falhas pode ser representada tanto por arquivos de configuração em XML descrevendo as falhas, quanto por classes Java que manipulam o estado do injetor e, portanto, da comunicação. Os arquivos XML são de leitura e manipulação mais fácil. Por outro lado, representar a *faultload* por classes Java permite o uso de FIRMI em conjunto com o *framework* de teste JUnit.

Para gerar a carga de falhas para FIRMI, dois cartuchos AndroMDA foram criados: `firmi-xml` para a geração de arquivos de configuração XML e `firmi-java` para classes Java. O desenvolvimento de dois cartuchos tem como objetivo mostrar que U2TP possibilita um reuso de modelos ao permitir gerar artefatos diferentes para os mesmos elementos.

Os dois cartuchos estão atrelados ao modelo de falhas especificado na Figura 5.1. Os modelos UML de entrada são varridos à procura de instâncias (elementos UML *Instance Specification*). Para cada instância é verificada se ela corresponde às classes `CrashFault` ou `DelayFault`. Os atributos do elemento são armazenados em variáveis internas, bem como os atributos da instância de `CommunicationFaultActivation` associada.

As variáveis com os atributos extraídos dos modelos UML são usadas por *templates* que geram os arquivos de saída. No cartucho `firmi-xml`, o *template* é um esqueleto de arquivo XML; no cartucho `firmi-java`, um esqueleto de classe Java com chamadas para ativar os filtros para comunicação RMI do injetor FIRMI.

Geração dos artefatos de teste

A quarta etapa no desenvolvimento deste exemplo foi a geração dos artefatos de teste através do processamento do modelo de teste pela ferramenta AndroMDA, usando os cartuchos criados, voltados aos injetores. AndroMDA foi ativado através de um *script* Ant, recebendo como entrada o arquivo com modelo UML e o diretório de saída para os artefatos gerados. A saída do cartucho `firmi-xml` pode ser conferida na Figura 5.4, e a saída do cartucho `firmi-java`, na Figura 5.5.

```
<XMLFaultload>
  <TriggerGroup>
    <Trigger id="pingInvokedTwoTimes">
      <Counter limit="2">
        <Condition class="UserAgentServer" method="ping"/>
      </Counter>
    </Trigger>
  </TriggerGroup>
  <FaultGroup>
    <CrashFault id="bentleyCrashed"
                activation="pingInvokedTwoTimes"
                host="bentley"/>
  </FaultGroup>
</XMLFaultload>
```

Figura 5.4: Carga de falhas XML FIRMI para o caso de teste

```
public class FirmiFaultload_bentleyCrashed extends Faultload {
  // Fault: bentleyCrashed
  // Activated by: pingInvokedTwoTimes
  private CrashFault fault = new CrashFault("bentley");

  // Invocation Activation: pingInvokedTwoTimes
  private long timesInvoked_pingInvokedTwoTimes = 0;

  public void beforeExecuting(Remote obj, Method method,
                              Object[] params) {
    if (method.getName().equals("ping")) {
      timesInvoked_pingInvokedTwoTimes++;
    }
    if (timesInvoked_pingInvokedTwoTimes == 2) {
      RMIRequestFilter.add(fault);
    }
  }
}
```

Figura 5.5: Carga de falhas Java FIRMI para o caso de teste

5.4 Exemplo 2: múltiplos injetores

O segundo exemplo de aplicação de U2TP-FI é a modelagem de um caso de teste de uma aplicação usando o *framework* de comunicação em grupo JGroups. Essa aplicação foi usada para demonstrar o uso do injetor de falhas de comunicação FIONA, cujo foco é a validação de aplicações que usam o protocolo UDP para comunicação (JACQUES-SILVA et al., 2006).

JGroups permite às aplicações efetuarem *multicast* confiável entre máquinas: ou seja, todas as máquinas participantes de uma sessão de comunicação, garantidamente, recebem todas as mensagens enviadas na mesma ordem em que foram enviadas. As máquinas são chamadas *membros de um grupo*, que constantemente se comunicam para detectar falhas como uma desconexão. JGroups permite ao programador escolher qual a estratégia de transmissão de dados e de detecção de falhas. A aplicação em questão, como era uma

demonstração de um injetor de falhas em UDP, usava *multicast* UDP para transmitir os dados entre as máquinas e, como estratégia de detecção de falhas, trocava periodicamente mensagens estilo "ping" também em UDP. A falha em receber uma resposta ao "ping" é interpretada como uma desconexão. Essa estratégia é denominada por JGroups como *detector FD*.

A aplicação de teste era executada em quatro máquinas: corvette, maverick, buick e bentley. Era oferecido ao usuário uma área de desenho; o que era desenhado em uma máquina era transmitido a todos os membros do grupo, que exibiam a imagem. No teste realizado, 50 segundos após o início do experimento, uma falha de particionamento de rede era ativada. Nesta falha, as máquinas eram isoladas em duas metades: corvette e maverick não podiam se comunicar com buick e bentley. Para o usuário da aplicação, o que antes era um grupo de quatro máquinas acabava transformado em dois grupos de duas máquinas. A Figura 5.6, retirada do artigo em que esse teste foi apresentado, exemplifica a injeção desta falha. Uma letra era desenhada em cada máquina do grupo. Nos 50 segundos iniciais, como as quatro máquinas participavam do mesmo grupo, todas receberam as imagens das letras. Após a ativação da falha de particionamento, formaram-se dois grupos. Novamente, uma letra foi desenhada em cada máquina. No entanto, como estavam em grupos diferentes, as letras *E* e *F* foram transmitidas para os dois primeiros membros, e as letras *G* e *H*, para os outros dois.

Figura 5.6: Aplicação de teste de JGroups com falha particionamento de rede. Fonte: (JACQUES-SILVA et al., 2006)

O framework JGroups oferece outra estratégia de detecção de falhas, conhecido como *detector FDSOCK*, que usa comunicação via protocolo TCP. A transmissão de dados continua com multicast UDP; apenas o mecanismo de detecção de falhas é alterado. Usando FDSOCK, sockets TCP são abertos entre os membros do grupo. Se houver uma desconexão, os sockets para aquele membro são fechados pelo sistema operacional. Esse fechamento é interpretado como uma falha.

Como FIONA apenas injeta falhas em comunicações usando o protocolo UDP, não pode ser aplicado para testar FDSOCK. Uma estratégia é usar em conjunto o injetor FIERCE, que atua sobre aplicações que usam o protocolo TCP (GERCHMAN; WEBER, 2006). FIONA atuaria sobre a transmissão dos dados, que ocorre usando multicast UDP, enquanto que FIERCE atuaria sobre o detector de falhas de JGroups, que usa TCP.

A seguir são detalhadas as etapas seguidas no desenvolvimento deste exemplo, conforme a metodologia sugerida na Seção 5.2.

Criação do modelo de falhas

Foi definido um modelo de falhas que abrangesse as capacidades de ambas as ferramentas de injeção. O modelo final pode ser visto na Figura 5.7. Das falhas oferecidas por

ambos os injetores, apenas a de particionamento de rede foi modelada. Ela é representada pela classe NetworkPartitionFault, que tem um atributo, nodeList. Para simplificar a implementação, esse atributo é uma *string* que lista as máquinas em cada uma das partições. Dentro de cada partição, as máquinas são separadas por vírgulas; as partições, pelo sinal de dois pontos. Para ativação, foram modeladas as classes TimeActivation, similar à encontrada na seção anterior, e BytesTransferredActivation, que sinaliza o início de injeção de uma falha a partir de um certo número de bytes transmitidos.

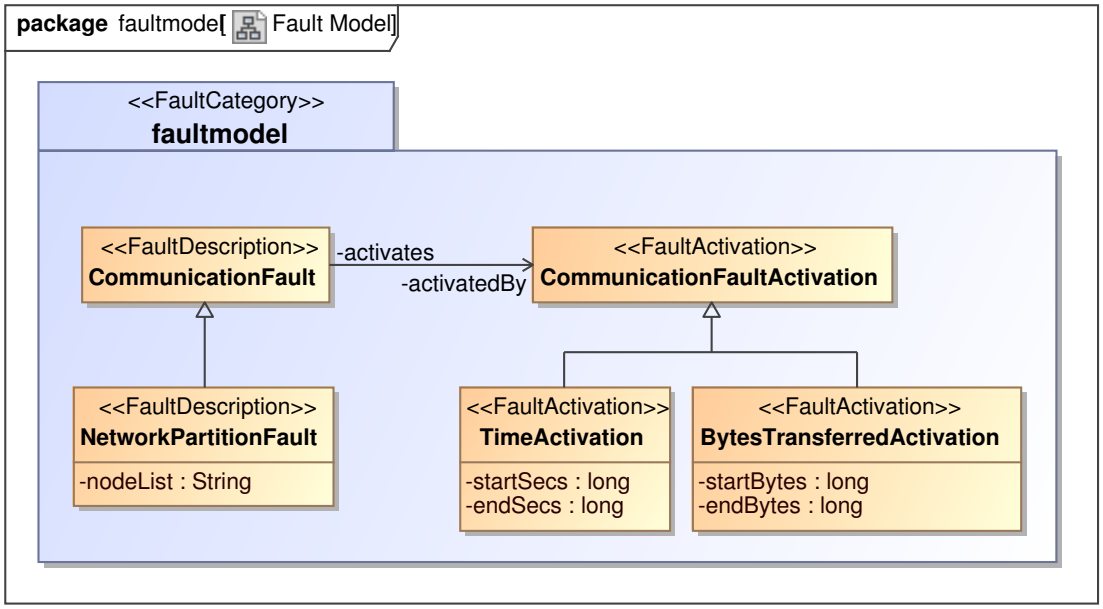

Figura 5.7: Modelo de falhas para a aplicação de teste JGroups

Ao contrário de FIONA, a ferramenta FIERCE não suporta de maneira direta uma falha de particionamento de rede. No entanto, ela pode ser facilmente emulada através da criação de falhas de colapso individuais entre os nodos. Isto é, para cada máquina de uma das partições são definidas falhas de colapso tendo como destino as máquinas que são membros das outras partições.

Modelagem do caso de teste

Como parte da modelagem do caso de teste, foi criada a carga de falhas na forma de instâncias de falhas. Essas instâncias apresentam os parâmetros que definem o seu comportamento durante a atividade de validação.

A carga de falhas foi definida no diagrama de classes que pode ser visto na Figura 5.8. A instância networkPartition modela a falha a ser injetada, especificando uma partição em dois grupos de máquinas. O primeiro grupo contém as máquinas corvette e maverick; o segundo, buick e bentley. Essa instância de falha é associada a afterFiftySeconds, uma instância que indica o início da injeção 50 segundos a partir do início do experimento.

Criação dos cartuchos AndroMDA

Foram desenvolvidos dois cartuchos para AndroMDA, fiona e fierce. Os dois cartuchos selecionam no modelo as instâncias do tipo NetworkPartitionFault e as passam para os *templates* de geração de arquivos de configuração. No caso de FIONA, o template

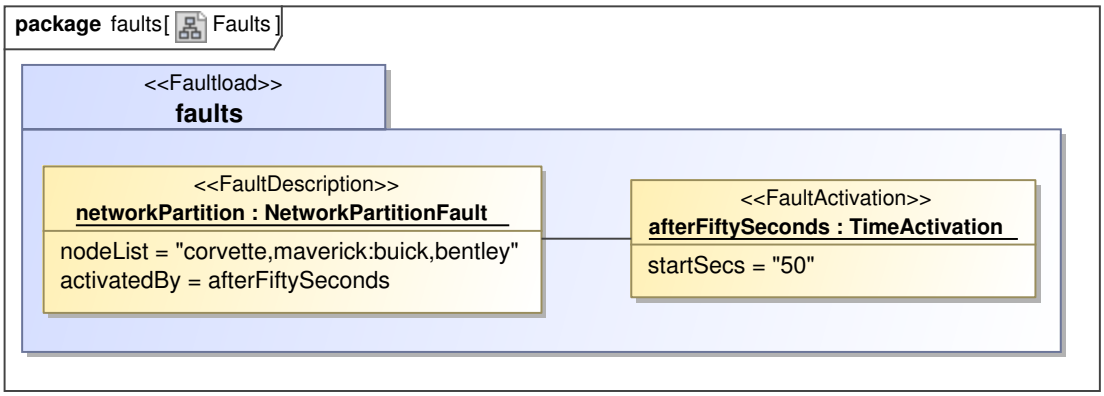

Figura 5.8: Carga de falhas para a aplicação de teste JGroups

é simples, diretamente gerando falhas do tipo UdpNetworkPartitioningFault. Para a geração dos arquivos para FIERCE, foi usada uma pequena classe auxiliar programada em Java. Essa classe apenas faz a quebra da *string* original, que lista os nodos das partições. O *template*, usando essa classe auxiliar, gera as múltiplas falhas do tipo TcpCrashFault necessárias para emular a falha de particionamento.

Geração dos artefatos de teste

A ferramenta AndroMDA foi invocada nesta etapa para gerar os arquivos necessários a partir dos modelos de teste. No arquivo de configuração de AndroMDA foram incluídas referências para os dois cartuchos criados, fiona e fierce. Desta maneira, em apenas uma execução da ferramenta, os dois cartuchos são invocados. Os arquivos de configuração para FIONA e FIERCE podem ser vistos na Figura 5.9.

Com os arquivos de configuração em mãos, a campanha de teste usando FIONA e FIERCE pode ser iniciada.

```
# FIONA: Fault: networkPartition, Activated by: afterFiftySeconds
UdpNetworkPartitioningFault:50:0:corvette,maverick:buick,bentley
```

```
# FIERCE: Fault: networkPartition, Activated by: afterFiftySeconds
TcpCrashFault:50:corvette:*:buick:*
TcpCrashFault:50:corvette:*:bentley:*
TcpCrashFault:50:maverick:*:buick:*
TcpCrashFault:50:maverick:*:bentley:*
TcpCrashFault:50:buick:*:corvette:*
TcpCrashFault:50:buick:*:maverick:*
TcpCrashFault:50:bentley:*:corvette:*
TcpCrashFault:50:bentley:*:maverick:*
```

Figura 5.9: Carga de falhas para FIONA e para FIERCE

5.5 Exemplo 3: geração de relatórios

Os exemplos anteriores mostraram a possibilidade de geração de arquivos de configuração para injetores de falhas a partir de modelos U2TP-FI. Este exemplo explora a geração automatizada de outros tipos de artefatos que podem ser usados por uma equipe de testes durante o processo de validação de sistemas.

Um diagrama UML é um formato usado também como documentação de um sistema. No entanto, em casos específicos, uma descrição textual pode ser considerada mais útil ou adequada. Um caso desse seria a redação de um documento de especificação onde seja necessária uma explicação em linguagem natural dos diagramas apresentados.

O objetivo do terceiro exemplo é demonstrar a possibilidade de geração de um relatório detalhando em linguagem natural das características de uma campanha de testes. Esse relatório conteria os dados presentes no modelo U2TP-FI, acompanhado de uma descrição textual das falhas e de suas condições de ativação. Esse relatório pode ser usado para auxiliar na documentação do processo usado ou como apoio para uma equipe de testes.

O terceiro exemplo é uma continuação do primeiro apresentado neste capítulo, a injeção de falhas de comunicação para a validação do *middleware* de computação em grade OurGrid. O objetivo é a criação de um novo cartucho AndroMDA que, ao ser executado, gera arquivos HTML com um relatório das falhas a serem injetadas. Os cartuchos novo, de geração de relatório, e os originais, de geração de carga de falhas para o injetor FIRMI, podem ser executados em conjunto como parte do processo de preparação para o teste.

A seguir são detalhadas as etapas seguidas no desenvolvimento deste exemplo, conforme a metodologia sugerida na Seção 5.2.

Criação do modelo de falhas e modelagem do caso de teste

Este exemplo foi construído em cima do modelo de falhas e do caso de teste do primeiro exemplo deste capítulo, apresentado na Seção 5.3. Eles não foram alterados.

Criação do cartucho AndroMDA

Para gerar o relatório, um novo cartucho AndroMDA, firmi-fault-report, foi desenvolvido. O cartucho seleciona no modelo todas as instâncias das classes `CrashFault` e `DelayFault`, que representam falhas de colapso e de temporização. Delas são extraídos os atributos que definem suas características, como *hostname* das máquinas envolvidas. Os atributos das condições de ativação, representadas por instâncias das classes `InvocationActivation` e `TimeActivation`, também são extraídos.

Esses atributos são passados para um esqueleto (*template*) de arquivo HTML, preenchendo lacunas com as informações presentes no modelo de teste U2TP-FI.

Geração dos artefatos de teste

Esse cartucho de geração de relatórios pode ser executado em conjunto com os cartuchos originais. Assim, os arquivos de configuração do injetor FIRMI são gerados ao mesmo tempo que os relatórios, garantindo a consistência das informações.

O relatório pode ser visto na Figura 5.10. Nele, está presente uma descrição da falha modelada na Seção 5.3 em conjunto com uma explicação de seus efeitos e suas condições de ativação.

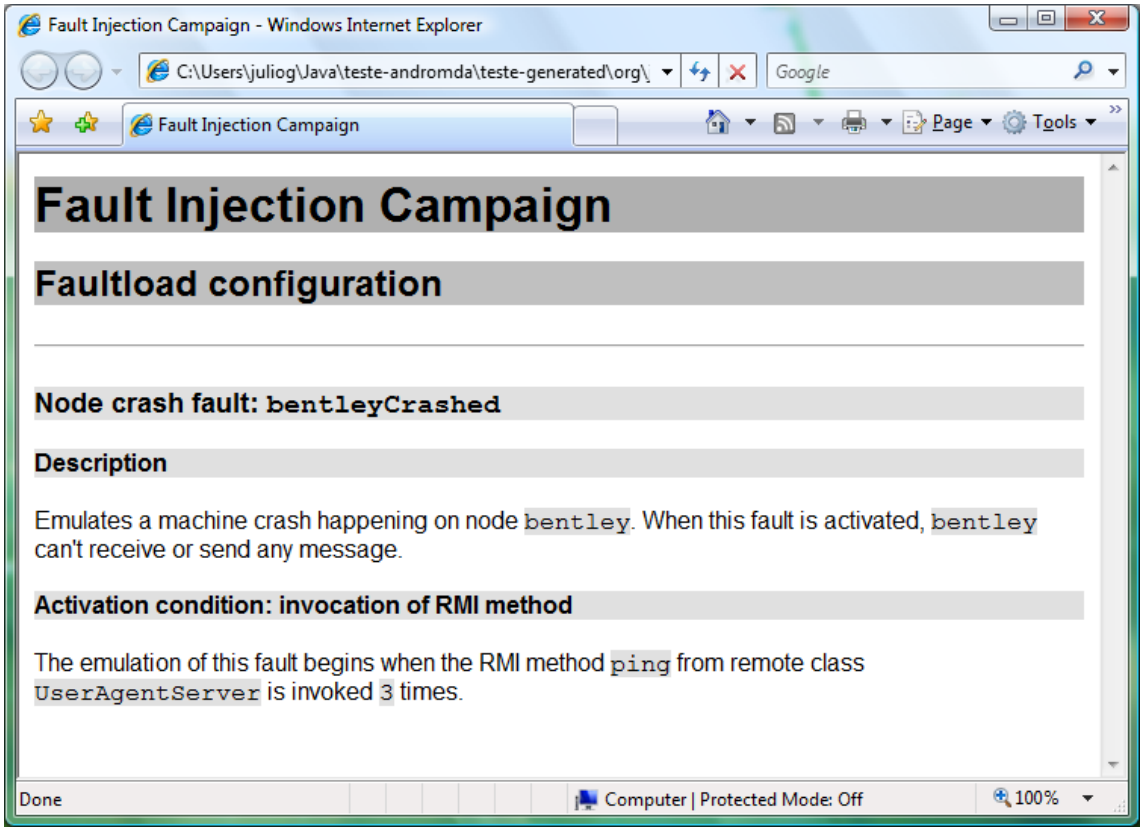

Figura 5.10: Relatório descrevendo a carga de falhas de um experimento

5.6 Conclusões

Observando os modelos de teste criados, as ferramentas usadas e os artefatos gerados, é possível dizer que os objetivos de U2TP-FI, listados na Seção 4.2, foram alcançados:

- *Linguagem de descrição de injeção de falhas.* Os estereótipos disponibilizados por U2TP-FI identificam os elementos de injeção de falhas de forma clara em um diagrama U2TP/UML. Os estereótipos «FaultDescription», «FaultActivation» e «FaultParameter» indicam classes que representam falhas e as propriedades que regem seu comportamento. Pacotes decorados com «FaultCategory» e «Faultload» identificam o modelo de falhas adotado e a carga de falhas a ser aplicada em uma atividade de teste. A relação de dependência «InjectFault» mostra claramente o local onde uma falha será injetada.

- *Independência de plataforma.* É possível modelar conceitos de injeção de falhas sem associar o modelo de testes a um injetor ou artefato específico, como no caso do segundo exemplo, em que o mesmo modelo de falhas foi usado para a aplicação de duas ferramentas diferentes. O primeiro exemplo também demonstrou que diferentes arquivos de configuração podem ser gerados a partir dos mesmos elementos.

- *Documentação de atividades de teste.* Descrever o *faultload* usando a linguagem UML facilita a transmissão de conhecimento na equipe ao impedir que as características do teste sejam ocultadas em arquivos de difícil compreensão. Um diagrama de classe é de mais fácil entendimento que arquivos de configuração, carregando mais informações de uma maneira visual.

- *Consistência entre diferentes artefatos de teste.* Os parâmetros de teste ficam centralizados no modelo: não há redundância de informações, o que impede que arquivos de configuração ou outros artefatos de interesse fiquem fora de sincronia. Se for necessária a alteração de algum parâmetro de injeção de falhas, esse parâmetro é alterado diretamente e unicamente no modelo. Os artefatos usados são gerados automaticamente a partir dessa nova versão.

- *Automação de tarefas repetitivas.* Ao centralizar os parâmetros de teste em um modelo UML e delegar a geração dos artefatos necessários à ferramenta MDA usada, a intervenção humana é diminuída, acelerando as tarefas e possibilitando uma maior padronização destas.

6 CONSIDERAÇÕES FINAIS

Como um número sempre crescente de sistemas computacionais são considerados de importância crítica por seus usuários, níveis de dependabilidade cada vez maiores são exigidos destes. No entanto, justificar e garantir esses níveis em ambientes complexos e sujeitos a problemas de operação e a ocorrência de falhas é um desafio cada vez maior.

O teste é uma atividade essencial para capturar erros existentes na construção de um sistema computacional e garantir sua qualidade. No entanto, a complexidade desses testes aumenta caso níveis de dependabilidade mais elevados forem exigidos. Essa exigência torna ainda mais necessária a realização de atividades de verificação e validação.

Injeção de falhas é uma técnica que pode ser usada para a validação experimental de mecanismos de tolerância a falhas destinados a oferecer níveis de dependabilidade mais elevados. Uma simulação ou um protótipo funcional do sistema é posto em execução em um ambiente em que falhas são artificialmente emuladas. O sistema é monitorado, verificando se os mecanismos implementados são efetivos para os cenários de teste montados.

Injeção de falhas é uma técnica de teste, e toda atividade de teste depende de um modelo que a descreva. Esse modelo, mesmo que não seja formalizado, contém representações do sistema sob testes, da carga de trabalho aplicada, das verificações necessárias e de outros elementos que fazem parte da atividade. No caso de uso de injeção de falhas, elementos pertinentes, como a carga de falhas, também devem ser descritos. A documentação desses modelos de teste usando uma linguagem padrão e visual, como diagramas UML, facilita a comunicação entre as equipes de teste e de desenvolvimento de um sistema, bem como a visualização e análise dele.

Para a criação de modelos de teste, o Object Management Group oferece o Perfil UML 2.0 de Testes (U2TP), linguagem padronizada que permite o projeto, visualização e construção de artefatos aplicados em verificação e validação de sistemas. Como o perfil é uma extensão de UML, os modelos usados para o desenvolvimento do restante do sistema podem ser reusados e integrados aos modelos de teste. U2TP, no entanto, não oferece por padrão elementos para a descrição de atividades de validação que usem técnicas de injeção de falhas.

Este trabalho propõe U2TP-FI (*UML 2.0 Testing Profile with Fault Injection*), uma extensão para U2TP que permite a descrição de aspectos estáticos e dinâmicos de injeção de falhas. U2TP-FI adiciona ao perfil original elementos pertinentes a atividades de teste que usem essa técnica. Diagramas U2TP-FI permitem a uma equipe de teste descrever cargas de falhas a serem injetadas, bem como indicar o local em que elas atuam e as condições que devem ser cumpridas para sua ativação. São oferecidos também elementos para agrupar e organizar os elementos criados.

U2TP-FI é organizado em três grupos de conceitos: descrição, ativação e organização de falhas. O grupo de descrição de falhas oferece elementos para especificar os parâmetros

que regem seu comportamento em um ambiente de execução. Os elementos do grupo de ativação de falhas são usados para descrever as condições a serem cumpridas para o início de sua injeção e para indicar o componente do sistema onde atuarão. Agrupamento e classificação de falhas são manipuladas pelos conceitos do terceiro grupo, organização.

O objetivo de U2TP-FI é o estabelecimento de uma linguagem de descrição de modelos para atividades de injeção de falhas que permita independência de plataforma, documentação de atividades de teste, consistência entre diferentes artefatos de teste e automação de tarefas repetitivas. Para provar esses pontos, foram organizadas provas de conceito, exemplos usando ferramentas, injetores de falhas e aplicações existentes. Diagramas de teste foram criados em sistemas CASE e manipulados usando ferramentas de transformação de modelos, gerando artefatos que poderiam ser usados por uma equipe de garantia de qualidade. A geração desses artefatos, como arquivos de configuração e relatórios, é automatizada e pode ser integrada a metodologias e processos já adotados no desenvolvimento de um sistema existente.

Por se tratar de uma linguagem de descrição de modelos, seguindo os princípios de UML, U2TP-FI não é atrelada a uma metodologia ou política de uso específica. Elementos são oferecidos para o usuário, que pode integrá-los a metodologias pré-existentes como bem entender. No entanto, a metodologia seguida durante o desenvolvimento das provas de conceito foi documentada como parte desse trabalho. Ela pode ser usada como sugestão ou mesmo base para o estabelecimento de processos próprios de usuários de U2TP-FI.

O desenvolvimento de U2TP-FI aproveitou-se da experiência adquirida pelo Grupo de Tolerância a Falhas/UFRGS durante a criação de diversas ferramentas de injeção de falhas e a aplicação destas na validação de sistemas. A extensão para injeção de falhas do Perfil UML 2.0 de Testes é flexível, podendo ser aplicada em diferentes ambientes e aplicações, bem como suportar diferentes injetores. Oferecendo uma linguagem padronizada, permite a documentação das atividades que usem essa técnica de teste, bem como a automação de tarefas antes realizadas manualmente.

6.1 Trabalhos futuros

São observados dois principais focos de desenvolvimento de trabalhos futuros envolvendo U2TP-FI: a criação de um *catálogo de falhas* e a *integração com metodologias existentes*.

Um catálogo de falhas seria uma biblioteca de classes e de relações que descrevem modelos de falhas comuns, amplamente empregados em validação de sistemas. Essa idéia é inspirada no catálogo de QoS, parte do Perfil QoS, que oferece um conjunto de categorias e características de qualidade de serviço que não são específicas a projetos ou domínios (OBJECT MANAGEMENT GROUP, 2006). No entanto, percebe-se na literatura a adoção de modelos de falhas comuns a um domínio específico, como falhas de memória ou de comunicação. O estabelecimento de um catálogo poderia facilitar o emprego de U2TP-FI no teste de sistemas ao permitir o reuso de modelos de falhas já existentes, sem que seja necessário a criação desses modelos pelo usuário.

U2TP-FI não é atrelado a uma metodologia específica de testes. No entanto, existem esforços para a definição de metodologias de teste que integrem técnicas de injeção de falhas, como a desenvolvida por Menegotto e Weber; esta adapta um processo usado para a análise de desempenho de sistemas, incluindo etapas para a descrição de modelos e cargas de falha (MENEGOTTO; VACARO; WEBER, 2007). A integração de U2TP-FI

com esses esforços para a definição de metodologias de testes abre novos rumos para o estudo e desenvolvimento destas.

REFERÊNCIAS

ANDRADE, N. et al. OurGrid: an approach to easily assemble grids with equitable resource sharing. In: INTERNATIONAL WORKSHOP ON JOB SCHEDULING STRATEGIES FOR PARALLEL PROCESSING, JSSPP, 9., 2003. **Revised Paper...** Berlin: Springer, 2003. p.61–86. (Lecture Notes in Computer Science, v.2862).

ANDROMDA.ORG. **What is AndroMDA?** Disponível em: <http://www.andromda.org/>. Acesso em: ago. 2007.

APFELBAUM, L.; DOYLE, J. Model Based Testing. In: SOFTWARE QUALITY WEEK CONFERENCE, 1997. **Proceedings...** [S.l.: s.n.], 1997. Disponível em: <http://www.geocities.com/model_based_testing/sqw97.pdf>. Acesso em: ago. 2007.

AVIZIENIS, A. et al. Basic Concepts and Taxonomy of Dependable and Secure Computing. **IEEE Transactions on Dependable and Secure Computing**, Los Alamitos, CA, USA, v.1, n.1, p.11–33, Jan. 2004.

BARCELLOS, M. P.; WOSZEZENKI, C. R.; MUNARETTI, R. S. Framework de Injeção de Falhas Simulada para Avaliação de Sistemas Distribuídos. In: SIMPÓSIO BRASILEIRO DE REDES DE COMPUTADORES, 23., 2005, Fortaleza. **Anais...** Porto Alegre: Sociedade Brasileira de Computação, 2005.

BARCELOS, P. P. A.; WEBER, T. S. INFIMO: um injetor de falhas de comunicação para aplicações RT-Linux. In: WORKSHOP SOBRE SOFTWARE LIVRE, 1., 2000, Porto Alegre. **Anais...** Porto Alegre: Sociedade Brasileira de Computação, 2000. p.41–44.

CARREIRA, J.; SILVA, J. G. Why do Some (weird) People Inject Faults? **SIGSOFT Software Engineering Notes**, New York, NY, USA, v.23, n.1, p.42–43, Jan. 1998.

DAWSON, S.; JAHANIAN, F.; MITTON, T. Experiments on Six Commercial TCP Implementations Using a Software Fault Injection Tool. **Software Practice and Experience**, [S.l.], v.27, n.12, p.1385–1410, 1997.

DELAMARO, M. E.; MALDONADO, J. C.; MATHUR, A. P. Integration testing using interface mutation. In: INTERNATIONAL SYMPOSIUM ON SOFTWARE RELIABILITY ENGINEERING, 7., 1996, Los Alamitos, CA, USA. **Proceedings...** [S.l.]: IEEE Computer Society, 1996. p.112–121.

EL-FAR, I. K. Enjoying the Perks of Model-Based Testing. In: SOFTWARE TESTING, ANALYSIS, AND REVIEW CONFERENCE,

STARWEST, 2001. **Proceedings...** [S.l.: s.n.], 2001. Disponível em: <http://www.geocities.com/model_based_testing/perks_paper.pdf>. Acesso em: ago. 2007.

FELDMAN, S. Quality Assurance: much more than testing. **Queue**, New York, NY, USA, v.3, n.1, p.26–29, Feb. 2005.

GERCHMAN, J.; JACQUES-SILVA, G.; DREBES, R. J.; WEBER, T. S. Ambiente Distribuído de Injeção de Falhas de Comunicação para Teste de Aplicações Java de Rede. In: SIMPÓSIO BRASILEIRO DE ENGENHARIA DE SOFTWARE, 19., 2005, Uberlândia. **Anais...** Rio de Janeiro: PUCRJ, 2005. p.232–246.

GERCHMAN, J.; JACQUES-SILVA, G.; JANSCH-PÔRTO, I. E. S.; WEBER, T. S. Implementação de um Injetor de Falhas de Comunicação para Aplicações Java Baseado em JVMTI. In: SALÃO DE INICIAÇÃO CIENTÍFICA, 16., 2004, Porto Alegre. **Anais...** Porto Alegre: UFRGS, 2004. p.46.

GERCHMAN, J.; JACQUES-SILVA, G.; WEBER, T. S. Implementação de um Injetor de Falhas de Comunicação para Aplicações Java Baseado em JVMTI. In: ESCOLA REGIONAL DE REDES DE COMPUTADORES, 2., 2004, Canoas. **Anais...** Porto Alegre: Sociedade Brasileira de Computação, 2004. v.1, p.23–28.

GERCHMAN, J.; WEBER, T. S. Emulando o Comportamento de TCP/IP em um Ambiente com Falhas para Teste de Aplicações de Rede. In: WORKSHOP DE TESTES E TOLERÂNCIA A FALHAS, 7., 2006, Curitiba. **Anais...** Porto Alegre: Sociedade Brasileira de Computação, 2006. v.1, p.41–54.

GERCHMAN, J.; WEBER, T. S. Integrando Injeção de Falhas ao Perfil UML 2.0 de Testes. In: WORKSHOP DE TESTES E TOLERÂNCIA A FALHAS, 8., 2007, Belém. **Anais...** [S.l.]: Sociedade Brasileira de Computação, 2007. p.87–98.

GONÇALVES, L. C. R.; WEBER, T. S. Injeção de Falhas via Depuradores. In: WORKSHOP SOBRE SOFTWARE LIVRE, 1., 2000, Porto Alegre. **Anais...** Porto Alegre: Sociedade Brasileira de Computação, 2000. p.45–48.

HSUEH, M.-C.; TSAI, T. K.; IYER, R. K. Fault Injection Techniques and Tools. **Computer**, Los Alamitos, CA, USA, v.30, n.4, p.75–82, Apr. 1997.

JACQUES-SILVA, G.; DREBES, R. J.; GERCHMAN, J.; TRINDADE, J. M. F. da; WEBER, T. S.; JANSCH-PÔRTO, I. A Network-Level Distributed Fault Injector for Experimental Validation of Dependable Distributed Systems. In: INTERNATIONAL COMPUTER SOFTWARE AND APPLICATIONS CONFERENCE, COMPSAC, 30., 2006, Chicago, Illinois. **Proceedings...** Los Alamitos: IEEE Computer Society, 2006. p.421–428.

JACQUES-SILVA, G.; DREBES, R. J.; GERCHMAN, J.; WEBER, T. S. FIONA: a fault injector for dependability evaluation of Java-based network applications. In: IEEE INTERNATIONAL SYMPOSIUM ON NETWORK COMPUTING AND APPLICATIONS, NCA, 3., 2004, Boston, Massachusetts. **Proceedings...** Los Alamitos: IEEE Computer Society, 2004. p.303–308.

JACQUES-SILVA, G.; DREBES, R. J.; WEBER, T. S.; MARTINS, E. Injecting Communication Faults to Experimentally Validate Java Distributed Applications. In: INTERNATIONAL SCHOOL AND SYMPOSIUM, ISSADS, 5., 2005, Guadalajara, Mexico. **Proceedings...** Berlin: Springer, 2005. p.235–245. (Lecture Notes in Computer Science, v.3563).

LEME, N. G. M.; MARTINS, E.; RUBIRA, C. M. F. A software fault injection pattern system. In: BRAZILIAN SYMPOSIUM ON FAULT-TOLERANT COMPUTING, 9., 2001, Florianópolis. **Proceedings...** Florianópolis: UFSC, 2001. p.99–113.

MARTINS, E.; RUBIRA, C. M. F.; LEME, N. G. M. Jaca: a reflective fault injection tool based on patterns. In: INTERNATIONAL CONFERENCE ON DEPENDABLE SYSTEMS AND NETWORKS, DSN, 2002, Bethesda, Maryland. **Proceedings...** Los Alamitos: IEEE Computer Society, 2002. p.483–482.

MENEGOTTO, C. C.; VACARO, J. C.; WEBER, T. S. Injeção de Falhas de Comunicação em Grid com Características de Tolerância a Falhas. In: WORKSHOP DE TESTES E TOLERÂNCIA A FALHAS, 8., 2007, Belém. **Anais...** [S.l.]: Sociedade Brasileira de Computação, 2007. p.71–84.

NO MAGIC INC. **MagicDraw UML**. Disponível em: <http://www.magicdraw.com/>. Acesso em: ago. 2007.

OBJECT MANAGEMENT GROUP. **Meta Object Facility Specification, joint revised submission**. [S.l.], 1997. (Document ad/97-08-14).

OBJECT MANAGEMENT GROUP. **White Paper on the Profile Mechanism, version 1.0**. [S.l.], 1999. (Document ad/99-04-07).

OBJECT MANAGEMENT GROUP. **UML 2.0 Testing Profile**. [S.l.], 2005. (Document formal/05-07-07).

OBJECT MANAGEMENT GROUP. **UML Profile for Modeling Quality of Service and Fault Tolerance Characteristics and Mechanisms**. [S.l.], 2006. (Document formal/06-05-02).

SCHNEIDER, F. B. What Good are Models and What Models are Good? In: MULLENDER, S. (Ed.). **Distributed Systems**. 2nd ed. Workingham: Addison-Wesley, 1993. p.17–26.

SUGETA, T.; MALDONADO, J. C.; WONG, W. E. Mutation Testing Applied to Validate SDL Specifications. In: TESTCOM, 16., 2004. **Proceedings...** Berlin: Springer, 2004. p.193–208. (Lecture Notes in Computer Science, v.2978).

VACARO, J. C.; WEBER, T. S. Injeção de Falhas na Fase de Teste de Aplicações Distribuídas. In: SIMPÓSIO BRASILEIRO DE ENGENHARIA DE SOFTWARE, 20., 2006, Florianópolis. **Anais...** Florianópolis: Sociedade Brasileira de Computação, 2006. v.1, p.161–176.

VOAS, J. Fault injection for the masses. **Computer**, Los Alamitos, CA, USA, v.30, n.12, p.129–130, Dec. 1997.

ZANDER, J.; DAI, Z. R.; SCHIEFERDECKER, I.; DIN, G. From U2TP Models to Executable Tests with TTCN-3 - An Approach to Model Driven Testing. In: TESTCOM, 17., 2005. **Proceedings...** Berlin: Springer, 2005. p.289–303. (Lecture Notes in Computer Science, v.3502).

APÊNDICE A HISTÓRICO DE DESENVOLVIMENTO

O Grupo de Tolerância a Falhas da UFRGS vem há anos desenvolvendo trabalhos na área de injeção de falhas por software, especialmente voltados ao desenvolvimento de ferramentas para o teste de aplicações de rede. Entre os primeiros trabalhos estão FIDe e INFIMO. FIDe realizava a interceptação de chamadas de sistema usando primitivas para depuração oferecidas pelo sistema operacional Linux; após a interceptação, o estado da aplicação era alterado de acordo com as necessidades da injeção de falhas (GONÇALVES; WEBER, 2000). No entanto, essa estratégia apresenta uma alta intrusão temporal, dificultando seu uso. INFIMO, além de aprofundar o modelo de falhas, dimunuía o problema da intrusividade temporal ao forçar o uso pela aplicação de bibliotecas de comunicação previamente instrumentadas (BARCELOS; WEBER, 2000).

O trabalho do autor junto ao Grupo de Tolerância a Falhas iniciou em 2004, auxiliando no desenvolvimento do injetor Jaca.Net. A ferramenta Jaca.Net é uma extensão da ferramenta de injeção de falhas Jaca, oferecendo ao usuário falhas de comunicação em aplicações Java que usam o protocolo UDP (JACQUES-SILVA et al., 2005). O autor auxiliou em experiências de instrumentação de *bytecode* Java, alterando arquivos compilados para referenciar classes de comunicação instrumentadas ao invés das originais.

Como Jaca.Net necessitava um pré-processamento das classes da aplicação antes de sua execução, foi explorada a possibilidade de instrumentação de código através da biblioteca de depuração e monitoramento JVMTI, oferecida pela plataforma Java a partir da versão 1.5. A partir dessa exploração, foi desenvolvido o injetor de falhas de comunicação FIONA, parte da dissertação de Mestrado de Gabriela Jacques-Silva. FIONA também é voltado ao teste de aplicações Java que usam o protocolo UDP. A ferramenta oferece também a funcionalidade de controle distribuído, onde múltiplos injetores em múltiplas máquinas interligadas têm seu funcionamento gerenciado através da rede. Este trabalho gerou artigos publicados em eventos regionais, nacionais e internacionais (GERCHMAN et al., 2004; GERCHMAN; JACQUES-SILVA; WEBER, 2004; JACQUES-SILVA et al., 2004; GERCHMAN et al., 2005; JACQUES-SILVA et al., 2006).

O modelo de falhas de FIONA foi estendido para que a ferramenta comportasse também o teste de aplicações que usassem o protocolo de comunicação TCP. O resultado desse trabalho foi o injetor de falhas FIERCE, desenvolvido como parte do trabalho de conclusão do autor (GERCHMAN; WEBER, 2006). FIERCE é uma extensão do injetor anterior, adotando a mesma estratégia de instrumentação de classes de sistema. O uso da ferramenta permite a emulação de condições de erro que ocorrem quando há problemas em uma comunicação usando *sockets* TCP em uma aplicação Java, como colapso de máquina ou encerramento de processo.

Durante o desenvolvimento dessas ferramentas, notou-se uma necessidade de uma maior integração das técnicas de injeção de falhas aos processos e metodologias usados

para análise, projeto e condução de testes. Dois focos diferentes foram tomados no Grupo de Tolerância a Falhas. Por um lado, conceitos de injeção de falhas foram integrados a processos de teste já existentes. Isto resultou em um trabalho onde etapas de planejamento e execução de injeção de falhas foram incorporados em uma metodologia de avaliação de desempenho, permitindo a validação de mecanismos de tolerância e a medição de parâmetros como eficiência e cobertura (MENEGOTTO; VACARO; WEBER, 2007).

Observando o aumento do interesse em metodologias de desenvolvimento dirigido a modelos, como MDA, o outro foco tomado foi a integração de injeção de falhas em atividades de modelagem de testes. Dessa iniciativa resultou U2TP-FI, a extensão para o Perfil UML 2.0 de Testes, descrita no presente trabalho e em artigo publicado em evento nacional (GERCHMAN; WEBER, 2007).

Artigos publicados

GERCHMAN, J.; JACQUES-SILVA, G.; JANSCH-PÔRTO, I. E. S.; WEBER, T. S. Implementação de um Injetor de Falhas de Comunicação para Aplicações Java Baseado em JVMTI. In: SALÃO DE INICIAÇÃO CIENTÍFICA, 16., 2004, Porto Alegre. **Anais...** UFRGS, 2004. p.46.

GERCHMAN, J.; JACQUES-SILVA, G.; WEBER, T. S. Implementação de um Injetor de Falhas de Comunicação para Aplicações Java Baseado em JVMTI. In: ESCOLA REGIONAL DE REDES DE COMPUTADORES, 2., 2004, Canoas. **Anais...** Sociedade Brasileira de Computação, 2004. v.1, p.23-28.

JACQUES-SILVA, G.; DREBES, R. J.; GERCHMAN, J.; WEBER, T. S. FIONA: a fault injector for dependability evaluation of Java-based network applications. In: IEEE INTERNATIONAL SYMPOSIUM ON NETWORK COMPUTING AND APPLICATIONS (NCA 2004), 3., 2004, Cambridge, Massachusetts. **Proceedings...** IEEE Computer Society, 2004. p.303-308.

GERCHMAN, J.; JACQUES-SILVA, G.; DREBES, R. J.; WEBER, T. S. Ambiente Distribuído de Injeção de Falhas de Comunicação para Teste de Aplicações Java de Rede. In: SIMPÓSIO BRASILEIRO DE ENGENHARIA DE SOFTWARE, 19., 2005, Uberlândia. **Anais...** Sociedade Brasileira de Computação, 2005. p.232-246.

JACQUES-SILVA, G.; DREBES, R. J.; GERCHMAN, J.; TRINDADE, J. M. F. da; WEBER, T. S.; JANSCH-PÔRTO, I. A Network-Level Distributed Fault Injector for Experimental Validation of Dependable Distributed Systems. In: ANNUAL INTERNATIONAL COMPUTER SOFTWARE AND APPLICATIONS CONFERENCE (COMPSAC 2006), 30., 2006, Chicago, Illinois. **Proceedings...** IEEE Computer Society, 2006. p.421-428.

GERCHMAN, J.; WEBER, T. S. Emulando o Comportamento de TCP/IP em um Ambiente com Falhas para Teste de Aplicações de Rede. In: WORKSHOP DE TESTES E TOLERÂNCIA A FALHAS, 7., 2006, Curitiba. **Anais...** Sociedade Brasileira de Computação, 2006. v.1, p.41-54.

GERCHMAN, J.; WEBER, T. S. Integrando Injeção de Falhas ao Perfil UML 2.0 de Testes. In: WORKSHOP DE TESTES E TOLERÂNCIA A FALHAS, 8., 2007, Belém. **Anais...** Sociedade Brasileira de Computação, 2007. p.87-98.

Artigos submetidos para avaliação

GERCHMAN, J.; WEBER, T. S. U2TP-FI: Extensão do Perfil UML de Testes para Uso de Injeção de Falhas. Submetido ao 21º. Simpósio Brasileiro de Engenharia de Software (SBES 2007).

GERCHMAN, J.; WEBER, T. S. Gerando cargas de falha a partir de modelos UML de teste. Submetido ao 1st Brazilian Workshop on Systematic and Automated Software Testing (SAST 2007).

Livros Grátis

(http://www.livrosgratis.com.br)

Milhares de Livros para Download:

Baixar livros de Administração
Baixar livros de Agronomia
Baixar livros de Arquitetura
Baixar livros de Artes
Baixar livros de Astronomia
Baixar livros de Biologia Geral
Baixar livros de Ciência da Computação
Baixar livros de Ciência da Informação
Baixar livros de Ciência Política
Baixar livros de Ciências da Saúde
Baixar livros de Comunicação
Baixar livros do Conselho Nacional de Educação - CNE
Baixar livros de Defesa civil
Baixar livros de Direito
Baixar livros de Direitos humanos
Baixar livros de Economia
Baixar livros de Economia Doméstica
Baixar livros de Educação
Baixar livros de Educação - Trânsito
Baixar livros de Educação Física
Baixar livros de Engenharia Aeroespacial
Baixar livros de Farmácia
Baixar livros de Filosofia
Baixar livros de Física
Baixar livros de Geociências
Baixar livros de Geografia
Baixar livros de História
Baixar livros de Línguas

Baixar livros de Literatura
Baixar livros de Literatura de Cordel
Baixar livros de Literatura Infantil
Baixar livros de Matemática
Baixar livros de Medicina
Baixar livros de Medicina Veterinária
Baixar livros de Meio Ambiente
Baixar livros de Meteorologia
Baixar Monografias e TCC
Baixar livros Multidisciplinar
Baixar livros de Música
Baixar livros de Psicologia
Baixar livros de Química
Baixar livros de Saúde Coletiva
Baixar livros de Serviço Social
Baixar livros de Sociologia
Baixar livros de Teologia
Baixar livros de Trabalho
Baixar livros de Turismo