Uma Arquitetura Reflexiva para Injetar Falhas em Aplicações Orientadas a Objetos

Amanda Cibele Apolinário Rosa

Dissertação de Mestrado

Uma Arquitetura Reflexiva para Injetar Falhas em Aplicações Orientadas a Objetos

Este exemplar corresponde à redação final da
Dissertação devidamente corrigida e defendida
por Amanda Cibele Apolinário Rosa e aprovada pela
Banca Examinadora.

Campinas, 20 de novembro de 1998.

Eliane Martins

Eliane Martins
(Orientadora)

Dissertação apresentada ao Instituto de Computação da
UNICAMP, como requisito parcial para
a obtenção do título de Mestre em Ciência da Computação.

FICHA CATALOGRÁFICA ELABORADA PELA BIBLIOTECA DO IMECC DA UNICAMP

Rosa, Amanda Cibele Apolinário

R71a Uma arquitetura reflexiva para injetar falhas em aplicações orientadas a objetos / Amanda Cibele Apolinário Rosa -- Campinas, [S.P. :s.n.], 1998.

Orientador : Eliane Martins

Dissertação (mestrado) - Universidade Estadual de Campinas, Instituto Computação.

1. Orientação a objeto. 2. Software - Testes. 3. Tolerância a falhas (Computação). I. Martins, Eliane. II. Universidade Estadual de Campinas. Instituto de Computação. III. Título.

Instituto de Computação
Universidade Estadual de Campinas

Uma Arquitetura Reflexiva para Injetar Falhas em Aplicações Orientadas a Objetos

Amanda Cibele Apolinário Rosa

Novembro 1998

Banca Examinadora:

- Eliane Martins (Orientadora)
  IC - UNICAMP
- José Carlos Maldonado
  ICMS - USP (São Carlos)
- Cecília M. F. Rubira
  IC - UNICAMP
- Maria Beatriz Toledo (suplente)
  IC - UNICAMP

Dissertação apresentada ao Instituto de Computação da
Universidade Estadual de Campinas, como requisito parcial para
a obtenção do título de Mestre em Ciência da Computação.

Tese de Mestrado defendida e aprovada em 09 de outubro de 1998 pela Banca Examinadora composta pelos Professores Doutores

Prof. Dr. José Carlos Maldonado

Prof. Dr. Cecília Mary Fischer Rubira

Profa. Dra. Eliane Martins

Agradecimentos

À minha orientadora, Profa. Dra. Eliane Martins, pelo amizade, incentivo e apoio dispensados ao longo da realização deste trabalho.

Aos meus familiares por sempre acreditarem e apoiarem meu esforço.

A todos meus amigos pelo incentivo e pelas momentos inesquecíveis que passamos durante o período do mestrado.

À FAPESP pelo apoio financeiro através da bolsa de mestrado processo nº 96/09931-0.

E, principalmente, à Deus por sempre me guiar.

Muito Obrigada!!

Resumo

Injeção de falhas por software é uma técnica que vem sendo muito utilizada para validar as propriedades de segurança de funcionamento (*dependability*) de sistemas de software. Essa técnica consiste em injetar padrões de erros em um software em execução. Para injetar falhas e monitorizar seus efeitos alguma forma de instrumentação deve ser introduzida na aplicação em teste (aplicação alvo). Essa instrumentação é intrusiva, ou seja, interfere na execução e na estrutura da aplicação alvo. No entanto, um dos objetivos de uma abordagem de instrumentação de software é ser o menos intrusiva possível. Isso requer que a instrumentação seja funcionalmente independente da aplicação alvo. Outras qualidades importantes de uma abordagem de instrumentação são *modularidade*, a fim de facilitar a incorporação de novas características, *reusabilidade*, para facilitar a adaptação para sistemas alvos diferentes, e *portabilidade*, para permitir o uso em plataformas diferentes com mudanças mínimas. Para alcançar esses objetivos é proposto neste trabalho o uso da programação orientada a objetos reflexiva. Reflexão reduz a interferência na aplicação alvo porque provê uma separação clara entre seus aspectos funcionais e não-funcionais, sendo os últimos relacionados aos aspectos de injeção de falhas e monitorização. O projeto de uma arquitetura de injeção de falhas reflexiva, uma ferramenta de injeção reflexiva (*FIRE - Fault Injection using a REflective Architecture*) e resultados experimentais são apresentados neste trabalho.

Abstract

Software-implemented fault injection is, nowadays, a largely used technique to validate dependability properties of software systems. This technique consists of the injection of error patterns into executing software. To inject faults and monitor their effects some form of instrumentation may be introduced into the system under test (target system). This instrumentation causes some level of intrusiveness, i.e., it imposes some interference upon the target system execution and structure. Therefore, a goal of a software instrumentation approach is to be the least intrusive possible. This requires instrumentation to be functionally independent from the target system. Other important qualities that an instrumentation should present are *modularity*, in order to allow easy incorporation of new features, *reusability*, to allow easy adaptation to different target systems, and *portability*, to allow the use in different hardware/software platforms with minimum changes. To obtain these qualities in this work the use of reflective programming is proposed. Intrusiveness in the target system is reduced, in that it allows a clear separation between functional and non-functional aspects, the later being related to fault injection and monitoring aspects. The design of a reflective fault-injection architecture, a reflective fault injection tool (FIRE - Fault Injection using a REflective Architecture) and results of experiments are presented in this work.

Conteúdo

Lista de Figuras

Lista de Tabelas

Capítulo 1

Introdução

O objetivo deste capítulo é apresentar o contexto do trabalho, sua motivação e contribuições. A terminologia da área de validação de sistemas confiáveis também é incluída devido a ausência de um consenso dos termos em português. No final do capítulo a estrutura da dissertação é apresentada.

1.1 Contexto

Sistemas computacionais são constituídos de uma série de componentes de hardware, software e outros, que podem falhar eventualmente. Essas falhas podem fazer com que os serviços oferecidos por esses sistemas não estejam de acordo com a sua especificação. Para alguns sistemas a ocorrência de defeitos, mesmo que temporários, pode implicar em custos muito elevados em termos econômicos ou até mesmo em termos de vidas humanas.

Como a ocorrência de falhas não pode ser de todo evitada, pode-se tentar contrabalançar seus efeitos através de redundância. Redundância pode ser aplicada para tolerar falhas em componentes de hardware ou software. Assim, por exemplo, para compensar o efeito de falhas de hardware pode-se usar processadores duplicados. A redundância é portanto a base das técnicas de tolerância a falhas, as quais visam garantir que o sistema continue a fornecer o serviço conforme a sua especificação, mesmo em presença de falhas de alguns de seus componentes. [Lap95]

Uma das dificuldades no desenvolvimento de sistemas tolerantes a falhas diz respeito a sua validação. Essa dificuldade vem do fato de que os testes desses sistemas requerem a consideração de situações anormais, que ativem os mecanismos de tratamento de falhas (MTF's) existentes em tais sistemas. Uma técnica que vem sendo muito utilizada para esse fim é a injeção de falhas, que consiste em introduzir erros/falhas em um sistema, de forma controlada, e observar seu comportamento.

Injeção pode ser aplicada de várias formas de acordo com a fase do ciclo de vida do sistema. Nas fases iniciais, quando o sistema pode ser representado por um modelo abstrato, a *simulação de falhas* geralmente é usada. Nesse caso, falhas lógicas são introduzidas em um modelo do sistema. Essa técnica é útil para estudar a propagação de erros e analisar medidas de segurança de funcionamento do sistema alvo. Durante as fases de testes, quando

um protótipo do hardware/software estão disponíveis, as técnicas mais usadas são a *injeção física* e a *injeção por software*. Na injeção física, um hardware dedicado, especialmente construído para esse fim, introduz falhas físicas no hardware do sistema através de técnicas como submissão de radiações, interferências magnéticas e submissão de voltagem ou mudança de correntes em pinos de circuitos integrados. Na injeção por software, erros induzidos pela ocorrência de falhas de hardware/software são introduzidos no sistema em teste por meio de controle de software. A injeção de falhas nesse caso pode ser feita em *tempo de compilação*, substituindo instruções originais por outras que injetam erros, ou em *tempo de execução*, executando um software extra que modifica o estado do programa em teste (seu conteúdo de memória e valores de registradores). Por fim essas técnicas listadas podem ser usadas em conjunto, dando origem a injetores híbridos. Segundo Carreira [Car95], alguns combinam injeção por software com hardware especial para ajudar na injeção de falhas ou para monitorar e estudar a propagação das falhas; uma outra possibilidade consiste em combinar injeção por software com simulação para tomar vantagem tanto da velocidade do processador do sistema alvo como da exatidão dos modelos de falhas de baixo-nível.

Dentre as técnicas de injeção, a injeção por software tem despertado grande interesse entre os pesquisadores da área por possuir diversas vantagens em relação as demais:

- não necessita de hardware especial, conseqüentemente, o custo é mais baixo e a portabilidade é maior comparando-se à injeção física;
- não há risco de danificar o sistema alvo, o que não é verdade na injeção física;
- oferece maior facilidade de controle e observação do sistema durante os testes, como na simulação, sem apresentar o tempo elevado de processamento dessa última e,
- pode ser usada para validar aplicações e sistemas operacionais, o que é difícil de fazer com injeção física.

Nos últimos dez anos houve um grande avanço na área de injeção de falhas e várias ferramentas de injeção por software foram desenvolvidas. Contudo, a técnica possui ainda algumas limitações. Uma limitação importante é a intrusividade, principalmente no que diz respeito à injeção em tempo de execução: devido a introdução da instrumentação necessária para injetar falhas e monitorar seus efeitos, a execução do sistema alvo é perturbada e ainda sua estrutura original, na maioria das vezes, é alterada.

Eliminar totalmente a intrusividade das ferramentas é uma tarefa muito difícil (talvez impossível), uma vez que o que caracteriza a injeção por software em tempo de execução é execução de um software extra. Algumas ferramentas conseguem reduzir a intrusividade, no entanto nesses casos, ou uma abordagem híbrida é usada como em [KKA95], na qual

algumas funções são executadas por um hardware especial, ou o ambiente de desenvolvimento ou operacional do sistema alvo oferece características que facilitam a injeção e/ou monitorização, como em [CMS95] [KJA98].

Uma possibilidade de diminuir a perturbação no sistema alvo, sem utilizar características especiais do ambiente alvo, é a introdução da instrumentação de forma que a estrutura original do programa sofra pouquíssimas alterações, e, principalmente, garantindo a separação de funcionalidades: as características de injeção e monitorização não devem ficar misturadas à funcionalidade do sistema em teste.

É possível ainda diminuir a interferência do impacto causado sobre o comportamento do sistema em teste utilizando mecanismos mais eficientes para ativar o código responsável pela injeção e monitorização. Por exemplo, algumas ferramentas, como em [LE93], utilizam a facilidade de execução no modo traço para ativar a injeção de falhas após a execução de cada instrução. Essa estratégia de ativação altera drasticamente a velocidade de execução do sistema alvo e inclusive pode alterar seus resultados, comprometendo o sucesso dos testes.

1.2 Motivação

A importância da validação das propriedades de segurança de funcionamento e as vantagens da injeção por software dentre as demais técnicas de validação representaram a motivação para o tema básico desta dissertação: injeção de falhas por software.

Como mencionado na Seção anterior, a injeção por software tem se sido alvo de grande interesse entre os pesquisadores da área de injeção de falhas, e grande e rápido avanço já foi alcançado. Porém, um ponto normalmente ainda apontado como limitação da técnica é o seu alto grau de intrusividade, evidenciando a necessidade de estudos nesse aspecto.

O problema da intrusividade não é exclusividade da injeção por software, ele é comum a técnicas que baseiam-se na incorporação de características não-funcionais a um sistema de software. Como exemplo, pode-se citar a tolerância a falhas. Quando o sistema operacional não provê protocolos de recuperação de erros, o programador de aplicações tolerantes a falhas é forçado a integrar na parte funcional da aplicação comandos para iniciar ou invocar mecanismos não-funcionais apropriados para fazer o processamento de erros [FNP+95]. Uma solução bastante elegante e eficaz utilizada por pesquisadores da área [FNP+95], [Cor97], consiste em utilizar reflexão computacional para introduzir à aplicação as características de tolerância a falhas.

Reflexão permite que a programação funcional fique separada da programação não-funcional. Isto é obtido por meio de uma nova arquitetura de software proposta: a arquitetura reflexiva. Na arquitetura reflexiva um sistema computacional é dividido em subsistemas interrelacionados: um subsistema objeto, que faz computações sobre um domínio externo ao sistema, e um subsistema reflexivo que faz computações sobre o subsistema objeto. Nos últimos dez anos, e, com maior intensidade recentemente,

pesquisadores de diversas áreas têm usado reflexão como forma de extensão de linguagens de programação e pela sua utilidade na implementação de diversos mecanismos, como por exemplo, distribuição, persistência, confiabilidade e segurança, de forma simplificada, transparente e independente do domínio da aplicação.

A idéia de utilizar reflexão computacional no cenário de injeção de falhas surge então como um solução viável para o problema de intrusividade da injeção de falhas, pelo menos no que diz respeito à perturbação na estrutura do sistema alvo. A instrumentação para injeção de falhas e monitorização pode então ser introduzida como um subsistema reflexivo o qual faz computações sobre o subsistema alvo injetando falhas e observando o seu comportamento. Quanto à perturbação no comportamento do sistema alvo, o custo adicional introduzido pela reflexão computacional é apontado como baixo pelos pesquisadores que vêm utilizando tal técnica. Segundo pesquisadores [RC97], [CM93] o *overhead* associado à computação adicional da linguagem OpenC++ 1.2 [Chi93], uma extensão de C++ que dá apoio reflexão, é desprezível em relação aos tempos de comunicação da rede e de acesso à memória estável.

A disponibilidade da linguagem reflexiva OpenC++1.2, a experiência de utilização da mesma em alguns trabalhos de tolerância a falhas por pesquisadores do Instituto de Computação da UNICAMP e do Instituto de Informática da Universidade Estadual do Rio Grande do Sul foram outros pontos que motivaram o desenvolvimento de uma ferramenta de injeção por software baseada em uma arquitetura reflexiva, utilizando OpenC++1.2.

Como OpenC++1.2 é uma linguagem reflexiva orientada a objetos, os possíveis sistemas alvo da ferramenta ficam restritos a aplicações tolerantes a falhas orientadas a objetos, mais especificamente as implementadas em C++ ou OpenC++1.2. No entanto, existe uma grande gama de aplicações que se encaixam nessa categoria. Por estruturar melhor os componentes de um sistema, orientação a objetos tem sido considerada pela comunidade científica em geral como um modelo de programação efetivo para a produção de software de melhor qualidade. Em particular, programação orientada a objetos vem sendo muito utilizada para implementar aplicações tolerantes a falhas.

1.3 Contribuições

Entre as contribuições feitas com o desenvolvimento desta dissertação, estão:

- a implementação de uma ferramenta de injeção de falhas por software baseada em uma arquitetura reflexiva e,
- um estudo da aplicabilidade da reflexão computacional para injeção de falhas por software para validar aplicações reflexivas e não-reflexivas.

1.4 Terminologia

Não existe ainda um consenso quanto à terminologia da área de tolerância a falhas a ser usada em português; os termos usados neste trabalho estão baseados em [Lei87].

- Segurança de funcionamento (*dependability*). Pode ser definida como a qualidade do serviço fornecido pelo sistema, tal que confiança possa ser justificadamente depositada nesse serviço.

- Falha (*fault*). Uma falha é causa direta de um erro. Exemplo: um curto-circuito em um circuito integrado.

- Erro (*error*). O sistema está em um estado de erro, quando qualquer outro processamento pode levar o sistema a comportar-se de forma distinta do especificado, ou seja, um erro é parte do estado do sistema que pode levar a um defeito. Exemplo: modificação da função do circuito.

- Defeito (*failure*). Um sistema é dito ter um defeito quando o serviço por ele oferecido não está de acordo com a sua especificação. Em outras palavras, um defeito ocorre quando um erro passa pela interface do sistema e afeta o serviço oferecido pelo mesmo.

- Validação. O processo de validação da segurança de funcionamento consiste na verificação, para retirada de falhas, e na avaliação, para estimar medidas de confiabilidade, disponibilidade, eficiência dos MTF's, entre outras.

1.5 Organização do texto

O Capítulo 2 apresenta alguns conceitos sobre validação da confiabilidade e descreve a técnica de injeção de falhas. O Capítulo 3 descreve o cenário da injeção de falhas por software: modelos de falhas, métodos de injeção, algumas ferramentas desenvolvidas e as vantagens e limitações da técnica. O Capítulo 4 destina-se a explicar reflexão computacional orientada a objetos, pontos como arquitetura reflexiva, o casamento entre reflexão e orientação a objetos e a utilização de reflexão em algumas áreas são abordados. O Capítulo 5 descreve a ferramenta desenvolvida e ilustra sua utilização. O Capítulo 6 discute os aspectos de implementação da ferramenta. O Capítulo 7 mostra como foi feita a validação da ferramenta e apresenta os resultados obtidos. Finalmente, o Capítulo 8 apresenta as considerações finais: conclusões e futuros trabalhos.

Capítulo 2

Validação da Segurança de Funcionamento

O objetivo deste capítulo é destacar a importância da validação de sistemas tolerantes a falhas, apresentar os métodos de validação até então desenvolvidos e, por fim, mostrar que a injeção de falhas por software é o método mais vantajoso de validação e o mais adequado quando o objetivo é validar os mecanismos de tolerância a falhas implementados em software.

2.1 Introdução

Sistemas computacionais têm sido utilizados na construção das mais variadas aplicações, de eletrodomésticos a satélites e centrais nucleares. É difícil imaginar o mundo sem computadores uma vez que eles são responsáveis por tarefas fundamentais. Em alguns casos nossas próprias vidas dependem de computadores como no tráfego aéreo, controle de trens e sistemas médicos. Nessas aplicações críticas qualquer falha, mesmo que temporária, pode ter conseqüências catastróficas. Em um outro contexto, computadores são utilizados para responder questões fundamentais das ciências modernas, e geralmente estas aplicações levam um longo tempo de processamento. Nesses casos, qualquer falha pode interromper ou alterar o serviço prestado pelo sistema, destruindo o trabalho de semanas, ou em alguns casos os erros provocados podem passar despercebidos e conduzir as aplicações a gerarem resultados errados. Em ambos os casos o funcionamento adequado dos sistemas computacionais deve ser garantido durante todo o seu período operacional.

Infelizmente, sistemas computacionais não estão livres da ocorrência de falhas, que podem ser falhas de hardware (memória, processadores, barramentos, etc) ou de software (falhas humanas de projeto/implementação). Segundo Laprie [Lap95], os meios de obter-se segurança de funcionamento em sistemas computacionais podem ser divididos em quatro classes:

- *Prevenção de falhas:* o objetivo é prevenir a ocorrência de falhas através de metodologias de projeto, regras de implementação e técnicas de garantia da qualidade.

- *Remoção de falhas:* o objetivo é reduzir a presença de falhas usando mecanismos de verificação, diagnóstico e correção.

- *Previsão de falhas:* o objetivo é estimar, por meio de avaliação, a incidência e as conseqüências de falhas futuras e obter medidas probabilísticas que permitam caracterizar o comportamento do sistema na presença de falhas.

- *Tolerância a falhas:* o objetivo é garantir que o sistema continue a fornecer o serviço conforme a sua especificação mesmo em presença de falhas de alguns de seus componentes. Isso é obtido através da incorporação de mecanismos de detecção e recuperação de erros. As técnicas de tolerância a falhas são baseadas em redundância. Redundância pode ser dividida em duas categorias: redundância de hardware ou redundância de software. A redundância de hardware é obtida através da replicação de componentes críticos do hardware. Em software a replicação de componentes não é suficiente para garantir a tolerância a falhas, pois caso um componente de software falhar uma réplica idêntica também falhará. Assim, a redundância de software é obtida através da replicação de componentes de software com diversidade de projeto.

A *prevenção de falhas* e a *tolerância a falhas* são estratégias usadas para prover a um sistema computacional as propriedades de segurança de funcionamento. A *remoção de falhas* e a *previsão de falhas* são estratégias usadas *na validação das propriedades de segurança de funcionamento*. Ou seja, o uso de técnicas que permitam evitar e/ou tolerar falhas por si só não são suficientes para garantir a confiança no serviço fornecido pelos sistema. Métodos de *prevenção de falhas* e *tolerância a falhas* não estão livres de falhas; portanto, é necessário o uso de técnicas para *remoção* de falhas residuais existentes no projeto e/ou implementação de sistemas. Estas técnicas, por sua vez, não são perfeitas, possibilitando ainda a ocorrência de falhas, sendo assim importante fazer uma *previsão* do efeito das mesmas sobre o sistema.

Um dos maiores problemas no desenvolvimento de sistemas confiáveis é como validar suas propriedades de segurança de funcionamento. Geralmente, há muitos atributos de segurança de funcionamento tais como confiabilidade, disponibilidade ,manutenibilidade e medidas relacionadas a performance. [Lap95]

A complexidade crescente do hardware e do software tende a tornar essa tarefa ainda mais difícil. Muitas abordagens têm sido propostas para validar as propriedades de segurança de funcionamento de um sistema. Basicamente, essas abordagens podem ser classificadas como analíticas e experimentais. Essas abordagens serão apresentadas adiante nesse capítulo. A avaliação experimental por injeção de falhas tem sido reconhecida como uma técnica muito poderosa para validação da segurança de funcionamento.

O teste por injeção de falhas - o qual consiste em introduzir erros/falhas em um sistema, de forma controlada, e observar seu comportamento - permite que sejam cobertos os dois

aspectos da validação. Na remoção de falhas, o objetivo da injeção é reduzir ao máximo as falhas de projeto/implementação existentes nos MTF's. Na previsão de falhas, o objetivo é avaliar a eficiência dos MTF's, obtendo estimativas de parâmetros tais como fator de cobertura e latência[1]. Segundo Martins [Mar95], na grande maioria das ferramentas de injeção de falhas os testes visam particularmente a previsão de falhas e os trabalhos que tratam da eliminação de falhas de maneira sistemática, em geral, não englobam a previsão de falhas.

O restante do capítulo segue a seguinte organização: a Seção 2.2 apresenta os métodos de validação até então desenvolvidos; a Seção 2.3 detalha as características do teste por injeção de falhas, e finalmente a Seção 2.4 resume e conclui este capítulo.

2.2 Métodos de Validação

Validação pode ser feita estaticamente através de provas formais e modelagem analítica ou dinamicamente por métodos experimentais.

O uso de técnicas formais de verificação, como prova de programas, é altamente desejável para garantir a correção do sistema, em particular, de seus mecanismos de tolerância a falhas. O uso de modelos analíticos, também é necessário para a obtenção de medidas da eficiência desses mecanismos. No entanto, devido à complexidade dos sistemas tolerantes a falhas, sua aplicação é limitada a algumas partes do sistema e à consideração de um número reduzido de falhas. Essas técnicas devem então ser complementadas com a validação experimental. A Subseção 2.2.1 apresenta a abordagem analítica e a Subseção 2.2.2 apresenta a abordagem experimental.

2.2.1 Abordagem Analítica

A abordagem analítica normalmente consiste em duas fases:

- a construção de um modelo probabilístico do sistema a partir de processos estocásticos elementares, o qual engloba o comportamento dos componentes do sistema e as interações entre eles e,
- o processamento do modelo para obter as expressões e os valores das medidas da segurança de funcionamento do sistema.

[1] Cobertura e latência são estimativas de parâmetros que caracterizam o comportamento operacional dos MTF's em relação ao potencial de detecção de erros e tratamento de falhas e ao intervalo de tempo entre a ativação e detecção das falhas, respectivamente.

A construção de modelos analíticos de sistemas reais é muito difícil uma vez que os mecanismos envolvidos no processo de ativação das falhas e de propagação de erros são altamente complexos e não são completamente conhecidos na maioria dos casos. Além disso, os modelos podem ficar muito grandes na prática, e as suposições de simplificação feitas para tornar a análise tratável reduz a utilidade dos resultados produzidos por essa abordagem [CMS95].

Por um outro lado, a precisão dos modelos depende dos parâmetros de segurança de funcionamento usados pelos componentes do sistema, os quais incluem distribuições de tempo e probabilidades. Esses parâmetros são difíceis de estimar, sendo desejável usar parâmetros obtidos através de testes experimentais do sistema. Portanto, sem as informações providas pela análise experimental é difícil construir com exatidão modelos analíticos do comportamento do software na presença de falhas [KI94].

2.2.2 Abordagem Experimental

Segundo Hsueh [HTI97], existem dois tipos de validação experimental da confiabilidade de sistemas: a análise baseada em medidas[2] e a injeção de falhas.

A análise baseada em medidas consiste em executar o sistema alvo usando dados reais, monitorizar o seu comportamento, registrando a ocorrência de falhas, erros e defeitos, e posteriormente, fazer uma análise dos dados coletados. Tal análise permite um entendimento sobre características de erros e falhas reais e auxiliar na percepção de modelos analíticos. Contudo, esse tipo de validação experimental pode ser muito difícil, uma vez que o intervalo entre defeitos (MTBF: *meantime-between-failure*) pode ser muito grande, principalmente em sistemas altamente confiáveis, exigindo um longo período de execução do sistema alvo. Dessa forma, para verificar a segurança de funcionamento de sistemas tolerantes a falhas, deve haver um modo de acelerar a ocorrência de falhas, erros ou defeitos no sistema alvo.

O teste por injeção de falhas baseia-se na introdução deliberada de falhas em um sistema a fim de observar seu comportamento [AAA+90]. Ao invés de analisar inúmeros experimentos, nos quais falhas podem ou não acontecer, a injeção introduz o conceito de aceleração da ocorrência de falhas a fim de validar MTF's durante a execução de programas.

Como a injeção de falhas é a técnica de validação utilizada neste trabalho, a próxima Seção apresenta uma visão um pouco mais completa da técnica, com o objetivo de situar o leitor no contexto deste trabalho.

---

[2] Do inglês, *measurement-based analysis.*

2.3 Injeção de falhas

Injeção de falhas consiste em uma sequência de testes cujos padrões de entrada, as falhas, são selecionados de acordo com modelos de falhas/erros que pretendem simular, tanto quanto possível, as conseqüências da ativação de falhas reais [AT95]. Essa técnica vem sendo muito utilizada, pois permite a validação de MTF's em presença de entradas especiais para as quais eles foram construídos para tratar: as falhas. Na remoção de falhas, a injeção tem por objetivo verificar se o comportamento dos MTF's está de acordo com a sua especificação. Na previsão de falhas o objetivo é obter medidas da eficiência desses MTF's, tais como cobertura e latência. Mais recentemente, injeção de falhas também vem sendo usada para quantificar cenários de riscos para componentes de software [VMK+97].

Esta Seção está organizada da seguinte forma: a Subseção 2.3.1 caracteriza a injeção de falhas em função dos conjuntos FARM (Falhas, Ativadores, Resultados e Medidas). A Subseção 2.3.2 apresenta as formas de aplicação da injeção de falhas: simulação, injeção física, injeção por software e técnicas híbridas. E, finalmente, a Subseção 2.3.3 compara tais formas de aplicação.

2.3.1 Caracterização

A injeção de falhas é caracterizada por um domínio de entrada e um domínio de saída. O domínio de entrada consiste em um conjunto de falhas a injetar **F**, e um conjunto de ativações **A** que especifica as entradas destinadas a exercitar o sistema e, conseqüentemente, as falhas injetadas. O domínio de saída corresponde a um conjunto **R** de dados coletados e a um conjunto **M** de medidas derivadas da análise e do tratamento dos conjuntos **F**, **A** e **R**. Os conjuntos **FARM** são os principais atributos da injeção de falhas [AT95].

- O conjunto **F**. O conjunto de falhas a injetar depende das características do sistema em teste e do nível de abstração no qual as falhas são injetadas (portas lógicas, chip, sistema ou rede). Em qualquer abordagem de injeção é importante garantir que o conjunto de erros produzidos pela injeção seja o mais próximo possível dos erros produzidos por falhas reais. Assim, um ponto muito importante no teste por injeção de falhas é a definição do modelo de falhas (ver próximo capítulo, Seção 3.2).

- O conjunto **A**. O conjunto de dados de entrada deve ser cuidadosamente determinado, pois é ele que ativa as falhas injetadas.

- O conjunto **R**. Quanto aos dados de saída, deve-se fazer uma análise em relação a execução do sistema em teste. A execução pode terminar antes de atingir o ponto final esperado. Isso pode ser forçado (1) por uma ferramenta de injeção de falhas devido a um estouro de *time-out*; (2) pelo sistema operacional; ou (3) pelo mecanismo de

tolerância a falhas caso ele tenha detectado um erro que não é capaz de tolerar. Esses três casos indicam a ocorrência de defeitos. Se um programa termina normalmente seus resultados podem não estar corretos devido a um erro não detectado, sendo então necessário verificar se seus resultados estão corretos. Esse problema é denominado *problema do oráculo*. Na maioria dos trabalhos de injeção de falhas a solução empregada consiste em comparar os resultados obtidos após as injeções com aqueles obtidos durante a execução sem injeção, a qual serve como uma referência.

- O conjunto **M**. Na maioria dos estudos, o objetivo visado é a avaliação de parâmetros tais como fator de cobertura e latência. Esses parâmetros podem ser usados tanto na construção de modelos analíticos para o cálculo da confiabilidade ou do desempenho, como em [KI94], quanto na validação de modelos preexistentes, como em [RS93].

A seqüência de testes por injeção de falhas é composta por uma série de experimentos, cada qual sendo caracterizado por um ponto no espaço **FxAxR**. O número de experimentos depende do objetivo da validação (remoção ou previsão de falhas) como mostra a Tabela 2.1.

	Remoção de falhas	Previsão de falhas
Objetivo	Eliminar o maior número possível de falhas de projeto/implementação do sistema.	Obter medidas da eficiência dos mecanismos de tolerância a falhas do sistema.
F	Número pequeno de falhas específicas, determinadas a partir das hipóteses feitas na fase de projeto do sistema.	Número elevado de falhas, representativas das falhas que possam ocorrer em fase operacional.
A	Entradas destinadas a exercitar as funções do sistema e ativar as falhas injetadas.	Entradas representativas do perfil operacional do sistema.
R	Indicações de ocorrência de eventos e/ou medidas de tempo associadas e informações para ajudar no diagnóstico ("dumps" de memória, ...)	Número de ocorrências de eventos e medidas de tempo associadas.
M	Veredicto: valor indicando se o sistema comporta-se como o esperado.	Estatísticas sobre a ocorrência de eventos e dos tempos entre as ocorrências.

Tabela 2.1 Os atributos FARM e os objetivos da validação da tolerância a falhas [Mar95].

2.3.2 Formas de Aplicação

Vários métodos foram desenvolvidos pela comunidade científica para injetar falhas em sistemas computacionais e estudar o seu comportamento na presença destas falhas. Embora haja um grande número de ferramentas e técnicas, os métodos desenvolvidos podem ser divididos em quatro classes de acordo com a forma de aplicação: simulação, injeção física, injeção por software e técnicas híbridas (que combinam as técnicas anteriores). [CMS95]

Injeção física de falhas

Consiste em aplicar falhas físicas a um protótipo do hardware do sistema. Essa técnica tem a vantagem de poder causar falhas de hardware verdadeiras, que podem estar bem próximas de um modelo de falhas realistas. Contudo, as falhas são injetadas por um dispositivo de hardware especialmente construído para esse fim, que interage com um sistema específico em teste, dificultando a portabilidade. Além disso, a grande complexidade e alta velocidade dos processadores atuais tornam o desenvolvimento desse hardware específico muito difícil, ou até mesmo impossível.

Os maiores problemas dessa técnica são as dificuldades para controlar e observar o efeito das falhas no interior do processador alvo, assim como a dificuldade para detectar falhas ativas. Por exemplo, segundo Carreira [CMS95], a injeção de falhas em pinos de um processador requer o uso de um hardware complexo de monitorização para saber se as falhas produziram erros internos no processador ou não, e a injeção feita através de radiação requer que os resultados do chip alvo sejam comparados pino-por-pino e ciclo-por-ciclo com a unidade principal (*gold unit*) para saber se as falhas injetadas produziram erros.

Dependendo do tipo de falha e da sua localização os métodos de injeção física podem ser divididos em dois grupos [HTI97]:

- *Injeção física com contato* - o injetor tem contado físico direto com o sistema alvo, produzindo externamente voltagem ou mudança de corrente nos pinos do circuito integrado.

- *Injeção física sem contato* - o injetor não tem nenhum contato físico direto com o sistema alvo. Ao invés disso, uma fonte externa produz algum fenômeno físico, tais como radiação de íons pesados e interferência magnética alterando o interior do circuito alvo.

Nesses casos os testes visam estudar principalmente o comportamento dos MTF's implementados em hardware, mas também pode-se testar software tolerante a falhas usando esse método. [Mar95]

Simulação de falhas

A injeção de falhas baseada em simulação é normalmente empregada para validar as propriedades de segurança de funcionamento de um sistema que está em fase conceitual ou de projeto. Nesse ponto o sistema alvo é apenas uma série de abstrações de alto nível (detalhes de implementação ainda não foram determinados). O sistema alvo é então simulado em um outro computador usando ferramentas de simulação complexas. Por exemplo, no caso de um simulador ao nível de portas lógicas, é preciso que o testador tenha disponível uma descrição completa do hardware do processador alvo. De fato essa descrição existe para todos os processadores, na maior parte em formato padrão (VHDL, por exemplo), mas normalmente não é disponível para a maioria dos pesquisadores.

Nesse tipo de injeção, as falhas são introduzidas em um modelo do sistema alvo, alterando valores lógicos durante a simulação. As falhas podem ser injetadas por módulos especialmente construídos para esse fim, ou através de mutações do modelo original.

Simulação permite controlar o tempo, o tipo de falha e o componente afetado no modelo, portanto, é útil para estudar o processo de ativação das falhas e de propagação de erros. Contudo, essa técnica envolve um grande esforço de desenvolvimento e sua utilização pode ser muito morosa devido ao elevado tempo da simulação. Além disso, não há garantia de que a simulação corresponderá à implementação real.

Injeção de falhas por software (SWIFI - *Software Implemented Fault Injection*)

Recentemente, o interesse de pesquisadores tem se voltado para o desenvolvimento de ferramentas de injeção de falhas por software (ou simplesmente injeção por software). Essa técnica introduz falhas lógicas no software do sistema com o objetivo de emular conseqüências de falhas de hardware/software. A injeção por software tornou-se atrativa porque não requer um equipamento específico e caro para a sua aplicação. Além disso, pode ser usada para validar aplicações e sistemas operacionais, o que é difícil com injeção física. [HTI97]

Quando usado para validar aplicações tolerantes a falhas, o injetor é inserido na própria aplicação alvo, ou em uma camada entre a aplicação e o sistema operacional. Se o alvo é o sistema operacional, o injetor de falhas deve ser inserido no próprio sistema operacional, já que é muito difícil adicionar uma camada entre a máquina e o sistema operacional. A injeção por software é a técnica abordada neste estudo, e está detalhadamente apresentada no Capítulo 3.

Técnicas híbridas

Essa abordagem resulta de uma mistura das técnicas anteriores. Segundo Carreira [CMS95], alguns combinam SWIFI com hardware especial para ajudar na injeção de falhas ou para

monitorizar e estudar a propagação das falhas; uma outra possibilidade consiste em combinar SWIFI com simulação para tomar vantagem tanto da velocidade do processador do sistema alvo como da exatidão dos modelos de falhas de baixo-nível.

2.3.3 Comparação das Técnicas de Injeção de Falhas

Muitos métodos para injeção física de falhas foram propostos, tais como injeção a nível de pinos e radiação de íons pesados. Contudo, essas ferramentas têm se tornado cada vez mais difíceis de desenvolver e usar, principalmente devido a complexidade das arquiteturas dos computadores atuais e o alto nível de integração das funções de um circuito integrado.[CMS95]

Como resultado, abordagens de simulação, métodos de injeção por software e técnicas híbridas foram desenvolvidos como alternativas viáveis à injeção física. Uma boa apresentação de ferramentas de injeção de falhas para todos esses métodos é apresentada em [CMS95].

A simulação de falhas apresenta a vantagem de poder ser realizada já nas fases iniciais de um projeto, começando a validação a partir de subsistemas do sistema alvo e considerando um nível de abstração mais elevado. Porém, os custos dessa técnica são muito elevados tanto em termos de dificuldades de desenvolvimento como em termos de tempo de processamento.

O contraste entre os métodos físicos e os de software diz respeito, principalmente, aos pontos de injeção que eles podem ter acesso, ao custo e ao nível de perturbação que eles introduzem.

A injeção física pode injetar falhas em pinos e componentes internos de chips, o que não é possível com o software. Por outro lado, a injeção por software muda o estado do software (memória, barramentos, etc). Assim, injeção física é mais adequada quando é necessário validar MTF's de nível mais baixo e injeção por software é propícia para validar os MTF's implementados em um nível de abstração mais elevado, por ter a capacidade de injetar erros específicos que ativam esses mecanismos, o que não pode ser garantido na injeção física.

Desde que a funcionalidade do hardware é altamente visível através do software, falhas de hardware podem ser emuladas, permitindo a injeção de falhas tanto de software como de hardware, o que não é possível na injeção física.

Além disso, é muito mais fácil repetir experimentos de injeção por software e coletar um maior número de dados.

Os métodos por software são menos caros em termos de esforço e tempo, entretanto, na maioria dos casos, eles provocam um alto grau de intrusividade porque executam software no sistema alvo.

A Tabela 2.2 resume as vantagens e desvantagens das três principais abordagens de injeção de falhas.

Abordagem	Vantagens	Desvantagens
Injeção Física	• usa software hardware reais; • provoca falhas reais;	• requer hardware especial dedicado a um sistema específico, cujo desenvolvimento é difícil e caro; • pode danificar o sistema alvo; • não monitora o software do sistema alvo, e assim, não pode injetar falhas em estados do sistema (por exemplo, alta utilização da CPU); • tem dificuldades de monitorização da ativação e detecção de falhas; • é difícil automatizar o processo;
Simulação	• permite o controle do tempo, do tipo de falha e do componente afetado no modelo; • pode ser aplicada ainda nas fases de projeto e concepção;	• exige um longo processamento; • exige parâmetros de entrada exatos; • o desenvolvimento de simuladores é bastante difícil e custoso;
Injeção por Software	• usa software real; • não é necessário que o hardware onde o sistema alvo vai residir esteja disponível; • esforço e custo de desenvolvimento reduzidos, pois não necessita de equipamento especial de hardware; • oferece maior facilidade na monitorização, como na simulação, sem apresentar o tempo elevado de processamento desta última; • maior portabilidade; • expansível (para novas classes de falhas, por exemplo); • não tem problemas com interferências físicas; • não há riscos de danificar o sistema alvo; • é adequada para validar sistemas a níveis de abstração mais elevados (aplicação, sistema operacional), o que é difícil através da injeção física;	• caso os modelos de falhas não sejam bem definidos e organizados os experimentos podem obter resultados pouco representativos; • não tem acesso a recursos escondidos do software (linhas de controle, temporizadores, etc); • pode alterar o comportamento e a estrutura do sistema alvo, e interferir nos resultados (ver no capítulo 3);

Tabela 2.2 Vantagens e desvantagens das abordagens de injeção de falhas.

2.4 Resumo

Atualmente sistemas tolerantes a falhas têm sido projetados e usados em inúmeras e diversas aplicações. Para que confiança possa ser justamente depositada em tais sistemas eles devem ser devidamente validados. A validação deve verificar se o sistema detecta e/ou tolera todas as falhas para as quais ele foi projetado para lidar, e ainda mais, avaliar a sua eficiência no cumprimento desse objetivo. Existem basicamente duas formas de validar a confiabilidade de tais sistemas: analítica e experimentalmente.

Os modelos analíticos são complexos devido a variedade de parâmetros, e, além disso, os pressupostos simplificadores usados para tornar a análise tratável reduzem a utilidade dos resultados. Normalmente a validação experimental é usada para complementar a validação analítica. Dentre os métodos experimentais, a injeção de falhas tem se mostrado o meio mais atraente, por acelerar a ocorrência de falhas específicas e permitir a validação dos MTF's.

Injeção de falhas pode ser aplicada de várias formas de acordo com a fase do ciclo de vida do sistema. Nas fases iniciais, quando o sistema pode ser representado por um modelo abstrato, a simulação de falhas geralmente é usada. Nesse caso, falhas lógicas são introduzidas em um modelo representando o sistema. Durante as fases de testes, quando um protótipo de hardware e/ou software estão disponíveis, as técnicas mais usadas são a injeção física e a injeção por software. Por fim, as técnicas podem ser combinadas, dando origem aos modelos híbridos.

Na injeção física, um hardware dedicado introduz falhas físicas em circuitos integrados (CI's). Falhas podem afetar os CI's externamente ou internamente. A injeção de falhas diretamente nos pinos do circuito é um exemplo de injeção de falhas externas, na qual a funcionalidade do CI é alterada. A submissão de componentes à radiação de íons-pesados ou interferência eletromagnética pode gerar falhas internas no circuito.

Na injeção por software, erros induzidos pela ocorrência de falhas de hardware ou software são introduzidas por software. Embora as primeiras ferramentas de SWIFI tenham aparecido muito depois das ferramentas de injeção física, elas têm apresentado enorme avanço desde que técnica foi proposta. Por outro lado, cada vez menos injetores físicos têm sido construídos e os existentes tornaram-se obsoletos. As principais vantagens da SWIFI são: baixa complexidade, baixo esforço de desenvolvimento e baixos custos. Em adição, as ferramentas desenvolvidas têm apresentado maior portabilidade e facilidade de expansão (para outras classes de erros, por exemplo). Apesar dessas características atrativas, a injeção por software ainda apresenta algumas limitações, uma importante delas é a intrusividade em relação ao comportamento e à estrutura do sistema alvo.

Como o objetivo deste trabalho é o desenvolvimento de uma ferramenta de injeção por software, o próximo capítulo é dedicado a apresentar detalhadamente tal técnica.

Capítulo 3

Injeção de Falhas por Software

O objetivo deste capítulo é apresentar os diversos aspectos envolvidos na injeção de falhas por software, apresentar as características de algumas ferramentas selecionadas e apontar algumas limitações da técnica.

3.1 Introdução

O objetivo da técnica de injeção de falhas por software[3] é modificar o estado do hardware/software do sistema alvo por meio de controle de software, fazendo com que o sistema se comporte como se uma falha tivesse ocorrido. [KKA92]

Injeção por software, é uma técnica que vem sendo muito utilizada para validar as propriedades de segurança de funcionamento de sistemas de computacionais e diversas ferramentas para tal propósito têm sido desenvolvidas. Tais ferramentas apresentam modelos de falhas e métodos de injeção diversos, que variam conforme o alvo da injeção de falhas, (mecanismos de tolerância a falhas, aplicação, sistema operacional ou canais de comunicação) e as dificuldades ou facilidades oferecidas pelo ambiente operacional ou de desenvolvimento do sistema alvo.

Quanto ao modelo de falhas, a maioria das ferramentas injetam falhas de hardware, ou seja, fazem com que o sistema alvo se comporte como se uma falha no sistema de memória ou de barramentos, no(s) processador(es), no sistema de disco ou nos canais de comunicação tivesse ocorrido. No entanto, conforme citado em [KIT93] estudos mostram que grande parte dos erros observados no software são causados por falhas de projeto e/ou implementação e, assim, algumas ferramentas passaram a incluir falhas de software em seus modelos.

[3] De acordo com a terminologia apresentada no capítulo 1, o termo mais adequado para a técnica seria injeção de erros por software, uma vez que são inseridas conseqüências de falhas de hardware ou software. No entanto, para não entrar em contradição com a literatura, o termo injeção de falhas por software é utilizado para significar que a ativação das mesmas é a causa dos erros manifestados no software. O termo injeção por software também é utilizado.

Quanto ao método de injeção, as ferramentas, normalmente, corrompem a imagem do programa alvo na memória ou alteram as instruções do seu código fonte ou objeto para simular a ocorrência de falhas. Essas duas formas de injeção de falhas ilustram os dois tipos de métodos de injeção: injeção durante a compilação e injeção durante a execução.

Apesar do grande avanço na área de injeção de falhas por software, as ferramentas desenvolvidas ainda possuem algumas limitações. Tais limitações são relativas, principalmente, à perturbação sobre o sistema alvo, decorrente da execução de um software adicional para controlar a injeção e monitorização.

Este capítulo está organizado da seguinte forma: a Seção 3.2 destina-se aos modelos de falhas; a Seção 3.3 apresenta os métodos injeção por software existentes; a Seção 3.4 ilustra algumas ferramentas selecionadas; a Seção 3.5 discute as limitações da técnica e a Seção 3.6 resume e conclui o capítulo.

3.2 Modelo de Falhas

Um ponto muito importante no teste por injeção de falhas é a definição do modelo de falhas, o qual depende das características do sistema alvo e do nível de abstração no qual as falhas são injetadas (portas lógicas, chip, sistema ou rede). Qualquer que seja a abordagem de injeção é importante garantir que o conjunto de erros produzidos pela injeção seja o mais próximo possível dos erros produzidos por falhas reais, permitindo que a validação da segurança de funcionamento seja a mais precisa possível.

Idealmente, um modelo de falhas deve cobrir os dois objetivos da validação da segurança de funcionamento: verificação e avaliação. Para fins de verificação, um modelo de falhas adequado deve permitir injetar uma determinada instância de falha, em uma determinada localização, em um determinado momento. Para fins de avaliação, o modelo de falhas deve permitir a geração de um conjunto de falhas que corresponda a falhas reais que possam ocorrer durante o período operacional do sistema a ser testado.

Esta Seção descreve os modelos de falhas geralmente usados por ferramentas de injeção por software. Nesses modelos, as falhas são classificadas em: falhas de hardware que conduzem a erros de software (erros de software induzidos pelo hardware) e falhas de software (falhas de projeto/implementação do software). As subseções abaixo apresentam esses tipos de falhas.

Uma outra dimensão de classificação de falhas é baseada no padrão de repetição: falhas podem ser transientes (nunca repetem), intermitentes (repetem em intervalos predefinidos) e permanentes (sempre repetem).

3.2.1 Falhas de Hardware

Embora falhas de hardware possam ocorrer em qualquer parte de um computador, apenas aquelas que afetam, direta ou indiretamente, a execução de programas têm sido consideradas pelas ferramentas existentes. Tais falhas são classificadas em falhas de memória, falhas de processador, falhas de barramentos, falhas de I/O e falhas de comunicação.

Falhas de Memória

Correspondem a falhas no sistema de memória do sistema alvo. O espaço de memória de um processo é dividido em muitas partes: uma para o segmento de código, outra para a pilha e uma ou mais para o segmento de dados. Falhas de memória podem ser simuladas por software escrevendo-se um valor inválido em qualquer localização do espaço de memória do processo da aplicação alvo. A localização pode ser escolhida explicitamente ou usando um gerador de números aleatórios.

Falhas de Processador

Correspondem a falhas em algum dos blocos funcionais do(s) processador(es) do sistema computacional alvo. Normalmente os blocos considerados são: registradores de propósito geral, registradores especiais (tais como, contador de programa - CP), unidade de lógica e aritmética (ULA), decodificador, e barramentos internos de dados e de endereços. A Tabela 3.1 apresenta as principais falhas de processador e os procedimentos utilizados para simular a ocorrência de tais falhas.

Componente falho	Procedimento
Registradores	O conteúdo do registrador alvo deve ser corrompido.
Unidade aritmética	O conteúdo do destino da operação aritmética (memória ou registrador) deve ser modificado depois da execução de uma instrução aritmética.
Unidade lógica	O valor do registrador de *status* é modificado depois de uma operação lógica.
Decodificador de *opcode*	A instrução a ser executada deve ser modificada.
Barramento interno de endereços	Registradores incorretos, fonte ou destino, devem ser acessados.
Barramento interno de dados	Um valor incorreto deve ser escrito no registrador destino ou lido do registrador fonte.

Tabela 3.1 Falhas de Processador.

Falhas de Barramentos

Correspondem a falhas nos barramentos externos de dados e endereços. Os procedimentos que simulam a ocorrência desse tipo de falha dependem da operação que está sendo executada: busca de uma instrução, ou busca/armazenamento de dados (ver Tabela 3.2).

Barramento	Operação	Procedimento
Endereço	Busca de uma instrução	São simuladas mudando o conteúdo do CP.
	Busca ou armazenamento de operandos	A instrução deve ser decodificada e modificada tal que o endereço efetivo do operando seja modificado.
Dados	Busca de uma instrução	São simuladas modificando o conteúdo do endereço apontado pelo PC.
	Busca de operandos	São simuladas modificando o conteúdo da localização fonte antes da execução da instrução.
	Armazenamento de operandos	São simuladas modificando o conteúdo da locação destino depois da execução da instrução.

Tabela 3.2 Falhas de Barramentos

Falhas de I/O

Correspondem a falhas no sistema de disco do computador alvo. Poucas ferramentas permitem injetar esse tipo de falha, geralmente os componentes alvo são memória e processador. Falhas de I/O podem ocorrer em qualquer operação de acesso a disco (abertura, leitura, escrita e fechamento).

Falhas de Comunicação

Falhas de comunicação são falhas de mais alto nível e correspondem a alteração, perda, retardo ou duplicação de mensagens em sistemas distribuídos. A alteração de mensagens pode ser simulada por software da mesma forma que a alteração da memória, escrevendo um valor incorreto na mensagem. Mensagens perdidas simplesmente não são entregues ao receptor. O retardo de mensagens pode ser simulado capturando a mensagem, permitindo algum processamento e depois enviando a mensagem para o receptor. Enviando uma mensagem mais de uma vez pode-se simular a duplicação de mensagens.

3.2.2 Falhas de software

Falhas de software englobam falhas de implementação e projeto provocadas por humanos durante o desenvolvimento de sistemas. São poucos os mecanismos que visam a tolerância dessas falhas e, talvez por esse motivo, são poucos os trabalhos de validação da segurança de funcionamento relacionados à injeção de falhas de software.

Vários estudos foram feitos a fim de descobrir a causa, a freqüência e as correlações das falhas de software. Esses estudos apresentam classificações diferentes para as falhas de software.

Em [KIT93] falhas de software são classificadas de acordo com sua causa. Especificamente, falhas são classificadas em falhas de inicialização, falhas de atribuição, falhas de decisão e falhas de função. Esses tipos de falhas são muito semelhantes a alguns operadores de mutação utilizados pela técnica de Análise de Mutantes [VBD+97].

Falhas de Inicialização

Essas falhas incluem variáveis não inicializadas e variáveis/parâmetros incializados de forma incorreta. O valor de uma variável não inicializada depende do compilador. Por exemplo, na linguagem C o valor de incialização será 0 se a variável for global e desconhecido se a variável for local.

Uma falha de inicialização pode ser injetada mudando as instruções que inicializam uma área de dados particular, de forma que o valor para inicialização seja incorreto ou nenhuma inicialização seja feita (incluindo instruções *nop*'s).

Falhas de Atribuição

Correspondem a omissões de comandos de atribuição ou a atribuições erradas. Uma falha de atribuição errada pode ocorrer no lado direito do comando causando um valor de dado incorreto (por exemplo, usar `a = b + c` ao invés de `a = b + d`, corrompe `a`) ou no lado esquerdo provocando a corrupção de dois valores (por exemplo, usando `a = b + c` ao invés de `d = b + c` corrompe `a` e `d`).

Uma falha de atribuição pode ser injetada alterando as instruções correspondentes de forma que (1) o destino receba um valor errado, (2) uma expressão avaliada seja atribuída a um destino errado (e o destino correto não receba o valor correto) ou (3) uma atribuição não seja realizada (incluindo uma instrução *nop*).

Falhas de Decisão

Correspondem a falhas em expressões condicionais ou a omissões das mesmas. Uma falha de decisão pode ser injetada trocando uma instrução de *branch* por uma instrução *nop*, para

eliminar o condicional, ou alterando expressões de condição, para que as verificações retornem valores errados.

Falhas de Função

Correspondem a falhas de um bloco de comandos e não apenas comandos isolados. Nesse caso um conjunto de instruções provoca o efeito da falha. Esse tipo de falha é complicado de injetar e normalmente envolve a re-escrita de funções.

3.2.3 Visões de Programa

Um aspecto interessante em relação aos modelos de falhas, diz respeito às diferentes visões de programas utilizadas pelas ferramentas. As características das falhas a serem simuladas por software podem ser apresentadas aos usuários em diversos níveis de hierarquia, cada qual apresentando as falhas em um nível de abstração diferente.

Falhas podem ser injetadas alterando-se instruções assembler do programa alvo. Essa visão de programa é adequada para injetar falhas de processador. Por exemplo, para injetar uma falha permanente de registrador é preciso garantir que o conteúdo do registrador em questão esteja errado toda vez que o programa alvo utilizá-lo. O único meio de fazer isso é tendo acesso ao programa ao nível de instruções assembler.

Para injetar falhas de memória, a visão de programa normalmente utilizada é em função do espaço de memória ocupado pelo programa. Nesse caso o usuário pode desejar corromper a Seção de memória ocupada pelo código do programa, pela pilha e/ou por seus dados.

A maioria das falhas de software pode ser injetadas tanto em termos de instruções assembler como em termos de instruções de linguagens de alto nível. As falhas de funções consideram blocos funcionais como métodos, procedimentos ou funções.

Para injetar falhas de comunicação, o programa pode ser visto como entidades trocando mensagens de comunicação e o alvo para injeção de falhas, normalmente são as mensagens trocadas.

Como mencionado no Capítulo 1 as aplicações alvo consideradas neste trabalho são as aplicações orientadas a objetos. Uma aplicação orientada a objetos pode ser vista como um conjunto de objetos trocando mensagens de solicitação de serviços. Objetos encapsulam suas características privadas e oferecem um conjunto de serviços (métodos e atributos públicos). Nesse caso, o alvo para a injeção de falhas pode ser o objeto ou as mensagens trocadas entre eles. Quando o alvo é o objeto, ele pode ser modificado internamente, ou ainda ter seus serviços alterados. No Capítulo 5 (Subseção 5.2.2) é apresentado o modelo de falhas elaborado para a ferramenta de injeção por software proposta neste trabalho, o qual adota essa visão de programa.

3.3 Métodos de injeção de falhas por software

A base da injeção de falhas por software é a modificação, via controle de software, da imagem computacional do sistema a ser testado, fazendo com que o sistema se comporte como se uma falha de hardware ou software tivesse ocorrido. A imagem computacional de um processo corresponde ao seu conteúdo (dados e instruções) de memória e aos seus valores de registradores. Vários métodos de injeção por software foram desenvolvidos. Segundo Hsueh [HTI97], esses métodos podem ser categorizados de acordo com o momento em que a modificação da imagem computacional é introduzida: durante a compilação ou durante a execução.

3.3.1 Injeção durante a compilação

Para injetar falhas durante a compilação (injeção estática), a estratégia usada é alteração de algumas instruções do programa alvo antes que ele seja carregado e executado. Nesse caso, o código da aplicação alvo ou o código de algum serviço utilizado pela aplicação (serviços do sistema operacional, bibliotecas de funções, por exemplo) é alterado fazendo com que operações incorretas sejam realizadas. Esse método não requer software adicional durante a execução, visto que as falhas são injetadas diretamente no código fonte ou objeto da aplicação alvo. Essa técnica pode ser comparada a estratégia de geração de mutantes utilizada na técnica de Análise de Mutantes [VDB+97]. Considere, o seguinte fluxo de instruções:

$$\mathbf{I} = I_1, I_2, I_3, \ldots\ldots I_{N-1}, I_N, I_{N+1}$$

Se I_N é a instrução cuja execução deve ser corrompida, então ela deve ser substituída por uma seqüência **F** que emula um tipo particular de falha. A seqüência de instruções fica da seguinte forma:

$$\mathbf{I} = I_1, I_2, I_3, \ldots\ldots I_{N-1}, F_1 \ldots F_M, I_{N+1}$$

Como as falhas são embutidas no código essa estratégia é adequada para injetar falhas permanentes, pois toda vez que o fluxo de execução do programa alvo passar por um trecho de código corrompido a falha vai ser ativada.

3.3.2 Injeção durante a execução

Quando a injeção de falhas é realizada durante a execução da aplicação alvo (injeção dinâmica), dois elementos são necessários: um software extra para injetar as falhas e algum

mecanismo para ativar o injetor de falhas. Ou seja, a aplicação precisa ser instrumentada para os experimentos de injeção de falhas.

O injetor corresponde ao software que produz as alterações necessárias para emular os tipos de falhas considerados pela ferramenta. Um injetor pode ser implementado das seguintes formas:

- como um código extra anexado ao código da aplicação alvo. Ou seja, injetor e aplicação alvo constituem o mesmo processo;

- como um processo separado que executa concorrentemente com o processo da aplicação alvo. O processo do injetor de falhas dever ter acesso ao estado do processo alvo para que as devidas modificações possam ser realizadas;

- como tratadores de exceções/interrupções. Nesses casos, os mecanismos de injeção são *linkados* com o vetor de tratamento de interrupções. Tratadores de exceções/interrupções têm acesso ao estado do processo em execução e podem alterá-lo simulando a ocorrência de falhas.

Quanto aos mecanismos de ativação dos injetores as seguintes abordagens são possíveis:

- ***Time-out.*** O injetor é ativado após um tempo pré-determinado. Quando um evento de *time-out* é gerado uma interrupção invoca o injetor. Uma das desvantagens dessa abordagem é a dificuldade de repetir experimentos. Isso acontece porque o sistema de relógio normalmente usado não é exato e além disso o tempo de execução da aplicação alvo é incerto. Uma vantagem é a certeza de que as falhas serão sempre injetadas, uma vez que a injeção não está relacionada a nenhuma ação específica da aplicação alvo. [CMS95]

- **Exceções/Interrupções.** Nessa abordagem, uma interrupção de hardware ou uma exceção de software transfere o controle para o injetor. Diferente da ativação temporal, exceções/interrupções podem injetar falhas quando certos eventos ou condições relacionados à aplicação alvo ocorrem. Exceções de software podem ser inseridas no código fonte do programa alvo de forma que elas sejam executadas antes que o fluxo de execução passe por determinadas instruções. Quando a instrução de exceção é executada gera-se uma interrupção transferindo o controle para um tratador de interrupção. Nesse caso a ativação é denominada espacial, pois está relacionada a endereços de instruções do programa alvo. Essa é a forma de ativação mais usada, no entanto, a incerteza da ativação e do momento da ativação das falhas é uma limitação desse método. As interrupções de hardware são utilizadas para invocar injetores quando eventos detectados pelo hardware ocorrem. Por exemplo, quando o modo

traço é ativado o hardware detecta este evento e gera uma interrupção depois da execução de cada instrução.

3.4 Ferramentas Selecionadas

Nesta Seção são apresentadas algumas ferramentas que ilustram os métodos de injeção por software apresentados na Seção 3.3 e modelos de falhas apresentado na Seção 3.2. Dentre a grande gama de ferramentas existentes, foram selecionadas apenas algumas, de forma que uma boa visão das estratégias de injeção utilizadas seja apresentada.

3.4.1 FIAT

FIAT- *Fault Injection Based Automated Testing Environment* - [SVS+88] é um ambiente de teste baseado em injeção de falhas por software desenvolvido para validar a confiabilidade de sistemas distribuídos de tempo real. O ambiente oferece um conjunto de ferramentas para cobrir todo o processo de validação: geração das falhas, preparação da aplicação, execução da injeção de falhas e coleta de dados e análise dos resultados.

FIAT pode injetar falhas tanto na aplicação (código e dados) como no sistema operacional. Uma aplicação é vista como um conjunto de tarefas que se comunicam. As falhas injetadas na aplicação incluem falhas de memória e falhas de comunicação (corrupção, perda e atraso de mensagens). As falhas no sistema operacional incluem falhas de memória e falhas de registrador. Em relação ao padrão de repetição, FIAT injeta apenas falhas transientes.

A injeção de falhas é dinâmica e as características de injeção de falhas e monitorização são anexadas ao código fonte da aplicação e ao sistema operacional. As falhas são ativadas quando a execução passa por posições de memória previamente especificadas (ativação espacial).

3.4.2 FERRARI

FERRARI - *Fault and ERRor Automatic Real-time Injection* - [KKA92] usa exceções de software e interrupções de relógio para injetar falhas de processador, memória e barramentos durante a execução da aplicação alvo.

Três métodos de ativação da injeção são utilizados:

- corrupção de memória: uma falha é injetada na imagem de memória da tarefa antes que a execução do programa inicie. Os valores corrompidos podem permanecer

durante a execução do programa, e valores corrompidos podem ser usados muitas vezes;

- injeção espacial: o processo de injeção de falhas é ativado após N (valor definido pelo usuário) ocorrências de um endereço aleatório ou escolhido pelo usuário no segmento de código do processo usuário.

- injeção temporal: o processo de injeção de falhas é ativado por uma interrupção de relógio após o tempo estabelecido ser alcançado.

Quando uma exceção é gerada, o injetor, localizado na rotina de tratamento da exceção, injeta falhas corrompendo valores da memória e de registradores. As falhas podem ser transientes ou permanentes.

3.4.3 FTAPE

FTAPE - *Fault Tolerance And Performance Evaluator* - [TI92], adiciona *drivers* ao sistema operacional para ativar e injetar falhas. FTAPE é constituído de um injetor de falhas e um gerador de *workload*[4] e foi implementado para validar computadores tolerantes a falhas. O gerador de *workload* pode gerar um conjunto específico de tarefas que exercitam o processador, a memória ou o sistema de I/O.

FTAPE injeta falhas de hardware, as quais incluem: falhas de memória, falhas de processador (em registradores acessíveis a usuários) e falhas de I/O (no sistema de disco). Falhas no sistema de disco são injetadas através da execução de uma rotina no código do *driver*, a qual gera exceções que emulam erros de I/O (como por exemplo, erros de barramentos e erros de temporizador).

Os mecanismos de ativação utilizados pela FTAPE englobam os ativadores espaciais e temporais. Além desses dois mecanismos, a ferramenta pode injetar falhas de acordo com a quantidade de atividade executada pelo sistema. Por exemplo, se no momento da injeção, o sistema de disco for o componente mais utilizado pelo sistema, então, uma falha de I/O é injetada. Além disso, a taxa de injeção de falhas aumenta de acordo com a taxa de atividade do sistema.

3.4.4 SFI e DOCTOR

SFI - *Software Fault Injector* - [RS93] e seu sucessor, Doctor - integrateD sOftware fault injeCTion enviRoment [HRS93], foram desenvolvidos para validar um sistema distribuído de tempo real chamado HARTS.

[4] *Workload*: conjunto de tarefas que caracterizam o comportamento operacional do sistema alvo.

SFI e Doctor injetam falhas de processador, de memória e de comunicação. Para esses três tipos são considerados ainda os padrões de repetição: transiente, intermitente e permanente. As ferramentas usam quatro abordagens diferentes para injetar tipos de falhas diferentes:

- Usam injeção estática para injetar falhas permanentes de processador. Através da alteração do código durante a compilação são simuladas falhas permanentes de ULA, de fluxo de controle, de registradores e de relógio.

- As falhas transientes e intermitentes de processador são injetados de duas formas: (1) quando o programa executa uma determinada instrução, se uma falha deve ou não ser injetada é determinado por um código inserido na aplicação em tempo de compilação; (2) quando o tempo de injeção é alcançado uma falha de processador, selecionada aleatoriamente ou pelo usuário, é injetada.

- Para injetar falhas de comunicação SFI e Doctor usam uma camada de protocolo de comunicação adicional que pode ser inserida em qualquer ponto da pilha de protocolo. A camada adicional de falhas opera interceptando operações entre o protocolo em teste e os protocolos das camadas inferiores. Se a operação deve sofrer injeção a camada injeta a falha adequada, caso contrário a operação é realizada sem modificação.
- Quando experimentos requerem falhas de memória, um processo de prioridade alta chamado *Memory Fault Injection Process* é executado para realizar a injeção de falhas. Quando o tempo preestabelecido é atingido, esse processo injeta falhas em endereços selecionados lendo, alterando e escrevendo novamente o conteúdo de localizações específicas. Como HARTS não tem proteção de memória qualquer área de memória pode ser corrompida.

3.4.5 ProFI

A ferramenta ProFI - *Processor Fault Injection* - [LE93] foi projetada inicialmente para validar um sistema tolerante a falhas denominado VDS *(Virtual Duplex System).*

A abordagem de injeção de falhas da ProFI usa a facilidade de modo traço dos processadores. Quando o modo traço está ativo, uma rotina é executada após cada instrução de máquina do programa alvo. Essa rotina executa no modo supervisor e tem acesso a toda a memória e ao conjunto de registradores, podendo assim simular as conseqüências de falhas de processador. Esse método facilita a injeção de falhas permanentes, pois a falha pode ser injetada após cada instrução. ProFI também permite a injeção de falhas transientes e intermitentes adicionando à aplicação em teste um trecho de código que implementa condições de ativação.

3.4.6 FINE e DEFINE

FINE - *Fault Injection and Monitoring Environment* - [KIT93], é um ambiente para experimentos de injeção de falhas com o objetivo de emular tanto falhas de hardware como falhas de software para estudar a propagação de falhas no núcleo do UNIX.

Como as aplicações do usuário não têm o privilégio de modificar o núcleo do sistema operacional, FINE tem um módulo implementado no núcleo UNIX, seu sistema alvo. A interface para a camada da aplicação é provida através da chamada à função do sistema *ptrace*. A parte servidor do injetor implementada no núcleo recebe os parâmetros de falhas do injetor cliente implementado em uma aplicação do usuário e injeta as falhas no núcleo. Um monitor, também embutido no núcleo, registra o fluxo de execução do núcleo, os valores de determinadas variáveis e dados.

DEFINE - *Distributed Fault Injection and Monitoring Environment* [KI94], é uma versão distribuída do ambiente FINE. Nessa versão o ambiente provê duas técnicas de ativação das falhas: utilizando interrupções do relógio e utilizando exceções (*traps*) de software. As interrupções de hardware garantem que falhas de barramento são sempre ativadas. As exceções indicam ao monitor se uma falha foi injetada e quando.

3.4.7 Um injetor híbrido

A injeção física não monitora a atividade de software do sistema, e assim, não pode injetar falhas quando o sistema atinge um estado crítico, como por exemplo, injetar falhas quando o sistema atinge uma alta taxa de utilização do processador. A injeção por software não permite a injeção de falhas em sinais e barramentos externos. Uma ferramenta híbrida [KKA95] foi construída com o objetivo de cobrir tais deficiências.

Nessa ferramenta híbrida um módulo controlador coordena as atividades e o fluxo de informações entre os seguintes componentes: um injetor baseado em software, um monitor de software, um monitor de hardware e um injetor baseado em hardware.

O injetor de software é uma extensão do injetor da ferramenta FERRARI. O monitor de software mede algumas atividades do sistema alvo tais como, o número de processos, o tráfego na rede, a freqüência de mudanças de contexto e a percentagem de utilização de I/O. Em adição ele tem rotinas que aumentam a atividade do sistema a um nível selecionado e uma vez atingido esse nível, o controlador ativa o injetor baseado em hardware.

O injetor de hardware aplica sinais lógicos a linhas e barramentos selecionados. O circuito lógico deste módulo é implementado sobre um LCA[5] (*Logic Cell Array*). O monitor de hardware tem a função de sincronizar o processo de injeção de falhas com determinadas

[5] Também conhecido como Vetor de Portas Programável (*Programmable Gate Array - FPGA*).

atividades dos barramentos do sistema alvo. Esse monitor é implementado sobre o mesmo LCA usado para implementar o injetor baseado em hardware.

3.4.8 ORCHESTRA

ORCHESTRA [DJM97] é uma ferramenta de injeção de falhas por software para testar implementações de protocolos distribuídos. A ferramenta é baseada em um framework para injeção de falhas em protocolos de comunicação denominado *Script-driven Probing and Fault Injection* [DJM97]. Nesse framework um protocolo distribuído é visto como uma coleção de camadas participantes que comunicam-se por troca mensagens. Cada camada oferece um serviço de comunicação abstrato para as camadas superiores.

Na abordagem da ORCHESTRA, uma camada de injeção de falhas é inserida abaixo da camada alvo. Toda mensagem enviada por ou para a camada alvo, é interceptada e examinada pela camada de injeção. A decisão de injetar determinados tipos de falhas é tomada como base em alguns atributos das mensagens capturadas (como o tipo e o conteúdo), no histórico de mensagens, ou em certos dados que a camada de injeção coleta (como contadores). Assim, ORCHESTRA consegue minimizar a intrusividade em relação ao protocolo alvo, uma vez que a camada alvo não é alterada.

A arquitetura da ORCHESTRA apresenta uma separação clara entre os mecanismos de injeção, o protocolo alvo e o código dependente da aplicação. Os mecanismos de injeção são independentes das mensagens sobre as quais eles operam. Para ORCHESTRA, uma mensagem é vista como um objeto abstrato e o modo como ela deve ser interpretada deve ser informado pelo usuário através de *scripts*. É através da execução de *scripts* que a ferramenta "orquestra" a computação do sistema para um estado particular e injeta falhas. Assim, uma outra característica importante da ferramenta é a portabilidade.

3.4.9 Xception

Xception - *Software Fault Injection and Monitoring in Processor Functional Units* - [CMS95] usa características especiais de depuração presente nos processadores modernos para injetar e monitorizar a ativação de falhas transientes de processador. Através das facilidades do mecanismo de depuração, o programador pode especificar pontos de injeção de falhas conforme a ocorrência de alguns eventos, tais como: a busca ou o armazenamento de dados, a busca de instruções ou mesmo a execução de alguns tipos de instruções (instruções de ponto-flutuante, por exemplo).

O injetor da Xception é implementado como rotinas de tratamento de interrupções, as quais são ativadas quando determinados eventos previamente definidos pelo usuário ocorrem. Diferente de outras ferramentas de injeção de falhas por software, Xception utiliza

interrupções de hardware ao invés de instruções de exceções. A Tabela 3.3 lista os tipos de interrupções do MPC601 utilizados pela Xception.

Tipo de Exceção	Condições de Ativação
Acesso a dados	Ocorre quando o endereço usado em uma instrução de *load/store* é igual ao endereço no DABR (*Data Acess Breakpoint Register*).
Modo de Execução	Ocorre quando o endereço efetivo de uma instrução que está sendo decodificada é igual ao endereço contido no IABR (*Instruction Address Breakpoint Register*).
Trace	Quando o MPC601 executa no modo *trace*, uma interrupção é ativada após cada instrução, sem causar um exceção ou mudança de contexto.
Ponto flutuante indisponível	A unidade de ponto flutuante pode ser desabilitada modificando um bit no MSR (*Machine Status Register*). Uma exceção de ponto flutuante ocorre quando é feita uma tentativa de execução de uma instrução de ponto flutuante quando a unidade está desabilitada.
Decrementador	Ocorre depois que o conteúdo do DEC (registrador que decrementa seu conteúdo em uma freqüência determinada) chega ao valor zero.

Tabela 3.3 Exceções utilizada pela Xception para ativar a injeção de falhas.

Interrupções de acesso a dados e de modo de execução são usadas para ativação espacial das falhas. O modo traço é usado durante o processo de injeção das falhas para executar passo-a-passo um pequeno conjunto de instruções, mas não é a principal característica da ferramenta. Interrupções de unidade de ponto flutuante são usadas para ativar falhas desta unidade. Interrupções de decrementador são usada para ativação temporal de falhas.

Xception programa diretamente o hardware especial do processador alvo para ativar as rotinas de injeção de falhas e monitorar a ativação de erros latentes. A injeção de falhas engloba alguns passos: (1) programar as condições de ativação das falhas na lógica de depuração do processador; (2) corromper registradores específicos ou o conteúdo de memória (tratadores de exceções fazem a corrupção) quando a exceção que injeta a falha é sinalizada, de acordo com o tipo de falha e da unidade funcional a ser afetada; (3) restaurar os valores originais do registrador ou célula de memória.

Xception roda inicialmente em sistemas baseados na família PowerPC. Segundo o estudo feito pelos autores, as características especiais para sua implementação estão ou estarão presentes em outras famílias de processadores.

3.4.10 FIESTA

FIESTA - *Fault Injection to Embedded System Target Aplications* - [KJA98] é uma ferramenta de injeção de falhas por software que foi desenvolvida para validar a segurança de funcionamento de aplicações de tempo real que rodam sobre *Vxworks,* um sistema operacional de tempo real comercial que possui um grande mercado na área de sistemas embutidos.

A ferramenta foi desenvolvida segundo uma metodologia de desenvolvimento de injetores de falhas por software para sistemas embutidos, proposta pelos autores. De acordo com tal metodologia, um injetor para sistemas embutidos pode ser facilmente desenvolvido usando a ferramenta de depuração existente no ambiente de desenvolvimento vendido com esses sistemas. O injetor é dividido em dois componentes complementares: o depurador, que injeta as falhas, e o controlador de injeção. A interface da ferramenta é provida pelo depurador.

FIESTA usa as operações oferecidas pela ferramenta de depuração *gdbm68k* para injetar falhas transientes de memória, processador e barramentos sobre a aplicação alvo. O procedimento de injeção é o seguinte: o controlador de injeção envia uma série de comandos de depuração para o depurador e esse, por sua vez, executa esses comandos sobre a aplicação injetando as falhas. Exemplos de comando de depuração são, entre outros:

- marcar e desmarcar pontos de parada;
- examinar e modificar registradores e memória;
- executar a aplicação passo-a-passo;
- executar a aplicação normalmente;
- capturar exceções e reportá-las ao usuário.

A principal característica dessa ferramenta é a intrusividade mínima em relação a estrutura da aplicação alvo, uma vez que não é necessário alterar a sua implementação.

3.4.11 IPA

IPA- *Interface Propagation Analisys* - [Voa98] é uma ferramenta de injeção por software baseada na corrupção de estados propagados através das interfaces entre os componentes da aplicação alvo. A injeção é realizada durante a execução através de uma rotina de software que troca um estado original por um estado corrompido. A instrumentação para injeção é introduzida no código fonte da aplicação alvo antes da compilação. Quando a aplicação modificada é executada, um processo de controle coleta dados sobre o comportamento da

aplicação em relação ao componente corrompido. Considerando a função `char* strcpy(char *s1, char *s2)`[6] a injeção pode ser feita sobre o resultado da chamada:

```
strcpy(old, new);
old = PERTURB(old);
```

Depois da chamada original o resultado em `old` é trocado por uma *string* diferente através da rotina `PERTURB`.

IPA foi implementada para determinar se uma aplicação pode tolerar falhas em algum de seus componentes.

3.5 Limitações da Injeção de Falhas por Software

Apesar do grande número de ferramentas de injeção de falhas por software e do considerável avanço alcançado nesse campo, as ferramentas existentes ainda possuem problemas e limitações. Para injetar falhas e monitorizar seus efeitos alguma forma de instrumentação para os testes deve ser introduzida na aplicação alvo. Como apresentado na Seção 3.3 essa instrumentação varia de acordo com o método de injeção de falhas adotado pela ferramenta.

Quando a injeção é feita durante a compilação, não é necessário adicionar à aplicação um código extra responsável pela injeção e monitorização, pois as falhas são injetadas diretamente no seu código fonte ou objeto. Assim, esse método não causa nenhuma perturbação no programa alvo durante a sua execução. Esse tipo de injeção é adequado para injetar falhas de software, pois essas são permanentes. No entanto, a injeção estática apresenta limitações para injeção de falhas de hardware (que em geral são temporárias), visto que permite pouca flexibilidade no que diz respeito à temporização das falhas. Como as falhas ficam embutidas no código, é difícil ou mesmo impossível injetar falhas transientes e intermitentes e falhas cuja ativação depende do número de execuções de um determinado endereço ou de um valor de relógio.

A injeção por software durante execução oferece maior flexibilidade, mas, o preço pago por ela é a perturbação no código do programa alvo. Essa perturbação compromete tanto o comportamento da aplicação, pois um código extra para controlar a injeção de falhas é executado, como a estrutura da aplicação, pois aspectos funcionais e não-funcionais (injeção de falhas e monitorização) da aplicação não ficam separados.

Por exemplo, a inserção de instruções de exceção para ativar a injeção de falhas comprometem tanto a estrutura como o comportamento da aplicação alvo. Métodos que utilizam a execução no modo traço causam um grande impacto sobre a velocidade da execução do programa alvo. Segundo Carreira [CMS95], pesquisas têm enfatizado o

[6] Essa função copia os caracteres da *string* s1 para s2.

impacto do software de injeção de falhas no desempenho dos mecanismos de tratamento de falhas, o que significa que as ferramentas de injeção de falhas podem interferir nos resultados das aplicações em teste.

Xception [CMS95] e FIESTA [KAJ98] não modificam a aplicação alvo, contudo, utilizam características especiais do ambiente do sistema alvo. Xception injeta falhas programando diretamente o hardware de depuração interno ao processador. Seu mecanismo de injeção de falhas é baseado nas características internas dos processadores atuais, as quais, normalmente, não estão disponíveis para usuários comuns. Por sua vez, FIESTA faz uso das características de uma ferramenta de depuração do ambiente de desenvolvimento de seu sistema alvo.

Do exposto acima, pode-se concluir que: a menos que o ambiente de teste ofereça características que facilitem a injeção e monitorização de falhas ou uma abordagem híbrida seja utilizada, alguma forma de instrumentação deve ser introduzida ao programa alvo, a fim de obter maior flexibilidade na injeção de falhas. Essa instrumentação é intrusiva, ou seja, interfere na execução da aplicação alvo. No entanto, um dos objetivos de uma abordagem de instrumentação de software é ser o menos intrusiva possível. Sabe-se que é impossível eliminar o processamento extra decorrente do código para controlar a injeção de falhas, mas é possível diminuir o impacto dos mecanismos usados para ativar esse código. Além disso, a estrutura do programa pode ser preservada através da separação das características funcionais e não-funcionais.

Para alcançar esses objetivo, neste trabalho é proposto o uso da programação orientada a objetos reflexiva. Reflexão foi escolhida por ser uma maneira simples de separar a funcionalidade da instrumentação da funcionalidade da aplicação alvo e por permitir um modo simples e quase transparente de ativação do código de injeção de falhas (ver próximo capítulo).

3.6 Resumo

Este capítulo apresentou uma visão geral da técnica de injeção de falhas por software, enfocando os seguintes tópicos: modelos de falhas, métodos de injeção, uma amostra de ferramentas existentes e as principais limitações da técnica.

Injeção de falhas por software consiste na injeção de padrões de erros em um software em execução. [KKA95]

Esses padrões representam erros que podem ser gerados pela ocorrência de falhas de hardware ou software. A maioria dos trabalhos consideram falhas de hardware, as quais são classificadas em falhas de: memória, processador, barramentos, I/O e comunicação. Falhas de software representam falhas de projeto e/ou implementação. O modelo apresentado toma

como base a causa das falhas e usa a seguinte classificação: falhas de inicialização, falhas de atribuição, falhas de decisão e falhas de funções.

Um outro ponto interessante do modelo de falhas, diz respeito as visões de programa diferentes utilizadas para a injeção de tipos de falhas diferentes: instruções de baixo nível, instruções de alto nível, blocos de instruções, espaço de memória ocupado pelo programa, mensagens trocadas em um sistema distribuído, objetos em aplicações orientadas a objetos.

Os métodos de injeção de falhas por software podem ser classificados de acordo com a fase na qual as falhas são injetadas: em tempo de compilação ou execução. Injeção em tempo de compilação apresenta baixa intrusividade, uma vez que nenhuma instrumentação é introduzida no código fonte do programa alvo para propósitos de injeção e monitorização. Essa abordagem, contudo, apresenta baixa controlabilidade e observabilidade. Injeção durante a execução apresenta maior controlabilidade e observabilidade, contudo, a intrusividade aumenta a menos que o injetor de falhas e o monitor usem algum dispositivo de hardware ou características especiais do ambiente do sistema alvo.

Apesar das limitações, injeção por software apresenta uma série de vantagens para validação de MTF's em relação a outras formas de injeção, conforme apresentado no capítulo anterior. Devido a essas vantagens o seu uso tem ampliado-se, especialmente por ser adequada para validar o impacto de defeitos de componentes de software, sendo útil tanto para avaliar "*software liability*" [VMK+97], quanto para auxiliar desenvolvedores na escolha de componentes de prateleira ou COTS (*Commercial off-the-shelf*) [Voa98], entre outros.

A proposta deste trabalho é utilizar reflexão computacional para minimizar a perturbação no programa alvo no teste de aplicações orientadas a objetos através de injeção de falhas por software durante a execução. O próximo capítulo justifica o uso de reflexão como técnica escolhida para instrumentação da aplicação alvo.

Capítulo 4

Reflexão Computacional

Este capítulo tem por objetivo apresentar as características relacionadas à reflexão computacional e mostrar que tais características são bastante adequadas para incorporar, de forma transparente, novas funcionalidades a uma aplicação. No contexto deste trabalho, reflexão é usada para incorporar as características de injeção de falhas e monitorização de seus efeitos ao sistema a ser validado.

4.1 Introdução

O conceito de reflexão aparece em diversas ciências como, física, filosofia, lingüistica, lógica e computação, geralmente com significados e motivações diferentes. Três dos vários significados do termo reflexão são adequados para definir o significado de reflexão no contexto computacional[7]:

1. No sentido de cognição humana, reflexão significa: *volta da consciência, do espírito, sobre si mesmo, para examinar o seu próprio conteúdo por meio do entendimento, da razão.* Este significado traduz o objetivo de reflexão computacional, que é a capacidade de um sistema de software atuar sobre si mesmo, realizando deduções e computações sobre dados internos do próprio sistema e não sobre dados do domínio externo para o qual ele produz resultados .[Lis97]

2. Como propriedade física, reflexão significa: *a modificação da direção da propagação de uma onda que incide sobre uma interface que separa dois meios diferentes, e retorna para o meio original.* Este significado está mais relacionado com a forma de implementação de reflexão computacional no modelo de objetos: ao ser enviada uma mensagem a um objeto ela é desviada para um outro objeto associado a ele, denominado metaobjeto, que passa a realizar computações sobre dados relacionados ao primeiro objeto ou à classe do primeiro objeto. [Lis97]

[7] Fonte: Novo dicionário da Língua Portuguesa, Aurélio B. H. Ferreira, Ed. Nova Fronteira, 1985.

3. Como ato ou efeito de refletir(-se), reflexão significa *reprodução da imagem de; repercussão*. Este significado explica o relacionamento entre meta-objeto e objeto, no qual uma modificação feita em um reflete (provoca efeito) no outro.

Em resumo reflexão computacional pode ser definida como a capacidade de um sistema computacional de desviar do seu processo normal de execução, fazer deduções ou computações em um outro nível de processamento e retornar ao nível de execução traduzindo o impacto das decisões, para então retomar o processo de execução. [Lis97]

Atualmente, o conceito de arquitetura reflexiva possui um significado mais diretamente relacionado com o contexto de orientação a objetos, embora seja encontrado também no modelo de programação funcional e programação lógica. O conceito de reflexão estende o modelo de abstração do paradigma de orientação a objetos: além do modelo de abstração baseado em classes, reflexão introduz um modelo de abstração baseado em domínios. Segundo esse modelo, uma aplicação pode ser separada em domínios de programação: um dos domínios compreende as funcionalidades do propósito da aplicação e o outro está relacionado à infra-estrutura de apoio da aplicação para a realização do seu propósito.

A organização do capítulo é a seguinte: a Seção 4.2 apresenta a nova arquitetura de software introduzida pelo conceito de reflexão computacional: a arquitetura reflexiva; a Seção 4.3 apresenta algumas conceitos de orientação a objetos; a Seção 4.4 descreve o casamento entre reflexão computacional e orientação a objetos; a Seção 4.5 lista o emprego de reflexão computacional em diversas áreas; a Seção 4.6 apresenta as diversas vantagens do uso de reflexão computacional para injeção de falhas por software, e por fim a Seção 4.7 resume e conclui o conteúdo apresentado neste capítulo.

4.2 Arquitetura Reflexiva

De modo geral, os requisitos de uma aplicação podem ser categorizados em dois grupos distintos, os *requisitos funcionais* que estão associados com o propósito da aplicação, e os *requisitos não-funcionais* (ou *administrativos*), que estão relacionados à infra-estrutura de apoio do sistema para a realização de tal propósito (por exemplo, tolerância a falhas, persistência e distribuição) ou ainda relacionados a facilidades de atuação sobre o comportamento da aplicação (por exemplo, monitorização, depuração e injeção de falhas).

Espera-se que uma aplicação bem modular apresente uma separação bem nítida entre esses requisitos. O uso de técnicas de orientação a objetos como encapsulamento e interfaces melhoram a modularidade, mas não são suficientes para garantir tal separação. [Cor97]

Essa separação de funcionalidades é facilmente obtida com o uso de reflexão computacional, uma vez que essa técnica baseia-se justamente na subdivisão de um sistema em níveis distintos, propondo uma nova arquitetura de software, a arquitetura reflexiva. A

arquitetura reflexiva é composta por dois níveis: o *nível base* que destina-se a implementar as estruturas de dados e ações sobre o domínio da aplicação, ou seja ideal para implementar requisitos *funcionais*, e o *meta-nível* relacionado a solução de problemas e armazenamento de informações sobre o nível base, portanto adequado para a implementação de requisitos *não-funcionais*.

A Figura 4.1 apresenta a arquitetura reflexiva, mostrando que as ações no meta-nível são executadas sobre dados que representam informações sobre o programa de nível base, enquanto que o programa de nível base executa ações sobre dados do domínio externo, com a finalidade de atender aos usuários de seus serviços. As computações reflexivas podem interferir nas computações de nível base. [Lis97]

A separação de domínios proposta pela arquitetura reflexiva permite a construção de arquiteturas de software reusáveis, nas quais podem ser reutilizados tanto os componentes do nível base como os componentes do meta-nível. Além de incentivar a reutilização, a separação de domínios permite ao programador da aplicação concentrar-se na solução do problema de programação específico do domínio da aplicação deixando para especialistas a solução dos problemas não-funcionais. Outro aspecto atraente da arquitetura reflexiva é que ela possibilita a incorporação de novas propriedades a aplicações de forma não intrusiva, ou seja, novas propriedades podem ser acrescidas na forma de um meta-nível sem alterar a estrutura da aplicação no nível base.

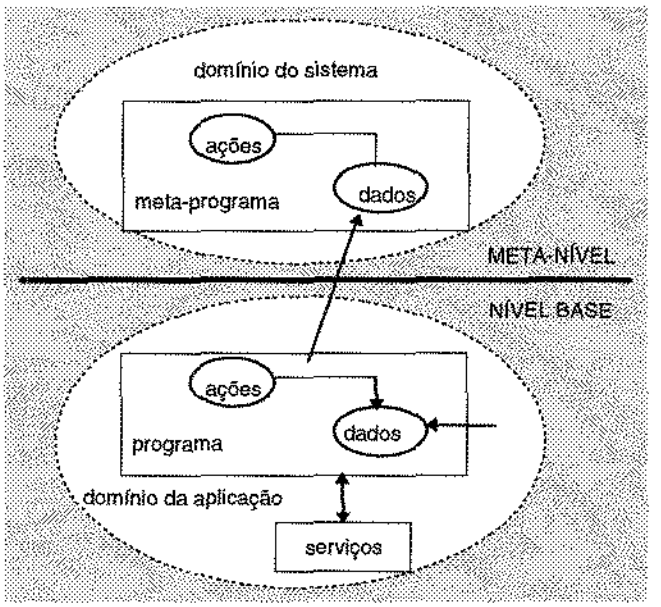


Figura 4.1 Arquitetura reflexiva[8]

[8] Figura adaptada de [Lis97].

É no modelo de orientação a objetos, com sua natural flexibilidade e facilidade de programação incremental, que a arquitetura reflexiva tem mostrado sua eficácia e elegância na obtenção de novas soluções para problemas de programação. As seções 4.3 e 4.4 tratam, respectivamente, do conceitos de orientação a objetos e do conceito de reflexão na programação orientada a objetos.

4.3 Conceitos de Orientação a Objetos

Esta seção apresenta algumas definições básicas do paradigma de orientação a objetos, o qual constitui uma coleção de conceitos cujo objetivo principal é prover componentes de software reutilizáveis. A reutilização é obtida através da noção de objetos, classes, encapsulamento, herança, ligação dinâmica e polimorfismo, que permite a estruturação dos componentes do sistema de forma modular e hierárquica.

As definições apresentadas a seguir foram baseadas em [BR98].

4.3.1 Tipos Abstratos de Dados

Um tipo é a descrição de uma interface que especifica o comportamento comum de componentes similares. Abstração de dados consiste na ocultação de informações como forma de tratar a complexidade. Um tipo abstrato de dados proporciona uma abstração sobre um tipo em termos de uma interface bem definida, a qual separa sua especificação da sua implementação. A parte de especificação define a sintaxe e a semântica da interface de um tipo abstrato de dado. A implementação de um tipo abstrato de dado descreve sua representação em termos de estruturas de dados primitivas e os algoritmos associados a cada operação.

4.3.2 Encapsulamento

O encapsulamento é uma forma de minimizar interdependências entre componentes do sistema e consiste na separação dos aspectos externos de um componente, acessíveis por outros componentes, dos detalhes de implementação. Desse modo, um componente pode ter sua implementação alterada sem que isso afete outros componentes que interagem com ele.

4.3.3 Objetos

Objetos são entidades que encapsulam dados e operações. Dados, (dados membro, atributos ou variáveis de instância) representam o estado do objeto. Operações, (métodos ou funções membro) representam o seu comportamento. A interface de um objeto constitui o conjunto

de operações que podem ser requisitados por outros objetos. A comunicação entre objetos é feita através de mensagens que identificam operações em objetos. Cada objeto possui ainda uma identidade que pode ser usada para identificá-lo unicamente ou distinguí-lo de objetos similares.

4.3.4 Classe

Uma classe é a descrição de um molde que especifica a estrutura e o comportamento de um conjunto de objetos similares, ou seja, ela define os atributos, os tipos e as operações que compõem os objetos. Um objeto é uma instância de apenas uma classe.

4.3.5 Herança

Herança é um mecanismo para derivar novas classes a partir de classes bases através de um processo de refinamento. Ou seja, é um tipo de relacionamento hierárquico entre classes através do qual é possível compartilhar representações de dados e operações. A classe base é denominada superclasse e as classes derivadas ou descendentes são denominadas subclasses. O maior benefício da herança é a reutilização de software, pois cada subclasse herda todas as propriedades de sua superclasse e acrescenta características próprias e exclusivas. Quando uma subclasse tem mais de uma superclasse surge o conceito de herança múltipla.

4.3.6 Classes Concretas e Abstratas

Uma classe concreta é uma classe que pode ser instanciada. Uma classe abstrata é aquela que não possui instâncias diretas, mas cujas classes descendentes têm instâncias diretas. Uma classe abstrata pode definir o protocolo para uma operação sem fornecer sua implementação, esta operação é chamada de operação abstrata e sua implementação deve ser definida nas subclasses concretas.

4.3.7 Polimorfismo

A palavra polimorfismo significa "muitas formas" ou "tendo muitas formas" No contexto de orientação a objetos, polimorfismo significa que diferentes tipos de objetos podem responder a uma mensagem de um modo diferente. Ou seja, é possível definir um mesmo método em classes diferentes de modo que cada "versão" desse método seja adaptada para cada tipo de objeto. A forma como cada tipo de objeto responderá a mensagem de solicitação de tal método dependerá do tipo do objeto. Quando um método é implementado diferentemente em classes distintas ele é denominado polimórfico.

4.3.8 Ligação dinâmica

Ligação ou acoplamento é o termo usado para denominar a associação entre entidades do programa e seus atributos (seu nome, seu tipo, sua área de armazenamento). Uma ligação é dita estática se ocorre antes da execução, e permanece inalterada durante a execução do programa. Uma ligação é dinâmica se ocorre durante a execução e muda durante o curso de execução do programa. Em orientação a objetos a ligação dinâmica implica na determinação em tempo de execução de qual operação será executada dependendo do nome da operação e do tipo do objeto. Ou seja, a associação entre o nome da operação e a sua implementação é feita apenas em tempo de execução.

4.4 Reflexão Computacional e Orientação a Objetos

Em orientação a objetos reflexão refere-se a obtenção de dados sobre propriedades de classes ou de objetos de uma aplicação, e a possibilidade de modificar esses dados, modificando-se assim as propriedades de classes ou objetos da aplicação. Ou seja, reflexão permite que atributos como invocação de mensagens, interfaces e herança possam também ser alvo de computação do próprio sistema.

Em linguagens orientadas a objetos, a reflexão computacional é realizada através de dois modelos que se distinguem por sua abrangência: meta-classes ou metaobjetos. Em ambos os modelos, o objetivo é fornecer informações sobre as propriedades do nível base. Essas informações são fornecidas por um protocolo de metaobjetos, que define quais informações serão acessíveis e como serão. Existem diferentes protocolos de metaobjetos. O próximos itens descrevem com mais detalhes os pontos acima listados. Uma maior ênfase é dada ao protocolo de metaobjetos de OpenC++1.2, que foi a linguagem utilizada neste trabalho.

4.4.1 Modelos de Reflexão

Segundo Lisboa [Lis97], no contexto de objetos duas abordagens de reflexão são utilizadas: a reflexão por meta-classes e a reflexão por metaobjetos. Na abordagem meta-classe, o meta-nível é formado por meta-classes, as quais contêm informações sobre a estrutura estática de componentes do nível base, como por exemplo a descrição de variáveis e de métodos que irão compor todas as suas instâncias. Esta estrutura pode ser alterada e conseqüentemente todas as instâncias também serão alteradas. No caso da abordagem metaobjeto, o meta-nível é composto por metaobjetos, que contêm informações sobre os aspectos comportamentais dos objetos do nível base, como por exemplo, como um determinado objeto trata uma mensagem recebida. Os tópicos a seguir descrevem essas duas abordagens.

O modelo meta-classe

Uma meta-classe ^X é uma classe que descreve a estrutura de uma classe X e cujas instâncias também são classes. Ou seja, uma meta-classe é a classe de uma classe. Meta-classes se distinguem de classes tradicionais, porque seu domínio é a descrição de classes, seus dados referem-se a nomes de métodos, instâncias e heranças, e seus métodos podem acessar e modificar esses dados.

Meta-classes, quando acessíveis aos usuários, permitem reflexão estrutural. Reflexão estrutural de uma classe X pode ser definida como toda atividade realizada em uma meta-classe ^X, com o objetivo de obter informações e realizar transformações sobre a estrutura estática da classe X. Informações e transformações típicas de reflexão estrutural são: (a) obter informações sobre a classe X: sua classe ascendente, suas classes descendentes, suas instâncias, seus métodos e interfaces; (b) alterar a classe X: modificar seus atributos e seus métodos; e ainda, (c) atuar sobre classes: criar novas classes, eliminar classes existentes e renomear classes. [Lis97]

O modelo metaobjeto

O modelo metaobjeto foi introduzido por Maes [Mae87] para implementar reflexão em sistemas orientados a objetos. Um objeto do nível base x pode ser associado a um objeto do meta-nível ^x o qual representa a meta-informação de x. Objetos do meta-nível são denominados metaobjetos. Existe entre objeto e seu metaobjeto uma relação de causa-efeito denominada conexão causal, de tal forma que qualquer modificação feita em um deles provoca efeitos no outro. Segundo Lisboa [Lis97], as principais características do modelo de metaobjetos são as seguintes:

- *Individualidade:* a cada objeto do nível base pode ser associado um ou mais meta-objetos. O objeto associado a um metaobjeto é denominado objeto reflexivo. Dessa forma, algumas instâncias de uma classe podem ser escolhidas para realização de reflexão computacional enquanto outras instâncias da mesma classe podem ser não-reflexivas.

- *Separação de classes:* a classe do metaobjeto é distinta da classe do objeto do nível base ao qual ele será associado. Assim, objetos de uma mesma classe podem ser associados a metaobjetos de classes diferentes e por outro lado, metaobjetos de uma mesma classe podem ser associados a objetos do nível base de diferentes classes.

- *Reflexão comportamental:* um metaobjeto pode controlar como o objeto a ele associado (objeto referente) reage diante de uma mensagem. Assim, o comportamento de um objeto reflexivo pode ser alterado pelo seu metaobjeto. Como o comportamento de um objeto é ditado pelos seus métodos e pelo estado de seus

atributos, é preciso dividir os métodos e atributos de um objeto em dois grupos, os reflexivos e os não-reflexivos, onde somente os reflexivos serão controlados pelo metaobjeto.

- *Associação dinâmica:* a associação entre um objeto e seu metaobjeto é feita em tempo de execução. Quando um objeto reflexivo é instanciado, o seu metaobjeto também é instanciado e eles são associados de forma causal.

- *Definição circular:* um metaobjeto é um objeto, uma vez que ele é instância de uma classe. Portanto um metaobjeto pode ser tratado com um objeto e também pode ser associado a outro metaobjeto. Esta característica permite a estruturação de diversos meta-níveis, caracterizando uma torre de reflexão: cada meta-objeto pode ter um ou mais metaobjetos a ele associado. Conceitualmente , a torre de reflexão é infinita, na qual cada meta-nível define o comportamento do nível base imediatamente inferior.

4.4.2 Protocolos de Metaobjetos (PMO)

Algumas linguagens possuem por concepção características reflexivas. Um exemplo é a linguagem Smalltalk, na qual todas as entidades são objetos, inclusive as classes (descritoras de objetos). Sendo uma classe considerada um objeto, é possível realizar operações sobre a classe e dessa forma alterar a estrutura dos objetos da classe. No entanto, existem linguagens reflexivas que não tinham reflexão como uma das suas características de projeto, sendo esta acrescida posteriormente através de uma interface com o compilador da linguagem. Esse procedimento tem se tornado bastante comum nas linguagens orientadas a objetos através da construção de protocolos de metaobjetos. PMO é uma interface entre objetos e metaobjetos. Existem abordagens distintas para a definição desses protocolos, mas em geral eles estabelecem três pontos:

- **Quais e como informações do nível base estarão disponíveis no meta-nível.**

Este ponto diz respeito aos dados que podem ser manipulados no meta-nível, ou seja refere-se as informações sobre classes e objetos que devem ser transformadas em dados para o meta-nível. Esta operação é denominada *reificação* ou *materialização.*

Durante o processo de tradução (compilação ou tradução) de um programa, o processador da linguagem possui informações tais como: o nome de todas as classes e sua hierarquia de herança, nomes e tipos de todos os atributos e métodos em todas as classes, e os nomes do possíveis objetos a serem instanciados e as mensagens a serem instanciadas. Estas informações são utilizadas durante a tradução, mas normalmente são descartadas ou escondidas após a geração do código objeto. O objetivo da reificação é relevar e disponibilizar ao programador da aplicação essas informações. Dessa forma, o meta-nível pode ser visto como uma interface de alto nível entre o programador e o programa de nível

base, habilitando o programador a fazer adaptações do programa para atender necessidades particulares. Como um objeto encapsula alguns de seus dados e métodos, sendo visíveis apenas pela sua interface externa, para obter dados sobre sua implementação é necessário quebrar a barreira do encapsulamento feito pelo compilador.

Os PMO's diferem em relação ao tipo de manipulação permitido sobre meta-informações. Alguns protocolos permitem que estas informações sejam alteradas e dessa forma permitem modificações dinâmicas no nível base, enquanto outros permitem apenas que o meta-nível use estas informações para observar o nível base.

- **Como e quando o sistema passa para o meta-nível.**

Este ponto refere-se a quem e como, no nível base, inicia o processo de reflexão no meta-nível. Segundo Maes [Mae87], o processo de reflexão pode ser iniciado por um objeto de nível base ou pelo sistema. Quando iniciado pelo objeto, este possui código mencionando seu metaobjeto. Quando iniciado pelo sistema, o metaobjeto é ativado pelo sistema quando ocorre algum evento envolvendo o objeto reflexivo a ele associado, como por exemplo uma mudança no estado de uma variável reflexiva ou o recebimento de uma mensagem dirigida a um método reflexivo.

No caso de interceptação de mensagens, todas as mensagens enviadas a algum método reflexivo de um objeto reflexivo são delegadas ao seu meta-objeto, o qual passa a ser o responsável pelo tratamento das mensagens. Um aspecto importante é que, graças a reificação de informações como o nome do método destinatário da mensagem e os argumentos que o cliente forneceu na mensagem, o metaobjeto pode reenviar a mensagem ao método da aplicação para a execução do método original, além de realizar outras computações no meta-nível. A Figura 4.2 mostra o fluxo de uma mensagem enviada a um objeto reflexivo: ela é interceptada e desviada para o meta-nível. Usualmente ela pode chegar inalterada ao objeto do nível base, mas ela pode ser trocada, e eventualmente pode não ser entregue.

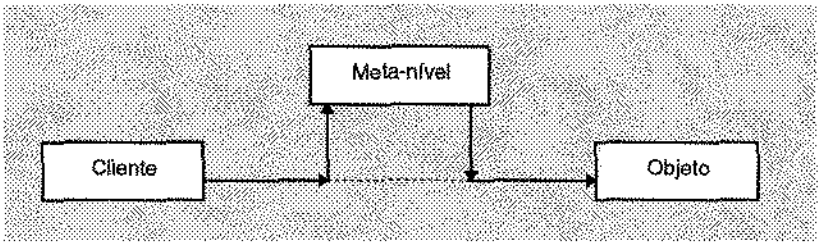


Figura 4.2 Interceptação de mensagens.

- **Como metaobjetos e objetos se relacionam.**

O terceiro ponto diz respeito à cardinalidade do relacionamento entre objetos e metaobjetos. Em alguns protocolos cada objeto do nível base é associado a um único metaobjeto e vice-versa. Alguns permitem que cada objeto do nível base seja associado a um grupo de metaobjetos. Outros permitem um relacionamento n:m.

Os conceitos apresentados nesta Subseção serão esclarecidos e exemplificados no próximo item no qual o protocolo de metaobjetos de OpenC++1.2 é apresentado.

4.4.3 OpenC++ 1.2

OpenC++ 1.2 [Chi93], uma extensão de C++ que dá apoio à reflexão, foi a linguagem utilizada neste trabalho. Suas principais características são [Chi93, Lis97]:

- Objetos no nível base podem ser reflexivos ou não. Objetos reflexivos podem possuir métodos e atributos reflexivos e são associados a metaobjetos, os quais controlam seu comportamento. Um objeto não-reflexivo é um objeto C++ convencional.
- Cada objeto reflexivo pode ser associado a um único metaobjeto. Ou seja, metaobjetos não podem ser compartilhados.
- Um metaobjeto pode interceptar e modificar o efeito de operações básicas tais como, a chamada de métodos reflexivos e o acesso (leitura e escrita) a atributos reflexivos do objeto reflexivo do nível base a ele associado.
- Um metaobjeto é uma instância de uma classe de meta-nível. Classes de meta-nível são associadas a classes do nível-base em tempo de compilação.
- Classes do meta-nível são derivadas da classe predefinida *MetaObj*, a qual define os métodos para um metaobjeto controlar o objeto reflexivo a ele associado. As subclasses de *MetaObj* podem redefinir seus métodos.
- Um nome_de_categoria pode ser associado a métodos e atributos reflexivos, permitindo que um metaobjeto altere tais métodos e atributos de acordo com a categoria especificada.
- Um programa em OpenC++1.2 é um programa C++ que contém diretivas para declarar classes, atributos e métodos como reflexivos. Essas diretivas são incorporadas como comentários C++ que começam com *//MOP*. Para declarar métodos e atributos como reflexivos eles devem ser precedidos com a diretiva: *//MOP reflect*. Para associar uma classe do nível base com uma meta-classe, a diretiva *//MOP reflect class* deve ser usada.

- O protocolo de metaobjetos de OpenC++1.2 baseia-se na intercepção de mensagens (ver Figura 4.3). Quando um método reflexivo é chamado no nível base, a chamada é capturada e manipulada no meta-nível pelo metaobjeto associado ao objeto reflexivo em questão. No caso de reflexão na invocação de um método o meta-método *Meta_MethodCall* (①) é então executado. Neste meta-método o usuário descreve como o metaobjeto deve controlar (modificar ou observar) o objeto base. Normalmente, *Meta_MethodCall* invoca o método do nível base usando outro método *Meta_HandleMethodCall* (②③) e executa algum processamento antes ou depois de tal chamada sendo que a chamada do método do nível base pode nem ser efetivada. No final do *Meta_MethodCall*, o controle retorna para o nível base e os resultados são retornados como em uma chamada de método normal (④). Isto implica que, quando um método reflexivo é chamado, o metaobjeto precisa saber qual é o método reflexivo em questão, quais são seus argumentos e onde armazenar seus resultados. Estas informações estão disponíveis em objetos "containers" da classe *ArgPac*, durante a invocação de um método reflexivo. Esta classe comporta-se como uma pilha e implementa operações *push()* e *pop()* para cada tipo primitivo.

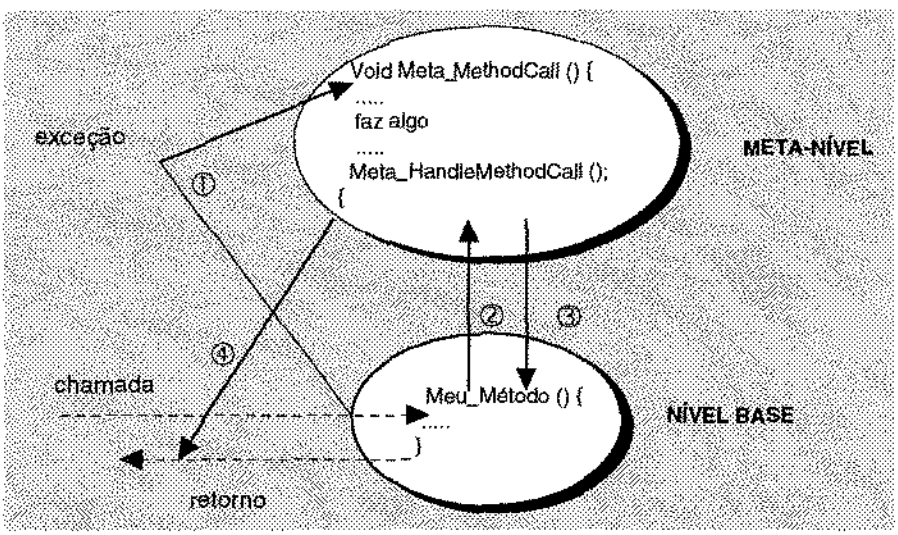


Figura 4.3 Chamada de um método reflexivo em OpenC++1.2 [Chi93].

A Figura 4.4 exemplifica as características listadas acima. Na linha 5, o método *método1()* é definido como reflexivo através da diretiva *//MOP reflect:*.A diretiva da linha 7, *//MOP reflect class Exemplo: ExemploMetaobj*, indica que um objeto da classe *Exemplo* pode ser reflexivo e seu meta-objeto será uma instância da classe *ExemploMetaObj*. Quando o pré-processador encontra a diretiva *//MOP reflect class Exemplo: ExemploMetaObj*, na linha 7, ele produz uma nova subclasse da classe *Exemplo*, cujo nome é nome da classe original acrescido da palavra-chave *refl_*. A linha 8 mostra a definição da meta-classe

ExemploMetaObj que herda e redefine alguns métodos da classe padrão *MetaObj*. Objetos normais são criados como instâncias de *Exemplo*, linha 16, e objetos reflexivos são criados como instâncias de *refl_Exemplo*, linha 17; Como e2 é um objeto reflexivo e *método1* é um método reflexivo, a chamada *e2.metodo1*, linha 18, é interceptada e a execução é transferida para o seu metaobjeto. Como *e1* é um objeto não-reflexivo, a chamada *e1.metodo1* é feita normalmente.

```
1.  class Exemplo{
2.  public:
3.    Exemplo();
4.  //MOP reflect:
5.    metodo1();
6.  };

7.  // MOP reflect class Exemplo: ExemploMetaObj

8.  class ExemploMetaObj : public MetaObj{
9.  public:
10.   void Meta_MethodCall(Id m_id, ArgPac& args, ArgPac& reply){
11.   // faz algo
12.   Meta_HandleMethodCall(m_id,args,reply);
13.   }
14. };

15. main () {
16. Exemplo e1;            // objeto não-reflexivo
17. refl_Exemplo e2;       // objeto reflexivo
18. e2.metodo1;
19. e1.metodo1;
20. };
```

Figura 4.4 Um exemplo em OpenC++ 1.2

4.5 Uso de Reflexão Computacional

Atualmente reflexão computacional vem sendo utilizada em diversas áreas como sistemas distribuídos, linguagens de programação e tolerância a falhas, com o objetivo de se obter sistemas mais flexíveis e modulares.

Existem muitas linguagens que foram projetadas ou estendidas para dar suporte a reflexão. Estudos sobre as características de tais linguagens são apresentados em [Cor97].

Programação reflexiva tem se mostrado útil para estruturação das características não-funcionais de um sistema, porque permite que tais características sejam implementadas e alteradas de acordo com novas necessidades sem, no entanto, interferir nas características funcionais. Quando usada desta forma, reflexão computacional é uma técnica bastante útil na construção de mecanismos genéricos por torná-los mais flexíveis. Nesse sentido vários trabalhos foram desenvolvidos na área de tolerância a falhas [Cor97], [FNP+95] entre

outros; na área de sistemas distribuídos reflexão é usada para o gerenciamento de recursos de sistemas de forma transparente e não intrusiva.

A proposta deste trabalho é o uso de reflexão computacional no teste de injeção de falhas por software. As definições abaixo revelam que a reflexão computacional parece oferecer um meio natural de injeção por software:

Reflexão computacional é a capacidade de um sistema computacional de desviar do seu processo normal de execução, fazer deduções ou computações em um outro nível de processamento e retornar ao nível de execução traduzindo o impacto das decisões, para então retomar o processo de execução. [Lis97]

Injeção de falhas por software basicamente consiste em interromper a execução da aplicação de alguma forma (normalmente inserido alguma exceção de software ou executando a aplicação no modo traço) e executar um código de injeção de falhas por software específico que emula falhas em diversas partes do sistema. [CMS95]

A idéia aqui é implementar o código de injeção de falhas e monitorização de seus efeitos como um meta-programa que possa ser incorporado de forma fácil, transparente e o menos intrusiva possível em relação ao código da aplicação a ser testada. A conexão causal entre o meta-programa de injeção de falhas e o programa da aplicação alvo implica que as alterações provocadas pelo código de injeção de falhas são refletidas no estado da aplicação alvo, e o efeito das alterações na aplicação alvo podem ser observados e armazenados pelo meta-programa.

4.6 Benefícios da Arquitetura Reflexiva para Injeção de Falhas

Reflexão e orientação a objetos apresentam diversas vantagens para injeção de falhas por software:

- Redução da perturbação no código da aplicação alvo. Os requisitos de injeção de falhas e monitorização (encapsulados em meta-objetos) ficam separados dos requisitos associados ao propósito da aplicação (encapsulados em objetos).

- Reutilização de componentes. A separação de domínios permite que os aspectos de injeção de falhas e monitorização sejam desenvolvidos independentemente da aplicação, propiciando a reutilização tanto de objetos da aplicação como de metaobjetos.

- Flexibilidade na injeção de falhas. O uso de metaobjetos possibilita injetar diferentes tipos de falhas em diferentes objetos do nível base, de acordo com suas características.

- Facilidade de uso. Metaojetos de injeção de falhas e monitorização podem ser facilmente incorporados e retirados da aplicação em teste.

4.7 Resumo

Neste capítulo reflexão computacional foi apresentada como uma técnica promissora quando é necessário dispor de informações especiais sobre a representação do programa, de forma independente da aplicação, como no caso da injeção de falhas por software.

Reflexão computacional permite a um sistema fazer computações sobre seus próprios dados, tendo como objetivo promover alterações e adaptações dinâmicas na estrutura e/ou no comportamento do sistema. Isso é possível através de uma arquitetura de níveis denominada arquitetura reflexiva, composta por um meta-nível, onde se encontram as estruturas e as ações a serem realizadas sobre o sistema alvo, localizado no nível base. A grande vantagem da arquitetura reflexiva é que ela permite estruturar o software de uma aplicação considerando a separação de domínios: o que diz respeito ao domínio da aplicação é implementado no nível base e o que não está relacionado a funcionalidade da aplicação é implementado no meta-nível. Essa vantagem é bastante adequada para solucionar o problema de perturbação no código da aplicação alvo no teste por injeção de falhas por software apresentado no capítulo anterior. Assim, consideramos que a aplicação a ser validada está localizada no nível base e incorporamos a ela o código de injeção de falhas e monitorização como um meta-nível.

Em linguagens orientadas a objetos, a reflexão pode ser estrutural ou comportamental. A reflexão é estrutural quando realizada por meio de meta-classes que permitem alterar todas as instâncias da classe; enquanto que reflexão comportamental é obtida através de um modelo de metaobjetos no qual é possível associar objetos a metaobjetos que podem observar e controlar comportamento dos objetos associados a eles. O modelo mais adequado para injeção de falhas por software em tempo de execução, é o modelo de metaobjetos uma vez que este permite alterar o estado dinâmico de objetos simulando a ocorrência de falhas. O modelo de metaclasses é mais adequado para injeção estática.

O relacionamento entre metaobjetos e objetos é definido através de um PMO. Existem diferentes PMO's desenvolvidos de acordo com as características da linguagem base e seu enfoque particular no modelo de objetos. Em resumo a função de um PMO é definir: como deve ser feita a materialização dos dados do nível base para o meta-nível; a cardinalidade da associação metaobjeto-objeto e, como transferir o controle de execução de um nível para o outro.

A linguagem OpenC++ 1.2 foi apresentada para exemplificar um protocolo de metaobjetos e principalmente para apresentar ao leitor algumas das características da linguagem usada neste trabalho com o objetivo de facilitar o entendimento das decisões de projeto e implementação da ferramenta, as quais serão apresentadas no próximo capítulo.

Foram citadas as atuais aplicações de reflexão computacional em diversas áreas e introduzimos como aplicar reflexão computacional como técnica para preparar aplicações tolerantes a falhas para testes de injeção de falhas por software e para realizar tais testes.

Por fim, foram listadas as diversas vantagens do uso de reflexão e orientação a objetos para injeção de falhas por software: redução da perturbação, reutilização de componentes, flexibilidade de injeção de falhas e facilidade de uso.

O próximo capítulo descreve a ferramenta de injeção de falhas por software desenvolvida utilizando programação reflexiva.

Capítulo 5

FIRE: Uma Abordagem Reflexiva Para Injeção de Falhas Por Software

O objetivo deste capítulo é apresentar a ferramenta de injeção de falhas por software FIRE (*Fault Injection using a REflective Architecture*), destacando como reflexão computacional foi utilizada para injeção e monitorização.

5.1 Introdução

Conforme mencionado anteriormente, a injeção de falhas pode ser realizada durante a compilação ou durante a execução. Este trabalho propõe o uso de reflexão computacional para instrumentar aplicações para testes por injeção de falhas por software durante a execução, sendo assim, a injeção estática fica fora do escopo deste trabalho.

A função da instrumentação, neste contexto, consiste em adicionar à aplicação alvo funcionalidades de injeção de falhas e monitorização de seus efeitos. Idealmente, uma abordagem de instrumentação deve ser o menos intrusiva possível, sem alterar a estrutura original da aplicação e interferindo o mínimo possível na sua execução. Isto requer que a instrumentação seja funcionalmente independente da aplicação alvo. Outras qualidades importantes de uma abordagem de instrumentação são: *modularidade*, a fim de facilitar a incorporação de novas características, *reusabilidade*, para permitir fácil adaptação em aplicações alvo diferentes, e *portabilidade*, para permitir a sua utilização em plataformas (software/hardware) diferentes com alterações mínimas. Como visto no capítulo anterior, a combinação de orientação a objetos com reflexão permite a definição de uma abordagem de instrumentação com as características acima listadas.

A fim de ilustrar como a abordagem reflexiva pode ser usada para instrumentar aplicações orientadas a objetos para testes de injeção de falhas, e, principalmente, verificar as facilidades e limitações da utilização de reflexão para essa técnica, foi implementada a ferramenta FIRE (*Fault Injection Using a REflexive Architecture)* [RM98a], [RM98b] e [RM98c]. FIRE foi implementada utilizando a linguagem C++ e o pré-processador OpenC++ 1.2, o qual prove as características reflexivas. Assim sendo, FIRE destina-se à validação de aplicações implementadas em C++ e OpenC++1.2 [Chi93].

FIRE usa as facilidades oferecidas pelo protocolo de metaobjetos de OpenC++1.2 para injetar falhas e monitorizar seus efeitos. A estrutura original da aplicação alvo não é alterada, apenas algumas diretivas (no formato de comentários C++) e algumas indicações são introduzidas; não são inseridas instruções de *traps,* não é necessário executar a aplicação no modo traço, e não são utilizadas características especiais do hardware nem do ambiente de desenvolvimento ou operacional da aplicação alvo. FIRE pode injetar falhas permanentes, intermitentes e transientes. O modelo de falhas proposto para a ferramenta apresenta uma nova abordagem de teste baseada na idéia de testes de circuitos em hardware, o qual inclue falhas internas e falhas externas. Atualmente, o mecanismo de injeção de falhas baseia-se na corrupção de valores de argumentos de métodos e de valores de atributos durante a leitura ou atualização.

A organização do capítulo é a seguinte: a Seção 5.2 apresenta os componentes, a organização e o funcionamento da FIRE; a Seção 5.3 descreve como utilizar a ferramenta; a Seção 5.4 resume e conclui este capítulo.

5.2 Apresentando FIRE

Esta Seção apresenta a estrutura da FIRE (o tipo e a forma de organização de seus componentes e o modo como esses componentes interagem entre si), mostra como metaobjetos da FIRE são ativados para injetar falhas e monitorizar objetos da aplicação alvo e apresenta os tipos de falhas considerados pela ferramenta. Assim as subseções 5.2.1, 5.2.2, 5.2.3, 5.2.4 e 5.2.5 tratam, respectivamente, da arquitetura da ferramenta, do seu modelo de falhas, do seu mecanismo ativação, do seu método de injeção e do seu método de monitorização.

5.2.1 Arquitetura da FIRE

A arquitetura reflexiva para injeção de falhas e monitorização proposta é apresentada na Figura 5.1. Seus principais componentes são: a *aplicação alvo* com os mecanismos de tolerância a falhas a serem validados, a *biblioteca de meta-nível* e o *controlador de experimentos.*

A *aplicação alvo* é uma aplicação tolerante a falhas orientada a objetos, implementada em C++ ou OpenC++1.2, que deve ser submetida a uma fase de instrumentação. Nessa fase, as diretivas de reflexão são introduzidas no código fonte da aplicação para indicar quais classes e seus respectivos métodos e atributos serão reflexivos, ou seja, poderão sofrer injeção e/ou monitorização. Como em OpenC++1.2 instâncias de classes reflexivas podem ser reflexivos ou não, também é necessário indicar quais objetos serão reflexivos.

A *biblioteca de meta-nível* é anexada à aplicação alvo em tempo de compilação e define metaobjetos denominados *escalonadores*, os quais são responsáveis pelo controle de execução no meta-nível; um *escalonador* é composto por um *injetor* e *sensor*[9]. Cada objeto reflexivo possui o seu metaobjeto *escalonador*. A Subseção 5.2.3 descreve o relacionamento entre *escalonadores* e objetos da aplicação alvo. Metaobjetos *escalonadores* capturam mensagens reflexivas (chamadas de métodos ou acessos a atributos) enviadas ao seus respectivos objetos reflexivos; verificam a categoria (monitorização e/ou injeção) associada às mensagens; decidem qual o tipo de ação a ser realizada (injeção de falhas, coleta de dados ou execução normal do serviço solicitado no nível base), e podem delegar operações a um *injetor* ou *sensor*. Um *injetor* corrompe valores do objeto base. Um *sensor* coleta dados sobre a ativação dos mecanismos de tolerância a falhas e sobre o valor de argumentos e atributos de objetos da aplicação definidos para ser monitorizados, além disso coleta dados sobre o processo de injeção de falhas.

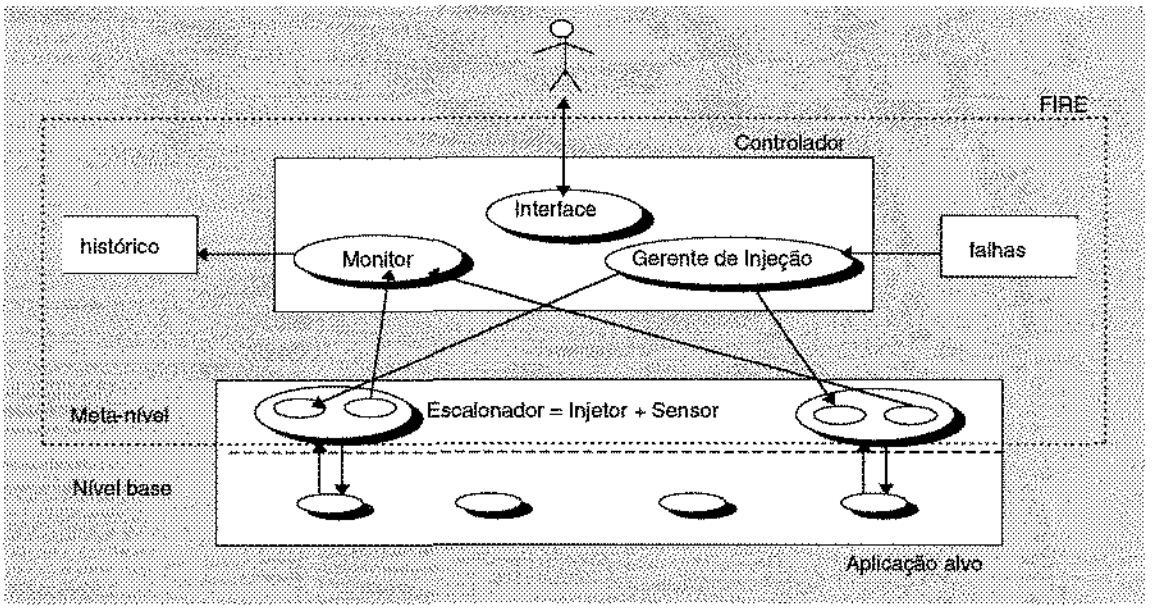


Figura 5.1 Uma arquitetura reflexiva para injeção de falhas e monitorização.

O *controlador* coordena os experimentos: inicia o processo da aplicação alvo, controla (via *escalonador*) *injetores* e *sensores*, coleta informações referentes ao término do processo da aplicação e armazena os dados coletados pelo *sensores*. O controlador é composto por três objetos: *gerente de injeção*, *monitor* e *interface do usuário*. O *gerente de injeção* lê a falha ser injetada do arquivo *falhas* e a transfere ao *escalonador*, que por sua

[9] O escalonador é então um objeto composto, criado devido a uma limitação da linguagem OpenC++ 1.2: cada objeto pode ser associado a um único metaobjeto. A separação das funções de injeção e monitorização é necessária para fins de modularidade e reusabilidade, justificando assim a necessidade de um objeto composto.

vez a transfere ao *injetor*. O *monitor* recebe os dados enviados pelos *sensores* e os armazena no arquivo de *histórico* do experimento. A *interface do usuário* auxilia os testadores a observar e manipular a execução de experimentos.

No decorrer do capítulo os componentes aqui apresentados serão explicados mais detalhadamente.

5.2.2 Modelo de Falhas

Esta Subseção descreve o modelo de falhas proposto para a FIRE. O modelo é apresentado em função da visão de programa da ferramenta e da caracterização das falhas em termos temporais e de localização. No final é apresentada uma discussão em relação à representatividade do modelo proposto.

Visão de Programa

Como mencionado no capítulo 3, FIRE "enxerga" a aplicação alvo como um conjunto de objetos trocando mensagens de solicitação de serviços. Através de *escalonadores*, FIRE pode injetar falhas em alguns objetos quando é feita a solicitação de serviços que envolvem atributos ou métodos reflexivos. As falhas são injetadas como o objetivo de corromper o estado de objetos e forçar a ativação dos mecanismos de tolerância a falhas.

A abordagem de teste FIRE, faz um paralelo com a abordagem de teste proposta por Hoffman [Hof89], na qual ele faz uma adaptação dos conceitos de testes de circuitos de hardware para o teste de regressão de módulos. Hoffman considera módulos como circuitos de software, FIRE considera objetos como circuitos de software. Quando essa consideração é feita, a mesma correspondência utilizada por Hoffman pode ser utilizada; ou seja, cada método (ou atributo) público de um objeto pode corresponder a um pino de um circuito, ou a um conjunto relacionado de pinos.

Com essa correspondência um outro paralelo pode ser feito, agora com a injeção física de falhas. Como visto no Capítulo 2, a injeção física de falhas pode ser realizada diretamente sobre os pinos do circuito alvo (através da submissão de voltagem, por exemplo) ou indiretamente (através de interferências magnéticas, por exemplo) alterando as funções internas do circuito. Fazendo um paralelo com a injeção por hardware, FIRE pode injetar falhas internas, afetando as características privadas[10] do objeto, ou falhas externas, afetando a interface do objeto. Na injeção física há uma dificuldade de controle e monitorização: é difícil prever o tipo de erro que será provocado com a injeção ou mesmo saber quais erros foram provocados. Isso não acontece na FIRE, pois o seu modelo de falhas oferece um boa

[10] Embora OpenC++ 1.2 não permita reflexão em características privadas de um objeto, falhas internas podem ser consideradas para as características protegidas.

controlabilidade e *sensores* podem retratar o que está acontecendo com o estado de um objeto quando esse é consultado ou atualizado por outros objetos.

Caracterização das falhas

A Figura 5.2 apresenta o formato padrão de uma instância de falha.

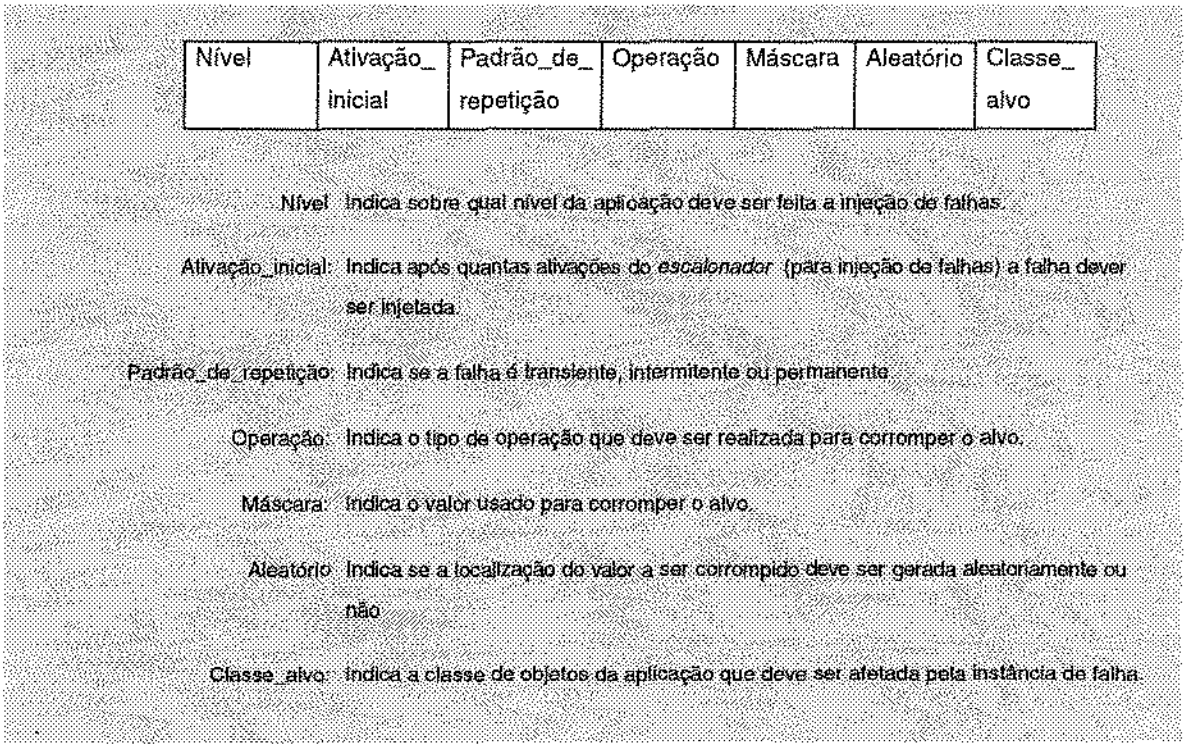


Figura 5.2 Formato de uma instância de falha na FIRE.

Os parâmetros `padrão_de_repetição`, `ativação_inicial` e `classe_alvo` são utilizados por *escalonadores* para verificar se um *injetor* deve ou não ser ativado (ver 5.2.3).

Os parâmetros `operação` e `máscara` são utilizados pelos *injetores* para corromper valores dos objetos durante a injeção de falhas.

O parâmetro `aleatório` é utilizado pelo *injetor* para calcular a localização do alvo de injeção de falhas na corrupção de argumentos de um método. O usuário pode querer corromper um determinado argumento ou um argumento aleatório (ver 5.2.4).

O parâmetro `nível` é utilizado pelo *escalonador* para determinar em qual nível da aplicação estão os dados a serem corrompidos. Essa característica é importante em situação em que a aplicação alvo já é reflexiva, ou seja, já possui um meta-nível. Nesse caso, a

biblioteca de injeção e monitorização pode ser introduzida como um meta-meta-nível e a aplicação passa a ter dois possíveis níveis alvo: o nível base ou o meta-nível.

Os itens seguintes apresentam os aspectos temporais e de localização do modelo de falhas.

- Visão temporal

A visão temporal do modelo de falhas FIRE é caracterizado pelos parâmetros `padrão_de_repetição` e `ativação_inicial` de uma falha (ver Figura 5.2) combinados com o mecanismo de interceptação de mensagens. Cada *escalonador* conta quantas vezes ele foi ativado para injeção de falhas e toma esse número como base para controlar a invocação do seu *injetor*. Uma instância de falha será injetada quando o valor de sua `ativação_inicial` for igual ao contador de ativações do *escalonador* e, a partir daí reinjetada, quantas vezes conforme indicado pelo seu`padrão_de_repetição`.

FIRE injeta falhas permanentes, intermitentes e transientes. A injeção de uma falha transiente na aplicação, implica que cada objeto alvo vai ser corrompido uma única vez em algum de seus pontos de injeção (execução de métodos reflexivos ou atualização/leitura de atributos reflexivos). A injeção de uma falha intermitente na aplicação, implica que cada objeto alvo vai ser corrompido periodicamente em algum de seus pontos de injeção. A injeção de uma falha permanente na aplicação, implica que cada objeto alvo vai ser sempre corrompido em todos os seus pontos de injeção.

Se, por exemplo, uma falha tem `ativação_inicial=4` e o seu `padrão_de_repetição=permanente`, na quarta ativação do *escalonador* (com categoria de injeção) ela será injetada e a partir daí em todas as ativações.

- Localização das Falhas

A localização do alvo da injeção de falhas possui vários níveis de detalhamento. Em um primeiro passo o usuário determina quais classes da aplicação poderão ter objetos que sofrerão injeção de falhas, isto é, serão reflexivas (com categoria de injeção). Em cada classe reflexiva o usuário define quais métodos e/ou atributos serão pontos de injeção. Quando a injeção é sobre argumentos de um método é preciso ainda indicar qual argumento será corrompido. É necessário também indicar quais objetos das classes reflexivas serão reflexivos.

Feito esse primeiro "mapa de injeção", num segundo momento, durante a geração das falhas, o usuário define para cada instância de falha, qual é a sua `classe_alvo`. Cada instância de falha é injetada apenas em objetos reflexivos cuja classe seja igual ao da sua `classe_alvo`. Dessa forma é possível injetar falhas diferentes em objetos de classes diferentes.

Representatividade do modelo de falhas FIRE

A representatividade dos modelos de falhas de hardware era um ponto normalmente apontado como limitação da técnica de injeção por software. Questionava-se se os erros gerados por métodos de software eram realmente próximos dos erros gerados por falhas de hardware reais, [Car95] lista uma série de trabalhos recentes de grupos diversos que demonstram a validade e a representatividade dessa técnica e de seus modelos.

FIRE propõe um outro nível de abstração para injeção de falhas no cenário de orientação a objetos: o objeto. As falhas injetadas podem conduzir os objetos a entrarem num estado de erro fazendo com que seus serviços prestados não estejam conforme sua especificação. Um defeito de um objeto resulta em uma falha para os demais objetos que interagem com ele e para o sistema.

Uma característica interessante do modelo de falhas FIRE é que ele não é específico para falhas de software ou hardware. O modelo é bastante genérico. Por exemplo, se em um determinado momento um atributo de um objeto está com valor incorreto, este erro pode ter sido provocado por qualquer tipo de falha (de barramento, memória ou processador ou ainda por alguma falha de software).

Para que a representatividade do modelo proposto possa ser demonstrada é necessário realizar um número maior de experimentos e um estudo comparativo dos resultados obtidos nos experimentos com a FIRE com aqueles obtidos por outras ferramentas que usam outros modelos de falhas.

5.2.3 Mecanismo de Ativação

Como mencionado anteriormente, na abordagem adotada neste trabalho falhas são injetadas dinamicamente, assim, é necessário um mecanismo para ativar os *injetores* nos momentos desejados durante a execução da aplicação alvo, bem como os *sensores*. Os mecanismos de ativação utilizados pela maioria das ferramentas de injeção de falhas por software são baseados na ativação espacial (quando a execução da aplicação passa por um endereço previamente especificado [KKA92], e/ou certos dados são utilizados [CMS95]) ou na ativação temporal (após intervalo de tempo predeterminado). FIRE apresenta uma nova forma de ativação: a interceptação de mensagens.

A intercepção de mensagens é a característica provida pelo protocolo de metaobjeto da linguagem OpenC++1.2 para transferir o fluxo de execução do nível base para o meta-nível, e pode ocorrer na chamada de métodos reflexivos ou no acesso a atributos reflexivos (ver capítulo 4, Seção 4.3.3).

Em FIRE a ativação de *injetores* e *sensores* é feita indiretamente através da ativação de *escalonadores*. A interceptação de mensagens é utilizada para "acordar" um *escalonador*,

ou seja, para passar o fluxo de execução para o meta-nível. No meta-nível o *escalonador*, quando necessário, ativa um *injetor* e/ou um *sensor* para realizar sua função.

Criação e Ativação de *Escalonadores*

Quando um objeto reflexivo é criado, o seu metaobjeto também é criado e eles são associados um ao outro. Assim, quando um objeto reflexivo da aplicação alvo é instanciado, automaticamente um *escalonador* é instanciado e associado a ele. Um *escalonador* fica inativo até que uma mensagem envolvendo um método ou um atributo reflexivos seja enviada ao seu objeto referente[11]. A Figura 5.3 mostra como um *escalonador* é ativado durante uma leitura de um atributo reflexivo.

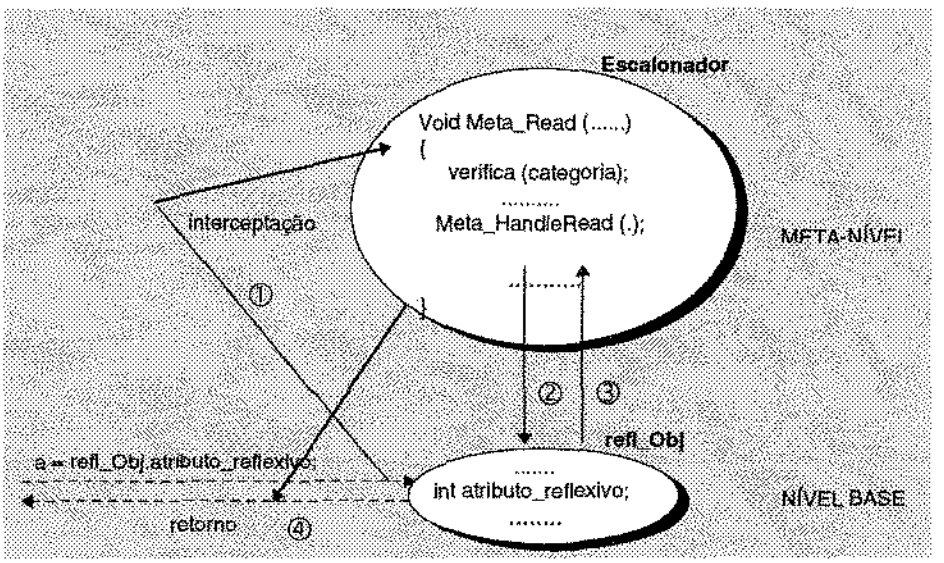


Figura 5.3 Ativação de um *escalonador* na leitura de um atributo reflexivo.

Quando uma mensagem de leitura de um atributo reflexivo é enviada a um objeto reflexivo, essa mensagem é capturada e manipulada no meta-nível pelo *escalonador* a ele associado, através do método *Meta_Read* (①). Através desse meta-método, tal *escalonador* verifica a categoria[12] associada ao atributo reflexivo, executa a operação determinada e então faz a leitura do atributo no nível base através do método *Meta_HandleRead* (②③). No final do *Meta_Read*, o controle retorna para o nível base e o cliente recebe o valor lido (④). O processo é o mesmo para atualização de atributos reflexivos e chamada de métodos reflexivos.

---

[11] Diz-se objeto referente de um metaobjeto, o objeto reflexivo do nível base a ele associado.

[12] Como um objeto pode ser injetado e/ou monitorado pelo metaobjeto, *escalonadores* precisam de um nome-de-categoria (conforme capítulo 4 item 4.3.3) para identificar a ação a ser executada de acordo com o método ou atributo em questão.

A interceptação de mensagem da linguagem OpenC++1.2 oferece um modo simples e eficiente para ativar o código responsável pela injeção e/ou monitorização.

Ativação de *Injetores* e *Sensores*

Uma vez que o fluxo de execução está no meta-nível, o *escalonador* ativo decide o tipo de operação a ser realizada. Para tomar essa decisão ele utiliza o *nome-de-categoria* associado a mensagem reflexiva.

Se a categoria for de injeção, o *escalonador* tenta ativar o seu *injetor*. O *injetor* será ativado se a classe do objeto referente estiver relacionada como alvo na falha a ser injetada e se for momento de injetar a falha. Se o objeto referente não estiver envolvido no processo de injeção de falhas ou se o momento de injeção ainda não foi alcançado a mensagem capturada é executada normalmente (sem injeção e/ou monitorização). Quando a injeção de falhas é sobre um resultado de função ou sobre um valor lido de um atributo, o *injetor* é ativado após a execução do serviço solicitado. No caso de um argumento ou de um valor a ser atribuído a um atributo, o *injetor* é invocado antes.

Se a categoria for de monitorização o *escalonador* ativo invoca o seu *sensor* para coletar os dados. A categoria de monitorização indica para o *sensor* se os valores a serem monitorizados são um resultado, argumentos de um método, um atributo ou a chamada de um método. No caso de atributos a categoria também indica qual é o tipo do atributo (*int, double,* ponteiro, *char* ou *char**). No caso de argumentos ou de retorno de um método, o *sensor* consulta uma lista de interfaces de métodos para saber como coletar adequadamente os valores. Essa lista tem que ser dada pelo usuário antes da execução do experimento.

Se a categoria for de monitorização&injeção valem as mesmas regras definidas para injeção de falhas. Caso o *injetor* não deva ser ativado o *sensor* também não o é.

5.2.4 Método de injeção

O método de injeção de falhas da FIRE, na versão atual, baseia-se na corrupção de valores de dados (atributos reflexivos e argumentos de métodos reflexivos) de objetos reflexivos reificados para o meta-nível. Em OpenC++1.2 esses valores são passados para o meta-nível através de objetos armazenadores da classe *ArgPac*, os quais apresentam a mesma interface de uma pilha.

Um *escalonador* conhece o endereço do topo da pilha de dados reificados e o informa para seu *injetor*. Um *injetor* aplica a máscara e a operação ao conteúdo desse endereço somado a um deslocamento. O valor do deslocamento pode ser determinado pelo usuário, pela ferramenta ou ainda por ambos. (1) O valor do deslocamento pode ser informado pelo usuário quando a injeção é baseada na corrupção de argumentos. Nesse caso, o usuário escolhe um determinado argumento para ser corrompido. (2) Ainda quando a injeção de

falhas é em argumentos, o usuário pode informar um deslocamento máximo para que a ferramenta calcule um deslocamento aleatório. Nesse caso, qualquer argumento na pilha pode ser corrompido. (3) Quando a injeção é baseada na corrupção de atributos e de resultados de funções, o deslocamento é dado pela ferramenta e é sempre zero, pois esses valores sempre ocupam o topo da pilha.

5.2.5 Método de monitorização

A monitorização é um aspecto muito importante da técnica de injeção de falhas. A monitorização é necessária para detectar a ativação de falhas, para coletar outras informações relevantes sobre o impacto das falhas ou mesmo para registrar o processo de injeção de falhas. São poucos os trabalhos de injeção de falhas que tratam da monitorização do sistema alvo, alguns exemplos são [KIT93], que usa um software para monitorização do fluxo de execução do núcleo do sistema alvo e os valores de variáveis chaves e dados, [KI94] onde instruções de *trap* são inseridas em determinadas localizações das aplicações alvo para prover informações como o fluxo de controle e dados importantes, e [CMS95] que usa características especiais do hardware processador alvo para obter informações sobre o comportamento da aplicação alvo na presença de falhas.

O protocolo de interceptação de mensagens de OpenC++ 1.2, apesar de não permitir a observação de características privadas de objetos, provê facilidades que permitem uma boa monitorização das mensagens trocadas entre os objetos da aplicação alvo. FIRE utiliza essas facilidades para monitorizar os valores de atributos e argumentos e a chamada de métodos.

A monitorização de chamada de métodos reflexivo é importante para detectar, por exemplo, se determinado MTF foi ativado.

A monitorização de atributos reflexivos e argumentos de métodos reflexivos pode ser feita de duas formas: toda vez que a mensagem reflexiva envolvendo esses valores for interceptada pelo *escalonador* ou quando esses valores forem alvo de injeção de falhas. Na primeira forma o método ou atributo deve ser associado a uma categoria de monitorização e na segunda forma a uma categoria de injeção&monitorização. Essa última categoria permite ao usuário observar se uma falha injetada realmente provocou o erro esperado no objeto.

Para que os valores dos argumentos e atributos possam ser devidamente coletados, é preciso que os *sensores* conheçam o tipo de cada um deles para uma interpretação correta. Na versão atual da FIRE essa informação deve ser fornecida pelo usuário antes da execução dos experimentos, através de categorias ou através de um arquivo contendo a assinatura dos métodos que serão monitorizados. Uma possibilidade de evolução da ferramenta é incorporar um componente que percorra o código fonte da aplicação alvo e recolha esse tipo de informação.

Os dados coletados pelos *sensores* durante a execução de um experimento são armazenados em um arquivo de *histórico* (tipo texto) para posterior análise.

5.3 Usando FIRE

Esta Seção descreve os passos envolvidos na utilização da FIRE em experimentos de injeção de falhas por software. Um experimento de injeção de falhas por software, normalmente, é constituído das seguintes fases: preparação da aplicação alvo, execução e análise dos resultados. FIRE não cobre a fase de análise de resultados: os dados de monitorização são coletados durante a execução do experimento e a análise deve ser feita posteriormente.

Cada experimento possui um arquivo de falhas, o qual contém todas as instâncias de falhas a serem injetadas, e um arquivo de histórico, o qual armazena todos os dados coletados durante o experimento. Uma aplicação pode ser preparada para uma série de experimentos ou para apenas um experimento. Os próximos itens descrevem a fase de preparação e a fase de execução.

5.3.1 Fase de Preparação

A fase de preparação envolve dois passos: a instrumentação da aplicação alvo e a geração de falhas. Na atual versão FIRE não automatiza a instrumentação da aplicação apenas as falhas podem ser geradas automaticamente pelo *gerador de falhas* FIRE.

Instrumentação da aplicação alvo

Para realizar testes de injeção de falhas utilizando FIRE é imprescindível que o código fonte da aplicação esteja disponível e que o usuário tenha um bom entendimento da estrutura e do funcionamento da aplicação. O código fonte deve ser analisado e o usuário deve estabelecer um planejamento para a realização de experimentos.

Cada experimento deve ser bem definido para que os resultados obtidos sejam significativos. Um experimento é dito bem definido quando as falhas são planejadas para serem injetadas nos serviços de objetos relacionados com a característica que está sendo validada durante o experimento, e ainda quando os pontos de monitorização definidos permitem uma boa observação do comportamento durante o experimento.

O primeiro passo é definir qual a característica (o mecanismo de tolerância a falhas ou o mecanismo de detecção de erro, por exemplo) a ser validada. Feito isso, o usuário deve verificar quais são as classes, atributos e/ou métodos e objetos envolvidos com tal característica e marca-los como pontos de ativação para injeção e/ou monitorização. Ou seja, incluir as diretivas de reflexão para indicar quais métodos ou atributos serão reflexivos.

Para indicar o que deve ser feito em cada ponto de ativação, o usuário deve associá-los a categorias. As categorias desempenham um papel muito importante na instrumentação da aplicação alvo. São as categorias que definem o tipo de ação que deve ser realizada pelos *escalonadores*. Uma listagem dos nomes de categorias usados atualmente na FIRE é apresentada no Apêndice A.

A instrumentação abrange as seguintes ações:

- incluir a definição do *escalonador* ①;
  `# include "sheduler.h"`

- indicar quais classes poderão ter instâncias que poderão sofrer injeção e/ou monitorização ② (através da inclusão da diretiva abaixo);
  `//MOP reflect class` *`nome_da_classe`*`:` **`SchedulerMetaObj;`**

- indicar quais métodos e/ou atributos deverão ser injetados e/ou monitorados ③ (através da inclusão da diretiva abaixo);
  `//MOP reflect` `(`*`category_name`*`):`

- indicar quais objetos são injetados e/ou monitorados ④ (acrescentando o prefixo `refl_` ao nome da classe na instanciação do objeto).
  `refl``_`*`nome_da_classe`* `objeto;`

A Figura 5.4 apresenta como fica o código fonte[13] de uma aplicação instrumentada para injeção de falhas e monitorização. No exemplo dessa figura, a diretiva na linha 20 indica que objetos da classe `Stack` podem ter um metaobjeto da meta-classe `SchedulerMetaObj` associado a eles. A palavra `refl_` define que `obj` vai ser reflexivo. As diretivas nas linhas, 11 e 13, as quais indicam que os métodos `push(int i)` e `marshaling()` são reflexivos, levam um `nome_de_categoria`. O método `push(int i)` é marcado como ponto de injeção de falhas e o alvo é o seu argumento, pois a categoria de reflexão associada ao método é `injection_0`. O número 0 na categoria indica que o deslocamento em relação ao topo da pilha de argumentos é zero. A função `marshaling()` é marcada como ponto de monitorização e o seu alvo é o resultado (`monitoring_re`) .

Conforme ilustrado, é muito simples e fácil instrumentar a aplicação. É possível instrumentar a aplicação uma única vez para uma série de experimentos. Pode-se marcar vários pontos de monitorização e/ou injeção e diferenciar os experimentos através de diferentes arquivos de falhas. Os pontos de injeção que não estiverem envolvidos com as falhas a serem injetadas, serão executados normalmente sem corrupção de valores. No entanto, essa facilidade deve ser usada com moderação, porque caso a aplicação tenha

[13]Este trecho de código fonte foi extraído da aplicação *Pilha_Robusta* [Pra98].

muitos objetos reflexivos e muitas mensagens reflexivas o desempenho da aplicação pode ser comprometido.

Antes da Instrumentação

```
class Stack {
protected:
 char op;
 int item;
 enum stackOutCome outcome;
public:
 Stack() {}
 ~Stack() ;
 void Atualiza(int i,char opl);
 virtual int pop(){};
 virtual void push(int item){};
 char* marshalling();
 void unmarshalling(char* buff);
 virtual void print(){};
 void Message(const char* text);
};

main (void){
 Stack obj;

 ....
}
```

Depois da Instrumentação

```
# include "scheduler.h" ①

1. class Stack {
2. protected:
3.    char op;
4.    int item;
5.    enum stackOutCome outcome;
6.    public:
7.    Stack() {}
8.    ~Stack() ;
9.    void Atualiza(int i,char opl);
10.   virtual int pop(){};
11.//MOP reflect (injection_1 ); ②
12.   virtual void push(int item){};
13.//MOP reflect (monitoring_re); ②
14.   char* marshalling();
15.public:
16.   void unmarshalling(char* buff);
17.   virtual void print(){};
18.   void Message(const char* text);
19.};

20.//MOP reflect class Stack: SchedulerMetaObj; ③

21.main (void){
22.refl_Stack obj; ④
23.}
```

Figura 5.4 Inclusão de diretivas na preparação da aplicação alvo.

Terminada a etapa de instrumentação a aplicação deve ser pré-processada usando o OCC (pré-processador OpenC++) e depois compilada e ligada com a biblioteca de metaobjetos. A Figura 5.5 ilustra o processo de preparação da aplicação alvo.

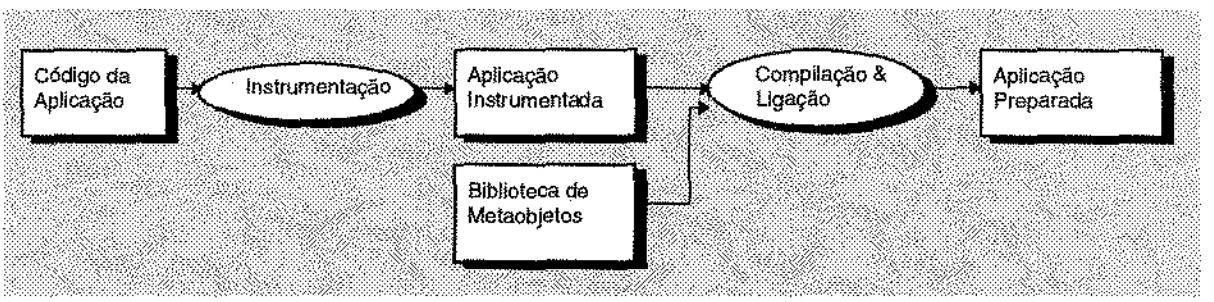


Figura 5.5 O Processo de preparação da aplicação alvo.

O passo seguinte é a geração das falhas, no qual devem ser geradas falhas que envolvam as classes marcadas como reflexivas.

Geração de Falhas

A geração de falhas pode ser feita manualmente pelo usuário, utilizando editores de texto, ou automaticamente através do *gerador de falhas* FIRE. A Figura 5.6 apresenta um exemplo do processo de geração de falhas na FIRE.

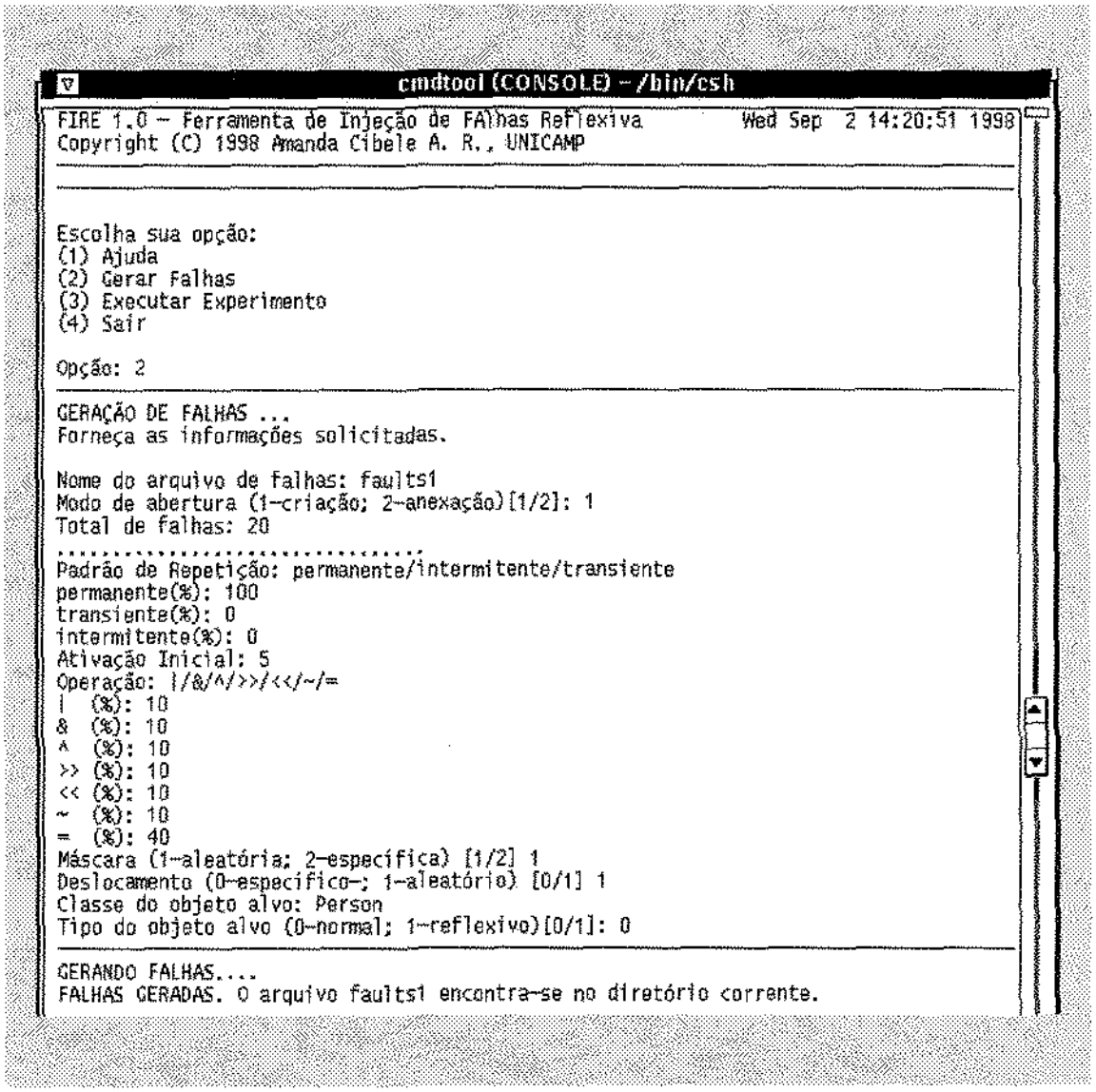


Figura 5.6 Um exemplo do processo de geração de falhas.

O gerador de falhas solicita as seguintes informações: o nome do arquivo de falhas, a quantidade de falhas a ser gerada e os atributos das falhas. Quanto aos atributos o usuário pode solicitar um valor específico ou aleatório. Um exemplo de arquivo de falhas é apresentado no Apêndice B.

5.3.2 Fase de execução

Terminada a fase de preparação o usuário pode iniciar a execução de experimentos. O *controlador de experimentos* permite a execução automática de experimentos.

No início da execução o usuário é solicitado a informar:

- o nome da aplicação alvo (preparada anteriormente),
- o nome do arquivo de falhas (gerado anteriormente) e,
- o nome do arquivo de histórico (a ser gerado).

A Figura 5.7 apresenta a tela de definição de experimentos da FIRE.

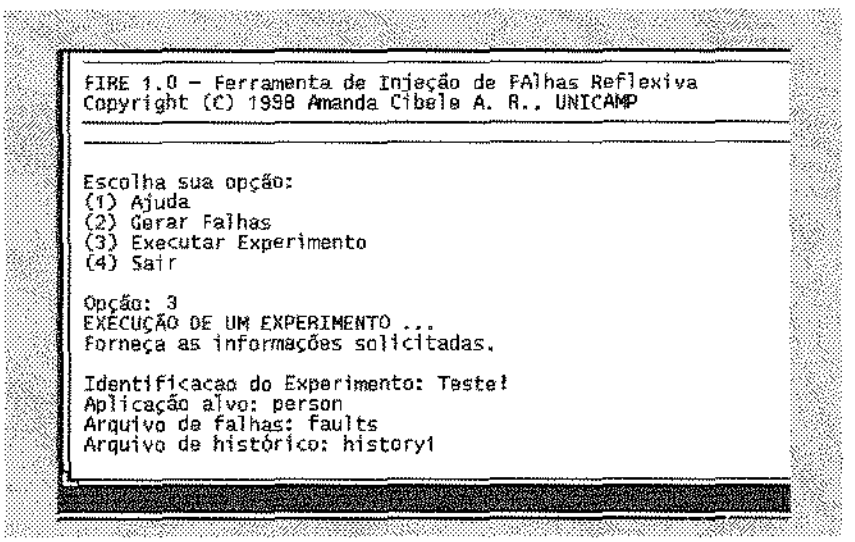


Figura 5.7 Definição de um experimento.

Em posse das informações fornecidas pelo usuário, o *controlador* :

1. lê uma instância de falha no arquivo de falhas e a transfere para um *buffer*;
2. inicia o processo da aplicação alvo e fica aguardando o seu término. A aplicação é executada, a falha em questão é injetada, os *sensores* coletam as informações e no final

da execução o controle volta para o *controlador* de experimentos. O *controlador* registra a condição de término do processo da aplicação.

Esse processo é repetido para cada falha no arquivo de falhas, e no final do experimento os dados de monitorização são armazenados no arquivo de histórico para posterior análise. Um exemplo de um possível arquivo de histórico é apresentado no Apêndice C.

A Figura 5.8 ilustra as informações sobre o processo de injeção de falhas oferecidas pela FIRE durante a execução de um experimento. FIRE indica o número da falha que está sendo injetada, exibe a saída da aplicação e no final do experimento indica onde encontrar os dados coletados.

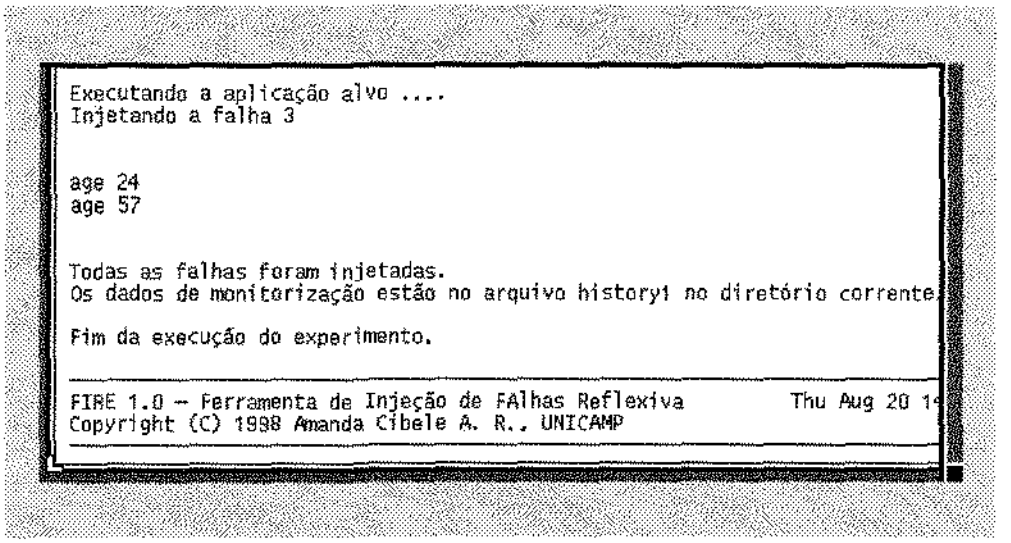


Figura 5.8 A execução de um experimento.

É interessante acompanhar o que acontece nas execuções da aplicação alvo durante a realização de um experimento. Quando a aplicação é iniciada, a *falha* no *buffer* é lida antes mesmo da instanciação de objetos da aplicação e de metaobjetos. A instância de falha lida é global a todos os *escalonadores*. Na medida em que os objetos reflexivos são instanciados, os *escalonadores* são instanciados e associados a eles. Quando um *escalonador* é instanciado ele verifica se o objeto reflexivo associado a ele está envolvido na injeção da falha a ser injetada. Em caso afirmativo ele "liga" o seu *injetor*. Os *sensores* estão sempre ligados, a menos que o usuário deseje desligá-los. Durante a execução, ocorre injeção e monitorização. *Escalonadores* são destruídos à medida que os objetos reflexivos são destruídos.

Considerando um exemplo de aplicação alvo cujas classes alvo são as classes X e Z. Supondo que em X há dois pontos de injeção e em Z há um ponto de injeção e um ponto de monitorização. A Figura 5.9 ilustra uma possível situação de execução.

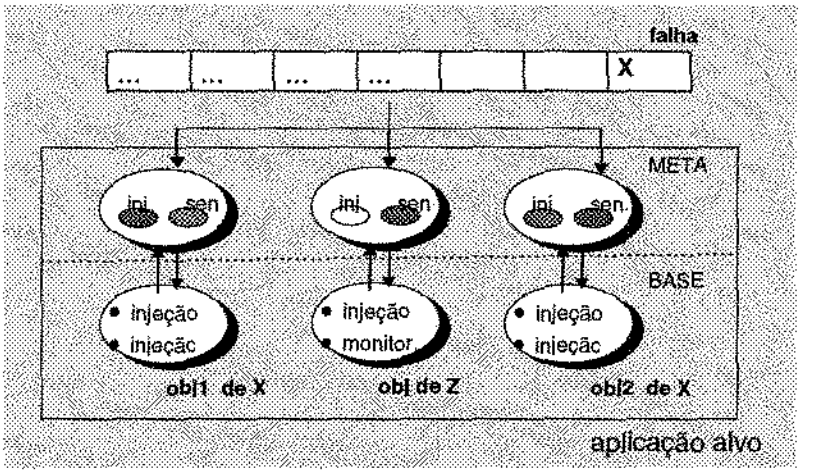


Figura 5.9 Representação de uma possível execução de uma aplicação alvo.

Todos os *sensores* na figura estão ligados (em cinza). Os *injetores* dos *escalonadores* associados aos objetos da classe X estão ligados e o *injetor* do *escalonador* do objeto da classe Z está desligado (em branco) porque a falha a ser injetada possui como `classe_alvo` a classe X. Assim, apenas os objetos da classe X sofrerão injeção de falhas e o objeto da classe Z será apenas monitorado, apesar de possuir um ponto de injeção. Se em uma próxima execução a classe alvo da instância de falha for a classe Z, apenas o *injetor* associado ao objeto da classe Z será ligado.

5.4 Resumo

Este capítulo apresentou uma ferramenta de injeção de falhas por software baseada em uma arquitetura reflexiva, denominada FIRE. FIRE usa as características do protocolo de metaobjetos da linguagem OpenC++1.2 para injetar falhas por software, e monitorar a ativação das falhas injetadas e o impacto causado no comportamento da aplicação alvo. A interferência na estrutura da aplicação alvo é quase nula e *overhead* introduzido pela reflexão computacional não perturba tanto o comportamento da aplicação alvo como mecanismos baseados na execução no modo traço, por exemplo. No entanto, um estudo comparativo deve ser feito para verificar a eficiência do uso de reflexão computacional comparado a outras técnicas, como inserção de *traps*.

Os componentes da ferramenta são: uma *biblioteca de metaobjetos* e um *controlador de experimentos*. A *biblioteca* é anexada à *aplicação alvo*, para permitir a injeção de falhas e a

monitorização, e o *controlador* é responsável por ler as *falhas* a serem injetadas, executar a aplicação, e armazenar os *dados coletados* durante os experimentos.

Um novo modelo de falhas foi proposto. Nesse modelo os componentes da aplicação alvo são os objetos, os quais podem ser afetados interna ou externamente. Tais objetos trocam mensagens para realizar o propósito da aplicação. É através da interceptação de mensagens, que FIRE permite a ativação de *injetores* e *sensores* durante a chamada de métodos, e manipulação de atributos.

Em FIRE as falhas são injetadas corrompendo o dados das mensagens trocadas entre objetos reflexivos da aplicação alvo. Através da intercepção as mensagens são capturadas e alteradas no meta-nível. Um objeto cliente que solicita um serviço a um objeto reflexivo, não toma conhecimento da ação de metaobjetos e, portanto, confia no serviço prestado. Dessa forma, falhas injetadas em um objeto podem propagar-se para outros. Para, acompanhar a propagação de falhas, FIRE contém *sensores* que permitem acompanhar se determinadas mensagens foram trocadas durante a execução da aplicação e qual o valor de dados trocados em determinadas mensagens.

A utilização da ferramenta é muito simples e consiste em: (1) instrumentar a aplicação, indicando quais objetos, atributos e métodos serão reflexivos, ou seja, sofrerão injeção e/ou monitorização; (2) compilar a aplicação instrumentada; (3) anexar a biblioteca de metaobjetos durante a ligação; (4) gerar as falhas, e (5) executar os experimentos.

Embora OpenC++1.2 ofereça capacidades limitadas de reflexão, ela foi útil para mostrar a viabilidade da implementação das características de injeção e monitorização no meta-nível. O uso da abordagem metaobjeto permite implementar tais características independentemente da funcionalidade da aplicação. Comparando-se a outras abordagens de injeção de falhas por software duas outras vantagens podem ser destacadas. Primeiro, esta abordagem facilita a introdução da instrumentação no código da aplicação: não são inseridas explicitamente instruções de *trap* no código da aplicação e não é necessário executar a aplicação em modo traço. Segundo, esta abordagem simplifica o uso, pois não é preciso executar a aplicação em modo privilegiado, nem é necessário utilizar as características especiais do hardware as quais não são disponíveis para a maioria dos usuários.

O próximo capítulo aborda os aspectos de implementação da FIRE.

Capítulo 6

Aspectos de Implementação

Este capítulo apresenta os aspectos de implementação da ferramenta e discute as dificuldades encontradas.

6.1 Introdução

O código fonte da FIRE corresponde a dois módulos: a biblioteca de metaobjetos e o controlador de experimentos. A biblioteca foi implementada utilizando C++ e o OpenC++1.2, e o controlador foi totalmente implementado em C++. A biblioteca controla a injeção de falhas durante a execução da aplicação alvo e deve ser *linkada* ao código da aplicação alvo para gerar o código executável com as características reflexivas. O controlador é o processo pai dos processos da aplicação alvo e controla os experimentos.

Para facilitar o entendimento da implementação da FIRE são apresentados os diagramas dos modelos de objetos da biblioteca e do controlador. A notação utilizada foi a Técnica de Modelagem de Objetos (TMO) [RBP+91].

A implementação da ferramenta foi relativamente simples. As dificuldades foram encontradas na fase de testes e dizem respeito, principalmente, a um problema de compatibilidade do pré-processador OpenC++1.2 com algumas bibliotecas incluídas nas aplicações testadas.

A organização do capítulo é a seguinte: a Seção 6.2 descreve as classes da biblioteca, a Seção 6.3 descreve o controlador, a Seção 6.4 discute as dificuldades encontradas e a Seção 6.5 resume e conclui este capítulo.

6.2 A biblioteca de Metaobjetos

A biblioteca de metaobjetos provê as funcionalidades de injeção e monitorização. O modelo de objetos da biblioteca é apresentado na Figura 6.1.

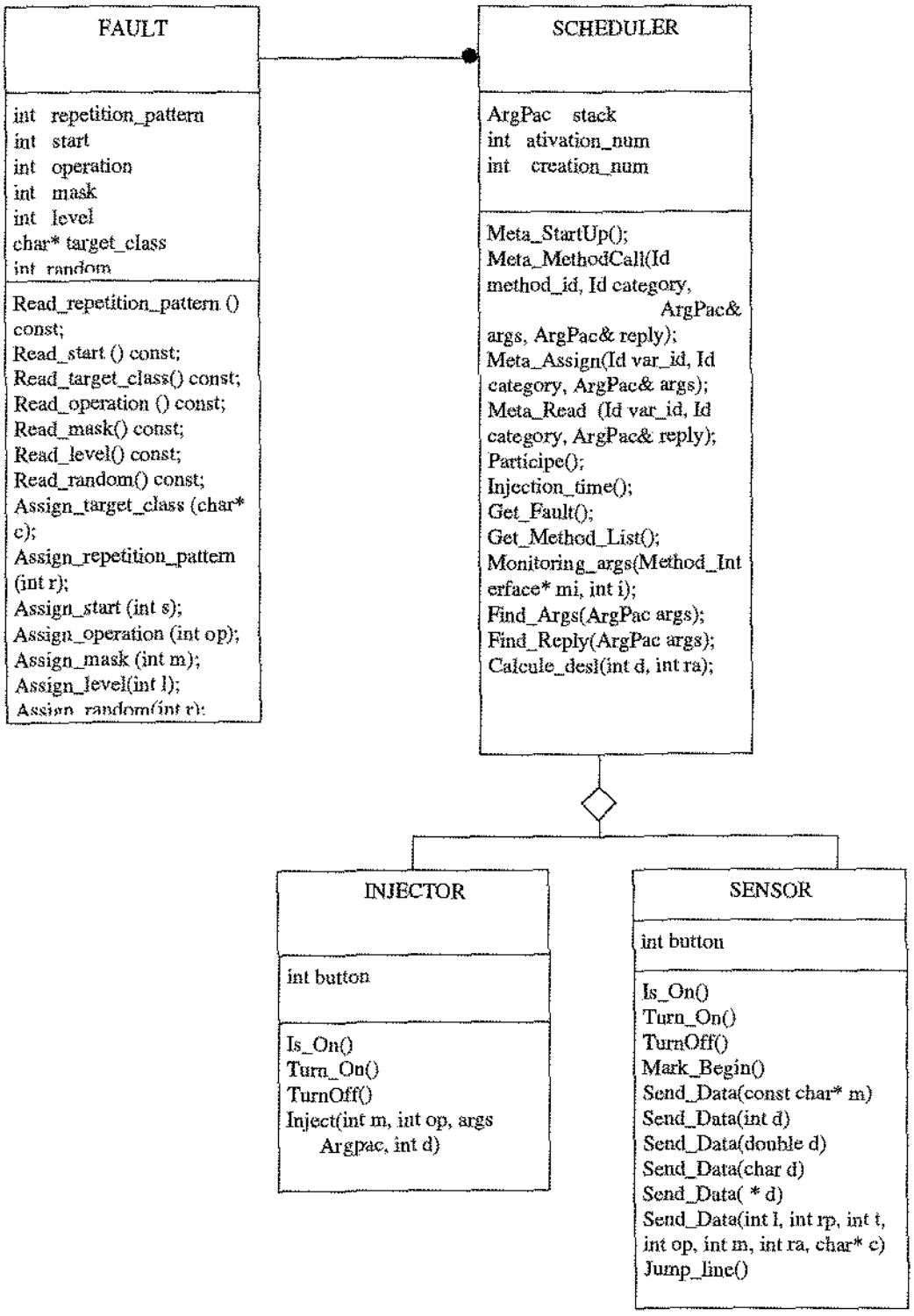


Figura 6.1 Diagrama de objetos da biblioteca.

6.2.1 Classe *Fault*

A classe *Fault* define o formato das falhas. A classe possui os seguintes atributos:

- `int repetition_pattern`: indica o padrão de repetição das falhas. Se o valor desse atributo for "1" indica que a falha é permanente; se "0" indica que a falha é transiente e, se "N> 1" indica que a falha é intermitente com período N.
- `int start`: corresponde a ativação inicial da falha.
- `char* target_class`: indica a classe alvo.
- `int operation`: corresponde a operação que deve ser usada para corromper o alvo. Os números de 1-7 indicam as 7 possíveis operações (atribuição e operações lógicas bit-a-bit - E, OU inclusivo, OU exclusivo, deslocamento a esquerda, deslocamento a direita e complemento).
- `int mask`: indica o valor a ser usado para corromper o alvo.
- `int level`: indica se o nível alvo da aplicação é o imediatamente inferior ou não.
- `int random`: se 1 a deslocamento para o calculo do endereço de injeção é aleatório; se 0 é específico.

A interface dessa classe é formada por métodos para ler e atualizar os valores dos atributos de uma falha.

6.2.2 Classe *Sensor*

Essa classe define a estrutura e o comportamento de *sensores*. *Sensores* possuem um atributo `button`, que indica se o *sensor* está ligado ou não. Todos os objetos sensores estão associados a um arquivo de histórico, ou seja, o arquivo é definido como um atributo estático da classe *Sensor*. A interface dos *sensores* é constituída por:

- métodos para ligar/desligar o *sensor* e um método para verificar se o *sensor* está ligado.

```
void Sensor::Turn_On();
void Sensor::Turn_Off();
int  Sensor::Is_On();
```

- métodos para controlar o formato do arquivo de histórico.

```
void Sensor::Mark_Begin();
void Sensor::Jump_line();
```

- métodos coletar os dados de monitorização. O nome da função `Send_Data()` foi sobrecarregado para tipos atômicos (char, char*, int, double e void*) e para enviar os dados de uma instância de falha que foi injetada.

6.2.3 Classe *Injector*

Essa classe define a estrutura e o comportamento de *injetores*. *Injetores* possuem um atributo `button`, que indica se o *injetor* está ligado ou não. A interface dessa classe é constituída por:

- métodos para ligar/desligar *injetores* e um método para verificar se o *injetor* está ligado.

```
void Injector::Turn_On();

void Injector::Turn_Off();

int  Injector::Is_On();
```

- método para injetar falhas. Este método recebe como parâmetros a máscara, a operação, o deslocamento e uma referência para um objeto da classe `ArgPac`, o qual contém os dados do objeto alvo. O método `Inject` é apresentado a seguir:

```
void Injector::Inject(int mask, int op, ArgPac& args, int d){
   int *ip,  target;
   ip = (int*)args.GetPtrToLastArgument(); // topo da pilha;
   ip = ip + d;  // calcula o endereço efetivo;
   target = *ip; // pega o conteúdo do endereço alvo;
   switch (op){ // corrompe o conteúdo conforme a operação
   case 1:
      target = target & mask;
      break;
   case 2:
      target = target | mask;
      break;
   case 3:
      target = target ^ mask;
      break;
   case 4:
      target = target >> mask;
      break;
   case 5:
      target = target << mask;
      break;
   case 6:
      target = ~target;
      break;
   case 7:
      target = mask;
   default:break;
   }
   *ip = target;  // corrompe o conteúdo endereço alvo.
```

```
}
```

6.2.4 Classe *SchedulerMetaObj*

A classe *SchedulerMetaObj*, na verdade, é uma meta-classe, ou seja, é a classe que define os metaobjetos que são associados aos objetos reflexivos da aplicação. Embora as demais classes que pertencem à biblioteca, não sejam meta-classes, os objetos instanciados a partir delas pertencem exclusivamente ao meta-nível. Ou seja, os objetos da aplicação desconhecem objetos como *injetores* ou *sensores*.

Como explicado no capítulo 4, a definição de meta-classes em OpenC++ 1.2 é realizada por meio de herança. Toda meta-classe herda e redefine alguns meta-métodos da classe padrão *MetaObj*. É através da redefinição de alguns desses meta-métodos que o programador altera o comportamento de objetos reflexivos.

Meta-métodos possuem parâmetros relativos a informações do objeto base. Esses parâmetros *são method_id, var_id, category, args* e *reply* e especificam respectivamente, o identificador do método chamado, o identificador do atributo lido ou atualizado, o nome da categoria do método chamado ou do atributo lido ou atualizado, uma referência a um objeto ArgPac, que armazena o valor dos argumentos do método chamado ou o valor a ser atribuído a uma variável, e uma referência a um objeto ArgPac que armazena valor de retorno do método chamado ou o valor lido de uma variável.

Para implementar a computação adicional de injeção de falhas e monitorização a meta-classe *SchedulerMetaObj* redefine alguns dos meta-métodos da classe padrão *MetaObj*.

- `void MetaMethodCall(Id method_id, Id category, ArgPac& args, ArgPac& reply)`
  Intercepta toda chamada de método reflexivo. As ações descritas neste método são executadas quando um método reflexivo é chamado.
- `void MetaAssign (Id var_id, Id category, ArgPac& args)`
  Implementa a atualização de atributos e é executado antes da atualização de um atributo reflexivo.
- `void MetaRead (Id var_id, Id category, ArgPac& reply)`
  Implementa a leitura de atributos é executado antes da leitura de um atributo reflexivo.

Estes meta-métodos foram herdados e redefinidos na meta-classe *SchedulerMetaObj* e implementam a mesma funcionalidade: verificar o tipo de ação a ser tomada (injeção e/ou monitorização ou execução normal) com base no nome_de_categoria; e invocar o *sensor* e/ou *injetor*. A diferença entre eles é que o primeiro é usado para injetar e/ou monitorizar métodos, o segundo a atualização de atributos e o terceiro a leitura de atributos.

- `void Meta_StartUp()`
  É chamado depois que um objeto reflexivo e seu metaobjeto são criados e associados.

Este meta-método implementa o procedimento realizado no momento da criação do *Escalonador* e do objeto base criado. A função dele é verificar se o objeto base criado deve ou não sofrer injeção de falhas e em função disto ativar ou não o *injetor*.

Além das características herdadas da classe *MetaObj*, a classe *SchedulerMetaObj* possui os seguintes atributos:

- `int creation_num, Fault*fault, Method_List* list` : todos esses atributos são estáticos, ou seja, globais para todos os *escalonadores*. O atributo `creation_num` indica quantos *escalonadores* foram criados (para monitorização), `fault` é um ponteiro para a falha a ser injetada e `list` é uma lista de assinatura dos métodos que serão monitorados (o escalonador usa essa lista para informar aos *sensores* o tipo dos dados que eles irão coletar).
- `Sensor* sensor` e `Injector* injector`: são ponteiros para um objeto *sensor* e para um objeto *injetor*, respectivamente.
- `ArgPac stack`: uma pilha auxiliar para copiar os dados a serem monitorados.

Outros métodos da classe *SchedulerMetObj* são os seguintes:

- `Get_Fault()` e `Get_Method_List()`: a função desses métodos é ler a falha a ser injetada e recuperar a lista de assinatura dos métodos, respectivamente. Eles são executados uma única vez antes da criação do primeiro *escalonador*.
- `void Monitoring_args(Method_Interface* mi, int i)`: informa ao *sensor* qual é o tipo do dado a ser monitorado baseado na interface do método alvo.
- `ArgPac* Find_Args(ArgPac args)` e `ArgPac* Find_Reply(ArgPac args)`: são usados para encontrar a referência para a pilha de argumentos de um objeto do nível base na pilha de argumentos de um metaobjeto. É usado no caso de aplicações reflexivas.
- `Calcule_desl(int category, int a)`: calcula o deslocamento para injeção de falhas conforme a categoria e o tipo de falha (aleatória ou específica).
- `int Participe()`: verifica se o objeto do nível base está envolvido no processo de injeção de falhas.
- `int Injection_time()`: verifica se em determinada ativação do injetor deve ocorrer injeção de falhas.

6.3 O controlador

O controlador utiliza as chamadas de sistema *fork()*, *exec(target)* e *wait(&status)* [Ste92] para controlar a execução da aplicação alvo. A chamada *fork()* cria um novo processo denominado processo filho, o qual é uma cópia exata do processo que o criou, o processo pai. A chamada *exec(target)* é usada no processo filho para substituí-lo pelo programa identificado por *target* e iniciar sua execução. A chamada *wait(&status)* é utilizada no processo pai para fazer com que ele aguarde o término da execução de um de seus processos filhos e para coletar a condição de término desse processo (armazenada em *status*).

As seguintes classes compõem o controlador: *FaultManager*, *Interface* e *Monitor*.

6.3.1 Classe *FaultManager*

Essa classe descreve o *gerente de falhas*. Um *gerente de falhas* possui duas funções:

- passar a falha a ser injetada para um *buffer* (para que o *escalonador* possa realizar a leitura durante a execução da aplicação alvo). Para isso ele utiliza os seguintes métodos:

```
void OpenFaults(char* f);

void ReadFault();

void WriteFault();

int  EndFaults();

void CloseFaults();
```

- e gerar arquivos de falhas a serem injetadas. Para isso ele usa os seguintes métodos:

```
void CreatFaultsFile(char* name, int modo);

void WriteFault(int l, int r, int s, int o, int m, char* c);

void CloseFaults();
```

6.3.2 Classe *Monitor*

Essa classe descreve as funções do *monitor* as quais incluem:

- criar o arquivo de histórico do experimento: `void CreateHistory(char* identification);`
- coletar os dados no final do experimento: `void ColectData(char* file_name);`

- gravar informações passadas pelo *controlador* sobre as condições de término da aplicação: `void WriteInfo(char* info);`

6.3.3 Classe *Interface*

Descreve funções de interface simples. Correspondem às funções para exibir e ler dados.

```
void DisplayMessage(char* message);
void DisplayQuestion(char* message);
void DisplayCabecalho();
void DisplayInt(int i);
void DisplayMenu();
void DisplayTraco();
char*  GetChar();
int    GetInt();
double GetDouble();
```

A Figura 6.2 apresenta o diagrama do modelo de objetos do *controlador*.

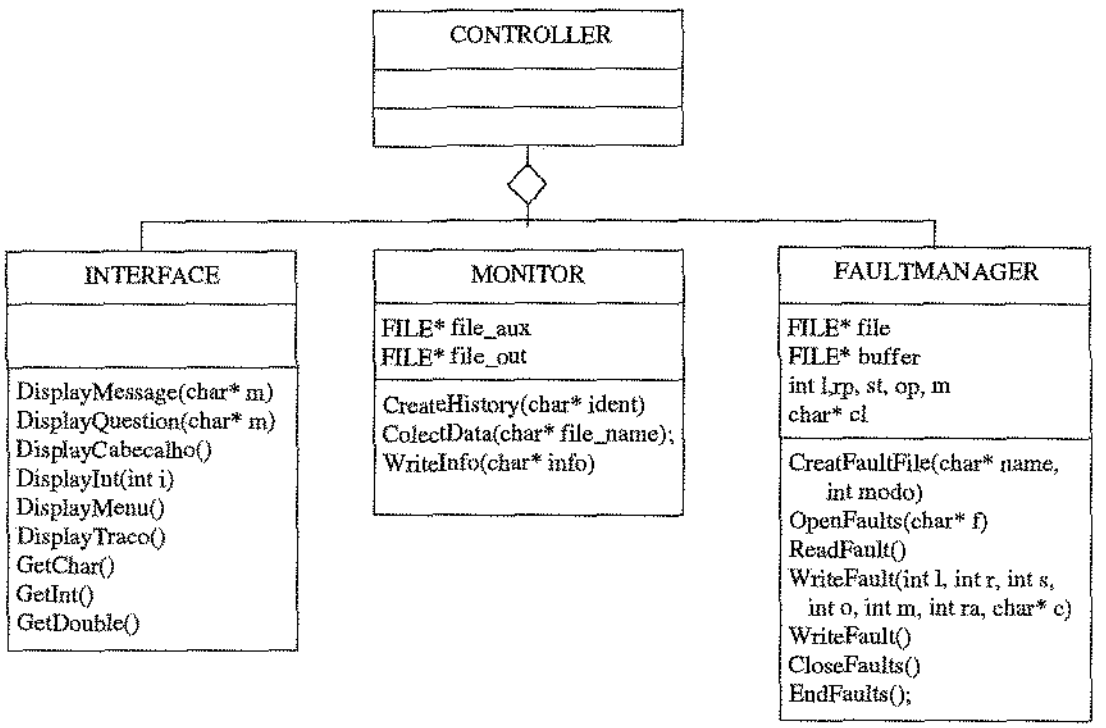


Figura 6.2 Diagrama do modelo de objetos do *controlador*.

6.4 Dificuldades Encontradas

As dificuldades encontradas no desenvolvimento da ferramenta dizem respeito às limitações da linguagem OpenC++1.2.

O primeiro obstáculo encontrado foi o fato do protocolo de OpenC++1.2 não permitir que um objeto reflexivo seja associado a mais de um objeto. Como o objeto reflexivo deve ser injetado e monitorizado ele deveria ser associado a pelo menos dois metaobjetos: um *sensor* e um *injetor*. A solução foi encontrada na fase de projeto, através da incorporação do metaobjeto *escalonador* que agrega um *sensor* e um *injetor*. Embora *sensores* e *injetores* não sejam metaobjetos, eles pertencem ao meta-nível e não alteram a estrutura da aplicação alvo.

O segundo obstáculo, já na fase de testes, foram os problemas do compilador da OpenC++1.2, o OCC, com algumas bibliotecas. As primeiras tentativas de utilização da ferramenta foram com uma aplicação que utilizava o Arjuna[14] [Bed97]. Após a instrumentação, quando o OCC era usado para compilar a aplicação alvo, ele revelava erros de compilação nos arquivos do Arjuna incluídos na aplicação. A princípio, a razão apontada para esse problema foi uma possível incompatibilidade entre OpenC++1.2 e Arjuna. No entanto, quando outras aplicações que não utilizavam o Arjuna foram utilizadas como alvo, o problema ainda permanecia com outras bibliotecas, como iostream.h[15].

Isso acontecia, porque o OCC além de gerar as características necessárias para a reflexão, faz o pre-processamento do código fonte e o compila. Por algum problema, causado durante o pre-processamento, erros de sintaxes era apontados na compilação.

A solução encontrada foi retirar os *includes* problemáticos antes de usar o OCC (ver Figura 6.3). O que acontece nesse caso é o seguinte: o OCC faz o pre-processamento da aplicação, gera o código fonte necessário para reflexão e obviamente durante a compilação aponta erros devido a falta dos *includes*. Para gerar o executável, os *includes* são novamente introduzidos no fonte da aplicação e o compilador C++ é utilizado para terminar a preparação da aplicação alvo.

---

[14] Arjuna é ambiente de programação orientado a objetos, totalmente implementado em C++, que fornce um conjunto de ferramentas para a construção de aplicações distribuídas tolerantes a falhas de hardware, estruturadas como ações atômicas operando sobre objetos persistentes.

[15] iostream.h é uma biblioteca C++ para entrada e saída de dados.

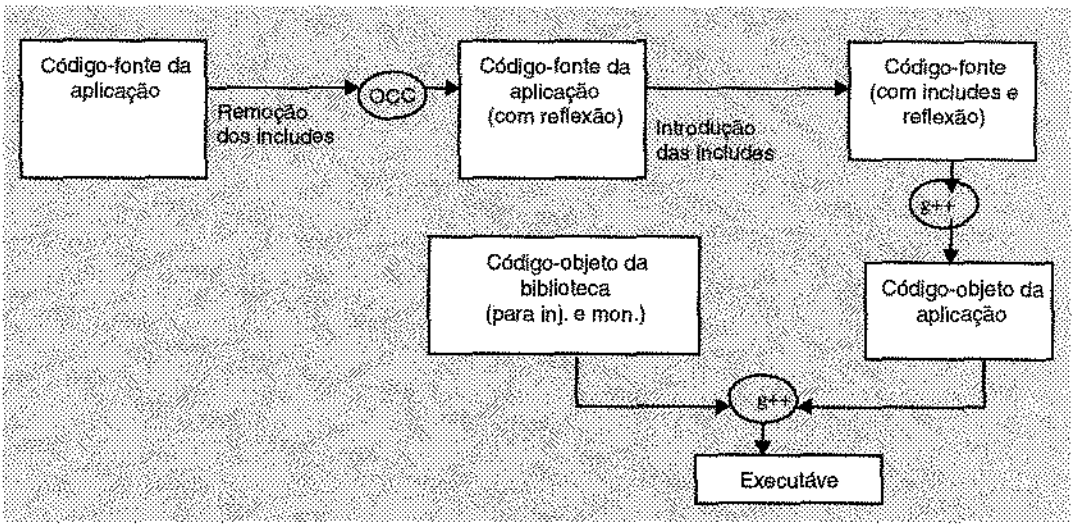


Figura 6.3 Solução para o problema com o OCC.

No mais, a implementação da ferramenta foi relativamente fácil, pois a interface de OpenC++1.2 é bem definida e explicada no seu manual de referência.

6.5 Resumo

Este capítulo apresentou os diagramas dos modelos de objetos da ferramenta e descreveu a estrutura e a funcionalidade de suas classes. A ferramenta foi implementada em C++ e OpenC++1.2.

Em geral, a adoção do modelo de objetos contribuiu para reduzir a complexidade do desenvolvimento da ferramenta. Em particular, a adoção de uma abordagem reflexiva contribuiu para a transparência e a independência das características de injeção e monitorização em relação funcionalidade de aplicações alvos.

Os primeiros testes com a ferramenta revelaram algumas incompatibilidades do pré-processador OpenC++1.2 com algumas bibliotecas. A solução proposta para esses casos foi utilizar o OpenC++1.2 apenas para gerar as características reflexivas e usar o compilador C++ para gerar o código executável.

A aplicabilidade da FIRE foi demonstrada em alguns experimentos de injeção de falhas. O próximo capítulo descreve esses experimentos e seus resultados.

Capítulo 7

Resultados Experimentais

O objetivo deste capítulo é descrever os primeiros experimentos realizados com a FIRE e apresentar os resultados obtidos.

7.1 Introdução

Foram realizados vários experimentos de injeção de falhas para validar a FIRE. O objetivo desses experimentos não foi validar as propriedades de segurança de funcionamento das aplicações usadas nos testes, mas verificar a aplicabilidade da ferramenta. Os primeiros testes foram feitos com uma aplicação tolerante a falhas distribuída. Nesse caso FIRE foi utilizada com finalidade de verificação. Ou seja, o objetivo era estudar como a ferramenta pode auxiliar na eliminação de falhas de projeto/implementação de aplicações. O passo seguinte foi verificar a aplicabilidade da ferramenta em aplicações reflexivas. Tais experimentos foram realizados usando programas demonstrativos simples do OpenC++1.2. Foram realizados também alguns experimentos para avaliar o desempenho do mecanismo de reflexão computacional na injeção de falhas. Esses testes foram realizados porque uma das questões mais freqüentes relacionadas à reflexão é justamente quando ao seu desempenho. Este capítulo segue a seguinte organização: a Seção 7.2 descreve a aplicação distribuída utilizada nos experimentos, a Seção 7.3 descreve os experimentos e apresenta os resultados obtidos e a Seção 7.4 resume e conclui o capítulo.

7.2 Aplicação alvo

A aplicação alvo escolhida, PilhaRobusta [Pra98], é uma implementação de uma pilha tolerante a falhas de software. A aplicação é simples e foi desenvolvida para validar o FOOD (Framework Orientado a Objetos Distribuídos para Tolerância a Falhas) [Pra98], proposto como dissertação de mestrado no Instituto de Computação da UNICAMP. Mais especificamente, o framework dá apoio para a construção de componentes ideais com diversidade de projeto e oferece serviços de tolerância a falhas de software para um ambiente distribuído, provendo os mecanismos de blocos de recuperação e N-versões [RC97]. Uma rápida descrição desses dois mecanismos é apresentada em 7.2.1 e 7.2.2.

FOOD possui as seguintes características:

- foi totalmente implementado em C++;
- utiliza o ambiente de programação distribuída Arjuna para a provisão de restauração de estados através do uso de ações atômicas;
- utiliza "sockets" TCP/IP para a comunicação dos objetos distribuídos e,
- utiliza mecanismos de tratamento de exceções para representar e tratar exceções geradas durante a execução de aplicações.

PilhaRobusta é um exemplo de utilização desse framework e suas características são apresentadas em 7.2.3.

7.2.1 Blocos de Recuperação

Como mencionado no Capítulo 2, para tolerar falhas de software e obter-se uma operação contínua do sistema, redundância de software é necessária, ou seja componentes com diversidade de projeto (i.e., variantes) são usualmente desenvolvidos. Blocos de recuperação constituem um tipo de redundância de software, onde dentre os componentes redundantes apenas um componente é ativo por vez e se um erro é detectado no componente ativo, então o próximo componente disponível passa a ser o ativo.

O esquema de blocos de recuperação realiza a detecção de erros através de um teste de aceitação aplicado seqüencialmente aos resultados da execução de variantes: se o teste de aceitação falhar ao passar pela primeira variante, o estado do sistema é restaurado utilizando-se técnicas de recuperação de erros por retrocesso, e a segunda variante é executada; esse processo continua até que todas as variantes se esgotem ou algum resultado passe pelo teste de aceitação. Se todas as variantes falharem uma exceção deve ser sinalizada. A sintaxe do bloco de recuperação é a seguinte:

```
  assegure  <teste de aceitação>
        em  <variante_1>
  senão em  <variante_2>
  senão em  <variante_N>
senão erro
```

Voas [VM98] apresenta dois problemas com o mecanismo de blocos de recuperação. O primeiro deles é a natureza seqüencial da execução de variantes. Por exemplo, o que aconteceria se uma variante entrasse em um laço infinito e não retornasse resultado para o teste de aceitação? O outro diz respeito à confiabilidade do teste de aceitação. Se um teste de aceitação é falho, ele pode aceitar resultados errados ou recusar corretos.

7.2.2 N-versões

Em N-versões, versões diferentes do programa executam simultaneamente e um mecanismo de votação (seletor, votador ou árbitro) determina um único resultado. O resultado aceito como correto (geralmente o produzido pela maioria) é então difundido para o resto do sistema. Caso o seletor não seja capaz de determinar o resultado, então uma exceção deve ser sinalizada. O funcionamento dessa técnica é ilustrado na Figura 7.1.

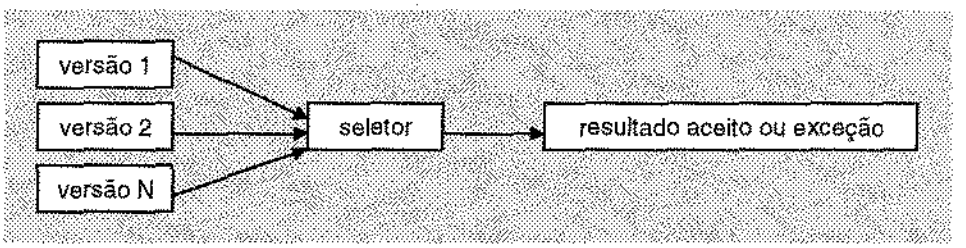


Figura 7.1 Funcionamento do N-versões.

7.2.3 PilhaRobusta

A aplicação alvo utiliza três variantes de pilha com diversidade de projeto: uma implementada através de lista encadeada, outra através de vetor e a última através de lista duplamente ligada. Assim, a execução da aplicação se dá através de 4 objetos (um cliente *RobustStack* e três servidores: *ListServer*, *ArrayServer* e *DoubleServer*) distribuídos em máquinas diferentes. A comunicação é feita através de sockets TCP/IP.

A classe PilhaRobusta oferece serviços de *empilha()* e *desempilha()* e quando é feita uma solicitação de um desses serviços, um MTF é invocado (blocos de recuperação ou N-versões, conforme opção do usuário). Nesses mecanismos, a mensagem é passada para a(s) variante(s). Cada variante executa o serviço solicitado e faz um simples teste de invariante sob o resultado obtido: verifica se o elemento empilhado ou desempilhado corresponde ao índice da pilha, por exemplo, se o 10° elemento da pilha é o número 10. Se a invariante da pilha for satisfeita, a variante retorna para o cliente *RobustStack* um valor de *status* "1" indicando que a operação foi realizada com sucesso, caso contrário envia "0".

Em blocos de recuperação o teste de aceitação aplicado sob cada variante invocada é a verificação do retorno da variante. Caso o retorno seja "0", o estado do objeto é recuperado (por um mecanismo de recuperação por retrocesso, fornecido pelo Arjuna) e a próxima variante é invocada. No caso de N-versões o seletor escolhe o resultado da maioria, se a maioria retorna um resultado favorável, então a operação solicitada no cliente é considerada executada com sucesso. Em ambos os casos, se não foi possível executar a operação solicitada, a aplicação não é interrompida e uma mensagem de erro é enviada.

A aplicação utilizada nos experimentos corresponde ao armazenamento de 100 elementos em uma pilha robusta. O trecho de código que implementa a solicitação de armazenamento no cliente é o seguinte:

```
for (i=1; i=<100; i++)
    rs.empilha(i);
```

7.3 Experimentos

O ambiente dos experimentos é formado por um conjunto de estações sob plataforma UNIX conectadas através do protocolo de comunicação TCP/IP. Foi realizada um série de testes para descobrir qual a melhor forma para validar a aplicação alvo, uma vez que a aplicação usa uma arquitetura distribuída e a ferramenta não. A Subseção 7.3.1 apresenta esses primeiros testes. Passada essa fase inicial de testes, foram organizados uma série de experimentos para validar os MTF's da aplicação alvo, a Seção 7.3.2 descreve esses experimentos. A Subseção 7.3.3 apresenta os resultados obtidos com testes de programas reflexivos simples. Tais resultados não dizem respeito à validação dos programas usados nos testes, mas à aplicabilidade da FIRE para testar aplicações reflexivas. A Subseção 7.3.4 apresenta um estudo do *overhead* introduzido pela reflexão, comparando os tempos de execução dos testes da aplicação PilhaRobusta com a FIRE e com injeção através de mutação.

7.3.1 As primeiras tentativas

Esses testes correspondem a estudos sobre quais classes, métodos e objetos da aplicação alvo poderiam sofrer injeção de falhas e/ou monitorização.

Corrompendo o elemento a ser inserido na PilhaRobusta

A primeira idéia foi injetar falhas corrompendo o argumento do método `empilha(i)` do objeto *RobustStack*. Esses primeiros testes não foram muito significativos, pois como o argumento era alterado antes de ser passado para os mecanismos e, consequentemente para as variantes, em ambos os mecanismos a operação não era realizada com sucesso em nenhuma das variantes. Quando foi usado o mecanismo blocos de recuperação, um valor corrompido era passado para a 1ª variante, a qual retornava um erro na inserção do elemento. Dessa forma, o teste de aceitação falhava, o estado do objeto era recuperado (i.e., o valor corrompido era recuperado) e passado para a próxima variante. O comportamento era o mesmo para todas as variantes. Quando foi utilizado o mecanismo N-versões, o mesmo valor corrompido era passado para as três versões, as três retornavam um resultado não favorável e o seletor informava que a operação não tinha sido executada com sucesso.

Corrompendo o resultado do teste de aceitação utilizado no método que implementa o mecanismo blocos de recuperação

Essa tentativa consistiu em corromper o resultado do teste de aceitação utilizado no mecanismo blocos de recuperação, forçando-o como "0".

- O impacto de falhas permanentes.
  Uma falha permanente no resultado do teste de aceitação implica que o resultado será alterado em todas as invocações do teste. Quando falhas permanentes foram injetadas, o resultado do teste de aceitação falhava para todas as variantes e o elemento não era empilhado.

- O impacto de falhas transientes.
  Uma falha transiente no resultado do teste de aceitação, implica que o resultado será corrompido em uma das invocações do teste. Dessa forma, a injeção desse tipo de falha deveria afetar apenas o resultado do teste sobre primeira variante. No entanto, um comportamento diferente foi observado. A injeção de falhas transientes ativadas no início da execução da aplicação (i.e., no primeiro `empilha(i))` afetavam o resultado do teste sobre a primeira variante, forçando a invocação da segunda variante. Como a falha era transiente, o resultado do teste de aceitação da segunda variante não era corrompido e a operação era realizada com sucesso (esse comportamento era esperado). No entanto, falhas transientes ativadas em um momento diferente do inicial (i.e., `empilha(i) e i ≠ 1)` propagavam-se para as demais variantes, ou seja, mesmo sem corromper o resultado, o teste de aceitação aplicado às demais variantes falhava e o elemento não era inserido.

- O impacto de falhas intermitentes
  Uma falha intermitente no resultado do teste de aceitação, implica que o resultado será corrompido a cada N (N = intervalo de injeção definido pelo usuário) invocações do teste. Ou seja, a mesma falha é injetada periodicamente. Dessa forma, a injeção desse tipo de falha deveria afetar, periodicamente, apenas o resultado da primeira variante, a menos que o intervalo definido fosse um, forçando a corrupção do resultado do teste da segunda e da terceira variante. No entanto, o resultado de todas as variantes falhavam na primeira injeção quando a ativação inicial era diferente do início da execução e em todas as próximas injeções independente da ativação inicial.

Corrompendo o elemento a ser inserido nas variantes

Os resultados obtidos corrompendo o teste de aceitação foram importantes pois revelaram um defeito no comportamento da aplicação e a possibilidade da existência de falhas de software. No entanto, a mesma técnica não pode ser utilizada para corromper o seletor do mecanismo de N-versões, pois o método que implementa o seletor possui como argumento

um array de apontadores, o que impossibilita tornar tal método reflexivo[16]. Então uma alternativa foi corromper o elemento a ser inserido em cada variante. Ou seja, nos primeiros testes com a FIRE, o processo de injeção de falhas ficava centralizado no objeto *RobustStack* e com a alternativa de injetar falhas nas variantes esse processo passou a ser distribuído. A Subseção 7.3.2 descreve os experimentos nos quais as falhas foram injetadas nas variantes.

7.3.2 Experimentos de injeção de falhas em objetos distribuídos

Os experimentos foram divididos por tipo de falhas (permanentes, transientes ou intermitentes), por objeto alvo (*ListServer*, *ArrayServer* ou *DoubleServer*) e por mecanismo (blocos de recuperação ou N-versões). Em alguns experimentos apenas uma variante foi alvo de injeção, em outros duas variantes foram alvos. Não foram considerados os experimentos envolvendo injeção de falhas nas três variantes, pois nesse caso a operação solicitada não seria realizada com sucesso. Em todos os experimentos o objeto cliente *RobustStack* foi apenas monitorado quanto ao item a ser inserido e o resultado devolvido pelas variantes.

No caso do mecanismo de blocos de recuperação, para injetar falhas na segunda variante invocada, foi preciso forçar a primeira variante a falhar sempre, assim falhas permanentes com ativação no início da execução foram injetadas na primeira variante para que a segunda pudesse ser testada. Os testes feitos anteriormente já haviam demonstrado que esse tipo de falha afetava apenas a variante alvo e não se propagava para as demais.

A Figura 7.2 ilustra a arquitetura de experimentos, nos quais tanto o *ListServer* como *ArrayServer* sofreram injeção de falhas. Cada variante alvo foi associada a um metaobjeto *escalonador*, o qual está relacionado a arquivos distintos.

No total foram realizados 12 experimentos e em cada experimento foram injetadas 20 falhas. O método considerado para injeção foi o *empilha(i)* executado nas variantes. Originalmente, a aplicação correspondia ao armazenamento de 100 elementos, mas, para simplificar a análise dos resultados ela foi modificada para empilhar apenas 10 elementos.

[16] OpenC++1.2 só oferece métodos para reificação de tipos atômicos, ou seja, se argumentos de um método, ou atributos são de um tipo não-atômico esse método e esses atributos não podem ser reflexivos. A menos que o programador da aplicação defina métodos de *marshaling* e *unmarshaling* para que o OpenC++1.2 saiba com manipular esses dados, o que não é adequado em nosso caso.

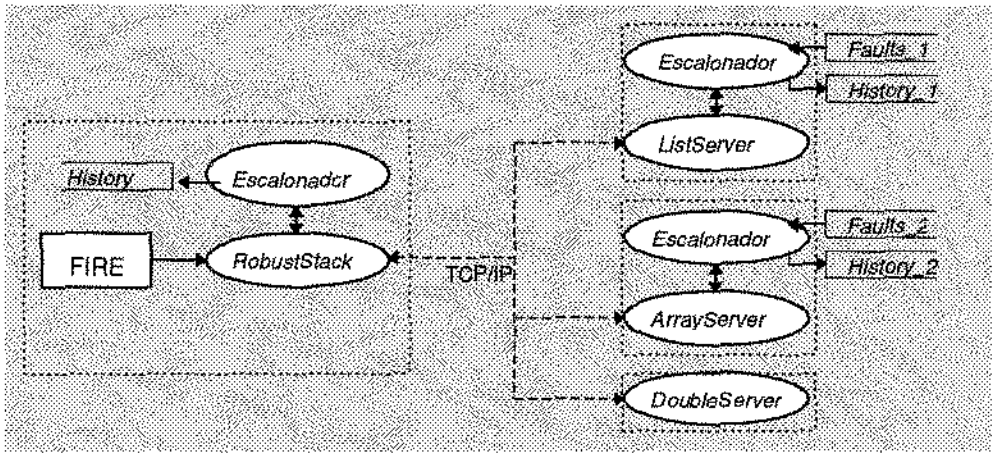


Figura 7.2 Injeção de falhas em objetos distribuídos.

Cada experimento foi resumido em uma tabela. Em cada tabela o significado das colunas é o seguinte:

A	F	E	I	II	III
#	#	#	#(s/n)	#(s/n)	#(s/n)
			#(s/n)	#(s/n)	#(s/n)
	total	total			

- **A** - ativação inicial - no caso específico dessa aplicação alvo, se a ativação inicial da falha for "*n*" então o alvo da injeção será o *empilha(n)* .
- **F** - número de falhas injetadas. O total dessa coluna indica o total de falhas injetadas no experimento.
- **E** - número de operações *empilha(i)* executadas. O total dessa coluna indica o total de operações executadas.
- **I** - número de operações nas quais apenas uma variante foi executada (para blocos de recuperação) ou número de operações nas quais apenas uma variante retornou um valor verdadeiro (para N-versões).
- **II** - número de operações nas quais duas variantes foram executadas (para blocos de recuperação) ou número de operações nas quais duas variantes retornaram um valor verdadeiro (para N-versões).
- **III** - número de operações nas quais três variantes foram executadas (para blocos de recuperação) ou número de operações nas quais três variantes retornaram um valor verdadeiro (para N-versões).

As três últimas colunas apresentam duas linhas: a linha em fundo cinza indica o comportamento esperado e a linha em fundo branco indica o comportamento observado. Nessas colunas os números podem vir acompanhados por um "s", o qual indica que o elemento foi inserido, ou por um "n", o qual indica que o elemento não foi inserido.

Experimento 1: Falhas permanentes no *ListServer* - Blocos de Recuperação (BR)

A	F	E	I	II	III
1	8	80	0	80s	0
			0	80s	0
2	1	10	1s	9s	0
			1s	0	9n
3	1	10	2s	8s	0
			2s	0	8n
4	2	20	6s	14s	0
			6s	0	14n
5	2	20	8s	12s	0
			8s	0	12n
6	1	10	5s	5s	0
			5s	0	5n
7	1	10	6s	4s	0
			6s	0	4n
8	1	10	7s	3s	0
			7s	0	3n
9	3	30	24s	6s	0
			24s	0	6n
10	0	0	0	0	0
	20	200			

- em 69,5% das operações executadas o comportamento da aplicação foi conforme esperado, sendo que em 29,5% das operações não houve injeção e 40% houve injeção com ativação no início da execução da aplicação.
- em 30.5% das solicitações o comportamento da aplicação foi diferente do esperado.
- em 40% das execuções a aplicação comportou-se como esperado o que corresponde a percentagem de execuções com injeção de falhas ativadas no início da execução. Nesses casos, 2 variantes foram executadas e a operação concluída.
- em 60% das execuções a aplicação não comportou-se como esperado o que corresponde a percentagem de execuções com injeção de falhas ativadas depois do *empilha()* inicial. Nesses casos, 3 variantes foram executadas e o item não foi inserido.

Tabela 7.1 Experimento 1.

Experimento 2: Falhas transientes no *ListServer* - BR

A	F	E	I	II	III
1	6	60	54s	6s	0
			54s	6s	0
2	4	40	36s	4s	0
			36s	0	4n
3	2	20	18s	2s	0
			18s	0	2n
4	0	0	0	0	0
5	2	20	18s	2s	0
			18s	0	2n
6	1	10	9s	1s	0
			9s	0	1n
7	0	0	0	0	0
8	0	0	0	0	0
9	2	20	18s	2s	0
			18s	0	2n
10	3	30	27s	3s	0
			27s	0	3n
	20	200			

- em 93% das solicitações de *empilha()* o comportamento da aplicação foi o esperado, sendo que em 90% das operações não houve injeção e 3% houve injeção com ativação no início da execução da aplicação.
- em 7% das solicitações o comportamento da aplicação foi diferente do esperado.
- em 30% das execuções a aplicação comportou-se como esperado, o que corresponde a percentagem de execuções com injeção de falhas ativadas no início da execução. Nesses casos, 2 variantes foram executadas e a operação concluída.
- em 70% das execuções a aplicação não se comportou como esperado, o que corresponde a percentagem de execuções com injeção de falhas ativadas depois do *empilha()* inicial. Nesses casos, 3 variantes foram executadas e o item não foi inserido

Tabela 7.2 Experimento 2.

Experimento 3: Falhas intermitentes no *ListServer* - BR

A	F	E	I	II	III
1	8	80	49s	31s	0
			49s	8s	23n
2	6	60	36s	24s	0
			36s	0	24n
3	0	0	0	0	0
4	0	0	0	0	0
5	4	40	32s	8s	0
			32s	0	8n
6	0	0	0	0	0
7	0	0	0	0	0
8	1	10	9s	1s	0
			9s	0	1n
9	1	10	9s	1s	0
			9s	0	1n
10	0	0	0	0	0
	20	200			

- em 71.5% das operações executadas o comportamento da aplicação foi conforme esperado, sendo que em 67.5% das operações não houve injeção e 4% houve injeção com ativação no início da execução da aplicação.
- em 28.5% das solicitações o comportamento da aplicação não foi o esperado.
- em 100% das execuções a aplicação não se comportou como esperado.

Tabela 7.3 Experimento 3.

Resultados dos Experimentos 1 - 3

Esses experimentos foram significativos, porque revelaram a presença de uma falha na implementação do mecanismo de blocos de recuperação e demonstraram que o momento da injeção foi um fator muito importante. As falhas injetadas desde o início da execução da aplicação alvo não revelaram um comportamento diferente do especificado, apenas com a variação do início da ativação da injeção de falhas esse defeito foi observado.

O gráfico da Figura 7.2, mostra a influência do início da ativação na revelação de defeitos. Tanto para falhas transientes como para as falhas permanentes, o comportamento errôneo da aplicação foi observado quando a ativação das falhas era diferente de 1. Para falhas permanentes, em 40% das execuções foram injetadas falhas com ativação inicial em 1 e o comportamento da aplicação foi correto e em 60% das execuções foram injetadas falhas com ativação inicial maior que 1 e o comportamento da aplicação foi errôneo. Para falhas transientes, em 30% das execuções foram injetadas falhas com ativação inicial em 1 e o comportamento da aplicação foi correto e em 70% das execuções foram injetadas falhas com ativação inicial maior que 1 e o comportamento da aplicação foi errôneo. Para falhas intermitentes o comportamento da aplicação foi sempre diferente do especificado independentemente da ativação inicial.

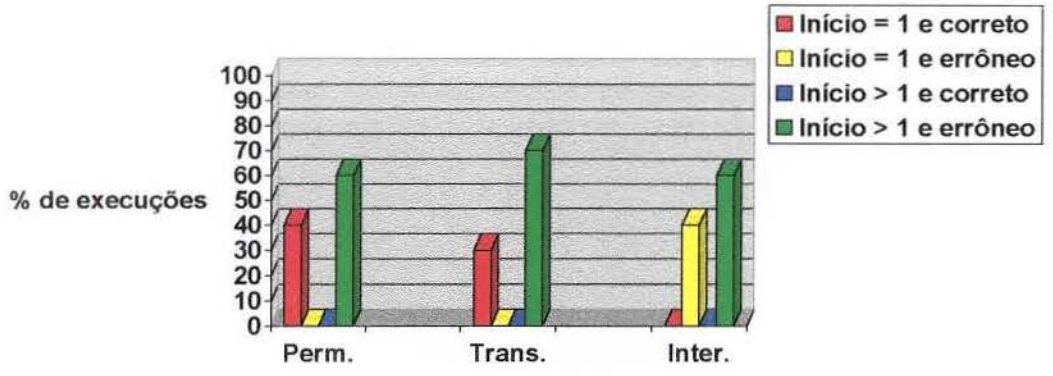


Figura 7.2 Gráfico de falhas no ListServer - Blocos de Recuperação

Experimentos 4, 5, 6: Falhas no *ListServer* - N-Versões

Esses experimentos não foram muito significativos, uma vez que não revelaram nenhum tipo de comportamento errôneo quando o mecanismo de N-versões foi utilizado. Por esse motivo as tabelas que resumem esses três experimentos não foram apresentadas. O gráfico da Figura 7.3 ilustra que o comportamento da aplicação foi sempre correto independentemente do tipo de falha injetada e do momento da injeção.

Os resultados obtidos com esse experimentos revelaram que os defeitos observados nos experimentos 1, 2 e 3 estavam realmente relacionados com o mecanismo de blocos de recuperação e não com as variantes. Acredita-se que isso ocorreu porque em N-versões a execução das variantes não é sequencial, como ocorre no mecanismo de blocos de recuperação. Em N-versões as variantes executam em paralelo de forma independente, enquanto que em blocos de recuperação a execução das demais variantes dependem do resultado da execução da variante anterior.

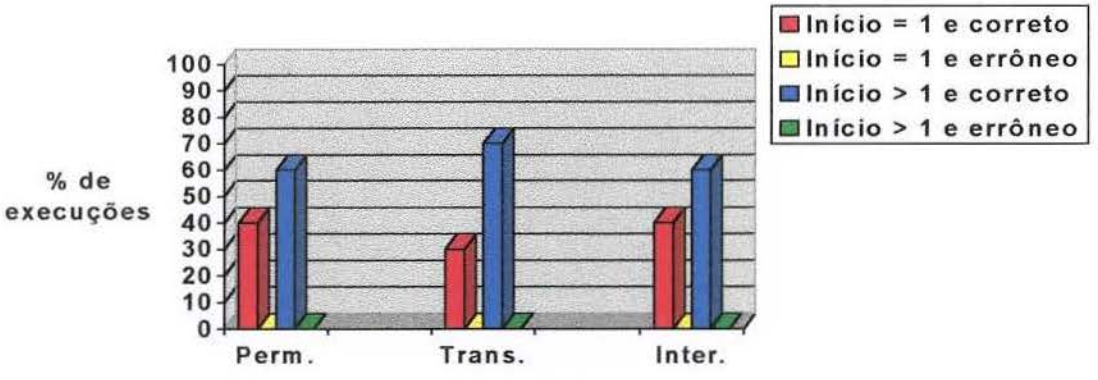


Figura 7.3 Gráfico de falhas no ListServer - N-versões

As próximas tabelas (experimentos de 7 - 12) correspondem às injeções de falhas na segunda variante *ArrayServer*. As falhas injetadas na primeira variante não estão

representadas nas tabelas, pois como mencionado anteriormente, são todas falhas permanentes com ativação no início da execução.

Experimento 7: Falhas permanentes no *ArrayServer* e no *ListServer* - BR

A	F	E	I	II	III
1	8	80	0	0	80s
			0	0	80s
2	1	10	0	1s	9s
			0	1s	9n
3	1	10	0	2s	8s
			0	2s	8n
4	2	20	0	6s	14s
			0	6s	14n
5	2	20	0	8s	12s
			0	8s	12n
6	1	10	0	5s	5s
			0	5s	5n
7	1	10	0	6s	4s
			0	6s	4n
8	1	10	0	7s	3s
			0	7s	3n
9	3	30	0	24s	6s
			0	24s	6n
10	0	0	0	0	0
	20	200			

Tabela 7.4 Experimento 7.

- em 69.5% das operações solicitadas o comportamento da aplicação foi o esperado sendo que em 29,5% das operações não houve injeção na segunda variante e em 40% houve injeção nas duas variantes com ativação no início da execução da aplicação.
- em 30.5% das solicitações o comportamento da aplicação foi diferente do esperado.
- em 40% das execuções a aplicação comportou-se como esperado o que corresponde a percentagem de execuções nas quais a injeção de falhas na segunda variante foi ativada no início da execução. Nestes casos 3 variantes foram executadas e a operação concluída.
- em 60% das execuções a aplicação não se comportou como esperado o que corresponde a percentagem de execuções nas quais a injeção de falhas na 2ª var foi ativada depois do push inicial. Nestes casos 3 variantes foram executadas e o elemento não foi inserido.
- .

Experimento 8: Falhas permanentes no *ListServer* e transientes no *ArrayServer* - BR

A	F	E	I	II	III
1	6	60	0	54s	6s
			0	0	60s
2	4	40	0	36s	4s
			0	4s	36n
3	2	20	0	18s	2s
			0	4s	16n
4	0	0	0	0	0
5	2	20	0	18s	2s
			0	8s	12n
6	1	10	0	9s	1s
			0	5s	5n
7	0	0	0	0	0
8	0	0	0	0	0
9	2	20	0	18s	2s
			0	16s	4n
10	3	30	0	27s	3s
			0	27s	3n
	20	200			

Tabela 7.5 Experimento 8.

- em 35% das solicitações de *empilha()* o comportamento da aplicação foi correto sendo que em 32% não houve corrupção na segunda variante e em 3% houve corrupção nas duas variantes com ativação no início da execução da aplicação.
- em 65% das solicitações o comportamento da aplicação foi diferente do esperado, sendo que em 58% não houve corrupção na segunda variante e em 7% houve corrupção nas duas variantes.
- em 100% das execuções a aplicação não se comportou como esperado. O comportamento observado é equivalente ao comportamento com injeção de falhas permanentes na segunda variante.

Experimento 9: Falhas permanentes no *ListServer* e intermitentes no *ArrayServer* - BR

A	F	E	I	II	III
1	8	80	0	49s	31s
			0	0	80s
2	6	60	0	36s	24s
			0	6s	54n
3	0	0	0	0	0
4	0	0	0	0	0
5	4	40	0	32s	8s
			0	16s	24n
6	0	0	0	0	0
7	0	0	0	0	0
8	1	10	0	9s	1s
			0	7s	3n
9	1	10	0	9s	1s
			0	8s	2n
10	0	0	0	0	0
	20	200			

- em 34% das solicitações o comportamento da aplicação foi o esperado sendo que em 18.5% não houve corrupção no *ArrayServer* e 15.5% houve corrupção com ativação no início da execução da aplicação.
- em 66% das solicitações o comportamento da aplicação não foi o esperado, sendo que em 49% não houve corrupção no *ArrayServer* e em 17% houve corrupção em ambas.
- em 100% das execuções a aplicação não se comportou como esperado.

Tabela 7.6 Experimento 9.

Experimento 10: Falhas permanentes no *ListServer* e no *ArrayServer* - N-versões (NV)

A	F	E	I	II	III
1	8	80	80	0	0
			80	0	0
2	1	10	9	1	0
			9	1	0
3	1	10	8	2	0
			8	2	0
4	2	20	14	6	0
			14	6	0
5	2	20	12	8	0
			12	8	0
6	1	10	5	5	0
			5	5	0
7	1	10	4	6	0
			4	6	0
8	1	10	3	7	0
			3	7	0
9	3	30	6	24	0
			6	24	0
10	0	0	0	0	0
	20	200			

- em 100% das solicitações o comportamento da aplicação foi o esperado sendo que em 70.5% não houve corrupção e 29.5% houve corrupção.
- em 100% das execuções a aplicação comportou-se como esperado. Apenas as variantes que sofreram injeção de falha falharam no seus resultados. O resultado correto foi da maioria e a operação concluída em todas as solicitações.

Tabela 7.7 Experimento 10.

Experimento 11: Falhas permanentes no *ListServer* e transientes no *ArrayServer* - NV

A	F	E	I	II	III
1	6	60	6	54	0
			60	0	0
2	4	40	4	36	0
			36	4	0
3	2	20	2	18	0
			16	4	0
4	0	0	0	0	0
5	2	20	2	18	0
			12	8	0
6	1	10	1	9	0
			5	5	0
7	0	0	0	0	0
8	0	0	0	0	0
9	2	20	2	18	0
			4	16	0
10	3	30	3	27	0
			3	27	0
	20	200			

- em 42% das operações o comportamento da aplicação foi correto sendo que em 32% não houve corrupção e 10% houve corrupção.
- em 100% das execuções a aplicação não se comportou como esperado.

Tabela 7.8 Experimento 11.

Experimento 12: Falhas permanentes no *ListServer* e intermitentes no *ArrayServer*-NV

A	F	E	I	II	III
1	8	80	31	49	0
			80	0	0
2	6	60	24	36	0
			54	6	0
3	0	0	0	0	0
4	0	0	0	0	0
5	4	40	8	32	0
			24	16	0
6	0	0	0	0	0
7	0	0	0	0	0
8	1	10	1	9	0
			3	7	0
9	1	10	1	9	0
			2	8	0
10	0	0	0	0	0
	20	200			

- em 51% das operações solicitadas o comportamento da aplicação foi conforme esperado sendo que em 18.5% não houve corrupção e 32.5% houve corrupção.
- em 49% das solicitações o comportamento da aplicação não foi como esperado sendo que em 49% não houve corrupção no *ArrayServer*.
- em 100% das execuções a aplicação não se comportou como esperado.

Tabela 7.9 Experimento 12.

Resultados dos experimentos 7-12

Nesses experimentos foram injetadas falhas nas variantes *ListServer* e *ArrayServer*. A variante *ListServer* sofreu injeção de falhas permanentes. A variante *ArrayServer* sofreu

injeção de todos os tipos de falhas. Quando foram injetadas falhas permanentes na *ArrayServer,* os resultados foram semelhantes aos obtidos nos experimentos anteriores, ou seja, observou-se o mesmo defeito em relação ao mecanismo de blocos de recuperação e nenhum defeito em relação o mecanismo de N-versões. No entanto, quando falhas temporárias foram injetadas na *ArrayServer* observou-se também um outro comportamento não esperado (independente do mecanismo utilizado): a partir do momento em que a *ArrayServer* sofria a primeira injeção e falhava no seu teste, ela continuava sempre falhando mesmo sem sofrer injeção. Ou seja, o resultado da segunda variante passava a não ser mais aceito revelando uma falha na implementação da*ArrayServer.*

7.3.3 Experimentos de injeção de falhas em aplicação reflexiva

Foi realizada uma série de testes com programas reflexivos simples para verificar a aplicabilidade da FIRE em aplicações desse tipo. O objetivo nesses casos foi investigar como pode ser realizada a injeção de falhas e a monitorização quando as aplicações já possuem metaobjetos.

Normalmente, em aplicações reflexivas o nível base implementa a funcionalidade da aplicação e o meta-nível implementa as características não-funcionais. Especificamente no caso de aplicações tolerantes a falhas, os MTF's ficam no meta-nível e são ativados por mensagens reflexivas do nível base. Dependendo de como a aplicação alvo está implementada, os seguintes casos podem ser desejados:

1. injetar falhas em objetos de um classe reflexiva do nível base;
2. injetar falhas em objetos de uma classe não-reflexiva do nível base;
3. injetar falhas no meta-nível;
4. monitorar objetos de uma classe reflexiva do nível base;
5. monitorar objetos de uma classe não-reflexiva do nível base e,
6. monitorar o meta-nível.

Injeção e/ou Monitorização de objetos de classes não-reflexivas do nível base

Os casos 2 e 5 são semelhantes aos casos nos quais a aplicação alvo não é reflexiva, afinal a classe a ser instrumentada não está associada a nenhuma meta-classe e pode ser associada à meta-classe de injeção e monitorização. Esses casos são cobertos pela FIRE.

Injeção e/ou Monitorização de objetos do meta-nível

Os casos 3 e 6 são semelhantes aos casos 2 e 5, a única diferença é que as classes fazem parte do meta-nível da aplicação alvo e, portanto, a biblioteca de injeção e monitorização deve ser introduzida como o meta-nível do meta-nível da aplicação. FIRE cobre esses casos.

Injeção e/ou Monitorização de objetos do nível base a partir do meta-meta-nível

Os casos 1 e 4 não são tão simples, pois o objeto alvo pertence ao nível base e já possui um metaobjeto que o controla. Assim, para injetar falhas e monitorar um objeto reflexivo, é preciso instrumentar a meta-classe da classe alvo incluindo a biblioteca de injeção como um meta-meta-nível, como nos casos 2 e 5. Mas, como na arquitetura reflexiva um nível controla o nível imediatamente inferior, o metaobjeto de injeção de falhas deve encontrar os argumentos do nível base entre os argumentos do objeto do meta-nível. Para que o *escalonador* saiba distinguir essa última situação o usuário deve indicar que a injeção não é feita diretamente sobre os dados do meta-nível, mas sobre os dados do nível base que foram materializados para o meta-nível.

Os testes com programas reflexivos mostraram que a FIRE cobre a injeção indireta, no entanto, não cobre a monitorização indireta de argumentos de um método do nível base através do meta-nível. Isso ocorre porque para coletar devidamente os valores dos argumentos, os *sensores* precisam conhecer a assinatura do método alvo. No entanto, o PMO de OpenC++1.2 oferece apenas métodos que fornecem o nome de métodos e classes do nível imediatamente inferior. Ou seja, *escalonadores* localizados no meta-meta-nível conseguem encontrar o identificador de um método do nível base, mas não conseguem descobrir qual o nome associado ao identificador em questão.

Injeção e/ou Monitorização de argumentos de saída e atributos

Como os experimentos com a aplicação alvo PilhaRobusta não cobriram todas as funcionalidades da FIRE, alguns outros testes com programas reflexivos foram realizados para estudar a injeção e/ou monitorização de atributos reflexivos e a injeção e/ou monitorização de argumentos de saída. Os resultados obtidos foram os seguintes:

- atributos só podem sofrer injeção e/ou monitorização quando estão envolvidos em troca de mensagens, ou seja quando o atributo é utilizado pela sua própria classe ele não é refletido, mesmo que ele seja um atributo público e,
- a injeção de falhas em argumentos de saída não é realizada no valor do argumento, mas no seu endereço. Isso ocorre porque, na verdade, o que é passado para o método é a referência do argumento e não o seu valor. Um outro ponto importante, é que a injeção de falhas tem que ser feita antes da execução do método reflexivo, porque de acordo com o protocolo de OpenC++1.2, os argumentos contidos no objeto *ArgPac* são

desempilhados e passados para o método reflexivo e após a execução do método o objeto fica vazio.

7.3.4 Tempos de execução

O objetivo da análise dos tempos de execução é avaliar o desempenho do mecanismo de reflexão computacional para injeção de falhas. Para que uma boa comparação possa ser feita seria necessário comparar o desempenho da FIRE com outras ferramentas de injeção de falhas. Como não foi possível disponibilizar outras ferramentas para um estudo comparativo, a solução foi comparar a solução reflexiva com a injeção através de mutação de código. Tal técnica já havia sido utilizada para validar o FOOD [Pra98]. É importante destacar que, embora a utilização de mutantes seja uma técnica de injeção de falhas eficiente, ela não possibilita a monitorização e a injeção de falhas intermitentes e transientes bem como a injeção de falhas com ativação posterior à execução inicial (a menos que um código extra de controle seja adicionado, o que não foi o caso da técnica utilizada nesse caso).

PilhaRobusta foi novamente utilizada nesses experimentos. O tempo foi coletado em segundos, e como a aplicação alvo foi implementada em um ambiente distribuído, o tempo utilizado pela rede não foi desprezado. Para facilitar a comparação, os tempos gastos durante o armazenamento de 200 elementos foram coletados considerando-se os seguintes casos: (1) injeção na 1ª variante de uma falha permanente com ativação no início da execução, e (2) injeção na 1ª e 2ª variante de uma falha permanente com ativação no início da execução. A Tabela 7.10 ilustra os tempos para as duas técnicas considerando os dois casos descritos.

Casos	PilhaRobusta e FIRE	PilhaRobusta e Mutação
Blocos de recuperação com injeção na 1ª variante	27.9s	25.9s
Blocos de recuperação com injeção na 1ª e 2ª variantes	39.4s	38.2s

Tabela 7.10 Tempos de execução para injeção reflexiva e injeção por mutantes.

Os tempos de execução demonstraram que o *overhead* introduzido pela reflexão é baixo quando comparado a tempos gastos na utilização de outros recursos do sistema, como comunicação da rede e acesso a disco.

7.4 Resumo

Este capítulo apresentou a aplicabilidade da FIRE em testes de injeção de falhas por software. Os resultados obtidos indicaram que, apesar de algumas limitações do

OpenC++1.2, a abordagem reflexiva é útil para injetar falhas pois seu uso é simples e a perturbação na aplicação alvo não é muito grande.

Um outro fator importante observado foi que o uso de reflexão computacional permitiu uma grande flexibilidade de injeção. Por exemplo, a facilidade de injetar falhas em um momento diferente do inicial é muito importante para validar aplicações que possuem funções recursivas e laços. Nesses casos é importante estudar o comportamento de métodos, ou o valor de atributos em um particular momento de execução ao invés do primeiro ou todos os momentos em que ele são executados ou manipulados. Além disso essa forma de injeção permite simular falhas de processador, memória, barramentos e comunicação, e essas falhas podem ser permanentes ou temporárias.

Embora FIRE seja implementada de forma centralizada, os experimentos mostraram que ela pode ser utilizada para validar aplicações distribuídas. Cada objeto distribuído pode ser associado a metaobjetos *escalonadores*, os quais são associados a arquivos de falhas e arquivos de históricos distintos. A análise dos históricos permite o estudo do comportamento de cada objeto e do relacionamento entre eles.

Os experimentos mostraram que é viável utilizar a FIRE em aplicações reflexivas, no entanto, caso os metaobjetos *escalonadores* tenham que ser introduzidos no meta-meta-nível não é possível realizar monitorização de dados do nível base, apenas de dados do meta-nível.

Durante os experimentos algumas limitações puderam ser observadas:

- não foi possível estudar as características privadas de objetos e,
- não foi possível instrumentar classes *templates*, operadores sobrecarregados nem dados do tipo lista.

Apesar das limitações, os testes permitiram revelar falhas na implementação da aplicação alvo:

- no mecanismo de blocos de recuperação e,
- em uma das variantes.

É interessante observar que os experimentos permitiram testar o comportamento dos mecanismos de blocos de recuperação e N-versões em presença de falhas permanentes e temporárias (transientes e intermitentes). Em presença de falhas permanentes observou-se um comportamento errôneo da aplicação quando as falhas não foram injetadas desde o início dos testes, devido a uma falha de implementação da aplicação (no mecanismo de blocos de recuperação). Portanto, apesar dos mecanismos serem voltados para tolerar falhas de software, as quais são permanentes, a aplicação apresentou um defeito.

As falhas temporárias permitiram encontrar uma falha na implementação de uma variante (*ArrayServer*): esta comportava-se como se a falha injetada fosse permanente. No

mecanismo de blocos de recuperação ocorria também o defeito na execução da aplicação devido à falha de implementação citada anteriormente. Com o mecanismo de N-versões essa falha na *ArrayServer* só foi encontrada devido a capacidade de monitorização da FIRE, pois o mecanismo tolerava a falha como era esperado.

Quanto ao desempenho da ferramenta, os tempos de execução demonstraram que o *overhead* decorrente da materialização das mensagens dos mecanismos de reflexão é baixo dependendo das atividades executadas no meta-nível, e bem pequeno quando comparado a tempos gastos na utilização de outros recursos do sistema, como comunicação da rede e acesso a memória estável. No entanto, ainda é necessário comparar o *overhead* da reflexão computacional com o *overhead* introduzido por outras técnicas de injeção diferentes da mutação, afinal a essa última não possui controle durante a execução.

Capítulo 8

Conclusões e Trabalhos Futuros

Este capítulo finaliza esta dissertação apresentando os resultados obtidos com o desenvolvimento da FIRE e sugerindo alguns trabalhos que envolvem o uso de reflexão computacional para injeção de falhas.

Atualmente sistemas tolerantes a falhas orientados a objetos têm sido cada vez mais projetados e usados em domínios diversos. Para que confiança possa ser justamente depositada em tais sistemas eles devem ser devidamente validados. Injeção de falhas por software tem se mostrado uma técnica promissora para avaliar tais sistemas. Apesar de suas características atrativas, essa técnica ainda apresenta algumas limitações. Uma importante delas é a intrusividade em relação ao comportamento e à estrutura do sistema alvo. A menos que o ambiente de teste ofereça características que facilitem a injeção e monitorização de falhas ou uma abordagem híbrida seja utilizada, alguma forma de instrumentação deve ser introduzida no programa alvo, para alcançar uma maior flexibilidade na injeção de falhas. Essa instrumentação é intrusiva, ou seja, interfere na estrutura (mistura aspectos da instrumentação com os aspectos da funcionalidade da aplicação) e na execução da aplicação alvo.

Esta dissertação apresentou uma solução reflexiva para minimizar o problema da intrusividade da instrumentação nos testes de injeção de falhas por software.

Reflexão foi escolhida por ser uma técnica promissora quando é necessário dispor de informações especiais sobre a representação do programa, de forma independente da aplicação, como é o caso da injeção. Além disso, reflexão permite uma simples separação entre a funcionalidade da instrumentação e a funcionalidade da aplicação alvo, e oferece um modo simples e quase transparente de ativação do código de injeção de falhas. Uma outra característica importante é que a separação de domínios propicia a reutilização tanto de objetos da aplicação como de metaobjetos.

A ferramenta de injeção de falhas FIRE foi construída para verificar a viabilidade da abordagem reflexiva para injeção. FIRE foi implementada utilizando a linguagem C++ e o pré-processador OpenC++ 1.2.

FIRE usa as facilidades oferecidas pelo protocolo de metaobjetos de OpenC++1.2 para injetar falhas e monitorizar seus efeitos. A estrutura original da aplicação alvo não é

alterada, apenas algumas diretivas são introduzidas. FIRE pode injetar falhas permanentes, intermitentes e transientes. O modelo de falhas proposto para a ferramenta inclue falhas internas e falhas externas em relação aos objetos da aplicação alvo. Atualmente, o mecanismo de injeção de falhas baseia-se na corrupção de valores de argumentos de métodos e de valores de atributos durante a leitura ou atualização.

Os experimentos com a FIRE revelaram as seguintes vantagens:

- facilita a instrumentação: não há necessidade de inserir explicitamente instruções de *traps* no código fonte da aplicação, nem executar a aplicação em modo traço.

- facilita o uso, uma vez que não é preciso executar a aplicação em modo privilegiado, nem usar características especiais do hardware, as quais na maioria das vezes não são disponíveis para usuários comuns.

- o uso de metaobjetos possibilita injetar diferentes tipos de falhas em diferentes objetos do nível base, de acordo com suas características.

- a facilidade de poder controlar o padrão de repetição e o momento de ativação inicial das falhas, mostrou-se bastante adequado para estudar o comportamento de métodos, ou o valor de atributos em um particular momento de execução ao invés do primeiro ou todos os momentos em que ele são executados ou manipulados. Essa facilidade é muito importante para validar aplicações que possuem funções recursivas e laços.

- embora FIRE seja centralizada ela pode ser utilizada para validar aplicações distribuídas. Objetos distribuídos podem ser associados a metaobjetos *escalonadores*, os quais são associados a arquivos de falhas e arquivos de históricos distintos. A análise dos históricos permite o estudo do comportamento de cada objeto e do relacionamento entre eles

Algumas limitações também foram observadas:

- a execução da aplicação alvo continua sendo perturbada. A intrusividade comportamental só poderá ser verificada com base na comparação do *overhead* causado pelo mecanismo de reflexão com o causado pelos mecanismos de outras ferramentas de injeção;

- o código fonte continua sendo necessário;

- a injeção e a monitorização não atingem as características privadas de objetos e,

- não é possível instrumentar classes *templates*, operadores sobrecarregados nem dados do tipo lista.

As três últimas limitações são decorrentes das características do protocolo de metaobjetos do OpenC++1.2. Um PMO adequado para injeção deveria atender as seguintes características:

- oferecer uma maior flexibilidade quanto à cardinalidade do relacionamento, permitindo, por exemplo, que um objeto reflexivo possa ser associado a dois metaobjetos: um para injetar falhas e outro para monitorar o efeitos da falhas injetadas; ou ainda, no caso da aplicação alvo já ser reflexiva, que um metaobjeto de injeção de falhas possa ser associado a um objeto que já tenha um outro metaobjeto associado a ele;
- permitir que objetos de uma mesma classe possam ser associados a metaobjetos diferentes;
- permitir que o metaobjeto tenha acesso ao máximo de informações sobre o objeto a ele associado, ou seja para injetar falhas é interessante poder quebrar totalmente a barreira do encapsulamento do objeto;
- permitir introduzir características reflexivas em código executável e,
- introduzir um *overhead* mínimo.

Embora OpenC++1.2 ofereça reflexão de uma forma limitada, seu modelo de programação em níveis mostrou-se bastante adequado para injeção de falhas por software e encoraja novas pesquisas nessa área. As sugestões para trabalhos futuros são as seguintes:

- realizar uma série de experimentos com a FIRE. (1) Para comparar o seu nível de perturbação comportamental em relação a outras técnicas, como por exemplo inserção de *traps*. (2) Para verificar sua aplicabilidade em experimentos de previsão de falhas, que envolvem estudos estatísticos.
- estudar os diversos protocolos de metaobjetos existentes, verificar qual o mais adequado para injeção de falhas e propor um framework reflexivo para injeção de falhas por software.
- estudar a viabilidade do modelo de meta-classes para injeção de falhas através de transformações na estrutura estática das classes da aplicação alvo.

Referências

[AAA+90] Arlat, J.; Aguera, M.; Amat, L.; Crouzet, Y.; Fabre, J.-C.; Laprie, J.-C.; Martins, E.; Powell, D., "Fault injection for dependability validation - a methodology and some applications". *IEEE Transactions on Software Engineering*, vol. 16, n. 2, Fev/1990, p. 166-182.

[AT95] Avresky, D.R.; Tapadiya, P.K., *"A Method for Developing a Software-Based Fault Injection Tool"*. Technical Report, Departament of Computer Science, Texas A&M University n° 95-021, 1995.

[Bed97] Beder D., *"Integração dos Mecanismos de Recuperação de Erros por Avanço e Retrocesso"*. Dissertação de Mestrado, UNICAMP, Campinas, Abr/1997.

[BR98] Buzato L.E.; Rubira C.M.F., *"Construção de Sistemas Orientados a Objetos Confiáveis"*. 11ª Escola de Computação, Departamento de Computação Eletrônica, Coppe Sistemas, Universidade Federal do Rio de Janeiro, 20 - 24/Jul/1998.

[Car95] Carreira, J.F.V., *"Software Fault Injection in Parallel Systems"*. Dissertação de Mestrado, Universidade de Coimbra, Portugal, Jul/1995.

[Chi93] Chiba S., *"Open-C++ Release 1.2 Programmer's Guide"*, Technical Report n° 93-3, Dept. Information Science, University of Tokyo, 1993.

[CM93] Chiba S.; Masuda T., "Designing an Extensible Distributed Language with Meta-level Architecture". *Proc. ECOOP'93 LNCS 707*, 1993. p. 28-37.

[CMS95] Carreira, J.; Madeira, H.; Silva, J.G., "Xception: a software fault injection and monitoring in processor functional units". *5th IFIP International Working Conference on Dependable Computing for Critical Applications*, Urbana-Champaign, Illinois, USA, 1995, p. 135 -149.

[Cor97] Corrêa S.L.,*"Implementação de Sistemas Tolerantes a Falhas Usando Programação Reflexiva Orientada a Objetos"*. Dissertação de Mestrado, UNICAMP, Campinas, SP, Dez/1997.

[DJM9] Dawson S.; Jahanian F.; Mitton T., *"ORCHESTRA: A Fault Injection Environment for Distributed Systems"*. Obtido via internet: www.eecs.umich.edu.

[FNP+95] Fabre, J-C; Nicomette, V.; Perennou, T.; Stroud, R.; Wu, Z., "Implementing Fault Tolerant Applications using Reflective Object-Oriented Programming". *Proc. FTCS-25*, Pasadena (CA), Jun/1995, p. 489-498.

[Hof89] Hoffman D., "Hardware Testing and Software IC's", *Proc. Pacific NW Software Quality Conference*. Portland, Oregon, Set/1989.

[HTI97] Hsueh, M.C.; Tsai, T.K.; Iyer, R.K., "Fault Injection Techniques and Tools". *IEEE Computer*,Abr/1997, p.75-82.

[HRS93] Han, S.; Rosenberg, H.A; Shin, K.G., *"DOCTOR: an Integrated Software Fault Injection Environment"*. Technical Report, University of Michigan n° CSE-TR-192-93, 1993.

[KGH+95] Kung D.; Gao J.; Hsia P.; et. al., "Developing an Object-Oriented Software Testing and Maintenance Environment". *Communications of the ACM*, vol. 38, n° 10, Out/1995.

[KKA92] Kanawati, N.; Kanawati, G.; Abraham, J., "FERRARI: A Tool for the Validation of System Dependability Properties". *Proc. FTCS-22*, IEEE CS Press, Los Alamitos, Calif., 1992, p. 336-344.

[KKA95] Kanawati, N.; Kanawati, G.; Abraham, J., "Dependability evaluation using hybrid fault/error injection". *IEEE Transactions on Computer*, Fev/1995, p. 224-233.

[KJA98] Krishnamurthy N.; Jhaveri V.; Abraham J., "A Design Methodology for Software Fault Injection in Embedded Systems". *Proc of the 1998 IFIP International Workshop on Dependable Computing and its Applications*, Johannesburg, South Africa, 12-14/Jan/1998, p. 237 - 248.

[KIT93] Kao W.; Iyer R.K.; Tang D., "FINE: A Fault Injection and Monitoring Environment for Tracing the Unix System Behavior Under Faults.". *IEEE Transactions on Software Engineering*, vol 19, n° 11, Nov/1993, p. 1105 - 1118

[KI94] Kao W.; Iyer R.K., "DEFINE: A Distributed Fault Injection and Monitoring Enviroment". Proc. *IEEE Workshop on Fault-Tolerant Parallel and Distributed Systems*, Jun/1994.

[Lap95] Laprie J.-C., "Dependability - Its Attributes, Impairments and Means". *Predictably Dependable Computing Systems,* Randell B.; Laprie J.-C; kopetz H.; Littlewood B.; (Eds), Basic Research Series, Springer, 1995.

[LE93] Lovric, T.; Echtle, K., "ProFI: a Processor Fault Injection for Dependability Validation". *IEEE International Workshop on Fault and Error Injection for Dependability Validation*, Gothenburg, Sweden, Jun/1993.

[Lei87] Leite J.C.B., Orlando G. Loques F°, Software. *II SCTF*, cap. 4 do mini-curso intitulado: Introdução à Tolerância a Falhas, Campinas, SP, 1987.

[Lis97] Lisboa, M.L.B., *"Reflexão Computacional no Modelo de Objetos"*. Mini-curso apresentado no II Simpósio Brasileiro de Linguagens de Programação, Campinas, SP, Brasil, Ago/1997.

[Mae87] Maes P., "Concepts and Experiments in Computaional Reflection". *Proc. OOPSLA'87*, 1987, p. 147-155.

[Mar95] Martins E., *"ATIFS - um Ambiente de Teste baseado em Injeção de Falhas por Software"*. Relatório técnico DCC-95-24, UNICAMP, Campinas, SP, 1995.

[Pra98] Prado D.P., *"Implementação de Sistemas Tolerantes a Falhas Usando Programação Orientada a Objetos"*. Dissertação de Mestrado, UNICAMP, Campinas, SP, Jan/1998.

[RBP+91] Rumbaugh J.; Blaha M.; Premerlani W.; Eddy F.; Lorensen W., *"Object-Oriented Modeling and Design"*. Prentice Hall, 1991.

[RC97] Rubira C. M. F., Corrêa S. L., "FORD: um Framework Orientado a Objetos Reflexivo para a Construção de Software Tolerante a Falhas". *Proc. VII Simpósio Brasileiro de Computação Tolerante a Falhas.* Campina Grande, PB, Brasil, 1997, p.75-89.

[RM98a] Rosa A.C.A.; Martins E., "Using Reflective Programming to Inject Faults into Object-oriented Systems". *Proc. of the 1998 IFIP Workshop on Dependable Computing and Its Applications*, Johannesburg, South Africa, 12-14/Jan/1998, p. 227-235.

[RM98b] Rosa A.C.A.; Martins E., "Utilizando Metaobjetos para Injetar Falhas e Monitorizar seus efeitos". *Anais do I Workshop de Tolerância a Falhas*, Porto Alegre, RS, 14-15/ Mai/1998, p. 37-42.

[RM98c] Rosa A.C.A.; Martins E., "Using a Reflective Architecture to Validate Object-oriented Applications by Fault Injection". *OOPSLA'98 Workshop on Reflective Programming in C++ and Java,* Out/1998.

[RS93] Rosenberg H.A., Shin K.G., "Software Fault Injection and its Application in Distributed Systems". *Proc. FTCS-23*, Toulouse, França, 1993.

[Ste92] Stevens W.R., *"Advanced Programming in UNIX Environment"*. Addison-Wesley Publishing Company, USA, 1992, 719 páginas.

[SVS+88] Segall Z.; Vrsalovic D.; Siewiorek D.P.,; Yaskin D.; Kownacki J.; Barton J.; Dancey R.; Robinson A.; Lin T., "FIAT-Fault Injection Based Automated Testing Environment". *Proc. FTCS-18*, Tokyo, Japão, 1988, p. 102-107.

[TI92] Tsai T.K.; Iyer R.K, "FTAPE: A Fault Tool To Measure Fault Tolerance". *Proc of AIAA Computing in Aerospace 10*, San Antonio, Texas, USA, 28-30 Mar/1992, p. 339-346.

[VBD+97] Vincenzi A.M.R.; Barbosa E.F.; Delamaro M.E.; Souza S.R.S.; Maldonado J.C., "Critério Análise de Mutantes: Estado Atual e Perspectivas". *Anais do Workshop do Projeto de Validação e Teste de Sistemas de Operação,* Águas de Lindóia, SP, 27-29 Jan/1997, p. 15-26.

[VMK+97] Voas J.M.; McGraw G.; Kassab L.; Voas L., "A 'crystal ball' for software ability". *IEEE Computer*, Jun/1997, p. 29-36.

[VM98] Voas J.; McGraw G., *"Software Fault Injection: Inoculating Programs Against Errors"*. Wiley Computer Publishing, EUA, 1998, 353 páginas.

[Voa98] Voas J.M., "Certifying Off-the-Shelf Software Components". *IEEE Computer,* Jun/1998, p. 53-59.

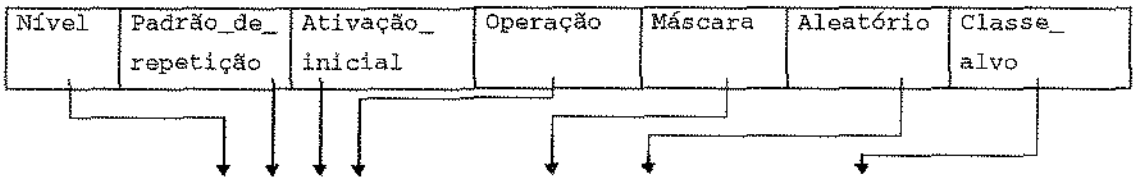
Apêndice A

Listagem dos nomes de categorias para injeção e monitorização.

Category_injection_1	Injeção com deslocamento 1.
Category_both_1	Injeção com deslocamento 1 e monitorização.
Category_injection_2	Injeção com deslocamento 2.
Category_both_2	Injeção com deslocamento 2 e monitorização.
Category_injection_3	Injeção com deslocamento 3.
Category_both_3	Injeção com deslocamento 3 e monitorização.
Category_injection_4	Injeção com deslocamento 4.
Category_both_4	Injeção com deslocamento 4 e monitorização.
Category_injection_5	Injeção com deslocamento 5.
Category_both_5	Injeção com deslocamento 5 e monitorização.
Category_injection_6	Injeção com deslocamento 6.
Category_both_6	Injeção com deslocamento 6 e monitorização.
Category_injection_7	Injeção com deslocamento 7.
Category_both_7	Injeção com deslocamento 7 e monitorização.
Category_injection_8	Injeção com deslocamento 8.
Category_both_8	Injeção com deslocamento 8 e monitorização.
Category_injection_9	Injeção com deslocamento 9.
Category_both_9	Injeção com deslocamento 9 e monitorização.
Category_injection_10	Injeção com deslocamento 10.
Category_both_10	Injeção com deslocamento 10 e monitorização.
Category_injection_re	Injeção no resultado.
Category_both_re	Injeção e monitorização no resultado.
Category_monitoring_me	Monitorização da chamada do método.
Category_monitoring_args	Monitorização de argumentos.
Category_monitoring_re	Monitorização do resultado.
Category_monitoring_var_int	Monitorização de atributo do tipo *int*.
Category_monitoring_var_dou	Monitorização de atributo do tipo *double*.
Category_monitoring_var_char	Monitorização de atributo do tipo *char*.
Category_monitoring_var_str	Monitorização de atributo do tipo *char**
Category_monitoring_var_ptr	Monitorização de atributo do tipo void*.
Category_injection_var	Injeção de atributo.
Category_both_var_int	Injeção e monitorização de atributo do tipo *int*.
Category_both_var_dou	Injeção e monitorização de atributo do tipo *double*.
Category_both_var_char	Injeção e monitorização de atributo do tipo *char*.
Category_both_var_str	Injeção e monitorização de atributo do tipo *char**
Category_both_var_ptr	Injeção e monitorização de atributo do tipo void*.

Apêndice B

Um exemplo de arquivo de falhas: cada linha corresponde a uma falha.




```
0 1 1 7 1767274722 0 ListStack
0 1 1 7 326272115 0 ListStack
0 1 1 7 1343201072 0 ListStack
0 0 1 7 675780521 0 ListStack
0 0 1 7 744964398 0 ListStack
0 0 1 7 1969681615 0 ListStack
0 2 1 7 1116175964 0 ListStack
0 2 1 7 861472101 0 ListStack
0 2 1 7 1303003706 0 ListStack
0 2 1 7 1686638315 0 ListStack
0 1 3 7 2000430152 0 ListStack
0 1 3 7 1868141281 0 ListStack
0 1 3 7 1860847622 0 ListStack
0 0 3 7 1448521415 0 ListStack
0 0 3 7 1628650740 0 ListStack
0 0 3 7 1249606685 0 ListStack
0 2 3 7 835903122 0 ListStack
0 2 3 7 763719779 0 ListStack
0 2 3 7 429995104 0 ListStack
0 2 3 7 1775811865 0 ListStack
```