Melhorando a Dependabilidade de Componentes com o uso de *Wrappers*

Naaliel Vicente Mendes[1], Regina Lúcia de Oliveira Moraes[2], Eliane Martins[3], Henrique Madeira[1]

[1]Faculdade de Ciências e Tecnologia – Universidade de Coimbra
3030-290 – Coimbra – Portugal

[2]Centro Superior de Educação Tecnológica – Universidade Estadual de Campinas (UNICAMP)
Caixa Postal 456 – 13.484-370 – Limeira – SP – Brasil

[3]Instituto de Computação – Universidade Estadual de Campinas (UNICAMP)
Caixa Postal 6.176 – 13.084-971 – Campinas – SP – Brasil

{naaliel,henrique}@dei.uc.pt,{regina@ceset, eliane@ic}.unicamp.br

Abstract. *Protective wrappers are used to make components behave in a robust way. They are particularly useful when the robustness of the reused component is not guaranteed. How to be sure that the wrapped component meets the system requirements? How to assure that the wrapper really protects the system against component failures? This paper presents the use of system-level fault injection to answer these questions. We also present a case study to illustrate the proposed methodology as well as the experimental results obtained.*

Resumo. *Wrappers são utilizados para fazer com que os componentes por eles protegidos se comportem da maneira especificada e robusta. Eles são úteis quando se quer adotar um componente do qual não se tem informação quanto à sua confiabilidade. Como assegurar que o componente protegido pelo wrapper atende os requisitos do sistema? Como assegurar que o wrapper realmente protege o sistema contra os defeitos do componente? Este trabalho apresenta o uso da técnica de injeção de falhas para responder a estas perguntas. Um estudo de caso é apresentado para ilustrar a metodologia proposta como também os resultados dos experimentos que foram obtidos.*

1. Introdução

A crescente pressão para reduzir prazos e custo faz do desenvolvimento baseado em componentes uma tendência para a construção dos novos sistemas. Um modelo que está apresentando crescimento na indústria de software é o desenvolvimento das partes mais específicas do novo sistema e a integração de componentes reutilizáveis para suprir as funcionalidades mais gerais. Apesar dos benefícios potenciais que esta técnica apresenta, alguns problemas são inerentes a esse modelo de construção de software, por exemplo, as condições operacionais onde o componente foi desenvolvido previamente diferem das atuais onde o mesmo está sendo reutilizado. Isto pode ativar falhas de software que antes não tinham sido reveladas, levando a que a confiabilidade esperada

não seja atingida [21]. Dessa forma, validar os componentes e os sistemas desenvolvidos com base na integração de componentes é uma tarefa essencial para garantir a qualidade do sistema que resulta desta integração. Porém, esta validação ainda apresenta um desafio para a comunidade de teste. A dificuldade está na falta de conhecimento a respeito do componente [2] [17]: por um lado, os desenvolvedores de sistemas não têm conhecimento de todas as maneiras que o componente poderá ser utilizado no futuro; por outro lado, os usuários não possuem informação a respeito da qualidade do componente e, mesmo que possuam esta informação, isso não é garantia de que o componente irá se comportar da mesma maneira quando utilizado em um novo contexto. Além disso, componentes de alta qualidade não garantem que o sistema como um todo irá apresentar o mesmo nível de qualidade devido à complexidade das interações entre os diversos componentes [18].

Uma técnica que pode ser utilizada para lidar com esta incerteza é a implementação de *wrappers* [1]. *Wrappers* de proteção são utilizados para que dados e fluxo de controle sejam interceptados entre um componente e seu ambiente [9]. Os *wrappers* são úteis quando se utilizam componentes que não apresentam nível de confiabilidade desejada ou, componentes sobre o quais não se tenha informação do nível de qualidade alcançado.

Inspirado no trabalho apresentado em [17], este trabalho procura responder as seguintes questões: (i) o *wrapper* protege os componentes *off-the-shelf* (OTS) contra os defeitos que ocorrem no restante do sistema (RS)? (ii) o *wrapper* protege o sistema contra os defeitos do componente?

Para responder a estas perguntas usamos, neste trabalho, duas técnicas de injeção de falhas. A injeção de falhas de software tem sido estudada na última década [3] [12] [14] evoluindo para técnicas que realmente emulam falhas de software realistas com bom nível de precisão [6][7]. Com a técnica denominada G-SWFIT (Generic SoftWare Fault Injection Technique) [6], as falhas de software injetadas representam os tipos de falhas de software mais frequentes, com base em estudos de campo [7]. Os operadores utilizados pela técnica são aplicados no código executável, ou seja, a técnica pode ser aplicada mesmo quando o código fonte não se encontra disponível. Na segunda técnica de injeção de falhas utilizada neste trabalho, falhas ou erros são injetados corrompendo os dados que passam através das interfaces entre o componente sob teste (CT) e o restante do sistema (RS). Baseada na *Interface Propagation Analysis* (IPA) [19] o uso da injeção de falhas de interface tem como objetivo verificar a proteção que se pode alcançar com a adição dos *wrappers*, quando estes são utilizados como filtros para erros que porventura sejam propagados através das interfaces. Com o uso da técnica de injeção de falhas de interface é possível observar se as conseqüências dos erros injetados no sistema são evitadas ou minimizadas. A instrumentação para a injeção de erros é feita em tempo de carga do sistema, usando a ferramenta Jaca [13]. Esta ferramenta injeta erros na interface em sistemas escritos na linguagem Java e não requer o código fonte do sistema sob teste.

Para se medir a efetividade do *wrapper* de proteção as falhas de software e de interface foram aplicados no sistema sob teste e os resultados apresentados foram classificados. Depois, um *wrapper* de proteção foi adicionado e os experimentos foram repetidos sobre o novo sistema (que contém além dos componentes anteriores o *wrapper* de

proteção). Os resultados foram classificados usando os mesmos critérios e, dessa forma, comparados com os anteriormente verificados.

Os resultados mostraram que o *wrapper* foi eficiente para proteger o componente quando este recebeu erros propagados através das interfaces entre o RS e o CT, respondendo positivamente a questão (i). Também quando consideramos as falhas de software, o *wrapper* foi eficiente para proteger o sistema das conseqüências das falhas residuais no CT uma vez que houve uma sensível melhora no nível de tolerância aumentando em 15 pontos percentuais, neste caso. Dessa forma, em relação a questão (ii) pode se afirmar que o *wrapper* protege o sistema quando as falhas no interior do componente se manifestam. Porém, essa proteção é parcial, uma vez que apesar do *wrapper* alguns defeitos ainda foram revelados.

A próxima seção apresenta as técnicas de injeção de falhas utilizadas, a versão corrente da ferramenta Jaca e trabalhos relacionados com este que está sendo apresentado. A seção 3 apresenta o estudo de caso que foi utilizado para os experimentos. A seção 4 apresenta as estratégias utilizadas para a escolha dos pontos de injeção, pontos de monitoramento e construção do *wrapper* de proteção. Os resultados dos experimentos são apresentados na seção 5. Conclusões e trabalhos futuros estão na última seção.

2. Injeção de Falhas por Software

Injeção de Falhas é uma técnica que simula as anomalias de software introduzindo falhas no sistema sob teste para que se possa observar este sistema e entender seu comportamento em presença das falhas injetadas. Tem sido amplamente utilizada para avaliar a dependabilidade (termo utilizado como tradução de *dependability* em inglês) e para validar os mecanismos de tratamento de erros de sistemas de software [14]. Esta técnica pode ser utilizada para validar um sistema tolerante a falhas, auxiliando na remoção de falhas, minimizando sua ocorrência e sua severidade, como também auxiliando na prevenção de falhas. A remoção e a prevenção de falhas que se obtêm, melhoram a dependabilidade dos sistemas que utilizam esta técnica de validação [20].

Entre as diversas abordagens (veja [10] para uma visão geral), a injeção de falhas por software (*Software Implemented Fault Injection* – SWIFI em inglês) tem sido bastante difundida [5] [8] [15]. Por não precisar de hardware especial e apresentar facilidade no controle da injeção e na observação da propagação de erros, esta abordagem tem se tornado mais popular entre os desenvolvedores de sistemas tolerantes a falhas e será a técnica utilizada neste projeto.

Inicialmente utilizada para emular falha transiente de hardware, a injeção de falhas por software tem sido utilizada mais recentemente na simulação de falhas e *bugs* de software. Neste caso, SWIFI pode simular falhas internas (ou falhas de software), que representam falhas de projeto e implementação (variáveis que estão erradas ou não inicializadas, atribuições incorretas ou verificações incorretas de condições), ou falhas externas que representam todos os fatores externos que não estão relacionados com falhas no código alvo, mas alteram o estado do software através das interações entre suas interfaces [19].

No que se refere a falhas de software, o maior problema é a representatividade das falhas injetadas. O estudo publicado em [4], utilizou dados de campo e propôs uma classificação, a *Orthogonal Defect Classification* (ODC) e serviu como base para o

trabalho apresentado em [3]. A necessidade de se ter dados preliminares coletados em campo reduz a possibilidade de uso do método, uma vez que esses dados dificilmente estão disponíveis. Apenas recentemente um estudo apresentado em [6] propôs uma abordagem para se injetar falhas representativas de software. A técnica que foi denominada *Generic Software Fault Injection Technique* (G-SWFIT) foi definida a partir de um estudo de sistemas abertos (*open sources*) onde falhas reais de software foram coletadas e classificadas [7] numa proposta em que a classificação ODC foi estendida. Esta nova classificação tomou por base o contexto do programa onde a falha específica ocorre e relacionou as falhas com as estruturas de programação existentes nas linguagens de alto nível. De acordo com esta classificação, um defeito de software é uma ou mais estruturas de programação que são escritas de maneira errada, que são esquecidas ou que são colocadas em excesso em um código fonte e podem ser classificadas como tal (em inglês, *Wrong construct, Missing construct, Extraneous construct*) subdividindo cada uma das classificações ODC. Dessa forma, baseado no estudo de campo feito em [7] como também em [3], a técnica G-SWFIT usa uma biblioteca de operadores de emulação de falhas que representam os tipos de falhas mais comuns observados em campo. Os operadores definidos consistem de pares "{padrão de código, alteração de código}". O "padrão de código" representa código executável que é relacionado com estruturas nas linguagens de alto nível onde normalmente os programadores cometem erros de implementação conforme observado em campo. A "alteração de código" representa modificações que simulam enganos típicos dos programadores conforme a estrutura representada pelo "padrão de código" relacionado. Dessa forma, os tipos de falhas injetadas segundo a técnica G-SWFIT representam o código executável que seria gerado por um compilador, caso o engano na escrita do código fonte na linguagem de alto nível tivesse realmente sido cometido pelo desenvolvedor de um produto de software. A Tabela 1 exemplifica operadores que foram definidos para a injeção de falhas em instruções de seleção, particularmente dois operadores que simulam construções esquecidas (*Missing construct*). O conjunto completo de operadores pode ser encontrado em [7].

Tabela 1: Fragmento do Conjunto de Operadores G-SWFIT (missing if cond)

Operador	Tipo de falha	Padrão de Busca	Alteração no código
OIA	**MIA** Missing "if (*cond*)" surrounding statement(s)	**CMP *reg*, ...** ***jcond* after:** **...*instructions*...after:**	**Remove a instrução de "jump"**
OIS	**MIFS** Missing "If (*cond*) { statement(s) }"	**CMP *reg*, ...** ***jcond* after:** **...*instructions*...after:**	**Remove a instrução de "jump" e todas as instruções contidas (*instructions*)**

Vale ressaltar que a biblioteca de operadores é criada em código executável, permitindo a utilização quando o código fonte do componente sob teste não esteja disponível. Pode-se criar a biblioteca de operadores considerando-se o código executável gerado para as diferentes linguagens de programação, o que torna a técnica portável [7]. O uso da técnica exige dois passos: (i) identificação dos locais onde as falhas de software podem ser injetadas utilizando para esta tarefa um "*scan*" do código executável, gerando o conjunto de falhas; (ii) injeção das falhas durante a execução do sistema. A intrusão durante a execução do sistema é bem pequena uma vez que a identificação dos locais de injeção já foi previamente efetuada.

Outra abordagem da injeção de falhas por software consiste em injetar dados anômalos que são compartilhados através das interfaces [20]. Falhas de interface podem

representar falhas que são inseridas no componente sob teste através de suas interfaces ou erros propagados por outros componentes do sistema. Na abordagem clássica as falhas são injetadas nas fronteiras do componente sob teste e neste caso para se validar a robustez do sistema, qualquer valor é válido. Normalmente são utilizados valores extremos para cada tipo de dado que está sendo substituído pela injeção de falhas [11]. Podemos dizer que um software é robusto se, alimentado com entradas anômalas, não propaga erros que levem o sistema a apresentar um defeito. Dessa forma, o sistema demonstra que pode produzir serviços de confiança mesmo quando colocado num ambiente hostil [20]. Generalizando a abordagem clássica, podemos injetar falhas nas interfaces entre os componentes para simular o envio de dados corrompidos por um componente que apresentou um defeito, a outros componentes que estão interagindo com o componente falho através de chamadas via *Application Programming Interfaces* (API). Nesse caso, é importante que o valor injetado seja representativo e emulem a conseqüência de falhas residuais de software que possam estar presentes no componente que faz a chamada para a API. Essa representatividade nem sempre é fácil de ser obtida.

Neste trabalho usamos a Jaca [13], uma ferramenta de injeção de falhas de interface (na verdade a Jaca injeta erros que simulam as possíveis conseqüências de falhas em componentes predecessores), permitindo avaliar a robustez de sistemas orientados a objetos escritos na linguagem Java. As características fundamentais da ferramenta estão descritas com maiores detalhes em [13]. Jaca é independente do código fonte, permitindo a validação de um sistema composto por componentes desenvolvidos por terceiros. A versão atual da Jaca (JacaC3.0) consegue afetar interfaces públicas de um componente, alterando valores dos seus atributos, parâmetros e valores de retorno de seus métodos. Estes valores podem ser simples (inteiros, reais e booleanos), alfanuméricos ou objetos.

Os experimentos de injeção de falhas são interessantes quando se pode executar em grande quantidade possibilitando a inferência estatística sobre o resultado. A Jaca traz no seu pacote de instalação diversas rotinas automáticas para viabilizar os testes estatísticos. A ferramenta pode controlar a execução sem que erros sejam injetados, para se criar e armazenar um padrão de resultados. Em seguida são executados os experimentos onde os erros são injetados e o resultado de cada experimento comparado, de maneira automática, ao conteúdo do padrão de resultados. As divergências são destacadas, permitindo que se possa analisar um grande número de resultados.

Outro ponto importante é poder definir quando o sistema entrou num estado infinito, isto é, o experimento nem termina com sucesso e nem apresenta qualquer tipo de erro (*hang* em inglês). Para isso, é possível ajustar um parâmetro de *timeout* na ferramenta que limita o tempo de cada um dos experimentos. Se o experimento não se encerrar nesse tempo, a ferramenta o encerra e acusa um defeito que é registrado no *log*.

Visando a automatização dos experimentos, antes e depois da execução de um componente, é possível executar arquivos *batch*. A vantagem desta funcionalidade, por exemplo, é quando um experimento utiliza um banco de dados e este precisa estar ativo antes que o sistema alvo inicie sua execução. Neste caso, pode-se especificar na interface o caminho onde se encontram os *Batch Files*.

Na nova versão aumentaram-se as opções para a definição de valores a serem injetados nos experimentos, podem ser definidos pelo usuário, podem ser gerados automaticamente a partir de um valor inicial e um incremento definido ou ainda podem

ser recebidos a partir de um arquivo com extensão "*.xml*". Isto garante que uma falha pode ser injetada repetidas vezes com valores diferentes, facilitando as análises estatísticas e simulando de maneira mais real uma utilização do sistema.

Em termos de resultados, os arquivos de *log* foram remodelados para que apresentassem uma organização mais adequada tendo sido subdividido em três arquivos. O primeiro apresenta os erros que foram injetados e o modo de defeito (*failure mode* em inglês) que foi utilizado. As ocorrências de cada injeção são registradas no segundo arquivo e as exceções, se existirem, podem ser encontradas no terceiro arquivo.

3. Estudo de Caso

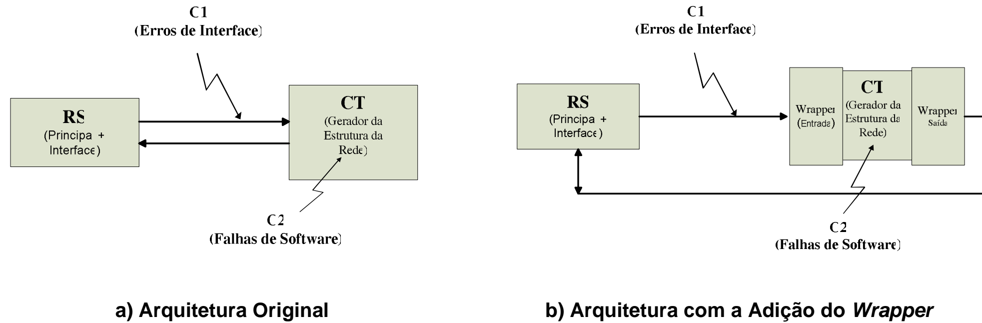
O estudo de caso utilizado para validar experimentalmente a metodologia foi um simulador de rede de computadores, SimuRed [16], que provê uma maneira visual de acompanhar os pacotes que estão sendo transacionados pela rede. Este simulador permite tanto a execução em *batch* quanto a interação direta com o usuário, sendo que este modo é o que foi utilizado neste trabalho. O projeto foi desenvolvido para uso didático e apresenta versão multiplataforma para Java, além das versões para Windows e Linux. Neste trabalho, foi utilizada a versão 2.1 para Java. Sua distribuição é gratuita, com código aberto (*open source*) nas linguagens Java e C++. Um conjunto de parâmetros é provido ao sistema através da sua interface. A Tabela 2 apresenta os parâmetros que foram considerados neste trabalho e seus respectivos valores mínimos e máximos previstos na especificação do sistema que estão indicados à frente do nome de cada parâmetro.

Tabela 2: Pontos de Injeção, Domínios e Valores Injetados – Testes de Robustez

Parâmetro	Valores Injetados		
	Válidos	Inválidos	Limites de Domínio
Dimensions [1 , 9]	5	-2147483648, -100, -1, 100, 1000, 2147483648	0, 1, 9,10
NodesDimensions [2 , 256]	10, 100	-2147483648, -100, -1, 0, 1000, 2147483648	1, 2, 256, 257
Virtuals [1 , 16]	10	-2147483648, -100, -1, 100, 1000, 2147483648	0,1, 16, 17
LenBuffer [1 , 256]	10, 100	-2147483648, -100, -1, 1000, 2147483648	0, 1, 256, 257
Buffer [1 , 16]	10	-2147483648, -100, -1, 100, 1000, 2147483648	0, 1, 16, 17
CrossBar [1 , 16]	10	-2147483648, -100, -1, 100, 1000, 2147483648	0, 1, 16, 17
Channel [1 , 16]	10	-2147483648, -100, -1, 100, 1000, 2147483648	0, 1, 16, 17
Switching [1 , 16]	10	-2147483648, -100, -1, 100, 1000, 2147483648	0, 1, 16, 17

O esquema da arquitetura do sistema usado neste trabalho é apresentado na Figura 1. Esta arquitetura foi utilizada para as primeiras campanhas de experimentos quando o sistema foi validado utilizando testes de robustez (injeção de falhas de interface) e injeção de falhas de software como indicado no esquema da Figura 1(a). Os locais de aplicação das campanhas de injeção de falhas estão apontados como C1 e C2 respectivamente.

Depois das duas primeiras campanhas, a arquitetura do sistema foi acrescida de um *wrapper* de proteção que foi construído baseado na especificação do sistema. O objetivo deste *wrapper* é validar as entradas que chegam do RS para o CT, bem como, validar os resultados do CT que devem retornar ao RS. O *wrapper* construído e integrado no sistema valida o domínio de cada parâmetro que é fornecido ao CT e captura exceções geradas internamente pelo CT, substituindo-as por mensagens que são devolvidas ao RS e por ele publicadas na interface. A Figura 1(b) apresenta o esquema da arquitetura do sistema modificado e os locais onde foram reaplicados os experimentos (C1 e C2).



a) Arquitetura Original b) Arquitetura com a Adição do *Wrapper*

Figura 1: Esquema da Arquitetura do Sistema SimuRed

4. Metodologia Proposta

Neste trabalho estamos considerando que, embora o sistema possa ser composto por componentes OTS que não disponibilizam o código fonte, a sua arquitetura, que representa como os diversos componentes estão interconectados, é conhecida. Nosso principal objetivo é responder as questões apontadas na primeira seção: (i) o *wrapper* protege o CT contra os defeitos do RS? e (ii) o *wrapper* protege o RS contra os defeitos do CT?

4.1. Determinando o Conjunto de Erros e Falhas

Neste trabalho injetamos erros de interface [17] e falhas de software [7]. Para se injetar falhas ou erros é preciso determinar um local, um tipo, uma condição que desencadeie o processo de injeção, um padrão de repetição e uma condição para o início da injeção. A subseção seguinte apresenta os locais onde erros e falhas de software foram injetados.

Quanto ao tipo, para a injeção de erros de interface foi utilizada corrupção de parâmetros e valores de retorno dos métodos que foram substituídos por outros valores válidos, valores inválidos e valores extremos do domínio especificado para os dados que estão sendo substituídos [11]. Para as falhas de software, foi utilizada a biblioteca G-SWFIT [7] que analisa as estruturas de programação existentes no código objeto da aplicação e substitui estas estruturas por outras que representem enganos cometidos por programadores na fase de escrita do código fonte do produto de software.

A condição para a injeção de falhas de interface é interceptar a chamada dos métodos para substituir os valores originais pelos novos valores, usando a ferramenta Jaca. Já para as falhas de software, um novo código objeto é gerado após a substituição das estruturas de programação. Este novo código é executado em substituição do original.

Quanto ao padrão de repetição, as falhas de software foram simuladas e, neste caso, a falha existe permanentemente (pode ser ou não ativada dependendo dos dados). Assim, a injeção das falhas (ou erros) deve se dar desde a primeira execução.

4.2. Pontos de Injeção

O conjunto de classes responsável por gerar a estrutura da rede do SimuRed foi escolhido como o componente sob teste (CT). O sistema considerado é então composto pelo CT e o resto do sistema (RS) que, por sua vez, é composto pela interface do usuário, o bloco principal e o simulador. Como o CT está sendo considerado um

componente “caixa-preta”, o nível de proteção que ele provê para assegurar a validação dos dados de entrada fornecidos de acordo com as especificações, só pode ser avaliado experimentalmente. Este conjunto de entradas é composto pelos valores dos parâmetros fornecidos pelo RS. O conjunto deve ser compatível com o que foi especificado ou, caso não o seja, deve ser tratado adequadamente provendo mensagens e encerrando a execução sem danos para o ambiente computacional. Para que se possa assegurar essa consistência, os erros devem ser injetados na interface entre o RS e o CT, como indicado no ponto C1 da Figura 1(a). Os valores utilizados para a injeção devem pertencer ao conjunto de valores válidos segundo a especificação, bem como ao conjunto de valores inválidos e ao conjunto de valores que pertençam ao limite do domínio especificado para cada entrada do sistema [11]. Dessa forma, estaremos testando a robustez do sistema, ou seja, se o sistema para qualquer que seja o conjunto de valores de entrada recebido consegue prover uma saída adequada e não apresente um defeito. A Tabela 2 apresenta os pontos de injeção e os valores que foram injetados em cada um desses pontos.

Nesse momento, um *wrapper* de proteção deverá ser adicionado ao sistema com o objetivo de proteger o CT de entradas não esperadas. Nesse caso, o *wrapper* baseado na especificação do sistema deve validar os valores que são trocados entre o RS e o CT e não deve permitir que valores não adequados sejam trocados entre as partes do sistema. A Figura 1(b) ilustra a localização do *wrapper* em relação ao sistema. O novo sistema (RS + CT + *wrapper*) será então submetido aos mesmos experimentos anteriores. Os pontos de injeção também devem ser conservados sendo que o CT + *wrapper* é considerado um conjunto atômico, ou seja, não se consideram públicas as interfaces entre ambos. Os resultados serão analisados para se observar a melhoria da qualidade do sistema que se obteve em decorrência da implantação do *wrapper* de proteção. Dessa forma, com a comparação dos resultados dos testes de robustez estaremos respondendo à pergunta (i).

Outra importante observação é o quanto o comportamento errôneo do CT pode afetar o RS. O que acontece com o RS se houver uma falha de software no CT que venha a se manifestar quando o componente estiver no ambiente operacional? Se o RS não conseguir se proteger das falhas no CT, pode ser que ao receber um erro propagado como conseqüência de uma falha residual no CT o sistema possa não mais responder adequadamente. Isso poderá trazer resultados ao ambiente operacional que, muitas vezes, pode ser catastrófico. Para isso, iremos injetar falhas de software no CT, usando a G-SWFIT e observar qual é a conseqüência da falha injetada nos demais componentes que pertencem ao RS. Assim, podemos avaliar o impacto que uma falha residual que venha a se manifestar pode ter no RS. A Tabela 3 apresenta os operadores utilizados e o número de falhas injetadas de cada tipo. Todas as falhas injetadas foram ativadas.

Quando se considera o sistema ao qual o *wrapper* foi adicionado iremos analisar, nesse caso, o quanto o *wrapper* de proteção é eficiente para evitar a propagação de erros advindos do CT. O *wrapper* baseado na especificação do sistema deve validar os valores que são trocados entre o CT e o RS, provendo mensagens e encerrando execuções quando for o caso. Nesse caso também os mesmos experimentos foram efetuados e os mesmos pontos de injeção foram utilizados.

Tabela 3: Operadores Injetados – Falhas de Software no CT

Operador	#Falhas Injetadas/ Ativadas
Retira inicialização feita através da atribuição de valor (OIV)	20
Retira atribuição feita através de um valor (OAV)	2
Retira instrução de seleção (*if*) (OIA)	1
Emula instrução de seleção (*if*) e as respectivas instruções (OIS)	1
Emula a omissão de parte da condição numa instrução de seleção (OLAC)	2

Com a comparação dos resultados dos testes onde foram injetadas falhas de software, estaremos respondendo à pergunta (ii) propostas na seção 1. Esses resultados são apresentados na seção 5.

4.3. Determinando os Pontos de Observação

Os pontos de observação foram fixados para que se pudessem observar as saídas inválidas enviadas através das interfaces. Estes pontos de observação geram registros no arquivo de log que posteriormente são analisados e comparados com a "*golden run*". Saídas inválidas podem ser entendidas como violação da especificação do sistema.

A monitoração do sistema, tanto para a injeção de falhas de interface quanto para a injeção de falhas de software, foi efetuada pela ferramenta Jaca. Foram observados a fronteira do sistema (para se verificar quais seriam os dados e exceções observados pelos usuários e quais desses dados seriam considerados um defeito do sistema) e a classe Red() que é a classe que cria a rede.

4.4. Resultados Esperados

Como resultado dos experimentos, o sistema pode apresentar um defeito ou tolerar as falhas injetadas. Tolerância às falhas injetadas (indicado como corretos) significa que o sistema produziu como resultado um valor que satisfaz sua especificação. Quando se está monitorando o sistema, a propagação de erros também é observada. Erros podem ser cancelados, quando os dados corrompidos são descartados ou sobrescritos. Os erros também podem permanecer latentes, quando os dados corrompidos permanecem sem uso [19]. Nestes casos, os erros são considerados tolerados pelo sistema.

Um defeito pode ser reportado retornando um valor inesperado (*wrong* em inglês) ou não reportado quando o sistema não termina (*hang* em inglês) ou interrompe seu processamento sem prévio aviso (*crash* em inglês).

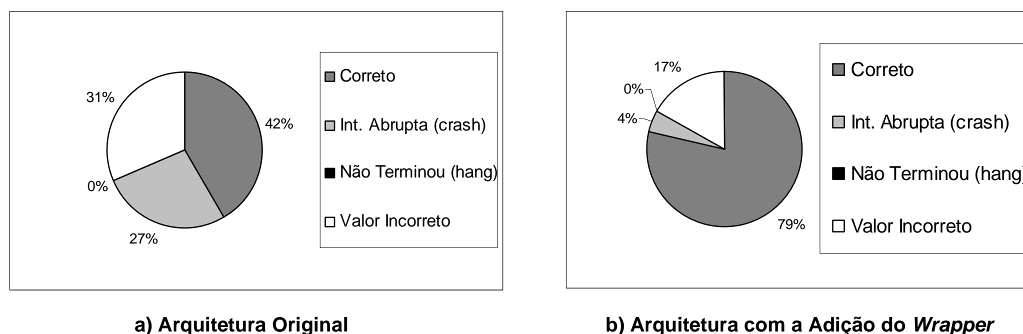
5. Resultados dos Experimentos

Na primeira campanha de injeção, foi considerado o sistema original (CT + RS). Esta campanha tinha como objetivo avaliar a robustez do CT. Para isso foram injetados erros de interface entre o RS e o CT, conforme apresentado na Tabela 2. Durante os experimentos, foram feitas 89 injeções. A Tabela 4 (sistema original), apresenta os resultados classificados de acordo com o comportamento observado após os experimentos. Nota-se que em aproximadamente 60% dos experimentos o resultado não atendeu a especificação do sistema, interrompendo o processamento sem prévio aviso (*crash*) ou reportando um resultado incorreto (*wrong*). O percentual de resultados corretos foi apresentado em pouco mais de 40% dos experimentos.

Tabela 4: Resultados - Testes de Robustez

Parâmetros	# Erros	Sistema Original								Sistema com Wrapper							
		Não Termin. (Hang)		Termina Abrupt. (Crash)		Resultados Errados (Wrong)		Corretos		Não Termina (Hang)		Termina Abrupt. (Crash)		Resultados Errados (Wrong)		Corretos	
		#	%	#	%	#	%	#	%	#	%	#	%	#	%	#	%
Dimensions	11	0	0	5	5,62	4	4,49	2	2,25	0	0	2	2,25	0	0,00	9	10,11
NodesDim	12	0	0	8	8,99	2	2,25	2	2,25	0	0	2	2,25	2	2,25	8	8,99
Virtuals	11	0	0	6	6,74	3	3,37	2	2,25	0	0	0	0,00	3	3,37	8	8,99
LenBuffer	11	0	0	5	5,62	1	1,12	5	5,62	0	0	0	0,00	4	4,49	7	7,87
Buffer	11	0	0	0	0,00	5	5,62	6	6,74	0	0	0	0,00	1	1,12	10	11,24
CrossBar	11	0	0	0	0,00	4	4,49	7	7,87	0	0	0	0,00	2	2,25	9	10,11
Channel	11	0	0	0	0,00	6	6,74	5	5,62	0	0	0	0,00	1	1,12	10	11,24
Switching	11	0	0	0	0,00	3	3,37	8	8,99	0	0	0	0,00	2	2,25	9	10,11
Total	89	0	0	24	26,97	28	31,46	37	41,57	0	0	4	4,50	15	16,85	70	78,65

A primeira campanha de testes foi re-aplicada. A Tabela 4, (sistema com *wrapper*) apresenta os resultados da campanha de injeções para o sistema acrescido do *wrapper*. Podemos observar que em relação aos resultados apresentados para o sistema original, tivemos uma melhora significativa de resultados corretos que passou de 41,57% para 78,65%. Mesmo entre os resultados que não atenderam às especificações, a severidade foi minimizada, uma vez que apenas 4,5% dos experimentos interromperam o processamento abruptamente (*crash*) e os resultados incorretos decresceram da ordem de 50% (*wrong*). Portanto, temos evidências suficientes para afirmar que o *wrapper* foi eficiente na proteção do CT, quando um defeito do RS tenta se propagar, respondendo a pergunta (i). A Figura 2 apresenta os resultados para os dois sistemas.



a) Arquitetura Original **b) Arquitetura com a Adição do *Wrapper***

Figura 2: Resultados dos Testes de Robustez

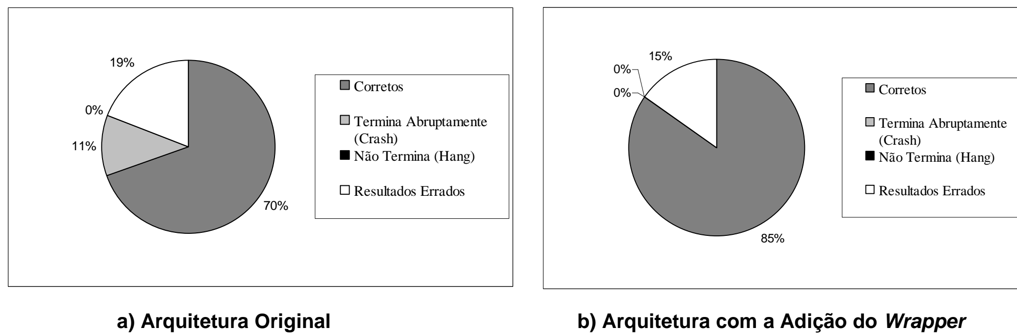
A segunda campanha de injeção foi feita para verificar o impacto no RS quando uma falha no CT é ativada. Injetamos falhas de software no CT conforme apresentado na Tabela 3. A Tabela 5 apresenta os resultados classificados de acordo com o comportamento observado após os experimentos para o sistema original. Nota-se que aproximadamente 70% dos experimentos apresentaram resultados corretos, quando as falhas injetadas não influenciaram os resultados observados. Entre os que apresentaram defeitos, 11,5% fizeram com que o sistema terminasse abruptamente (*crash*) e 19,2% apresentaram resultados incorretos (*wrong*).

A segunda campanha de teste também foi re-aplicada. Também neste caso, a existência do *wrapper* aumentou significativamente a tolerância a falhas injetadas no interior do CT elevando em mais de 15 pontos percentuais a falhas toleradas pelo sistema indicando que as exceções geradas pelo CT foram tratadas pelo *wrapper*.

Tabela 5: Resultados – Falhas de Software

Operadores	# Falhas	Sistema Original								Sistema com Wrapper							
		Não Termin. (Hang)		Termina Abrupt. (Crash)		Resultados Errados (Wrong)		Corretos		Não Termina (Hang)		Termina Abrupt. (Crash)		Resultados Errados (Wrong)		Corretos	
		#	%	#	%	#	%	#	%	#	%	#	%	#	%	#	%
Dimensions	20	0	0	1	3,8	3	11,5	16	61,5	0	0	0	0	2	7,7	18	69,2
NodesDim	2	0	0	1	3,8	1	3,8	0	0	0	0	0	0	1	3,8	1	3,8
Virtuals	1	0	0	1	3,8	0	0	0	0	0	0	0	0	0	0	1	3,8
LenBuffer	1	0	0	0	0	1	3,8	0	0	0	0	0	0	1	3,8	0	0
Buffer	2	0	0	0	0	0	0	2	7,7	0	0	0	0	0	0	2	7,7
Total	26	0	0	3	11,5	5	19,2	18	69,2	0	0	0	0	4	15,4	22	84,6

O *wrapper* se mostrou eficiente para tratar as conseqüências das falhas de software. A Tabela 5 (sistema com *wrapper*) apresenta os resultados para o sistema acrescido do *wrapper* e na Figura 3 os resultados em que falhas de software foram injetadas.



a) Arquitetura Original **b) Arquitetura com a Adição do *Wrapper***

Figura 3: Resultados dos Testes – Falhas de Software

6. Conclusões e Trabalhos Futuros

Apresentamos uma metodologia para proteger componentes *OTS* quando estes são integrados num sistema mais complexo. Ao se integrar um componente em um novo sistema é preciso garantir que a qualidade do componente que está sendo integrado não comprometa a qualidade do sistema. Propusemos o uso de wrapper para proteger os componentes e validamos a metodologia utilizando injeção de falhas por software.

Injetando falhas de interface e falhas de software mostramos que é possível avaliar tanto a robustez do componente em relação ao restante do sistema como o fragilidade do sistema em relação a falhas residuais que possam ser reveladas no componente. Através dos testes de robustez pode-se verificar a efetividade do *wrapper* quando este é utilizado para proteger os componentes contra os defeitos que ocorrem no restante do sistema. Injetando falhas de software a efetividade do *wrapper* para proteger o sistema contra os defeitos do componente foi igualmente verificada.

Pudemos observar que a utilização do *wrapper* protege parcialmente o sistema, melhorando em aproximadamente 15% a tolerância a falhas do sistema alvo quando se considera falhas residuais no componente e aproximadamente 37% quando o sistema é submetido a erros de interfaces.

Esta metodologia pode ser utilizada para selecionar um componente quando vários se encontrem disponíveis para prover uma mesma funcionalidade. Nesse caso, eles podem

ser integrados ao sistema um a um e a metodologia aplicada. A escolha deve recair no componente que após protegido apresente a melhor adaptação ao sistema no qual está sendo integrado, apresentando na validação pela injeção de falhas um maior nível de tolerância às falhas injetadas. Como trabalho futuro, pretende-se verificar a eficiência de se fazer a escolha de componentes baseada nesta metodologia.

Agradecimentos

Os autores agradecem a CAPES (Brasil), GRICES (Portugal) e FCT (Portugal) pelo patrocínio parcial deste trabalho.

Referências

[1] Anderson, T. *et. al*.: Protective Wrapper Development: A Case Study. Lecture Notes in Computer Science (LNCS), Vol. 2580, pp. 1-14, Spring Verlag, (2003).
[2] Beydeda, S. , Volker, G.: State of the art in testing components. Proc. of the International Conference on Quality Software, (2003).
[3] Christmansson, J., Chillarege, R. "Generation of an Error Set that Emulates Software Faults". Proc. of The 26th IEEE Fault Tolerant Computing Symp. – FCTS-26, Sendai, Japan, (1996).
[4] Chillarege, R. "Orthogonal Defect Classification". Handbook of Software Reliability Engineering, ch. 9, M. Lyu, Ed.: IEEE Computer Society Press, McGraw-Hill, (1995).
[5] De Millo, R. A., Li, T., Mathur, A. P. Architecture or TAMER: A Tool for dependability analysis of distributed fault-tolerant systems. Purdue University, (1994).
[6] Durães, J., Madeira, H. "Emulation of Software Faults by Educated Mutations at Machine-Code Level". Proc. of The Thirteenth Int. Symposium on Software Reliability Engineering – ISSRE'02, Annapolis, USA, (2002).
[7] Durães, J., Madeira, H. "Definition of Software Fault Emulation Operators: A Field Data Study". Proc. of The Int. Conference on Dependable Systems and Networks – DSN2003, pp. 105-114, San Francisco, USA, (2003).
[8] Fetzer, C., Högstedt, K., Felber, P. Automatic Detection and Masking of Non-Atomic Exception Handling. Proc. of *DSN 2003*, San Francisco, USA, pp. 445-454, (2003).
[9] Garlan, D., Allen, R., Ockerbloom, J.: Architecture Mismatche: Why Reuse is so Hard. IEEE Software, Vol. 12(6), pp. 17-26, (1995).
[10] Hsueh, Mei-Chen; Tsai, Timothy; Iyer, Ravishankar. "Fault Injection Techniques and Tools". *IEEE Computer*, pag 75-82, (1997).
[11] Koopman, P. Siewiorek, D, DeVale, K., DeVale, J., Fernsler, K., Guttendorf, D., Kropp, N., Pan, J., Shelton, C., Shi, Y.: Ballista Project : COTS Software Robustness Testing. Carnegie Mellon University. Disponível na World Wide Web em: http://www.ece.cmu.edu/~koopman/ballista/, (2003), acessado em 16/08/2005.
[12] Madeira, H. Vieira, M., Costa, D. "On the Emulation of Software Faults by Software Fault Injection.", Proc. of the Int. Conf. on Dependable System and Networks – DSN00, NY, USA. (2000).
[13] Martins, E., Rubira, C. M. F., Leme N.G.M.: Jaca: A reflective fault injection tool based on patterns. Proc of The 2002 International Conference on Dependable Systems & Networks, Washington D.C. pp. 483-487, (2002).
[14] Ng, W., Aycock, C., Chen, P. "Comparing Disk and Memory's Resistance to Operating System Crashes". Proc. of The 7th IEEE International Symposium on Software Reliability Engineering, ISSRE'96, New York, NY, USA, (1996).
[15] Rosenberg, L, Stapko, R, Gallo, A Risk-based Object Oriented Testing. Proc. Of the 13th International Software / Internet Quality Week (QW2000) , San Francisco, California USA, (2000).
[16] SimuRed, Multicomputer Network Simulator, http://tapec.uv.es/simured/index_en.php, março/06.
[17] Voas, J. An Approach to Certifying Off-the-Shelf Software Components. IEEE Computer, Vol. 31(6), pp. 53-59, (1998).
[18] Voas, J. M.,Charron, F., McGraw, G., Miller, K.,Friedman, M. Predicting how Badly Good Software can Behave. IEEE Software, pp. 73-83, (1997).
[19] Voas, J., McGraw, G.: Software Fault Injection: Inoculating Programs against Errors. John Wiley & Sons, New York, EUA, (1998).
[20] Voas, J.: Marrying Software Fault Injection Technology Results with Software Reliability Growth Models. Fast Abstract ISSRE 2003, Chillarege Press, (2003).
[21] Weyuker, E.J. "Testing Component-Based Software: A Cautionary Tale". IEEE Software, pp 54-59, (1998).