Uso de Risco na Validação de Sistemas Baseados em Componentes

Regina Lúcia de Oliveira Moraes

Tese de Doutorado

Instituto de Computação
Universidade Estadual de Campinas

Uso de Riscos na Validação de Sistemas Baseados em Componentes

Regina Lúcia de Oliveira Moraes
Dezembro de 2006

Banca Examinadora:

- Prof ª Drª Eliane Martins (Orientadora)
  Instituto de Computação - UNICAMP
- Prof ª Drª Cecília Mary Fischer Rubira
  Instituto de Computação - UNICAMP
- Prof. Dr. Rodolfo Jardim de Azevedo
  Instituto de Computação - UNICAMP
- Prof. Dr. José Carlos Maldonado
  Departamento de Ciência da Computação e Estatística – USP/SC
- Profª. Drª. Taisy Silva Weber
  Instituto de Informática – UFRGS
- Prof. Dr. Mário Jino
  Faculdade de Engenharia Elétrica e de Computação - UNICAMP
- Profa. Dra. Maria Beatriz Felgar de Toledo (Suplente)
  Instituto de Computação - UNICAMP

Moraes, Regina Lucia de Oliveira

M791u Uso de risco na validação de sistemas baseados em componentes / Regina Lucia de Oliveira Moraes -- Campinas, [S.P. :s.n.], 2006.

Orientador: Eliane Martins; Henrique Santos do Carmo Madeira

Tese (doutorado) - Universidade Estadual de Campinas, Instituto de Computação.

1. Software – Validação. 2. Engenharia de software – Injeção de falhas. 3. Avaliação de riscos . I. Martins, Eliane. II. Madeira, Henrique Santos do Carmo. III. Universidade Estadual de Campinas. Instituto de Computação. IV. Título.

(mjmr/imecc)

TERMO DE APROVAÇÃO

Tese defendida e aprovada em 15 de dezembro de 2006, pela Banca examinadora composta pelos Professores Doutores:

Prof. Dr. José Carlos Maldonado
USP - SC

Profa. Dra. Taisy Silva Weber
UFRGS

Prof. Dr. Mario Jino
FEE - UNICAMP

Profa. Dra. Cecília Mary Fischer Rubira
IC - UNICAMP

Prof. Dr. Rodolfo Jardim de Azevedo
IC - UNICAMP

Profa. Dra. Eliane Martins
IC - UNICAMP

Uso de Risco na Validação de Sistemas Baseados em Componentes

Campinas (SP), 15 de dezembro de 2006.

Prof ª Drª Eliane Martins
(orientadora)

Prof Dr Henrique Santos do Carmo Madeira
(co-orientador)

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

Para minha família
"...pelo apoio, amor e
paciência com minha
ausência"

Agradecimentos

À minha orientadora, Profa. Dra. Eliane Martins, pela amizade, pela paciência, pelo tempo que me dedicou, pelos conhecimentos que comigo compartilhou e principalmente pelo incentivo e apoio durante todo o tempo em que estive envolvida no trabalho de pesquisa e redação desta tese. A ela devo muito, muito mais do que ela possa imaginar.

Ao meu co-orientador Prof. Dr. Henrique Madeira, a quem não tenho palavras para agradecer. Qualquer coisa que dissesse seria muito ínfimo e pequeno frente à imensa ajuda para o desenvolvimento deste trabalho e para o meu desenvolvimento na pesquisa. Ao Henrique, o amigo, queria agradecer por me fazer sentir querida e por partilhar momentos da sua vida familiar. Família que aprendi a admirar e a amar e a quem agradeço com enorme gratidão.

À Profa. Dra. Cecília Mary Fischer Rubira, pelos valiosos conselhos que me auxiliaram a entender melhor diversos aspectos envolvidos nessa pesquisa e pelo incentivo para que prosseguisse neste trabalho.

Aos meus familiares, que me deram estímulo e apoio ao longo do meu trabalho, compreendendo meus momentos de ausência e distanciamento. Quero agradecer em especial às minhas filhas que sempre foram a razão da minha existência e hoje são grandes amigas. Apoiamos-nos umas às outras no caminho que precisamos trilhar. Corajosas mulheres, capazes de grandes feitos. Sempre foram meu orgulho e me deram incentivo para enfrentar todas as dificuldades que a vida me reservou.

Ao amigo Naaliel Vicente Mendes, por ter aprimorado a ferramenta de injeção de falhas para que pudéssemos obter um melhor desempenho nos experimentos. Agradeço ainda por ter me auxiliado diretamente na execução dos experimentos. Além do âmbito profissional, gostaria de agradecer a inestimável amizade que demonstrou por ocasião da minha estada em Coimbra, quando sua presença foi essencial para resolver os trâmites burocráticos, amenizar a solidão, agradecimento que neste sentido estendo à sua esposa Jaqueline.

Ao amigo João Durães que me ajudou a entender muitas partes deste trabalho que só mesmo uma grande capacidade como a dele poderia me fazer entender. Agradeço também pelas suas críticas sempre minuciosas e construtivas com as quais espero poder contar ainda depois de ter concluído este trabalho.

À Critical Software, e em particular ao Ricardo Barbosa, por nos ajudar com o ambiente computacional que utilizamos para os experimentos.

Ao amigo Patrick que tem uma bondade sem tamanho e que abre mão dos seus afazeres para ajudar a um amigo. À amiga Camila que muitas vezes me ajudou a resolver questões das disciplinas e dos artigos. Também à amiga Lenita que me ajudou a obter o equilíbrio neste período tão difícil e ao amigo Sandro pelas revisões. Agradecimento aos amigos do Harpia, Bruno, Ivan e Henrique.

A todos os meus amigos, que muito contribuíram com seus incentivos, muitas vezes acreditando em mim mais do que eu mesma. Em especial aos meus amigos portugueses que me acolheram tão bem na terra distante do velho mundo e aqui não poderia deixar de citar o Raimundo, a Henriqueta e filhos que fizeram muito mais do que seria razoável esperar de um amigo.

À Universidade Estadual de Campinas (UNICAMP), que tem sido parte da metade da minha existência proporcionando todas as oportunidades para o meu desenvolvimento intelectual e profissional.

À Universidade de Coimbra por ter me acolhido e colocado à minha disposição recursos para o desenvolvimento deste trabalho.

Às pessoas que me desafiaram e que muitas vezes me decepcionaram, pois assim me ensinaram a me recuperar de cada tropeço na estrada da vida.

Resumo

A sociedade moderna está cada vez mais dependente dos serviços prestados pelos computadores e, conseqüentemente, dependente do software que está sendo executado para prover estes serviços. Considerando a tendência crescente do desenvolvimento de produtos de software utilizando componentes reutilizáveis, a dependabilidade do software, ou seja, a segurança de que o software irá funcionar adequadamente, recai na dependabilidade dos componentes que são integrados. Os componentes são normalmente adquiridos de terceiros ou produzidos por outras equipes de desenvolvimento. Dessa forma, os critérios utilizados na fase de testes dos componentes dificilmente estão disponíveis. A falta desta informação aliada ao fato de se estar utilizando um componente que não foi produzido para o sistema e o ambiente computacional específico faz com que a reutilização de componentes apresente um risco para o sistema que os integra. Estudos tradicionais do risco de um componente de software definem dois fatores que caracteriza o risco, a probabilidade de existir uma falha no componente e o impacto que isso causa no sistema computacional. Este trabalho propõe o uso da análise do risco para selecionar pontos de injeção e monitoração para campanhas de injeção de falhas. Também propõe uma abordagem experimental para a avaliação do risco de um componente para um sistema. Para se estimar a probabilidade de existir uma falha no componente, métricas de software foram combinadas num modelo estatístico. O impacto da manifestação de uma falha no sistema foi estimado experimentalmente utilizando a injeção de falhas. Considerando esta abordagem, a avaliação do risco se torna genérica e repetível embasando-se em medidas bem definidas. Dessa forma, a metodologia pode ser utilizada como um *benchmark* de componentes quanto ao risco e pode ser utilizada quando é preciso escolher o melhor componente para um sistema computacional, entre os vários componentes que provêem a mesma funcionalidade. Os resultados obtidos na aplicação desta abordagem em estudos de casos nos permitiram escolher o melhor componente, considerando diversos objetivos e necessidades dos usuários.

Abstract

Today's societies have become increasingly dependent on information services. A corollary is that we have also become increasingly dependent on computer software products that provide such services. The increasing tendency of software development to employ reusable components means that software dependability has become even more reliant on the dependability of integrated components. Components are usually acquired from third parties or developed by unknown development teams. In this way, the criteria employed in the testing phase of components-based systems are hardly ever been available. This lack of information, coupled with the use of components that are not specifically developed for a particular system and computational environment, makes components reutilization risky for the integrating system. Traditional studies on the risk of software components suggest that two aspects must be considered when risk assessment tests are performed, namely the probability of residual fault in software component, and the probability of such fault activation and impact on the computational system. The present work proposes the use of risk analysis to select the injection and monitoring points for fault injection campaigns. It also proposes an experimental approach to evaluate the risk a particular component may represent to a system. In order to determine the probability of a residual fault in the component, software metrics are combined in a statistical model. The impact of fault activation is estimated using fault injection. Through this experimental approach, risk evaluation becomes replicable and buttressed on well-defined measurements. In this way, the methodology can be used as a components' risk benchmark, and can be employed when it is necessary to choose the most suitable among several functionally-similar components for a particular computational system. The results obtained in the application of this approach to specific case studies allowed us to choose the best component in each case, without jeopardizing the diverse objectives and needs of their users.

"Depois que conhece uma nova idéia, a mente do homem nunca pode voltar as suas dimensões originais."

(Oliver Wendell Holmes Jr.)

"Os fatos, afinal, e mesmo as teorias, são apenas histórias. É o 'processo' que é a ciência viva, é o que torna a atividade excitante para os que a praticam."

(Hall Hellman)

"Meu mestre, dai-me paciência e tolerância para suportar a crítica agressiva, que tende a destruir meu eterno processo de aprendizado, e a sabedoria para administrar meu ego, quando a crítica construtuva encher-me de vaidade e prepotência".

(Sérgio Longo, 2004)

Sumário

Lista de Tabelas

Lista de Figuras

Parte I

Resumo Estendido em Português

Esta tese é uma coletânia de artigos publicados em vários congressos na área de sistemas que têm como foco principal a segurança no funcionamento (*dependable systems*). Como a maioria dos artigos foi publicada em congressos internacionais que exigem a redação na língua inglesa, optamos por redigir na Parte I, um Resumo Estendido em Português. O objetivo deste resumo é contextualizar o trabalho desenvolvido e mostrar como os diversos artigos colaboram para o objetivo da tese. Também, neste resumo, apresentamos de uma maneira mais completa, os trabalhos relacionados às áreas de conhecimento que de alguma forma colaboraram para que o objetivo desta tese fosse atingido.

Capítulo 1-Introdução

A disseminação dos computadores e os produtos de software que neles são utilizados têm tido um impacto crescente na sociedade moderna. Desde simples aplicações domésticas até as mais complexas, que servem de apoio à decisão de grandes corporações, todas se apóiam nos computadores e no software que está sendo executado. O software tem demonstrado ser um elemento crítico quando se pretende assegurar o correto funcionamento dos sistemas computacionais. Dessa forma, a importância da confiabilidade do software tem crescido e se tornado uma preocupação constante dos desenvolvedores e usuários de produtos de software.

Neste contexto, a validação de um produto de software vem se destacando e se revelando como uma das mais importantes, trazendo modificações nas práticas desta atividade. Mesmo com a crescente importância da fase de teste no ciclo de desenvolvimento do software, a ocorrência de defeitos (*failure* em inglês) na fase operacional dos sistemas computacionais continua sendo um acontecimento comum, desafiando as mais modernas técnicas para tolerar falhas (*fault tolerance* em inglês) em sistemas computacionais. Historicamente têm-se informações que mesmo sistemas críticos, exaustivamente testados, apresentam defeitos [74]. A ocorrência de defeitos traz conseqüências graves, na grande maioria dos domínios de aplicação, podendo causar perdas de bens e vidas humanas.

Um modelo cada vez mais utilizado na indústria de software e particularmente atrativo é o desenvolvimento das partes mais específicas do novo sistema e a reutilização de componentes para suprir as funcionalidades mais gerais. Os sistemas são, então, construídos integrando componentes de terceiros (em inglês, *common off-the-shelf* – COTS ou *off-the-shelf* - OTS) e componentes desenvolvidos internamente. Para lidar com as incompatibilidades que podem surgir nessa integração, os conectores são desenvolvidos para unir componentes que interagem de acordo com a arquitetura planejada para esses sistemas. Esta prática de desenvolvimento apresenta diversas vantagens, podendo ser considerada um dos meios mais eficazes para se lidar com a **alta complexidade** que os

produtos de software apresentam, para **diminuir o tempo** necessário para que esse produto chegue ao mercado e para **reduzir o custo**, atendendo restrições impostas por esse mesmo mercado que se revela cada vez mais competitivo.

Apesar dos benefícios potenciais que esta técnica apresenta, alguns problemas são inerentes a esse modelo de construção de software como: (i) o código fonte de um ou mais componentes pode não estar disponível por ocasião da sua integração no sistema [127]; (ii) o entendimento sobre o componente pode ser limitado, mesmo quando o código fonte está disponível, facilitando a introdução de falhas se modificações se fizerem necessárias [127]; (iii) a integração de componentes desenvolvidos por diferentes organizações pode trazer incompatibilidades arquiteturais (*architectural mismatches*, do inglês), principalmente devido à maneira como os erros em componentes servidores são sinalizadas para seus componentes clientes [101]; (iv) as condições operacionais nas quais o componente foi desenvolvido previamente que diferem das atuais, nas quais o componente está sendo reutilizado, podendo ativar falha de software (*software fault* em inglês) que antes não tinha sido revelada [93] e levando o sistema a apresentar defeitos que não são aceitos pelo novo contexto no qual o sistema se insere.

Na prática, é preciso testar o componente em cada novo ambiente em que ele é reutilizado para assegurar a qualidade e a confiabilidade do sistema e minimizar o risco para o sistema que reutiliza componentes COTS. Na verdade, a reutilização de componentes extensivamente utilizados em outros contextos traz a falsa sensação de que o componente irá funcionar corretamente, ou seja, traz a falsa sensação de confiabilidade do componente colaborando para a sua dependabilidade (*dependability*, em inglês).

Esta tese foca nos meios para se obter esta dependabilidade, particularmente foca na validação de sistemas baseados em componentes utilizando a técnica de injeção de falhas. Injeção de falhas é uma técnica de validação que deliberadamente introduz falha (ou erro) no sistema e acelera sua ativação para observar o comportamento do sistema em presença dessa falha. Nossa abordagem consiste em utilizar o risco para decidir quais partes do sistema devem ser alvos das injeções de falhas (ou erros), visando a **eliminação de falhas** (redução do número e severidade das falhas), como também, avaliar o risco de utilizar e reutilizar componentes para o sistema no qual este componente está integrado, visando a

predição de falhas (estimar a presença, criação e conseqüências das falhas). Na avaliação do risco, a injeção de falhas será utilizada para avaliar o impacto que falhas residuais em um componente podem causar no sistema e será completada por um modelo estatístico para estimar a probabilidade de haver falhas residuais.

1.1. Terminologia

Ainda não há um consenso sobre a terminologia a ser utilizada no campo de Tolerância a Falhas na língua portuguesa. Já ocorreram várias discussões a esse respeito nos Workshops e Simpósios feitos na área, porém um padrão ainda não foi adotado. Para um melhor entendimento do conteúdo deste trabalho, antes de prosseguir, estamos introduzindo alguns termos relevantes. Os termos relacionados a seguir são utilizados com os seguintes significados, baseado em Leite e Orlando [44], Avizienis *et al*. [5] e Laprie [41]:

- Falha (fault): é o componente, tanto de software como de hardware, que apresenta comportamento anômalo dentro do sistema.
- Erro (error): sendo encontrada uma falha dentro de um dado processamento do sistema, essa falha levará a uma modificação no estado do sistema. Esse estado causado pela falha é denominado erro.
- Defeito (failure): tendo a falha levado a um erro, esse erro por sua vez fará com que o sistema apresente um defeito. Um sistema apresenta um defeito quando o serviço que deveria oferecer não está de acordo com o que tinha sido especificado.
- Validação: a validação da dependabilidade é um processo no qual se avalia um software durante ou após o desenvolvimento, para determinar se o produto satisfaz os requisitos.
- Benchmark: representa um acordo amplamente aceito pela indústria de software e pela comunidade dos usuários dos produtos de software.
- Dependabilidade (dependability): dependabilidade de um sistema computacional é a habilidade deste sistema entregar serviços que justificadamente possam ser confiáveis. São atributos da dependabilidade a disponibilidade, confiabilidade, segurança (safety), confidencialidade, integridade e manutenibilidade.

- Disponibilidade: caracteriza a disponibilidade do serviço correto tal como é percebido pelo usuário.
- Confiabilidade: caracteriza a continuidade do serviço sob a forma de tempo entre defeitos.
- "*Safety*": caracteriza o sistema quanto à gravidade dos defeitos. Um sistema é seguro quando não apresenta defeitos catastróficos.
- Confidencialidade: caracteriza a capacidade do sistema em manter oculta a informação considerada confidencial.
- Integridade: caracteriza a capacidade do sistema em manter a coerência da informação.
- Manutenibilidade: indica a capacidade do sistema em aceitar modificações e atualizações.
- "*Security*": caracterizada pela existência concorrente da disponibilidade somente para usuários autorizados, da confidencialidade e da integridade.
- Tolerância a falhas: capacidade do sistema continuar a prover o serviço esperado mesmo em presença de falhas.
- Prevenção de falhas: compreende metodologias destinadas a prevenir a introdução de falhas durante o desenvolvimento do sistema.
- Eliminação de falhas: compreende métodos de engenharia aplicados durante o desenvolvimento de forma a minimizar a existência de defeitos.
- Predição de falhas: tem por objetivo estimar as conseqüências da ativação de possíveis falhas que existam no sistema.
- Teste de software: processo que exercita um software para verificar se ele satisfaz os requisitos especificados e para detectar falhas.
- Teste de robustez: processo que exercita um software para verificar se ele opera corretamente em presença de entradas inválidas ou quando colocado em um ambiente hostil.
- Teste de interface: tipo de testes de robustez que usa a interface do sistema para prover valores válidos e inválidos com o objetivo de verificar se o sistema opera corretamente.

- Injeção de falhas: técnica de validação que usa a introdução deliberada de falha (ou erro) no sistema e acelera sua ativação para observar o comportamento do sistema em presença dessa falha (ou erro).
- *"Workload"*: especifica as tarefas que são submetidas ao sistema e representam o trabalho típico para uma classe específica de sistema.
- *"Faultload"*: especifica quais as falhas que vão ser injetadas no sistema para simular a existência de falhas.
- Modo de Defeito (*failure mode*): expressam a maneira como o sistema pode apresentar defeito.

1.2. Relevância da avaliação do risco de utilizar e reutilizar componentes

Num cenário em que componentes COTS são cada vez mais utilizados para compor sistemas mais complexos, as falhas residuais e as complexas interações entre componentes COTS e outras partes do sistema representam um risco crescente para o software [24]. Sendo assim, a indústria precisa urgentemente encontrar uma maneira para estimar e reduzir o risco de usar componentes COTS nos sistemas por ela produzidos, como também, definir uma maneira prática para se escolher, entre diversos componentes que provêem a mesma funcionalidade, aquele que represente o menor risco quando integrado em um sistema específico.

Apesar dos riscos que a reutilização de componentes oferece para o sistema como um todo [93, 101, 127], esta tendência deve permanecer nos ambientes de desenvolvimento, pois a opção seria desenvolver internamente, para cada ambiente específico, todos os componentes necessários, o que é muito mais dispendioso e nem sempre atinge níveis melhores de qualidade. A eliminação de todos os defeitos de um software durante o processo de desenvolvimento é um objetivo praticamente impossível de ser alcançado, o que se verifica também no processo de desenvolvimento de componentes COTS [80, 83]. Sendo assim, a indústria de software admite que os sistemas computacionais tenham falhas de software e que, essas falhas trazem um risco para a organização.

O risco associado às perdas causadas pela ativação de falhas residuais, também é, freqüentemente, utilizado como critério para determinar em que momento os testes do sistema devem ser finalizados [90]. A análise do risco está quase sempre ligada ao

gerenciamento do risco [77, 81] e relaciona a estimativa do risco com modelos de qualidade de desenvolvimento de software, heurísticas ou experiências de desenvolvedores [69, 88, 90].

Em algumas áreas de aplicação, tais como controle de vôos ou controle de estações nucleares, a estimativa do risco está fortemente relacionada com *safety* e avaliação de confiabilidade e é regulamentada por rigorosos padrões da indústria [75]. A análise do risco baseada na arquitetura do software também é bastante utilizada especialmente para se estimar o risco nas primeiras fases do desenvolvimento do software [103, 105, 110].

Apesar da quantidade de pesquisas sobre a análise de risco [69, 88, 90], a estimativa do risco de se utilizar ou reutilizar um componente em um sistema mais complexo ainda é um problema difícil de resolver. Um fator que contribui para esta dificuldade é a avaliação de aspectos relacionados com o comportamento dinâmico do componente, que só podem ser feitos por meio de uma avaliação experimental utilizando o componente ou um protótipo do componente. Outro fator de contribuição é a estimativa de falhas residuais que se baseia em modelos estatísticos, trazendo incertezas inerentes às estimativas.

Nesta tese, utilizamos a análise e a avaliação de risco. Os aspectos relacionados com o comportamento dinâmico do componente são avaliados utilizando a injeção de falhas. Dessa forma, os seguintes fatores contribuem para a dificuldade de analisar e avaliar um componente quanto ao risco, usando injeção de falhas:

- **A dificuldade de compreender as falhas de software**. Muitas vezes defeitos são descobertos pelos usuários durante a utilização do software já no ambiente operacional. Por um processo de retrocesso, a falha é descoberta. Porém, muitas vezes não há uma causa única ou não é possível identificá-la.
- **A dificuldade de escolher os locais onde injetar falhas**. Um produto de software moderno é composto por um grande número de componentes e durante uma campanha de injeção de falhas é difícil saber qual o melhor componente a ser injetado e qual critério deve ser utilizado para a escolha.
- **A dificuldade de prever a propagação de erros em decorrência da ativação de uma falha.** Uma falha em um componente pode alterar o comportamento de

outros componentes que pertençam à sua cadeia de dependência. Muitas vezes, o componente que apresenta o defeito não contém uma falha, mas é influenciado por erros advindos de outros componentes via propagação.

- **A dificuldade de identificar a representatividade das falhas**. Para efeito de avaliação do risco (obtenção de medidas) e *benchmark* quanto ao risco é importante que sejam entendidas as características de falhas que permanecem ocultas após a fase de teste do sistema. Esse entendimento, muitas vezes, depende de informação sobre falhas que foram descobertas em campo e que geralmente não se tornam públicas. Particularmente para falhas que emulamos através da injeção de falhas de interface, esta representatividade não é clara.
- **A escassez de ferramentas para a injeção de falhas internas**. Ferramentas que injetem falhas nas estruturas internas do software, emulando uma falha residual, ainda são escassas e de difícil utilização.

E ainda:

- **A dificuldade de estimar a probabilidade de existirem falhas residuais em componentes do sistema**. Modelos para se estimar falhas residuais são pouco utilizados, de difícil validação e normalmente baseados em informações de campo que nem sempre estão disponíveis.

1.3. Objetivos da Tese

A avaliação experimental é uma técnica largamente difundida para a avaliação e validação de sistemas computacionais [3, 31]. Uma das técnicas mais comuns para se avaliar experimentalmente um produto de software é a injeção de falhas. O uso de injeção de falhas para avaliar experimentalmente o impacto dos defeitos em sistemas computacionais vem sendo utilizado por muito tempo [3, 31, 32]. Este impacto é geralmente descrito utilizando modos de defeitos (*failure modes*) que expressam a resposta do sistema quando uma falha é injetada.

Modelos estatísticos baseados em regressão têm sido utilizados para estimar a propensão a falhas de um componente [70, 84] e validam o uso das métricas orientadas a

objetos como um indicador para a previsão de classes propensas a falhas, combinando estas métricas por meio de um modelo estatístico.

O objetivo principal da tese é o uso da **análise de risco** para selecionar os pontos de injeção e monitoração durante uma campanha de injeção de falhas e a **avaliação de risco** que o uso de um componente de software, que está integrado a um sistema computacional, oferece para este sistema. Nossa proposta é obter esta avaliação experimentalmente, combinando a probabilidade de haver uma falha residual no componente e a gravidade do impacto que a ativação desta falha causará no sistema.

1.4. Avaliação Experimental do Risco de uso de um Componente para o Sistema

A avaliação experimental do risco da utilização e reutilização de um componente em um determinado sistema tem sido pouco explorada na literatura. Vários trabalhos apresentaram uma maneira de avaliar o risco para produtos de software [68, 88, 90]. Na maioria destes trabalhos, a equação utilizada para se estimar o risco é basicamente a mesma e se baseia na probabilidade de um componente de software apresentar um comportamento não adequado e o impacto (ou custo) que este comportamento causa no sistema. Porém, a equação é interpretada de maneira diferente dependendo da abordagem utilizada.

Na equação apresentada por Rosenberg *et al.* [88] o risco é calculado considerando a probabilidade que um evento indesejado E_i aconteça ($p(E_i)$) e o custo para o sistema se este evento realmente ocorrer ($c(E_i)$). No contexto da estimativa de risco em sistemas de software, um evento indesejado é um defeito em um componente de software. A equação (1) apresenta a equação proposta no trabalho de Rosenberg:

$$\text{Risco} = \sum(p(E_i) * c(E_i)) \qquad (1)$$

O primeiro termo da equação (1) $p(E_i)$ representa a probabilidade de haver uma falha residual em um componente, ou seja, representa a propensão a falhas de um determinado componente. Nossa proposta é estimar esta probabilidade utilizando as métricas de complexidade do componente combinando estas métricas por meio de um modelo estatístico baseado em regressão logística [116].

Embora a injeção de falhas tenha sido amplamente utilizada para avaliar experimentalmente o impacto dos defeitos em sistemas computacionais [3, 31], não se

encontra na literatura o seu uso para estimar o risco; em particular, para estimar o risco que um componente de software traz para o sistema no qual ele está integrado. O impacto (ou custo) da ativação de uma falha é representado pelo segundo termo da equação (1), $c(E_i)$. Nossa proposta é avaliar este impacto de maneira experimental através da injeção de falhas de software no componente sob teste. Na avaliação do custo através da injeção de falhas leva-se em consideração a probabilidade de uma falha ser ativada, a probabilidade do comportamento do componente sofrer um desvio e a conseqüência do defeito observado no sistema.

Assim, o risco de se utilizar ou reutilizar um componente em um sistema pode ser estimado como sendo o resultado da multiplicação da probabilidade de haver falha residual no componente obtida pelas métricas de complexidade e do impacto observado em decorrência da ativação desta falha, emulada pela injeção de falhas de software no componente.

1.5. Contribuições do Trabalho

O objetivo da tese é proporcionar uma maneira de **analisar o risco para selecionar os pontos de injeção e monitoração** durante uma campanha de injeção de falhas e **avaliar o risco que o uso de um componente de software oferece para o sistema** no qual esse componente está integrado. A avaliação do risco implica a previsão da probabilidade de haver uma falha residual no componente e a gravidade do impacto que a ativação desta falha irá causar no sistema. Os resultados parciais das pesquisas desenvolvidas para atingir este objetivo, proporcionaram outras contribuições:

- Um estudo sobre a propagação de erros através das interfaces de componentes quando uma falha (ou um erro) é injetada no componente ou na aplicação que interage com o componente.
- Um estudo sobre a importância do risco de um componente para a seleção de locais onde os erros devem ser injetados e os experimentos monitorados durante uma campanha de validação de um sistema através da injeção de falhas.
- Um estudo sobre a importância da arquitetura do sistema para reduzir a quantidade de injeções de erros necessárias para a validação de sistemas computacionais.

- Um estudo sobre as limitações das falhas de interface para avaliar experimentalmente o risco de se utilizar um componente que é parte de um sistema computacional.
- Um estudo de campo para validar um modelo estatístico que tem como objetivo estimar a probabilidade de haver uma falha residual em um componente de software.
- Uma maneira de se fazer uma distribuição representativa das falhas a serem injetadas numa campanha de injeção de falhas.
- Uma metodologia para a avaliação experimental do risco que o uso de componentes traz para um sistema de software.
- Uma maneira de escolher o melhor componente para ser integrado em um sistema específico.
- Um exemplo de avaliação do risco para dois sistemas operacionais largamente utilizados em ambientes computacionais. *Real-Time Operating System for Multiprocessor Systems* (RTEMS) e o *Kernel* do Linux (RTLinux) foram avaliados quanto ao risco, seguindo a metodologia proposta.

Além destas contribuições que foram experimentalmente verificadas, fizemos um estudo preliminar para entender a utilidade da avaliação proposta para a **certificação de componentes quanto ao risco**. A certificação de software quanto ao risco deve atestar que o produto alvo da certificação apresenta um risco dentro de limites estipulados por um contrato ou estabelecidos por um padrão de certificação, como por exemplo, os padrões ISO 9126 [117] ou ISO/IEC 25051 [118]. A certificação de componentes tem sido apontada como uma atividade necessária para o futuro do desenvolvimento de software baseado em componentes [114, 128].

1.6. Organização do texto

O presente trabalho está organizado da seguinte forma:

A Parte I apresenta o contexto geral da tese. Dentro da Parte I, o capítulo 1 apresenta um contexto, nossa motivação, objetivos e contribuições.

O capítulo 2 apresenta os “Conceitos Fundamentais e Trabalhos Relacionados”. Na Seção 2.1 são apresentados conceitos e trabalhos relacionados com componentes de software e suas implicações quando os componentes são integrados para compor um sistema. Conceitos e trabalhos relacionados com arquitetura de software e dependências arquiteturais são apresentados na Seção 2.2. Na Seção 2.3. são apresentados os conceitos e trabalhos relacionados com testes de robustez. A injeção de falhas, suas abordagens e ferramentas são apresentadas na Seção 2.4. A Seção 2.5. apresenta os trabalhos relacionados com *benchmark* de dependabilidade. Uma visão geral dos trabalhos sobre análise de risco tal como usada em testes de software é abordada na Seção 2.6. Finalmente a Seção 2.7 apresenta conceitos e trabalhos relacionados que embasaram a avaliação do risco.

O capítulo 3 apresenta “Análise de Risco para Injeção de Falhas”. A Seção 3.1 apresenta a “Análise do Risco com base na Complexidade dos Componentes”, mostrando os conceitos e os trabalhos apresentados sobre métricas de complexidade, particionamento baseado em amostragem estatística, injeção de erros nas interfaces para a avaliação do impacto da complexidade no risco dos componentes. A Seção 3.2 apresenta a utilização das métricas de complexidade para definir partições de equivalência. A Seção 3.3 apresenta a “Análise de Risco na perspectiva da Arquitetura de Software”, apresentando como a arquitetura de software pode ser utilizada para auxiliar a técnica de injeção de falhas.

O capítulo 4, “Avaliação Experimental do Risco”, apresenta nossos trabalhos direcionados para a obtenção de medidas do risco que um componente representa para o sistema do qual ele faz parte. A Seção 4.1 apresenta uma “Comparação de Falhas de Interface e Falhas Internas” e as limitações das duas abordagens. A Seção 4.2 apresenta a “Avaliação de Risco utilizando Falhas Internas” baseando-se em estudos sobre a estimativa de densidade de falhas e do impacto das falhas ativadas.

O capítulo 5 apresenta nossas “Discussão e Conclusões” apontando alguns trabalhos futuros.

A Parte II – “Artigos Publicados” apresenta os artigos que foram publicados no decorrer do desenvolvimento do trabalho e que foram referenciados nas seções deste Resumo Estendido.

No Apêndice A podem ser encontrados os conceitos estatísticos utilizados nesta Tese.

Capítulo 2–Conceitos Fundamentais e Trabalhos Relacionados

O objetivo deste capítulo é apresentar os conceitos fundamentais e trabalhos relacionados com os assuntos envolvidos nesta tese. Embora os artigos que compõem a tese já apresentem alguns trabalhos relacionados com os assuntos específicos que abordam, a limitação do espaço imposta pelo tamanho dos volumes impressos não permite que a lista dos trabalhos relacionados apresentada nos artigos seja completa.

2.1. Componentes de Software

Componentes de software são unidades de software desenvolvidas para atender a um propósito específico, podendo ser utilizados em várias aplicações que precisam implementar a mesma funcionalidade. Um componente tem interfaces bem definidas, tanto para suas dependências (interfaces requeridas) quanto para os serviços oferecidos (interfaces providas).

Segundo Szyperski [108], o desenvolvimento baseado em componentes propõe que o sistema seja estruturado em "unidades de composição com interfaces especificadas por meio de contratos e dependências de contexto explícitas. Um componente de software pode ser instalado independentemente e utilizado para a composição de sistemas de terceiros". Esta estrutura favorece a reutilização, já que o acoplamento entre componentes é minimizado.

Segundo Whitehead [130], componentes são unidades independentes de software que encapsulam seu projeto e implementação, ou seja, são componentes de software auto-contidos. Desta forma, um componente desenvolvido por uma equipe pode perfeitamente ser utilizado por outra equipe de desenvolvimento ou, até mesmo, fazer parte de sistemas de outras empresas que apresentam soluções para outros contextos de uso. Esse emprego diversificado de um mesmo componente pode apresentar um risco para as aplicações que o

integram, uma vez que um produto de software é sensível ao ambiente e ao contexto de utilização [93].

Componentes desenvolvidos por terceiros (em inglês, *Off-the-Shelf* – OTS ou *Common Off-The-Shelf* – COTS) têm sido utilizados em larga escala como parte dos sistemas computacionais modernos [8]. Devido à complexidade e à diversidade de conhecimentos envolvidos no desenvolvimento de sistemas, as equipes estão se tornando cada vez mais especializadas em um determinado segmento de desenvolvimento de software. Essa segmentação tem exigido que os sistemas sejam construídos como uma composição de diversos componentes advindos de outras empresas ou de outras equipes dentro da mesma empresa.

Um sistema baseado em componentes integra diversos componentes que interagem entre si para prover as funcionalidades do sistema. Para Weyuker [93] a utilização de componentes possibilita o desenvolvimento a um custo mais baixo, mas não garante a melhoria de qualidade exigida para a construção de sistemas confiáveis. Embora estes componentes tenham sido utilizados em outros sistemas computacionais eles devem ser amplamente testados, pois podem se comportar de maneira distinta em cada contexto em que são utilizados.

Além de serem testados a cada reutilização, a realização de testes nestes componentes apresenta algumas dificuldades adicionais, que são causadas pela falta de conhecimento de aspectos importantes [111]: por um lado o fornecedor não conhece os diferentes contextos em que os componentes podem ser empregados, assumindo algumas hipóteses para que possa desenvolvê-lo e testá-lo. O cliente, por outro lado, só conhece e só tem acesso à interface pública do componente o que representa um problema para testar o componente.

O uso de componentes terceirizados tem se estendido para ambientes críticos, que colocam em risco vidas humanas ou investimentos de grande monta. Como exemplo, podemos citar o acidente com o Therac-25, máquina de irradiação para terapia que causou a morte de três pessoas e sérios ferimentos em outros pacientes devido ao excesso de irradiação [74]. A explosão do foguete Ariane-5, cujo desastre foi causado pela conversão mal sucedida de dados [87]. A queda de aeronave americana na Colômbia em 1995, causando a morte de 159 pessoas e de uma outra aeronave coreana, em 1997, devido à falha

em radar, causando a morte de 225 pessoas [74]. Dessa forma, utilizar componentes como parte de um sistema de software sempre apresenta um risco para o sistema.

2.2. Arquitetura de Software e Dependências Arquiteturais

Arquitetura de software pode ser definida como uma representação do sistema, baseado nos componentes que o integram e suas interações [105]. O diagrama da arquitetura é um modelo de alto nível do sistema que representa seus componentes e como eles se interconectam.

A existência de um número muito grande de interfaces entre os componentes de software acaba aumentando a complexidade do projeto. Componentes especiais, os conectores, têm sido freqüentemente utilizados para resolver problemas de incompatibilidade que surgem entre as interfaces de componentes que se interconectam. A incompatibilidade da arquitetura (*architectural mismatch*) acontece quando a mensagem esperada por um componente não é compatível com a mensagem enviada a ele por outros componentes ou pelo ambiente no qual ele está inserido [97].

Soluções arquiteturais têm sido utilizadas para a obtenção do nível de qualidade esperado dos produtos de software modernos. Para a obtenção de software mais confiáveis, diversas pesquisas foram desenvolvidas para estudar a influência da arquitetura [104, 106, 109]. A maioria delas aborda o problema da dependência, que é causado pelos relacionamentos complexos inerentes à programação orientada a objetos, tais como, heranças, agregações, associações, polimorfismo, ligações dinâmicas (*dynamic binding*).

A análise de dependência tem sido tradicionalmente baseada em fluxo de dados associado às funções e variáveis de programas [107]. A sua utilização é importante para as atividades da engenharia de software, tais como, entendimento de programas, testes, engenharia reversa e manutenção [96]. Também é útil para revisões de código e para avaliar o acoplamento em uma aplicação ou uma biblioteca.

Surgiram diversas abordagens para se analisar as dependências com base na arquitetura [94, 99, 102]. Uma limitação da abordagem tradicional é que normalmente ela precisa do código fonte para ser avaliada o que dificulta o seu uso para componentes OTS para os quais o código fonte pode não estar disponível. Além disso, a análise estática não

pode lidar com questões de polimorfismo que só são reveladas durante a execução do sistema.

No estudo publicado por Kung *et al*. [100] foi definido o conceito de um *firewall* de dependência. O *firewall*, segundo Kung, identifica as mudanças e identifica as classes afetadas pelas mudanças efetuadas em uma determinada classe. Esse conceito pode auxiliar na análise das falhas e definir até que ponto a cadeia de dependência permite a propagação de erros. O ambiente para teste e manutenção de software orientado a objeto proposto por Kung inclui diversos diagramas: *object relation diagram* (ORD), *block branch diagram* (BBD), *object state diagram* (OSD) que são extraídos do código por meio da engenharia reversa. O problema dessa abordagem é que quando não se tem acesso ao código fonte não é possível identificar os relacionamentos de polimorfismos e encadeamentos dinâmicos que são determinados em tempo de execução.

A abordagem apresentada por Stafford *et al.*, denominada *Chaining* [107], tem como objetivo reduzir a extensão da arquitetura do sistema que deve ser examinada quando, por exemplo, se pretende testar o sistema. Nesta abordagem, uma ligação associa componentes da arquitetura que estão diretamente ligados e incluem ligações de componentes que estão indiretamente ligados. A técnica considera que os seguintes elementos podem compor um componente: (i) elementos de interação (provê significado para conexões ou interações com outro componente); (ii) elementos de dados (contém o dado a ser transformado, provido ou utilizado pelo componente); (iii) elementos de processamento (provê significado para transformação, ou transmissão de dados e/ou estado do componente); (iv) elementos de estado (contém o estado corrente do componente em termos de elementos de dados e de processamento).

Existem três tipos de ligações: (i) *affected-by chain* que é composto pelo conjunto de componentes que podem potencialmente afetar o componente sob teste; (ii) *affects chain* que é composto pelo conjunto de componentes sobre os quais o componente sob teste pode ter algum efeito; (iii) *related chain* que combina os dois tipos de ligação, *affected-by* e *affects chain*. Os relacionamentos guardam uma transitividade, ou seja, uma vez identificado um relacionamento, todos os "chain" do componente relacionado compõem a cadeia de "chain" do componente sob teste. A Figura 1 apresenta esse conceito. Nesta

figura, o componente sob teste apresenta cinco elementos em seu *affected-by chain* e quatro elementos em seu *affects chain*.

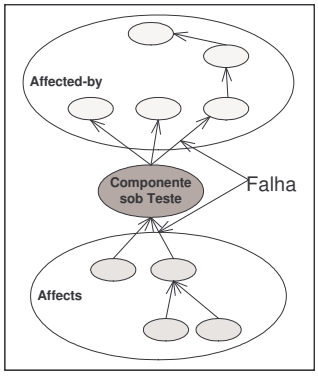


Figura 1: *Chain* de um Componente

Particularmente para a validação por injeção de falhas, o conceito de *chain* pode auxiliar na seleção das interfaces que devem ser injetadas, no estabelecimento dos pontos de monitoração dos experimentos e para definir o conjunto mínimo de componentes que precisam ser analisados caso um defeito ocorra. Para priorizar a injeção das falhas o número de *related chain* pode ser utilizado como um critério de complexidade e, consequentemente, um indicativo de que este componente deve ser alvo da injeção de falhas, por afetar um maior número de componentes do sistema. Além disso, outras análises podem ser feitas, falhas injetadas na interação do componente sob teste e os elementos no seu conjunto de *affects chain* podem impactar os componentes sucessores do componente sob teste, falhas injetadas na interação do componente sob teste e os elementos no seu conjunto de *affected-by chain* simulam falhas nos componentes predecessores e podem impactar o componente sob teste.

2.3. Testes de Robustez

Testes de robustez têm como objetivo assegurar que o sistema pode lidar com entradas inesperadas ou defeitos advindos de outros sistemas ou do ambiente computacional [15]. 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 [66].

Os testes de robustez são úteis para exercitar os mecanismos de tratamento de exceções implementados no sistema. No trabalho de Mukherjee e Siewiorek [50] é apresentada uma visão geral dos conceitos envolvidos nos testes de robustez.

Os testes de robustez podem assumir diferentes nomes sendo que alguns desses tipos de testes são equivalentes e outros representam um tipo específico de testes de robustez [15]. Por exemplo, *Ad hoc* testing é um tipo de teste de robustez no qual são exercitados casos de testes de maneira aleatória com o objetivo de causar um defeito catastrófico no sistema. *Ad hoc* testing inclui *Negative Testing* que é um tipo de teste que tem como objetivo mostrar que o software não funciona bem.

Um dos tipos de teste de robustez bastante utilizados são os testes de limites (*Boundary value analysis/testing*). Este tipo de teste tem seu foco nos limites de domínios estabelecidos pela especificação. Isso quer dizer que, por exemplo, se uma função espera receber valores inteiros em um faixa de 100 a 1000, os valores que devem ser fornecidos ao sistema são 99 e 1001, que são os valores inválidos mais próximos dos limites do domínio estabelecido pela função [15, 39].

Outro tipo de teste de robustez utilizado com freqüência é o *Stress testing* que avalia um sistema ou componente sob carga elevada para determinar qual é o limite de carga que o sistema suporta [15].

Do ponto de vista de testes para avaliar a robustez de componentes, testes de interface são bastante úteis, podendo considerar:

- que falhas internas ao componente se manifestam na interface (modos de defeitos).
- que modos de defeitos são visíveis pelos componentes que interagem com o componente alvo.
- que podemos assim emular modos de defeitos do componente alvo e componentes que com ele interagem.
- que pela interface é que se propagam (a maioria) os erros [19].
- que na validação de componentes terceirizados, essa pode ser uma das poucas maneiras possíveis de testá-los em presença de falhas ou erros.

Um guia para os testes de interface pode ser encontrado em Cohen *et al.* [15] que indica os defeitos relacionados com a interface que devem ser validados. Resumidamente,

incluem problemas com redimensionamento de janelas, apresentação de janelas e com entrada de dados.

Particularmente o teste das entradas inválidas assegura que um valor fora da especificação não é aceito. Na maioria dos casos é muito difícil testar todas as entradas possíveis e algumas entradas representativas devem ser escolhidas. O teste utilizando valores inválidos é uma parte importante para os testes de interface e pode ser auxiliado pela técnica de injeção de falhas.

2.4. Injeção de Falhas

O uso de injeção de falhas para avaliar experimentalmente o impacto (ou custo) de defeitos (*failure* em inglês) em sistemas computacionais tem sido muito freqüente [3, 31]. Injeção de falhas procura reproduzir ou simular a presença de falhas e observar o sistema para verificar qual será sua resposta nessas condições [2]. Na validação de um sistemas é importante conhecer o comportamento deste sistema, quando possíveis falhas (*fault* em inglês) nele existentes são ativadas. Porém, esperar pela ativação natural destas falhas não é viável, uma vez que essa ativação raramente acontece (caso contrário, as falhas seriam facilmente encontradas e eliminadas na fase de teste). A injeção de falhas é um mecanismo útil para acelerar a ativação das falhas.

Ao acelerar a ocorrência de erros (*errors* em inglês) e defeitos, através da injeção de falhas, a técnica se revela como uma valiosa abordagem para validar sistemas nos quais *safety* é um requisito indispensável, avaliar o impacto da detecção desses erros como também o impacto dos mecanismos de recuperação de erros sobre o desempenho do sistema. É uma técnica largamente utilizada para avaliar a dependabilidade de sistemas computacionais [41].

Injeção de falhas pode ser utilizada para eliminar falhas (*fault removal*) ou para predizer falha (*fault forecasting*). Quando o objetivo é **eliminar falhas**, a técnica é utilizada para localizar e eliminar o maior número possível de falhas de projeto / implementação do sistema. Nesse caso, requer um pequeno número de falhas que são determinadas a partir das hipóteses feitas na fase de projeto do sistema. As entradas são destinadas a exercitar as funções do sistema e ativar as falhas injetadas. Os resultados indicam a ocorrência de eventos e/ou medidas de tempo associados ao experimento. Já para a **predição de falha** o

objetivo é obter medidas da eficiência dos mecanismos de tolerância a falhas do sistema. Nesse caso, um número elevado de falhas representativas das falhas passíveis de ocorrer em fase operacional deve ser injetado. Sendo assim, as entradas dependem do perfil operacional do sistema e os resultados se traduzem em um número de ocorrências de eventos e o tempo entre as ocorrências desses eventos.

O impacto de defeitos quando se utiliza injeção de falhas é geralmente descrito por meio de modo de defeitos (*failure modes*) que expressa a resposta do sistema a cada falha injetada (por ex., saídas inválidas, término do sistema de maneira inesperada, etc.).

Na verdade, para se aplicar a técnica de injeção de falhas é preciso que se definam quatro conjuntos: o conjunto F que define as falhas a injetar; o conjunto A que define o domínio de ativação do sistema e especifica a funcionalidade do sistema que deve ser exercitada; o conjunto R que especifica as observações (*readouts*) a efetuar e quais as características do sistema que interessa observar; o conjunto M que especifica o modelo que se deve aplicar às observações do conjunto R de modo a obter um conjunto de medidas com significado prático [1, 2]. Mais recentemente, o conjunto F é referido como *faultload,* o conjunto A como *workload* e o conjunto M como modo de defeitos (*failure mode*).

Um modelo de falhas inclui a informação a respeito das ações orientadas à reprodução de falhas de uma determinada classe. A *faultload* é a especificação de um conjunto de falhas pertencentes a um determinado modelo de falhas e pode incluir regras de injeção, descrevendo parâmetros tais como: onde, quando e quantas vezes as falhas são injetadas [31]. A *workload* pode ser classificada como real (tarefas reais submetidas pelos usuários do sistema), realística (tarefas mais representativas do sistema) ou sintética (permitem ativar de forma controlada funcionalidades específicas).

Na visão de Tsai *et al.* [60], as *workloads* reais causam a ativação do sistema de uma forma generalizada, enquanto que, as *workloads* sintéticas, por não serem focadas na representação típica do sistema, podem ser definidas com o propósito de ativar partes (caminhos) bem definidas do sistema. As experiências que utilizam *workloads* reais são designadas por Tsai como *stress-based injection*, enquanto que as experiências que utilizam *workloads* sintéticas são designadas como *path-based injection*. Neste último tipo, a ativação das falhas injetadas é mais eficiente, uma vez que as falhas apenas serão injetadas na parte do sistema que se sabe de antemão que será executada.

2.4.1. Injeção de Falhas de Hardware

Os primeiros injetores foram implementados em hardware e projetados para injetar falha ou erro de hardware [2, 16, 36, 45]. Habitualmente designados por ferramentas HWIFI – *Hardware Implemented Fault Injection*, estas ferramentas apresentam um modelo bastante simples, usando a inversão de bits ou a modificação do valor que se encontra em uma determinada posição de memória. O modelo de falhas das ferramentas HWIFI é bastante restrito pela tecnologia em que são implementados. Além disso, é preciso que se tenha acesso a pontos de contatos no interior dos componentes onde se pretende injetar as falhas, o que foi dificultado com a diminuição do tamanho dos componentes e o aumento da escala de integração dos circuitos. Como alternativa surgiram os injetores de falhas implementados em software (SWIFI – *Software Implemented Fault Injection*), pois envolvem menor custo e são mais versáteis.

Atualmente as ferramentas SWIFI são dominantes no cenário de injeção de falhas. Além disso, nos sistemas atuais, as falhas de software são mais relevantes devido à complexidade inerente destes sistemas [66]. Por estes motivos, o foco deste trabalho é voltado para a injeção de falhas por software e as ferramentas HWIFI não serão mais citadas.

2.4.2.Emulação de Falhas de Hardware por Software (SWIFI)

Injetores SWIFI são implementados através de programas e, dessa forma, permitem que sejam utilizados modelos de falhas mais elaborados, podendo utilizar algoritmos complexos para a injeção de falhas. Inicialmente, os injetores SWIFI foram utilizados para a injeção de falhas de hardware.

O trabalho apresentado em Madeira *et al.* [46] investiga a hipótese de se injetar falhas de software utilizando ferramentas SWIFI, através da injeção de erros que emulam falhas de software com base no trabalho de Christmansson e Chillarege [14]. O trabalho de Madeira concluiu que as ferramentas SWIFI tradicionais tinham a capacidade de injetar um modelo simples de falhas de software, apresentando um diferencial importante sobre as ferramentas HWIFI que não apresentavam esta capacidade.

Considerando a emulação de falhas por injeção de erros nos dados, a ferramenta FERRARI [34, 35] injeta erros por meio do *ptrace* do sistema Unix antes da execução do

programa no sistema alvo ou por meio da modificação de posições de memória e de registros do processador durante a execução. FIAT [55] consegue injetar erros por meio de rotinas especiais que são ligadas ao código alvo e, por esse motivo, depende da disponibilidade do código objeto do sistema alvo. O modelo de falhas inclui corrupção e atrasos em mensagens, atrasos e terminação abrupta de tarefas. A ferramenta DOCTOR [28] injeta erros através da corrupção ou atraso de mensagens e a corrupção de dados na memória do sistema alvo.

A ferramenta DTS [61] injeta erros nos parâmetros das chamadas às bibliotecas do sistema operacional Windows NT. A corrupção de valores é feita por interceptação das chamadas que são efetuadas pela *workload*.

A ferramenta FTAPE [58, 60] injeta erros através da manipulação dos valores presentes nos registros do processador e em posições de memória. A ferramenta Xception [11] utiliza as características de depuração e aferição de desempenho que estão disponíveis nos processadores modernos para injetar erros. Utiliza as exceções dos processadores para disparar a injeção de falhas. A injeção de falhas é implementada através dos tratadores de exceção do kernel. O modelo de erros inclui a modificação dos valores presentes nos registros do processador, em posições de memória, ou presentes no BUS. Permite que se injetem falhas permanentes e transientes e tem um impacto pequeno no desempenho do sistema, uma vez que não exige que o sistema opere em modo de *debug*. Inicialmente desenvolvida para injetar falha de hardware, pode ser utilizada para injetar um modelo simples de falhas de software.

2.4.3. Injeção de Falhas de Interface

Injeções de falhas de interface emulam falhas externas que representam todos os fatores externos que não estão relacionados com falhas internas, mas alteram o estado do software de algum modo [66]. A injeção de uma falha de interface altera o estado do sistema alvo (normalmente seus dados), uma vez que podem emular o efeito de uma possível falha de software ou emular uma falha de hardware. São também consideradas ferramentas SWIFI, mas são mais voltadas para os testes de robustez.

As falhas mais comuns nessa categoria são aquelas que representam conseqüências de falhas de hardware, como valores de registradores da CPU ou modificações em posições

específicas da memória. Como estudos têm mostrado que falhas de software são causas importantes de defeitos de sistemas computacionais em operação, manifestações de falhas de software no nível de máquina ou mais alto têm sido levadas em consideração.

A injeção de falhas de interface (erros) utilizando valores excepcionais ou inválidos através das interfaces dos componentes tem sido útil para testar a robustez de produtos de software [15], revelando pontos fracos dos sistemas operacionais [22, 37, 38, 50] e a maneira de contorná-los (por exemplo, com a implementação de *wrappers* [95] - partes de software que tem a função de proteger um componente de dados e fluxos de controle indesejados e inesperados). Estes trabalhos se apóiam no argumento de que se um programa que contém uma falha (por exemplo, o sistema operacional) invoca um outro programa enviando parâmetros anormais, o programa invocado deve se comportar de maneira robusta e não deve terminar de maneira abrupta ou entrar em um estado inconsistente. Dessa forma, dizemos que um software é robusto quando, mesmo em presença de falhas, o sistema responde adequadamente, atendendo às especificações dos requisitos deste software.

Testes de interface são apresentados como parte relevante dos testes de robustez (ref. Seção 2.3) [15]. Para se testar a robustez dos sistemas com os testes de interface, valores excepcionais ou inválidos são enviados através das interfaces do sistema, isto é, pelas interfaces do usuário ou parâmetros da API (*Application Program Interface*).

A técnica de emulação de falhas por corrupção de parâmetros é bastante relevante para os testes de robustez. Neste contexto, a incerteza quanto à representatividade e realismo dos erros injetados em relação às falhas reais de software não é importante, pois a prerrogativa dos testes de robustez é a injeção de valores inválidos ou excepcionais enviados pelas interfaces dos sistemas. Nesta abordagem clássica, normalmente são utilizados valores extremos para cada tipo de dado que está sendo substituído pela injeção de erros, valores válidos e valores inválidos [39].

A ferramenta Jaca [47], que foi utilizada extensivamente neste trabalho, utiliza o padrão para ferramentas de injeção de falhas proposto por Hsueh *et al.* [29] e injeta falha de interface (erro) em sistemas orientados a objetos escritos na linguagem Java. Jaca usa programação reflexiva para injetar os erros, processo que é auxiliado por um protocolo de meta-objetos, o Javassist [12]. Jaca possibilita a instrumentação do sistema sob teste durante sua execução de uma maneira bastante transparente para o usuário. A versão atual

da Jaca – JacaC3.0 [48] afeta interfaces públicas de um componente e traz no seu pacote de instalação diversas rotinas automáticas para viabilizar os testes estatísticos. Arquivos com extensão ".xml" podem ser utilizados para receber os valores a serem injetados, bem como gravar os resultados que são produzidos durante os experimentos.

Ballista [39,40] foi desenvolvida na universidade Carnegie-Mellon e seu principal objetivo é testar componentes off-the-shelf (OTS) em relação a problemas de robustez. A ferramenta combina teste de software com injeção de falhas. Os testes são feitos utilizando como entrada, uma combinação de valores excepcionais e válidos para os parâmetros das chamadas direcionadas ao kernel do sistema operacional. Esses valores são escolhidos de acordo com o tipo de dado dos parâmetros que irão substituir. Os valores dos parâmetros são randomicamente extraídos de um banco de dados onde estão armazenados testes pré-definidos. Os resultados são classificados em cinco níveis, três destes níveis têm detecção automática, os demais exigem um tratamento manual. A principal contribuição da Ballista é a geração automática do código fonte dos casos de testes que são depois executados pela ferramenta. Os casos de testes são programas que contém a declaração de parâmetros e das chamadas ao sistema operacional.

Microkernel Assessment by Fault injection AnaLysis and Design Aid (MAFALDA) [23] foi desenvolvida pelo Laboratoire d'Analyse et d'Architecture des Systèmes (LAAS-CNRS) em Toulouse na França. MAFALDA é uma ferramenta que caracteriza o comportamento dos microkernels em presença de falhas e facilita a implementação de mecanismos de detecção de erros para realçar o comportamento falho e a dependabilidade do microkernel sob teste. MAFALDA pode injetar falhas nos parâmetros das chamadas direcionadas ao microkernel, como também, nos segmentos de memória que executam o microkernel. Uma versão mais recente da ferramenta (MAFALDA-RT) estendeu a análise dos resultados para abranger novas medidas, tais como, tempo de resposta, time out, etc., que são medidas essenciais para sistemas de tempo real, facilitando a medição da intrusão que a ferramenta causa no sistema com os processos de injeção e monitoramento das falhas.

Generalizando o uso da injeção de falhas de interface, podemos injetar falhas nas interfaces entre os componentes para simular o envio de dados corrompidos de um componente que apresentou um defeito, a outros componentes que estão interagindo com o componente falho por meio de chamadas via API. Nesse caso, é importante que o valor

injetado seja representativo e emule a conseqüência de falhas residuais de software que possam estar presentes no componente que faz a chamada para a API. Utilizando a própria interface do sistema alvo, as falhas de interface são mais fáceis de serem injetadas, porém, sua representatividade nem sempre é fácil de ser obtida.

2.4.4. Injeção de Falhas de Software

A injeção de falha de software tem por objetivo reproduzir a falha propriamente dita (por exemplo, um engano do programador ao escrever o código fonte) sendo efetuada por meio da modificação das instruções do programa.

Falhas de software (ou falhas internas) são enganos cometidos pelos programadores. A injeção de falhas de software tem por objetivo reproduzir a falha propriamente dita, o que se traduz pela modificação das instruções do programa. Uma das primeiras formas de injeção de falhas utilizadas foi a injeção de falhas por mutação de código [18], que era utilizada para identificar os melhores conjuntos de casos de testes ou para estudar o processo de propagação de erros [21]. A injeção de falhas que realmente representem falhas existentes em produtos de software foi estudada por diversos autores nos últimos dez anos [14, 21]. No entanto, mesmo sem garantias de que os modelos de falhas de software injetadas são representativos, diversos estudos têm usado injeção aleatória de falhas de software. O trabalho de Voas *et al.* [64] revelou a fragilidade do software quando injeções aleatórias de falhas de software foram efetuadas. Outros exemplos de injeção de falhas utilizados para o teste de produtos de software podem ser encontrados nos trabalhos de Bieman *et al.* [6] e Blough e Torii [7]. Injeção de falhas de software emuladas por corrupção de código também já foi utilizada para validar os mecanismos de tolerância a falhas [51].

Diversos estudos de campo têm dado uma importante contribuição para a caracterização de falhas de software. Gray *et al.* [25, 26, 27] apresenta uma visão sobre os defeitos que ocorreram em sistemas Tandem, baseando-se no estudo de relatórios de defeitos, e concluiu que de 25% a 62% desses defeitos foram provocados por falhas de software, dependendo do contexto observado. Uma das causas apontadas pelo autor para esta variação é a complexidade do software. Os trabalhos apresentados por Lee e Iyer [42, 43] mostraram resultados sobre um estudo de aproximadamente 200 defeitos em sistemas

GUARDIAN90, que revela um cenário ainda mais preocupante. Neste estudo 76% dos defeitos foram causados por falhas de software.

Falhas de software, particularmente as falhas de interface, descobertas durante o desenvolvimento de programas computacionais foi o foco do trabalho de Perry e Evangelist [52]. Neste trabalho foram relacionados os tipos de falhas com as correções que se fizeram necessárias e proposto um método de classificação. Embora bastante interessante e fundamental para o trabalho de outros autores, o trabalho de Perry apresenta vários problemas na classificação das falhas, entre eles a mistura de falhas de níveis diferentes e categorias sobrepostas.

Sullivan e Chillarege [56, 57] apresentam estudos acerca do impacto de defeitos de software na disponibilidade de sistemas IBM, baseando-se em dados sobre defeitos registrados durante cinco anos. A conclusão de que a distribuição de falhas, quando classificadas pelo tipo de defeito, é dependente do estágio do desenvolvimento do produto, permitiu a Chillarege [13] estender o conceito e propor o esquema de classificação ODC – *Orthogonal Defect Classification*. Esta classificação é composta por categorias ortogonais. O problema de se injetar falhas de software que realmente representam falhas que podem ter sido cometidas por programadores durante o desenvolvimento foi, pela primeira vez, publicado por Christmansson e Chillarege [14] no âmbito do projeto ODC da IBM. A classificação das falhas, da maneira como foi proposta por Christmansson, é baseada em um conhecimento prévio das falhas reveladas no sistema, informação que, raramente, se encontra disponível para os sistemas existentes e, muito menos, para os sistemas que acabam de ser desenvolvidos.

Porém, quando o objetivo é a reprodução de falhas no código de um programa, a classificação ODC é ainda insuficiente. Além disso, o trabalho apresentado por Madeira *et al.* [46] mostrou que uma boa parte das falhas apontadas pelo estudo de Christmansson não podem ser injetadas pelas ferramentas típicas de injeção de falhas, pois requerem alterações no código que os injetores não são capazes de reproduzir.

Durães e Madeira [21] estenderam a classificação ODC baseando-se em um estudo de campo. Uma análise das falhas foi feita sob o ponto de vista do contexto do programa no qual esta falha ocorre e na relação existente com as construções lógicas de programação. Cada classificação ODC foi dividida em três subclassificações, considerando a natureza das

falhas: construção de omissão (*missing construct*), construção errada (*wrong construct*) ou construções espúrias (*extraneous construct*). Esta classificação é pertinente para a emulação de falhas, uma vez que a emulação de omissão (*missing construct*) é substancialmente diferente da emulação de uma construção espúria (*extraneous construct*), por exemplo. A Tabela 1 apresenta a classificação estendida da ODC proposta por Durães e Madeira [21].

Tabela 1 – Natureza das falhas estendendo os tipos da classificação ODC

Tipos ODC	Natureza	Exemplos	#falhas
Atribuição (Assignment)	Omissão	Não foi atribuído um valor a uma variável, a variável não foi inicializada, etc.	44
	Errada	Um valor errado (ou resultado de uma expressão, etc.) foi atribuído a uma variável	64
	Espúria	Uma variável não deveria ser atribuída (valor, resultado de expressão, etc.)	10
Verificação (Checking)	Omissão	Uma construção "if" é omitida, parte da condição lógica é omitida, etc.	90
	Errada	Uma condição "if" errada, condição de iteração errada, etc.	47
	Espúria	Uma condição "if" é supérflua e não deveria estar presente	0
Interface	Omissão	Um parâmetro em uma função foi omitido; expressão incompleta usada como param.	11
	Errada	Informação errada foi passada para uma função (valor, resultado expressão, etc.)	32
	Espúria	Dados supérfluos são passados para uma função (i.e. parâmetros excedentes)	0
Algoritmo (Algorithm)	Omissão	Algumas partes do algoritmo são omitidas (i.e. chamada de função, etc.)	155
	Errada	Algoritmo é codificado de maneira errada	37
	Espúria	O algoritmo tem passos supérfluos; Uma função foi chamada	6
Função (Function)	Omissão	Novos módulos são requeridos	21
	Errada	A estrutura de código deve ser redefinida para corrigir uma funcionalidade	15
	Espúria	Porção de código é completamente supérflua	0

Um estudo de campo analisou 532 falhas em trabalhos anteriores que pertenciam a quatro classes ODC. Estes tipos de falhas representam as falhas que foram observadas em campo com maior freqüência [21] e cobrem aproximadamente 50 % das falhas analisadas.

Tabela 2 – Representatividade dos tipos de falhas da faultload

Tipos de Falhas	Descrição	Perc. Observada em campo	ODC classes
MIFS	"If (*cond*) { statement(s) }" omitida	9.96 %	Algoritmo
MFC	Chamada de função omitida	8.64 %	Algoritmo
MLAC	"and expr" omitida em uma expressão usada em uma condição	7.89 %	Verificação
MIA	"if (*cond*)" omitida	4.32 %	Verificação
MLPC	Partes do algoritmo omitidas	3.19 %	Algoritmo
MVAE	Atribuição de variável usando uma expressão omitida	3.00 %	Atribuição
WLEC	Expressão lógica usada numa condição errada	3.00 %	Verificação
WVAV	Valor atribuído erradamente	2.44 %	Atribuição
MVI	Inicialização de variável omitida	2.25 %	Atribuição
MVAV	Omissão de atribuição de valor a variável	2.25 %	Atribuição
WAEP	Expressão aritmética errada usada como parâmetro	2.25 %	Interface
WPFV	Variável errada usada como parâmetro	1.50 %	Interface
	Total falhas cobertas	**50.69 %**	

A percentagem apresentada na terceira coluna da Tabela 2 representa a melhor aproximação da distribuição de cada tipo de falhas no código, conforme foi observado em campo.

A técnica G-SWFIT (*Generic Software Fault Injection Technique*) [21] relaciona tipos de falhas de software e os locais do código onde estes tipos de falhas possam realmente existir. A técnica utiliza uma biblioteca de emuladores de falhas de software que representam as falhas de software que ocorrem com maior freqüência. As modificações efetuadas pela injeção de falhas são feitas de modo que o novo código poderia ter sido gerado por um compilador da linguagem na qual o programa foi escrito. Tanto o padrão das falhas quanto a modificação feita sobre o código são definidos tendo como base o estudo de campo efetuado, estudo este que utilizou falhas de software localizadas em programas que estavam sendo utilizados em ambientes reais. A definição do conjunto de falhas a ser injetado é baseada no seguinte: considerando-se os tipos de falhas apresentados na Tabela 2, analisa-se o código alvo e identificam-se todas as localidades onde uma determinada falha pode ser realisticamente emulada, isto é, a falha que vai ser emulada poderia estar nessa localidade do código, fruto de um engano cometido por um programador.

Os tipos de falhas apresentadas na Tabela 2 são os tipos de falhas que devem ser utilizados na definição de *faultloads* baseadas em falhas de software. A classificação apresentada inclui detalhes que permitem reproduzir a falha que se pretende injetar no código do sistema alvo.

A técnica G-SWFIT emula estes tipos de falhas diretamente no código executável, não sendo necessário o código fonte, característica importante para se validar componentes produzidos por terceiros.

G-SWFIT é baseada numa metodologia de dois passos. Os locais onde as falhas devem ser injetadas são localizados no primeiro passo, gerando um conjunto de falhas a ser injetado. Esse primeiro passo ocorre antes dos experimentos propriamente dito. As falhas são realmente injetadas no segundo passo, durante a execução do sistema alvo. A técnica é pouco intrusiva, uma vez que a seleção do local e do tipo de falha a ser injetada já foi definida no primeiro passo. O processo de análise de código efetuado no primeiro passo, localiza no código de máquina a tradução das estruturas de programação e, dessa forma, modifica estas estruturas, emulando falhas apropriadas para cada um dos tipos de estruturas.

É importante ressaltar que a técnica G-SWFIT difere da abordagem de mutação utilizada no contexto de testes de mutação. Embora igualmente importante, a mutação de código tradicional é utilizada para avaliação dos casos de teste, baseada no código fonte, durante a fase de desenvolvimento e utiliza tipos de falhas que não têm a obrigatoriedade de serem representativas ou realísticas. O objetivo dos testes de mutação é exercitar o código fonte após uma falha ter sido inserida, para verificar se o caso de teste que está sendo executado identifica a falha injetada e, ocorrendo isso, assume-se que se este tipo de falha tiver sido cometido em outros locais do código o caso de teste está apto a revelá-la.

O objetivo da G-SWFIT é alterar o código executável para reproduzir falhas residuais que podem ter sido cometidas pelos programadores. Dessa forma, a representatividade da falha é extremamente relevante, o contexto de uso não se restringe à equipe de desenvolvimento e a técnica pode ser aplicada mesmo que não se tenha o código fonte da aplicação.

É importante destacar a maior facilidade de se injetar erros através das interfaces, corrompendo parâmetros e valores de retorno dos métodos invocados pelo sistema, uma vez que esta técnica utiliza a própria interface do sistema para injetar os erros. Por outro lado, a modificação de código feita pela técnica G-SWFIT permite um maior realismo e representatividade na emulação de falhas de software quando comparado com a injeção de erros através da interface.

2.5. *Benchmark* de Dependabilidade

Um *benchmark* de dependabilidade deve prover uma maneira genérica e reprodutível para caracterizar o comportamento de componentes e sistemas em presença de falhas. Um *benchmark* sendo um acordo técnico deve descrever o estado em que se encontra o sistema a ser avaliado, as medidas que são pertinentes, a maneira como estas medidas devem ser obtidas e o domínio no qual elas serão válidas [17]. De uma maneira resumida, um benchmark de dependabilidade é uma forma padronizada de medir os atributos de dependabilidade, isto é, medir a disponibilidade, a confiabilidade, confidencialidade, integridade, manutenibilidade, *safety* e *security*.

O usuário espera poder comparar componentes similares utilizando os resultados de um *benchmark*. Para isso, o *benchmark* precisa ser estatisticamente reprodutível,

representativo para o domínio do sistema que está sendo avaliado e portável, disponibilizando isso a um custo aceitável [17]. A maior parte dos *benchmark*s consiste em uma especificação do procedimento que se deve seguir para se obter as medidas relacionadas ao comportamento do sistema em presença de falhas, de maneira suficientemente precisa para que o usuário possa implementar essa especificação e compreender os resultados produzidos.

O projeto de um framework conceitual para avaliar a dependabilidade de componentes OTS ou de sistemas baseados em componentes implica em definir o contexto do benchmark e o conjunto de questões relacionadas à dependabilidade [17]. Isso inclui avaliação dos aspectos de dependabilidade que abrange técnicas baseadas em modelos (model-based technique em inglês) e técnicas baseadas em medidas (measurement-based technique em inglês). A diferença entre as duas abordabens é, principalmente, o nível de abstração considerado para descrever (modelar) o comportamento do sistema e as suposições sobre a distribuição dos processos estocásticos sobre os parâmetros do modelo [4].

Já foram propostos *benchmarks* para diversos tipos de sistemas e de componentes. Entre eles, *benchmark* de desempenho, *benchmark* de robustez e *benchmark* de disponibilidade.

A importância de um *benchmark* se mede pelo número de pessoas que o adotam e o aumento da sua importância é diretamente proporcional ao número de usuários. Considerando este critério, os *benchmarks* mais significativos são aqueles apresentados pelo *Transaction Processing Performance Council* (TPC) [125], SPEC *benchmarks* [123] e Wisconsin OO7 [112].

Estudos sobre *benchmark* de disponibilidade foram publicados para dois tipos de sistemas [10]. Um desses estudos [9] avalia a disponibilidade de um sistema de software de controle de discos RAID e examina a variação da qualidade dos serviços entregues pelo sistema quando este sistema foi alvo de injeção de falhas simulando falhas e defeitos de disco. Um outro estudo do mesmo autor [8] é baseado em um Gerenciador de Base de Dados Relacional (DBMS) utilizando a mesma metodologia desenvolvida em Brown e Patterson [9] adaptada para sistemas transacionais, na qual as medidas se basearam, tendo

como foco medidas de desempenho e consistência utilizando como *workload* o *benchmark* para sistemas transacionais TPC-C [125].

Alguns trabalhos já publicados propuseram *benchmark* de dependabilidade para determinadas famílias de sistemas. O trabalho apresentado em Kalakech *et al.* [33] aborda aspectos de *benchmark* de dependabilidade para sistemas operacionais. O foco principal do *benchmark* proposto é a robustez do sistema operacional (*kernel* do sistema operacional) quando uma aplicação falha interage com o sistema. O *faultload* consiste em corrupções nas chamadas do sistema (*system calls*).

Uma proposta preliminar para um *benchmark* de dependabilidade de sistemas de tempo real para sistemas espaciais é feita em Moreira *et al*. [49]. Este *benchmark* chamado DBench-RTK, apresenta seu foco principal na avaliação de predição de tempo de resposta para chamadas em RTK (*Real-Time Kernel*). Em outras palavras, permite que seja mensurada e comparada a chamada em RTK usada no domínio das aplicações espacial, medindo a robustez da interface RTK em relação às aplicações falhas que interagem com a interface.

No trabalho desenvolvido por Durães *et al.* [20] é especificado um *benchmark* de dependabilidade para *web-servers*. O trabalho consistiu de uma evolução do *benchmark* SPECWeb99 [123], ao qual foram acrescentadas novas medidas relacionadas à dependabilidade e ao *faultload* que incluem falhas de software, de hardware e de rede (network).

Em Vieira e Madeira [62] uma proposta para *benchmark* de dependabilidade para Banco de Dados é apresentada, definindo um *faultload* genérico com base nas falhas de operadores, para as quais uma classificação foi proposta. Um conjunto de diretrizes para a definição e validação de *benchmarks* de dependabilidade para Banco de Dados, centrado em sistemas OLTP (On-Line Transaction Processing) é apresentado em Vieira *et al.* [63]. Neste último trabalho é enfatizada a estreita relação que existe entre a definição do *benchmark* e os aspectos de validação.

Para a área automotiva, Ruiz *et al.* [54] apresenta uma proposta para avaliar a robustez de aplicações de sistemas de controle que rodam dentro de unidades de controle elétricas (ECU), focando mais precisamente em falhas transientes de hardware. Este *benchmark* provê meios para se comparar *safety* entre diversos ECUs. O *workload* é

baseado em padrões utilizados na Europa e o *faultload* consiste em falhas transientes de hardware, emuladas por bits presos (*bit-flips*) que afetam células de memória utilizadas nesses controles.

Por parte da indústria, houve esforços da Sun Microsystems [67] que define um *framework* de alto nível dedicado especificamente a *benchmark* de disponibilidade. O framework proposto se baseia em três componentes: Taxa de falhas/manutenção, robustez e recuperação (*recovery*). Além da Sun, a IBM [30] desenvolveu *benchmark* para quantificar a capacidade de se auto configurar, auto otimizar, auto proteger e auto corrigir.

2.6. Análise de Risco de Software

Risco pode ser pensado como uma probabilidade de que alguma circunstância adversa realmente venha a ocorrer. Os riscos podem ameaçar o projeto, afetando sua programação ou seus recursos; o software que está sendo desenvolvido, afetando sua qualidade ou seu desempenho; ou a organização, afetando a organização que está desenvolvendo ou adquirindo o software [122]. A abordagem deste trabalho é voltada para o risco de se utilizar ou reutilizar um componente de software no sistema que o integra.

Os trabalhos de Rosenberg *et al*. [88], Sherer [90], Amland [68], Bach [69], Perry [53] e Peters e Pedrycz [86] foram relevantes para se obter uma visão do conceito de análise de risco no âmbito de produtos de software. A maioria dos trabalhos procura estimar o risco para, baseado nesta estimativa, priorizar o esforço dos testes de software.

Teste de software é motivado por riscos [69]. Os testes não seriam necessários se não houvesse risco de um produto de software apresentar defeitos. Dessa forma, a análise de risco de um software pode ser usada para a fase de teste e validação deste software.

A definição de uma estratégia baseada em risco pode auxiliar a reduzir o número de experimentos necessários para validar o sistema. Essa estratégia pode direcionar o esforço de teste para as partes do software que apresentam maior risco para a aplicação, sempre que não for possível testar todo o software de maneira adequada [71]. Dessa forma, a escolha das partes que devem ser testadas pode ser feita de maneira criteriosa. Teste baseado em risco é uma técnica que tem se mostrado eficiente para corrigir os defeitos mais importantes de um determinado software.

O risco de um software é caracterizado pela combinação de dois fatores conforme apresentado na equação (1) extraída do trabalho de Rosenberg *et al.* [88], já referenciada na Seção 1 (refere-se à equação da página 9) e repetida a seguir para facilidade de leitura:

$$\textbf{Risco} = \sum(p(E_i) * c(E_i)) \qquad (1)$$

onde:

i : numera cada evento que tenha falha em potencial

E_i: evento que tenha falha em potencial

p: probabilidade de que erro em E_i ocorra

c: custo se um defeito ocorrer em E_i

Embora essa definição seja quase um padrão entre os autores, não existe um consenso sobre como obter a probabilidade e o custo de um defeito, sendo que o assunto é abordado de maneira diferente pelos diversos autores. Além disso, vale ressaltar que a severidade do risco é dependente da família de sistemas na qual o evento ocorre, uma vez que um determinado risco pode ser quantificado de maneira diferente quando o objetivo do sistema é distinto [88].

Para Bach [69] deve-se fazer uma **lista priorizada de riscos**, executar um ou mais testes que exploram cada risco e fazer um ajuste do esforço de teste quando há mudanças no cenário de riscos. A priorização é feita levando-se em conta que, quanto maior o impacto sobre o sistema se um defeito ocorrer e, quanto maior a probabilidade do evento que contém uma falha em potencial ocorrer no sistema, maior é o risco associado ao evento. Porém, a quantificação dessa medida é feita **baseada em heurísticas** e fica dessa forma, dependente da experiência do profissional que está atuando na avaliação. A escala de medidas é feita com base na escala "baixa, normal, alta" e o esforço de teste para os eventos considerados de risco alto é o dobro daqueles de risco médio, que por sua vez é o dobro daqueles de risco baixo.

A abordagem de Sherer [90] é baseada no **custo de defeitos apresentados em campo** e a seleção dos casos de testes é baseada no **perfil operacional**, que reflete o uso esperado do software em campo. A autora defende que se deve testar o software até que as conseqüências das falhas não mais justifiquem o custo dos testes. Além disso, o custo dos defeitos em campo depende da complexidade do sistema, da qualidade do processo de desenvolvimento do software, do ambiente operacional, da distribuição dos defeitos e do

custo das diversas entradas que alimentam o sistema. Assim, para Sherer [90], o risco de um módulo (parte de um software) é a probabilidade de uma falha ser ativada quando esse módulo é ativado, multiplicado pela exposição do módulo, que se traduz num fator econômico. A exposição do módulo é estimada por meio de entrevista com os usuários e a estimativa do impacto que um defeito pode trazer inclusive para a credibilidade da empresa que coloca esse software no mercado.

Perry [53] define os fatores de **risco associados à verificação e validação** (V&V). Entre esses fatores estão corretude (*correctness*), confiabilidade (*reliability*), portabilidade, manutenibilidade, desempenho, facilidade de uso. Defende que os fatores para cada família de software devem ser escolhidos pelos usuários e profissionais de testes para cada **fase do desenvolvimento** (Requisitos, Projeto, Implementação, Testes de Unidade, Testes de Integração, Manutenção). A distribuição deve ser feita considerando-se um terço de fatores de alto risco (quantificado como três ou quatro), um terço de fatores de médio risco (quantificado como dois) e um terço de fatores de baixo risco (quantificado como um) para a parte estrutural, técnica e de tamanho. Para obter-se o escore de cada risco, além dessa classificação, é atribuído um peso, que é multiplicado pela representação quantitativa da classificação anterior. Finalmente, para se obter o escore total, os escores estruturais, técnicos e de tamanho são somados.

Também Peters e Pedrycz [86] apresentam a avaliação de risco **baseada em fatores**. A qualidade do software é avaliada em termos de atributos de alto nível, chamados fatores, que são medidos em relação a atributos de baixo nível, os **critérios**. Um fator de software é uma visão orientada ao usuário da qualidade do produto (efeito), enquanto que os critérios são características orientadas ao software que indicam a qualidade do produto (causa). Por exemplo, o fator confiabilidade é definido pelos critérios de tolerância a falhas, consistência, exatidão e simplicidade. A cada critério e a cada fator são atribuídos valores máximos e mínimos que são estabelecidos por uma equipe de planejamento. Esta margem de valores deve ser respeitada pelo software que está sendo avaliado. Caso um determinado componente não respeite estes limites para um determinado critério ou fator, este componente será considerado um componente de risco e deverá ser direcionado a ele um maior esforço de teste.

Para Amland [68] a conseqüência de um defeito no ambiente operacional deve ser analisada sob o **ponto de vista do cliente** e sob o **ponto de vista do fornecedor** do software. Estas duas visões são consideravelmente distintas. Para o cliente, por exemplo, o software pode levá-lo a praticar atitudes ilegais involuntárias, a descumprir regras governamentais e à perda de mercado no seu segmento de atuação. Para o fornecedor, a conseqüência de publicidade negativa e altos custos de manutenção podem ser relevantes. Sendo assim, Amland expandiu a fórmula original contemplando as duas visões.

$$\text{Risco} = \sum(p(E_i) * \frac{\left(C_c(Ei) + C_v(Ei)\right)}{2})$$

onde:

i : numera cada evento que tenha falha em potencial

E_i: evento que tenha falha em potencial

p: probabilidade de que erro em E_i ocorra

C_c: conseqüência se um defeito ocorrer em E_i na visão do cliente

C_v: conseqüência se um defeito ocorrer em E_i na visão do vendedor.

Rosenberg *et al.* [88] abordou a avaliação da probabilidade de existir uma falha se **baseando nas métricas CK** [113], que é um conjunto de métricas que captura diversos aspectos de aplicações orientadas a objetos, incluindo complexidade, acoplamento e coesão. A fundamentação da abordagem utilizada é que quanto mais complexo é um componente de software, maior a probabilidade de apresentar falhas (*fault prone*). Neste trabalho, além de apresentar a complexidade como uma maneira de se avaliar o risco, os autores **apresentam valores padrão**, **valores esperados e aceitos** para cada uma das métricas do conjunto CK. Esses valores padrão foram obtidos em experimentos feitos com mais de 20.000 classes de sistemas orientados a objetos.

Schaefer [89] defende que devem-se testar os **cenários de uso** mais importantes para a aplicação. A escolha do cenário a testar deve-se basear nos cenários mais críticos, na visibilidade e na freqüência de utilização do cenário específico. Para se determinar quais áreas são críticas deve-se basear no ambiente e na conseqüência dos defeitos e, ainda, levar em conta redundância, facilidades de *backup/restore* e a possibilidade de se fazer verificação manual. A visibilidade do cenário depende do número de usuários que será

impactado por um defeito e deve levar em conta a tolerância dos usuários quando um defeito é apresentado. De maneira geral, para se classificar os cenários que devem ser testados deve-se considerar a complexidade, o fato de que o código que implementa o cenário foi modificado, se o software terá seu primeiro uso nesse tipo de aplicação, a utilização de novas tecnologias, o número de pessoas envolvidas. Além disso, é importante considerar se houve troca de pessoas responsáveis, se a equipe se encontra espalhada geograficamente, o tempo que se dispõe a exigência de otimização, defeitos apresentados anteriormente. Também devem ser considerados fatores locais, tais como, falhas de comunicação, pessoas que fizeram a tarefa, entre outras. Dessa forma, os testes devem ser direcionados para detectar os defeitos mais importantes, nas mais importantes funcionalidades (funcionalidades que dão suporte à função que o sistema deve desempenhar) e que apresentam maior criticidade. O autor ainda recomenda que, quando não se conhece o projeto, deve-se executar um teste preliminar com os cenários mais típicos do negócio de maneira que cubra todo o software. As áreas que apresentarem um maior número de erros nos testes preliminares deverão ser consideradas áreas críticas para as quais os testes devem ser direcionados.

Para Pfleger [87], além do impacto do risco (ou perda associada ao risco) e a probabilidade do evento ocorrer, deve-se levar em conta a **capacidade que se tem para modificar o impacto, anulando-o ou minimizando-o**. Na classificação de Pfleger, o risco pode ser genérico ou específico de cada projeto, pode ser voluntário ou involuntário. Para melhor avaliar o risco deve-se fazer um checklist baseado nas ocorrências de defeitos em projetos anteriores. Enfatiza ainda que, não é possível estimar a probabilidade como uma medida exata mas os dados experimentais podem ser utilizados para se avaliar os riscos.

No trabalho apresentado por Ntafos [85], o autor usa **modelo de teste de partição** para demonstrar o ganho que se pode obter ao considerar o custo de defeitos de campo para a alocação de testes. Os casos de testes são selecionados com base no domínio das entradas que são fornecidas ao software, de acordo com o perfil operacional, que reflete o uso esperado do software em campo. Segundo o autor, os casos de testes devem ser alocados proporcionalmente ao custo do defeito multiplicado pela probabilidade de existir uma falha dentro de cada subdomínio das partições. O custo do defeito em campo deve levar em conta a complexidade do sistema, a qualidade do processo de desenvolvimento do software, o

ambiente operacional, a distribuição dos defeitos e o custo das diversas entradas (ou classes de entradas).

2.7. Avaliação de Risco do uso de Componente no Software

A avaliação em sistemas computacionais pode ser baseada em modelos ou em medidas. Entre as técnicas baseadas em modelos podemos utilizar a modelagem analítica e simulação e entre as baseadas em medidas a observação, medidas de campo e injeção de falhas.

A simulação permite um melhor controle do tempo, do tipo de falha e do componente afetado do modelo e pode ser aplicada na fase inicial do projeto. Porém, a técnica requer um longo tempo de processamento, valores de entrada exatos e simuladores cujo desenvolvimento é caro e difícil [4].

Estimativas de medidas de dependabilidade são normalmente obtidas a partir de modelos analíticos. Nesses modelos, o foco é a seleção da arquitetura e a avaliação das medidas de dependabilidade, tais como, confiabilidade, disponibilidade, manutenibilidade, *safety* entre outras. A escolha de quais medidas utilizar depende fortemente da característica do sistema sob teste [4].

Na modelagem analítica a avaliação de dependabilidade é dividida em três fases: (i) escolha das medidas de dependabilidade, (ii) construção do(s) modelo(s), (iii) processamento do(s) modelo(s) para avaliar as medidas de dependabilidade.

A modelagem analítica pode ser completada com a injeção de falhas e com as medidas de campo. Utilizando-se medidas de campo pode-se avaliar a representatividade das falhas. Esta representatividade tem sido apontada como a característica chave, uma vez que é importante para os experimentos que utilizam a técnica de injeção de falhas e para um *benchmark*.

A obtenção de medida de campo envolve três etapas: (i) coleta de dados, definindo o quê coletar e como coletar, (ii) validação dos dados coletados, analisando quanto a sua corretude, consistência e completude, (iii) processamento dos dados, interpretando as análises estatísticas e avaliando as medidas quantitativas que caracterizam a dependabilidade [4].

Com a injeção de falhas é possível avaliar como o sistema se comporta em presença de falhas, como os diferentes tipos de falhas afetam o sistema e a eficiência dos mecanismos de tolerância a falhas.

Normalmente, modelos estatísticos são utilizados para se estimar a probabilidade de existir falha em um componente [80, 91] e análise de perigos (*hazard analysis*) é utilizada para estimar o impacto (ou custo) da ativação da falha [79].

Muitos estudos tentaram atenuar os problemas associados às falhas do software e estimar seu risco. Destacam-se estudos sobre teste de software, modelagem da confiabilidade de software e confiabilidade da análise de risco do software [76, 78, 80, 83].

Métricas de complexidade de software têm sido muito utilizadas para estimar a probabilidade de haver uma falha em um componente. O estudo apresentado por Basili *et al.* [70] validou experimentalmente o uso das métricas de projetos orientados a objetos como indicadores de qualidade, predizendo a propensão para falhas das classes do componente. Este estudo concluiu que diversas métricas são úteis para fazer esta previsão, durante as fases iniciais do ciclo de vida do software.

Segundo Lyu [80], a probabilidade de um componente apresentar defeitos está diretamente ligada à complexidade do componente. Na verdade, a relação entre métricas de complexidade e a propensão para a existência de defeitos foi apresentada em diversos estudos [72, 78]. Embora o uso de métricas de complexidade para este propósito tenha sido contestado por Fenton e Ohlsson [73], que dizia que as métricas não eram úteis para a propensão de falhas, Menzies [82] apresentou uma explicação para as divergências apontadas e confirmou a utilidade das métricas de complexidade para prever a existência de falhas em um componente. Menzies destacou ainda que esta previsão não possa ser utilizada como indicadores exatos, mas sim, com caráter estatístico. Além disso, Menzies destacou a importância de se escolher um conjunto coerente de métricas para cada problema e de se utilizar um grande número de dados quando se querem generalizar os resultados.

Diversos métodos foram utilizados para estimar a existência de falhas residuais. Alguns estudos utilizaram modelos baseados no registro dos defeitos que ocorreram durante a utilização do software [72, 92] e outros utilizaram heurísticas associadas ao

estabelecimento de valores de complexidade considerados como padrões de limites aceitáveis [69, 88].

As propostas de Basili *et al.* [70], Tang *et al.* [92] e Munson e Khoshgoftaar [84] para estimar o primeiro termo da equação do risco (equação (1) apresentada na Seção 1.4), seguem um modelo baseado na regressão logística [116]. Conceitos fundamentais sobre regressão logística podem ser encontrados no Apêndice A desta tese.

Combinando o impacto observado pela injeção de falhas e a probabilidade estimada pelos modelos estatísticos é possível classificar individualmente cada componente do sistema quanto ao risco que este componente apresenta para o sistema do qual é parte.

Neste capítulo apresentamos os trabalhos relacionados com os conhecimentos que foram utilizados nesta tese, O objetivo é complementar as informações contidas nos artigos publicados. Os próximos capítulos apresentam uma cronologia que mostra o desenvolvimento do trabalho e a maneira como os artigos se completam para atingir nosso objetivo de estimar experimentalmente o risco que um componente apresenta para o sistema no qual ele é integrado.

Capítulo 3–Análise de Risco para Injeção de Falhas

Na primeira fase deste trabalho estudamos o risco que representa cada componente de software para selecionar quais componentes devem ser utilizados como pontos de injeção de falhas em uma campanha de validação do sistema quanto a sua robustez. O objetivo do trabalho nesta fase se concentra na análise do risco visando a eliminação de falhas.

O objetivo principal deste capítulo é mostrar ao leitor como os artigos que foram publicados utilizando a análise de risco são seqüenciados. Embora possam ser encontradas redundâncias entre o conteúdo deste capítulo e os artigos que foram relacionados nas seções que se seguem, essas redundâncias se devem, antes de tudo, ao nosso objetivo de proporcionar um resumo do conteúdo e conclusões de cada artigo na língua portuguesa. Desta forma, o leitor poderá compreender o conteúdo desta tese caso não queira lê-los em detalhes.

3.1. Análise do Risco com base na Complexidade dos Componentes

O risco é freqüentemente calculado com base nas métricas de complexidade de cada componente. Neste caso, a escolha do componente de maior risco como alvo da validação por injeção de falhas se justifica uma vez que, quanto mais complexo for o componente maior é a probabilidade de este ter uma falha [71, 88].

No primeiro trabalho publicado no "*First Latin-American Symposium on Dependable Computing* – LADC", em 2003, apresentamos uma estratégia para a validação da robustez usando um gerenciador de banco de dados orientado a objetos (ODBMS) como componente. As falhas são injetadas nas interfaces (na verdade trata-se de injeção de erros) entre a aplicação e o ODBMS. A aplicação escolhida foi um *benchmark* de desempenho criado para comparar diversos gerenciadores de base de dados orientados a objetos. A nossa

intenção era simular um cenário real de utilização, uma vez que um *benchmark* para avaliação de um gerenciador deve implementar os principais cenários de uso de uma aplicação de banco de dados. Para a injeção dos erros (altera o estado de variáveis e parâmetros) foi utilizada a ferramenta Jaca, que injeta erros em sistemas programados na linguagem Java.

Os critérios utilizados para a seleção dos locais de injeção foram baseados:

- nas métricas de complexidade
- na existência de uma interação direta entre classes da aplicação e classes do componente
- no fato das classes da aplicação apresentarem um relacionamento de herança das classes do gerenciador de bases de dados.

Os erros foram injetados na aplicação (*benchmark*) e propagados para o componente (ODBMS Ozone).

Neste trabalho concluímos que a classe que apresentava a maior complexidade em diversas métricas e interagia diretamente com o componente foi a classe responsável pelos defeitos observados. Por outro lado, o método responsável por estes defeitos não foi aquele que apresentava a maior complexidade, mas sim o método responsável pelo cenário mais crítico, isto é, aquele que implementa a função mais importante e mais freqüente da aplicação .

Concluímos também que, uma estratégia baseada no risco do componente é útil para a seleção dos locais onde as falhas devem ser injetadas durante uma campanha de injeção de falhas, pois direciona a escolha do componente alvo, reduzindo o esforço de teste. Com poucas injeções conseguimos perceber a fragilidade do componente quanto à sua robustez.

A ferramenta Jaca é útil para validar componentes OTS (*Off-The-Shelf*), mesmo que não se possa injetá-los diretamente (quando se tem informações apenas da *Application Protocol Interface-API* do componente). Nesse caso, a robustez do componente pode ser validada pela propagação dos erros através da interface entre a aplicação e o componente. No Capítulo 6 da Parte II, o artigo "*A Strategy for Validating an ODBMS Component using a High-Level Software Fault Injection Tool*" é transcrito, detalhando metodologia, resultados e conclusões.

3.2. Amostra Estratificada e Teste de Partição

Um segundo artigo utilizou as métricas de complexidade como critério para definir partições formadas pelas classes do componente (ODBMS Ozone) que apresentavam métricas dentro de limites estipulados por valores padrão de complexidade (*threshold values*) [88] e classes que apresentavam métricas acima destes valores. Além disso, como o componente era composto de muitas classes, teria sido impossível considerar todas as classes como ponto de injeção. Nem mesmo teria sido possível injetar falhas em todas as classes cujas métricas foram calculadas como estando acima dos valores padrão, uma vez que o número de classes nesta condição também era muito grande. Dessa forma, uma amostra estratificada foi definida com base na teoria da amostragem [126]. A amostra estratificada foi utilizada para fazer uma seleção baseada em critérios, levando-se em conta a complexidade. Os estudos foram definidos com base na métrica WMC (*Weighted Methods per Class*) [113]. Baseando-se nos valores aceitáveis desta métrica definidos no trabalho de Rosenberg *et al.* [88] as classes do software foram separadas em estratos [119]. As classes com métrica acima do valor padrão compuseram o estrato S2 e abaixo do valor padrão o estrato S1. O tamanho de cada estrato foi definido considerando o software completo. O tamanho da amostra foi então calculado utilizando a expressão genérica apresentada na Figura A.1 (ref. Apêndice A) e a teoria de proporção que garantiu que se conservasse a mesma representação do software completo na amostra. Dessa forma, os estratos da amostra foram determinados respeitando a proporção de cada estrato no software completo e o tamanho total da amostra. Esta abordagem foi utilizada para que se pudesse ter confiança estatística na amostra.

Os erros foram injetados em classes de ambas as partições, isto é, os pontos de injeção de falhas foram escolhidos dentro das classes de cada estrato. O objetivo desta escolha era verificar se realmente as classes que apresentavam complexidade acima do valor padrão eram responsáveis pelos defeitos que foram provocados.

Os resultados mostraram que o comportamento do componente (ODBMS Ozone) foi diferente quando os erros foram injetados nas diferentes partições. Porém, ao contrário das nossas expectativas, a partição que continha classes com a métrica WMC acima do valor padrão não produziu os defeitos mais severos (corrupção do banco de dados).

Concluímos que uma métrica não é suficiente para a escolha das partições, sendo necessários outros fatores como, por exemplo, as dependências entre o gerenciador de Base de Dados e a aplicação. Além disso, dentro de cada partição escolhemos as classes que apresentavam valores mais altos da métrica WMC, o que pode ter influenciado os resultados.

Para esclarecer se a escolha dos componentes com valores mais altos da métrica WMC realmente teve influência nos resultados obtidos, refizemos os experimentos utilizando a escolha aleatória dentro das partições. Ainda assim, continuamos a considerar apenas a métrica WMC. Esses experimentos foram feitos após o prazo de entrega do artigo e o resultado não pôde ser integrado na publicação, motivo pelo qual são descritos mais detalhadamente.

Foram efetuadas 209 injeções distribuídas em classes com alta, média e baixa medida WMC. A Tabela 3 apresenta as classes que serviram como alvo dessas injeções.

Contrariamente à hipótese de que as classes com mais alta medida WMC seriam responsáveis pelos defeitos apresentados, os resultados nos mostraram que os defeitos ocorreram quando uma única classe com baixa medida WMC é alvo das injeções. Uma análise criteriosa do arquivo de log gerado pela ferramenta mostrou que a classe em que a injeção de falhas provocou defeitos foi realmente a única em que as injeções efetuadas foram ativadas.

O cenário de uso utilizado pelo *benchmark* Wisconsin OO7 invocava diretamente a classe onde as falhas foram injetadas e por este motivo foram ativadas. As demais classes não foram invocadas diretamente pela aplicação, nem tampouco foram invocadas por outras classes na cadeia de dependências.

Dessa forma, podemos concluir que usar a métrica de complexidade como único critério para se analisar o risco de um componente provocar defeito não leva a resultados satisfatórios.

Tabela 3 – Seleção de Classes do Ozone para Injeção de Falhas

Nome	WMC	Falha Ativada?	Quantas Injeções?	Estrato
core.wizardStore.ClusterStore	103	Não	5	S2
ClientCacheDatabase	90	Não	4	S2
core.UserManager	64	Não	10	S2
core.wizardStore.WizardObjectContainer	50	Não	4	S2
core.Env	40	Não	10	S1
tools.OPP.CDHelper	38	Não	4	S1
xml.util.CXMLContentHandler	37	Não	4	S1
data.SimpleArrayList	35	Não	10	S1
DbCacheChunk	35	Não	4	S1
xa.OzoneXAResource	34	Não	10	S1
core.DatabaseImpl	34	Não	4	S1
tools.OPP.OPP	33	Não	10	S1
DxLib.DxMultiMap	32	Não	10	S1
core.xml.HashtableContentHandler	31	Não	10	S1
core.AbstractObjectContainer	30	Não	4	S1
core.admin.AdminObjectContainer	30	Não	4	S1
DocumentImpl	29	Não	2	S1
ExternalTransaction	28	Não	4	S1
blob.BLOBContainerImpl	26	Não	10	S1
xml.util.ChunkOutputStream	25	Não	10	S1
ExternalDatabase	17	Não	10	S1
blob.BLOBInputStream	11	Não	10	S1
RemoteDatabase	**11**	**Sim**	**10**	S1
core.ClassManager	9	Não	4	S1
tools.Statistics	9	Não	4	S1
core.KeyGenerator	8	Não	5	S1
core.xml.ValueObjElement	7	Não	4	S1
blob.BLOBOutputStream	6	Não	10	S1
blob.BLOBPageImpl	6	Não	10	S1
core.xml.ValueElement	6	Não	4	S1
tools.BenchmarkProgressLog	3	Não	5	S1

No Capítulo 7 da Parte II, o artigo "*Using Stratified Sampling for Fault Injection*" publicado no "*Second Latin-American Symposium on Dependable Computing* – LADC", em 2005 é transcrito e provê maiores detalhes sobre estes experimentos.

3.3. Análise de Risco na perspectiva da Arquitetura de Software

Como foi apontado no experimento da amostra estratificada, as métricas de complexidade não são suficientes para determinar o maior risco de um componente para o software no qual esse componente está integrado. Por este motivo resolvemos investigar outros fatores que poderiam colaborar para determinar o melhor ponto de injeção, segundo uma estratégia baseada em risco. Neste cenário, a arquitetura do software nos pareceu um bom campo de investigação.

3.3.1. Análise de Risco com base em Heurísticas

Em nossa primeira investigação neste campo, utilizamos um sistema baseado na integração de componentes como técnica de construção do sistema de software. Dessa forma, os trabalhos efetuados nesta fase diferem de investigações anteriores principalmente porque não mais consideram um componente isolado, mas sim um conjunto de componentes integrados sendo que alguns deles podem ser adquiridos de terceiros (*Common-Off-The-Shelf* – COTS), sendo nesse caso, considerados "caixa-preta". Em conseqüência, a estratégia para a escolha dos pontos de injeção baseada em risco não pode se fundamentar em métricas de complexidade dependentes do código fonte. Além disso, a unidade considerada para os experimentos, usando a técnica de injeção de falhas, não é mais uma classe e sim um componente.

No trabalho apresentado no "*Workshop on Architecting Dependable Systems - The International Conference on Dependable Systems and Networks (DSN04)*" utilizamos uma estratégia baseado em heurísticas, proposta por Bach [69]. Na proposta de Bach, um conjunto de critérios deve ser considerado para se avaliar o risco de um componente para o sistema no qual este componente se encontra integrado. São eles: (i) a unidade de software é um componente novo; (ii) o componente foi alterado; (iii) o componente pode propagar falhas para outros componentes do sistema devido às dependências arquiteturais; (iv) o componente pode ser influenciado por outros componentes que propagam falhas devido às dependências arquiteturais; (v) a unidade de software é um componente que implementa uma funcionalidade crítica para o sistema; (vi) a unidade de software é um componente popular, isto é, muito utilizado; (vii) a unidade de software é um componente estratégico, isto é, um componente que agrega um diferencial perante o mercado consumidor; (viii) o

componente é adquirido de terceiros (COTS); (ix) a unidade de software é um componente externo que, de certa forma, é utilizado pelo sistema. Além desses critérios propostos por Bach, acrescentamos mais um critério, a dificuldade de se entender o componente. Ou seja, a falta de informação disponível ou a maneira deficiente como esta informação está apresentada, dificultando o entendimento do componente. Estes critérios foram combinados numa matriz e, de acordo com o número de critérios que é conferido ao componente, este componente é classificado como sendo de alto, médio ou baixo risco. Dessa forma, escolhemos os componentes que apresentem alto risco como pontos de injeção.

Além de utilizarmos a arquitetura para definir o risco do componente para o sistema (particularmente considerando os critérios iii e iv), a arquitetura também foi utilizada para definirmos os pontos de monitoração dos experimentos. Analisando a arquitetura, quando um componente é escolhido como ponto de injeção, os componentes que interagem diretamente com o componente injetado são escolhidos como ponto de monitoração.

As conclusões a que chegamos com este trabalho é que, quando se está testando um sistema baseado em componente, que integra componentes COTS e componentes *in-house* (produzidos na organização, mas não obrigatoriamente para o mesmo contexto de uso), os componentes *in-house* podem ser a única opção para se injetar falhas com o objetivo de se validar um componente COTS quanto à sua robustez. A interface do componente COTS pode ser a única informação disponível sobre o componente e esta interface pode não oferecer o controle necessário para que as ferramentas possam injetar as falhas. Nesse caso, a injeção nos componentes *in-house* pode ser feita com o objetivo de validar o componente COTS através da propagação de erros. Outra importante observação é a relevância dos conectores como pontos de injeção e observação dos experimentos. Injetando nos conectores que integram componentes COTS podemos testar os COTS pela propagação dos erros através das interfaces entre esses COTS e os conectores que os integram. Por outro lado, quando a injeção é feita em componentes *in-house* que são integrados com outros componentes por meio de conectores, esses conectores podem ser utilizados como pontos de observação.

Quanto à estratégia utilizada, concluímos que ela ajuda a selecionar os pontos de injeção e monitoração com base em risco, fazendo um bom uso da arquitetura do sistema,

mas, por outro lado, um critério baseado em heurística traz uma grande subjetividade à avaliação e um critério mais determinístico seria desejável.

Uma contribuição deste trabalho foi a constatação das limitações da ferramenta Jaca quanto a dificuldade de se lidar com diferentes "*threads*". Esta constatação levou à introdução de melhorias na ferramenta. Este artigo, intitulado "*Architecture-based Strategy for Interface Fault Injection*", se encontra transcrito no Capítulo 8 da Parte II para maiores detalhes.

3.3.2. Análise de Risco com base nas Dependências Arquiteturais

No segundo trabalho desta série exploramos as dependências arquiteturais. Como estratégia para avaliarmos os componentes de maior risco, para com isso direcionarmos as injeções de erros, utilizamos uma adaptação da proposta de Stafford e Richardson [107] já apresentada na Seção 2.2.

A técnica proposta para minimizar a parte do software que deve ser analisada quando o software sofre modificações (por exemplo, para a reaplicação de testes de regressão) foi utilizada para se determinar as classes de maior risco com base no número de elementos que participam do conjunto *affected-by chain* e *affects chain*. O número de classes nesses dois conjuntos é considerado como um critério para definir as classes que apresentam maior risco para a aplicação. Quanto maior o número de elementos existente nestes conjuntos, maior é a chance de um erro se propagar através das interfaces. Nesse caso, tanto o componente alvo pode receber a propagação de erros de um grande número de componentes (e em decorrência disso não se comportar de maneira robusta provocando defeito no sistema), quanto um erro propagado pelo componente alvo pode influenciar um grande número de componentes fazendo com que estes não se comportem de maneira robusta e provoquem um defeito no sistema. Dessa forma, com uma única injeção é possível testar a robustez de um maior número de componentes do sistema sob teste. Além de se definir os componentes de maior risco para direcionar a injeção de falhas, a dependência arquitetural foi utilizada para se definir os pontos de monitoração dos experimentos. Componentes que se relacionam diretamente com o componente alvo na cadeia de dependência foram utilizados como pontos de monitoração.

A metodologia proposta para os testes se baseou em sete passos: (i) modelar o sistema (caso um diagrama de arquitetura não esteja disponível); (ii) construir a matriz de dependência; (iii) determinar as dependências dos componentes do sistema baseados na técnica *Chaining*; (iv) determinar o componente a ser injetado, com base no número de elementos dos conjuntos *affected-by* e *affects chain*; (v) determinar as interfaces a serem consideradas para se injetar as falhas (assume-se que o componente pode ter várias interfaces requeridas e/ou providas); (vi) determinar os valores a serem injetados de acordo com o tipo de dados dos parâmetros e/ou valores de retorno de métodos públicos do componente; (vii) definir os possíveis resultados que possam ser obtidos e classificá-los.

Como estudo de caso nesse trabalho, utilizamos uma implementação do sistema de caldeiras proposto no trabalho de Anderson *et al*. [95] que teve sua arquitetura adaptada primeiramente ao estilo C2 de arquitetura proposta por Taylor *et al.* [109] e depois complementada para uma solução tolerante a falhas, *idealised* C2 *Component* (iC2C) [98]. O estilo C2 de arquitetura é um estilo em camadas, baseado em componentes. No estilo C2 não existe a integração direta entre componentes, sempre haverá um conector entre dois componentes que será responsável pelo roteamento de mensagens, broadcasting e adaptações necessárias. O estilo iC2C complementa o estilo C2 e separa componentes que tratam o comportamento normal de componentes que tratam o comportamento excepcional dentro da arquitetura de software.

Como resultado dos experimentos pôde-se observar que os defeitos reportados foram apresentados quando os erros foram injetados entre os componentes do sistema e os conectores do *framework* C2, indicando que também o *framework* precisa ser protegido por *wrapper* para que o sistema atinja níveis esperados de dependabilidade.

A principal contribuição do trabalho é a metodologia que foi apresentada utilizando a análise da arquitetura de software e as dependências arquiteturais para guiar os experimentos que utilizam injeção de falhas. Uma vantagem da utilização da abordagem baseada na arquitetura é que a validação do sistema pode ser planejada nas fases iniciais do processo de desenvolvimento, tão logo já se tenha a definição da arquitetura do sistema alvo. Portanto, esta abordagem é independente de se ter disponível o código fonte. Mesmo no caso em que não se tem o modelo da arquitetura do sistema disponível, podem-se utilizar ferramentas que conseguem extrair o modelo a partir do código executável (mesmo

quando um grande número de classes está envolvido). Isso é importante para sistemas que integram componentes COTS, uma vez que nesse caso, o código fonte geralmente não se encontra disponível.

A análise de dependência pode ser útil para diversos propósitos (por exemplo, para determinar o impacto de modificações). Para a injeção de falhas pode ser útil para: (i) determinar os componentes a que se devem dirigir as injeções; (ii) determinar os melhores pontos de observação dos experimentos; (iii) determinar os componentes que podem ser utilizados como pontos de injeção de falhas para atingir, através da propagação, o componente alvo quando este não tiver o controle necessário para se fazer as injeções de falhas (ou erros) ou monitorar os experimentos.

Este trabalho foi publicado como um capítulo de livro na série "*Architecture Dependable Systems III*" com o título "*Fault Injection Approach based on Architectural Dependencies*" e se encontra no Capítulo 9 da Parte II para melhor detalhar resultados e conclusões.

O último trabalho que considera as dependências arquiteturais para definir uma estratégia para validar o sistema que integram componentes utilizou a mesma estratégia para validar um sistema que integrava um componente real utilizado diariamente em aplicações comerciais, o gerenciador de base de dados Ozone. Neste trabalho expandiu-se o conceito de componente para uma granularidade mais fina, considerando uma classe do sistema como um componente. Cada versão do componente deve ser muito bem testada e validada, constatação que pode ser inferida a partir das listas de discussão e mensagens trocadas entre desenvolvedores e usuários do componente. Os experimentos efetuados foram concentrados nas sete classes do componente que apresentavam maior número de elementos no conjunto *related chain* (componentes que antecedem e que sucedem o componente alvo).

Mesmo sendo o Ozone um componente altamente testado, em 11,5% dos experimentos, os dados produzidos pelo componente foram corrompidos, fazendo com que o componente apresentasse defeito severo no contexto da aplicação. A utilização da estratégia propiciou uma redução de pelo menos 75% dos experimentos necessários para cobrir as classes que mantinham interação direta entre componente e aplicação.

Concluímos que a análise de dependência é eficiente para selecionar as classes a serem injetadas e observamos que a classe que continha o maior número de elementos no seu conjunto *affected-by chain*, foi aquela que apresentou os defeitos mais severos. No Capítulo 10 da Parte II o artigo "*Fault Injection Approach based on Dependence Analysis*" pode ser encontrado para mais detalhes. Este artigo foi publicado no *First International Workshop on Testing and Quality Assurance for Component-Based Systems.*

Um outro artigo surgiu como resultado dos experimentos efetuados, visando o entendimento da importância da arquitetura para a técnica de injeção de falhas. Um estudo para avaliar o uso de *wrappers* gerou o artigo "*Melhorando a Dependabilidade de Componentes com o uso de Wrappers*" que se encontra no Capítulo 11 da Parte II e foi apresentado no *VII Workshop de Testes e Tolerância a Falhas – WTF' 06*.

Capítulo 4 – Avaliação Experimental do Risco

Depois de explorar a análise do risco baseada em métricas de complexidade e na arquitetura do software para selecionar os melhores pontos de injeção e de observação (Capítulo 3 e artigos relacionados), resolvemos investigar como poderia ser feita uma medida desse risco, ou seja, como medir o risco que um componente de software representa para o sistema no qual está integrado. O objetivo principal dos trabalhos deste capítulo é prover uma medida experimental do risco de usar um determinado componente em um sistema e com isso poder avaliar as melhores opções para se obter sistemas com maior dependabilidade.

Assim como no capítulo 3, nas seções deste capítulo podem ser encontradas redundâncias entre o conteúdo do capítulo e os artigos relacionados. Novamente, isto se deve à nossa preocupação com os leitores que, dessa forma, poderão compreender o conteúdo desta tese caso não queira lê-los em detalhes.

4.1. Comparação de Falhas de Interface e Falhas Internas

As falhas de interface têm sido utilizadas com muito sucesso para testar a robustez de sistemas de software e, neste contexto, a injeção de erros nas interfaces é perfeitamente válida. Mas para generalizar o uso da injeção de falhas de interface e, particularmente para se extrair medidas de confiabilidade, a representatividade das falhas/erros injetados durante os experimentos precisa ser considerada.

Ao estudar a arquitetura do sistema e utilizá-la para determinar as dependências arquiteturais pudemos avaliar o papel importante que os componentes predecessores (componentes que provêem informações para o componente alvo) e sucessores (componentes para os quais o componente alvo provê informações) desempenham para a escolha das interfaces entre componentes visando tornar as injeções mais eficientes. Assim, nos deparamos com uma questão: serão os erros injetados nas interfaces estatisticamente equivalentes às conseqüências de falhas residuais que se encontram nos componentes predecessores? Se assim fosse, a injeção de falhas internas poderia ser substituída por

falhas de interface que são muito menos complexas e mais fáceis de serem injetadas, uma vez que utilizam as interfaces dos próprios componentes para efetuar essas injeções.

Dessa forma, fizemos um estudo que comparou as conseqüências das falhas injetadas no interior dos componentes (falhas de software) com as conseqüências dos erros injetados nas interfaces (falhas de interface).

A representatividade das falhas de software (falhas internas) foi alvo de um estudo de campo que avaliou diversos produtos de software para classificar e determinar os tipos de falhas que ocorrem com maior freqüência devido a enganos cometidos por programadores [21]. A técnica G-SWFIT (*Generic Software Fault Injection Technique*), já descrita na Seção 2.4.4 e utilizada nesta tese, é uma abordagem que oferece boa representatividade das falhas internas, uma vez que analisa a estrutura de programação existente no código fonte do produto e reproduz uma falha que seria passível de ser cometida naquela estrutura específica. Dessa forma, a falha emulada representa uma falha que, se estivesse no código fonte do produto, não impediria que o programa compilasse e poderia perfeitamente passar despercebida pela equipe de teste.

Considerando as falhas de interface, os valores que foram injetados nos experimentos deste artigo são os valores que comumente são utilizados nos testes de robustez (menor inteiro, maior inteiro, zero, limites dentro do domínio, primeiro valor fora do domínio, valores nulos, etc.).

Os resultados foram classificados de acordo com o seguinte modo de defeito (*failure mode*): término abrupto (*crash*), tempo excedido (*hang*), término normal e resultados errados (*wrong*) e término normal e resultados corretos (*correct*).

Para estes experimentos foram utilizados dois *setups* bastante distintos para que diferentes cenários de uso fossem experimentados. Os dois *setups* são compostos por aplicações reais: no primeiro foi utilizado o gerenciador de base de dados orientado a objetos escrito em Java, o Ozone, e no segundo foi utilizada uma aplicação escrita na linguagem C, de tempo real, que gerencia o controle de bordo e a comunicação de um sistema de satélite, utilizado pela European Space Agency (ESA). Para injetar falha interna, a técnica G-SWFIT foi utilizada nos dois *setups*, uma vez que a técnica é independente da linguagem de programação utilizada. Para a injeção de erros nas interfaces foram necessárias ferramentas distintas. Para o *setup* Java a ferramenta Jaca foi utilizada,

enquanto que para o *setup* C foi utilizada a ferramenta Xception. A diversidade de ferramentas utilizadas e sistemas de domínios bastante distintos objetivaram a generalização dos resultados.

Os resultados observados sugerem que as falhas de interface não representam as falhas internas, uma vez que o comportamento do sistema quando as falhas de interface são injetadas diferem substancialmente do comportamento deste mesmo sistema quando as falhas internas são injetadas. Pode-se observar que existe uma intersecção, mas as falhas de interface não cobrem as falhas internas. Essa conclusão sugere que não há equivalência entre as falhas de interface e os erros propagados em conseqüência de falhas residuais que podem existir no interior de componentes predecessores.

Seria desejável se pudéssemos injetar falhas de interface em vez de injetar falhas internas, pois a injeção de falhas de interface é muito mais simples e reduz o tempo necessário para se efetuar os experimentos, mas os resultados obtidos nos experimentos não recomendam que esta substituição seja feita. Pelo contrário, os resultados sugerem que a injeção de falhas de interface e a injeção de falhas internas são complementares e que uma técnica não substitui a outra.

Outra conclusão interessante é que os valores utilizados nas injeções de falhas de interface apresentam, em geral, a mesma contribuição para o modo de defeitos e não são particularmente relevantes para determinar o impacto produzido no sistema. Este fato pode ser considerado para reduzir os valores utilizados para as injeções de erros nas interfaces, uma vez que a variedade de entradas não parece contribuir para os experimentos. O artigo "*Injection of faults at component interfaces and inside the component code: are they equivalent*?" pode ser encontrado no Capítulo 12 da Parte II para mais detalhes sobre este estudo. Este artigo foi apresentado no *Sixth European Dependable Computing Conference – EDCC'06*.

4.2. Avaliação de Risco utilizando Injeção de Falhas Internas

A estimativa do risco que um componente de software apresenta para um sistema tem sido feita com base em heurísticas e experiência de desenvolvedores [69, 90]. Nossa proposta é avaliar o risco de maneira experimental.

Baseando-se na equação básica do risco utilizada por diversos autores [68, 69, 88, 90] tem-se que o risco de um software é caracterizado pela combinação da severidade do impacto (ou custo) que eventos que tenham falhas em potencial possam causar no sistema e a probabilidade de que esse evento ocorra [88].

Dessa forma, utilizamos a conjugação de duas técnicas para se avaliar experimentalmente o risco. Para estimar a probabilidade de uma falha existir no componente utilizamos modelos estatísticos baseado na regressão logística e utilizamos a injeção de falhas para avaliar a severidade do impacto quando uma determinada falha é revelada.

Cientes da não equivalência das falhas de interface e das falhas internas e, não tendo neste momento um estudo da representatividade das falhas de interface, escolhemos a injeção de falhas internas para avaliar o impacto que uma falha revelada em um componente tem sobre o sistema. A representatividade das falhas internas foi alvo de vários estudos [13, 21] sendo razoável a sua utilização para se obter medidas mais realísticas deste impacto.

4.2.1. Estimativa de Falhas Residuais

A estimativa da probabilidade de falhas em produtos de software foi apresentada por Basili *et al.* [70]. Naquele trabalho, foi feito um estudo de campo para se levantar as falhas encontradas em produtos de software na fase de teste e estes dados juntamente com as métricas de complexidade orientadas a objetos [113] foram utilizados num modelo estatístico baseado na regressão logística. Nesse modelo, as diversas métricas foram combinadas para se fazer uma previsão das falhas residuais que permanecem no software após este entrar na fase operacional. No nosso caso utilizamos um modelo semelhante, mas consideramos que a informação relativa às falhas encontradas na fase de teste não está disponível.

Uma vez que se assume que não há informação prévia sobre as falhas, utilizamos um estudo publicado pelo Rome Laboratory [120] para assumir a densidade de falhas usada como ponto de partida. No estudo do Rome Laboratory, foi analisada de maneira empírica a densidade de falhas em produtos de software e concluiu-se que a densidade de falhas pode ser tomada no intervalo de 0,1 a 1 falha para cada mil linhas de código (Kloc). Tomando-se esta densidade e o número de linhas de código, fizemos uma estimativa do número de

falhas de cada módulo e usamos este número para substituir a informação que Basili tinha inicialmente no seu estudo. Esta informação foi combinada com as métricas de software levantadas, utilizando-se um modelo baseado na regressão logística. Vale ressaltar que o intervalo de densidade de falhas inicial ([0,1 , 1]) pode ser utilizado para parametrizar excelências no processo de desenvolvimento, podendo ser escolhido uma densidade menor (tendendo a 0,1) quando o processo de desenvolvimento for mais rigoroso e a experiência da equipe for maior ou uma densidade maior (tendendo a 1) caso contrário.

Com o objetivo de validar o modelo estatístico proposto, fizemos um estudo de campo com diversos produtos de software *open source*, analisando 350 bugs reports. Nesse estudo, levantamos os *bugs* apontados por usuários durante a utilização dos produtos de software e comparamos com a estimativa obtida. Depois de validado, utilizamos a estimativa para propor uma distribuição de falhas para os experimentos que utilizam a técnica de injeção de falhas visando a avaliação experimental do risco. A representatividade da distribuição de falhas nestes experimentos é importante para simular mais realisticamente o comportamento em campo. Estes resultados foram publicados em um trabalho intitulado "*A field data study on the use of software metrics to define representative fault distribution*" apresentado no "*Workshop on Empirical Evaluation of Dependability and Security (WEEDS) - The International Conference on Dependable Systems and Networks* (*DSN 06)"*, que pode ser encontrado para maiores detalhes no Capítulo 13 da Parte II.

Neste trabalho observamos que a densidade de falhas gerada pela nossa abordagem é consistente com o que foi observado em campo para os módulos pequenos e médios e desviam do observado nos módulos maiores e mais complexos. Porém, a densidade observada está sempre abaixo da densidade estimada, o que pode significar que a complexidade e extensão dos módulos não permitiram que as falhas residuais tivessem sido descobertas e que a estimativa feita pela nossa abordagem ainda venha a se concretizar, uma vez que os produtos de software utilizados para observação dos *bugs* ainda estão em uso. Além disso, em se tratando de estimativas, o conservadorismo é indicado e a estimativa pode ser considerada pessimista devido a adoção da densidade inicial.

Outra conclusão importante foi relacionada à análise do *p-value* (significância estatística de cada métrica). Observamos que ao eliminar as métricas cujos valores de *p-*

value estavam acima do valor de corte estabelecido (valor utilizado como corte foi 0,1), tivemos uma melhoria significativa na estimativa obtida pela abordagem. A densidade de falha estimada se aproximou significativamente da quantidade de *bugs* observados.

4.2.2. Avaliação Experimental do Custo

Para a avaliação do impacto das falhas residuais no componente utilizamos a técnica experimental, através da injeção de falhas internas, com as quais estimamos o custo para o sistema se uma falha residual for ativada em um de seus componentes. Para o propósito de se injetar falhas internas a técnica G-SWFIT foi utilizada. As falhas foram injetadas uma a uma e o modo de defeitos (*failure mode*) utilizado foi o mesmo já apresentado na Seção 4.1 (*crash, hang, wrong, correct*).

Quando uma falha de software é injetada em um componente, esta falha pode ser ou não ser ativada e mesmo quando a falha é ativada pode causar ou não causar um desvio no comportamento do componente. Ainda assim, apenas uma fração das falhas que causam um desvio no comportamento do componente irá levar o sistema a apresentar defeito (*crash, hang, wrong*). Isso irá depender da arquitetura do sistema, dos mecanismos de tolerância a falhas que foram implementados, do conjunto de dados de carga do experimento (*workload*) e do perfil operacional. Algumas falhas podem ser toleradas (quer pela redundância inerente no sistema, quer por mecanismos de tolerância a falhas específicos) e não causar nenhum efeito visível. Isso significa que para se medir o custo da ativação de uma falha (ref. ao segundo termo da equação (1) apresentada na Seção 1.4) usando a técnica de injeção de falhas é preciso considerar dois termos:

$$\boldsymbol{cost\ (E_i) = prob(E_i ativa) * c(failure)} \qquad (2)$$

onde:

cost(E_i): impacto (custo) de uma falha residual

prob($E_i$ativa): probabilidade de uma falha, uma vez ativada, causar erro

c(failure): conseqüência da ativação da falha (defeito observado no sistema).

Porém, os dois termos (*prob($E_i$ativa)* e *c(failure))* são avaliados em conjunto durante os experimentos de injeção de falhas. As falhas injetadas sendo ativadas, poderão ou não alterar o comportamento do componente causando um erro (*prob($E_i$ativa)*) e, alterando o

comportamento do componente poderão causar maior ou menor impacto (*c(failure)*). Dessa forma, o custo é traduzido por um percentual calculado entre o número de experimentos que apresentaram defeitos (*crash, hang, wrong*) e o total de falhas injetadas no componente alvo.

O risco da utilização de um componente para o sistema no qual este componente está integrado é calculado como sendo o produto da probabilidade de haver uma falha residual no componente, estimado através do modelo estatístico baseado em regressão logística (ref. equação 4 do Apêndice A) e o custo avaliado experimentalmente através da injeção de falhas de software.

A metodologia proposta para a avaliação experimental do risco foi aplicada em um estudo de caso que avaliou um gerenciador de base de dados orientado a objetos, o Ozone e dois sistemas operacionais largamente utilizados em ambientes computacionais, o Real-Time Operating System for Multiprocessor Systems (RTEMS) e o Kernel do Linux (RTLinux). O Ozone representa uma aplicação Java e os sistemas operacionais representam uma aplicação de tempo real, desenvolvida na linguagem C. Os sistemas operacionais foram utilizados em um sistema de controle de satélite, o Command and Data Management System (CDMS), que foi constituído ora integrando o RTLinux, ora o RTEMS.

Buscando uma melhor aproximação da probabilidade de falhas, foram utilizados dois conjuntos de métricas. O conjunto de métricas de complexidade CK [113] que é um conjunto de métricas bastante utilizado e o conjunto de métricas de Halstead [115] que tem um foco diferente das métricas CK. As métricas CK medem a complexidade do software sob o ponto de vista da estrutura enquanto as métricas de Halstead (que são freqüentemente usadas para medir a complexidade de um produto de software) medem a complexidade sob o ponto de vista léxico e/ou textual. Por este motivo as abordagens se complementam em termos de medidas de complexidade.

Para a extração das métricas foram utilizadas três ferramentas, a CMTJava e a CMT++ [124] para extrair as métricas de Halstead para o *setup* Java e o *setup* C respectivamente e a ferramenta a RSM [121] para extrair as métricas CK para ambos *setups*.

Para a avaliação do Ozone, consideramos como componentes três classes do Ozone com tamanho e complexidades distintas. Na análise dos resultados pudemos observar que todas as falhas injetadas na classe menor e menos complexa causaram um defeito severo.

Apesar do impacto causado por uma falha ativada nesta classe, sua complexidade (e tamanho) é muito baixa, o que sugere que é muito pouco provável que um programador cometa uma falha nessa classe. Devido a isso, o risco deste componente para o sistema é também muito pequeno.

Outra constatação é que embora o custo do componente mais complexo seja apenas duas vezes o custo do componente de complexidade média, o risco do componente mais complexo se apresenta como sendo dezoito vezes maior do que o risco calculado para o componente de média complexidade. Isso se deve ao fato de que a densidade de falhas é muito mais alta no componente mais complexo, isto é, existe muito mais probabilidade de haver uma falha residual neste componente.

No segundo setup, que foi usado em duas versões, cada versão com um sistema operacional diferente (RTLinux e RTEMS), consideramos o sistema operacional (componente para o qual se calculou o risco) como um único componente, sem considerar a sua estrutura interna. Embora o RTEMS tenha uma complexidade geral menor, existe um maior número de componentes com complexidade maior o que fez com que a densidade de falhas estimada fosse maior. Apesar disso, o custo do RTLinux foi maior para a maioria das falhas injetadas.

Os resultados mostraram que para a aplicação em causa (o sistema de controle de satélites, Command and Data Management System) o RTEMS apresenta um risco menor para a maioria dos modos de defeitos (*failure mode*). Estes resultados geraram um trabalho intitulado "*Experimental Risk Assessment using Software Fault Injection*" submetido presentemente para publicação ao "*The International Conference on Dependable Systems and Networks*", que pode ser encontrado no Capítulo 14 da Parte II.

Capítulo 5 – Discussão e Conclusões

Este capítulo faz uma revisão dos problemas que esta tese se propôs a resolver, relaciona as soluções propostas e lista as contribuições deste trabalho. O capítulo também expõe limitações da abordagem e sugere alguns trabalhos futuros que poderão fazer uso do conteúdo desta tese.

5.1. Problemas e Soluções

Esta tese propõe duas abordagens para o uso do risco de produtos de software para a validação de sistemas baseados em componentes. Na primeira abordagem, visando a **eliminação de falhas**, propõe-se utilizar a análise do risco de um componente para selecionar os locais de injeção e de observação dos resultados dos experimentos durante uma campanha de injeção de falhas. Esta abordagem define um critério importante de seleção, quando não se podem injetar todos os componentes do sistema (o que é freqüente no caso de sistemas complexos). Neste caso, a seleção dos locais de injeção é guiada por métricas de complexidade e/ou dependências arquiteturais, de forma a tornar as campanhas de experimentos mais eficazes.

Na segunda abordagem, propõe-se uma avaliação experimental do risco de se utilizar um componente específico como parte integrante de um sistema construído a partir da integração de vários componentes que, em seu conjunto, complementam a funcionalidade esperada do sistema. Esta abordagem estima o risco colaborando com a **predição de falhas** e baseia-se em um modelo estatístico e na injeção de falhas de software.

Como resultados desta tese podemos destacar as seguintes soluções que foram propostas para atender às dificuldades apontadas:

1. Escolha dos locais onde injetar falhas

A análise do risco com base nas métricas de complexidade permitiu que propuséssemos uma maneira de selecionar os componentes alvos da injeção de erros visando os testes de robustez. Os resultados deste estudo apontam que além da análise do risco com base nas métricas de complexidade, é preciso levar em conta outros critérios. Cenário de utilização da aplicação e a interação existente entre componentes são critérios relevantes que complementam o resultado obtido por meio da análise do risco com base nas métricas de complexidade. Esses critérios podem ser utilizados para melhor definir os locais para a injeção de falhas ou monitoração dos experimentos.

Além das métricas de complexidade, a arquitetura do software e a análise das dependências arquiteturais foram utilizadas para determinar os componentes que devem ser injetados em uma campanha de validação por injeção de erros. O princípio envolvido na utilização da análise de dependência é que quanto mais dependências um componente apresentar, maior é sua complexidade. Dessa forma, escolhem-se os componentes mais complexos como alvo das injeções de erros, ou seja, escolhem-se os componentes que apresentam maior número de componentes dependentes.

2. Previsão da Propagação de Erros

O entendimento da propagação de erros através das interfaces entre os componentes do sistema e, interfaces entre componentes do sistema e as interfaces de fronteira, é uma questão importante para a validação através da injeção de falhas.

O estudo sobre a propagação de erros através das interfaces de componentes, permitiu um melhor entendimento do risco que um valor fora da especificação pode trazer ao sistema. A robustez do componente pode ficar seriamente ameaçada em decorrência da propagação de erros através das interfaces.

Pelos resultados obtidos pudemos perceber que componentes que não foram alvo de injeções de erros foram influenciados por erros propagados através da interface. Pudemos concluir que o componente escolhido pode ser validado pela propagação de erros contornando o problema da falta de controle para diretamente injetar erros/falhas. Nesse caso, as injeções devem ser feitas em outros componentes que interajam diretamente com o componente escolhido.

Também a arquitetura do software e a análise de dependência foram importantes para um maior entendimento da propagação de erros. A probabilidade de o sistema apresentar defeito cresce com o número de dependências do componente alvo. O aumento da probabilidade de apresentar defeito deve-se à propagação de erros para um maior número de componentes que acaba por afetar a robustez do sistema. Quando um erro é injetado, todos os componentes desta cadeia podem receber o efeito do erro injetado e pode-se testar um componente sobre o qual não se tem controle, através da injeção de erros/falhas em outro componente que pertença à sua cadeia de dependência.

A propagação de erros através da cadeia de dependências também foi útil para definir pontos de monitoração durante os experimentos. A monitoração dos componentes na cadeia de dependência possibilita a avaliação do comportamento em presença de falhas de todos os componentes que recebem a propagação do erro injetado no componente alvo.

3. Identificar se as Falhas de Interface são ou não representativas

A representatividade das falhas que devem ser emuladas nos experimentos de injeção de falhas é difícil de ser identificada e esta representatividade é um requisito fundamental quando se quer avaliar o risco. A obtenção de medida do risco exige representatividade, caso contrário a medida obtida pode não ter nenhum significado. Embora exista estudo sobre a representatividade das falhas internas, não se conhecia estudos sobre a representatividade das falhas de interface.

O impacto das falhas de interface foi comparado com o impacto causado pela ativação de falhas residuais de software no interior do componente. O objetivo era verificar se o impacto causado pelas falhas de interface correspondia ao impacto causado pelas falhas internas. Se houvesse correspondência, o impacto da injeção das falhas de interface e das falhas internas deveria ser, ao menos em termos estatísticos, o mesmo. Nesse caso, a injeção de erros através da interface poderia substituir a injeção de falhas internas com vantagens quanto à facilidade e custo de aplicação da técnica. Porém, os resultados deste trabalho mostraram que não há correspondência entre o impacto causado pelas duas abordagens. Concluiu-se que as abordagens são complementares e uma abordagem não substitui a outra.

4. Avaliação Experimental do Risco

Pelo que temos conhecimento, esta é a primeira vez que se propõe a avaliação experimental do risco que um componente de software apresenta para o sistema no qual este componente está integrado. A abordagem proposta considera os aspectos estáticos e dinâmicos de um produto de software.

Os aspectos estáticos são representados pelas métricas de complexidade que, combinadas em um modelo estatístico, baseado na regressão logística, fornece uma estimativa da probabilidade de haver uma falha residual (densidade de falha) em um componente alvo. O modelo foi validado baseado em falhas de software reais reportadas na fase operacional em vários produtos de software de plataforma livre. Esta prova de conceito mostrou que o modelo apresenta uma boa aproximação da estimativa com a real situação que se apresenta em campo para componentes de baixa e média complexidade e estima um número relativamente alto para os componentes mais complexos. No entanto, o número observado é sempre menor que o número estimado pelo modelo, o que pode indicar que nem todas as falhas residuais tenham sido encontradas nos componentes mais complexos e que a previsão feita possa vir a se concretizar. Além disso, há de se considerar a posição conservadora utilizada na estimativa quando se considerou a densidade de falhas inicial.

Para medir o risco que a utilização de um componente apresenta para o sistema é importante a representatividade da distribuição de falhas a ser injetada durante uma campanha. Neste trabalho foi proposto que as falhas sejam proporcionalmente distribuídas, considerando-se a densidade de falhas. Dessa forma, um maior número de falhas serão injetadas nos componentes que apresentem maior propensão a falhas.

Os aspectos dinâmicos relacionados com a avaliação experimental do risco são emulados pela injeção de falhas internas. A idéia é emular a ativação da falha residual e a propagação de um erro em conseqüência desta falha que leva (ou não) o componente a um desvio de comportamento causando (ou não) um defeito no sistema. Apoiado pela representatividade provida pela técnica G-SWFIT apresentamos uma maneira de se avaliar experimentalmente o impacto da ativação de uma falha residual no sistema em que o componente está integrado.

Combinando a probabilidade de haver uma falha residual no componente e a probabilidade de que o impacto da ativação desta falha leve o sistema a apresentar defeito

estima-se o risco deste componente para o sistema no qual o componente está integrado. Desta forma propusemos uma maneira de estimar experimentalmente o risco do componente para o sistema.

A avaliação do risco permite que sejam classificados os componentes de software quanto ao risco que representam para o sistema no qual estão integrados. Esta classificação pode indicar os componentes que precisam de melhorias ou auxiliar na distribuição do esforço de teste.

A avaliação do risco pode também ser útil para a escolha do melhor componente para um sistema. Uma vez que temos uma maneira de classificar os componentes de software quanto ao risco que eles apresentam para o sistema, podemos integrar os diferentes componentes que provêem a mesma funcionalidade, medir o risco de cada um e escolher aquele que apresente o menor risco para o sistema. Essa mesma idéia pode ser utilizada para *benchmark* de componentes quanto ao risco, uma vez que a avaliação é provida de uma forma genérica e repetível, embasada em medidas bem definidas.

Seguindo a metodologia proposta apresentamos a avaliação do risco para dois sistemas operacionais largamente utilizados em ambientes computacionais: *Real-Time Operating System for Multiprocessor Systems* (RTEMS) e o *Kernel* do Linux (RTLinux). Os resultados mostraram que o RTEMS apresenta um risco menor para a maioria dos modos de defeitos (*failure mode*) considerando o sistema de controle de satélites, *Command and Data Management System* (CDMS), no qual os componentes foram integrados para serem avaliados. Para o CDMS, o sistema operacional RTEMS é a melhor opção entre os dois sistemas operacionais.

Apresentamos a avaliação do risco para algumas classes do Gerenciador de Banco de Dados Ozone. Considerando este resultado foi possível entender como as duas probabilidades colaboram para a medida do risco. Um componente que apresentou um impacto severo, mas é composto por poucas linhas de código fonte, apresentou como resultado um risco pequeno. O baixo risco estimado deve-se à pequena probabilidade de haver falhas residuais, indicando que em um número muito reduzido de linhas de código a probabilidade de um programador cometer um engano é baixa.

O objetivo da utilização de sistemas com características diversas e diferentes ferramentas foi a generalização dos resultados.

5.2. Limitações da Abordagem

A abordagem proposta para a avaliação experimental do risco não reflete o risco real observado em campo. Além dos fatores já considerados na nossa abordagem, outros fatores deveriam ter sido considerados e deverão ser alvo de trabalhos de investigações futuras. O risco real depende diretamente do perfil operacional, do cenário de utilização, a arquitetura de software entre outros fatores que ainda precisam ser investigados. Porém, como primeira tentativa de se fazer uma avaliação experimental do risco da utilização de um componente em um sistema mais complexo, a abordagem proposta é útil para se comparar a melhor opção entre dois componentes que provêem a mesma funcionalidade. Esta comparação há de ser feita integrando os componentes alvos da avaliação em um mesmo sistema validando os mesmos cenários de utilização em um mesmo ambiente operacional.

5.3. Relação das Contribuições e Publicações

- Uma metodologia para selecionar os pontos de injeção de falhas de interface com base em métricas de complexidade.
  Parte I, Capítulo 3, Seção 3.1 e 3.2. Parte II, Capítulos 6 e 7.
  Artigos Publicados:

  "*A Strategy for Validating an ODBMS Component using a High-Level Software Fault Injection Tool*", "*First Latin-American Symposium on Dependable Computing* – LADC' 03".

  "*Using Stratified Sampling for Fault Injection*" publicado no "*Second Latin-American Symposium on Dependable Computing* – LADC' 05",

- Uma metodologia para selecionar os pontos de injeção de falhas de interface com base na arquitetura de software. Parte I, Capítulo 3, Seção 3.3. Parte II, Capítulos 8 a 10.
  Capítulo de Livro:

  "*Fault Injection Approach based on Architectural Dependencies*", "*Architecture Dependable Systems III*", Springer Verlag, 2005.

  Artigos Publicados:

  "*Architecture-based Strategy for Interface Fault Injection*", "*Workshop on Architecting Dependable Systems - The International Conference on Dependable Systems and Networks*" – DSN 04.

"Fault Injection Approach based on Dependence Analysis", "First International Workshop on Testing and Quality Assurance for Component-Based Systems", 2005.

- Um estudo da representatividade de falhas de interface. Parte I, Capítulo 4, Seção 4.1. Parte II, Capítulo 12.

  Artigo Publicado:

  "*Injection of faults at component interfaces and inside the component code: are they equivalent*?", *Sixth European Dependable Computing Conference* – EDCC'06.

- Um modelo para estimar a densidade de falhas. Uma metodologia para a distribuição representativa de falhas com base na densidade de falhas estimada. Parte I, Capítulo 4, Seção 4.2, Subseção 4.2.1. Parte II, Capítulo 13.

  Artigo Publicado:

  "*A field data study on the use of software metrics to define representative fault distribution*" "*Workshop on Empirical Evaluation of Dependability and Security (WEEDS) - The International Conference on Dependable Systems and Networks*" *– DSN 06.*

- Uma metodologia para se avaliar experimentalmente o risco. Uma metodologia para escolha do melhor componente entre componentes de mesma funcionalidade. Parte I, Seção 4.2, Subseção 4.2.2. Parte II, Capítulo 14.

  Artigo Submetido:

  "*Experimental Risk Assessment using Software Fault Injection*", "*The International Conference on Dependable Systems and Networks*" *– DSN 07*.

5.4. Outras Publicações

Publicados em Congressos Nacionais

Mendes, N. V. ; MORAES, R. ; Martins, E. ; Madeira, H. . Jaca Tool Improvements for Speeding Up Fault Injection Campaigns. In: Simpósio Brasileiro de Engenharia de Software, 2006, Florianópolis. Proceedings of the SBES 2006.

Jacques, G, S. ; MORAES, R. ; Weber, T. S. ; Martins, E. . Validando Sistemas Distribuídos Desenvolvidos em Java Utilizando Injeção de Falhas de Comunicação por Software. In: WTF 2004, 2004, Gramado - RS. Proceedings of the WTF 2004, 2004.

Demonstração de Ferramentas

MORAES, R. ; MARTINS, E. . Jaca - A Software Fault Injection Tool. Los Alamitos, California: Proceedings The International Conference on Dependable Systems and Networks, 2003 (Demonstração de Ferramenta (Tool Demo)).

Resumos

MORAES, R. . Strategy for Object-Oriented Database Test using Software Fault Injection. In: The Internation Conference on Dependable Systems & Networks, 2003, São Francisco. DSN 2003 - Suplemental Volume. Los Alamitos, CA, USA : IEEE Computer Society Publication, 2003. v. 2. p. A-43-A-45.

MORAES, R. ; MARTINS, E. . Jaca - A Software Fault Injection Tool. In: The International Conference on Dependable Systems and Networks, 2003, São Francisco, CA. Proceedings of the 2003 International Conference on Dependable Systems and Networks. Los Alamitos, California : Computer Society, 2003. v. 1. p. 667-667.

MORAES, R. . Methodology for OODBMS COTS Tests using Software Fault Injection. In: The International Conference on Dependable Systems & Networks, 2002, Bethesda - Maryland. DSN 2002 - Suplementar Volume. Los Alamitos , CA , USA : IEEE Computer Society Publications, 2002. v. 2. p. A-15-A-17.

5.5. Trabalhos Futuros

Como trabalho futuro, o estudo da maneira como uma falha de software (falha interna) atinge a interface dos componentes é um bom campo de pesquisa. Desse entendimento pode-se indicar uma **maneira representativa de se emular uma falha de interface**, semelhante ao estudo que foi efetuado para as falhas de software. Dessa forma, a injeção de falhas de interface poderá ser utilizada para a estimativa de medidas com vantagens quanto à facilidade e custo de aplicação da técnica. Este estudo, muito provavelmente, irá nos indicar adaptações relevantes a serem feitas nas ferramentas injetoras de falhas de interface.

Os **fatores a serem considerados para avaliar experimentalmente o risco** precisam ser alvos de investigações mais completas. Já identificamos que os cenários de utilização, a arquitetura de software e o perfil operacional são fatores importantes que devem compor as métricas para as estimativas. Importante também considerar a taxa de ativação das falhas quando o impacto de uma falha no componente é avaliado experimentalmente através da injeção de falhas. A avaliação precisa refletir as diferenças nestes fatores para os diversos sistemas avaliados.

Uma utilização importante da avaliação experimental do risco é voltada à **certificação de componentes quanto ao risco** que este componente apresenta para o sistema no qual está integrado. Esta certificação deve atestar que o produto alvo da certificação apresenta um risco dentro de limites estipulados por um contrato ou estabelecidos por um padrão de

certificação. Em algumas áreas de aplicação, especialmente aplicações críticas, a certificação é essencial ou mesmo exigida por lei. Consideramos que a certificação de componentes deverá ser consolidada como uma atividade necessária para o futuro do desenvolvimento de software.

Um desenvolvimento importante para facilitar o uso da metodologia de avaliação do risco proposta é o desenvolvimento de uma **ferramenta para injetar falhas** internas segundo a técnica G-SWFIT para sistemas desenvolvidos na linguagem Java. A ferramenta existente hoje é difícil de ser utilizada e consegue injetar falhas internas de maneira automática somente em sistemas desenvolvidos na linguagem C.

Uma limitação que encontramos para a avaliação do risco é a ausência de técnicas que **extraiam medidas de complexidade a partir do código executável**. A avaliação do risco, da maneira como foi proposta, não precisa do código fonte para ser efetuada, mas como depende de métricas de complexidade, acaba por esbarrar na falta de ferramentas que possa obtê-las independentemente de se ter o código fonte disponível. A independência do código fonte é uma condição indispensável para que se possa trabalhar com componentes COTS. Embora já existam alguns trabalhos que indicam uma preocupação com este assunto [129], o conjunto de métricas ainda é muito limitado. Porém, ferramentas de extração de medidas estão fora da nossa linha de pesquisa e espera-se que este trabalho incentive a construção de ferramentas que independam do código fonte para a extração de medidas de complexidade de software.

Referências

Referências Relacionadas com Injeção de Falhas e *Benchmark*

[1] Arlat J., Crouzet Y., Laprie, J.C. "Fault Injection for Dependability Validation of Fault-Tolerant Computing Systems". In: *Proc. of the 19th IEEE International Symposium on Fault Tolerant Computing – FTCS´89*, Chicago, Illinois, pp. 348-355, 1989.

[2] 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, pp. 166-174, 1990.

[3] Arlat, J., Costes, A., Crouzet Y., Laprie, J.C. "Fault Injection and Dependability Evaluation of Fault Tolerant Systems". *IEEE Transaction on Computers*, vol. 42, n. 8, pp.919-923, 1993.

[4] Arlat, J., Kanoun, K., Madeira, H., Busquets, J., Jarboui, T., Johansson, A., Lindström, R. "DBench: Dependability Benchmarking – State of the Art". LAAS-CNRS, França, Tech. Rep. IST-2000-25425, 2001.

[5] Avizienis, A., Laprie, J. C., Randell, B. "Fundamental Concepts of Dependability". UCLA CSD, Tech. Rep. 010028, 2001.

[6] Bieman, J., Dreilinger, D., Lin, L. "Using Fault Injection to Increase Test Coverage". In: *Proc of The 7th IEEE International Symposyum on Software Reliability Engineering - ISSRE'96*, New York, NY, USA, 1996.

[7] Blough, D., Torii, T. "Fault-Injection-Based Testing of Fault-Tolerant Algorithms in Message Passing Parallel Computers". In: *Proc. of The 27th IEEE Int.Fault Tolerant Computer Symposium - FCTS-27*, Seattle, USA, pp. 258-267, 1997.

[8] Brown, A., Wallnay, K.: "The Current State of CBSE". *IEEE Software,* 15 (5), IEEE Computer Society Press, pp. 37-48, 1998.

[9] Brown, A. Patterson, D. "Towards Availability Benchmarks: A Case Study of Software RAID Systems". In *Proc. of the 2000 USENIX Annual Technical Conference*, San Diego, CA, 2000.

[10] Brown, A. "Availability Benchmarking of a Database System". In: *EECS Computer Science Division*, University of California at Berkeley, 2001.

[11] Carreira, J., Madeira, H., Silva, J.: "Xception: Software Fault Injection and Monitoring in Processor Functional Units". *IEEE Transaction on Software Engineering*, 24 (2), 1998.

[12] Chiba, Shigeru. "Javassist – A Reflection-based Programming Wizard for Java". In: *Proc of the ACM OOPSLA'98 Workshop on Reflective Programming in C++ and Java*, 1998.

[13] Chillarege, R., *Orthogonal Defect Classification*. Handbook of Software Reliability Engineering, M. Lyu, Ed.: IEEE Computer Society Press, McGraw-Hill, Ch. 9, 1995.

[14] Christmansson, J., Chillarege, R. "Generation of an Error Set that Emulates Software Faults". In: *Proc. of The 26th IEEE Fault Tolerant Computing Symp. – FCTS-26*, Sendai, Japan, 1996.

[15] Cohen, J., Plakosh, D., Keeler, K. "Robustness Testing of Software-Intensive Systems: Explanation and Guide". Tech. Rep. CMU/SEI-2005-TN-015, 2005.

[16] Cusick, J., Koga, R., Kolasinski, W., King, C. “SEU vulnerability of the Zilog Z-80 and NSC-800 microprocessors”. *IEEE Transaction on Nuclear Science*, vol. NS-32, pp. 4206-4211, 1986.

[17] “DBench: Dependability Benchmark – Final Report”. LAAS-CNRS, França, Tech. Rep. IST-2000-25425, 2004.

[18] DeMillo, Richard A., Guindi, D., McCracken, W., Offut, A., King, K. “An Extended Overview of the Mothra Software Testint Environment” In: *Proc. ACM Sigsoft/IEEE 2nd Workshop on Software Testing, Verification and Analysis*, pp 142-151, 1988.

[19] DeMillo, Richard A., Li, T., Mathur, A. P. “Architecture of TAMER: A Tool for Dependability Analysis of Distributed Fault-Tolerant Systems”. Department of Computer Sciences, Purdue University, 1994.

[20] Durães, J., Vieira, M., Madeira, H. “Dependability Benchmarking of Web-Servers”. In: *Proc. of the 23rd International Conference on Computer Safety, Reliability and Security, SAFECOMP 2004*, Potsdam, Germany, 2004.

[21] Durães J., Madeira, H. "Emulation of Software Faults: A Field Data Study and a Practical Approach ". *IEEE Transactions on Software Engineering*, vol. 32, n. 11, ISSN: 0098-5589, November 2006.

[22] Fabre, J.-C., Salles, F., Moreno, M., Arlat, J. “Assessment of COTS Microkernels by Fault Injection”. In: *Proc.of The 7th IFIP Working Conference on Dependable Computing for Critical Applications- DCCA’99*, San Jose, CA, USA, pp. 25-44, 1999.

[23] Fabre, J.-C., Rodriguez, M., Arlat, J., Salles, F., Sizun, J. “Building Dependable COTS Microkernel-based System using MAFALDA”. In: *Proc.of The 2000 Pacific Rim International Symposium on Dependable Computing – PRDC’00*, pp. 85-92, 2000.

[24] Gil, P., Arlat, J., Madeira, H., Crouzet, Y., Jarboui, T., Kanoun, K., Marteau, T., Durães, J., Vieira, M., Gil, D., Baraza, J-C., Gracia, J. DBench: Dependability Benchmarking – Fault Representativeness. LAAS-CNRS, França, Tech. Rep. IST-2000-25425, 2002.

[25] Gray, J. Why do Computers Stop and What can be Done About It? Tandem Tech. Rep. 85.7, 1985.

[26] Gray J. “A Census of Tandem Systems Availability Between 1985 and 1990”. *IEEE Transactions on Reliability*, vol. 39, pp. 409-418, 1990.

[27] Gray, J., Siewiorek, D. “High-Availability Computer Systems”. *IEEE Computer*, vol. 24, pp. 39-48, 1991.

[28] Han, S., Rosenberg, H., Shin, K. “DOCTOR: An IntegrateD Software Fault InjeCTiOn EnviRonment”. In: *Proc. of the IEEE International Computer Performance and Dependability Symposium – IPDS´95*, Erlangen, Alemanha, pp. 204-213, 1995.

[29] Hsueh, M-C., Tsai, T., Iyer, R.: “Fault Injection Techniques and Tools”. *IEEE Computer*, pp. 75-82, 1997.

[30] IBM Autonomic Computing Initiative, Disponível na World Wide Web em http://www.research.ibm.com/autonomic/, consultado em 16/08/2005.

[31] Iyer, R. “Experimental Evaluation”. Special Issue FTCS-25 Silver Jubilee. In: *Proc. 25th IEEE Symposium on Fault Tolerant Computing*, pp. 115-132, 1995.

[32] Johansson, A. “Software Implemented Fault Injection Used for Software Evaluation”. Extended Report for I. Crnkovic and M. Larsson, Ch. 10, Artech House, ISBN 1-58053-327-2, 2002.

[33] Kalakech, A., Kanoun, K., Crouzet, Y., Arlat, A. “Benchmarking the Dependability of Windows NT, 2000 and XP”. In: *Proc. of the International Conference on Dependable Systems and Networks, DSN 2004*, Florence,

Italy, 2004.

[34] Kanawati, G., Kanawati, N., Abraham, J. “FERRARI: A Tool for the Validation of System Dependability Properties”. In: *Proc. of the 22th IEEE International Fault Tolerant Computing Symposium, FCTS´22*, pp. 336-344, 1992.

[35] Kanawati, G., Kanawati, N., Abraham, J. “FERRARI: A Flexible Software-Based Fault and Error Injection System”. IEEE Transaction on Computers, vol. 44, pp. 248-260, 1995.

[36] Karlsson, J., Gunneflo, U., Lidén, P., Torin, J. “Two Fault Injection Techniques for Test of Fault Handling Mechanisms” In: *Proc. of the International Test Conference*, pp. 140-149, 1991.

[37] Koopman, P., Sung, J., Dingman, C., Siewiorek, D. Marz, T. “Comparing Operating Systems Using Robustness Benchmarks”. In: *Proc.of The 16th International Symposium on Reliable Distributed Systems – SRDS’97*, Durham, NC, USA, pp. 72-79, 1997.

[38] Koopman P., DeVale, J. “The Exception Handling Effectiveness of POSIX Operating Systems”. *IEEE Transactions on Software Engineering*, vol. 26, pp. 837-848, 2000.

[39] 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 16/08/2006.

[40] Kropp, N., Koopman, P. & Siewiorek, D. “Automated Robustness Testing of Off-the-Shelf Software Components”. In: *Proc. 28th Fault Tolerant Computing Symposium*, pp. 230-239, 1998.

[41] Laprie, J-C. “Dependability – Its Attributes, Impairments and Means”. Predictability Dependable Computing Systems (B. Randell, J.-C. Laprie, H. Kopetz, B. Littlewood, eds). Springer, Berlin, Alemanha, CNUCE/CNR Tech. Rep. n. C95-42, pp. 3-18, 1995.

[42] Lee, I, Iyer, R. “Faults, Symptoms and Software Fault Tolerance in the Tandem GUARDIAN90 Operating System”. In: *Proc. of The 23rd IEEE International Symposium on Fault Tolerant Computing – FTCS´93*, Toulouse, France, pp. 20-29, 1993.

[43] Lee, I, Iyer, R. “Software Dependability in the Tandem GUARDIAN System”. *IEEE Transactions on Software Engineering*, vol. 21, pp. 455-467, 1995.

[44] Leite, J., Orlando G.: “Software”. In: *II SCTF*, cap. 4 do mini-curso intitulado: Introdução à Tolerância a Falhas, Campinas, SP, Brasil, 1987.

[45] Madeira, H., Rela, M., Moreira, F., Silva, J.G. “RIFLE: A General Purpose Pin-Level Fault Injector”. In: *Proc. of the First European Dependable Computing Conference- EDCC´94*, pp. 199-216, 1994.

[46] Madeira, H., Costa, D., Vieira, M. “On the Emulation of Software Faults by Software Fault Injection”. In: *Proc. of The Int. Conf. on Dependable System and Networks – DSN00*, NY, USA. 2000.

[47] Martins, E., Rubira, C., Leme, N.: “Jaca: A reflective fault injection tool based on patterns”. In: *Proc of the 2002 International Conference on Dependable Systems & Networks*, Washington D.C. USA, pp. 483-487, 2002.

[48] Mendes, N., Moraes, R., Martins, E., Madeira, H. “Jaca Tool Improvements for Speeding Up Fault Injection Campaigns”. In: *Proceedings of XIII Sessão de Ferramentas do SBES – XX Simpósio Brasileiro de Engenharia de Software – SBES’06*, Florianópolis, Brasil, pages 91-96, 2006.

[49] Moreira, F., Maia, R., Costa, D., Duro, N., Rodríguez-Dapena, P., Hjortnaes,

K. “Static and Dynamic Verification of Critical Software for Space Applications”. In: *Proc. of the Data Systems In Aerospace - DASIA 2003*, 2003.

[50] Mukherjee, A., Siewiorek, D. “Measuring Software Sependability by Robustness Benchmarking”. *IEEE Transaction on Software Engineering*, vol. 23, pp. 366-378, 1997.

[51] Ng W., Chen, P. “Systematic improvement of fault tolerance in the RIO file cache”. In: *Proc. Of The 30th IEEE Fault Tolerant Computing Symp., FTCS-29*, Madison, WI, USA, 1999.

[52] Perry, D., Evangelist, W. “An Empirical Study of Software Interface Faults”. In: *Proc. of the IEEE International Symposium on New Directions in Computing*, Trondheim, Norway, pp. 32-25, 1985.

[53] Perry, W. “Effective Methods for Software Testing”. John Wiley and Sons, Nova York, USA, 1995.

[54] Ruiz, J-C., Yuste, P. , Gil, P., Lemus, L. “On Benchmarking the Dependability of Automotive Engine Control Applications”. In: *Proc. of IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2004*, Florence, Italy, 2004.

[55] Segall, Z., Vrsalovic, D., Siewiorek, D., Kownack, J., Barton, J., Dancey, R., Robison, A., Lin, T. “FIAT – Fault Injection Based Automated Testing Environment”. In: *Proc. of the 18th IEEE International Symposium on Fault Tolerant Computing – FTCS´88*, pp. 102-107, 1988.

[56] Sullivan, M., Chillarege, R. “Software Defects and their Impact on Systems Availability – A Study of field failures on Operating Systems”. In: *Proc. of the 21st IEEE Fault Tolerant Computing Symposium, FCTS´91*, Sendai, Japan, pp. 2-9, 1991.

[57] Sullivan, M., Chillarege, R. “A comparison of Software Defects in Database Management Systems and Operating Systems”. In: *Proc. of the 22nd IEEE Fault Tolerant Computing Symposium, FCTS´92*, pp. 475-484, 1992.

[58] Tsai, T., Iyer, R. “Measuring Fault Tolerance with the FTAPE Fault Injection Tool”. In: *Proc. of the 8th International Conference on Modeling Techniques and Tools for Computer Performance Evaluation*, Heidelberg, Alemanha, pp. 26 -40, 1995.

[59] Tsai, T., Iyer, R., Jewitt, D. “An Approach to Benchmarking of Fault-Tolerant Commercial Systems”. In: *Proc. of the 26th IEEE International Fault Tolerant Computer Symposium, FCTS´26*, pp. 314-323, 1999.

[60] Tsai, T., Hsueh, M-C., Zhao, H., Kalbarczyk, Z., Iyer, R. “Stress-Based and Path-Based Fault Injection”. *IEEE Transaction on Computers*, vol. 48, pp. 1183-1201, 1999.

[61] Tsai, T., Singh, N. “Reliability Testing of Applications on Windows NT”. In: *Proc. of the IEEE International Symposium on Dependable Systems and Networks – DSN’00*, New York, NY, USA, pp. 427-436, 2000.

[62] Vieira, M., Madeira, H. “Portable Faultloads based on Operator Faults for DBMS Dependability Benchmarking”. In: *Proc. of the 28th Annual International Computer Software and Applications Conference, COMPSAC 2004*, Hong Kong, 2004.

[63] Vieira, M., Durães, J., Madeira, H. “Especificação e Validação de Benchamrks de Confiabilidade para Sistemas Transacionais”. *IEEE América Latina (IEEE Latin America Transactions)*, ISSN 1548-0992, 2005.

[64] Voas, J., Charron, F., McGraw, G., Miller, K., Friedman, M. “Predicting How Badly ‘Good’ Software can Behave”, *IEEE Software*, 1997.

[65] Voas, J., McGraw, G. “Software Fault Injection: Inoculating Programs against Errors. John Wiley & Sons, New York, EUA, 1998.

[66] Voas, Jeffrey M., “Certifying Off-the-Shelf Software Components.”. *IEEE Computer*, vol. 31, no 6, DBLP, http://dblp.uni-trier.de, pp. 53-59, 1998.

[67] Zhu, J., Mauro, J., Pramanick, I. “R3 - A Framwork for Availability Benchmarking”. In: *Proc. of the IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2003*, San Francisco, CA, USA, pp. B-86-87, 2003.

Referências Relacionadas com o Risco de Software

[68] Amland, S. "Risk-based Testing: Risk analysis fundamentals and metrics for software testing including a financial application case study". *The Journal of Systems and Software*, 53, pp. 287-295, 2000.

[69] Bach, J. "Heuristic Risk-Based Testing". *Software Testing and Engineering Magazine*, 1999.

[70] Basili V., Briand, L., Melo, W., "A Validation of Object-Oriented Design Metrics as Quality Indicators". *IEEE Transaction on Software Engineering*, vol. 22, n. 10, pp. 751-761, 1996.

[71] Chen, Y.: "Specification-based Regression Testing Measurement with Risk Analysis". Tech. Rep. Ottawa-Carlton Institute for Computer Science, School of Information Technology an Engineering, University of Ottawa, Ontario, Canada, 2002.

[72] El Emam, K.; Benlarbi, S.; Goel, N.; Rai, S. "Comparing Case-based Reasoning Classifiers for Predicting High Risk Software Components". *Systems and Software*, vol. 55, n. 3, pp. 301-320, 2001.

[73] Fenton, N.; Ohlsson, N. "Software Metrics and Risk". In: *Proc. of The 2nd European Software Measurement Conference (FESMA´99)*, 1999.

[74] Gage, D., McCormick, J. "Why Software Quality Matters", *Baseline Magazine*, pp. 32-59, 2004.

[75] Herrmann, D. "Software Safety and Reliability: Techniques, Approaches, and Standards of Key Industrial Sectors". Wiley-IEEE Computer Society Press, 1st edition, January, 2000.

[76] Hudepohl, J., Jones, W., Laque, B. "EMERALD: A Case Study in Enhancing Software Reliability". In: *Proc. of IEEE Eight Int. Symposium on Software Reliability Engineering – ISSRE98*, pp.85-91, 1998.

[77] Karolak, D. "Software Engineering Risk Management". Wiley-IEEE Computer Society Press, 1st edition, November, 1995.

[78] Khoshgoftaar, T., Halstead, R. "Process Measures for Predicting Software Quality". In: *Proc of High Assurance System Engineering Workshop – HASE'97*, 1997.

[79] Leveson, N. "Safeware, System Safety and Computers". Addison-Wesley Publishing Company, 1995.

[80] Lyu, M. "Handbook of Software Reliability Engineering". IEEE Computer Society Press, McGraw-Hill, 1996.

[81] McManus, J. "Risk Management in Software Development Projects". Butterworth-Heinemann, November, 2003.

[82] Menzies, T.; Greenwald, J.; Frank, A. "Data Mining Static Code Attributes to Learn Defect Predictors". *IEEE Transactions on Software Engineering*, vol.32, n. 11, pp. 1-12, 2007. Disponível na World Wide Web em: http://menzies.us/pdf/06learnPredict.pdf, acessado em 11/11/2006.

[83] Musa, J. "Software Reliability Engineering", McGraw-Hill, 1996.

[84] Munson, J., Khoshgoftaar, T. "Software Metrics for Reliability Assessment". Handbook of Software Reliability Engineering, Comp. Society Press, Michael R. Lyu editor, ch. 12, 1995.

[85] Ntafos, S.: "The Cost of Software Failures". In: *Proc. of IASTED Int. Conference on Software Engineering*, pp. 53-57, 1997.

[86] Peters, J., Pedrycz, W.: "Engenharia de Software". Tradução do original "An Engineering Approach", Editora Campus, Rio de Janeiro, Brasil, 2001.

[87] Pfleeger, S. “Risky Business: what we have yet to learn about risk management”. *The Journal of Systems and Software*, 53, pp. 265-273, 2000.

[88] Rosenberg, L., Stapko, R., Gallo, A. “Risk-based Object Oriented Testing”. In: *Proc of. 13th International Software / Internet Quality Week-QW*, San Francisco, California, USA, 2000.

[89] Schaefer, H.: “Strategies for Prioritizing Tests Against Deadlines – Risk based testing”. Disponível na World Wide Web em: http://home.c2i.net/schaefer/testing/risktest.doc, acessado em 29/08/05.

[90] Sherer, S. “A Cost-Effective Approach to Testing”. *IEEE Software*, vol.8, n. 2, pp. 34-40, 1991.

[91] Singpurwalla, N. “Statistical Methods in Software Engineering: Reliability and Risk”. Springer; 1st edition, August, 1999

[92] Tang, M.; Kao, M.; Chen, M. “An Empirical Study on Object-Oriented Metrics”. In: *Proc. of the Sixth International Software Metrics Symposium*, pp. 242-249, 1999.

[93] Weyuker, E.J. “Testing Component-Based Software: A Cautionary Tale”. *IEEE Software*, pp 54-59, 1998.

Referências Relacionadas com a Arquitetura de Software

[94] Allen, R., Garlan, D.: "Formalizing Architectural Connection". In: *Proc. of the 16th International Conference on Software Engineering*, IEEE Computer Society, pp. 71-80, 1994.

[95] Anderson, T. Feng, M., Riddle, S., Romanovsky, A. "Protective Wrapper Development: A Case Study". *Lecture Notes in Computer Science (LNCS)*, Vol. 2580, Spring Verlag, pp. 1-14, 2003.

[96] Chen, Z., Xul, B., Zhao, J. "An Overview of Methods for Dependence Analysis of Concurrent Programs". *ACM SIGPLAN Notices*, vol. 37, issue 8, pp. 45-52, 2002.

[97] Garlan, D., Allen, R., Ockerbloom, J.: "Architecture Mismatche: Why Reuse is so Hard". *IEEE Software*, vol. 12, n. 6, pp. 17-26, 1995.

[98] Guerra P., Rubira, C., Romanovsky, A., Lemos, R. "A Dependable Architecture for COTS-based Software Systems using Protective Wrappers". *Lecture Notes in Computer Science (LNCS),* vol. 3069, Spring Verlag, pp. 147-170, 2004.

[99] Inverardi, P., Wolf, A.L., Yankelevich, D.: "Checking Assumptions in Component Dynamics at the Architectural Level". In: *Proc. of the 2nd International Conference on Coordination Models and Languages*, Springer-Verlag, 1997

[100] Kung, D., Gao, J., Hsia, P.: "Developing an Object-Oriented Software Testing Environment". *Communications of the ACM*, vol. 38, n. 10, pp. 75-87, 1995.

[101] Lemos, R., Guerra, P., Rubira, C. "A fault-tolerant architectural approach for dependable systems. *IEEE Software*, vol. 23, n. 2, pp. 80-87, 2006.

[102] Naumovich, G., Avrunin, G.S., Clarke, L. A., Osterweil, L.J.: "Applying Static Analysis to Software Architectures". In: *Proc. of the 6th European Software Engineering Conference*, Springer-Verlag, 1997.

[103] Popstojanova, K.; Trivedi, K. "Architecture Based Approach to Reliability Assessment of Software Systems". In: *Perf. Evaluation*, vol. 45, n. 2-3, pp. 179-204, Jun/01, 2001.

[104] Saridakis, T., Issarny, V.: "Developing Dependable Systems using Software Architecture". In: *Proc. of the 1st Working IFIP Conference on Software Architecture*, pp. 83-104, 1999.

[105] Shaw, M.; Clements, P. "A Field Guide to Boxology: Preliminary Classification of Architectural Styles for Software Systems". In: *Proc. 21st International Computer Software and Applications Conference*, pp. 6-13, 1997.

[106] Sotirovski, D.: "Towards Fault-Tolerant Software Architectures"(R. Kazman, P. Kruchten, C. Verhoef, H. Van Vliet editors), Working IEEE/IFIP Conference on Software Architecture, Los Alamitos, CA, pp. 7-13, 2001.

[107] Stafford, J.A., Richardson, D.J., Wolf, A.L.: "Chainning: A Software Architecture Dependence Analysis Technique". Tech. Rep. CU-CS845-97, Department of Computer Science, University of Colorado, 1997.

[108] Szyperski, C. "Component Software: Beyond Object-Oriented Programming". Addison-Wesley, 1998.

[109] Taylor, R. N., Medvidovic, N., Anderson, K.M., Whitehead Jr.E.J., Robbins, J.E., Nies, K.A., Oreizy, P., Dubrow, D.L.: "A Component and Message-based Architectural Style for GUI Software". *IEEE Transactions on Software Engineering*, vol. 22, n. 6, pp. 390-406, 1996.

[110] Yacoub, S.; Ammar, H. "A Methodology for Architectural- Level Reliability Risk Analysis". *IEEE Transaction on Software Engineering*, vol. 28, n. 6, pp. 529-547, Jun/02, 2002.

Referências Relacionadas Complementares

[111] Beydeda S., Gruhn, V.: "State of the art in testing components". In: *Proc. 3rd International Conference on Quality Software*, 2003.

[112] Carey, M., DeWitt, D., Naughton, J. "The OO7 Benchmark". Computer Sciences Department, University of Wisconsin, Madison, 1993.

[113] Chidamber, R., Kemerer, F.: "A Metric Suite for Object-Oriented Design". *IEEE Transaction of Software Engineering*, vol. 20, n. 6, 1994.

[114] Councill, B. "Third-Party Certification and Its Required Elements". In: *Proc. The 4th Workshop on Component-Based Software Engineering (CBSE)*, Lecture Notes in Computer Science (LNCS), Springer-Verlag, Canada, 2001.

[115] Halstead, M. "Elements of Software Science". Elsevier Science Inc., New York, NY, USA, 1977.

[116] Hosmer, D.; Lemeshow, S. "Applied Logistic Regression". John Wiley & Sons, 1989.

[117] ISO/IEC 9126-1. International Organization For Standardization ISO/IEC 9126-1, Software Engineering – Software product quality – Part 1: Quality Model; Geneve ISO, 2001.

[118] ISO/IEC 25051 Software Engineering – Requirements for quality of Commercial Off-The-Shelf (COTS) software product and instructions for testing. Final Draft International Standard, 2006.

[119] Podgurski, A., Yang, C.: "Partition Testing, Stratified Sampling and Cluster Analysis". In: *Proc.of the 1st ACM SIGSOFT symposium on Foundations of software engineering*. Los Angeles, USA, pp. 169-181, 1993.

[120] Rome Laboratory (RL). "Methodology for Software Reliability Prediction and Assessment", Tech. Rep. RL-TR-92-52, vol. 1-2, 1992.

[121] Resource Standard Metrics - RSM, Version 6.1. http://msquaredtechnologies.com/m2rsm/rsm.htm. Acessado em 2005.

[122] Sommerville, I. "Engenharia de Software". Addison Wesley, 6a. Edição, 2003.

[123] Standard Performance Evaluation Corporation. Disponível na World Wide Web em: www.spec.org/, acessado em 16/08/2006.

[124] Testwell Oy Ltd. http://www.testwell.fi. Accesso em Março/06, 2006.

[125] Transaction Processing Performance Council: TPC_C Benchmarks. Disponível na World Wide Web em: http://www.tpc.org/tpcc/default.asp, consultado em 16/08/2006.

[126] Triola, M. F.: "Introcução a Estatística", 7th Edition. LTC Editor, Rio de Janeiro, 1999.

[127] Wallnau, K., Hissam, S., Seacord, R. "Building Systems from Commercial Components. SEI Series in Software Engineering. Addison-Wesley, 2002.

[128] Wallnau, K., "Software Component Certification: 10 Useful Distinctions". Tech. Rep. CMU/SEI-2004-TN-031, 2004. Disponível em www.sei.cmu.edu/publications/documents/ 04.reports/04tn031.html. Acessado em Maio 2006.

[129] Washizaki, H., Yamamoto, H., Fukazawa, Y.: A Metrics Suite for Measuring Reusability of Software Components, In: *Proc. of the Metrics'2003*, 2003.

[130] Whitehead, K. "Component-Based Development – Principles and Planning for Business Systems". Addison-Wesley, 2002.

Parte II

Trabalhos Publicados

A Parte II desta tese tem como objetivo apresentar os artigos publicados durante o desenvolvimento deste trabalho. Como uma coletânia de artigos, estes são partes integrantes deste trabalho. Dessa forma, os artigos aqui incluídos representam a reprodução, na íntegra, dos trabalhos publicados que foram referenciados na Parte I, particularmente no Capítulo 3 e no Capítulo 4. A formatação de cada um dos artigos foi conservada idêntica à publicação e incluído a numeração das páginas em atendimento à exigência de formatação da tese.

Capítulo 6 – Estratégia para Validação de Componentes

O artigo reproduzido neste capítulo foi publicado no Simpósio Latino-Americano no ano de 2003. Neste artigo foi apresentado uma estratégia para a validação da robustez de um componente. Na validação da estratégia foi utilizado um gerenciador de base de dados orientado a objetos como componente alvo e um *benchmark* de desempenho como aplicação, sendo que o *benchmark* interage com o componente para prover suas funcionalidades.

A estratégia baseada no risco do componente utilizou a técnica de injeção de falhas de interface. As interfaces entre a aplicação e o banco de dados foram escolhidas como locais para a injeção das falhas. A estratégia direciona a escolha do componente alvo, reduzindo o esforço de teste. Um resumo do artigo já foi apresentado na Seção 3.1 desta tese.

A referência deste artigo é a que segue: ***Moraes, R***., *Martins, E. "A Strategy for Validating an ODBMS Component". In: Proceedings of The First Latin-American Symposium on Dependable Computing, LADC 2003, pp. 56-68, São Paulo, Brazil, 2003*.

A Strategy for Validating an ODBMS Component using a High-Level Software Fault Injection Tool

Regina Lúcia de Oliveira Moraes[1], Eliane Martins[2]

[1] State University of Campinas (UNICAMP),
Superior Center of Technological Education (CESET), regina@ceset.unicamp.br ,
http://www.ceset.unicamp.br/~regina
[2] State University of Campinas (UNICAMP),
Institute of Computing (IC), eliane@ic.unicamp.br,
http://www.ic.unicamp.br/~eliane

Abstract. We present a strategy for the validation of an Object Oriented Database Management System (ODBMS), an off-the-shelf component software, when we deliberately introduced faults and observe its behavior in the presence of these faults. A tool designed for the injection of faults in OO applications, especially Java language systems, was used. This Software Fault Injection tool, Jaca, was developed at UNICAMP and has the ability to inject faults in objects' attributes and methods. For the experiments we used an ODBMS performance test benchmark, Wisconsin OO7. The experiments were aimed at validating the robustness of the ODBMS component in the presence of errors originated at the application. For that purpose a fault injection strategy was proposed to help answer the questions: Where to inject? What error model to use? This strategy was applied, so some results and their analysis are presented. Improvement for Jaca was addressed as an experiments' result.

1 Introduction

Software development needs to complete the life-cycle in a short time with a high level of reliability, availability, safety and security. To achieve this, software's project is reducing its custom-built software to give space to third-parties [8] or Commercial-Off-The-Shelf (COTS) components [9], such as database management systems.
Despite the increase in productivity, the use of components still presents some difficulties, especially with respect of validation and maintenance, because users generally have access only to information regarding the components' interfaces, which can be insufficient for the aforementioned activities. There is no information regarding the quality, the tests undertaken or information on dependencies that would help to analyze the impact of the component on the overall system. So, components tests are important not only to determine whether they provide the expected services but also to check whether they do not present unexpected harmful behavior. Fault injection is useful to achieve the second goal, in which faults are deliberately introduced into a system in order to observe its behavior. Through the acceleration of errors and failures this technique is valuable as it may help a user to better understand how robust the

software is, achieve reliability in it, analyze how efficient is it when recovering its normal execution after a non successful transaction, what is the impact of its detection and recovery mechanisms on the application's performance which is interacting with it.
When injecting faults into a target system, at least the following questions should be answered: how to inject faults? Which faults to inject? Where to inject in the target application? For the first question, in this study we used the Jaca tool, developed in a former project [15], which is aimed at injecting during runtime, affecting an object's interface, i.e, public attributes and methods. To answer the other questions we proposed a strategy which combines robustness tests as proposed by Ballista methodology [13], and risk-based testing [3] [25]. The strategy is applied in the validation of an ODBMS component - Ozone [21].
The objectives of the experiments carried out were: i) to validate the proposed strategy and verify its effectiveness ; ii) to evaluate Ozone's capacity of detection and recovering mechanisms iii) to evaluate the Jaca tool, its ability to inject faults in order to recommend improvements.
Section II presents some aspects of fault injection and the Jaca tool, as well as similar studies carried out with different databases. In section III, we describe the Ozone ODBMS and the benchmark used to activate Ozone's functions. Section IV, characterizes the proposed strategy, describing the sets F (Faults/errors to be injected), A (mode of Activation of the component), R (data collected during the experiments) and M (measures indicating conformities, or not, with the desired properties), the FARM model [1]. Section V, addresses the injection campaign used in the strategy's application. Section VI addresses the results obtained, while in section VII, we present our conclusions and the next studies to be developed.

2 Fault Injection Techniques

2.1 Fault Injection

Fault injection techniques have been widely used to evaluate the dependability of a system and to validate error handling mechanisms. This technique is useful to validate the solutions designed to handle with exceptional situations. Fault Injection approaches can vary according to the system life time which it is applied and to the type of faults that are injected.
This study uses the software fault injection approach, which consists of altering a system's code or state in order to emulate software faults as well as faults that occur in external components that somehow affects the software[28]. In software fault injection, faults are introduced in a prototype of the system and have the capacity to inject specific error conditions that permit the activation of those mechanisms[19]. It has become more popular due to its lower costs (it does not require specially developed circuits, as does hardware fault injection do), better versatility (it is easier to adapt codes to make fault injection in another system than adaptation of circuits) and better control, which facilitates the observation of the system during tests.
For software fault injection the more common faults considered are memory faults (alter the content of memory positions), processor faults (affect the content of registers, the result of calculus, control flux or instruction), bus faults (affect the addressing lines or data that are

being transmitted by the bus) and, in distributed systems, the communication faults (affect messages that are transmitting through a communication channel: they can be lost, altered, duplicated or delayed)[12].
Recently more attention has been given to the consequences of software faults. Software faults represent the faults resulting from mistakes committed by developers during system development or in modifications made in the phases of maintenance. Software faults have turned out to be the main cause of field failures.

2.2 The Jaca tool

Software Fault Injection can affect either the code (source or assembler) or the state of a target system. To alter a system's state, a tool is needed to inject faults or errors during runtime. These tools differ according to the mechanism used to trigger faults[12]. Most of these tools are aimed at emulating hardware faults, so faults are injected at low-level, affecting processor registers, I/O device drivers, memory positions. With the increasing importance of software faults, some studies present tools aimed to inject faults at higher level (application source code) during runtime.
Jaca offers mechanisms for the injection of high level faults in object-oriented systems written in Java language. Jaca is an evolution of the FIRE tool [24], uses reflective programming. The reflection mechanism introduces a new architectural model by the definition of two levels: the meta-level (implements fault injection and monitoring features) and the base level (implements the system's functionalities) [16]. Computational reflection allows the target system's instrumentation to carry out their functions through introspection (useful for the system's monitoring) or alter the system during runtime (useful for the injection) without changing the system's structure. Jaca does not need the application source code to perform fault injection. This occurs because Jaca was implemented using the Javassist reflection toolkit [6], which allows the instrumentation to be introduced at byte code level during load time. Jaca's current version can affect the public interface of an application by altering values of attributes, method's parameters and return values. Jaca is described in more detail in [15], [18].

2.3 Related Work.

The use of fault injection to validate component-based systems is an active research area. In what concerns the validation of DBMS components there are also some works. In [14] a server is tested in the presence of faults introduced at the NFS (Network File System). Faults were introduced by a human operator, who shut down the machine where the NFS server resided. Another work presents the availability of a client-server application based on the Microsoft SQL Server 2000 [2]. Faults were inserted at the database server using a disk emulator, a machine that is perceived as a disk by others sharing the same SCSI (Small Computer System Interface) bus, emulating in such way disk and bus faults.
In Coimbra University a group realized a series of fault injection experiments for the validation of different versions of the Oracle DBMS[9] [11] [27]. Most of these works used the XceptionNT tool, which is Windows NT version of the Xception, a fault injector

developed by the group [10]. Hardware exceptions are used to trigger a fault injector that operates at the exception handler level.
In Brazil, the Federal University of Rio Grande do Sul also realized studies in database validation using in-house developed tools. In [23], the DBMS Interbase was validated using the FiDE tool that emulates hardware faults through processor registers and memory corruption. In [17] the validation of the fault-tolerance mechanisms of the DBMS Progress is presented. Two tools developed by the group were used: FiDE, to emulate hardware faults such as memory corruption and disk I/O faults, and ComFIRM, to emulate communication faults with the database server.
Our work differs from those in basically two aspects. First, we use an OO database management system (ODBMS), instead of a relational one. Second, the errors we introduced during runtime were aimed at emulating software faults, as in [11], but fault injection was performed at high-level, affecting attributes and methods (parameters and return values) of Java applications.

3 The target system.

3.1 The Target Component

Ozone is an object manager written in Java which allows for Java persistent objects in a transactional environment. It is an Open Source project under a LGPL license (*GNU Library General Public License*). Persistent objects are programmed following the syntax of the programming language.
The Ozone environment is based on a client/server architecture, where clients connect to the database using "sockets" with a protocol that plays a similar role as RMI (Remote Method Invocation). To avoid object replication, Ozone uses a unique instance architecture, where the actual instance resides in the server. At the client, objects are controlled via "proxy" objects, which can be seen as a persistent reference. To handle these proxies, an external interface is added to each class, which is linked to the original class by the Ozone Pos Processor (OPP) generating a new source file.
Ozone can be used standalone, in which case it resides in the same machine as the client application, or in a distributed architecture, in which case it resides at the server side. The experiments performed so far use a local configuration.

3.2 The Target Application

As in other works relative to DBMS validation by fault injection, we also used a benchmark to activate the target component. Searching on the existing benchmarks, the implementation of Wisconsin OO7 [4] was found on the Ozone's website [21].
This benchmark was originally written to be used as framework for the development of CAD and CASE tools. The main component of the OO7 benchmark is a set of *composite parts*. Each composite part corresponds to a design primitive such as a procedure in a CASE application; the set of all composite parts forms the *design library*. Associated with each composite part is a *document* object, which documents the referenced composite part. The

composite part also has an associated graph of *atomic parts,* which, for example, can represent a variable in a Case procedure. Assembly objects are more complex structures, which may be composed of composite parts (base assembly) or other assembly objects (complex assembly). Assemblies are organized in hierarchies, each of which constitutes a module. Associated with a module is a manual, which documents the module.
The benchmark can create three database sizes: large, medium and small. In this study we are using the small size, which contains a single module or one assembly hierarchy with seven levels, an assembly object in one level being composed of two other assemblies. The assemblies in the lowest level are defined by composite parts with a total of 500 per module, 20 atomic parts each, giving 10,000 atomic parts[4].
The Ozone version of the benchmark implements three main functions: store objects creating an assembly hierarchy (create), search root objects (query match) and traverse the composite part objects' hierarchy, navigating from object to object (query traversal)[4]. These methods were used in the experiments reported here to check database's consistency, as is explained in VI.1.

4 The fault injection strategy.

For Jaca tool, an application can be viewed as a set of objects which interact through the exchange of messages, where the errors introduced may affect both the attributes as well as the methods (parameter and return values).
To validate Ozone's robustness we proceeded as follows: i) faults were introduced into the application (OO7) to observe how robust Ozone is with respect to errors occurred in the application using it; and ii) faults were introduced into Ozone to observe its robustness with respect to errors originated in the target component itself. This work reports results from the experiments in (i). A strategy was proposed aimed to answer the following questions: i) where to inject the faults, specifically what objects to inject and what attributes and methods to select within an object; and ii) which error model to use.

4.1 Where to inject faults

The idea of risk-based testing is to allocate test effort in the parts of code that are most error prone, and where failure would have the highest impact. So, the first task of risk-based testing is to determine how likely it is that each part of the software will fail [26].One way to determine error prone parts of a system is by collecting field data, as in [5]. When these data are not available complexity metrics can help since the parts of the code that are more complex are more prone to errors[25]. The second task is then to select the complexity metrics to be used. Once identified the metrics, it is necessary to define guidelines to assist in identifying the risky parts of the code. This is achieved by the establishment of threshold values for the metrics being considered. Classes for which two or more metrics exceed their threshold are considered as risky.
Besides complexity metrics other criteria were used: i)whether the application class has an association with Ozone classes, indicating exchange messages at execution time, propagating, in this way, the errors to the database server; ii)visibility, since we are affecting only the application's public interface, and iii)inheritance of the component.

4.2 Which error model to inject.

Based on Ballista's approach for robustness tests, together with the ones proposed in [28], the possible values to be used for each type of data are:

- Integer: 0, 1, -1, MinInt, MaxInt, neighbor value (current value ± 1)[1]
- Real floating point: 0, 1, -1, DBLMin, DBLMax, neighbor value (current value * 0.95 or * 1.05)
- Boolean: inversion of estate (true -> false; false ->true)
- String: null, biggest string, string with all ASCII, string with pernicious file modes and printf format.

These values potentially represent exceptional test values for each data type.

5 Strategy's application

The strategy can be resumed as showed bellow and will be demonstrated in this section with a case study: 1) Define the complexity metrics to be used; 2) Establish a threshold for these metrics; 3) Obtain the metrics for the target system classes; 4) Identify the classes which have direct interaction with ODBMS; 5) Identify the risky classes: the ones with one or more metrics whose values are beyond the threshold and which satisfy criteria in 4; 6) For each class, apply the same criteria in 2 and 3 for their methods; 7) For each of those methods choose the parameters or return values of types compatible with those types that Jaca can injected; 8)Write a fault specification for each critical value aforementioned in accordance with the selected error model.

5.1 Identification of classes

The benchmark has 22 classes, divided in 11 interface classes and 11 classes that implement these interfaces. Utilizing RSM software[26], Panorama tool[22] and the Wisconsin OO7 benchmark classes diagram, we calculated the OO software complexity metrics called CK metrics[7], of all these classes and took the classes which have metrics out of the accepted values. The CK metrics were chosen because they were implemented by the quoted tools. From CK metrics we used the following metrics: **NOM** – Number of Methods, **WMC** – Weighted Methods per Class (sum of methods in a class), **DIT** – Depth of Inheritance Tree (The depth of a class within the inheritance hierarchy, i.e, number of ancestor classes), **NOC** – Number Of Children (The number of children, immediate subclasses subordinate to a class in the hierarchy). Table V.1 presents the OO7's classes and their corresponding metrics. The values below the metric's names (showed between parentheses) indicate the ideal and the maximum acceptable values of each metric for Java applications[25]. These values were obtained in a research with over 20,000 classes from programs written in both C++ and Java[25]. So, we can use these values, since we are testing Java language applications.

[1] According to [27] the neighbor value (domain twiddle) must not create values out of the limits for the data type.

The last three columns of Table V.1 show the other criteria considered for class selection: classes which directly interact with the ODBMS's classes; classes that have public visibility and classes which inherits from Ozone's interface. This procedure cover step one to five from the proposed strategy.

Table V.1. Identification of classes to be injected

Class	WMC (25;40) [2]	NOM (0;50)	DIT (2;5)	NOC	Direct Association with Component	PublVisibility.?	Inher the component
BenchmarkImpl	45	13	1	0	Yes	Yes	Yes
OO7_Manual Impl	10	10	1	0	Yes	Yes	Yes
OO7_ConnectionImpl	9	9	1	0	Yes	Yes	Yes
OO7_DocumentImpl	7	7	1	0	Yes	Yes	Yes
OO7_DesignObjectImpl	7	7	1	0	Yes	Yes	Yes
OO7_ModuleImpl	7	7	2	0	No	Yes	Yes
OO7_CompositePartImpl	11	11	2	0	No	Yes	Yes
OO7_BaseAssemblyImpl	5	5	3	0	No	Yes	Yes
OO7_ComplexAssemblyImpl	3	3	3	0	No	Yes	Yes
OO7_AssemblyImpl	5	5	2	0	No	Yes	Yes
OO7_AtomicPartImpl	13	13	2	0	No	Yes	Yes

Table V.2: Methods of the Classes to be injected by Jaca

Class	Public Attributes	Public Methods				
		Name	Parameter Type	Return Type	Cyclom Complexity	Uses Component
BenchmarkImpl	None	main create getAtomicPartOid	args:String[] anScale:Int None	None None Long	**11** 1 1	No Yes **Yes**
OO7_ManualImpl	None	setTitle setId setText title Id text	x:String x:Long x:String None None None	None None None String Long String	1 1 1 1 1 1	No No No No No No

5.2 Identification of methods and attributes to be injected

[2] Ideal and acceptable limit values according to [25]. These values were obtained over three years tests, where over 20,000 Java classes where collected and analyzed.

Having chosen the classes, the next step consists of selecting public methods and attributes which have data types compatible with those that Jaca can inject. Amongst all possibilities of injection we look for methods which have high complexity and methods which use any component's methods. Then, we choose, from these methods, parameters and return values having compatible data types. The errors were uniformly distributed among these parameters. Table V.2 presents some classes and their respective methods, parameters and return values. For sake of space we do not show the totality of the methods here. OO7 classes have no public attributes. The procedures described here cover the step 6 from the strategy.

5.3 Identification of values to be injected

Next we determine values for parameters and return of methods. Table V.3 shows part of the table which presents the injected values and the operation realized by Jaca to introduce these values. The apostrophe (') associated with the parameter's name indicates the final value obtained. It is important to mention that for the MaxInt and MinInt (32767 and –32768 respectively) we used an operand a little smaller than the extreme value (32760 was used) to avoid an overflow value. This procedure covers the strategy's seventh step. As the last step, we create the specification and monitor classes files, based on Table V.3.

Table V.3: Injected values

Parameter: Type→Method	Operation used for injection[3]
anScale:Int→ create	Addition (+) anScale'= anScale + 1
	Subtraction (-) anScale'= anScale – 1
	Addition (+) anScale'= anScale + 32760 (extreme integer's values)
	Subtraction (-) anScale'= anScale – 32760 (extrem integer's values)
	Multiplication (*) anScale'= anScale * 0 (critical integer's values)

6 Results and Analysis

As local configuration was used for these experiments, a single machine was utilized, where Ozone and OO7 reside. This machine is a notebook with a Pentium III processor, 366 MHz, 64 MB of RAM memory, and 6 GB of hard disk. The operational system used was Conectiva

[3] Operations implemented by Jaca according to the type of attribute / parameter / return of method

Linux 6.0 (Linux version 2.2.17.14d). Future tests will be done with a client-server platform, to verify the other aspects related to communication. Further results are presented in [20].

6.1 Collected Data

Our objective is to observe Ozone's behavior in the presence of faults. To that end, data are collected from several sources. (i) From Ozone's interface: number of stored clusters, exceptions thrown by Ozone, but not treated. (ii) From Benchmark's interface on Jaca: exceptions thrown that are not treated by the application. (iii) From Ozone's log: data that are not similar to those of Ozone's interface. (iv) From Jaca's log: specification of the injected faults. (v) From stored data: to determine whether database consistency was guaranteed. (vi) To check other database properties we implemented further traversals and queries, but for the results presented here only the functions already present at OO7 were needed.

6.2 Characterization of Ozone's Behavior

In order to characterize Ozone's behavior in the presence of application faults we can envisage the following situations: i) ideal case, in which both OO7 and Ozone have normal termination and the database created is in a consistent state. ii) an exception is generated at OO7, as consequence of fault injection, but Ozone has normal termination and the database created is in a consistent state. This case characterizes the robustness of Ozone with respect to application failures; iii) an exception is thrown by Ozone, which terminates abnormally, but the database created is in a consistent state; and finally, iv) Ozone terminates abnormally and the database created is in an inconsistent state. This case characterizes failure of the database manager, in that it allows stored data to be corrupted in consequence of non-successful transaction.

To certify the consistent state of the database, we developed additional functions in order to verify if all transactions committed were in fact stored in the database and no data was lost. We verified all requests in each execution of the benchmark considering the actual request and if interruption occurs, we verify the stored data to certify the non-residual data.

6.3 Description of the fault injection campaign

Table VI.1 presents the experiments performed. The first column gives an ID for the groups of experiments. The second column indicates whether fault injection was triggered at the first method's invocation or later. We also vary the repetition pattern by injecting faults permanently (at all method's invocations), transiently (faults are injected only once) and intermittently (faults are injected at predefined periods). The third column shows the number of method's parameters or return values of types compatible with those types that Jaca can injected. The fourth column presents the patterns values to numeric types based on Ballista's approach (five patterns values) multiplied by five as each fault was repeated five times as suggested in [4]. Finally, the last column presents the total experiments per start time and repetition pattern.

The sequence of operations used for test was: 1) to open the connection with the Ozone component; 2) to provoke the injection activating the faults through the benchmark; 3) to close

the database connection, in case it remains opened after fault injection; 4) to reopen the connection with the database; 5) to verify the stored files through a Query Match to check whether all atomic parts were created; 6) to verify the stored files through a Query Traversal to check whether the assembly hierarchy was correctly created; 7) to close the connection with the Ozone component; 8) to store the results.

Table VI.1 – Distribution of Tests

Classes of Experiments	Start Time	Repetition Pattern	N°of Integer/ Real Parameters/ Return Values	N° of Injection based on Balista	Total Injection
1P	First Occurrence	Permanent	18	25	450
1T		Transient	18	25	450
1I		Intermittent	18	25	450
2P	After First Occurrence	Permanent	18	25	450
2T		Transient	18	25	450
2I		Intermittent	18	25	450
	Total N° of Tests				**2700**

6.4 Results

The 2700 experiments were uniformly distributed among all methods that gave a total of 150 experiments per method. A total of 45 failures occurred in this campaign, all of them occurred for the class BenchmarkImpl_Proxy when method *create* was injected. These failures occurred for experiments 1P, 1I and 1T,(at the first method's invocation). These results were observed for the error models: neighbor positive value, MAXINT and MININT. The injection executed with a positive neighbor value, causes the increase of the quantity of information to be stored (observed through the number clusters presented on Ozone's interface), once the injected parameter was used to control the size of the database. Due to lack of storage or memory resources, the application was not capable of being completed and it caused an exception.

In fact, case (iv) mentioned in section VI.2 occurred, in that an exception was raised by OO7 as well as by Ozone and accused on both interfaces, but the object database did not recover its state. This was observed when injecting a positive neighbor value (see VI.1) in which case an exception occurs and the execution was interrupted. After the occurrence, we did a new connection with the database and invoked a query match, where no errors were detected, that is, the root objects were adequately created. However, when the query traversal was invoked, a *NullPointerException* occurred, indicating that the assembly hierarchy was only partially

created. Similar situation occurs when injecting maximum and minimum integer or real values. An exception was also raised for both OO7 and Ozone, in the beginning of the execution (*Array Index Out of Bound*) and the same inconsistency on Ozone was observed.
The same tests were conducted by injecting faults beyond the first interaction and none presented failures, not even those that fail during the first campaign, since the fault location *create* method is invoked at the beginning of the OO7 application execution to define the database's size and create the assembly hierarchy according to the defined size , but it is not activated later.
It is worth noting that experiments were performed with Ozone version 1.0. When reporting these results to their developers, they said these problems in Ozone's recovery mechanism was already corrected in version 1.1. In what concerns the fault injection strategy proposed, we could notice that failures occurred, as expected, when injecting the class that presented greater complexity and interacted with the database component. However, differently of what was expected, it was the method *create*, and not the method *main*, where failures occurred.

7 Conclusion and Future Works

This paper presented a strategy for fault injection validation of an ODBMS component, Ozone. To activate Ozone's functions we used a benchmark application, OO7, that was found in the Ozone's Website. For fault injection we used a tool, Jaca, developed to inject faults in applications written in this language. Jaca injects interface faults, that is, it affects methods and attributes at an object's public interface.
The strategy was proposed to help select objects and methods for injection based on potential risk. To identify the risky objects and methods the strategy considers several factors: complexity metrics, interaction with the Ozone component, among others.
The experiments reported here were aimed at verifying Ozone's behavior in the presence of faults in the application. From the 2700 experiments performed, failure occurrence was observed in 45 of them, which was characterized by database corruption.
Results obtained confirmed in part what was inferred by the fault injection strategy, that is, the class with greater complexity that interacts with the Ozone was the one responsible for the observed failures. On the other hand, it was not the method with greater complexity, but the one which implemented a critical function that was responsible for the results reported above. This indicates that the strategy must be further improved to consider other criteria, such as the frequency of use, the importance of the class or method to the target application, among others.
Although further experiments are required, we can state that: i) a strategy based on risk is useful for selecting fault locations when no field data is available; and ii) Jaca is useful to validate off-the-shelf components, for which only the API is available.
Further experiments are being carried out, this time injecting faults directly into the target component's classes. In the long term we also envisage to validate Ozone version 1.1 to compare the improvements with respect to the actual version.

Acknowledgments: Roberto Carlos Torres who implemented an interface to the Jaca tool, making it more user friendly, and also Fabiana Cristina de Souza Alves for implementing new transactions to the actual Ozone's site OO7 benchmark.

References

1. 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*, 16 (2), Feb/1990, pag 166-182.
2. Brown, A. "Availability Benchmarking of a Database Systems", EECS Computer Science Division, University of California at Berkeley, 2000.
3. Bach, J "Heuristic risk-based testing" , Software Testing and Quality Engineering Magazine, Nov/1999.
4. Carey, M. J. ; DeWitt, D. J. ; Naughton, J. F. "The OO7 Benchmark" http://www.columbia.edu/, 1994.
5. Chillarege, R.; Christmansson, J "Generation of an Error Set that Emulates Software Faults-Based on Fields Data, 26th International Syposium on Fault-Tolerant Computing, pp 304-13, Sendai, Japan Jun/1996.
6. Chiba, Shigeru. "Javassist – A Reflection-based Programming Wizard for Java". *Proc of the ACM OOPSLA'98 Workshop on Reflective Programming in C++ and Java*, Oct/1998.
7. Chidamber; Kemerer "Principal Components of Orthogonal Object-Oriented Metrics" http://satc.gsfc.nasa.gov, 1994.
8. Clapp,J.A;Taub, A "Management Guide to Software Maintenance in COTS-Based Systems", Eletronic Systems Center, Nov/1998.
9. Costa, D.; Madeira, H "Experimental Assessment of COTS DBMS Robustness under Transient Faults" , Pacific Rim Dependability Computing, Hong Kong,,1999.
10. Carreira, J.; Madeira, H.; Silva, J. G. "Xception: Software Fault Injection and Monitoring in Processor Functional Units". *5th IFIP International Working Conference on Dependable Computing for Critical Applications*. Urbana-Champaign, EUA, 1995, _ág 135-149.
11. Costa, D.; Rilho, T.; Madeira, H " Joint Evaluation of Performance and Robustness of a COTS DBMS through Fault Injection" , New York, DSN 2000.
12. Hsueh, Mei-Chen; Tsai, Timothy; Iyer, Ravishankar. "Fault Injection Techniques and Tools". *IEEE Computer*, Apr/1997, pag 75-82.
13. Koopman, Phil; Siewiorek, Dan; DeVale, Kobey; DeVale, John; Fernsler, Kim; Guttendorf, Dave; Kropp, Nathan; Pan, Jiantao; Shelton, Charles; Shi, Ying "Ballista Project : COTS Software Robustness Testing", Carnegie Mellon University, http://www.ece.cmu.edu/~koopman/ballista/, 2003.
14. Lee, J. Bill; Herwadkar, Rahul V. "Oracle Network Storage System Compatibility Fault Injection Tests" Oracle, www.oracle.com, March/1999.
15. Leme, Nelson G. M. "A Software Fault Injection Systems based on Patterns" Master Thesis, UNICAMP, Brasil, 2001. (in Portuguese).
16. Maes P "Concepts and Experiments in Computational Reflection" Proc. OOPSLA'87, p. 147-155, 1987.
17. Manfredini, Ricardo Augusto, "Conduction of Injection Fault's experiments in Distributed Database", Máster Thesis, Federal University of Rio Grande do Sul, supervising Profa. Dra. Taysy Silva Weber, 2001 (in Portuguese).
18. Martins, E.; Rubira, C. M. F.; Leme N.G.M. "Jaca: A reflective fault injection tool based on patterns" *Proc of the 2002 Intern Conference on Dependable Systems & Networks, Washington D.C. USA*, 23-267 June/2002 pag 483-487.
19. Martins, Eliane "Injection Faults in dependable systems", I Regional Symposium of Fault Tolerance Systems, Campinas, 1996 (in Portuguese), pag 181-196.
20. Moraes, R; Martins, E "Testing Component-based Applications in the Presence of Faults" Proc.of the 7th World Multiconference on Systemics, Cybernetics and Informatics (SCI 2003)", Jul/2003
21. Site of Ozone – Object Oriented Database Management System www.ozone-db.org/, 2002.
22. Site of Panorama Tool - www.softwareautomation.com/, 2003.
23. Rodegheri, P. R. "Validation of Fault Tolerance Mechanisms of SGBD Interbase through Fault Injection", Master Thesis, UFRGS, 2001 (in Portuguese).
24. Rosa, Amanda. *Uma Arquitetura Reflexiva para Injetar Falhas em Aplicações Orientadas a Objetos*. Master Thesis, UNICAMP, Campinas, Brasil, 1998. (in Portuguese).
25. Rosenberg, L; Stapko, R; Gallo, A "Risk-based Object Oriented Testing ", 13th International Software / Internet Quality Week (QW2000) , San Francisco, California USA, 2000.
26. Resource Standard Metrics, Version 6.1, http://msquaredtechnologies.com/m2rsm/rsm.htm, 2003.
27. Vieira, M.; Madeira, H "Recovery and Performance Balance of COTS DBMS in Presence of Operator Fault" , IPDS 2002, Bethesda, Wahington DC, 2002.
28. Voas, Jeffrey; McGraw, Gary. Software Fault Injection: Inoculating Programs against Errors. John Wiley & Sons, New York, EUA, 1998.

Capítulo 7 – Amostragem Estratificada como Critério de Análise de Risco

O artigo reproduzido neste capítulo foi publicado no Simpósio Latino-Americano no ano de 2005. Neste artigo as métricas de complexidade foram utilizadas como critério para definir partições (ou estratos) formadas pelas classes do componente. As classes do gerenciador de base de dados orientado a objetos (componente alvo) apresentavam métricas dentro de limites estipulados por valores padrão de complexidade (*threshold values*) e classes com métricas acima destes valores. Com base nesta divisão, uma amostra estratificada composta por dois estratos foi definida com base na teoria da amostragem.

Como o componente era composto de muitas classes, teria sido impossível considerar todas as classes como sendo alvo das injeções de falhas. Dessa forma, usou-se a amostra estratificada escolhendo dentro de cada estrato os pontos de injeção de falhas.

Os resultados não confirmaram nossas expectativas, uma vez que a partição que continha classes com a métrica WMC acima do valor padrão não produziu os defeitos mais severos (corrupção do banco de dados). Concluímos que uma métrica não é suficiente para a escolha das partições, sendo necessários outros fatores como, por exemplo, as dependências entre o gerenciador de Base de Dados e a aplicação. Um resumo do artigo já foi apresentado na Seção 3.2 desta tese.

Sua referência é a que segue: ***Moraes, R**., Martins, E., Catapani, E., Mendes, N. "Using Stratified Sampling for Fault Injection". In:Proceedings of the Second Latin-American Symposium on Dependable Computing, LADC 2005, pp. 9-19, São Paulo, Brazil, 2005*.

Using Stratified Sampling for Fault Injection

Regina Lúcia O. de Moraes[1], Eliane Martins[2], Elaine C. Catapani Poletti[1], Naaliel Vicente Mendes[1]

[1]Superior Centre of Technological Education (CESET) State University of Campinas (UNICAMP)
{regina, elainec, naalielb@ceset.unicamp.br}
Phone: +55 19 3788-5872/ Fax: +55 19 3404-7164
[2]Institute of Computing (IC) State University of Campinas (UNICAMP)
{eliane@ic.unicamp.br}
Phone: +55 19 3404-7165/ Fax: +55 19 3788-5847

Abstract. In a previous work we validated an ODBMS component injecting errors in the application's interface. The aim was to observe the robustness of the component when the application that interacted with it failed. In this work we tackle the injection of errors directly into the interfaces among the target component's classes. As the component under test has several classes, we use stratified sampling to reduce the amount of injections without losing the ability to detect faults. Strata are defined based on a complexity metric, Weighted Methods in a Class – WMC. Experiments show that this metric alone is not sufficient to select strata for testing purposes.

1 Introduction

Increased pressures on time and money make component-based software development a current trend in constructing new systems. The development of a system that is an integration of several Off-The-Shelf (OTS) components brings hypothetical benefits, such as system quality enhancement, since the components are used in other systems, and time and money savings, since the source code does not need to be rewritten. Moreover, components and component-based system validation is still a challenge.

The difficulty stems from the degree of knowledge that developers and users have about the component [2] [18]. When developing a component, the developer cannot picture every possible use this component may have in the future. Component users do not know the acquired component's quality level, and even if it is known, there is no guarantee that the component will present the same quality level when used in a new context. Furthermore, the use of high-quality components does not guarantee that the overall system will have high quality, due to the complexity of interaction among components [19].

Component validation is therefore a very important task. It allows us to determine whether the component provides the expected services, and to check whether it does not present unexpected harmful behaviour. Fault injection is a useful technique in which faults or errors are deliberately introduced into a system in order to observe its behaviour and thus better understand how robust the software is, how efficient is it when recovering

its normal execution after a non-successful transaction, and the impact of its detection and recovery mechanisms on the application's performance.
In a previous work [10] we validated the Object-Oriented Database Management System (ODBMS) Ozone [12], an OTS component, aimed at evaluating its robustness in the presence of errors originated in the application. The benchmark Wisconsin OO7 was our target application. The Jaca tool [9] was used to inject errors at the interface between OO7 and Ozone. A risk-based strategy [1] [15] was proposed and applied for the selection of OO7 classes in which to inject. In that work, we injected in the selected classes and in all OO7 classes to evaluate the effectiveness of the strategy and to compare the results.
In this work the component under test has several classes and injection in all classes would be too time-consuming. We consider stratified sample and ratio theory to determine the number of elements that allows us to get a confidence in the sample. The number of elements taken from each stratum conserves the same proportion presented by the set of all component classes. One difficulty with stratified sampling is the determination of the strata. In order to address this difficulty, in this work we present a risk-based strategy used in [10]. Section 4 briefly presents this strategy. The reminder of the paper is organized as follows. Ozone as well as OO7 are presented in Section 2. Fault Injection fundamentals as well as some related works are shortly presented in Section 3. The results of the stratified sampling strategy applied to the case study are presented in Section 5. Finally, Section 6 concludes this work.

2 Case study description

The case study used for strategy testing is a system composed by two main components, an ODBMS called Ozone and the OO7, a benchmark used to evaluate ODBMS performance. In this experiment the benchmark is seen as the application responsible for the activation of injected faults and the propagation of errors to the component under test, the Ozone database.

2.1 The Target Component

Our target component is an object database management system (ODBMS) called Ozone [12]. Written in Java, it allows Java objects in a transactional environment to persist according to the structure defined by the application. Based on client-server architecture, clients connect to the database using sockets with a RMI protocol. To guarantee a unique instance in the server, Ozone uses "proxy" objects that can be seen as a persistent reference. These proxies are generated by the Ozone Post Processor (OPP) as a result of two linked files, the class file and an external interface that is created for each class. The experiments performed in this work use a local configuration, but Ozone can also be used in a distributed architecture.

2.2 The Target Application

We use Wisconsin OO7 [3] as a benchmark application to activate the target component.

Wisconsin OO7 was found in Ozone's website [12] and was originally used to evaluate the ODBMS performance.
The main component of the benchmark is a set of composite parts. Each composite part has a document object and a graph of associated atomic parts. A set of all composite parts forms the design library. Assembly objects are more complex structures, which may be composed of composite parts (base assembly) or other assembly objects (complex assembly). These assemblies are organized in hierarchies; each of which constitutes a module. There is a manual to document each module.
The Ozone's version of OO7 implements three main functionalities, one to store objects and create an assembly hierarchy (create), one to search root objects (query match), and another to traverse the composite part objects' hierarchy (query traversal) [3]. To check the ACID properties (Atomicity, Consistency, Isolation and Durability) we implement extra functionalities that are based on TPC-C benchmark [16] and OO7 specification [3]. The extra functionalities are used to check the database state before and after an experiment execution. They are described in more detail in [10]. In short, to verify atomicity we use OO7 queries and other queries created, and then compare the stored data before and after fault injection. To check consistency, a query is performed to verify whether the new data stored in the database is in accordance with OO7 specification. Durability is checked by disconnecting and connecting the database and comparing its state through the queries results. As the experiments are performed in a local machine, isolation is not checked.
Among three possible sizes of the database created by OO7, this work uses the smallest one, which contains one assembly hierarchy with seven levels, composed of two other assemblies. Composite parts with a total of 500 per module define the assemblies in the lowest level. Each composite part contains 20 atomic parts, comprising a total of 10,000 atomic parts [3].

3 Software fault injection

Fault injection is a technique that simulates anomalies by introducing faults into the systems under test and then observing their behaviour. Among the various existing fault injection approaches (see [7] for an overview), software fault injection has been widely adopted. It can be used to simulate internal faults, as well as faults that occur in external components interacting through interfaces [18]. One approach of software-implemented fault injection consists of injecting anomalous input data that comes into the software through its interface [20]. This study uses this approach, allowing software acquirers to determine its robustness. The software can be stated as robust if it is fed by anomalous input without propagating the error that may cause a failure. This demonstrates that the software can produce dependable service even in presence of an aggressive external environment [20].
To apply this approach, a tool is needed to inject faults during runtime. We use Jaca [9], a software-implemented fault injection tool that offers mechanisms for the injection of interface faults in Java object-oriented systems. Jaca is source code independent, allowing the validation of a system that may be composed of multiple third-parties components.

Jaca's current version can affect the public interface of a component by altering values of attributes, method parameters and returns. These values must be simple (integer, float and Boolean), strings or objects. Jaca is described in more detail in [9].
A similar approach is presented by Ballista [8] and Mafalda tools [6], but in those cases the errors are injected in the parameters of operating system calls instead of the components interfaces. As with the Mafalda tool, we also consider the errors published by Ballista approach. TAMMER [5] is another similar work in which the injection of interface faults is used to observe fault propagation focused on code coverage, while in our work we are interested in the exceptions raised as well as whether these exceptions cause the whole system to fail. Unlike TAMMER, we do not need the source code.

4 Characterization of the Experiments

The target component Ozone contains 430 classes. In this case, injecting all classes would be a hard and unpractical work. We need a way to select a sample of classes to be injected. For this purpose we use stratified sampling. Stratified sampling and partition testing are presented in [13] to estimate reliability. In our work we use them to characterize the strata in order to test the robustness of a component-based system.
To use stratified sampling, the steps needed to define an experiment are the following: (1) define the stratification criteria and categorize Ozone classes in each stratum; (2) calculate the sample size; (3) apply the theory of proportion to determine the sample size for each stratum; (4) select the classes that belong to each stratum in the sample; (5) characterize the fault injection campaign.

4.1 Definition of the Stratification Criteria and the Strata

Stratified sampling is a sample technique that divides, based on any criteria, the population into strata and then associates another method to select the elements that should compose the sample. Partition testing can be considered a kind of stratified sampling, in which a system input domain is divided into partitions according to operational behaviour, and one input from each stratum is selected. One difficulty in stratified sampling is defining the stratum. In this work we use a risk-based strategy to determine the strata. In a previous work we used a set of complexity metrics, namely, the CK metrics suite, to determine class complexity [4]. According to a pre-specified threshold for each metric obtained in an experimental study with several real world classes [15], we were able to define the classes with high complexity, i.e., those for which one or more metric values lie outside the threshold. In this study we select one metric of the CK suite, the Weighted Method for a Class, or WMC. The WMC metric represents the complexity of a class in terms of the number of its methods and their complexities, and thus it is reasonable to consider them as a first choice for our assessment. The assumption is that the higher the WMC, the higher the error proneness of the class, making it a good candidate for fault injection. Thus, we calculated the WMC metric of all Ozone classes.

Based on the WMC metric obtained, we classify all the Ozone's classes and separate them in two strata according to the WMC metric thresholds: (S1) WMC metric is equal or smaller; (S2) WMC metric is greater [15].

4.2 Calculating the Sample Size

To estimate the sample size we need to determine the percentage of success and non-success, the confidence level and the error tolerance.

$$n = \frac{\left(Z_{\alpha/2}\right)^2 . \hat{p} . \hat{q}}{E^2} = 30{,}1181 \equiv 31 \text{ classes}$$

Where:

$E \equiv 0.05$ (5% - error tolerance)

$Z_{\alpha/2} \equiv 1.96$ (critical value related to the reliability on the failure ratio - 95%)

$\hat{p} \equiv 0.02$ (failure ratio based on the previous experiments = 45 failures / 2700 experiments)

Fig. 1. **Sample Size Estimation [17]**

From the failure ratio of previous experiments [10], 45 out of 2700 experiments performed resulted in failure. In this way, the value of $\hat{p}$ is 45/2700, which is approximately 2%; thus, the complementary value, $\hat{q}$ is 98%. The confidence level considered is 95%, which implies a critical value of 1.96 and an error tolerance of 5% (complementary percentage related to 95%). Based on these values we obtained a sample size of 31 classes [17]. Figure 1 presents the sample size estimator.

4.3 Obtaining the Sample Size for each Stratum

Given that NS1 represents the number of classes in stratum S1, NS2 the number of classes in S2, N the number of Ozone classes and n the estimated sample size, then the sample size (nSx) for a stratum (x) is given by: $nSx \equiv NSx / N * n$ and $\sum nSx = n$, according to the theory of proportions mentioned in Section 4. Since stratum S1 represents 89% of Ozone classes and stratum S2 11% of the total of classes, and considering a sample size of 31 classes, we need to select 27 classes from stratum S1 and 4 classes from stratum S2.

4.4 Selecting the Classes in each Stratum

To select the classes to be sampled, we rank the classes in each stratum (Sx) in a decreasing order based on the WMC metric. Then we take nSx classes from the top WMC. We also take into account the class's visibility since we can inject only in a public class. Among the top classes of stratum S1 and stratum S2 there are classes in which is not possible to inject due to technical restrictions (all methods are protected).
So we consider the next one in the rank. From now on, we inject firstly into the classes that belong to the stratum S2, secondly into all classes of the sample, and only then compare the results. To confirm our strategy, no major different failures should occur when we compare both results.

4.5 Characterization of the Fault Injection Campaign

A fault injection campaign is characterized by a faultload, a workload and readouts to be collected. A faultload describes the set of faults that are going to be inserted in the target system, defined according to the fault representativeness and the established fault selection criteria [18]. A faultload is determined by a fault location, a fault's type, triggering condition, repetition pattern and injection start. These elements can be described as follows:

- *(i)* *Fault Location*: In this work we inject interface faults [21].
- *(ii)* *Fault's Type:* Corruption of the parameters and returned values, replacing them with invalid values, combining the Ballista approach [8] with boundary value testing [14] (based on the system's specification).
- *(iii)* *Triggering Condition*: interception of operation calls at the component interfaces.
- *(iv)* *Repetition Pattern*: the frequency of the injection (permanent, intermittent or transient).
- *(v)* *Injection Start*: how many times an operation must be called before the first injection.

The values to be injected are based on Ballista approach for robustness tests, together with the ones proposed in [18]. Table 1 presents these values, which should be chosen according to the parameters' or returned values' data type.

Table 1. Values to Inject based on the Ballista Approach

Data Type	Values to Inject
Integer	0, 1, -1, MinInt, MaxInt, neighbour value (current value ± 1)
Real Floating Point	0, 1, -1, DBLMin, DBLMax, neighbour value (current value * 0.95 or * 1.05)
Boolean	inversion of state (true -> false; false ->true)
String	Null

The workload is the based program(s) that run(s) on the system when the experiment is conducted [18]. In this study the workload is the Benchmark Wisconsin OO7.
The readouts are collected from several sources: (i) From Ozone's interface, we extract the number of stored clusters, and the exceptions thrown by Ozone that were not treated. (ii) From Benchmark's interface on Jaca, we extract exceptions thrown that are not treated by the application. (iii) From Ozone's log, we take out data that are not similar to those of Ozone's interface. (iv) From Jaca's log, we extract specification of the injected faults and exceptions raised. (v) From stored data, we determine whether database consistency is guaranteed and we perform the existent queries to verify if all committed transactions were stored in the database. If an interruption occurs, we need to verify the stored data to certify the non-residual data. These outcomes are used in this study to characterize Ozone's behaviour in the presence of faults.
Ozone's behaviour can be characterized as follows: I) is the ideal case, in which both OO7 and Ozone have normal termination and the database created is in a consistent state. EXC OO7) is an exception generated at OO7 as a consequence of fault injection, but Ozone has normal termination and the database created is in a consistent state. This case characterizes the robustness of Ozone with respect to application failures. EXC OZ) is an exception thrown by Ozone, which terminates abnormally but the database created is in a consistent state. N) is used when Ozone terminates normally but the database created is in an inconsistent state, which violates the ACID properties. Finally, A) occurs when Ozone terminates abnormally and the database created is in an inconsistent state. Types N and A characterize failure of the database manager, in that it allows stored data to be corrupted as a result of non-successful transaction.
An error is said to have been tolerated when the system does not crash and the ACID properties are kept; a failure occurs when the system crashes or the ACID properties are not kept. A non-effective error is an error that causes no change in the system, and an error is considered non-detected when the system does not perceive the occurrence of an error and a failure.

5 Experimental Results

5.1 Strata Definition

Table 2 presents the selected Ozone classes in which to inject according to the strategy described in Section 4. As described in Section 4.3, the sample size should be composed by 27 classes from stratum S1 and 4 classes from stratum S2. The classes JavaCodeAnalyzer, Table and NumberLineEmitter are not considered since it is impossible to inject into them due to technical restrictions (all its methods are protected), as explained in Section 4.4.

Table 2. Selection of Ozone's classes to be injected

Class	WMC (25;40)[4]	Stratum	Can Inject into this Class?
SAXChunkProducer	133	S2	YES
WizardStore	117	S2	YES
JavaCodeAnalyzer	113	S2	NO
ProxyGenerator	111	S2	YES
NodeImpl	108	S2	YES
HTMLTableRowElementImpl	39	S1	YES
ParamEntity	38	S1	YES
OzoneODMGTransaction	38	S1	YES
CDHelper	38	S1	YES
CollectionImpl	37	S1	YES
CXMLContentHandler	37	S1	YES
HTMLObjectElementImpl	36	S1	YES
DbCacheChunk	35	S1	YES
SimpleArrayList	35	S1	YES
Table	35	S1	NO
AbsoluteLayout	34	S1	YES
DatabaseImpl	34	S1	YES
OzoneXAResource	34	S1	YES
CharacterDataImpl	33	S1	YES
CollectionImpl	33	S1	YES
DxAbstractCollection	33	S1	YES
HTMLElementImpl	33	S1	YES
NumberLinesEmitter	33	S1	NO
OPP	33	S1	YES
BLOB	32	S1	YES
DxMultiMap	32	S1	YES
HashtableContentHandler	31	S1	YES
AbstractObjectContainer	30	S1	YES
AdminObjectContainer	30	S1	YES
DocumentImpl	29	S1	YES
HTMLAnchorElementImpl	29	S1	YES
HTMLSelectElementImpl	29	S1	YES
Enh Properties	28	S1	YES
ExternalTransaction	28	S1	YES

[4] Ideal and acceptable limit values according to [15]. These values were obtained in tests conducted over a period of three years, in which over 20,000 Java classes where collected and analyzed.

The values below the metric's name (between parentheses) indicate the ideal and the maximum acceptable values of each metrics for Java applications (threshold values) [15]. The signaled classes are those with WMC metric greater than the maximum acceptable values.

5.2 Fault Injection Campaign

A total of 31 injection points with integer, long, string and objects data type are injected. We also vary the repetition pattern and the start time. Table 3 resumes the campaign.

Table 3. The Campaign Experiments' Distribution

Classes of Experiments	Start Time	Repetition Pattern	Number of Parameters/ Return Values Injected in S1	Number of Parameters/ Return Values Injected in S2	Total Injection
O1P	First Occurrence	Permanent	31	4	35
O1T		Transient	31	4	35
O1I		Intermittent	31	4	35
O2P	After First Occurrence	Permanent	31	4	35
O2T		Transient	31	4	35
O2I		Intermittent	31	4	35
	Total N° of Experiments		186	24	210

5.3 Experimental Results Analysis

From a total of 210 injections, 24 are performed on stratum S2 and 186 on S1. On stratum S2, 20 injections are type I (which cause no violation on Ozone's behaviour nor on stored data), and 4 are type EXC OO7 (which are tolerated by the system, causing the execution as well as all the queries performed to terminate normally). On stratum S1, 180 injections are type I. Among the other injections one of them is type A (which did not terminate normally and impacted the system leading it to a failure); the other five injections are type N (which presented no abnormality in Ozone's interface but the queries could not be performed, pointing that the stored data was corrupted, which in turn violated the ACID properties). To check stored data, we make a new connection with the database, invoke a query match (in which the root objects are checked) and a query traversal (which allows us to check the assembly hierarchy). Figure 2 presents the results for each stratum.

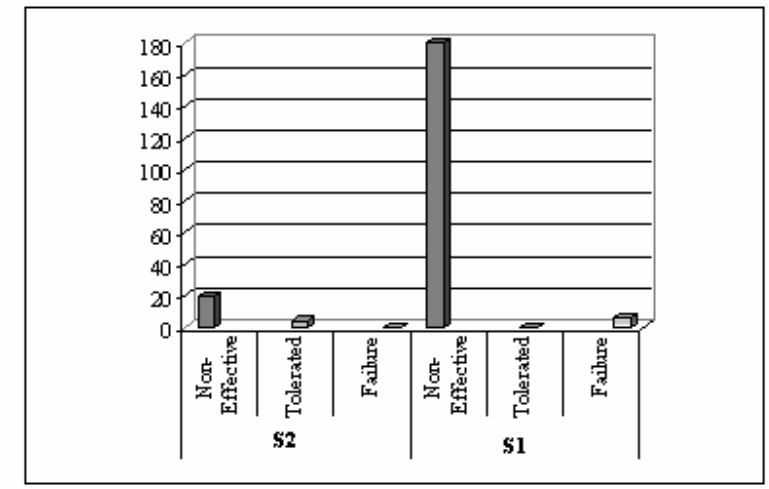


Fig. 2. Strata's Results

These results show that although the classes with higher WMC are more prone to errors than the ones with lower WMC, the impact of their faults in the system is not severe. This leads us to consider, in further experiments, other factors to define the strata.

6 Conclusions and Future Work

This work presents the use of stratified sampling for robustness testing purposes. The idea is to select components to inject in each stratum, instead of randomly selecting them from the whole set of system components. To define the strata, we use a complexity metric, WMC. The components are in fact divided into two strata: one for component with higher WMC value than the threshold value of this metric, and the other for lower WMC value than the same threshold value. We apply the approach for testing a database component, Ozone.

We perform experiments to evaluate the robustness of an off-the-shelf (OTS) component. Using a fault injection technique, we inject errors at chosen interfaces into Ozone.

The results show that Ozone's behaviour was different for each stratum, as expected. However, differently from our expectations, the stratum containing the classes with higher WMC does not produce the most severe failures; they do not cause database corruption.

In a previous work [10], the classes were selected according to several objected-oriented metrics selecting the classes with higher risk. In that work, the risk depended on various factors; among them the WMC metric.
This work is based on a single metric, the WMC, to select the Ozone classes in which to inject. The results show that the exclusive use of this metric is not sufficient to choose the strata. Other factors should be taken into account. For example, in the aforementioned work [11], we analyse the dependences among Ozone and OO7 classes. The results obtained are more promising results, highlighting that the dependence is more important in [10] than the WMC metric. Furthermore, the methods that implement a critical function in the system must be considered as a selection criterion, as shown in [10].
Further experiments are envisaged to define other criteria for stratification. As a long term goal, we intend to use stratified sampling to obtain inferences about a system's reliability.

Acknowledgment

This research is partly supported by CNPq – Brazil's National Council for Scientific and Technological Development – through the ACERTE project.

References

1. Bach, J.: Heuristic risk-based testing. Software Testing and Quality Engineering Magazine, (1999)
2. Beydeda, S., Volker, G.: State of the art in testing components. In: Proc. Of the International Conference on Quality Software, (2003)
3. Carey, M. J., DeWitt, D. J., Naughton, J. F.: The OO7 Benchmark. http://www.columbia.edu/, (1994), recovered February (2005)
4. Chidamber, K.: Principal Components of Orthogonal Object-Oriented Metrics. http://satc.gsfc.nasa.gov, (1994), recovered November (2004)
5. De Millo, R. A., Li, T., Mathur, A. P.: Architecture of TAMER: A Tool for dependability analysis of distributed fault-tolerant systems. Purdue University, (1994)
6. Fabre, J-C, Rodriguez, M., Arlat, J., Sizum, J-M.: Building dependable COTS microkernel-based systems using MAFALDA. In: Proc. of 2000 Pacific Rim International Symposium on Dependable Computing - PRDC'00, Los Angeles, USA, (2000)
7. Hsueh, M. C., Tsai, T., Iyer, R.: Fault Injection Techniques and Tools. In: *IEEE Computer*, (1997), pp. 75-82
8. 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, http://www.ece.cmu.edu/~koopman/ballista/ (2003)
9. Martins, E., Rubira, C. M. F., Leme N.G.M.: Jaca: A reflective fault injection tool based on patterns. In: Proc. of the 2002 Intern Conference on Dependable Systems & Networks, Washington D.C. USA, Vol. 23(267), (2002), pp. 483-487

10. Moraes, R., Martins, E.: A Strategy for Validating an ODBMS Component Using a High-Level Software Fault Injection Tool. In: Proc. of the First Latin-American Symposium, LADC 2003, pages 56-68, São Paulo, Brazil, (2003)
11. Moraes, R., Martins, E., Mendes, N.: Fault Injection Approach based on Dependence Analysis. In: Proc. of the First International Workshop on Testing and Quality Assurance for Component-Based Systems – TQACBS, (2005)
12. Ozone, Object Oriented Database Management System, www.ozone-db.org/, (2004)
13. Podgurski, A., Yang, C.: Partition Testing, Stratified Sampling and Cluster Analysis. In: Proc.of the 1st ACM SIGSOFT symposium on Foundations of software engineering. pp. 169-181, Los Angeles, USA, (1993)
14. Pressman, R. S.: Software Engineering a Practitioner Approach, 4th edition. Mc Graw Hill1, (1997)
15. Rosenberg, L., Stapko, R., Gallo, A.: Risk-based Object Oriented Testing. In: Proc. 13th International Software / Internet Quality Week (QW2000), San Francisco, California USA, (2000)
16. Transaction Processing Performance Council "TPC-C – Benchmarks". http://www.tpc.org/tpcc/default.asp, (2005)
17. Triola, M. F.: Introcução a Estatística, 7th Edition. LTC Editor, Rio de Janeiro, (1999) (in portuguese)
18. Voas, J., McGraw, G.: Software Fault Injection: Inoculating Programs against Errors. John Wiley & Sons, New York, EUA, (1998)
19. Voas, J. M., Charron, F., McGraw, G., Miller, K., Friedman, M.: Predicting how Badly Good Software can Behave.In: *IEEE Software*, (1997), pp. 73–83
20. Voas, J.: Marrying Software Fault Injection Technology Results with Software Reliability Growth Models. Fast Abstract ISSRE 2003, Chillarege Press, (2003)
21. Voas,J.: An Approach to Certifying Off-the-Shelf Software Components. In: IEEE Computer, 31(6), (1998), pp. 53-59

Capítulo 8 – Estratégia para Validação com base na Arquitetura

O artigo reproduzido neste capítulo foi publicado no *Workshop on Architecting Dependable Systems* que aconteceu em conjunto com *The International Conference on Dependable Systems and Networks* - DSN04.

Neste artigo utilizamos um sistema baseado na integração de componentes como técnica de construção do sistema de software. Dessa forma, considera-se um conjunto de componentes integrados sendo que alguns deles podem ser adquiridos de terceiros (*Common-Off-The-Shelf* – COTS), sendo nesse caso, considerados "caixa-preta". Em conseqüência, a estratégia para a escolha dos pontos de injeção baseada em risco não pode se fundamentar em métricas de complexidade dependentes do código fonte e a unidade considerada para os experimentos é mais um componente.

A estratégia utilizada neste trabalho é baseada em heurísticas, na qual um conjunto de critérios foi considerado para se avaliar o risco de um componente para o sistema no qual este componente se encontra integrado. Estes critérios foram combinados numa matriz e, de acordo com o número de critérios que é conferido ao componente, este componente é classificado como sendo de alto, médio ou baixo risco. Dessa forma, escolhemos os componentes que apresentem alto risco como pontos de injeção.

Concluiu-se que a estratégia utilizada ajuda a selecionar os pontos de injeção e monitoração com base em risco, fazendo um bom uso da arquitetura do sistema, porém critérios baseados em heurísticas trazem uma grande subjetividade à avaliação e um critério mais determinístico seria desejável. Um resumo do artigo já foi apresentado na Subseção 3.3.1 desta tese.

A referência deste artigo é a que segue: ***Moraes, R**., Martins, E. " Architecture-based Strategy for Interface Fault Injection". In: Proceedings of the International conference on Dependable Systems and Networks – DSN 04, Florence, Italy,2004*.

Architecture-based Strategy for Interface Fault Injection

Regina Lúcia de Oliveira Moraes
State University of Campinas (UNICAMP)
Superior Center of Technology Education (CESET)
regina@ceset.unicamp.br

Eliane Martins
State University of Campinas (UNICAMP)
Institute of Computing (IC)
eliane@ic.unicamp.br

Abstract

This paper presents a strategy that is an adaptation and an evolution of a previous one proposed to validate an isolated component. Faults are injected using a previously developed tool, Jaca that has the ability to inject faults into Java objects' attributes and methods. One of the key issues in component-based systems is its architecture, not only for development but also for testing. By analyzing the architecture we can define the points of control and observation of the system's components during testing. Another important issue is the selection of components to be injected and monitored. A risk-based strategy is proposed, in order to prioritize the components for testing which represent higher risks for the system. In this way, test costs can be reduced without undermining the system's quality.

1. Introduction

1.1. Motivation

Increasingly systems are being developed as a composition of several components; they can be developed in house (by the same team or by another) or by third-party. The use of components helps to attend the increasing pressures on reduced time and money, but it can introduce new problems, such as architectural mismatch [8] that can arise when the expectation of a component do not match those of other components or the environment in which they are operate. This increases the importance of the systems' architecture: the system's quality is increased when the used architecture is a good solution for the system. However the success of the implementation of architecture depends on two aspects [19]: i) that each component's implementation behaves in accordance to its specification, and ii) that components interact adequately.

This paper addresses the second point mentioned above. Specifically, our interest is to determine whether a malfunctioning in the interaction among components can compromise the overall system quality. Fault injection is used for that purpose.

1.2. Fault Injection

Fault injection has been widely used to evaluate the dependability of a system and to validate error-handling mechanisms. This technique consists of introducing faults during an execution of the system under test and then observing its behavior. By doing so, it is possible to know how the system will behave in the presence of faults in its components or in its environment.

Fault injection approaches vary according to the system's life cycle where they are applied, and the type of faults that are injected. Among the various existing approaches (see [9]), software fault injection is getting more popular. In this approach, logical faults are introduced in a prototype of the system and specific error conditions are injected to simulate software faults (internal, e.g., variable/ parameters that are not initialized among others), as well as faults that occur in external components that interact with the tested application (all external factors that alter the system's state)[20].

Fault injection can be a valuable approach in component-based development, not only to validate components in isolation but also to validate their integration into a system. By introducing faults or errors in different components of a system (in-house or third-party components), it is useful to answer questions such as: does a component fail when receiving invalid inputs from other components or from the environment? Does a failure in a component cause the whole system to fail?

1.3. Jaca - Fault Injection Tool

Our approach is based on the introduction of interface faults, in which faults are introduced at a component's interfaces by affecting input or output parameters as well as returned results. A software fault injection tool, Jaca [12], was used to introduce the faults, aiming at validating Java applications. Jaca does not require access to an application's source code, though it is a solution for the validation of a system composed of multiple COTS (Commercial Off The Shelf) components, which are generally black box. All instrumentation needed for fault injection and monitoring purposes are introduced at byte code level during load time. However, a minimum controllability (ease in controlling a component's inputs and outputs) and observability (the ability to observe a component's inputs, outputs and operational behavior) is required.

In order to inject faults using Jaca the controllability can be achieved when the tester knows the public methods' signature and may observes the tests' results and exceptions raised through Jaca's interface. Nevertheless, when the architecture is composed by third-party and in-house developed codes, specially parts that "glue" components together, the tester may use these

units to improve the system's controllability and observability by taking them as control and observation points during the process of fault injection.

1.4. Objectives of This Work

The goal of this work is to propose a fault injection strategy to test component-based systems. Our interest is in the interaction among components. For that reason, we introduce interface faults, by corrupting input data as well as interface output data.

Interface fault injection is useful in that it is at the interfaces that corrupt data come into a component. Components' internal faults can reach their interfaces, the errors' propagation occurs through the interfaces, and in the validation of component-based software this technique can be the only way to inject faults.

In Section II we discuss related works. Section III presents our proposed strategy. In section IV we point out some difficulties of the strategy and Section V presents the contributions of this work. Finally, in section VI we outline future research.

2. Related Works

This work is an evolution of that presented in [13] where we tested an isolated component. In that work we used an application to activate the component under test. The differences between them are twofold: (i) the target is no longer a component in isolation, but a system integrating various heterogeneous components, where some of them may be black-box; (ii) the units considered for fault injection purposes are components rather than classes. Thus, the strategy cannot use source code dependent metrics as in [13] and the architecture becomes essential for planning fault injection.

A closely related approach to the one presented here is the Interface Propagation Analysis (IPA) [24, ch.9.2]. IPA takes a black-box view of software components, injecting faults at the interfaces between the hardware and the software as well as between the operating system, microkernel and so on. The difference is that we are considering that not all components are black box.

The steps described in our work were strongly influenced by those described in [4], where the generation of error sets was based on field data. The difference in our case is that we considered field data to be unavailable.

TAMER [6] is another work that describes a tool that injects interface faults aiming to observe fault propagation. The main focus of that work is code coverage. We are not interested in source code coverage, but rather in the exceptions raised by the component, and whether these exceptions cause the whole system to fail.

The work in [7] is quite similar to ours since they use a tool based on computational reflection, called Java Wrapper Generator (JWG), which modifies the bytecode at load time, in order to provoke an exception and observe the behavior of the exception handlers. In their case the focus is on objects, whereas in our approach the focus is on components, which may be composed by several classes.

From the Ballista [10] approach we utilized the definition of the error model that was proposed by the authors for robustness testing.

We also borrowed ideas from studies that use risk for test costs reduction. Many risk-based testing strategies have been proposed [2], [17]. The approach presented here is specially related to [2], from which we use the heuristic risk-based testing presented below.

3. The Approach

Fault injection experiments can be characterized by the FARM model [1], where F designates the set of faults/errors to be injected, A the activation mode of the system, R is the set of data collected during the experiments and M the verdict of whether or not the system behaves as specified, when the fault injection goal is to reveal design and/or implementation faults.

The approach used to validate a component-based system using fault injection is quite similar to that proposed in [20, ch.9.2], where faults are injected at the interfaces between components to simulate the situation where a component fails and its outputs corrupted information to the others components System robustness is determined by checking post-conditions at specific points as well as at the system-level interface.

The remainder of this section presents an approach that helps to determine the F set by answering: which components should be injected? How should they be injected? What fault model should be used? and When should they be injected?

3.1. Architectural View

The software architecture of a system represents the software structures that form the skeleton of the application. It defines how the system is structured in terms of the components that form the system, assigns the responsibilities to the components, defines the interactions among them and assures that the component's interactions satisfy the system requirements [5][18]. An important issue is the definition of an interconnection mechanism for gluing the pieces together, the connectors [8][19]. Thus a system architecture is defined as a collection of components, which is a unit of software that performs a function at run-time and a collection of connectors, which is a unit of software that "glues" components together and mediates the interactions (communication, coordination, or cooperation) among these components [8]. Through the connectors' format conversions, two incompatible components can share data as well as connectors augmented by performance and behavior monitoring, authentication and audit-trail capabilities [18]. Although

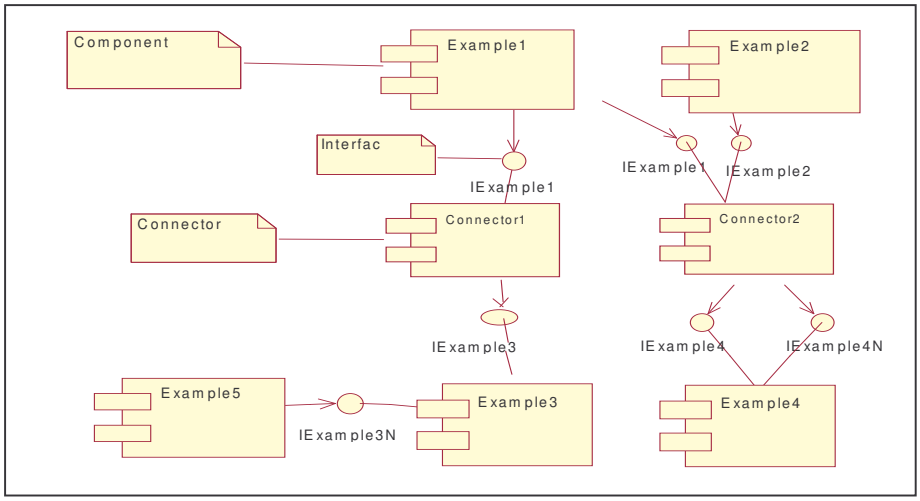


Figure 3.1: A generic view of a component-based system

the system may be composed of COTS components, from which no source code is available, the system is not considered a black-box: the system's architecture, which represents how the various components are interconnected, is known. Figure 3.1 shows, using UML notation and the system's architecture as viewed for fault injection purposes.

Components can provide one or more interfaces, through which they specify their services. They can also require interfaces that describe the services they need in order to provide theirs own interfaces [5][14]. In Figure 3.1 the Example4 component offers its services through the interface designated as IExample4 and IExample4N. The Example5 component requires services specified by interface IExample3N.

Connectors can also provide services, such as persistence, invocation and transactions, which are independent of the interacting component's functionality [19] and are responsible for the association between a required interface and one or more provided interfaces. This association is called interface connection [8] [19].

As shown in Figure 3.1, connectors link the interface required by a component to an interface provided by another component (shown as "lollipops in the UML notation for interfaces), but these interfaces can interact directly with each other without the mediation of a connector (interaction between Example3 and Example5).

3.2. Generation of the F Set

Inspired by the work of [4] we defined the following steps to determine the F set:

1. Prioritize the components
2. Select the components that should be injected
3. Select the operations of each selected component
4. Generate a list of injection points within each operation
5. Define error model
6. Decide temporal characterization of the faults

Steps 1 and 2 are not injection tool dependent, all other steps are. To answer the question "Which components to inject?" we defined steps 1 and 2. The question "How to inject?" can be answered in steps 3 and 4. "What fault model to use?" can be answered in step 5 and finally "When to inject?" in step 6.

3.2.1. Components prioritization

Given a system of multiple components, what components should be selected, when it is not possible to inject all of them? An approach could be to select the components based on information about the distribution of faults over the component observed either during development or in the field [4]. If such information is not available, a risk-based testing strategy can be applied. The idea is to allocate test effort in code parts that are most error prone, and where failure would have the highest impact [17].

There are several techniques to assess risk (see [11], [17]). We use a heuristic method, as proposed in [2], where the following criteria are used to assess a components' risk: New (freshly developed), Changed (has been modified), Upstream Dependency (failure in it will cause cascading failure in the rest of the system) Downstream Dependency (it is especially sensitive to failures in the rest of the system), Critical (failure in it could cause substantial damage), Popular (will be used a lot), Strategic (has special importance to business-feature that set apart from the competition), Third-party (developed outside the project), Distributed (spread out in time or space, yet whose elements must work together). To these criteria we add one more: Understandable (how much information about the component is provided and how it is presented).

With these criteria, a component risk matrix can be constructed, as shown in Table 3.1. If a check is placed

Table 3.1: Summary table of component risks

Component	New	Changed	Upstream Dep	Downstream Dep	Critical	Popular	Strategic	Thid-Party	Distributed	Not Understandable	Risk
Component1		✓			✓	✓	✓	✓		✓	High
Component2			✓				✓				Low
Component3					✓			✓		✓	Medium
Component4	✓			✓		✓	✓	✓	✓	✓	High

in a column, it means that this criterion is significant for that component. Although we can count the number of checks placed for a component, the risk judgments cannot be considered with this simplicity. It is possible to have a situation in which a component has less heuristics checked than another but it is considered more risky than the latter. These heuristics have to be seen as tools for assessing risk not for determining it. They do not substitute an expert's opinions [14].

3.2.2. Selection of the Components

Given the high risk components obtained in step 1, we continue to select those components that can be injected using Jaca. Two more criteria are used:

- *Controllability*: which shows how easy it is to inject faults on a component's inputs/outputs.
- *Observability*: which regards the ease with which a component can be monitored in terms of its operational behaviors, input parameters, and outputs.

When Jaca is used, the injectors and sensors needed for injection and monitoring purposes during runtime are inserted at bytecode level when the system is being loaded. This is achieved through the use of the Javassist toolkit[3], which extends the Java Reflective Interface.

If a selected component has low controllability and observability, the user has the following options: (i) to inject faults into the component's predecessors and successors developed in-house which have the desired controllability and observability (ii) to inject faults into the user-developed connectors (connectors can be taken as injection points or as observer points, not both, as the results could be influenced by the faults injected).

3.2.3. Selection of the Operations

The components' operations are distributed among the components' interfaces. Once a component has been chosen, faults are injected in the component's interfaces indistinctly, considering all of their operations as a whole. To select the operations in which parameters will be affected during fault injection, we can use a technique similar to partition testing, as presented in [16, ch.22.5], where each operation can be categorized, for example, as Initialization, Computational, Queries and Termination operations. We can also use a state-based partitioning that categorizes operations according to their ability to change the state of the component. Faults are selected so that they affect operations in each category at least once. Faults are distributed uniformly among the categories.

3.2.4. Generation of the List of Injection Points

Once the operations that are to be injected are selected, the next steps are to determine the parameters to inject, and what error model to apply.

In what concerns the parameter selection, we have to cope with Jaca's current limitations: only non-structured values can be affected (integer, real, and Boolean). The only structured type that can be affected is string [13].

3.2.5. Selection of the Error Model

The error model is based on Ballista's [10], alongside the one proposed by [20]. According to these two approaches the values to be used for each type are:

- Integer / long: 0, 1, -1, MinInt, MaxInt, neighbor value (current value ± 1)
- Real floating point: 0, 1, -1, DBLMin, DBLMax, neighbor value (current value*0.95 or *1.05)
- Boolean: inversion of state (true -> false; false -> true)
- String: null, largest string, string with all ASCII, string with pernicious file modes and printf format which can be composed by conflicting characters.

These values potentially represent exceptional test values for each data type.

3.2.6. Temporal Characterization

The temporal characterization of faults is strongly related to the mechanisms used to trigger fault injection. In Jaca, faults are triggered when the target operation is accessed.

Faults can be injected either permanently (each time a target operation is accessed), transiently (where faults are injected only once), or intermittently (injected repeatedly according to a pre-specified frequency, established in terms of the number of accesses to an operation).

4. Difficulties of the strategy

We identified some difficulties, such as:

(i) How many criteria must be satisfied?
(ii) How should these criteria be weighted?
(iii) How should each factor be quantified?

This should be based on experts' experiences as suggested by [14]. Experts can comprise developers, key users, customers and test personnel, who should brainstorm the risks associated with the specific software system [15]. Some factors are more difficult to quantify

than others, such as, popular, critical, upstream /downstream dependence, understandable.

(iv) How should the risk of a component be determined?

Risks are normally a quantitative value. To obtain this final value, it is necessary to rank risks for each factor, i.e. in the range 1-3 rank low to high (relational severity indicator, one-third for each degree). The severity associated with each factor is also application-domain-dependent and must be established by experts. It can be determined subjectively, as in [15]. It can also be determined according to safety analysis classification, as in [11]. . The risk of the component should be the sum of all partition's risk[20].

(v) How should successors and predecessors be determined? Following the determination of upstream/downstream dependency?

Based on [20], predecessors are components upon which the target component depends for input information and conversely, successors are those components to which the target component sends information. Thus, the predecessor can send corrupted data for the component under test and the latter can send corrupted data to its successors. Static analyses can help in determining successors and predecessors of a component[15].

(vi) What error model should be selected?

We combine the partition testing model with the error model based on Ballista's robustness testing. However, the later considers only the input space, which, as determined by boundary-value analysis, is not enough. The output space should also be considered. Which inputs may cause the component or the system to produce the wrong output? Using Fault Tree Analysis we can map the wrong output to the potential inputs.

(vii) How can good *controllability* and *observability* of the system's components be achieved?

Jaca is designed for interface fault injection, so, only externally visible operations are considered. Nonetheless, there are still some limitations to overcome, such as injection and monitoring of non-scalar types; insertion of post-condition and invariant checkers to indicate failure occurrence; and tracking of exceptions and activation of exception handlers. We are proposing the use of connectors to achieve the desired controllability and observability.

5. Contributions of Our Work

This work focus on the systems' architecture, in particular the connectors, for improving the controllability and observability that is necessary for fault injection tests.

What is being proposed is a systematic way to perform fault injection to characterize the behavior of components and systems in the presence of faults. This also contributes to dependability benchmark, in that it provides a uniform and repeatable way to perform fault injection.

6. Future Research

Our short-term goals are (i) define how safety techniques (methods based on FMEA, FMECA, FTA) can help to better define the selection of the injections points; (ii) the application of the strategy in a real world application aiming to assess the value of using the architecture to guide fault injection (we will perform experiments comparing the results obtained when fault injection points are randomly chosen among the components).

References:

[1] 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, 16 (2), Feb/1990, pag 166-182.

[2] Bach J. "Heuristic Risk-Based Testing", Software Testing and Quality Engineering Magazine, November 1999

[3] Chiba, Shigeru. "Javassist – A Reflection-based Programming Wizard for Java", proceedings of the ACM OOPSLA'98 Workshop on Reflective Programming in C++ and Java, Oct/1998.

[4] Christmansson, J ; Chillarege, R. "Generation of an Error Set that Emulates Software Faults-Based on Fields Data", 26th Int Syposium on Fault-Tolerant Computing, pp 304-13, Sendai, Japan Jun/1996.

[5] Cheesman, J; Daniels, J "UML Components – A Simple Process for Specifying Component-Based Software", The Component Software Series, Addison-Wesley, 2001.

[6] 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.

[7] Fetzer, C.; Högstedt, K.; Felber, P. "Automatic Detection and Masking of Non-Atomic Exception Handling", proceedings of DSN 2003, pages 445/454, San Francisco, USA, June/2003.

[8] Garlan, D. ; Allen, R.; Ockerbloom, J. "Ärchitecture Mismatche, or, why it's hard to build systems out of existing parts", proc. of the 17th ICSE, April/95.

[9] Hsueh, M.C; Tsai, T.; Iyer, R.. "Fault Injection Techniques and Tools". IEEE Computer, Abril/1997.

[10] 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, http://www.ece.cmu.edu/~koopman/ballista/, 2003.

[11] Levenson, N.G. "Safeware, System Safety and Computers" Addison-Wesley Publishing Company, 1995.

[12] Martins, E.; Rubira, C. M. F.,Leme, N. G. M., "Jaca: A reflective fault injection tool based on patterns", proceeding of the IPDS, 2002.

[13] Moraes, R; Martins, E "A Strategy for Validating an ODBMS Component Using a High-Level Software Fault Injection Tool", proc. of the First Latin-American Symposium, pages 56-68, SP, Brazil, 2003.

[14] Peters, J. F.; Pedrycz, W. "An Engineering Approach", John Wiley & Sons Inc, 2000.

[15] Perry, W. "Effective Methods for Software Testing", John Wiley & Sons, New York, 1995.

[16] Pressman, R. S. "Software Engineering a Practitioner Approach", 4th edition, Mc Graw Hill , 1997.

[17] Rosenberg, L; Stapko, R; Gallo, A "Risk-based Object Oriented Testing ", 13th International Software / Internet Quality Week (QW2000), San Francisco, California USA, 2000.

[18] Shaw, M.; Clements, P. "Toward Boxology: Preliminary Classification of Architectural Styles", proceedings of SIGSOFT 96 Workshop, San Francisco, CA, USA, 1996.

[19] Silva, M. C.Jr.; Guerra, P. A. C.; Rubira, C. M.F. "A Java Component Model for Evolving Software Systems", proc. of Automated Software, Engineering, Canada, 2003.

[20] Voas, J.; McGraw, G.. "Software Fault Injection: Inoculating Programs against Errors", John Wiley & Sons, NY, EUA, 1998.

Capítulo 9 – Injeção de Falhas com base em Dependências Arquiteturais

O artigo reproduzido neste capítulo foi publicado como um capítulo de livro na série *Architecting Dependable Systems* III. Neste artigo, a técnica *Chaining* (refere-se a Seção 2.2) foi utilizada para se determinar as classes de maior risco com base no número de elementos que participam dos conjuntos *affected-by chain* e *affects chain*. O número de classes nesses dois conjuntos é considerado como um critério para definir as classes que apresentam maior risco para a aplicação. Quanto maior o número de elementos existente nestes conjuntos, maior é a chance de um erro se propagar através das interfaces. Além de se definir os componentes de maior risco para direcionar a injeção de falhas, a dependência arquitetural foi utilizada para se definir os pontos de monitoração dos experimentos.

A principal contribuição do trabalho é a metodologia que foi apresentada utilizando a análise da arquitetura de software e as dependências arquiteturais para guiar os experimentos que utilizam injeção de falhas. Esta abordagem é independente de se ter disponível o código fonte. Um resumo do artigo já foi apresentado na Subseção 3.3.2 desta tese.

A referência deste artigo é a que segue: ***Moraes, R.***, *Martins, E.* "*Fault Injection Approach based on Architectural Dependencies*". *Architecting Dependable Systems III, Book Chapter, Lecture Notes in Computer Science, Springer-Verlag Berlin Heidelberg New York, pp. 300-321, 2005.*

Fault Injection Approach based on Architectural Dependencies

Regina Lúcia de Oliveira Moraes[1], Eliane Martins[2]

[1] State University of Campinas (UNICAMP),
Superior Centre of Technological Education (CESET), regina@ceset.unicamp.br ,
http://www.ceset.unicamp.br/~regina
[2] State University of Campinas (UNICAMP),
Institute of Computing (IC), eliane@ic.unicamp.br,
http://www.ic.unicamp.br/~eliane

Abstract. In a previous paper we described a fault injection strategy that applies risk-based analysis to select the system's riskiest components for testing. Among other criteria, this analysis considers the number of upstream and downstream dependencies of a component in a system. In order to obtain this number, we propose the use of architectural-level dependency analysis. One advantage of an analysis at architectural level is that systems may often contain COTS components from which no source code is available. The approach is illustrated with a case study, and the preliminary experimental results are also discussed.

1 Introduction

The increased pressures on time and money make component-based software development a current trend in the construction of new systems. In this method, instead of bespoke design and development, a system integrates off the shelf (OTS) components developed by third parties.

Despite the potential benefits of component-based development, the validation of components and component-based systems is still a challenge. The difficulty stems from lack of knowledge [5] [34]. On the one hand, component users do not know the acquired component's quality level, and even if it is known, there is no guarantee that the component will present the same quality level when used in a new context. This means that the acquired component must be validated each time it is used in a new context. However, users generally do not have enough information about the OTS component to perform this task.

On the other hand, using high-quality components is no guarantee that the overall system will present high quality. The complexity of the interaction among components can cause unexpected errors to emerge from component interfaces [35]. According to [35], 50% of bugs are detected after component integration, not during component development.

There is an increasing demand nowadays for high quality, critical and non-critical applications. For example, both e-commerce and military systems require availability and security. In order to achieve these quality properties, a focus on solutions at architectural level can be foreseen. Research into describing software architectures with respect to their dependability properties has recently gained considerable attention [31] [28] [33].

A good architectural solution is an important step, but it is not enough to guarantee that the final system will present the required quality level. Systems are increasingly complex, integrating thousands or millions of (hardware and/or software) components. There are numerous interfaces among these components, which increase design complexity. Components' interfaces comprise the assumptions that components make about each other. Architectural mismatch [13] can arise when the expectations of a component do not match those of other components or the environment in which it operates. Furthermore, the coupling between components to achieve the system's goals makes them highly interdependent; consequently, a failure in one component can rapidly affect the state of other components [35].

Validation is thus a necessary step to establish whether an architectural solution achieves the required system qualities. Moreover, it is important to assess the robustness of the interfaces with respect to component failures as well as problems that enter the system from external sources [34].

We propose the use of Software-Implemented Fault Injection (SWIFI) to observe how interfaces behave when data passing through them are intentionally corrupted. SWIFI is a useful complement to other validation techniques, in that it allows us to observe whether the failures of components or interfaces among components affect the services provided by the system.

Our approach is based on the introduction of interface errors, that is, errors are introduced at a component's interface by affecting input or output parameters, as well as returned results. A component's internal faults can generate errors that may propagate to its interfaces. Thus, interface errors may represent a component's failure modes as well as the failure modes of other components that interact with the target component. Errors are introduced using a software-implemented fault injection tool, Jaca [20], in order to validate Java applications. Jaca does not require access to an application's source code, since it is a solution for the validation of a system that may be composed of multiple, generally black-box OTS. All instrumentation needed for fault injection and monitoring purposes are introduced at byte code level.

Given that a component-based system may contain too many components and interfaces, it may not be practical to inject errors in all of them. Although the system may be composed of OTS components, from which no source code is available, the system is not considered a black-box; the system architecture, representing how various components are interconnected is known. We will use this knowledge to help select a subset of components and interfaces in which to inject errors. One approach that has been used in tests is to perform the selection based on risk analysis: more test effort is concentrated on those parts of the system that may present higher risk of causing the system to fail [4] [27]. Various factors can be considered when evaluating a component's risk. Complexity metrics can be one of them [27]; in a previous work we used a set of OO metrics to select

a component to inject [22]. One limitation of this approach was the need of source code to better categorize such metrics.

In the current approach we use dependency analysis, at the architectural level, to guide fault injection. In brief, the idea is to select components based on upstream dependencies (components whose failure will cause cascading failures in the rest of the system) as well as downstream dependencies (components particularly affected by failures in the rest of the system) [4].

In the following section we give a fault injection overview. In Section 3 we describe the dependency analysis approach that we use in this work. The proposed strategy is discussed in Section 4; a case study is presented in Section 5 and the tests' results are presented in Section 6. Finally, in Section 7 we present our conclusions and future research.

2 Fault Injection Overview

2.1 Fault Injection

Fault injection techniques have been widely used to evaluate a system's dependability and to validate its error-handling mechanisms. The technique provides a means to dynamically demonstrate the software quality and to observe the system's behaviour [36]. By doing so, it is possible to know how the system will behave in the presence of faults in its components or in its environment. Fault injection enables accelerated system testing under stressful conditions, and can help uncover design and implementation faults in the systems [3]. This technique is useful to validate the solutions designed to handle exceptional situations.

Fault Injection may be used to validate a fault tolerant system, to help with fault removal aimed at reducing the occurrence of faults and their severity, as well as to assist with faults forecasting. Fault removal and fault forecasting could be used to quantify dependability attributes. In this work we are interested in its fault removal aspects.

Fault Injection approaches may vary according to the system life cycle in which they are applied and to the type of faults that are injected. Among the various existent approaches (see [15] for an overview), software-implemented fault injection has been widely used [9] [12] [26]. It has become more popular due to its lower costs (it does not require specially developed circuits, as does hardware fault injection), better versatility (it is easier to adapt codes to make fault injection in another system than to adapt of circuits) and better control, which together facilitate the observation of the system during tests. One approach of software-implemented fault injection consists of injecting anomalous input data that come into the software through its interface [36], instead of altering a system's code or state in order to emulate the software [34]. This study uses this approach, allowing software acquirers to determine its robustness. The software can be stated as robust if it is fed by anomalous input and does not propagate into a failure, demonstrating that the software can produce dependable service even in presence of an aggressive external environment [36].

For software-implemented fault injection the most common faults considered are memory faults (which alter the content of memory positions), processor faults (which affect the content of registers, the result of calculus, control flux or instruction), bus faults (which affect the addressing lines or data that are being transmitted by the bus) and, in the case of distributed systems, communication faults (which affect messages that are transmitted through a communication channel: they can be lost, altered, duplicated or delayed)[15].

Recently more attention has been given to the consequences of software faults. Software faults represent the faults resulting from mistakes committed by developers during system development or in modifications made in the phases of maintenance. Software faults have turned out to be the main causes of field failures.

In this work we are using a software-implemented fault injection tool, called Jaca, to inject errors and test the system's robustness. A similar approach was presented by Ballista Project [17] and Mafalda tool [11], but in those cases the errors were injected in the parameters of operating system calls instead of the component interfaces.
TAMMER [9] is another similar work in which the injection of interface faults is used to observe fault propagation focusing on code coverage. The tool Jaca, used in this work, is presented in section 2.2.

2.2 The Jaca Tool

Software Fault Injection can affect either the code (source or assembler) or the state of a target system. To alter a system's state, a tool is needed to inject faults or errors during runtime. These tools differ according to the mechanism used to trigger faults [15]. Most of these tools are aimed at emulating hardware faults, so faults are injected at low-level, affecting processor registers, I/O device drivers and memory positions. Nowadays, with the increasing importance of software faults, some studies present tools that aim to inject faults at higher level during runtime [9] [12] [26].

Jaca offers mechanisms for the injection of interface errors in object-oriented systems written in Java language. Jaca is an evolution of the FIRE tool [26] and it uses reflective programming. The reflection mechanism introduces a new architectural model by definition of two levels: the meta-level (implements fault injection and monitoring features) and the base level (implements the system's functionalities) [19]. Computational reflection allows the target system's instrumentation to carry out its functions through introspection (useful for the system's monitoring) or by altering the system during runtime (useful for the injection) without changing the system's structure. Jaca does not need the application source code to perform fault injection. This occurs because Jaca was implemented using the Javassist reflection toolkit [6], which allows the instrumentation to be introduced at byte code level during load time. The source code independency is an important feature of the tool, allowing the validation of a system that may be composed of multiple third-party components. Jaca's current version can affect the public interface of an application by altering attributes' values, method's parameters and return values. The tool needs to get a class's interface information in order to inject the errors, and it can do

this through introspection when the source code is not available. Jaca is described in more detail in [18], [20].

2.3 Related Work

The use of fault injection to validate component-based systems is an active research area. This work is an evolution of that presented in [21] in which we test an isolated component. In that work we used an application to activate the component under test; it differs from the current work in two respects: (i) the target is no longer a component in isolation but a system integrating various heterogeneous components, some of which may be black-box; (ii) the units considered are no longer classes, but components. Hence, the strategy cannot use source code dependent metrics as in [21] and the architecture becomes essential for planning fault injection. This work also extends that presented in [23], where we introduced the idea of architectural relevance for testing a component-based system without describing our criteria in detail. Our current work tackles the dependencies criterion.

A closely related approach to the one presented here is the Interface Propagation Analysis (IPA) [33, ch.9.2]. IPA takes a black-box view of software components, injecting faults at the interfaces between hardware and software, as well as between operating system, microkernel and so on. The difference is that we are not considering all components as black-box.

TAMER [9] is another study which describes a tool that injects interface faults aimed at observing fault propagation. The main focus of that work is code coverage. Here we are not interested in source code coverage but in the exceptions raised by the component, as well as whether these exceptions cause the whole system to fail.

The work in [12] is quite similar to ours since they use a tool based on computational reflection, called Java Wrapper Generator (JWG). JWG modifies the bytecode at load time, which in turn provokes an exception and allows the observation of the exception-handlers behaviour. In Fetzer's case, the focus is on objects, whereas in our approach the focus is on a component that may be composed by several classes.

MAFALDA (Microkernel Assessment by Fault injection Analysis and Design Aid) [11] is another tool that provides quantitative information on COTS microkernels to support their integration into dependable systems using error confinement wrappers. The proposal of MAFALDA is the injection of errors in the parameters of operating system calls, instead of the parameters between components as in our case.

The dependency analysis at the architectural level used in our work is strongly influenced by the study in [32] where they propose the chaining concept to reduce the portions of an architecture that must be examined in order to test or debug a system. In our work we use this idea to select the components to inject and monitor the fault injection.

From the Ballista approach [17] we derive the definition of error model, which is proposed by the authors as a means to test robustness.

We also borrow ideas from studies that use risk for test costs reduction. Many risk-based testing strategies have been proposed [4] [27] [30]. The approach presented here is

particularly related to [4], from which we use the heuristic risk-based testing presented below.

3 Dependency Analysis at the Architectural Level

Abstractly, software architecture is a representation of the system based on the components that integrate the system and their interactions [29]. The set of components integrated in a system can be component interaction elements or connectors, data elements, processing (or behavioural) elements or state elements that contain the current state of the component both in terms of data and processing elements. Architecture diagram is a high-level model of software system that represents the system's components and how they are interconnected.

A component may be defined as "unit of composition with contractually specified interfaces and explicit dependencies. A software component can be deployed independently and is subject to composition by third parties" [8]. A component's provided interface allows one component to provide information or stimulus to another component, whereas the required interface allows one component to ask for information or receive stimulus from other components.

Dependency analysis has been traditionally based on control and data flow relationships associated with functions and variables of a program [32]. This has worked primarily for compiler optimization. It has been used widely in software engineering activities such as program understanding, testing, debugging, reverse engineering, and maintenance [7]. It has also been useful for code reviewers and architects when assessing the coupling within an application or library. A limitation of the traditional approaches is that they are generally source-code based. This is not useful to us for two main reasons: (i) the source code of some OTS components might not be available; (ii) polymorphism (the ability to bind a reference to more than one object) and dynamic binding (where a specific bond between a reference and an object is determined at runtime) replace explicit compile-time binding with implicit runtime binding. This means that a receiver of a polymorphic message is only known at runtime. In this way, the interaction among components is better determined through the analysis of the system's architecture.

Dependency relationships at the architectural level appear from the connection between components and the constraints on their interaction [32].

There are several architectural-based dependencies' approaches [16] [1] [24]. The approach used in this work is based on Chaining, dependency analysis technique that was primarily aimed at reducing the part of the architecture to be examined for a given purpose, for example testing and debugging [32]. Individual links in a chain associate architecture components that are directly related. A chain of dependencies includes association among components that are indirectly related.

There are three types of chains: (i) affected-by chains, which contain the set of components that could potentially affect the component of interest; (ii) affects chains, which contain the set of components on which the component of interest can potentially

have an effect; (iii) related chains, which are a combination of affected-by and affects chains.

The next section presents how the chaining technique can be useful for fault injection.

4 Using Dependency Analysis for Fault Injection

Dependency analysis can be useful for fault injection in many ways: (i) to help select target components, e.g. components whose interfaces are to be injected; (ii) to establish monitoring points to determine error propagation, and (iii) to select a minimal set of components to monitor for debugging purposes in case of system failure.

As discussed above, components with high affects chain and affected by dependencies are possible targets for fault injection.

Figure 1 illustrates the chains of a given component. In this figure, the component designated as "Target Component" has 5 downstream dependencies (number of elements in its affected-by chain) and 4 upstream dependencies (number of elements in its affects chain). Faults that are introduced in the links indicated in the figure can have an effect on other components that do not need to be directly injected. Faults in a link between the target component and an affects chain component represent failure modes in components used by the target one (its successors) and could be impacted by a change in the target component. Conversely, faults in a link with an affected-by component represent failure modes from components that use the target one (in other words, failures on its predecessors) and may affect it.

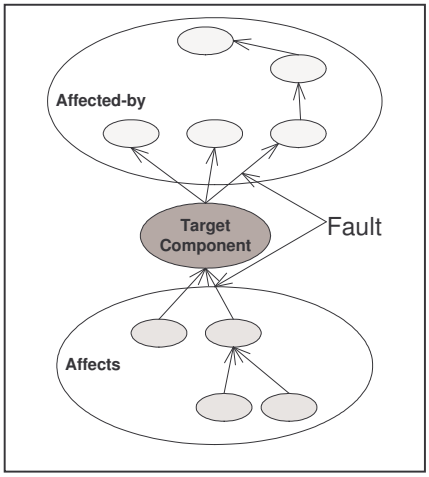


Figure 1: Chains

We also need to determine where to observe in order to understand the effect of the corruptions. When a selected component does not have the required observability (easiness of monitoring) the observation point should be transferred to another component in the same chain (affected-by or affects).

In this way, we can assess the failure tolerance of the interfaces regarding component failures and corruptions that may enter into the system from external sources.
We must bear in mind that in this context the related chains can comprise any components that integrate the system (protective wrappers, exception handles and so on).

Our approach encompasses the following steps:
1. Modelling the system's architecture
2. Constructing the dependency matrix
3. Determining the chain for each component
4. Determining the component to be injected
5. Determining the failure mode
6. Determining the values to be injected
7. Determining the expected outcomes

The following sections describe these steps in more detail.

4.1. Modelling the System's Architecture

The kinds of dependencies that can be considered among components are influenced by the notation used to represent the system's architecture [32]. In our approach we are interested only in behavioural relationship, particularly in input and output interactions among the components. Figure 2 presents an example of the architectural model considered in this study where the interactions among components through the provided and required interfaces are in relief.

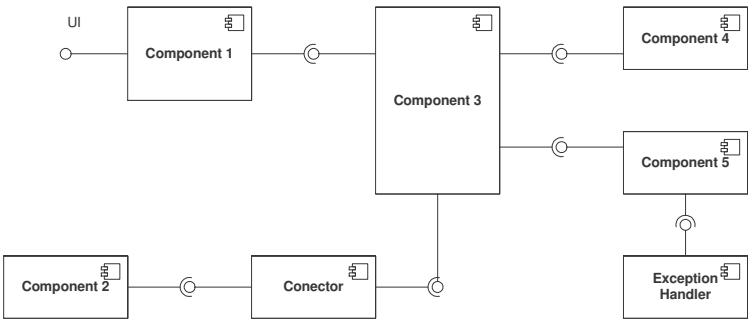


Figure 2: The System Architecture Model

4.2. Constructing the Dependency Matrix

We use a matrix to represent the components relationships as suggested in [32]. Although the chaining technique considers different types of connections, only the connections between a provided interface and a required interface are being considered in this study. In the dependency matrix, the columns represent the dependent in the relationship. The rows represent the object of dependency in addition to any events generated in the system's environment that can be acquired by an interface (e.g. a user interface - UI). Thus, if component A is dependent on B, the cell at column A and row B hold that relationship (indicated by an "x" in this cell). Table 1 shows the matrix corresponding to the architecture in Figure 2.

Table 1: Dependency Matrix

	Component1	Component2	Connector	Component3		Component5
Connector		X				
Component3	X		X			
Component4				X		
Component5					X	
Exception Handler						X
Component1	X					

4.3. Determining the Chain for Each Component

A component's chain can be determined by creating links when there is a mark (X) in a cell, which indicates that a relationship holds. To determine the affected-by chain for a component, select its corresponding column and locate cells containing an "X" in this column. The component in the corresponding row is in the affected-by chain of this component. To determine the affects chain for a component select its corresponding row and locate cells in this column containing an "X". The component in the corresponding column is in the affects chain of this component. Both affected-by and affects chain are transitive; once you create a link you construct in a similar manner the next link, beginning by the newly identified element in the chain.

For example, to determine the affected-by chain for Component2 in the matrix presented in Table 1, and to identify which component potentially affected it, it is necessary to take the column Component2 and locate all the related rows in the matrix. Only the Connector has been identified. Continuing by the transitivity property, one must take the column Connector and locate all the related rows in the matrix, reaching Component3. Taking the column Component3, Component4 and Component5 are identified and finally the column Component5 identifies the Exception Handler. In this way, the affect-by chain of Component2 is Connector, Component3, Component4, Component5 and Exception Handler.

The affects chain can be obtained in a similar way beginning with the row, for example, taking Component3. To identify which components it may affect, one obtains Component3 affects chain as Component1, Connector, Component2 and information acquired from the boundary of the system through the user interface (UI).

4.4 Determining the Candidate Targets

To select a component in which to inject errors we are considering the number of components in affects chain and affected by chain of each component. Based on Pareto's 80/20 rule [25], the components whose sum of both chains is classified between the top 20%, must be selected to inject the faults.

As mentioned above, these components may affect many others, so we can observe the impact of faults without having to inject all the components' interfaces.

Based on Table 1, where we have six rows, if we apply Pareto's 80/20 rule we should inject errors into one or two components. The highest sum of affected-by and affects chain belongs to Component3, which has this sum equalling seven, followed by Component5 and ExceptionHandler equalling six. If the tester must make a choice and inject error in only one component, Component3 should be selected; otherwise, he should select Component5 and Exception Handler, too.

However, there is a need to cope with constraints on controllability (easiness to inject faults) imposed by fault injection tools. Jaca, namely, can only inject and observe components written in Java. Another limitation is that it can only affect parameters or return values that are not objects. As Jaca, other fault injectors also have their limitations. If the selected component is not considered controllable, the related chain can be used to select another target component.

4.5. Determining Where to Inject

Once the components that should be tested are selected, each component should be considered (CUT) in order to determine in which of their interfaces to inject errors. As we are considering only the connections between a provided interface and a required interface, our focus is on parameters and returned values that flow between components. The locations to inject are:

(i) input parameters provided to the target component by its predecessors (according to links in affected-by chain);

(ii) a returned value (or output parameter) provided by the target component to its predecessors (according to links in affected-by chain);

(iii) input parameters provided by the target component to its successors (according to links in affects chain);

(iv) a returned value (or output parameter) provided to the target component by its successors (according to links in affects chain);

(v) a parameter or returned value that comes into the system through its boundary components.

4.6. Determining the Values to Inject

To determine which values we should inject into input/output parameters or returned values of the operations in a component's interface, one should consider the component's specification when there are valid domains of these values or constraints imposed by the

component's contract. In such case, boundary value testing can be used [25]. However, if these domains are not specified, the Ballista approach may be used [17]. In this case, the values to be used for each data type are presented in Table 2.

Table 2: Values to Inject based on Ballista's Approach

Data Type	Values to Inject
Integer	0, 1, -1, MinInt, MaxInt, neighbour value (current value ± 1)
Real Floating Point	0, 1, -1, DBLMin, DBLMax, neighbour value (current value * 0.95 or * 1.05)
Boolean	inversion of estate (true -> false; false ->true)
String	Null

4.7. Determining the Expected Outcomes

An output value space is the set of all possible output values of the program. In this set, as a result of an experiment, the system may fail or tolerate the injected faults. Tolerance to the injected faults means that the system outputs the expected results, i.e. that the architecture offers the required tolerance level. Otherwise, if failure occurs it means that modifications to the system are needed. Failures may be reported, raised exceptions or returned wrong values. Application hangs, application crashes or erroneous values as outputs characterize a non-reported failure.

In order to decide if an output value is a wrong value or not, an oracle mechanism is needed. An oracle is a predicate on input/output pairs that checks whether the desired behaviour has been implemented correctly by some function [34]. Oracles can be obtained from predicates that characterize the system's expected properties. These predicates can be implemented as contract assertions.

To observe the outputs, the tester should collect the assertions at the system's interfaces and the output generated by the system, such as, messages to users, exit codes, generated files, output to hardware devices, exceptions that appear at the system's interface, among others.

Furthermore, error propagation is also observed. Predecessors and successors are useful for the observation of errors propagation, too. Errors can be cancelled (corrupt data is flushed or overwritten) or hidden (corrupt data remains unchanged but unused); these are considered as tolerated by the system. An error is detected when error handlers are activated. Detected errors are considered as recovered when the error handler successfully recovers the system's state, meaning that the system terminates successfully; otherwise a failure of exception handlers may cause the system to fail.

To observe error propagation, the tester should monitor the exception handlers or other error detection mechanism and monitor the injected component output to check if a bad input produces a bad output.

Table 3 presents the observation points related to the injection points presented in the failure mode in section 4.5. They are useful to monitor the system when the injection has been carried out for:

Table 3: Values to Inject based on Ballista's Approach

Failure Mode	Observation Points
(i)	the direct successors and the predecessors of the target component
(ii)	the direct predecessors of the component
(iii)	the target component or its direct predecessors, the first component in the direct successors' affects chain
(iv)	the target component or its direct predecessors
(v)	the direct successors, the direct predecessors of the target components

In all cases, the exception handler and the system boundary components should be observed to detect the outgoing data and raised exceptions that will be seen by the users.

It is possible that the direct CUT' predecessors or successors do not have enough observability (easiness of monitoring). If so, the tester must find in its respective chain the next predecessor or successor directly linked to the CUT. The newly selected component should be as closer as possible to the CUT.

The observation points will guide the tester in developing, monitoring and runtime checking capabilities. One must bear in mind that the monitoring of the system will generate a logfile. Checking should be selected carefully; too little information logged may not be enough for debugging purposes. Too much information, on the other hand, may be too time-consuming to analyze. One can choose to activate observation points only when a failure is detected, and then re-execute the tests. The risk, in this case, is when the situation that has led to a failure is an intermittent failure and perhaps a non-repeatable one.

5 The Case Study

In this section we present a case study that has been utilized in our experiments. The system's specification used in our case study has been retrieved from [2] and implemented by a PhD student whose work we used to better understand the system and the architectural aspects presented in the following sections [14].

Anderson presented a general approach to engineering protective wrappers as a means to detect error and undesired behaviour in a component-based system, composed by some COTS components. The wrappers were also used to launch appropriate recovery actions. Thus, the protective wrappers, allow detection and tolerance of typical errors caused by unavailability of signals, violations of constraints and oscillations. In [2], the author presented experiments' results using a Simulink model of a steam boiler system, together with an Off-the-Shelf (OTS) Proportional Integral and Derivative controller (PID controller).

5.1 The System

The overall system has two main components: the boiler system and the control system. The control system comprises a PID controller (the OTS items), and the ROS, which is simply the remainder of the control system. The control system is represented by three PID controllers dealing with the feed water flow, the coal feeder rate and the airflow. The ROS consists of: (i) the boiler sensors. These are "smart" sensors, which monitor variables providing input to the PID; (ii) controller: Drum Level, Steam Flow, Steam Pressure, Gas Concentrations and Coal Feeder Rate; (iii) actuators. These devices control a heating burner, which can be ON/OFF, and adjust inlet/outlet valves in response to outputs from the PID controller: Feed Water Flow, Coal Feeder Rate and Air Flow; (iv) configuration settings. These are the "set-points" for the system: Oxygen and Bus Pressure, which must be set up in advance by the operators. Smart sensors and actuators interact with the PID controller through a standard protocol.

5.2 The Architectural Solution

The system's architecture presented in [2] was extended in [14] and was implemented as a fault tolerant system. As a system that integrates OTS components, the system should consider these components as a potential source of faults. The overall software system should be able to support OTS components while preventing the propagation of errors. So the system should be able to tolerate faults that may reside or occur inside the OTS components, but should not be able to directly inspect or modify their internal states or behaviour.

An architectural solution to tackle this problem was presented in [14], to encapsulate a COTS component adding fault tolerant capabilities, aiming to improve error detection and error recovery. The main concept used is the idealised C2 component (iC2C), which is an evolution of the C2 architectural style.

The C2 architecture style [33] is a component-based style that supports large grain reuse and flexible system composition. The components in C2 architecture are integrated by connectors that are responsible for message routing, broadcasting and filtering. Wrappers encapsulate each component to cope with interface and architectural mismatches [13]. In C2 style, the system has a layered architecture.

Each side of a connector may be connected to any number of components or connectors. In C2, requests are messages that flow up the architecture, and their responses (notifications) flow down. Figure 3 shows the Boiler System architecture of the Boiler System that was first presented by [2] and that has now been adapted to the C2 style. By analysing Figure 3 we can infer that the system's boundary is the Conn3 component that represents the interface between the system and the hardware components, and the Boiler Controller that represents the user's interface.

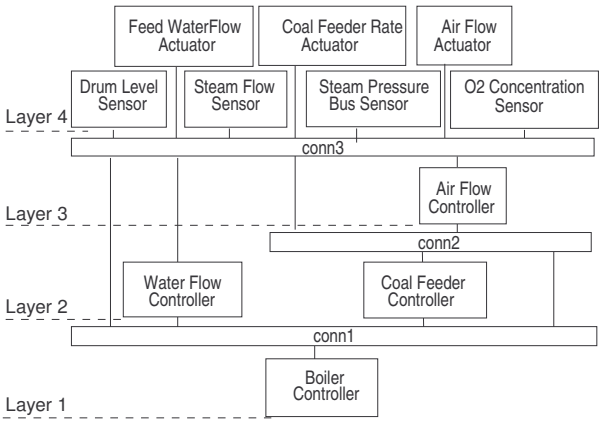


Figure 3: C2 Configuration of the Boiler System

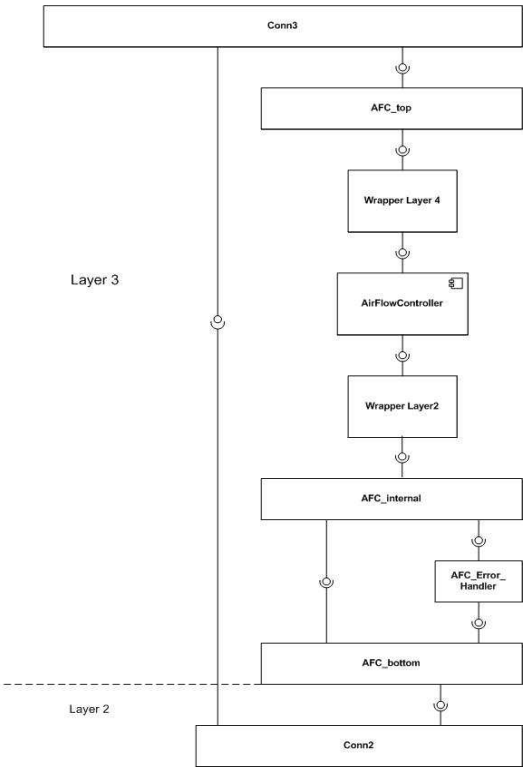


Figure 4: Configuration for the Layer 3 of the Boiler System

(Idealized fault tolerant component)

The iC2C style was proposed to allow the structuring of software architectures compliant with the C2 architectural style [33] [14]. The main idea is the separation of the normal and the abnormal activity parts of the idealised component, in order to minimise the impact of fault tolerance provisions on the system's complexity. The iC2C normal activity component (represented by the Air Flow Controller in Figure 3) implements the normal behaviour. It is responsible for error detection during normal operation as well as for signalling the interface and internal exceptions. The iC2C abnormal activity component (represented by the AFC Error Handler in Figure 3) is responsible for error recovery and for signalling the failure exceptions.

iC2C connectors are specialised reusable C2 connectors and have the following roles: (i) the iC2C_bottom connector (represented by the AFC_bottom in Figure 4), connects the iC2C with the lower components of a C2 configuration and serialises the requests received. When a request is completed, a notification is sent back. This request may be a normal response, an interface exception or a failure exception; (ii) the iC2C_internal (represented by the AFC_internal in Figure 4) controls message flow inside the iC2C selection its destination; (iii) the iC2C_top connector (represented by the AFC_top in Figure 4) connects the iC2C with the upper components of a C2 configuration. The overall structure defined for the iC2C is fully compliant with the component rules of the C2 architecture style, allowing an iC2C to be integrated into any C2 configuration and to interact with components of a larger system.

To improve fault-tolerance capability, a protective wrapper for a COTS software component may be added to the system's architecture, resulting in an idealized COTS component (iCOTS). In this approach, the COTS component (Air Flow controller) is connected to two specialised connectors (Wrapper Layer2 and Wrapper Layer4), which in turn act as error detectors to compose iCOTS normal activity components.

When a constraint violation is detected, the wrappers send an exception notification, which is handled by the abnormal activity component, following the rules defined for the iC2C. The abnormal activity component is responsible for both error diagnosis and error recovery. The abnormal activity component reacts to exceptions raised by the normal activity component and either sends notifications to activate the error handlers or stands as a service provider for requests sent by the error handlers. Figure 4 presents the configuration of the Boiler System, graphically representing all these ideas. The case study consists of a system implementation based on this configuration.

6 Experimental Results and Analysis

6.1 The Case Study Dependency Matrix

We use the architectural model presented in Figures 3 and 4 to construct the dependency matrix for the Boiler System, according to what was explained in Section 4.

For each component with a required interface we create a column in the matrix and for each component that has a provided interface we create a row in the matrix. We also

create a row for the user interface (UI) and a row for the interface with the hardware (actuators and sensors).

When a component requires information to a provided interface of another component, we place a mark (X) in the corresponding cell of the matrix. For example, Conn2 has a required interface connected to Conn3. In the matrix presented in Table 4, we put a mark in the cell that crosses these two components. We also place a mark in the respective cell of the component Boiler Controller with the UI and Conn3 with the hardware, since they represent the boundary components of the system. Table 3 presents the resulted matrix.

Table 4: The Case Study Dependency Matrix

	Boiler Controller	conn 1	Coal FeederCtl	Water Flow Ctl	conn 2	Afc_ bottom	AfcErrorHandler	Afc_ internal	WrapperLayer2	AirFlow Ctl	WrapperLayer4	Afc_ top	conn 3
UI	X												
Conn1	X												
CoalFeederCtl		X											
WaterFlowCtl		X											
Conn2		X	X										
Afc_bottom					X								
Afc_ErrorHandler						X							
Afc_internal						X	X						
WrapperLayer2								X					
AirFlowCtl									X				
WrapperLayer4										X			
Afc_ top											X		
Conn3		X		X	X							X	
Hardware													X

6.2 Determining the Chain for each Component

After constructing the dependency matrix, we have to obtain the related chains for each component, as presented in section 4.3. These chains are represented in Figure 5.

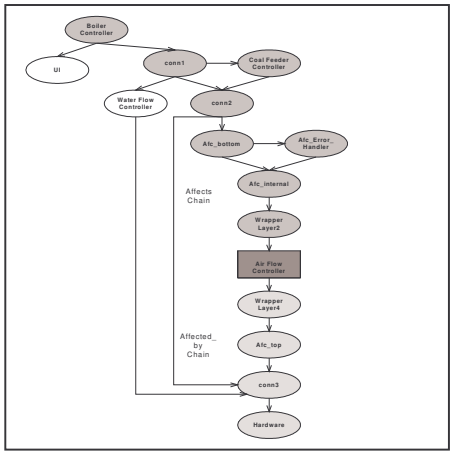


Figure 5: Chain among components

Based on the graph in Figure 5, Table 5 presents the number of components in affected-by and affects chain for each component.

Table 5: Number of affected-by and affects chain

Component	Affected-by	Affects	Total
Boiler Controller	14	0	14
conn1	12	1	13
Coal Feeder Controller	10	2	12
Water Flow Controller	2	2	4
conn2	9	3	12
Afc_bottom	8	4	12
Afc_Error_Handler	7	5	12
Afc_internal	6	6	12
Wrapper Layer2	5	7	12
Air Flow Controller	4	8	12
Wrapper Layer4	3	9	12
Afc_top	2	10	12
conn3	1	12	13

6.3 Selecting the Target Components

According to Table 5, we should select the top 20% components with the highest number of affected-by and affects chain. As we have 13 components in the matrix row, we should select three components. These three components are emphasized in Table 5. We should analyse the chain and select parameters and returned values in accordance with the failure mode presented in 4.5 for each component.

6.4 Summary of the Experiments

Table 6 summarizes the experiments executed with the three selected components. Experiments were also carried out with the other components to compare the results obtained. For the sake of space these experiments are not described, but the values injected are similar, according to Table 2.

Table 6: Injection Points based on the Strategy

Component	Interface Method	Injection Location	Values to Inject
Boiler Controller	setConfiguration	1st Parameter	0, 1, 1.1, -0.9 DBLMax, DBLMin
Boiler Controller	setConfiguration	2nd Parameter	0, 0.1, -0.9, 0.2, DBLMax, DBLMin
BoilerController	connectTop	1st Parameter	null
conn1	connectTop	1st Parameter	null
conn1	setConfiguration	1st Parameter	0, 1, 1.1, -0.9, DBLMax, DBLMin
conn1	setConfiguration	2nd Parameter	0, 0.1, -0.9, 0.2, DBLMax, DBLMin
conn3	ReadO2Concentration	Returned Value	0, 1, 1.1, -0.9, DBLMax, DBLMin
conn3	readSteamFlow	Returned Value	0, 125, 126, -0.9,DBLMax, DBLMin
conn3	readBusPressure	Returned Value	0, 20, 21, -0.9, DBLMax, DBLMin
conn3	setFeedWaterFlow	1st Parameter	0, 1, 1.1, -0.9, DBLMax, DBLMin
conn3	SetAirFlow	1st Parameter	0, 0.1, -0.9, 0.2, DBLMax, DBLMin

6.5 Results

Table 7 contains the results obtained. The column "#Errors" indicates the number of errors injected. The "#Detections" column shows the number of times an error detection mechanism is activated. The next column indicates the number of successful executions, which comprises: (i) number of experiments in which the errors injected were non-effective (errors were masked, flushed or remained latent), (ii) number of experiments where the error detection mechanisms were activated, the exception was handled and the

system recovered execution. The last columns contain the number of failures, which can be reported or non-reported. A reported failure means that an exception was raised and then the system crashed. A non-reported failure represents the case in which invalid values are not detected and the system terminates normally. This is a dangerous situation because the system violates safety conditions.

The reported failure occurred when errors were injected when the connections between components and connectors were being established. In the C2 architecture, these connections are established during runtime. This result indicates that the C2 framework components also need a protective wrapper.

Table 7: Results Obtained

Component	#Errors	#Detections	#Successes	#Failures	
				Reported	Non-reported
Boiler Controller	13	4	12	1	0
conn1	13	4	12	1	0
CoalFeeder Controller	16	0	16	0	0
WaterFlow Controller	1	0	0	1	0
conn2	13	4	12	1	0
Wrapper Layer2	6	4	6	0	0
AirFlow Controller	6	4	6	0	0
Wrapper Layer4	12	8	12	0	0
conn3	30	4	30	0	16
Total	110	32	106	4	16

As expected, the results obtained when injecting the rest of the components (ROC) were similar to the ones obtained when injecting the selected ones. Nevertheless, when analyzing the execution log that Jaca generates, we observed that the components were effectively executed according to the dependency graph in Figure 5.

The system's behaviour was similar when we injected the Boiler Controller component and the conn1 connector. One may think that only one of them could have been selected, but the Boiler Controller is the only component in which fault injection can emulate operator faults (in this case, erroneous values provided through the system interface). On the other hand, by selecting conn1 connector it is possible to affect the Coal Feeder Controller, given that this component is not on the affected-by chain of neither the Boiler Controller nor the conn3 connector.

6.6 Approach Limitations

When fault injection technique is used to validate a system, the intrusion caused by the tool is unavoidable. However, in previous experiments [21] we had evaluated the performance of a system composed by a benchmark application interacting with an object-oriented database management system (ODBMS). To analyze the impact of the tool's intrusion on the ODBMS's performance, the application was executed threefold: 1)without the use of Jaca, 2) using Jaca without the injection of errors and 3) using Jaca to inject errors. The tests were executed using the operations available on benchmark application (create, query match and query traversal). For each experiment the execution timing presented by the benchmark's interface was noted (each injection was repeated five times with the same parameters and the average was taken as result). Before each execution of the experiment, the object manager as well as the operational system were re-initiated so that no information residue from the cache memory or temporary disk storage were used (the aim was to avoid compromising the results). From the results it was possible to see that the benchmark presented a similar performance in all case. Some small variations were observed when Jaca was used only to monitoring and to inject the errors, since the difference between one case and another is the execution of arithmetic operations between the injected error and the value previously existent.

Currently, we are working to test the ODMBS through error injection, with the aim to evaluate the impact of those errors in the application.

Another limitation is the analysis of the system's architecture as we could fall into two situations: (i) when the system is composed of several components and (ii) when there is no information about the system's architecture. In both case we could use a tool to analyze the dependencies. There are some tools, however, that show the dependencies through the analysis of the bytecode [10] and do not need the source code for this purpose.

7 Conclusion and Future Work

This study represents a first step toward our investigation on the use of a risk-based fault injection approach. The main contribution of this study is the use of architectural analysis to guide fault injection. Specifically, a dependency analysis technique was applied to establish the relationships among components, based on the interactions through their provided and required interfaces.

Fault injection can be a valuable approach to validate whether an architectural solution achieves the required reliability level. An advantage in analysing a system's architecture is that fault injection can be planned early in the development process. Another advantage is that it allows dependencies to be established even when the source code is not available, which can be the case when third-party components are used in a system.

Dependency analysis can be used for many purposes, one of them being change impact determination. It can be helpful in fault injection to determine the components that are worth injecting. In other words, under time or cost pressures, fault injection efforts can

be concentrated on those components whose failures may have greater impact on the system. Moreover, when a component of interest has low controllability (or observability), the dependency relationships can be used to determine the target components in which to inject the faults (or to observe): components that can be affected by the one of interest (to determine the impact of a faulty component on the rest of the system) or those that can affect the component of interest (to determine the effect on the latter when the rest of the system fails).

A simple case study and its experiments' results were used to show the benefits of the approach. Clearly, further experiments must be made in order to corroborate these results. The value of this approach will also be tested in a real world application.

Acknowledgment

The authors wish to thank Carlos Eduardo Rodrigues de Almeida , a UNICAMP student, for performing the experiments and also for his invaluable support with the Jaca tool. This research is partly supported by CNPq – Brazil's National Council for Scientific and Technological Development – through the ACERTE project and the State of São Paulo's Research Foundation.

References

1. Allen, R., Garlan, D.: Formalizing Architectural Connection. In: Proc. of the 16th International Conference on Software Engineering, IEEE Computer Society, (1994) 71-80
2. Anderson, T. et. al.: Protective Wrapper Development: A Case Study. Lecture Notes in Computer Science (LNCS), Vol. 2580, Spring Verlag, (2003) 1-14
3. 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 (2), (1990) 166-182
4. Bach, J.: Heuristic risk-based testing. Software Testing and Quality Engineering Magazine, (1999)
5. Beydeda, S., Volker, G.: State of the art in testing components. In: Proc. Of the International Conference on Quality Software, (2003)
6. Chiba, S.: Javassist – A Reflection-based Programming Wizard for Java. In: Proc of the ACM OOPSLA'98 Workshop on Reflective Programming in C++ and Java, (1998)
7. Chen, Z., Xul, B., Zhao, J.: An Overview of Methods for Dependence Analysis of Concurrent Programs. ACM SIGPLAN Notices, Vol. 37, Issue 8 (2002) 45-52
8. Clemens, S.: Component Software: Beyond Object-Oriented Programming, New York, Addison-Wesley, (1998)
9. De Millo, R. A., Li, T., Mathur, A. P.: Architecture of TAMER: A Tool for dependability analysis of distributed fault-tolerant systems. Purdue University, (1994)
10. Dependency Finder tool project, http://depfind.sourceforge.net/, (2005)
11. Fabre, J-C, Rodriguez, M., Arlat, J., Sizum, J-M.: Building dependable COTS microkernel-based systems using MAFALDA. In: Proc. of 2000 Pacific Rim International Symposium on Dependable Computing - PRDC'00, Los Angeles, USA, (2000)
12. Fetzer, C., Högstedt, K., Felber, P.: Automatic Detection and Masking of Non-Atomic Exception Handling. In: Proc. of DSN 2003, San Francisco, USA, (2003) 445-454

13. Garlan, D., Allen, R., Ockerbloom, J.: Architecture Mismatche: Why Reuse is so Hard. IEEE Software, Vol. 12(6), (1995) 17-26
14. Guerra P. et. al.: A Dependable Architecture for COTS-based Software Systems using Protective Wrappers. Lecture Notes in Computer Science (LNCS), Vol. 3069, Spring Verlag, (2004) 147-170
15. Hsueh, Mei-Chen, Tsai, Timothy, Iyer, Ravishankar.: Fault Injection Techniques and Tools. IEEE Computer, (1997) 75-82
16. Inverardi, P., Wolf, A.L., Yankelevich, D.: Checking Assumptions in Component Dynamics at the Architectural Level. In: Proc. of the 2nd International Conference on Coordination Models and Languages, Springer-Verlag, (1997)
17. 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, http://www.ece.cmu.edu/~koopman/ballista/, (2003)
18. Leme, N. G. M.: A Software Fault Injection Systems based on Patterns. Master Thesis, UNICAMP, Brasil, (2001) (in Portuguese)
19. Maes, P.: Concepts and Experiments in Computational Reflection. In: Proc. of OOPSLA'87, (1987) 147-155
20. Martins, E., Rubira, C. M. F., Leme N.G.M.: Jaca: A reflective fault injection tool based on patterns. In: Proc of the 2002 International Conference on Dependable Systems & Networks, Washington D.C. USA, (2002) 483-487
21. Martins, E., Moraes, R.: Testing Component-based Applications in the Presence of Faults. In: Proc. of the 7th World Multi-conference on Systemic, Cybernetics and Informatics, (2003) 112-117
22. Moraes, R, Martins, E.: A Strategy for Validating an ODBMS Component Using a High-Level Software Fault Injection Tool. In: Proc. of the First Latin-American Symposium, LADC 2003, São Paulo, Brazil, (2003) 56-68
23. Moraes, R., Martins, E.: An Architecture-based Strategy for Interface Fault Injection. In: Proc. of the International Conference no Dependable Systems and Networks, Firenze, Italy, (2004)
24. Naumovich, G., Avrunin, G.S., Clarke, L. A., Osterweil, L.J.: Applying Static Analysis to Software Architectures. In: Proc. of the 6th European Software Engineering Conference, Springer-Verlag, (1997)
25. Pressman, R. S.: Software Engineering a Practitioner Approach. Mc Graw Hill1, 4th edition, (1997)
26. Rosa, A.: A Reflexive Architecture to Inject Faults in Object-Oriented Applications. Master Thesis, UNICAMP, Campinas, Brasil, (1998) (in Portuguese)
27. Rosenberg, L, Stapko, R, Gallo, A.: Risk-based Object Oriented Testing. In: Proc. of 13th International Software / Internet Quality Week (QW2000), San Francisco, California USA, (2000)
28. Saridakis, T., Issarny, V.: Developing Dependable Systems using Software Architecture. In: Proc. of the 1st Working IFIP Conference on Software Architecture, (1999) 83-104
29. Shaw, M., Clements, P. C.: A Field Guide to Boxology: Preliminary Classification of Architectural Styles for Software Systems. In: Proc. of the 21st International Computer Software and Applications Conference, (1997) 6-13
30. Sherer, S. A.: A Cost-Effective Approach to Testing. IEEE Software, (1991)
31. Sotirovski, D.: Towards Fault-Tolerant Software Architectures. In: R. Kazman, P. Kruchten, C. Verhoef, H. Van Vliet, editors, Working IEEE/IFIP Conference on Software Architecture, Los Alamitos, CA, (2001) 7-13
32. Stafford, J.A., Richardson, D.J., Wolf, A.L.: Chainning: A Software Architecture Dependence Analysis Technique. Technical Report CU-CS845-97, Department of Computer Science, University of Colorado, (1997)

33. Taylor, R. N., Medvidovic, N., Anderson, K.M., Whitehead Jr.E.J., Robbins, J.E., Nies, K.A., Oreizy, P., Dubrow, D.L.: A Component and Message-based Architectural Style for GUI Software. IEEE Transactions on Software Engineering, Vol. 22(6): 390-406, (1996)
34. Voas, J., McGraw, G.: Software Fault Injection: Inoculating Programs against Errors. John Wiley & Sons, New York, EUA, (1998)
35. Voas, J., Charron, F., McGraw, G., Miller, K., Friedman, M.: Predicting how Badly Good Software can Behave. IEEE Software, (1997) 73–83
36. Voas, J.: Marrying Software Fault Injection Technology Results with Software Reliability Growth Models. Fast Abstract ISSRE 2003, Chillarege Press, (2003)

Capítulo 10 – Injeção de Falhas com base na Análise de Dependência

O artigo reproduzido neste capítulo foi publicado no *First International Workshop on Testing and Quality Assurance for Component-Based Systems*. Assim como o artigo apresentado no Capítulo 9, este artigo também utilizou a técnica *Chaining* (refere-se à Seção 2.2) e a mesma estratégia definida para aquele artigo. Porém, a validação da estratégia foi feita utilizando um sistema que integrava um componente real utilizado diariamente em aplicações comerciais, o gerenciador de base de dados Ozone. O conceito de componente foi aplicado considerando-se para uma granularidade mais fina, quando uma classe do sistema foi considerada um componente. Os experimentos efetuados foram concentrados nas sete classes do componente que apresentavam maior número de elementos no conjunto *related chain*.

Concluímos que a análise de dependência é eficiente para selecionar as classes a serem injetadas e observamos que a classe que continha o maior número de elementos no seu conjunto *affected-by chain*, foi aquela que apresentou os defeitos mais severos. Um resumo do artigo já foi apresentado na Subseção 3.3.2 desta tese.

A referência deste artigo é a que segue: ***Moraes, R.,*** *Martins, E., Mendes, N.* "*Fault Injection Approach based on Dependence Analysis*". *In: Proceedings of First International Workshop on Testing and Quality Assurance for Component-Based Systems - TQACBS, pp. 181-188, 2005*.

Fault Injection Approach based on Dependence Analysis

Regina L. O. Moraes[1]
regina@ceset.unicamp.br
+55 19 3404-7105

Eliane Martins[2]
eliane@ic.unicamp.br
+55 19 3788-5872

Naaliel Vicente Mendes[1]
naalielb@ceset.unicamp.br
+55 19 3845-5754

State University of Campinas (UNICAMP)
Superior Center of Technological Education (CESET)[1] Institute of Computing (IC)[2]

Abstract

Fault Injection is used to validate a system in the presence of faults. Jaca, a software injection tool, is used to inject faults at interfaces between classes of a system written in Java. We present a strategy for fault injection validation based on dependence analysis. The dependence analysis approach is used to help in reducing the number of experiments necessary to cover the system's interfaces. For the experiments we used a system that consists of two integrated components, an ODBMS performance test benchmark, Wisconsin OO7 and an ODBMS, Ozone. The results of the experiments and their analysis are presented.

1. Introduction

Current trend in Software Engineering nowadays is the construction of software systems through the assembly of off-the-shelf (OTS) components. The interest in component-based development lies in the reduction of development time. However, this does not preclude testing. Even popular components, that is, those used in many software systems, must be tested every time they are reused in a new context. Besides, the integration of components can give rise to failures due to errors in the interaction among them.

Validation is a necessary step to establish whether a solution achieves the system's required qualities. It is also important to assess the robustness of the interfaces regarding component failures as well as problems that enter the system from external sources [13].

We propose the use of Software-Implemented Fault Injection (SWIFI) that allows us to observe whether the failures of classes, or interfaces among classes, affect the services provided by the system. Our approach is based on the introduction of interface faults by affecting input or output parameters, as well as returned values. Interface faults may represent a component's failure modes that are consequences of errors that propagate through the interfaces.

In this work, faults are introduced using a software fault injection tool, named Jaca [6], developed to introduce faults into Java applications. Jaca does not require access to an application's source code. All instrumentation needed for fault injection and monitoring purposes are introduced at byte code level.

A strategy must be chosen to select a subset of classes for fault injection when the system contains several classes, because injecting in all of them would be a great effort. The strategy proposed is an adaptation of Chaining [12] based on dependences, to reduce the portions of an architecture that must be examined in order to test or debug a system. For the strategy presented in this work we will consider a component or a class with the same meaning because the strategy can be applied interchangeably to both.

In the following section we give a fault injection overview. The dependence analysis approach is described in Section 3. In Section 4, the proposed strategy is discussed and a case study is presented in Section 5. The tests' results and related works are presented in Section 6 and Section 7 respectively. Conclusions and future works in Section 8.

2. Fault Injection Overview

Fault injection is a technique that corresponds to the artificial insertion of faults into a system aiming to accelerate the errors and failures in order to observe the system's behaviour in the presence of faults in its components or in its environment [13]. Fault Injection techniques have been used to evaluate a system's dependability and to validate its error-handling mechanisms, forcing a system to deal with faults [1].

Among the various existent approaches (see [4] for an overview), software fault injection has been widely used. It emulates software faults as well as faults that occur in external components that somehow affect the software [13].

Software fault injection is useful for robustness testing. This study uses this approach, allowing component users to determine its robustness. The software can be stated as robust if it receives anomalous input and does not propagate into a failure, demonstrating that the software can produce dependable service even in presence of an aggressive external environment [14].

To test the system's robustness due to component failures we inject faults during runtime. For this purpose,

Jaca, a software injection tool developed in a previous work [6] is used. It injects faults into object-oriented systems written in Java. Jaca uses reflective programming to inject interface faults. Computational reflection allows the target system's instrumentation to carry out its functions through introspection (useful for monitoring) or by altering the system during runtime (useful for the injection) without changing the system's structure.

Jaca does not need the application source code to perform fault injection allowing the validation of third-party components. The instrumentation is introduced at byte code during load time. The tool gets a class's interface information through introspection when the source code is not available and can alter attributes' values, method's parameters and return values. Jaca is described in more detail in [6].

3. Dependence Analysis

Dependence analysis has been traditionally based on control and data flow relationships associated with functions and variables of a program. It has been used widely in program understanding, testing, debugging, reverse engineering, and maintenance [3]. A limitation of the traditional approaches is that they are generally source-code based. This is not useful to us as the source code of some OTS components might not be available. Hence, in this work we consider dependences based on the classes' diagram of the system.

A problem that may occur with this approach is the polymorphic behavior and dynamic binding. This means that a receiver of a polymorphic message is only known at runtime. In this way, when we have several implementations to the same interface in the system's classes diagrams we consider all of them.

Dependences occur when one class uses the services of another class [12]. Dependence relationship may appear among classes of one component or among interoperated classes of different components. The approach used in this work is an adaptation from Chaining, a software architecture dependence analysis technique that was primarily aimed to reduce the part of the architecture to be examined for a given purpose [12].

There are three types of chains: (i) affected-by chains, which contain the set of components that could potentially affect the component of interest; (ii) affects chains, which contain the set of components on which the component of interest can potentially have an effect; (iii) related chains, which are a combination of affected-by and affects chains. The next section presents how the chaining technique can be useful for fault injection.

4. Dependence Analysis for Fault Injection

Dependence analysis can be useful for fault injection in many ways: (i) to help select target classes whose interfaces are to be injected; (ii) to establish monitoring points to determine error propagation, and (iii) to select a minimal set of classes to monitor for debugging purposes in case of system failure. The dependence analysis considers the class diagram instead of the system architecture as we did in [7].

Classes with high affects chain and affected-by chain may be possible targets for fault injection and the set of classes in its affects chain could be also used to monitor. Figure 1 illustrates the chains.

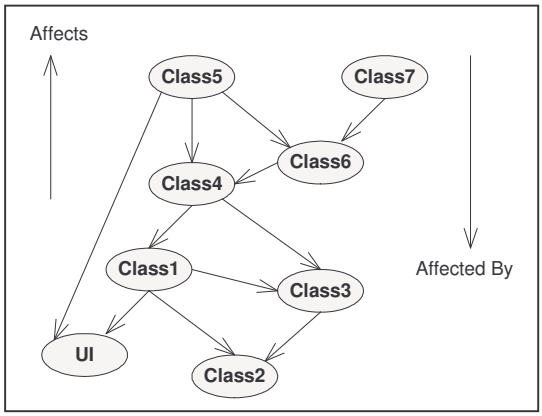


Figure 1: Chains

In this figure, the class designated as Class3 has 1 affected-by chain and 5 affects chain. Faults in a link between the target class and an affects chain class can have an effect on other classes that do not need to be directly injected, representing failure modes in classes used by the target one (its successors) and could be impacted by a change in the target class. Conversely, faults in a link with an affected-by class represent failure modes from classes that use the target one (its predecessors) and may affect it.

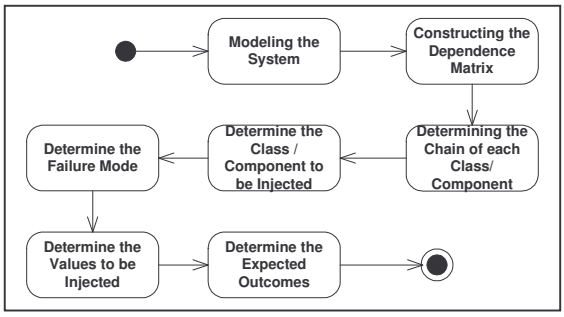


Figure 2:Steps of our Approach

If a selected class does not have the required observability (easiness of monitoring) the observation point should be transferred to another class in the same chain (affected-by or affects). In this way, we can also assess the failure tolerance of the interfaces regarding class failures and corruptions that may enter into the system from external sources. The steps included in our

approach are presented in the activity diagram in Figure 2. The following sections describe it in more detail.

4.1. Modeling the System

The dependences among classes to be modeled could be inheritances, polymorphism, attributes whose types are of another class, pointers to other class or methods of a class that calls a method on an object of another class [12]. Figure 3 presents, using UML notation, an example of some static dependences considered.

4.2. Constructing the Dependence Matrix

We use the dependence matrix [12] to represent the class' relationships. In the dependence matrix, the columns represent the dependent and the rows represent the object of dependence plus any events generated in the system's environment that can be acquired by an interface (e.g. a user interface - UI).

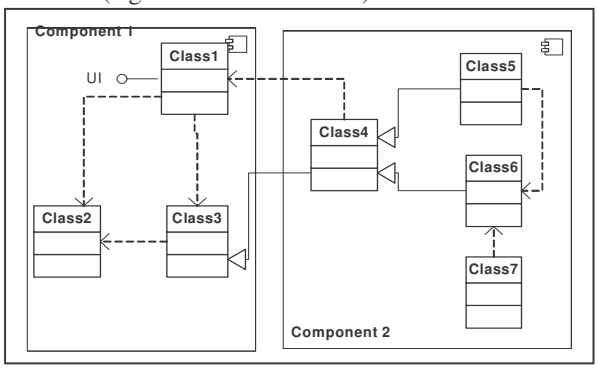


Figure 3: The System's Diagram

Table 1: Dependence Matrix

	Class1	Class3	Class4	Class5	Class6	Class7
Class1			X			
Class2	X	X				
Class3	X		X			
Class4				X	X	
Class6				X		X
UI	X					

Thus, if class A is dependent on B, the cell at column A and row B hold that relationship (indicated by an "X" in this cell). Table 1 shows the matrix corresponding to the diagram in Figure 3.

4.3. Determining the Chain for Each Class

A class's chain can be determined by creating links when there is a mark (X) in a cell, which indicates that a relationship holds.

Let M designates a dependence matrix. If M[Class1, Class2] = 'X' then Class1 ∈ set of affected-by from Class2 and Class2 ∈ set of affects from Class1.

Both affected-by and affects chain are transitive relations. In this way, we construct the affects-by (affected-by and affects) chains of a class by using the transitive closure of the relationship. In other words, if Class1 ∈ set of affected-by from Class2 and Class2 ∈ set of affected-by from Class3, then Class1 belongs to the set of affected-by from Class3.

Using the above concepts we can construct the dependence graph for the example system whose classes are presented in Table 1. This graph is shown in Figure 1.

4.4. Determining the Candidate Targets

To validate the system, the tester should select a class or a set of classes where to inject the faults. For this purpose, in this work we are considering the number of classes in the affects chain and affected-by chain of each class. Based on Pareto's 80/20 rule [9], the classes whose sum of both chains is classified between the top 20% must be selected for fault injection. These classes may affect many others, so we can observe the impact of faults without having to inject all the interfaces.

Based on Table 1, where we have seven classes and the UI, if we apply Pareto's 80/20 rule we should inject faults into one or two classes. There is a tie of this score among three classes, C1, C4 and C6 and in this case, even though Pareto's rule indicates one or two classes to be selected, we should select all three classes.

We also need to cope with constraints on controllability (easiness to inject faults) imposed by fault injection tools. Jaca, namely, can only inject and observe classes written in Java, and in this version can only affect parameters or return values that are not objects. If the selected class is not considered controllable, the related chain can be used to select another target class.

4.5. Determining the Failure Mode

Once the classes that should be tested are selected, each class should be considered (CUT) in order to determine in which of their interfaces, faults should be injected. Our focus is parameters and the returned values that flow between classes. The failure modes are:

(i) input parameters provided to the target class by its predecessors (using links in affected-by chain);
(ii) a returned value provided by the target class to its predecessors (using links in affected-by chain);
(iii) input parameters provided by the target class to its successors (using links in affects chain);
(iv) a returned value provided to the target class by its successors (using links in affects chain);
(v) a parameter or returned value that comes into the system through its boundary components.

4.6. Determining the Values to Inject

To determine which values we should inject into input/output parameters or returned values of the operations in a class's interface, one should consider the class's specification when there are valid domains of these values or constraints imposed by the system's specification. We used the Ballista approach [5] to select these values. Table 2 presents the values used according to each data type.

Table 2: Values to Inject based on Ballista's Approach

Data Type	Values to Inject
Integer	0, 1, -1, MinInt, MaxInt, neighbour value (current value ± 1)
Real Floating Point	0, 1, -1, DBLMin, DBLMax, neighbour value (current value * 0.95 or * 1.05)
Boolean	**inversion of state (true -> false; false ->true)**
String	Null

4.7. Determining the Expected Outcomes

An output value space is the set of all possible output values of the program. In this set, as a result of an experiment, the system may fail or tolerate the injected faults. Tolerance to the injected faults means that the system outputs the expected results. Failures may be a reported failure (raised exception or returned wrong values) or non-reported failure (application hangs, application crashes or erroneous values).

In order to decide if an output value is a wrong value or not, an oracle mechanism is needed. An oracle is a predicate on input/output pairs that check whether the desired behavior has been implemented correctly by some function [13]. For that purpose, data is collected during the experiments, such as, messages to users, exit codes, generated files, output to hardware devices, exceptions at the system's interface and so on.

Furthermore, error propagation is also observed, through the predecessors and successors of a target class. So, we should also monitor the exception handlers or other error detection mechanism and the injected class output to check if a bad input produces a bad output. The system boundary should be observed to detect the outgoing data and raised exceptions that will be seen by the users Table 3 presents the observation points related the failure mode in section 4.5.

If the direct CUT's predecessors or successors do not have enough observability (easiness of monitoring) one must find in its respective chain the next predecessor or successor directly linked to the CUT. The new selected class should have the minimal distance from the CUT when you consider the dependence transitivity chain.

The observation points will guide the tester in developing, monitoring and runtime checking capabilities. Checking should be selected carefully, since the information in the log file should be the minimum enough for debugging purposes. One can choose to activate observation points only when a failure is detected, and then re-execute the tests. The risk is that the failure could be intermittent failure and perhaps a non-repeatable one.

Table 3: Observation Points based on Failure Mode

Failure Mode	Observation Points
(i)	direct successors and predecessors of the target class
(ii)	direct predecessors of the class
(iii)	target class or its direct predecessors, the first class in the direct successors' affects chain
(iv)	target class or its direct predecessors
(v)	direct successors, direct predecessors of the target classes

5. The Case Study

In this section we present a case study that has been used in our experiment. The system is a composition of two main components, our target component, an object-oriented database management system (ODBMS) called Ozone and our target application, a benchmark used to evaluate ODBMS, the Wisconsin OO7 benchmark.

5.1 The target component

Ozone is an object management system written in Java that allows for Java persistent objects in a transactional environment [8]. Persistent objects are programmed following the syntax of the programming language and all of its structure is stored according to the application's definition.

The Ozone environment is based on a client/server architecture. To avoid object replication, Ozone uses a unique instance architecture, where the actual instance resides in the server. At the client, objects are controlled via "proxy" objects, seen as a persistent reference.

The experiments performed so far use a local configuration where Ozone in the same machine as the client application.

5.2 The target application

The Wisconsin OO7 benchmark [2] was taken as our target application. The main component of the OO7 benchmark is a set of composite parts. The set of all composite parts forms the design library that has a. document object associated to it. The composite part also has an associated graph of atomic parts. Assembly objects may be composed of composite parts (base assembly) or other assembly objects (complex assembly). Assemblies are organized in hierarchies, each of which constitutes a module. Associated with a module is a manual that documents the module.

The benchmark can create three database sizes: large, medium and small. In this study we are using the small size [2].

Even though the original functions are useful to evaluate Jaca's performance and efficacy in injecting

faults, they are insufficient to verify the ACID properties. Thus, extra functions were implemented: to insert a new composite object, delete a composite object, search an object document, search an object manual and query the hierarchy tree starting from a base assembly object.

Table 4: The System Dependence Matrix

	Bench Mark	OO7_ Assembly	OO7_ Atomic Part	OO7_Base Assembly	OO7_ Complex Assembly	OO7_ Composite Part	OO7_ Connection	OO7_ Design Object	OO7_ Document	OO7_ Manual	OO7_ Module
OzoneObject	X						X	X	X	X	
DxArrayBag	X		X	X	X	X					X
DxVectorIterator	X										
Database	X										
ExternalDatabase	X										
OzoneProxy	X						X	X	X	X	
RemoteDatabase	X										
DxListBag			X	X	X	X					X
OO7_Assembly	X			X	X						X
OO7_AtomicPart	X					X	X				
OO7_BaseAssembly	X					X					
OO7_ComplexAssembly	X	X									X
OO7_CompositePart	X		X	X							
OO7_Connection	X		X								
OO7_DesignObject		X	X			X					X
OO7_Document	X					X					
OO7_Manual	X										X
OO7_Module	X	X								X	
UI	X										

5.3 The system's dependence analysis

We use the ODBMS and the target application class diagram to analyses the dependences among the classes, and to build the dependence matrix presented in the next section. For lack of space the class diagram was not presented, please see [2].

6. Experimental Results and Analysis

6.1 The Case Study Dependence Matrix

The dependence matrix for the system composed by OO7 and Ozone components was constructed according to what was explained in Section 4. For the analysis purposes, we ignore the interface classes and transfer their dependences to the classes that implement each of them. Table 4 presents the obtained matrix.

To calculate the related chains we also need the dependences for the Ozone classes with other classes of the ODBMS. A new matrix should be constructed. For lack of space, this matrix is not presented here.

6.2 Determining the Chain for each Class

After constructing the dependence matrix, we obtained the related chains for each class, as presented in section 4.3. The size of each class' chains is represented in Table 5.

Table 5: Number of affected-by and affects chain

Class	Affected-by	Affects	Total
OO7_AssemblyImpl	26	8	34
OO7_BaseAssemblyImpl	27	5	32
OO7_AtomicPartImpl	26	5	31
OO7_ConnectionImpl	26	5	31
BenchmarkImpl	31	0	31
OO7_ComplexAssemblyImpl	23	8	31
OO7_CompositePartImpl	26	5	31
OO7_Module	21	8	29
OO7_Manual	15	8	23
Cache ObjectContainer	14	9	23
Ozone Proxy	14	9	23
Database	14	9	23
External Database	14	9	23
Remote Database	14	9	23
Local Database	14	9	23
Transaction	14	9	23
OO7_DesignObjectImpl	12	9	21
External Transaction	12	9	21
Client Cache Database	12	9	21
OO7_Document	12	5	17
Object Container	0	11	11
Ozone Compatible	0	10	10
Transaction Manager	0	10	10
DxListBag	0	10	10
DbRemoteClient	0	9	9
Ozone Remote	0	9	9
DxArrayBag	0	9	9
Ozone Object	2	0	2
DxVector Iterator	0	1	1

6.3. Selecting the Target Classes

To select the classes to be injected, we analyze Table 5 and select the top 20% classes with highest number of both chains. As we have 27 classes (number of row) in the table, we should select five or six classes. There is a

tie of score among five classes and for that reason we selected seven classes. These seven classes are emphasized in the Table 5. We must analyze the chain and select parameters and returned values in accordance to the failure mode presented in 4.5 for each one of them.

6.4. Characterization of System's Behavior

In order to characterize the system's behavior in the presence of faults we can envisage the following situations: i) ideal case, in which both OO7 and Ozone have normal termination and the database created is in a consistent state; ii) an exception is generated at OO7, as consequence of fault injection, but Ozone has normal termination and the database created is in a consistent state, characterizing the robustness of Ozone with respect to application failures; iii) an exception is thrown by Ozone, which terminates abnormally, but the database created is in a consistent state; and finally, iv) Ozone terminates abnormally and the database created is in an inconsistent state, characterizing failure of the database manager, in that it allows stored data to be corrupted in consequence of non-successful transaction. The injection campaigns' results will be evaluated in accordance to the situations (i) to (iv).

6.5. Fault Injection

In the injection campaign we aimed to observe the behavior of the system when faults have been injected in the top 20% classes with a high number of chains.

Errors can be cancelled (corrupt data is flushed or overwritten) or hidden (corrupt data remains unchanged but unused); these are considered as tolerated by the system. An error is detected when error handlers are activated. Detected errors are considered as recovered when the error handler successfully recovers the system's state (terminates successfully); otherwise a failure of exception handlers may cause the system to fail. Table 6 presents the injections done and Figure 4 presents the results.

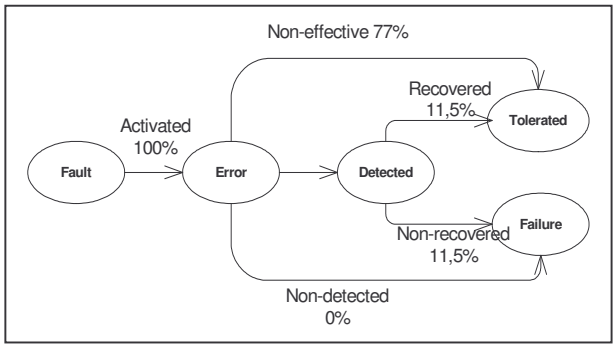


Figure 4: The Campaign results

In Table 6, column 1 and 2 presents the injected class and the method's name, column 2 the injection points (parameters and return values), its type and number of injection done in it in accordance with Ballista's values. The column "Failure Mode" presents the referenced failure mode of this point in accordance with Section 4.5 and the last column the points of observation based on Table 3.

Table 6: The Injection Campaign

Class	*Method*	Injection Point / Type / # injections	Failure Mode	Observation Point
Benchmark	create	First Parameter / int / 7 injections	(i)	OO7_CompositePart OO7_Module OO7_ComplexAssembly
Benchmark	addComposite	FirstParameter / string / 2 injections	(i)	OO7_CompositePart, OO7_Module OO7_ComplexAssemblyImpl
OO7_Assembly	getId	Method return value / int / 2 injections	(iv)	Benchmark
OO7_AtomicPart	getId	Method return value / int 2 injections	(iv)	Benchmark
OO7_AtomicPart	setDocId	First Parameter / long / 2 injections	(iii)	Benchmark, OO7_CompositePart
OO7_CompositePart	getId	Method return value / int / 2 injections	(iv)	Benchmark
OO7_CompositePart	getIdDocumentation	Method return value / int / 2 injections	(iv)	Benchmark
OO7_Connection	setLength	First Parameter / long / 2 injections	(iii)	Benchmark, OO7_AtomicPart
OO7_Connection	length	Method return value / long / 2 injections	(iv)	Benchmark
OO7_BaseAssembly	getId	Method return value / int / 2 injections	(iv)	Benchmark
OO7_ComplexAssembly	GetId	Method return value / int / 2 injections	(iv)	Benchmark

7. Related Works

The use of fault injection to validate component-based systems is an active research area. This work is an evolution of those presented in [7], tackling here the dependences criterion but considering classes instead of components.

A closely related approach to the one presented here is the Interface Propagation Analysis (IPA) [13, ch.9.2]. IPA, inject faults at interfaces. Although we did not take into account the source code, we are considering that the design class diagram is known.

The dependence analysis used in our work is strongly influenced by the study in [12], we use this idea to select the classes in which to inject and to monitor.

From the Ballista approach [5] we derive the definition of error model, which is proposed by the authors as a means to test robustness.

We also borrow ideas from studies that use risk for test costs' reduction. Many risk-based testing strategies have been proposed [10] [11].

8. Conclusion and Future Works

A strategy for fault injection validation based on dependence analysis was applied in a real application. Ozone is a real component used in several everyday software, to manage the transactions that need to persist objects in the database. We believe that each stable release was hardly tested, due to the messages exchanges in its developer's list, where several experienced developers have been working in code changes and tests.

We had performed experiments in the top seven classes with highest number of related chain. If we had to injected in all classes we will need to perform at least 3 times more experiments only to cover the application's classes and the component's classes that have a directly relationship with the application. The proposed strategy allows us to save time and money and helps to determine where to inject faults or errors in case that it has controllability problems.

Nevertheless, in 11,5% of our experiments the database was corrupted by the application error that was propagated to the database presenting a type iv failures (referred to Section 6.4) that may be considered as severe in a database application. This result may indicate that fault injection is an interesting technique to complete the tests previously done.

The dependence analysis was effective in helping the selection of the classes to be injected. The class that presented more severe failures was the one that had greater affected-by chain. The class with the greatest related chain also presented an exception but Ozone tolerated it, indicating a less severe failure but even so a failure was raised in that class.

In future works we intend to use clusters in order to reduce the experiments and evaluate if the results with and without cluster lead to the same results.

Acknowledgment

The authors wish to thank Carlos Eduardo Rodrigues de Almeida , UNICAMP student, for his invaluable support with the Jaca tool. This research is partly supported by CNPq – Brazil's National Council for Scientific and Technological Development – through the ACERTE project and the State of São Paulo Research Foundation (FAPESP).

References:

1. 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*, 16 (2), pp. 166-182 , Feb/1990.
2. Carey, M. J., DeWitt, D. J., Naughton, J. F.: The OO7 Benchmark. http://www.columbia.edu/, 1994, recovered Feb/2005.
3. Chen, Z., Xul, B., Zhao, J.: An Overview of Methods for Dependence Analysis of Concurrent Programs. *ACM SIGPLAN Notices*, Volume 37, Issue 8 , pp. 45-52, Aug/2002.
4 Hsueh, Mei-Chen, Tsai, Timothy, Iyer, R.: Fault Injection Techniques and Tools. *IEEE Computer*, pp. 75-82, Apr/1997.
5 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, www.ece.cmu.edu/~koopman/ballista/, 2003.
6 Martins, E., Rubira, C. M. F., Leme N.G.M.: Jaca: A reflective fault injection tool based on patterns. *Proc of the 2002 Intern Conference on Dependable Systems & Networks, Washington D.C. USA*, 23-267, pp. 483-487 June/2002.
7 Moraes, R., Martins, E.: Fault Injection Approach based on Architectural Dependences. Architecting Dependable Systems III, Lecture Notes in Computer Science, Springer-Verlag Berlin Heidelberg New York. (to appear).
8 Ozone, Object Oriented Database Management System, www.ozone-db.org/, 2004.
9 Pressman, R. S.: Software Engineering a Practitioner Approach, 4th edition, Mc Graw Hill1, 1997.
10 Rosenberg, L., Stapko, R. Gallo, A. Risk-based Object Oriented Testing. *13th International Software / Internet Quality Week (QW2000)*, San Francisco, California USA, 2000.
11 Sherer, Susan A.: A Cost-Effective Approach to Testing, *IEEE Software*, March 1991.
12 Stafford, J. A., Richardson, D.J., Wolf, A.L.: Chainning: A Software Architecture Dependence Analysis Technique, *Technical Report CU-CS845-97*, Department of Computer Science, University of Colorado, September/1997.
13 Voas, J., McGraw, G.: Software Fault Injection: Inoculating Programs against Errors, John Wiley & Sons, New York, EUA, 1998.
14 Voas, J. (2003) "Marrying Software Fault Injection Technology Results with Software Reliability Growth Models", Fast Abstract ISSRE 2003, Chillarege Press.

Capítulo 11 – Melhorando a Dependabilidade de Componentes

O artigo reproduzido neste capítulo foi publicado no Sétimo Workshop de Testes e Tolerância a Falhas. O objetivo deste artigo foi um melhor entendimento da importância das soluções arquiteturais para a técnica de injeção de falhas. O artigo avalia o uso de *wrappers* na arquitetura do software e compara a robustez de um sistema quando não está protegido por *wrappers*. A seguir, aplicam-se os mesmos experimentos sobre o mesmo sistema que é modificado adicionando *wrappers* para proteger o componente alvo.

Concluímos que os *wrappers* adicionados foram eficientes para proteger o sistema alvo melhorando significativamente a robustez do sistema. O artigo já foi citado na Subseção 3.3.2.

A referência deste artigo é a que segue: *Mendes, N.,* ***Moraes, R.,*** *Martins, E., Madeira, H.* "*Melhorando a Dependabilidade de Componentes com o uso de Wrappers*". In: *Proceedings of VII Workshop de Testes e Tolerância a Falhas – WTF´06, pp. 171-181, 2006*.

Melhorando a Dependabilidade de Componentes com o uso de Wrappers

Naaliel Vicente Mendes1, Regina Lúcia de Oliveira Moraes2, Eliane Martins3, Henrique Madeira1

[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 robustway. 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.

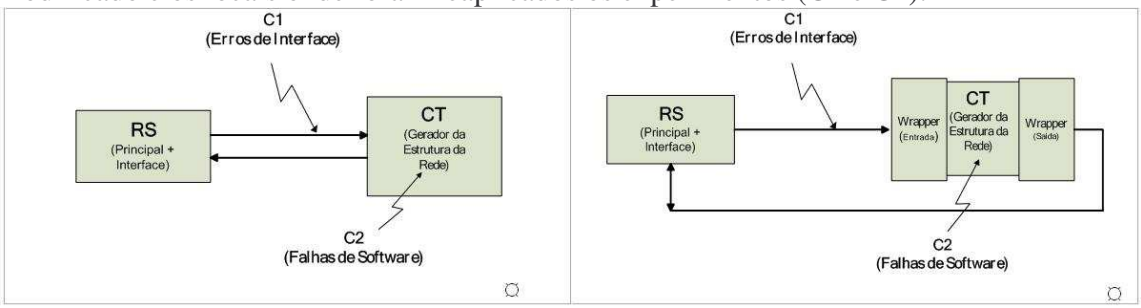
Tabela 2: Pontos de Injeção, Domínios e Valores Injetados – Testes de Robustez

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

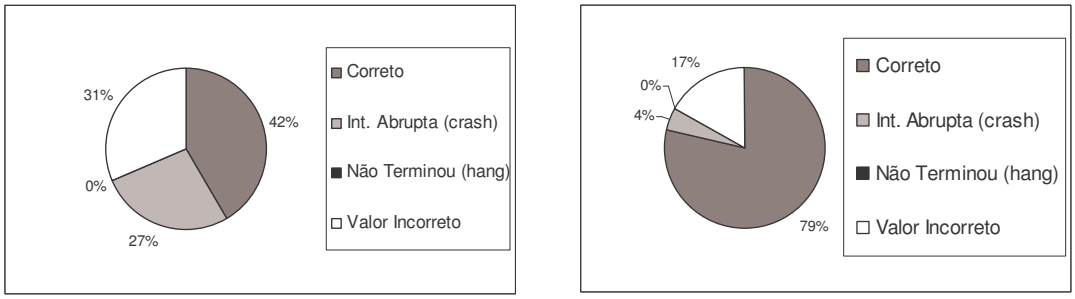
Na primeira campanha de injeção, foi considerado o sistema original (CT + RS). Esta campanha tinha como objetivo avaliar a robustez do CT. Para isso foram injetados erros de interface entre o RS e o CT, conforme apresentado na Tabela 2. Durante os experimentos, foram feitas 89 injeções. A Tabela 4 (sistema 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

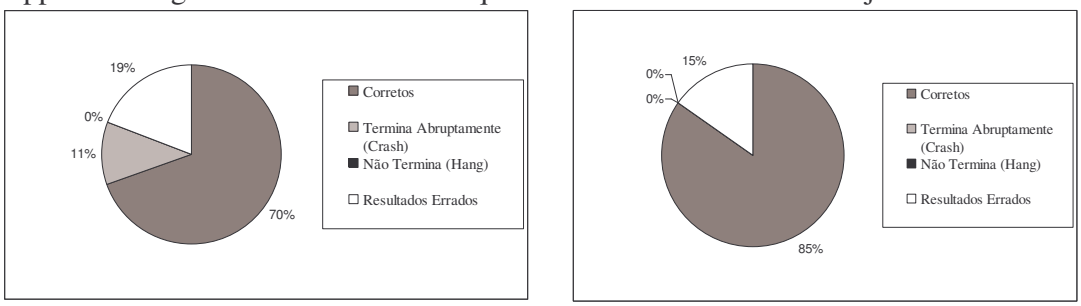
A segunda campanha de injeção foi feita para verificar o impacto no RS quando uma falha no CT é ativada. Injetamos falhas de software no CT conforme apresentado na Tabela 3. A Tabela 5 apresenta os resultados classificados de acordo com o comportamento observado após os experimentos para o sistema original. 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).

Capítulo 12 – Comparação de Falhas de Interface e Falhas Internas

O artigo reproduzido neste capítulo foi publicado no Sixth *European Dependable Computing Conference*. A generalização do uso da injeção de falhas de interface para a extração de medidas de confiabilidade exige uma representatividade das falhas injetadas. Até onde conhecemos, não havia registro do estudo desta representatividade até a publicação deste artigo. Como tínhamos interesse em utilizar a injeção de falhas para a avaliação experimental do risco que um componente de software apresenta para o sistema em que está integrado, este estudo foi desenvolvido.

A pergunta que tentamos responder é: serão os erros injetados nas interfaces estatisticamente equivalentes às conseqüências de falhas residuais que se encontram nos componentes predecessores? Caso se confirmasse esta equivalência, a injeção de falhas internas poderia ser substituída pela injeção de falhas de interface que são menos complexas.

O estudo, que comparou as conseqüências das falhas injetadas no interior dos componentes (falhas de software) com as conseqüências dos erros injetados nas interfaces (falhas de interface) concluiu que as falhas de interface não representam as falhas internas, uma vez que o comportamento do sistema difere substancialmente nos dois casos. Pode-se observar que existe uma intersecção, mas as falhas de interface não cobrem as falhas internas. Um resumo do artigo já foi apresentado na Seção 4.1 desta tese.

A referência deste artigo é a que segue: ***Moraes, R.,*** *Barbosa, R., Durães, J., Mendes, N., Martins, E., Madeira, H.* "*Injection of faults at component interfaces and inside the component code: are they equivalent?*". *In: Proceedings of Sixth European Dependable Computing Conference – EDCC´06, pages 53-62, 2006*.

Injection of faults at component interfaces and inside the component code: are they equivalent?

R. Moraes1, R. Barbosa2, J. Durães3, N. Mendes3, E. Martins1, H. Madeira3
[1]*State University of Campinas, UNICAMP, São Paulo, Brazil*

[2]*Critical Software SA, Coimbra, Portugal*

[3]CISUC, University of Coimbra, Portugal

{regina@ceset, eliane@ic}.unicamp.br, rbarbosa@criticalsoftware.com, jduraes@isec.pt, {naaliel, henrique} @dei.uc.pt

Abstract

The injection of interface faults through API parameter corruption is a technique commonly used in experimental dependability evaluation. Although the interface faults injected by this approach can be considered as a possible consequence of actual software faults in real applications, the question of whether the typical exceptional inputs and invalid parameters used in these techniques do represent the consequences of software bugs is largely an open issue. This question may not be an issue in the context of robustness testing aimed at the identification of weaknesses in software components. However, the use of interface faults by API parameter corruption as a general approach for dependability evaluation in component-based systems requires an in depth study of interface faults and a close observation of the way internal component faults propagate to the component interfaces. In this work we present the results of experimental evaluation of realistic component-based applications developed in Java and C using the injection of interface faults by API parameter corruption and the injection of software faults inside the components by modification of the target code. The faults injected inside software components emulate typical programming errors and are based on an extensive field data study previously published. The results show the consequences of internal component faults in several operational scenarios and provide empirical evidences that interface faults and software component faults cause different impact in the system.

1. Introduction

Fault injection techniques have been extensively used to evaluate specific fault tolerance mechanisms and to assess the impact of faults in systems and have progressively been adopted by the computer industry [1]. In general, the types of the faults injected emulate hardware transient faults. However software faults are currently more relevant than hardware faults as software is becoming extremely complex and the most common types of faults discovered in deployed systems are in fact software faults.

The problem of emulation of software faults by fault injection has been addressed in the last decade [8, 9, 22, 27], leading to practical techniques that emulate software faults with acceptable accuracy [11, 12].

An alternative to the actual injection of software faults inside components is the emulation of its effects by injecting exceptional or invalid values at the components interface. In this context it is assumed that these exceptional or invalid values represent in fact the possible consequences of real software faults in the preceding components/programs (i.e. faults in the component that requires some information through an API call and sends parameters to the target component). Although this approach has provided useful results concerning the exposure of hidden robustness weaknesses and how to remedy them (e.g., through wrappers), the issue of the representativeness of the values injected relative to effects of real residual faults is still an important issue, particularly in contexts of dependability benchmarking.

It is worth noting that the use of exceptional or invalid values at the system/component API interface has been largely used in the context of robustness testing. In fact, robustness testing has been used with remarkable success to expose robustness weaknesses of operating systems [13, 18, 19, 26], and it relies on a simple and logical argument: if a faulty program invokes another (e.g., the operating system through its API) with abnormal parameters, the invoked program should behave in a robust way and not crash in any circumstances. In this context, fault

representativeness is not an issue in classical robustness testing, as this kind of testing does not specifically require the emulation of realistic software faults. In this paper we do not evaluate the equivalence of the robustness testing to the effects of actual residual faults. Instead, we compare the impact of interface faults with the effects caused by faults injected inside the component.

Although started as a robustness testing approach, the idea of injecting faults at interface level has been generalized to be applied to fine grain software components (and not just to operating systems). This generalization of robustness testing is often called as component interface fault injection and has been used to evaluate the impact of faulty components (simulated by erroneous values passed to subsequent components) in the rest of the system [24, 25, 30]. Additionally, component interfaces faults also emulate component incompatibilities, such as appropriated components that can not interoperate [32].

Unlike what happens to robustness testing, the use of interface fault injection as a general approach for component-based systems evaluation raises the question of knowing whether the faults injected at interface level do represent possible consequences of residual software bugs in preceding components. Unfortunately, the mapping between component software faults and interface faults is not clear, which means that the representativeness of interface fault injection experiments may be questionable. To the best of our knowledge there are no works that address the possible equivalence between software faults in a given component and the interface faults injected in the outgoing calls from that component to the remaining system.

This work addresses this problem by observing the visible effects of realistic software faults in a given component, as they may appear at interface level of other components. This experiment gives a clear contribution to the knowledge about interface faults by identifying how software fault injection can guide interface fault injection (and even robustness testing) experiments. Our study also contributes to a better understanding of error propagation in component based system, which has been a central issue in many recent researches (e.g., [14, 17]).

The paper is organized as follows: Section 2 provides a short overview of fault injection at component interface level and injection of software faults inside the components. The experimental setup and the tools that were used in the experiments are presented in Section 3. Section 4 presents and discusses the results obtained and Section 5 concludes the paper and outlines our future work.

2. Two ways of Emulating Software Faults

Software components may contain faults and are exposed to faults from the surrounding environment (other components and systems). Therefore, a major challenge for designers and programmers is to assess how sensitive the system is to faults in any of its software components. Internal and interface faults can be correlated, at least in a partial way, as interface faults can represent the error propagation caused by an internal fault from the preceding components.

The emulation of internal faults may conceivably be achieved through the injection of interface faults, which is relatively inexpensive to develop and execute, as the fault injection mechanism would use the normal component interfaces. However, this assumption requires that the faults injected at the interface level must present the same impact of the errors propagated by the faults that may realistically exist inside the preceding components.

Another approach for the emulation of software faults is the actual modification of the target code in order to inject software faults defined according the most frequent types of real software faults found in field studies (e.g., [12]). This approach offers a better representativeness as the modification injected in the code reproduces the instruction sequences that would be present in the target code if the intended fault were indeed there in the first place (i.e., if a given programming error were indeed present at the target source code). The reproduction of the programming errors directly at the executable code level presents technical challenges and the modification of the target code may be not acceptable in several evaluation contexts [20].

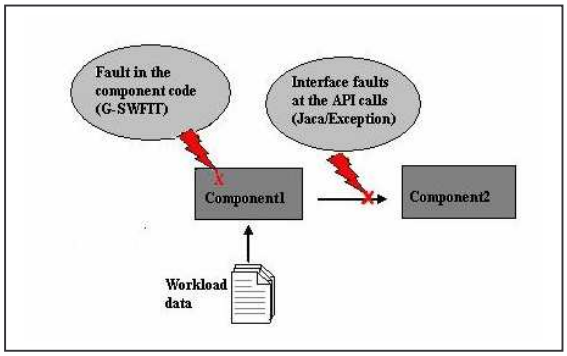


Figure 1: Fault Injection Location

It is worth noting that internal component faults are used in the present work to provide a better understanding on how interface faults can be compared to the injection of software faults, and how

both techniques can be used together. Thus, the modification of the target code is not an issue concerning the goals of the present paper. Figure 1 presents a schema to explain where internal and interface faults are injected.

2.1. Injection of Software Faults

Few studies have addressed the problem of injection of software faults, especially when compared to the vast literature on injection of hardware faults. Fault injection using simple fault models has been used with success in several research works where software faults are the most relevant class of faults. Several examples of software weaknesses revealed by faults injected at random can be found in [30]. Other examples of fault injection used for software testing can be found in [2, 3]. Software fault injection emulated through simple program code corruption was also used to validate fault tolerance mechanisms [28].

The problem of injecting representative software faults was first addressed in [8]. That work was done in the context of IBM's Orthogonal Defect Classification (ODC) project [7] and the proposed method requires field data about real software faults in the target system or class of target systems. This requirement (the knowledge of previous faults) greatly reduces the usability of the method, as this information is seldom available and is simply not possible to obtain for new software. Furthermore, as shown in [22], typical fault injection tools are not able to inject a substantial part of the types of faults proposed in [8].

To the best of our knowledge, the first practical approach to inject software faults was proposed in [11]. The approach is based on a technique named Generic Software Fault Injection Technique (G-SWFIT) and is supported by the findings from a field study on real software faults in a variety of programs [12]. In order to better emulate software faults by fault injection G-SWFIT uses an extension of Orthogonal Defect Classification (ODC) [7] and classifies faults according to the point of view of the program context in which they occur, and closely relates faults with programming language constructs. According to this idea, a software defect is one or more programming language constructs that are either missing, wrong or in excess. The resulting fault types describe the morphology of software faults according to the language constructs where they can realistically appear (which helps the definition of code changes that must be injected to emulate each fault type), and relates the faults with the surrounding program context (which greatly improves the selection of appropriates locations for fault injection).

Based on the field study presented in [12], and on field data from IBM's ODC project [8], the G-SWFIT software fault injection technique uses a library of fault emulation operators that represent the most common type of faults observed in the field. The code changes actually injected with G-SWFIT represent the code that would have been generated by the compiler if the intended software faults were in fact in the high level source code. Both the code pattern and the code changes were defined based on an extensive field data study in real software faults discovered in deployed software [12].

The G-SWFIT library of fault emulation operators was established for executable code. This means that G-SWFIT can be used in components even when we do not have the component source code (obviously, G-SWFIT can also be used at source code level, whenever the source code is available). The G-SWFIT operator library also considers the different aspects that have impact on the code structure and on the type of bugs it supports, such as different programming languages, compilers and compilers optimization options, which means that the technique is portable.

G-SWFIT is based on a two steps methodology. The fault locations are identified in the first step, resulting in the set of faults to be injected. This step occurs before the actual experimentation. The faults are actually injected in step two during the target execution in a very simple and low intrusive task, as each fault location have been previously identified in step one. The result of the scan process is a map of the target identifying the locations suitable for the emulation of specific fault types.

2.2. Injection of Interface Faults

The injection of interface errors attempts to emulate the consequences of errors that are propagated from faulty components (as mentioned, in robustness testing contexts this emulation requirement is often relaxed). The values injected are normally based on the semantics of the system/component interface and in the data type used in the interface. Typical values used are extreme values of the data type used in the interface or valid and invalid values according to the interface semantics (e.g. Ballista project [18]).

Interface faults can be injected directly at the interface between components to simulate the situation where a component fails and outputs corrupted information to the other components. The

fundamental assumption behind this methodology in component-based systems is that the parameter corruptions represent a plausible consequence of real residual software faults existing in the software component that calls the API. However there is no guarantee that the values being injected really represent real residual faults produced by the API call. Even if some of the values injected can be produced by an internal fault, there is no assurance that such internal fault is a representative one.

Works about interface fault injection used in the robustness testing context are relatively abundant. BALLISTA is well-known tool aimed at the automated robustness testing [18, 21], which submits the target interface with valid and invalid inputs and can be interpreted as emulating the behavior of an erroneous module calling the target API.

The DTS tool [29] is a robustness testing tool that injects faults in the parameters of the calls to Windows NT system libraries. Faults are injected during actual calls performed by real programs during run-time. The RIDDLE tool [15, 16] is a robustness benchmark which uses a combination of random input, malicious input and boundary values to test the robustness of black-box components. MAFALDA [14] is a tool aimed at the evaluation of microkernel behavior in the presence of faults that uses system-call parameter corruption specifically aimed at the emulation of the effects of residual faults existing at the application level. A robustness benchmark tool based on the corruption of the parameters to the operating systems calls is presented in [10]. This tool uses a tailored workload to specifically call the OS services using invalid parameters. The injected values were defined based on the semantic of the target OS call.

Xception is a set of tools aimed at the experimental dependability evaluation [5]. Xception can inject interface faults for several processor architectures and it includes a set of modules to assist the execution of fault injection campaigns and information collection. The fault model supports the traditional robustness testing concept and includes a broad range of data types.

The Jaca tool [23] is used to inject interface faults in object-oriented systems written in Java programming language. Jaca uses reflective mechanisms of the Java language to inject the faults without changing the target system structure and is able to monitor the target system to examine the data flow across components without actual injecting faults. Jaca uses the Javassist reflection toolkit [6] to perform instrumentation at the bytecode level and does not require the source code of the target system. Current version of Jaca can operate with the public interface of classes by altering values of attributes, method parameters and return values.

3. Experimental setup and methodology

To understand the equivalence (or differences) between faults injected in the component code and faults injected at the component interface, we injected faults according to both methodologies and compared the impact of the injected faults in the target system (characterized through the well-known notion of failure mode). From a statistical point of view, faults injected at interface-level should have similar consequences as the faults injected in the component code.

In addition to the fault impact on the system, we also analyze the values that are passed in the interface calls after the injection of an internal fault with the values used when injecting faults at the interface. This analysis is aimed at the understanding of the relation between the faults injected through the interface and realistic faults existing in the preceding components.

We used two different experimental setups: the first is an object oriented database application written in Java (Ozone) and the second is a real-time application written in C. Each setup involves the use of component-based software system so that there is a clear boundary between modules (this a necessary requirement for the injection of the interface faults, as well as for the delimitation of the code where faults are injected). Both experimental setups were subjected to both types of fault injection.

We used G-SWFIT to inject faults in the component code. Jaca and Xception were used to inject faults at the interface. This selection was guided by the fact that these tools/techniques were developed in the context of previous works from our group and are well understood technologies.

We describe the experimental setups in the following sub-sections.

3.1. Ozone/OO7 experimental setup

This setup is composed by the object-oriented database system Ozone 1.1 and the Wisconsin OO7 benchmark acting as the database workload. The OO7 benchmark is an object-oriented database application inspired in the notion of a design library composed by a set of Computer Aid Software Engineering (CASE) documents organized hierarchically. This setup runs on top of the 1.5 Java runtime environment. As a benchmark, Wisconsin OO7 can be considered as a representative

application for object-oriented systems and implements the main functionalities expected from such kind of applications [4], and can be applied on any object-oriented database management system.

This system is perceived as a software system composed of two large components which are the Ozone and the OO7. In this scenario the target component for the injection of faults in the code (i.e., internal faults) is the OO7. Faults were injected directly into its OO7 classes using G-SWFIT. Interface faults were injected at the interface of the Ozone classes whose public methods are invoked by the OO7 code. The injection of interface faults was carried out using the Jaca tool (see Figure 2). The observation of the behavior of the OO7 and the Ozone, as well as the database state, allows us to compare the effects of each type of faults and understand the similarity of interface faults regarding the faults existing in the code of the components.

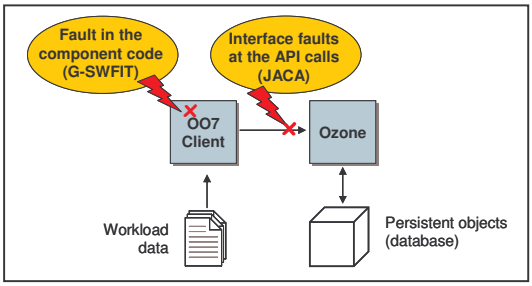


Figure 2: Ozone/OO7 Setup Overview.

3.2. Data-Handling/RTEMS experimental setup

This setup represents a real-time space application based on real-mission project requirements from the European Space Agency (ESA). This setup consists of a Command and Data Management System (CDMS) responsible for the management of the on-board systems and communication between the spacecraft and the ground control. It was implemented in C and runs on top of the Real Time Executive for Multiprocessor Systems (RTEMS) operating system. It is composed by the following components/subsystems:

- Packet Router (PR): This subsystem is composed by two distinct components:
  - Telecommand Manager (TC): responsible for the reception of ground commands and redirecting to the correct sub-system.
  - Telemetry Manager (TM): responsible for selectively sending telemetry to ground.
- Power Conditioning System (PCS): responsible for management of the power sources and the power circulation in the spacecraft.
- On Board Storage (OBS): responsible for storing telemetry to be sent to ground.
- Data Handling System (DHS): responsible for managing all data transactions between ground systems and spacecraft.
- Reconfiguration Manager (RM): responsible for recovering the spacecraft from failures.
- Payload (PL): responsible for performing science related activities, in this case, controlling the telescope and acquiring images.
- Hardware Abstraction Layer: responsible for handling low level communications, namely, controlling the communications devices;
- Hardware Communication Module: responsible for passing the command and telemetry to and from the hardware abstraction layer;
- System Startup Module: responsible for system initialisation and operations start up.

The workload executed in this setup is a mission scenario where a simulated space telescope is being controlled. The system controls the telescope parameters (aperture, exposition time, etc.), collects data and sends it to ground system. All data involved in this scenario is predefined which allows deterministic experiments. Also, independent mission scenario runs will produce exactly the same result, which allows further determinism. The starting point of the workload is an acknowledgement command sent from the CDMS to the ground control. After that, the ground control sends a series of commands for the CDMS to adjust telescope settings and capture image. For each command sent the CDMS sends back telemetry information. The timing of the commands and the contents of the telemetry information are used to detect the system correctness/failure during the experiments.

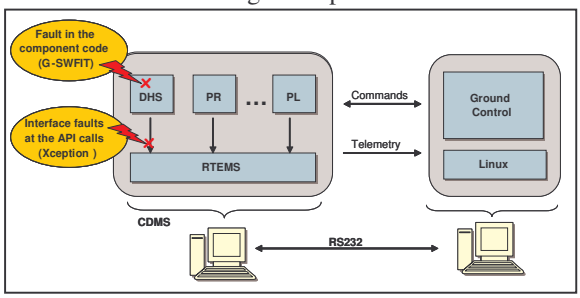


Figure 3: Data Handling/RTEMS Setup Overview.

As shown before, CDMS is divided in several components. The components used in the mission scenario were selected for the injection of fault in the code (G-SWFIT). In other words, in this setup we are

comparing the effects of faults in the selected components with the effects of faults injected at the interface of the components used by these selected components. All the explicit interaction of this component with remaining system is made via the operating system services. Thus, the target for interface faults can only be the OS.

Table 1 – Representativity of the fault types included in the faultload

Fault types	Description	Perc. Observed in field study	ODC classes
MIFS	Missing "If (*cond*) { statement(s) }"	9.96 %	Algorithm
MFC	Missing function call	8.64 %	Algorithm
MLAC	Missing "AND EXPR" in expression used as branch condition	7.89 %	Checking
MIA	Missing "if (*cond*)" surrounding statement(s)	4.32 %	Checking
MLPC	Missing small and localized part of the algorithm	3.19 %	Algorithm
MVAE	Missing variable assignment using an expression	3.00 %	Assignment
WLEC	Wrong logical expression used as branch condition	3.00 %	Checking
WVAV	Wrong value assigned to a value	2.44 %	Assignment
MVI	Missing variable initialization	2.25 %	Assignment
MVAV	Missing variable assignment using a value	2.25 %	Assignment
WAEP	Wrong arithmetic expression used in parameter of function call	2.25 %	Interface
WPFV	Wrong variable used in parameter of function call	1.50 %	Interface
	Total faults coverage	**50.69 %**	

The ground control software is hosted in a computer running Linux. CDMS runs on a X86 architecture based PC on top of the RTEMS kernel and is connected to the ground system via a RS232 link. The interface faults are injected with the Xception tool [5]. The faults in the CDMS code are injected with the aid of G-SWFIT. Figure 3 illustrates the key aspects of this setup.

4. Experimental results and discussion

An important aspect of the experiments is the definition of the set of faults to inject. We begin this section by presenting the details of the faults to inject in the target code (sub-section 4.1), and the details of the interface faults (sub-section 4.2).

4.1. Set of faults in the target component code.

The emulation of software faults in the target code requires the precise knowledge of which faults are to be injected: different faults types will require different target code modifications, and the identification of the appropriate locations for fault injection is dependent on the exact fault type (see [12] for more details). Moreover, we are specifically interested in the faults that are representative of actual residual faults discovered in deployed systems.

We selected the fault types to inject in our experiments based on information obtained on field data from previous works (see Table 1). These fault types represent the most frequent fault types observed in the field study [12] and cover about 50% of the 532 faults analyzed in the field study, belonging to four different ODC classes. The percentage shown in Table 1 (third column) is the best approximation for the distribution of these types of faults in the code and was observed in a field study.

The definition of the set of faults to inject is based on a simple algorithm: taking into account the fault types presented in Table 1, we analyze the target code and identify all locations were a given fault can be realistically emulated (by realistically emulated we mean that the intended fault could indeed be present at the original source-code construct relative to that location in the target executable code).

We identified 232 faults for the CDMS/RTEMS setup (recall that the targets for the injection of fault in the code are the components used by the mission scenario, not all components in the system). Concerning the Ozone/007, we observed that the interface between the OO7 and the Ozone is done primarily through the OO7 class BenchmarkImpl. In this case, the code of all the methods of this class provided 77 faults.

Each fault is injected separately from the others. Additionally, each fault is present from the beginning of the experiment to its end. This is in accordance to the notion that a software fault is a permanent fault (i.e., it is not a transient fault). Thus, each fault implies a completely new experiment (involving the execution of the entire workload).

4.2. Set of interface faults

Concerning the injection of interface faults, the most relevant issues are the selection of the relevant interface (API), and the definition of the values to inject in the parameters of the selected interface API. Recall that the injection of interface faults consists in corrupting the parameter values of the API/methods.

We followed the typical methodology used in robustness testing, which recommends the injection of extreme values for the parameter data type (e.g. for an integer value, 4294967295), specific values typically associated to well defined meanings (e.g., NULL, 0, -1, etc.). We also injected some

intermediary values with increasingly large intervals (e.g., 10, 100, 1000, etc.). Normally extreme values of the data type are used to verify the efficiency of exception handling mechanisms and valid/invalid inputs are used to corrupt data going from one component to its successor component in order to simulate failures of the predecessor component [31]. We choose these values to observe their impact (as the use of all possible values is impracticable) to verify if the same experiments used in robustness testing can be useful for general approach experiments.

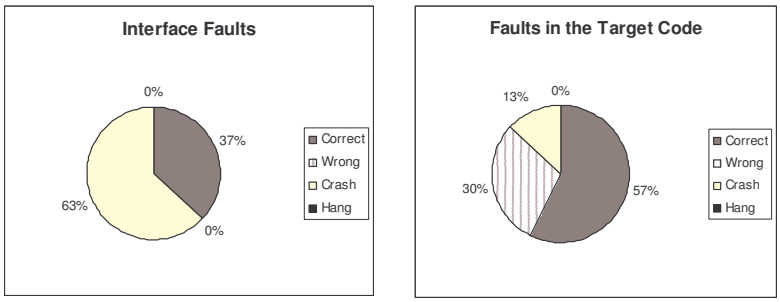


Figure 4: Interface Fault Injection Results

The injection of the interface fault is a straightforward process: a list of API/methods and the corresponding parameters and values to be injected, in accordance with each data type, is provided to the fault injector (Jaca or Xception, depending on the experimental setup) and most of the process is automatic.

The injection of each interface fault corresponds to the execution of a complete experiment involving the execution of the entire workload. In other words: during one experiment only one value is used in one parameter on one API/method. Note that the fault injection can occur many times during one experiment (one for each time the API/method is invoked).

Regarding the CDMS/RTEMS setup, we defined a total of 3384 different faults. Concerning the Ozone/OO7 setup, we observed that the Ozone classes used in the interaction between Ozone and OO7 are only the RemoteDatabase class. The methods of this class that are public and actually implemented in the class (and not in a base class) are only three (open, createObject, and objectForName). Thus, the number of interface faults is less than in the other setup, consisting in 46 faults only, which is not sufficient to provide fully statistically significant results. Nevertheless, we decided to include the experiments with the Ozone and OO7 setup, as they are useful as a first insight and represent a completely different scenario when compared to CDMS/RTEMS setup.

4.3. Experimental results

We defined four failure modes for the Ozone/OO7 setup: Hang (the OO7 benchmark does not terminate in the allotted time), Crash (the OO7 terminates abruptly before completing the assigned workload), Wrong (the workload terminates but the resulting information in the database is not correct when compared to the execution without faults), and Correct (there are no errors reported and the resulting information is correct).

Figure 4 shoes the results obtained in the Ozone/OO7 setup. We observed that the behavior of the system when subjected to faults in the target code differs substantially from the behavior when subjected to interface faults. Interface faults caused a greater number of failure occurrences and the failure mode pattern is completely different form the one observed with faults injected in the code of components. Considering that the faults injected in the target code are representative of residual software defects (according to our previous field data works), the results suggest that interface faults are not representative of realistic software defects in the target code (at least for this setup).

Given these results, we analyzed more closely the log files produced during the experiments to better understand how the errors are propagated from the target (OO7 class BenchmarkImpl) to Ozone. More precisely, we wanted to discover if there is a specific set of methods and parameter values related to the situations where the behavior of the system was not correct. If indeed there is an identifiable set of methods and parameter values, then this information is relevant to guide the robustness testing experiments.

We compared the log files that describe the sequence of all methods invoked by the OO7 when no faults were injected with the files produced when a fault was injected in the code (each injected fault has its own log file). This comparison allows us to identify any differences concerning different methods being called and different values being used in the parameters. We observed that the methods being called are the same, but at some point the values used in the parameters differ. These differences are related to a parameter of the data type reference, and the wrong value is not a NULL value or any other remarkable or identifiable value. Additionally, these differences tend to appear in two methods

(readExternal and writeExternal). This information allows the refinement of the interface faults experiments towards the improvement of injected interface faults in order to approximate as close as possible from software fault impact. In this case, faults should be centered in the reference data type.

We analyzed the contribution to the failure modes from each specific value used in the injection of interface faults in the Ozone/OO7 setup. These values range from -231 to 231. The result of this analysis is presented in Table 2 in the form of percentage. The results show that each value has approximately the same contribution to the overall failure mode distribution. This result is surprising as it suggests that the value injected at the interface is not relevant to the system behavior. Thus we can infer that the system behavior is more dependent on the internal system architecture and implementation than in the values injected in the interface, and therefore it seems that interface faults do not represent residual software faults.

Table 2 – Contribution of each value injected at the interface to the overall failure mode (Ozone/OO7 setup)

	-231	-100	-1	0	1	10	100	1000	231	String	Null	All
Correct	50	50	50	50	50	50	50	50	50	29	29	37
Wrong	0	0	0	0	0	0	0	0	0	0	0	0
Crash	50	50	50	50	50	50	50	50	50	71	71	63
Hang	0	0	0	0	0	0	0	0	0	0	0	0

The four failure modes for the CDMS/RTEMS setup are the same defined previously: Hang, Crash, Wrong, and Correct, with the same meaning as used before. The results obtained in the CDMS/RTEMS setup are those presented in the charts of Figure 5, where some of the injected module is compared (first row presents the results of the injection through the interface and second row the results of the injection into the classes).

As in the Ozone/OO7, the results suggest that faults injected at interface are not representative of faults in the component code even when we compare the results obtained for each class. The current Xception configuration did not allowed us to trace all the API calls as in the previous setup; therefore we cannot easily identify the set of API calls, parameters and values responsible for the error propagation when a fault is injected in the target component code.

Analyzing the contribution to the failure modes using the values that were injected in this setup for interface faults as well we obtained the results presented in Table 3 in the form of percentage. The results also show that each value has approximately the same contribution to the overall failure mode distribution (this finding is in accordance to the results of the previous setup and suggests that the values used in the injection at interface level are not correlated to the resulting failure modes).

Table 3 – Contribution to the overall failure mode from each value injected at the interface (CDMS/RTEMS setup)

	0	1	64	65	4K	4K+1	256K	256K+1	16M	16M+1	231	231+1	232-1	Null	All
Correct	43	46	50	50	34	33	35	36	32	33	42	43	41	27	39
Wrong	0	1	0	0	0	1	1	1	1	1	1	1	1	7	0
Crash	48	40	39	38	52	52	57	54	61	58	51	47	52	55	50
Hang	9	13	11	12	14	14	7	9	6	7	7	9	6	11	9

Although more experiments and more setups are

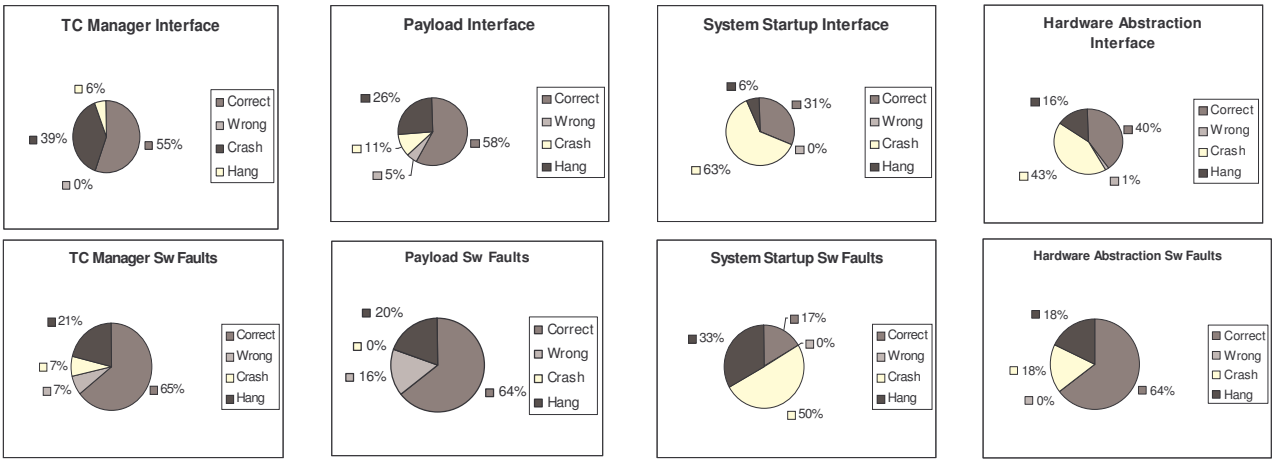


Figure 5 – Results obtained in the CDMS/RTEMS setup

required to fully validate this claim, these results support the notion that, at least for some classes of systems, interface faults do not represent well software faults in component code. This means that the injection of faults at component interfaces and at the component code level should be regarded as complementary techniques, and one technique cannot be replaced by the other.

In practice, we would like to use interface faults instead of the more complex injection of faults in the code of the component, but the results obtained in our experiments do not recommend that.

5. Conclusion

This paper presents an experimental assessment of the equivalence of faults injected in the interface of components and faults injected in the code of the components. The goal was to compare the results obtained with the two fault injection techniques in order to draw a possible equivalence between internal component faults and interface faults. As component interface faults are (in general) much easier to inject than faults in the component code (as interface fault injection uses the regular component interfaces), the establishment of a possible equivalence between the two types of faults would be very useful to simplify and speedup experiments (by replacing the complex injection of faults in the component code by interface faults) or to devise coordinated ways to use both injection techniques.

The experiments were performed in two quite different setups in order to cover a good variety of scenarios. It includes a database benchmark running on top of an object-oriented database written in Java and a real-time space application written in C and running on top of a well known real-time operating system (the RTEMS). As fault injection tools we used the G-SWFIT for the injection of software faults in the code of components and Xception and Jaca for the injection of interface faults (in the C and Java environments, respectively).

The observed results suggest that interface faults do not represent well residual software faults, as the behavior of the system when interface faults are injected differs substantially from the behavior observed when faults are injected in the code of components. This means that we cannot replace the injection of faults in the code of components by the injection of faults at the component interfaces, which is normally much simpler. On the contrary, our results suggest that interface faults and faults injected in the component code are complementary techniques and one cannot replace the other.

Another interesting conclusion is that the values used in the injection of interface faults are not particularly relevant to determine the impact produced in the target system. On one hand, this conclusion also supports the non-equivalence between interface faults and faults in the code of the preceding component; on the other hand, this fact can be used to reduce the number of values used in the experiments using interface fault injection, as a big variety of erroneous input values does not seem to introduce visible benefits in the experiments.

Future work includes new experiments with more programs and the improvement of the tracing abilities of the tools in order to better identify the relationship between faults in the component code and error propagation-paths.

Acknowledges

The authors thank CAPES (Brazil), FAPESP (Brazil), GRICES (Portugal) and FCT (Portugal) to partially support this work.

6. References

[1] Arlat, J., Crouzet, Y. “Faultload Representativeness for Dependability Benchmarking”. Workshop on Dependability Benchmarking, DSN02, 2002.

[2] Bieman, J., Dreilinger, D., Lin, L. "Using Fault Injection to Increase Test Coverage". Proc of The 7th IEEE International Symposyum on Software Reliability Engineering, ISSRE'96, New York, NY, USA, 1996.

[3] Blough, D., Torii, T. "Fault-Injection-Based Testing of Fault-Tolerant Algorithms in Message Passing Parallel Computers". Proc. of The 27th IEEE Int.Fault Tolerant Computer Symposium, FCTS-27, Seattle, USA, 1997, pp. 258-267

[4] Carey, M. J. DeWitt, D. J. Naughton, J. F. “The OO7 Benchmark” http://www.columbia.edu/, 1994, accessed Feb/2006.

[5] Carreira, J., Madeira, H., Silva, J. "Xception: Software Fault Injection and Monitorintg in Processor Functional Units". IEEE Trans. on Software Engineering, vol. 24, 1998.

[6] Chiba, Shigeru. “Javassist – A Reflection-based ProgrammingWizard for Java”, proceedings of the ACM OOPSLA’98 Workshop onReflective Programming in C++ and Java, Oct. 1998.

[7] Chillarege, R. "Orthogonal Defect Classification". Handbook of Software Reliability Engineering, M. Lyu, Ed.: IEEE Computer Society Press, McGraw-Hill, Ch. 9, 1995.

[8] 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.

[9] Christmansson, J., Hiller, M., Rimén, M. “An

Experimental Comparison of Fault and Error Injection". Proc. of The 9th Int.Symposium on Software Reliability Engineering – ISSRE 98, pp. 369-378, 1998.

[10] Dingman, C., Marshall, J., Siewiorek, D. "Measuring Robustness of a Fault Tolerant Aerospace System". Proc. of The 25th IEEE International Symp. on Fault Tolerant Computing - FTCS'95, Passadena, pp. 522-527, CA, USA, 1995.

[11] Durães, J., Madeira, H. "Emulation of Software Faults by Educated Mutations at Machine-Code Level". Proc. of The Thirteenth International Symposium on Software Reliability Engineering – ISSRE'02, Annapolis, USA, 2002.

[12] Durães, J., Madeira, H. "Definition of Software Fault Emulation Operators: A Field Data Study". Proc. of The International Conference on Dependable Systems and Networks – DSN2003, pp. 105-114, San Francisco, USA, 2003.

[13] Fabre, J.-C., Salles, F., Moreno, M., Arlat, J. "Assessment of COTS Microkernels by Fault Injection". Proc.of The 7th IFIP Working Conference on Dependable Computing for Critical Applications- DCCA'99, pp. 25-44, San Jose, CA, USA, 1999.

[14] Fabre, J. C., Rodríguez, M., Arlat, J., Salles, F., Sizun, J. "Bulding Dependable COTS Microkernel-based Systems using MAFALDA". Proc. of the 2000 Pacific Rim International Symposium on Dependable Computing - PRDC'00, pp. 85-92, 2000.

[15] Ghosh, A., Schmid, M., Shah, V. "Testing the Robustness of Windows NT Software". Proc. of the 9th IEEE International Symposium on Software Reliability Engineering - ISSRE'98, pp. 231-236, 1998.

[16] Ghosh, A., Shah, V., Schmid, M. "An Approach for Analyzing the Robustness of Windows NT Software".Proc.of The 10th IEEE International Symposium on Software Reliability Engineering - ISSRE'99, 1999.

[17] Hiller, M., Jhumka A., Suri , N. "An Approach for Analysing the Propagation of Data Errors in Software". Int. Conf. on Dependable Systems and Networks, DSN, Gothenburg, Sweden, 2001.

[18] Koopman, P., Sung, J., Dingman, C., Siewiorek, D. Marz, T. "Comparing Operating Systems Using Robustness Benchmarks".Proc.of The 16th International Symposium on Reliable Distributed Systems - SRDS'97, Durham, NC, USA, pp. 72-79, 1997.

[19] Koopman P., DeVale, J. "The Exception Handling Effectiveness of POSIX Operating Systems" IEEE Transactions on Software Engineering, vol. 26, pp. 837-848, 2000.

[20] Koopman, P. "What's Wrong With Fault Injection As A Benchmarking Tool?", in Proc. of The Internat. Conf. on Dependable Systems and Networks – DSN2002, Washington D.C, USA, 2002.

[21] Kropp, N., Koopman, P. & Siewiorek, D., "Automated Robustness Testing of Off-the-Shelf Software Components," 28th Fault Tolerant Computing Symposium, pp. 230-239, 1998.

[22] 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).

[23] Martins, E.; Rubira, C. M. F.; Leme N.G.M. "Jaca: A reflective fault injection tool based on patterns" Proc of the 2002 Intern Conference on Dependable Systems & Networks, pp. 483-487, Washington D.C. USA, 23-267, 2002.

[24] Moraes, R; Martins, E "A Strategy for Validating an ODBMSComponent Using a High-Level Software Fault Injection Tool", proc.of the First Latin-American Symp, pp. 56-68, SP, Brazil, 2003.

[25] Moraes, R. and Martins, E. "An Architecture-based Strategy for Interface Fault Injection", Workshop on Architecting Dependable Systems, IEEE/IFIP International Conf. on Dependable Systems and Networks, Florence, Italy, June 28 – July 1, 2004.

[26] Mukherjee A.,Siewiorek, D. "Measuring Software Dependability by Robustness Benchmarking," IEEE Transactions on Software Engineering, vol. 23, 1997, pp. 366-378.

[27] 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.

[28] Ng W., Chen, P. "Systematic improvement of fault tolerance in the RIO file cache". Proc. of The 30th IEEE Fault Tolerant Computing Symp., FTCS-29, Madison, WI, USA, 1999.

[29] T. Tsai and N. Singh, "Reliability Testing of Applications on Windows NT", in Proceedings of the IEEE International Symposium on Dependable Systems and Networks - DSN'00, New York, NY, USA, pp. 427-436, 2000.

[30] J. Voas, F. Charron, G. McGraw, K. Miller, and M. Friedman, "Predicting How Badly 'Good' Software can Behave", IEEE Software, 1997.

[31] J. Voas, "A Defensive Approach to Certifying COTS Software", Reliable Software Technologies, (1997).

[32] Weyuker, E.J. "Testing Component-Based Software: A Cautionary Tale". IEEE Software, pp. 54-

Capítulo 13 – Validação do Modelo Estatístico

O artigo reproduzido neste capítulo foi publicado no *Workshop on Empirical Evaluation of Dependability and Security* que aconteceu em conjunto com *The International Conference on Dependable Systems and Networks*-DSN´06. O artigo tinha como objetivo a validação do modelo estatístico proposto para estimar a probabilidade de haver uma falha em um componente de software, bem como, propor uma representatividade para a distribuição das falhas a serem injetadas durante uma campanha de injeção de falhas. Para estes propósitos foi utilizado um modelo estatístico baseado na regressão logística. Nesse modelo, as diversas métricas foram combinadas para se fazer a previsão das falhas residuais que permanecem no software após este entrar na fase operacional.

Com o objetivo de validar o modelo estatístico proposto, fizemos um estudo de campo com diversos produtos de software *open source*, analisando 350 bugs reports. Nesse estudo, levantamos os *bugs* apontados por usuários durante a utilização dos produtos de software e comparamos com a estimativa obtida. Depois de validado, utilizamos a estimativa para propor a distribuição de falhas para os experimentos que utilizam a técnica de injeção de falhas. A representatividade da distribuição de falhas nestes experimentos é importante para simular mais realisticamente o comportamento em campo quando o objetivo é obter uma avaliação do risco. Um resumo do artigo já foi apresentado na Subseção 4.2.1 desta tese.

A referência deste artigo é a que segue: ***Moraes, R.,*** *Durães, J., Martins, E., Madeira, H.* "*A field data study on the use of software metrics to define representative fault distribution*". *In: Proceedings of The International Conference on Dependable Systems and Networks – DSN 06, 2006.*

A field data study on the use of software metrics to define representative fault distribution

R. Moraes1, J. Durães3, E. Martins2, H. Madeira3
[1]CESET, [2]IC, State University of Campinas, UNICAMP, São Paulo, Brazil
[3]CISUC, University of Coimbra, Portugal
regina@ceset.unicamp.br, jduraes@isec.pt, eliane@ic.unicamp.br, henrique@dei.uc.pt

Abstract: *This paper evaluates the use of software complexity metrics to define representative fault distributions for software fault injection experiments. A field data study on more than 350 bug reports available from open software initiatives was used to compare the fault distributions generated by our approach with the real fault distributions observed in the field. Results show that the way we distribute software faults for fault injection is consistent with field observations when the size and complexity of the software modules are not very high. For very large modules the fault density observed are lower than the estimated by our approach. Possible explanations and improvements are discussed.*

1. Introduction

Empirical evaluation approaches often relies on fault injection. This technique has been extensively used to evaluate fault tolerance mechanisms and to assess the impact of faults in systems. Software faults are currently recognized as the most relevant type of faults, as software is becoming extremely complex and most residual faults in deployed systems are effectively software faults. The emulation of software faults by fault injection has been addressed in the last decade [1, 2] leading to practical techniques that emulate realistic software faults with acceptable accuracy [3].

A key notion in fault injection is the representativeness of fault distributions. Random fault distributions based on the size of physical devices have been used for the injection of hardware transient faults with good acceptance. However, the injection of software faults requires much more elaborated fault distributions. Even though, the very few studies on the representativeness of software fault injection [1, 3] just addressed the problem of finding realistic fault models, and ignored how software faults should be distributed among the different software components in the target system. Simple fault distribution models such as exhaustive fault coverage for small software modules (all possible faults are injected in the module) or uniform sampling for the large ones are the approaches used so far [4, 5]. In [6] a complexity metric was used to distribute interface faults among components, but that approach is different from the one used in this paper and was never validated with field data.

We propose a methodology to define representative distributions of software faults using software complexity metrics to estimate fault densities of each modules of the target software and distribute the injected faults accordingly. Orthogonal Defect Classification (ODC) [7] and field data on how real faults map to ODC classes are used to guide the distribution within each module.

As the proposed methodology is based on the estimation of fault densities, its validation requires the assessment of the fault density accuracy. We evaluate this crucial aspect by comparing the fault distributions generated using our approach with the real fault distributions observed in the field. The field data includes more than 350 bug reports available from open software projects. The results show good estimation accuracy for small and medium size software modules.

The remainder of this paper is organized as follows: Section 2 presents the proposed methodology. Section 3 presents the procedure to evaluate the approach using field data and discuss the results. Section 4 concludes the paper and outlines future work.

2. Representative software fault distributions

Representativeness of fault distributions is one of the most important issues in fault injection. However, it has been largely neglected in the literature, especially in what concerns to the injection of software faults. Our proposed methodology to define fault distributions includes the following steps:

1. estimate components fault densities using software complexity metrics;
2. define the total number of faults to inject in the campaign;
3. assign a first target number of faults per software component considering component fault densities;
4. scan the code to calculate the maximum number of possible injection locations per component;
5. determine the final number of faults to inject in each component and distribute faults in each component

according to the most frequent fault types found in field [3, 1].

The next sections detail the most relevant steps of this methodology.

2.1. Estimate component fault density

The probability of a residual software faults in a component has a dichotomous nature: the component either has residual faults or it has not. Due to this dichotomous nature we used logistic regression [8] to estimate the fault density based on complexity metrics. Logistic regression is widely used for analysing multivariate data involving binary responses [9].

The multivariate prediction model analyzes how well we can predict the fault-proneness of modules when software metrics are combined. We used seven software metrics in this work: Lines of Code (LoC without comments lines), Cyclomatic Complexity (measures the control flow complexity of a module and is dependent on the number of predicates [10]), Function Parameters (number of parameters), Function Returns (number of returns statements inside the module), Maximum Nesting Depth (measures the maximum indentation depth, e.g., in C language measures how deep is the maximum { } nesting in the module), and two Halstead metrics [11]: Program Length (sum of the total number of operators and operands in the module), and Vocabulary Size (sum of the unique operators and operands in the module).
A starting point for the logistic regression is obtained by adopting fault densities ranges accepted by industry. In our work we used the values reported by Rome Laboratory [12] that analysed fault density empirically and concluded that this value ranges from 0.1 to 1 fault per thousand lines of code (KLoC). Considering that the existence of a fault is independent of the existence of other faults in the remaining lines and using the LoC metric in each module i and standard fault density p of 0.1 faults per KLoC, we get the number of lines with residual faults within the module i as a binomial random variable with parameters LoCi and p that defines a preliminary fault density for module i. This preliminary density is then adjusted using the set of other metrics (without considering LoC metric again) that had been evaluated for module i using appropriated tools [13, 14].

The regression coefficients are then calculated through regression analysis available in statistical tools and were used to estimate the fault density probability p(X1,X2,…,Xn) for each module through the following equation:

$$p(X_1, X_2, .., X_n) = \frac{\exp(\beta_0 + \beta_1 X_1 + \beta_2 X_2 + ... + \beta_n X_n)}{1 + \exp(\beta_0 + \beta_1 X_1 + \beta_2 X_2 + \beta_n X_n)}$$

where βi are the coefficients obtained through regression analysis, Xi are the software metrics used (e.g., Cyclomatic Complexity, Function Parameters, and so on) and exp is the inverse function of ln. The larger the absolute value of the coefficient is, the stronger the impact of the variable on the probability p.

We used statistical significance p-value to verify the accuracy level of the coefficient estimation. Historically, common p-value thresholds are 0.01, 0.05 or 0.1 [15]. The impact of the variable is more believable when the p-value is lower. For this work we considered 0.1 as a cut-off value, e.g., the corresponding metric is considered as an independent variable in the regression analysis if its p-value is less than 0.1.

2.2. Assign a target number of faults per module

According to the estimated fault density, the preliminary distribution should be performed using the following equation:

$$numfault = \frac{p(X_1, X_2, ...X_n) * Tfault}{\sum p(X_1, X_2, ..., X_n)}$$

where Tfault is the total number of fault to be injected in a campaign and p(X1,X2,…,Xn) is the estimated fault density of a specific module.

2.3. Scan the code to calculate the maximum number of faults per module.

The number of software faults that can be injected in a component is limited by the internal structure of the target component. We use the G-SWFIT technique [3] to scan the target code and identify the maximum number of faults of each type in each given module (variable *Tscan*). G-SWFIT is based on the mapping between high-level constructs and its translation to low-level code. This technique can emulate software faults by changing the target executable code to reproduce the instruction sequences that would had been generated by the compiler if a software fault of a given type were present in the high-level source code.

The output of the code scanning using G-SWFIT is the set of all possible locations for the injection of each type of software fault (see Table 1 for the type of faults considered). It is worth noting that each fault location identified is associated to exactly one type of fault, and each location implicitly identifies the fault type for that location. Each fault location (and associated fault type) has one or more associated code changes, and each code change represents a single fault to be injected [3].

G-SWFIT may seem similar to traditional mutation techniques used in mutation testing. However, G-SWFIT has some important differences: it can inject faults directly at the executable code without requiring the target source code, and the type of faults injected are those that were found in previous studies as being representative of residual software faults in deployed software.

For the work at hand we are interested in injecting realistic fault distributions. Thus, *Tscan* is the number of fault locations for all the representative fault types. We

consider only fault types that were recognized in previous works as representative of residual faults found in deployed software [3, 1]. To better clarify this point, we reproduce in Table 1 examples of fault types that were identified as being representative in [3].

Table 1- Examples of representative fault types.

Missing	variable initialization (MVIV)
	variable assignment using a value (MVAV)
	variable assignment using an expression (MVAE)
	"if (*cond*)" surrounding statement(s) (MIA)
	"AND EXPR" in expression used as branch condition (MLAC)
	function call (MFC)
	"If (*cond*) { statement(s) }" (MIFS)
	"if (*cond*) statement(s) else" before statement(s) (MIEB)
	small and localized part of the algorithm (MLPC)
	functionality (MFCT)
Wrong	value assigned to variable (WVAV)
	logical expression used as branch condition (WLEC)
	arithmetic expression in param. of func. call (WAEP)
	variable used in parameter of function call (WPFV)
	algorithm - large modifications (WALL)
	data types or conversion used (WSUT)
Extra.	variable assignment using another variable (EVAV)

2.4. Determine the number of faults to inject

If Tscan is lower than the number of faults assigned to the target module (numfault) then all possible fault locations identified in the previous step are effectively used for fault injection.(i.e., numfinj = Tscan), which means a complete fault injection coverage.

If Tscan is greater than numfault, then the faults actually injected are selected based on the ODC defect-type to which they belong to. Previous field data works (e.g., [3, 1]) suggest that residual faults in deployed systems tend to follow a specific distribution (see Table 2).Thus, we adjust the number of fault injected in each fault type according the relative weight of each ODC defect type shown in Table 2. As the two distributions are similar but not exactly the same, we can use either one, or even the average value observed for each ODC type.

3. Evaluation using field data on software faults

We analysed bugs reports available from two open-source applications at equivalent level of maturity. VIM, version 6.0, is a text editor composed by 88 C modules. Joe, version 2.9.8, is a text editor as well, composed by 80 C modules. According to VIM and Joe website forums, the estimated number of users is equivalent and both editors are used in many Linux distributions. Figures 1 and 2 present the bug frequency distributions for VIM and Joe, considering all the bug reports available.

Both charts show that the distribution of bugs along the time is practically uniform, with some occasional outliers. This result suggests that the bug frequency tends to remain nearly constant across successive versions. This means that bugs persist throughout the time and bug correction is a continuous activity. It also means that the 305 bugs found in VIM and the 52 bugs found in Joe do not represent the total number of bugs in the programs (we just know the programs had at least those bugs, but most probably there are other hidden bugs). Thus, the observed fault density is lower than the real fault density (which is actually unknown). This is quite relevant when comparing the estimated fault density with the observed one. If the estimated fault density is lower than the observed density, we can conclude that the estimation is wrong. Otherwise nothing can be concluded for sure.

Table 2 - Fault distribution across ODC types.

ODC Type	Fault Distribution [3]	Fault Distribution [1]
Assignment	22.1%	21.9%
Checking	25.7%	17.5%
Interface	8.0%	8.2%
Algorithm	37.2%	43.4%
Function	6.7%	8.7%

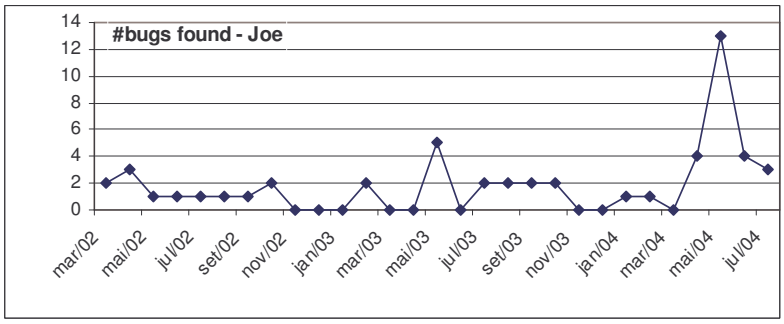


Figure 1: Reported Bugs for the VIM Software

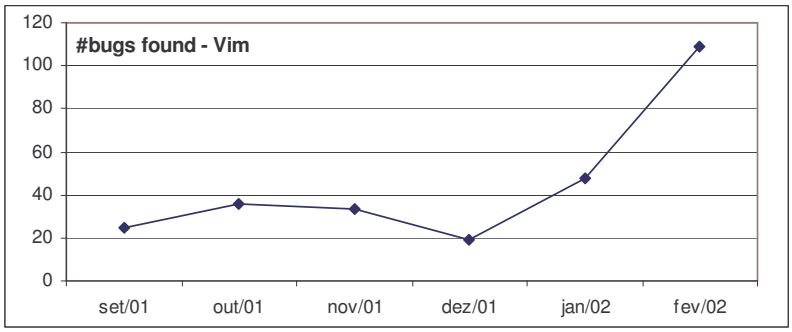


Figure 2: Reported Bugs for the Joe Software

Another interesting point is the analysis of bug distribution considering component complexity. Figure 3 shows that there is a growing trend when the complexity increases. This correlation of software complexity with the number of bugs found is consistent with some previous works (e.g., [16, 17]) and provides empirical evidence that can be used to predict component fault density. However, Fenton [18] shows that this trend does not hold in some cases.

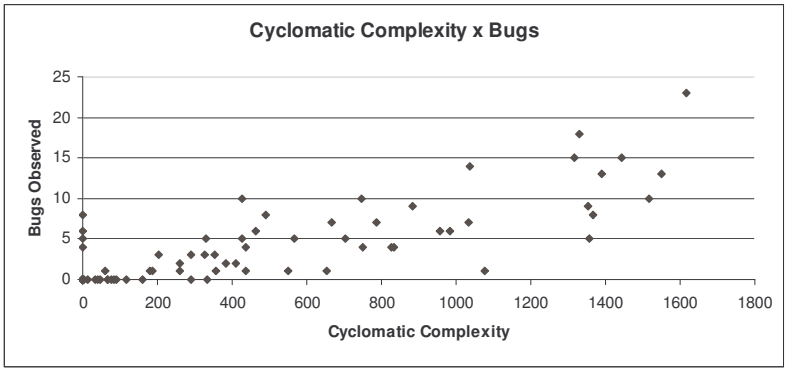


Figure 3: Distribution of Bugs Observed for VIM software

Some explanations for this apparent contradiction are provided in [19]. They support the use of static code measures as predictors if these predictors are treated as probabilistic and not categorical indicators. They also reinforce the importance of finding good attributes set for each problem and to analyse a large amount of data sets in order to generalize the results.

The correlation between each metric and the number of bugs observed is consistent for all the used metrics and for both programs. As all charts are similar, we just show the chart for cyclomatic complexity (Figure 3).

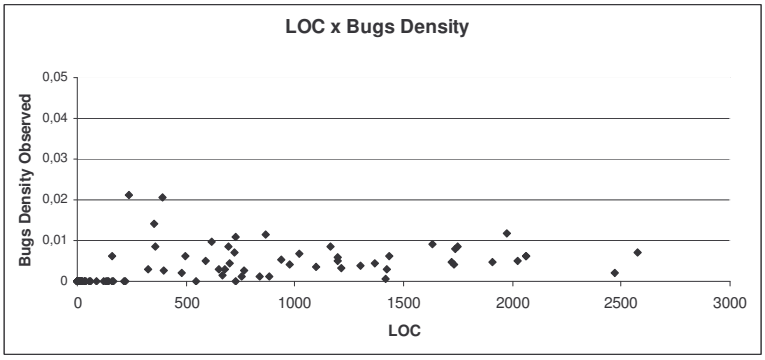


Figure 4: Distribution of Bugs Density

Another interesting aspect is to observe how bug density correlates to complexity metrics. All the metrics used in our experiments suggest that bug density is practically constant, no matter the size and complexity of the components. Figure 4 shows the distribution for the metric lines of code (for other metrics is similar).

This observation validates the use of the standard fault density values [12] as a first estimation (refer to section 2.1 where we referenced the value 0.1 fault per KLoC as standard for fault density). It also confirms the notion of the independence of the faults occurrences from one module to the others.

As we use lines of code to compose the first estimation of fault density, we do not consider this metric in the subsequent multivariate analysis to estimate component fault densities. The statistical values obtained in multivariate analyze are presented in Table 2.

The coefficient of Program Length metric for VIM is the lowest one among metrics used, indicating the minor impact that this metric has on $p(X_1,X_{2,...},X_6)$. The Function Returns *p-values* obtained for VIM is higher than 0.1, which points to discard this metric as it does not present a good level of significance.

Table 2: Statistical Values for Multivariate Analyze

Metrics	VIM		Joe	
	Coefficient	p-value	Coefficient	p-value
Intercept	-7.058395	1.69*E-41	-7,157724	2,94 E-56
C. Complexity	0.003236	0.016997	0,002766	0,5363
F. Parameters	0.012136	0.007802	0,010949	0,4230
F. Returns	-0.004209	0.330229	0,003109	0,8032
Progr. Lenght	-0.000207	0.041095	-0,000235	0,3414
Vocab. Size	0.002643	0.002693	0,008330	0,00017
MaxNestDepth	0.379924	9.81*E-06	0,243237	0,00016

Figure 5 presents the estimated fault density and bugs density in a chart where the modules are ordered by lines of code. The outliers for small modules represent libraries modules (.h files) with few instructions. Nevertheless, these modules receive maintenance as a consequence of faults in other modules that use them.

The fault density estimation is acceptable for small or medium size modules but presents a poor approximation for large ones.

As we have outliers for the libraries files in VIM software, the simulation in Joe does not include .h files. For large modules we obtained very high values as the estimated fault density is placed out of the range of the chart in Figure 6 that presents this result.

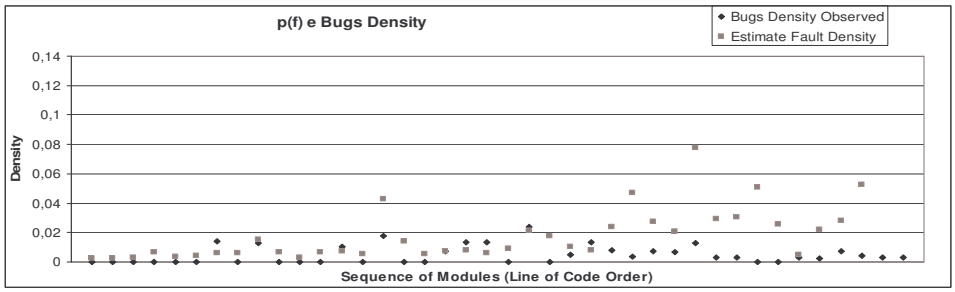


Figure 5: Faults and Bugs Density – VIM

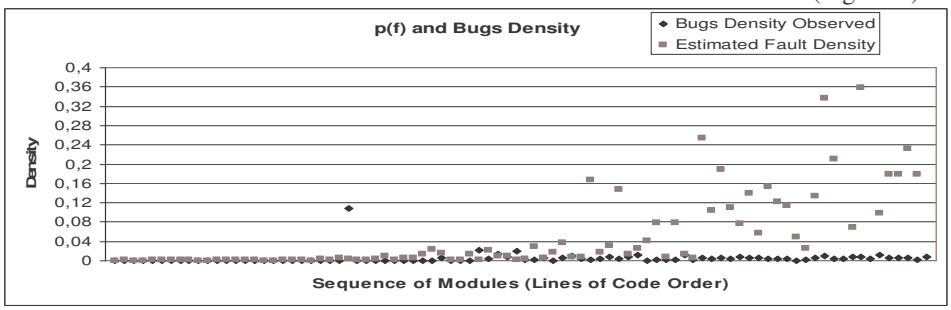


Figure 6: Faults and Bugs Density - Joe

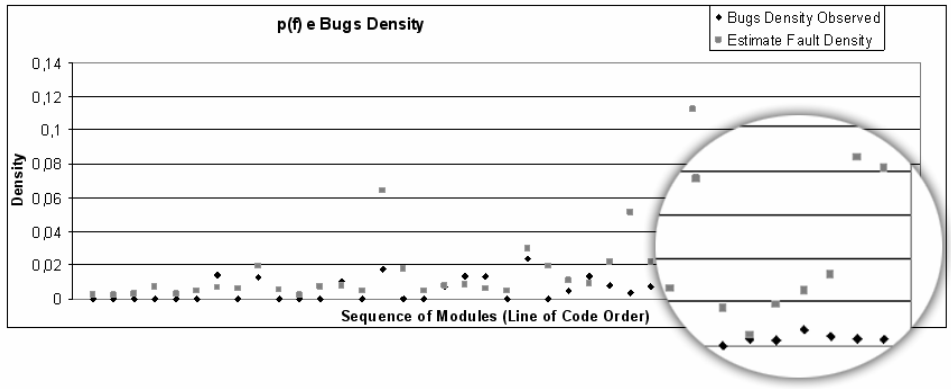


Figure 7: Faults and Bugs Density with two metrics – Joe

Table 3: Distribution for Joe Software (fragment)

Module	$p(X_1,X_2,X_3)$	*numfinj*
menu.c	0,001714	73
hash.c	0,001814	78
uerror.c	0,002242	96
uedit.c	0,002895	124
usearch.c	0,002934	125
main.c	0,003196	137
ublock.c	0,003284	140
termcap.c	0,003648	156
macro.c	0,004029	172
uformat.c	0,004212	180
regex.c	0,004254	182
w.c	0,005102	218
bw.c	0,005307	227
rc.c	0,006133	262
….	….	…
Total *#fault*		10000

As we can observe in Table 2, there are metrics for Joe that presents high *p-value* (greater than 0.1). We eliminated one by one of this metric and performed new simulations with the other metrics (Figure 7). The new results show a significant estimation improvement for large modules.

Table 3 presents a sample of the fault distribution generated using our approach when the total number of faults to inject is set to 10000 faults (just a small number of modules is shown). This illustrates the way the proposed methodology to generate fault distribution for fault injection is used in practice. In the example shown in Table 3, component uerror.c will receive 96 faults, component ublock.c 140, and so on and so forth.

4. Conclusions

In this paper we propose a methodology to define representative fault distributions for software fault injection. The approach is based on the estimation of fault densities of components using software metrics, and defines the number of faults that should be injected in each software component of the system under evaluation.

The representativeness of the fault distributions obtained with the proposed approach is tightly connected to the accuracy of the fault density estimation. To validate this important point, we analyzed more than 350 bug reports available from open software initiatives. The analysis of this field data allow us to compare the fault distributions generated by our approach with the real fault distributions observed in the field.

We observed that the fault distributions generated by our approach are consistent with real faults for low and medium size components. The fault density estimated for large and complex modules deviate from the one

observed in our field data. After the analysis of *p-value,* we concluded that the elimination of low significance metrics lead to the improvement of the fault density estimation. Still, the estimation of fault density in large components requires further improvement.

Future work includes the use of other metrics to improve the accuracy for large modules. Software architecture metrics should also be considered as well as operational usage.

Acknowledgment
The authors thank to CAPES/GRICES and COMPGOV project to partially support this work. We thank also to MSquared Technologies for gracefully providing the full version of RSM tool, and Testwell Oy Ltd for CMT++ and CMTjava tools.

References

[1] Christmansson, J., Chillarege, R., "Generation of an Error Set that Emulates Software Faults-Based on Fields Data", Proc. of 26th Int. Symp. on Fault-Tolerant Computing, pp 304-13, Sendai, Japan, 1996.

[2] Madeira, H., Vieira, M., Costa, D., "On the Emulation of Software Faults by Software Fault Injection". Proc. of The Int. Conf. on Dependable Systems and Networks, NY, USA, 2000.

[3] Durães, J., Madeira, H., "Definition of Software Fault Emulation Operators: A Field Data Study". Proc. of The Int. Conf. on Dependable Systems and Networks–DSN2003,pp.105-114,S.Francisco, USA, 2003.

[4] Durães, J., Madeira, H., "Dependable Systems and Networks". Proc. of The Int. Conf. on Dependable Systems and Networks, pp. 285-294, Florence, Italy, 2004.

[5] Voas, J., G. McGraw; L. Kassab; L.Voas. A., 'Crystal Ball' for Software Liability. IEEE Computer, pp 29-36, 1997.

[6] Moraes, R., Martins, E., Poleti, E., Mendes, N., "Using Stratified Sampling for Fault Injection", Proc. of the First Latin-American Symp., LADC 2005, pp. 09-19, São Paulo, Brasil, 2005.

[7] Chillarege, R., "Orthogonal Defect Classification", Ch. 9 of "Handbook of Software Reliability Engineering", M. Lyu Ed., IEEE Computer Society Press, McGraw-Hill, 1995.

[8] Hosmer, D., Lemeshow, S., "Applied Logistic Regression". John Wiley & Sons, 1989.

[9] Dobson, A., "An Introduction to Generalized Linear Models", Ch. 8, Chapman & Hall, London, UK, pp. 104-122, 1990,

[10] Chidamber, R., Kemerer, F., "A Metric Suite for Object-Oriented Design", In IEEE Transaction of Software Engineering, v. 20 (6), 1994.

[11] Halstead M., "Elements of Software Science". Elsevier Science Inc., New York, NY, USA, 1977.

[12] Rome Laboratory (RL). "Methodology for Software Reliability Prediction and Assessment, Tec. Report RL-TR-92-52, v. 1-2, 1992.

[13] Resource Standard Metrics, Version 6.1, http://msquaredtechnologies.com/m2rsm/rsm.htm. Last access 2005.

[14] Testwell Oy Ltd, CMT++ Tool, www.testwell.fi/company.html, accessed March/2006.

[15] Basili V., Briand, L., Melo, W., "A Validation of Object-Oriented Design Metrics as Quality Indicators". IEEE Transaction on Software Engineering, v. 22 (10), pp. 751-761, 1996.

[16] Tang, M.-H., Kao, M.-H., Chen, M.-H., "An Empirical Study on Object-Oriented Metrics". Proc. of the Sixth Int. Software Metrics Symp., pp. 242-249, 1999.

[17] Rosenberg, L., Stapko, R., Gallo, A., "Risk-based Object Oriented Testing". Proc of. 13th Int. Software and Internet Quality Week-QW, San Francisco, California, USA, 2000.

[18] Fenton, N., Ohlsson, N., "Quantitative Análisis of Faults and Failures in a Complex Software System", IEEE Transactions on Software Engineering, 2000.

[19] Menzies, T., Greenwald, J., Frank A., "Learning Defect Predictors". Submitted to a Journal, http://menzies.us/, accessed February/2006.

Capítulo 14 – Avaliação Experimental do Risco

O risco da utilização de um componente para o sistema no qual este componente está integrado é calculado como sendo o produto da probabilidade de haver uma falha residual no componente e o custo que uma falha ativada causa no sistema onde o componente está integrado.

Para estimar a probabilidade de haver uma falha residual no componente utilizamos um modelo estatístico baseado na regressão logística. Para a avaliação do custo (ou impacto) das falhas residuais no componente utilizamos a técnica experimental, através da injeção de falhas internas.

A metodologia proposta para a avaliação experimental do risco foi aplicada em um estudo de caso que avaliou um gerenciador de base de dados orientado a objetos, o Ozone e dois sistemas operacionais largamente utilizados em ambientes computacionais, o RTEMS e o RTLinux. Através dos experimentos pudemos observar a influência dos aspectos estáticos e dinâmicos na avaliação do risco.

Embora a abordagem proposta ainda não possa fornecer uma avaliação real do risco ela se mostrou útil para que pudéssemos comparar componentes que provêem a mesma funcionalidade. Através desta comparação é possível que o usuário escolha entre componentes disponíveis para uma determinada funcionalidade aquele que apresente o menor risco. Um resumo do artigo já foi apresentado na Subseção 4.2.2 desta tese.

O artigo intitulado "*Experimental Risk Assessment using Software Fault Injection*" foi submetido para a conferência *The International Conference on Dependable Systems and Networks – DSN 07*. No momento se encontra em fase de avaliação.

Experimental Risk Assessment Using Software Fault Injection

R. Moraes[1], J. Durães[2], R. Barbosa[3], N. Mendes[2], M. Nogueira[2], E. Martins[1], H. Madeira[2]
[1]*State University of Campinas, UNICAMP, São Paulo, Brazil*
[2]*CISUC, University of Coimbra, Portugal*
[3]*Critical Software SA, Coimbra, Portugal*
regina@ceset.unicamp.br, jduraes@isec.pt, rbarbosa@criticalsoftware.com, naaliel@dei.uc.pt, memn@dei.uc.pt, eliane@ic.unicamp.br, henrique@dei.uc.pt

Abstract

Component-based system development is a current trend in modern systems. An important aspect of integrating components in a system is choosing a component that will best fit a specific proposal. The selection criterion should identify the less risky component in order to reach the expected system reliability. This is particularly relevant for software systems, as the development of this type of system is increasingly based on general purpose COTS components. Risk assessment has been previously evaluated based on heuristics and developer's experience. This paper proposes a new methodology for experimental risk assessment based on fault injection. The risk evaluation equation considers the specific aspects evaluated by fault injection, such as the probability of fault activation and fault impact, as well as software complexity metrics to estimate the probability of residual defects in software components. The injected faults emulate typical programming errors based on an extensive field data study previously published. The proposed approach is demonstrated and evaluated in two different systems representing realistic component-based applications developed in Java and C and using off-the-shelf components such as RTEMS and RTLinux real time operating system.

1. Introduction

A common practice for large scale software development today is to use available general-purpose components (often referred as COTS – commercial off-the-shelf, even when the components are not commercial) and develop from scratch only the domain-specific components. The general-purpose components and the components developed in-house are then assembled into the system using "glue" code. Given the high costs of designing and implementing software components, engineers see the reuse of general-purpose components as a way to reduce development effort and to achieve shorter time-to-market. However, in spite of the advantages of COTS, their utilization introduces new problems as well, as the new operational conditions may differ substantially from those the components were initially designed for. Even if the fact that the components have already been used in the field may bring a false feeling of safety, the reality has shown that new utilizations of heavily used components have exposed software faults that had not been disclosed before [42]. In practice, it is necessary to test the component in the new environment every time the component is reused to ensure high quality and reliability of software programs.

The complete elimination of software defects during software development process is very difficult to attain in practice, and this is also true for COTS components [26, 21]. Therefore, the current scenario in the computer industry is having systems in which software defects do exist but no one knows exactly where they are, when they will reveal themselves, and, above all, the possible consequences of the activation of the software faults. In a world where COTS are used more and more to build larger systems, the residual software faults represent a growing risk. Furthermore, the strong interaction of many COTS components (e.g., general purpose operating systems, database management systems, etc) with other parts of a larger system makes residual faults in COTS components particularly dangerous as they may cause serious risk of system outage.

In spite of these difficulties, component based software development with intensive reuse of components is a solid trend in the industry and is not likely to disappear, as the alternative would be the much more expensive write-from-scratch approach. Therefore, the software industry needs practical and effective methods to help estimate (and reduce) the risk of using COTS components or help in choosing the most reliable alternative when several alternative components are available.

Risk assessment approaches typically address risk management in software development projects [23, 16] and relate the risk estimation to quality

models of software development, heuristics, and developers' experience [4, 32, 36]. In highly demanding application areas, such as avionics or nuclear power plants, the risk estimation is tightly associated to safety and reliability assessment and is regulated by strict industrial standards [13]. Modeling approaches based on architectural-level risk analysis are also quite popular, especially when applied to the early development phases [35, 30, 43].

Despite the extensive research in risk assessment, the estimation of the risk associated to the use of a given component remains a hard problem. One factor that contributes to this difficulty is the fact that the dynamic behaviour of the component is difficult to assess without experimentation using the actual component or a prototype.

In this paper we propose a new method to estimate the risk of using off-the-shelf components in a larger software system based on an experimental approach. Our proposal is based on recent software fault injection techniques [9], combined with the use of well established software complexity metrics [21, 38]. We address the two classical terms of the risk equation (the probability of occurrence of an undesired event and the cost of resulting consequences) by using complexity metrics of the component under analysis to estimate the probability of residual software faults in that component, and by using software fault injection to evaluate the cost of possible component failures in the whole system (we use the term cost to refer to the impact of a component failure, as the term cost is generally used in risk works).

Our goal is to provide a quantitative answer to the following basic question: "What is the risk of using component C in system S considering that component C may have hidden faults?" We consider that system S is developed using the traditional component based software development approach and that component C is an off-the-shelf component or just a custom made component that is being reused in system S. The type of risk we want to evaluate is the probability of system S to experience a failure (e.g., to produce erroneous results, or to experience a safety failure, or a timing failure, or a security failure, or become unavailable, etc.) due to a faulty behavior in component C, which was caused by the activation of a residual software fault in this component.

The risk estimation based on an experimental approach, as proposed in this paper, is particularly very useful in the following scenarios:

- Identify software components that represent higher risk and require more testing effort or improvements.
- Help software designers and engineers to choose from alternative off-the-shelf components to be used in the system under development (i.e., choose the one that represents the lowest risk).
- Provide a quantitative evaluation of the risk reduction due to improvements introduced in components used in the system (COTS or not), such as wrapping the component to overcome robustness weakness [2]. It is worth noting that wrapping could also introduce new bugs and may change component behavior, which means that a thorough evaluation of the changed component is necessary (and it is not trivial).
- Help in tuning complex COTS components to minimize risk and increase the system dependability (e.g., a database management system is difficult to tune, especially in what concerns recovery features that have huge impact on whole system dependability [40]).

The use of fault injection to predict worst case scenarios and help identify weaknesses in software that could cause catastrophic disasters was proposed in [41]. However, to the best of our knowledge, this is the first time the injection of realistic software faults (based on a field study on the most common types of faults [9]) is used to experimentally estimate software risk. It is worth noting that the fault injection technique used in our method does not require the source code of the target components, which means that it can be used even in COTS for which the source code is not available. However, our methodology uses software metrics, which implies either the existence of tools able to extract software metrics from executable code, or the availability of the source code itself. In the experiments presented in this paper we actually have access to the source code of all the components and we used that fact to simplify experiments, especially in what concerns the use of software complexity metrics tools, as we extract metrics based on source-code.

The remainder of this paper is organized as follows: the next section presents related work. Section 3 shows how to evaluate the risk using the proposed methodology. Two case studies to illustrate the actual use of the proposed methodology are presented in Section 4. Section 5 concludes the paper.

2. Related Work

The approach proposed in this paper is related to three main research lines: risk evaluation, the use of software metrics to estimate component fault density, and injection of software faults to evaluate cost of

component failures. In this section we summarize the most relevant works for our proposal.

Software risk is often assessed based on rigorous risk analysis methods or by using heuristics [4]. Heuristic risk analysis includes a checklist of questions, suggestions or guidewords, such as "is the component unstable or new?", "does the component implements a complex business rule?", etc. Rigorous risk analysis generally apply statistical models such as software reliability modeling to estimate the component failure likelihood [21, 37], and hazard analysis to estimate the consequence of failures [18]. By combining the consequence and the likelihood of failures, it is possible to rank the risk of each individual components of a system.

Many studies have tried to mitigate the problems associated to software faults and estimate their risk with particular emphasis on studies on software testing, software reliability modelling, and software reliability risk analysis [21, 26, 17].

The software risk assessment equation used in most of the literature is basically the same and reflects the probability of faulty behavior in a given software component and its impact (or cost). However, this equation is interpreted in different ways, depending on the approach used for risk assessment in each particular work.

The equation presented in [32] considers the object-oriented CK metrics [7] to estimate how error-prone the component is. The higher the metrics the more error-prone the component is. The risk is evaluated considering the probability that an undesirable event Ei occurs (p(Ei)) and the cost to the system if this event really occurs (c(Ei)), as shown in equation (1). In the context of estimating risk in software systems, an undesirable event is a component failure.

$$\text{Risk} = \sum(p(Ei) * c(Ei)) \qquad (1)$$

Sherer presents another concept of risk [36], as a function of fault activation probability in a pre-defined time, the quality of the development process and the operational profile. The work presented in [1] expands the Rosenberg's equation [32] in order to consider the component exposure from the point of view of the customer and from the point of view of the vendor.

Software complexity metrics have been widely used to estimate the probability of component faults, which are obviously related to the probability of component failure required in the typical risk equation (e.g., in equation (1)). The study presented in [5] experimentally validates object-oriented design metrics as quality indicators to predict fault-prone classes and conclude that several of these metrics appear to be useful to predict class fault-proneness during the early phases of the life-cycle.

The component failure likelihood is directly related to the complexity of that component [21]. In fact, complexity metrics have been used in many studies that show a clear link between component complexity and error proneness [29, 17, 10]. However, Fenton shows that this trend does not hold in some cases [11]. Some explanations for this apparent contradiction are provided in [24], and the use of static code metrics is recommended as predictors if these predictors are treated as probabilistic and not as categorical indicators. Menzies et. al. also reinforce the importance of finding good attributes set for each problem and to analyze a large amount of data sets in order to generalize the results.

Concerning the estimation of the impact of component failures (the term cost in the risk equation), the Failure Mode and Effect Analysis (FMEA) technique [18] is widely used to estimate component failure cost (in FMEA this is called severity analysis). This technique is particularly used in the development of software for highly regulated application areas such as avionics, space, and nuclear applications.

The use of fault injection to evaluate experimentally the cost (i.e., the impact) of failures in computer systems is also widely used [3, 15]. The impact of failures (equivalent to cost in the risk equation) is generally described in fault injection works as failure modes, which express the system response to the injection of each fault (e.g., crash, hang, erroneous output, etc).

Although fault injection techniques are quite popular, their use to estimate risk has not been addressed in the literature, especially in what concerns software risk. In fact, the evaluation of the impact of software component failures would need the injection of software faults, and techniques to inject this type of faults have been largely absent from the fault injection research. Most of the fault injection works actually inject faults that emulate hardware transient faults. Very often, faults are injected using the SWIFI approach (Software Implemented Fault Injection), but this must not be confused with the injection of software faults (i.e., program defects or bugs), as SWIFI tools actually emulate hardware faults through the injection of errors by software.

The problem of injecting representative software faults was first addressed in [8]. That work was done in the context of IBM's Orthogonal Defect Classification (ODC) project and the proposed method requires field data about real software faults

in the target system or class of target systems. This requirement (the knowledge of previous faults in the target system) greatly reduces the usability of the method, as this information is seldom available and is simply not possible to obtain for new software. Furthermore, as shown in [22], typical fault injection tools are not able to inject a substantial part of the type of faults proposed in [8].

To the best of our knowledge, the first practical approach to inject software faults was proposed in [9]. The approach is based on a technique named Generic Software Fault Injection Technique (G-SWFIT) and is supported by the findings from an extensive field study [9]. As we use this technique in the approach proposed in this paper it will be explained in more detail in section 3.2.

3. An Experimental Approach for Software Component Risk Assessment

Our goal is to evaluate the potential risk of using a given software component in a larger system. This component is typically a COTS but the proposed technique can actually be applied to any component of the system under analysis. The risk of using a given component in a system is calculated as in equation (2).

$$\text{Risk} = prob(f) * cost(f) \quad (2)$$

The first term, prob(f), represents the probability of residual faults in the component, i.e., corresponds to the component fault-proneness. The term cost(f) represents the consequence (i.e., impact in the system) of the activation of a fault f in the component.

Our proposal is to evaluate cost(f) experimentally by injecting software faults in the target component and measuring the faults impact in the system under analysis, and estimate prob(f) by using complexity metrics of the target component. If the injection of faults in a given component shows that a large percentage of faults cause a big impact in the system (high cost(f)) and that component is also very complex (high prob(f)), that means the component represents a high risk. If, on the contrary, faults injected in the component do not have significant impact or the component is not complex, then the result of equation (2) is a low risk.

3.1. Residual Fault Probability Estimation

The prediction of residual faults can be based on various methods. Some works use parametric models based on defect history [38, 10] and others use heuristics by comparing complexity metrics with a threshold [32]. Our work elaborates from previous proposals [5, 38, 27] to estimate prob(f) and follows a model based on logistic regression. Logistic regression [14] was the used statistical analysis to address the relationship between metrics and the fault-proneness of modules (in this work module and component is used as having the same meaning). Logistic regression is a classification technique widely used in experimental sciences based on maximum likelihood estimation of dependent variables (e.g. the failure likelihood) in terms of independent variables (e.g. complexity metrics). Logistic regression gives to each independent variable, also called "regressor", an estimated regression coefficient βi, which measures the regressor contribution to variations in the dependent variable. The larger the coefficient in absolute value the more important the impact of the variable on the probability of a fault to be detected in a component is. In our case, this probability prob represents the probability that such component has a residual fault and, consequently, (1 – prob) is the probability that this component does not have a fault. To be able to establish a linear relationship we need a logistic transformation and a logit of prob is taken. The logit(prob) is defined as ln (prob / (1 - prob)). The value of prob is given by equation (3) where α and β are the estimated logistic regression coefficients and exp is the inverse function of ln.

$$prob = \frac{\exp(\alpha + \beta x)}{1 + \exp(\alpha + \beta x)} = \frac{e^{(\alpha+\beta x)}}{1 + e^{(\alpha+\beta x)}} \quad (3)$$

Using the logit it is possible to obtain a simple form of a multivariate logistic regression model based on the relationship presented in equation (4) where several independent variables (in our case, static metrics) can be used.

$$\text{logit}(prob) = \ln\left(\frac{prob}{1-prob}\right) = \alpha + \beta_1 x_1 + .. + \beta_n x_n \quad (4)$$

To estimate the residual fault probability we need to identify which metrics are relevant, since the chosen metrics strongly dependent on the system characteristics, operational profile, the risk type, and the particular aspects that are being evaluated. When there is more than one metric available, we need to select which of them is best suited to the evaluation of the software complexity.

The size (in terms of lines of code - LoC) of the component was emphasized as an example of the direct relationship of measures of software complexity and measures of software quality [27] and it is one of the first and most common forms of

software complexity measurement. LoC was used to compose the field observation and was combined with empirical fault density provided by Rome Laboratory [31]. We start by considering the cyclomatic complexity (Vg) as regressors for prob(f). Vg measures the control flow complexity of a program and is dependent on the number of predicates (logical expression such as if, while, etc). Vg is also one of the most used software metric and it is language independent.

The accuracy of the results obtained in the first experiments has been evaluated through the analysis of bug reports available from open software initiatives. In consequence of this study (see [25]), some other metrics have been added to achieve a better approximation for the estimated fault density when compared to the bugs observed in field. The new metrics added are: number of parameters, number of returns, maximum nesting depth, program length and vocabulary size [12]. Halstead's metrics and Vg measure two distinct program attributes [27] leading to a better fault prediction capability [20].

According to equation (4) and considering six metrics, the probability that a component has a residual fault (prob(f(X1, X2, X3, X4, X5, X6)) can then be expressed as in equation (5). This equation allows us to use any number of metrics we consider appropriate to calculate the probability of the existence of residual fault. For that purpose we only need to add one more term (Bi Xi) for each metric to be considered in the equation.

$$prob(f) = \frac{\exp(\alpha + \beta_1 X_1 + ... + \beta_6 X_6)}{1 + \exp(\alpha + \beta_1 X_1 + ... + \beta_6 X_6)} \quad (5)$$

In the above equation, Xi represents the product metrics (independent variables) and α and βi the estimated logistic regression coefficients. We used statistical significance p-value to check the accuracy level of the coefficient estimation. Historically, common p-value thresholds are 0.01, 0.05 or 0.1 [5]. The calculated impact of the variable is more believable when the p-value is lower. Based on these analyses we decided which metrics give us the better estimation and eliminated the others. In the present work we used 0.1 as p-value threshold.

In order to obtain the coefficients for logistic regression [14] we proceed as follows:
- Evaluate the complexity metrics of each module.
- Adopt fault density ranges accepted by the industry community as a preliminary estimation of fault densities. In our work we use the empirical fault density reported by Rome Laboratory [31] as a starting estimation for the logistic regression. We used this preliminary estimation to replace the field observation used in any regression analysis.
- Use the binomial distribution. Taking into account prob as 0.1 fault per KLoC and LoC metric for each module we get the number of lines with residual faults in the module i as a binomial random variable with parameters LoCi and prob, and defines a preliminary fault density for module i. This preliminary density is then refined with the contribution of the other metrics using regression. The binomial distribution is used as we consider the existence of a fault is independent from the existence of other faults in the remaining component program lines.
- Apply the regression using the value obtained from natural logarithm of the preliminary fault density and the chosen metrics aim to obtain the coefficients. We use the regression algorithm available in Microsoft® Excel.
- Estimate the probability of fault of each component by using the computed coefficients in the logistic equation presented in (5).

When necessary to estimate the probability of residual fault in a set of components (the case of a big component formed by several sub-components) we have to use the prob(f) estimated for each sub-component combined with the complexity weight of each sub-component in the global component. This is obtained by equation (6), where Metricsi represents any of the available metrics for each component i. One can choose the metric that best represent the system characteristics (for example, maximum nesting depth if the system has several nested structures).

$$probg(f) = \sum probi(f) * (Metricsi / \sum Metricsi) \quad (6)$$

3.2. Failure Cost Estimation Through Injection of Software Faults

The component failure cost is estimated using software fault injection. We use the G-SWFIT [9] technique to inject the software faults. G-SWFIT is based on a set of fault injection operators that reproduce directly in the target executable code the instruction sequences that represent most common types of high-level software faults. These fault injection operators resulted from a field study that analyzed and classified more than 500 real software faults discovered in several programs, identifying the most common (the "top-N") types of software faults [9]. Table 1 shows the 12 most frequent types of faults found in [9]. We use these 12 fault types in the present paper.

The locations where the injection are performed are selected by the analysis of the target code (done by the G-SWFIT tool), which allows the identification of the places where a given fault type could indeed realistically exist, and avoids using locations where faults of the intended type could not exist. For example, MIFS fault type in Table 1 can only be injected in target code locations that represent an IF structure. The analysis of the target code is based on the knowledge of how the high-level constructs are translated into low-level instruction sequences [9]. The distribution of the number of fault injected in each component is based in our previous proposal [25]. Furthermore, for large components with a very large number of fault locations, faults are internally distributed according to the distribution show in Table 1 (third column). For small components with a small number of fault locations is not possible to follow the distribution in Table 1. In this case, faults, inside the component, are injected using the best approximation for the distribution in Table 1.

Table 1 – Most frequent fault types found in [9]

Fault types	Description	Perc. Observed in field study	ODC classes
MIFS	Missing "If (*cond*) { statement(s) }"	9.96 %	Algorithm
MFC	Missing function call	8.64 %	Algorithm
MLAC	Missing "AND EXPR" in expression used as branch condition	7.89 %	Checking
MIA	Missing "if (*cond*)" surrounding statement(s)	4.32 %	Checking
MLPC	Missing small and localized part of the algorithm	3.19 %	Algorithm
MVAE	Missing variable assignment using an expression	3.00 %	Assignment
WLEC	Wrong logical expression used as branch condition	3.00 %	Checking
WVAV	Wrong value assigned to a value	2.44 %	Assignment
MVI	Missing variable initialization	2.25 %	Assignment
MVAV	Missing variable assignment using a value	2.25 %	Assignment
WAEP	Wrong arithmetic expression used in parameter of function call	2.25 %	Interface
WPFV	Wrong variable used in parameter of function call	1.50 %	Interface
	Total faults coverage	**50.69 %**	

The evaluation of the cost of component failures is done by injecting one fault at the time. After the injection of each fault, the cost is measured as the impact of the fault injected in the component in the whole system. This impact is translated in the following failure mode: Hang – when the application is not able to terminate in the pre-determinate time; Crash – the application terminates abruptly before the workload complete; Wrong – the workload terminates but the results are not correct; Correct – there are no errors reported and the result is correct.
Considering the four failure modes proposed, only the "Correct" failure mode means that the system has delivered correct service after the injected fault. This means that we could in fact reduce the failure mode to only two: correct or incorrect behavior. However, we decide to keep the four failure modes to have more detailed information on the impact of each fault. When a software fault is injected in a given component, that fault may or may not cause faulty behavior in the component. Furthermore, only a fraction the faults that cause erroneous behavior in the component, will cause the system to fail, depending on the system architecture and the operational profile (the remaining faults are tolerated or have no visible effect). This means that the results measured by using fault injection already include two important terms:

$$cost(f) = prob(fa) * c(failure) \quad (7)$$

where prob(fa) is the probability of fault activation and consequent deviation in the component behavior and c(failure) is the consequence of a failure, for example the probability that the system crash.

4. Case Studies

We selected two case-studies to show the experimental risk estimation as proposed in our approach. The first case-study is a client-server Java application and the second is a real-time C application. In the second case-study, the C application has been implemented for both the RTMES and RTLinux operating systems and allows us to assess the risk for each of these alternative components.

The metrics of each software component were obtained with the Resource Standard Metrics [33], CMT++, and CMTJava [39] tools. The metric Max Nesting Depth is not provided for Java components. The following sub-sections describe the static and dynamic aspects of the risk assessment for each case study.

4.1. Client-server Java Application

This setup consists of an Object-Oriented Database Management System (Ozone [28]) and an implementation of the Wisconsin OO7 Benchmark

[6]. Ozone environment is based on a client / server architecture and the connection between clients and the database is established via "sockets" with a protocol that plays a similar role as RMI (Remote Method Invocation). To avoid object replication, Ozone uses unique instance architecture and the actual instance resides in the server. At the client, objects are controlled via "proxy" objects which can be seen as a persistent reference. Ozone runs on a Windows XP Professional operating system with j2sdk version 1.4.

Ozone is considered the target component, as it represents the role of a typical of-the-shelf component in this setup. As mentioned, the injection of software faults into Ozone is done using the G-SWFIT [9] technique and associated tools. The main component of the OO7 benchmark is a set of composite parts associated to a document object. The OO7 benchmark uses an object-oriented database to store a design library composed by a set of documents organized hierarchically. Figure 1 shows the used setup.

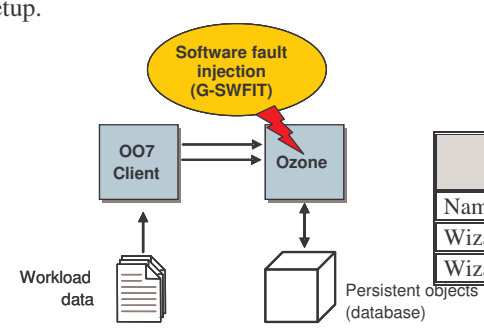


Figure 1: The Client-server Java Application Setup

Ozone is composed by 443 classes ranging from 1 to 368 lines of code (LoC) and the global LoC metric for Ozone is 12249. The interception coefficient (α) computed in regression analysis is -8.323204 (explained in section 3.1). Table 2 presents the global values of each metrics and the coefficients obtained in regression analysis.

The second term of the risk equation (cost) is empirical evaluated using fault injection. In order to show the estimation of risk for different components we decided to perform the experimental risk analysis for three different Ozone classes (instead of performing the risk estimation for the whole Ozone, which would produce a single number). Three selected Ozone classes have very different sizes and complexity: NameTableChange (low complexity), WizardObjectContainer (medium complexity), and WizardStore (higher complexity). This different complexity and size target components are essential to evaluate our proposal, as we expect to see the impact of different component size and complexity in the evaluated risk.

Table 2: Metrics and Logistic Regression Coefficients Ozone

Metrics	Global Value	Coefficients	*p-value*
C. Complexity	6312	0.0029272	0.622162
N. Parameter	2694	0.0030662	0.662503
N. Return	3670	0.012606	0.00671
Progr. Length	146768	-0.001427	1.30 E-13
Vocab. Size	27907	0.030578	8.38 E-54

We identify 196 injection locations in the three Ozone components. Note that the number of software faults is dependent on the actual component code. For example, the component NameTableChange, with very low complexity, only allows the definition of 3 faults. That is, it is not possible to consider that a programmer would have more than three places to make a mistake that could be considered as a realistic [9]. Table 3, summarizes the Ozone failure modes obtained after the injection campaign.

Table 3: Failure Modes and Results – Ozone

Class	#Fault Injected	Hang	Crash	Wrong	Correct
NameTableChange	3	0%	100%	0%	0%
WizardObjectContainer	42	0%	12%	0%	88%
WizardStore	151	1%	24%	0%	74%

Table 4 shows the risk evaluation concerning several system failure modes for the OO7 application. Different components really represent different risks for the system. It is clear that WizardStore represents a higher risk when compared to the other two components.

An interesting result shown in Table 4 is emphasized by the very small size and complexity of the class NameTableChange. The impact of all of the faults injected in NameTableChange is very high (cost = 1 for the failure mode Crash). Nevertheless, as the complexity metrics of that component is very low, the estimated prob(f) is also very low, showing that there is a very low probability of residual fault in that component. Thus, the measured risk related to that component is also quite low. Additionally, despite the fact that WizardStore cost is twice more than the WizardObjectContainer, its risk is almost 18 times higher than the risk of WizardObjectContainer. This is due to the high fault density estimated for WizardStore, which is 900% higher than the fault density estimated for WizardObjectContainer.

Table 4: The Risk Evaluation – Failure Mode Ozone

Classe	*prob(f)*	Crash		Hang		Incorrect Behavior	
		cost(f)	risk	*cost(f)*	risk	*cost(f)*	risk
NameTableChange	0.0005167	1	0.052%	0	0%	1	0.052%
WizardObjectContainer	0.0134783	0.12	0.162%	0	0%	0.12	0.162%
WizardStore	0.1205071	0.24	2.892%	0.01	0.121%	0.25	3.013%

Many other dimensions of risk could be calculated based on the failure modes. For example, the risk of not delivering the adequate service could be assessed considering all the faults that did not cause "correct" failure mode (i.e., the ones that caused incorrect behavior, crash or hang or wrong results). Although Table 3 does not show failure modes based on timing aspect, in fact we could calculate the risk of getting the results too late (assuming a given minimum response time has been defined), as the timing behavior was also recorded in the experiments.

4.2. Real-time C Application

The second case study is a satellite data handling system named Command and Data Management System (CDMS). A satellite data handling system is responsible for managing all data transactions between ground systems and a spacecraft. The CDMS application was developed in C and runs on top of two alternative real-time operating systems: RTEMS [34] and RTLinux [19]. This is particularly relevant as it allows us to show an interesting utilization of the proposed approach, which is to help developers in choosing between alternative of-the-shelf components (RTEMS and RTLinux in this case).

Figure 2 shows the satellite data handling system setup. The CDMS system is composed by six subsystems (only partially shown in figure 2): Packet Router (PR), Power Conditioning System (PCS), On Board Storage (OBS), Data Handling System (DHS), Reconfiguration Manager (RM), and Payload (PL).

The CDMS runs a mission scenario where a space telescope is being controlled and the data collected is sent to ground system. All data involved in this scenario is predetermined which allows deterministic experiments. The workload starts when an acknowledgement command is sent from the CDMS to the ground control. After that, the ground control sends a series of commands for the CDMS requesting telemetry information. The CDMS sends back telemetry information for each command sent. The timing of the commands and the contents of the telemetry information are used to detect the system correctness/failure during the experiments. The ground control software is hosted in a computer running Linux.

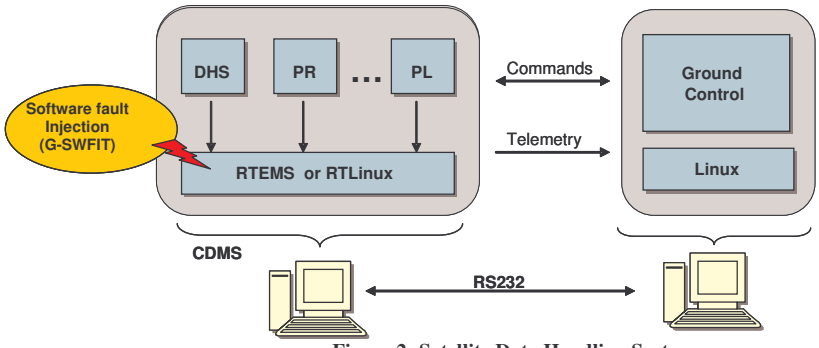


Figure 2: Satellite Data Handling System

Table 5: Metrics and Logistic Regression Coefficients – Real-time C application setup

Metrics	RTLinux			RTEMS		
	Global Values	Coefficients	p-value	Global Values	Coefficients	p-value
C. Complexity	39604	0.0072393	6.51 E-11	28536	0.0063537	7.09 E-05
N. Parameters	10778	-0.0051718	0.185622	8454	0.0117627	0.012413
N. Returns	13268	0.0431363	1.75 E-52	10240	0.0161907	0.000616
Progr. Length	1172521	-0.0001692	0.001896	787949	-0.0005537	7.9 E-20
Vocab. Size	171408	0.0011511	3.69 E-05	108550	0.0104020	2.48 E-47
Max. Nest. Depth	3963	0.3746203	1.0 E-140	2478	0.2354918	3.88 E-27

Analysing the two alternative operating systems for the real-time C application setup we observed that RTLinux is composed by 2211 modules with a total of 85108 lines of code, and the interception

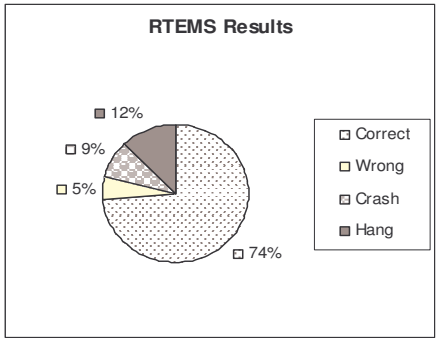


Figure 3: RTEMS Results

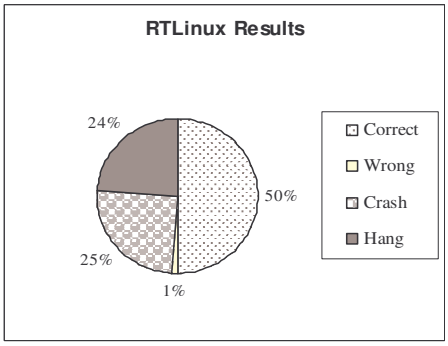


Figure 4: RTLinux Results

coefficient (α) is equal to -7.8443977 (calculated by regression as explained in section 3.1). RTEMS is composed by 1257 modules with a total of 63258 lines of code. The interception coefficient for RTEMS is -7.944308.

Table 5 presents a summary of metrics evaluation and coefficients for the two operating systems, showing the global values for each metric for both components and the coefficients (βi) obtained. These coefficients were applied in the logistic equation (refer to equation 5) to obtain the estimated prob(f) of each component.

The global probg(f) estimated for RTLinux is 6.50% and for RTEMS is 7.49 %. These values are calculated using equation (6) as explained in section 3.1.

A close observation of Table 5 shows that the complexity metrics of RTLinux are higher than the RTEMS metrics. It is then surprising why the probg(f) estimated for RTEMS is higher than the probg(f) estimated for RTLinux. An in depth analysis shows that RTEMS has a higher percentage of modules with high complexity when compared to RTLinux. Although the global complexity (i.e., sum of complexity metrics of all modules) of RTEMS is smaller than the global complexity for RTLinux, the large number of modules with high complexity in RTMES is responsible for the higher probg(f) of RTEMS when compared to RTLinux.

Application	# Module	LoC			C. Complexity			Global
		< 100	100 - 400	> 400	< 25	25-40	> 40	$prob_g(f)$
RTEMS	1257	87,0%	11,0%	2,0%	80,0%	6,0%	14,0%	7,5%
RTLinux	2212	90,0%	9,0%	1,0%	84,0%	6,0%	10,0%	6,5%

Table 6: Metrics Distribution and Global probg(f)

Table 6 shows the complexity distribution of modules for the LoC and Cyclomatic Complexity metrics. Considering the large size of both the RTEMS and RTLinux code, there are a very large number of fault locations in both off-the-shelf components.

Current results correspond to 231 faults injected in the CDMS version running on top of RTMES and 341 faults injected in the RTLinux version of CDMS. In both cases, the software faults have been injected in the operating system code (RTMES or RTLinux) using the G-SWFIT technique as explained in section 3.2.

Table 7: The Risk Evaluation – Failure Mode Real-time C application setup

Component	prob(f)	Crash		Wrong		Hang		Incorrect Behavior	
		cost(f)	risk	cost(f)	risk	cost(f)	risk	cost(f)	risk
RTEMS	0.0749	0.09	0.67%	0.05	0.37%	0.12	0.89%	0.26	1.94%
RTLinux	0.0650	0.25	1.62%	0.01	0.06%	0.24	1.56%	0.50	3.25%

Figure 3 and 4 present the failure modes obtained in the fault injection campaigns in both operating systems and Table 7 shows the risk evaluation. The risk is evaluated considering each failure mode that represents erroneous and the combination of all the erroneous failure modes (Incorrect Behavior column in Table 7).

As we can see, RTLinux represents a higher risk than RTEMS for most of the failure modes considered, and RTMES seems to be a better choice for this application.

One exception is related to the wrong results failure mode, as the CDMS version running on RTLinux represents a lower risk of wrong results when compared to the RTEMS version (i.e., the RTLinux version shows fail silent behavior more frequently). It is worth noting that, as

both operating systems have Posix compliant API, the CDMS is practically the same for both operating systems. Thus, the differences observed in the measured risk do account for the operating system.

5. Conclusion

This paper presents a first approach to evaluate the risk of using a given software component by software fault injection. The risk is evaluated using both software metrics and software fault injection. The faults injected are meant to represent component residual faults realistically, and provide a measure of the impact of component failures (failure of the component where the faults are injected). Several software metrics are considered such as cyclomatic complexity, number of parameters, number of returns, maximum nesting depth and Haltead´s program length and vocabulary size. The complexity metrics are used to estimate the component fault probability. Logistic regression analysis was used to fit the expression of the fault probability with these metrics. Our risk equation considers the probability of fault activation to model the fact that some faults are not activated or simply tolerated.

The proposed experimental risk assessment approach was evaluated using two different setups: a Java object-oriented database (Ozone) running the well-known OO7 benchmark and a satellite data handling real time application written in C. The risk assessment technique is illustrated in each setup at very different levels of component granularity. In the Java setup, several internal Ozone components were analyzed to show how different components may represent very different risk. In the satellite data handling system the risk of using two well-know off-the-shelf components (RTEMS and RTLinux operating systems) was analyzed. Results show that RTEMS represents a considerably lower risk than the RTLinux for that application.

6. Acknowledgment

The authors thank to CAPES/GRICES and FAPESP to partially support this work. We thank also to MSquared Technologies for gracefully providing the full version of RSM tool, and Testwell Oy Ltd for CMT++ and CMTjava tools.

7. References

[1]Amland, S. “Risk-based Testing: Risk analysis fundamentals and metrics for software testing including a financial application case study”. The Journal of Systems and Software, 53, pp. 287-295, 2000.

[2] Anderson, T.; Feng, M.; Riddle, S.; Romanovsky, A. “Protective Wrapper Development: A Case Study”. Lecture Notes in Computer Science, vol 2589, pp. 1-14, Springer Verlag, London, UK, 2003.

[3] Arlat, J. et al. “Fault Injection and Dependability Evaluation of Fault Tolerant Systems”. IEEE Transaction on Computers, vol. 42, n. 8, pp.919-923, 1993.

[4]Bach, J. “Heuristic Risk-Based Testing”. in Software Testing and Engineering Magazine, 1999.

[5]Basili, V.; Briand, L.; Melo, W. "Measuring the Impact of Reuse on Quality and Productivity in Object-Oriented Systems". Technical Report, University of Maryland, Dep. Of Computer Science, Jan. 1995, CS-TR-3395.

[6]Carey, M.; Dewitt, D.; Kant, C.; Naughton, J. “A Status Report on the OO7 OODBMS Benchmarking Effort”. Computer Sciences Department, University of Wisconsin Madison,1994 .

[7] Chidamber, R.; Kemerer, F. “A Metric Suite for Object-Oriented Design”. In IEEE Transaction of Software Engineering, 20 (6), 1994.

[8] Christmansson, J; Chillarege, R. “Generation of an Error Set that Emulates Software Faults-Based on Fields Data”. Proc. of 26th Int. Symp. on Fault-Tolerant Computing, pp 304-13, Sendai, Japan, 1996.

[9] Durães J., Madeira, H. "Emulation of Software Faults: A Field Data Study and a Practical Approach ", IEEE Transactions on Software Engineering, November 2006 (Vol. 32, No. 11), ISSN: 0098-5589

[10]El Emam, K.; Benlarbi, S.; Goel, N.; Rai, S. “Comparing Case-based Reasoning Classifiers for Predicting High Risk Software Components”. Journal of Systems and Software, vol. 55, n. 3, pp. 301-320, 2001.

[11]Fenton, N.; Ohlsson, N. “Software Metrics and Risk”. Proc. of The 2nd European Software Measurement Conference (FESMA´99), 1999.

[12]Halstead, M. “Elements of Software Science”. Elsevier Science Inc., New York, NY, USA, 1977.

[13]Herrmann, D. “Software Safety and Reliability: Techniques, Approaches, and Standards of Key Industrial Sectors”. Wiley-IEEE Computer Society Press, 1st edition, January, 2000.

[14]Hosmer, D.; Lemeshow, S. “Applied Logistic Regression”. John Wiley & Sons, 1989.

[Hudepohl98] Hudepohl et al. “EMERALD: A Case Study in Enhancing Software Reliability”. in Proc. of IEEE Eight Int. Symposium on Software Reliability Engineering - ISSRE98, pp.85-91, 1998.

[15] Iyer, R. “Experimental Evaluation”. Special Issue FTCS-25 Silver Jubilee, 25th IEEE Symposium on Fault Tolerant Computing, pp. 115-132, 1995.

[16]Karolak, D. “Software Engineering Risk Management”. Wiley-IEEE Computer Society Press, 1st edition, November, 1995.

[17] Khoshgoftaar et al. “Process Measures for Predicting Software Quality”. in Proc of High Assurance System Engineering Workshop – HASE’97, 1997.
[18]Leveson, N. “Safeware, System Safety and Computers”. Addison-Wesley Publishing Company, 1995.
[19]The linux kernel. www.kernel.org. Accessed on Feb/06, 2006.
[20]Lyu, M.; Chen, J.; Avizienis, A. “Experience in Metrics and Measurements for N-Version Programming”. Int. Journal of Reliability, Quality and Safety Engineering, vol. 1, n. 1., pp. 41-62, 1994.
[21] Lyu, M. “Handbook of Software Reliability Engineering”. IEEE Computer Society Press, McGraw-Hill, 1996.
[22]Madeira, H.; Vieira, M.; Costa, D. “On the Emulation of Software Faults by Software Fault Injection”. Proc. of The Int. Conf. on Dependable Systems and Networks, NY, USA, 2000.
[23] McManus, J. “Risk Management in Software Development Projects”. Butterworth-Heinemann, November, 2003.
[24] Menzies, T.; Greenwald, J.; Frank, A. “Data Mining Static Code Attributes to Learn Defect Predictors”. IEEE Transactions on Software Engineering, Vol.32, n. 11, pp. 1-12, 2007.
[25]Moraes, R., Durães, J., Martins, E., Madeira, H. "A field data study on the use of software metrics to define representative fault distribution”. Proc. of Workshop on Empirical Evaluation of Dependability and Security – WEEDS in conjunction with DSN06, 2006.
[26] Musa, J. “Software Reliability Engineering”, McGraw-Hill, 1996.
[27] Munson, J.; Khoshgoftaar, T. “Software Metrics for Reliability Assessment”. in:Handbook of Software Reliability Engineering, Comp. Society Press, Michael R. Lyu editor, ch. 12, 1995.
[28]Ozone, OO Database Management System. http://www.ozone-db.org,. Accessed on Feb/06, 2006.
[29] Kitchenham, B.; Pfleeger, S.; Fenton, N. "Towards a framework for software measurement validation”. IEEE Transactions on Software Engineering, 21(12), pp. 929-944, 1995.
[30] Popstojanova, K.; Trivedi, K. “Architecture Based Approach to Reliability Assessment of Software Systems". Perf. Evaluation, vol. 45, nos. 2-3, pp. 179-204, Jun/01, 2001.
[31] Rome Laboratory (RL). “Methodology for Software Reliability Prediction and Assessment”. Technical Report RL-TR-92-52, vol. 1 and 2, 1992.
[32]Rosenberg, L.; Stapko, R.; Gallo, A. “Risk-based Object Oriented Testing”. In Proc of. 13th International Software / Internet Quality Week-QW, San Francisco, California, USA, 2000.
[33]Resource Standard Metrics, Version 6.1, http://msquaredtechnologies.com/m2rsm/rsm.htm. Last access 2005.
[34]Real-Time Operating System for Multiprocessor Systems. www.rtems.com, accessed in Feb/06, 2006.
[35] Shaw, M.; Clements, P. “A Field Guide to Boxology: Preliminary Classification of Architectural Styles for Software Systems”. Proc. 21st International Computer Software and Applications Conference, pp. 6-13, 1997.
[36] Sherer, S. “A Cost-Effective Approach to Testing”. In IEEE Software, 8 (2), pp. 34-40, 1991.
[37] Singpurwalla, N. “Statistical Methods in Software Engineering: Reliability and Risk”. Springer; 1st edition, August, 1999.
[38] Tang, M.; Kao, M.; Chen, M. “An Empirical Study on Object-Oriented Metrics”. In: Proceedings of the Sixth International Software Metrics Symposium, pp. 242-249, 1999.
[39]Testwell Oy Ltd. http://www.testwell.fi. Accessed on March/06, 2006.
[40]Vieira, M.; Madeira, H. “Recovery and Performance Balance of a COTS DBMS in the Presence of Operator Faults”. In: Proc. of The International Conference on Dependable Systems and Networks – DSN2002, pp. 615-624, Washington D.C., USA, 2002.
[41]Voas, J.; Charron, F.; McGraw, G.; Miller, K.; Friedman, M. “Predicting how Badly ‘Good’ Software can Behave”. IEEE Software, 1997.
[42] Weyuker, E. “Testing Component-Based Software: A Cautionary Tale”. IEEE Software, 1998.
[43]Yacoub, S.; Ammar, H. “A Methodology for Architectural- Level Reliability Risk Analysis". IEEE Trans. Software Eng, vol. 28, no. 6, pp. 529-547, Jun/02, 2002.

Apêndice A
Conceitos Fundamentais de Estatística Utilizados

A.1. Teoria da Amostragem e Testes de Partição

Um dos conceitos estatísticos utilizados no desenvolvimento desta tese foi a amostragem estratificada. Amostragem estratificada [126] e testes de partição são apresentados no trabalho de Podgurski [119] para estimar confiabilidade.

Amostragem estratificada é utilizada quando a distribuição da variável que desejamos estimar tem alta variância relativa. Nesse caso, o uso da amostra aleatória simples causaria uma amostra quase tão grande quanto a população. Assim, para melhorar a precisão da estimativa, a população é dividida em grupos denominados estratos, tomando por base um determinado critério. Os elementos de cada estrato apresentam o máximo de homogeneidade, ou seja, são similares em relação à variável utilizada. Sendo similares, uma amostra pode ser escolhida dentro de cada estrato, de forma que ao combinar os resultados da amostra dos diferentes estratos obtem-se uma estimativa do parâmetro da população com boa precisão.

Teste de partição pode ser considerado um tipo particular de amostragem estratificada, uma vez que o domínio de entrada do sistema é dividido em partições de acordo com o comportamento operacional e dentro de cada estrato assim obtido, é selecionada uma entrada para exercitar o sistema.

A teoria da amostragem estuda as relações existentes entre uma população e as amostras extraídas dessa população. É útil para avaliação de grandezas desconhecidas da população, ou para determinar se as diferenças observadas entre duas amostras são devidas ao acaso ou se são verdadeiramente significativas. Amostragem é o processo de

determinação de uma amostra a ser pesquisada, enquanto que amostra é a parte de uma população estatística que foi selecionada.

Na teoria da amostragem é necessário que se defina o tamanho da amostra para que os elementos da amostra representem estatisticamente a população. O tamanho da amostra é baseado: (i) no percentual de sucesso e insucesso da população considerando o critério estabelecido; (ii) no nível de confiança que se pretende trabalhar; (iii) na taxa de erros que se pode tolerar. A expressão apresentada na Figura A.1 define com base na teoria da amostragem, o tamanho da amostra que minimamente representa a população alvo do estudo estatístico [126].

$$n = \frac{\left(Z_{\alpha/2}\right)^2 . \hat{p} . \hat{q}}{E^2}$$

Onde:

E ≡ taxa de erros que se pode tolerar

$Z_{\alpha/2}$ ≡ valor crítico relacionado à confiabilidade que se pode depositar na taxa de defeito que está sendo utilizada

$\hat{p}$ ≡ taxa de defeito que se conhece da população

$\hat{q} = 1 - \hat{p}$

Figura A.1: Expressão Genérica para Cálculo do Tamanho da Amostra

O objeto de um estudo de amostragem pode incidir sobre um atributo ou qualidade dos elementos de uma população, nomeadamente sobre o estudo da proporção de indivíduos da população que tem um determinado atributo. A proporção (p) é então obtida considerando os indivíduos que tem o atributo (r) sobre o total da população (n).

$$p = \left(\frac{r}{n}\right)$$

A teoria da amostragem estratificada aliada à teoria de proporção foi utilizada para se representar estatisticamente um sistema com um grande número de componentes, baseando-se nas métricas CK [113] para a seleção dos estratos. Assim, separou-se os estratos com base nos valores da métrica WMC [113], considerando-se o valor padrão

sugerido por Rosenberg [88] e conservando a mesma proporção de elementos do componente completo na amostra estratificada.

A.2. Regressão Logística

Regressão logística [116] é a análise estatística utilizada para estabelecer um relacionamento entre métricas e a propensão a falhas de um componente. É uma técnica amplamente utilizada para análises experimentais baseada na estimativa da probabilidade máxima de haver falhas, calculada em termos de variáveis independentes (por exemplo, métricas de complexidade). A regressão logística estima um coeficiente de regressão (β_i) para cada variável independente (regressor). A regressão logística é dada pela expressão (3) sendo que *p*, representa a probabilidade da variável dependente.

$$P = \frac{\exp(\alpha + \beta x)}{1 + \exp(\alpha + \beta x)} = \qquad \frac{e^{(\alpha+\beta x)}}{1 + e^{(\alpha+\beta x)}} \qquad (3)$$

O coeficiente estimado mede a contribuição do regressor na estimativa da variável dependente (probabilidade da existência de falhas residuais). Por exemplo, na maneira como foi utilizada neste trabalho, quanto maior é o coeficiente estimado, maior é a importância do impacto da métrica de complexidade na probabilidade de uma falha residual ser revelada no componente.

Para que se possa estabelecer uma relação linear, é preciso fazer uma transformação e o *logit* da probabilidade do componente ter uma falha residual *p* deve ser tomado como base. O *logit*(*p*) é definido como sendo *ln* (*p* / (1 – *p*)). Assim, o valor de *p* é dado pela equação (4) onde α e β são os coeficientes estimados pela regressão logística. Usando o *logit* é possível obter uma equação simples (linear) para um modelo de regressão logística utilizando várias variáveis independentes (nesse caso, métricas de complexidade). A expressão (4) a seguir apresenta essa transformação.

$$\text{logit}(p) = \ln\left(\frac{p}{1-p}\right) = \alpha + \beta_1 x_1 + \beta_2 x_2 + \ldots \beta_n x_n \qquad (4)$$

Depois de calculados os coeficientes de regressão estes são utilizados para se estimar a probabilidade da variável dependente (probabilidade de haver uma falha residual) em cada módulo que compõe o sistema utilizando-se a seguinte equação:

$$p(X_1, X_2, .., X_n) = \frac{\exp(\alpha + \beta_1 X_1 + \beta_2 X_2 + ... + \beta_n X_n)}{1 + \exp(\alpha + \beta_1 X_1 + \beta_2 X_2 + \beta_n X_n)} \quad (5)$$

onde:

α ,βi: coeficientes obtidos através da regressão logística

Xi: métrica de complexidade de software e

exp: inverso da função *ln.*

Quanto maior, em valor absoluto, for o coeficiente *βi* maior é o impacto que a métrica *Xi* tem sobre a estimativa da probabilidade de falhas *p*. A significância estatística de cada métrica foi avaliada considerando-se o *p-value*. O *p-value* verifica a acurácia da estimativa dos coeficientes estimados pela regressão. Historicamente, valores padrões aceitos para o *p-value* são 0,01, 0,05, 0,1 [70]. O impacto da variável é mais confiável quanto mais baixo é o *p-value* associado.