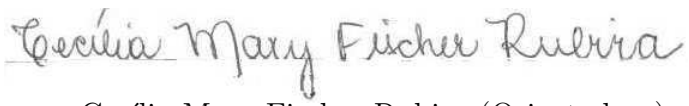
Uma Abordagem Arquitetural para o Desenvolvimento Rigoroso de Sistemas Confiáveis Baseados em Componentes

Este exemplar corresponde à redação final da Tese devidamente corrigida e defendida por Patrick Henrique da Silva Brito e aprovada pela Banca Examinadora.

Campinas, 17 de agosto de 2010.



Cecília Mary Fischer Rubira (Orientadora)

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

FICHA CATALOGRÁFICA ELABORADA PELA BIBLIOTECA DO IMECC DA UNICAMP
Bibliotecária: Silvania Renata de Jesus Ribeiro Cirilo - CRB8 / 6592

Brito, Patrick Henrique da Silva
B777a Uma abordagem arquitetural rigorosa para desenvolver sistemas de software confiáveis baseados em componentes/Patrick Henrique da Silva Brito-- Campinas, [S.P. : s.n.], 2009.

Orientador : Cecília Mary Fischer Rubira
Tese (Doutorado) - Universidade Estadual de Campinas, Instituto de Computação.

1. Tratamento de exceções. 2. Software - Arquitetura. 3. Tolerância a falha (Computação). 4.Engenharia de software. 5. Software - Desenvolvimento.. I. Rubira, Cecília Mary Fischer.. II. Universidade Estadual de Campinas. Instituto de Computação. III. Título.

Título em inglês: A rigorous architectural approach to develop dependable component-based software systems

Palavras-chave em inglês (Keywords): 1. Exception handling. 2. Software architecture. 3. Fault-tolerant computing. 4. Software engineering. 5. Software development.

Área de concentração: Engenharia de Software

Titulação: Doutor em Ciência da Computação

Banca examinadora: Profa. Dra. Cecilia Mary Fischer Rubira [Orientadora] (IC-UNICAMP)
Profa. Dra. Itana Maria de Souza Gimenes (DI-UEM)
Profa. Dra. Taisy Silva Weber (II-UFRGS)
Profa. Dra. Eliane Martins (IC-UNICAMP)
Prof. Dr. Ricardo de Oliveira Anido (IC-UNICAMP)

Data da defesa: 15/05/2009

Programa de Pós-Graduação: Doutorado em Ciência da Computação

TERMO DE APROVAÇÃO

Tese Defendida e Aprovada em 15 de maio de 2009, pela Banca examinadora composta pelos Professores Doutores:

Profª. Drª. Itana Maria de Souza Gimenes
DIN / UEM

Profª. Drª. Eliane Martins
IC / UNICAMP

Prof. Dr. Ricardo de Oliveira Anido
IC / UNICAMP.

Profª. Drª. Taisy Silva Weber
INF / UFRGS

Profª. Drª. Cecília Mary Fischer Rubira
IC / UNICAMP.

Instituto de Computação
Universidade Estadual de Campinas

Uma Abordagem Arquitetural para o Desenvolvimento Rigoroso de Sistemas Confiáveis Baseados em Componentes

Patrick Henrique da Silva Brito[1]

27 de março de 2009

Banca Examinadora:

- Cecília Mary Fischer Rubira (Orientadora)
- Itana Maria de Souza Gimenes
  Departamento de Informática – Universidade Estadual de Maringá
- Taisy Silva Weber
  Instituto de Informática – Universidade Federal do Rio Grande do Sul
- Eliane Martins
  Instituto de Computação – Universidade Estadual de Campinas
- Ricardo de Oliveira Anido
  Instituto de Computação – Universidade Estadual de Campinas

[1]Bolsa de doutorado FAPESP (processo 06/02116-2) 2005–2008, bolsa de doutorado sanduíche CAPES (processo 0722-07-3) 2007–2008

Resumo

A incorporação de tolerância a falhas em sistemas de software normalmente acarreta em um aumento da complexidade, o que consequentemente torna a sua análise mais difícil. Além disso, o uso de mecanismos de tratamento de exceções de uma maneira não-sistemática pode acarretar na adição de novas falhas ao sistema.

Esta tese apresenta uma abordagem rigorosa e centrada na arquitetura para o desenvolvimento de sistemas de software tolerantes a falhas. Dependendo do modelo de falhas e da disponibilidade de recursos, abstrações arquiteturais diferentes podem ser utilizadas para representar explicitamente questões relacionadas a tolerância a falhas, tais como detecção e tratamento de erros e tratamento de falhas. Essas abstrações arquiteturais e os seus respectivos detalhamentos internos podem ser instanciados em componentes e conectores concretos durante o projeto de arquiteturas de software tolerantes a falhas. De forma complementar, a solução proposta também define atividades que combinam o uso de métodos formais e casos de teste baseados em modelos para sistematizar a verificação e validação do comportamento do sistema relativo à propagação e tratamento de erros e tratamento de falhas no nível arquitetural.

A verificação e validação do software ocorrem em duas fases complementares do processo de desenvolvimento do software, ambas baseadas em cenários arquiteturais que descrevem a propagação e tratamento de erros envolvendo elementos arquiteturais (componentes e conectores). Primeiramente, utilizando a ferramenta de verificação de modelos ProB, que combina o uso de teoria de conjuntos matemáticos (B-Method) com álgebra de processos (CSP), a arquitetura de software é verificada formalmente com o intuito de antecipar a identificação de falhas relacionadas ao projeto do sistema. Segundo, casos de teste são gerados a partir da arquitetura de software utilizando uma abordagem baseada em modelos. O objetivo dos casos de teste gerados é verificar a consistência entre os modelos arquiteturais já verificados formalmente e a implementação do sistema.

Finalmente, para auxiliar as atividades de verificação, a solução proposta também contempla a definição de regras de transformação automática de diagramas UML para especificação formal em B-Method e CSP. A diferença semântica existente entre a especificação semi-formal da UML e a especificação formal em B-Method e CSP é compensada

utilizando-se estereótipos e “tags” nos modelos UML.

A aplicabilidade prática da solução proposta foi avaliada no contexto de três estudos de caso: (i) uma aplicação com requisitos críticos de tempo real e confiabilidade; (ii) uma aplicação bancária real com requisitos críticos de confiabilidade e disponibilidade; e (iii) uma aplicação para dispositivos móveis.

Abstract

The incorporation of fault tolerance into systems normally increases their complexity, which consequently makes their analysis more difficult. Moreover, the use of exception handling mechanisms to develop robust software systems in a non-systematic manner can be a source of many design faults.

This thesis presents a rigorous and architecture-centric development approach for developing fault-tolerant software systems. Depending on the fault model and the resources available, different architectural abstractions can be employed for representing issues that are related to fault tolerance, such as error detection, and error and fault handling. These architectural abstractions and their internal views can be instantiated into concrete components and connectors for designing fault-tolerant software architectures. In a complementary way, the proposed rigorous solution also defines activities which use formal methods and model-based test cases to systematise the verification and validation of the system's behaviour related to error propagation and handling at the architectural level.

The verification and validation occur in two complementary phases of the software development, both of them based on architectural scenarios describing error propagation and handling involving architectural elements (components and connectors). First, using the ProB model checker, which combines the use of set-theory (B-Method) and process algebra (CSP), the software architecture is formally verified in order to anticipate the identification of faults related to the system's model. Second, model-based test cases are generated in order to assess the consistency between the verified software architecture and the implementation of the software system.

Finally, the proposed solution also defines rules for model transformation from UML diagrams to formal specification in B-Method and CSP. To overcome the gap between the semi-formal specification of UML and the formal models, the UML diagrams are complemented with predefined stereotypes and tags.

The feasibility of our approach was evaluated in the context of three case studies: (i) a critical real-time application; (ii) a real banking system; and (iii) a mobile application.

Agradecimentos

Primeiramente, como não poderia deixar de ser, gostaria de agradecer a Deus, que possibilitou essa oportunidade ímpar e providenciou tudo para que cada momento fosse proveitoso e trouxesse um aprendizado inesquecível e transformador! Nessa grande lição de vida, também gostaria de agradecer a alguns dos personagens que se destacaram durante o processo. Agradeço aos meus pais e irmãos, que me apoiaram de forma incondicional em todos os momentos, incentivando e ajudando a buscar sempre o melhor em cada objetivo. À minha noiva (Nem), que por mais difícil que fosse para nós, sempre me incentivou e participou ativamente de cada passo da minha vida profissional; me mostrando acima de tudo que o Amor é capaz de se fortalecer cada vez mais, ainda que estejamos fisicamente separados por distâncias continentais. Aos meus sogros e cunhados, o meu especial agradecimento pelo laço de família que há entre nós, que faz brotar um sentimento de segurança e a confiança de ter um futuro compensador. Aos meus espelhos acadêmicos: Cecília Rubira, minha orientadora, Rogério de Lemos, meu orientador no período em Canterbury, e Eliane Martins, minha orientadora para questões de teste de software. Apesar do conhecimento imenso que vocês me passaram, o maior bem de toda a herança que levo de vocês é o aprendizado de vida, caráter, profissionalismo e determinação. Como diria o Fernando Castor: "quando crescer, pretendo ser como vocês". Um agradecimento mais que especial aos meus amigos da Unicamp e testemunhas das alegrias e sofrimentos; vocês não imaginam (ou melhor, imaginam) o quanto motivaram cada passo desse caminho... sem vocês eu poderia nem chegarar ao final! Para evitar a gafe de omitir nomes importantes, dessa vez resolvi não listá-los... sintam-se todos abraçados e recebam os meus sinceros agradecimentos! De forma igualmente especial, gostaria de agradecer aos amigos de outras datas e locais: o pessoal de Alagoas, da pastoral da comunicação, do Centro Cultural do Castelo, e todos com quem tive o prazer de conviver e compartilhar experiências no decorrer da jornada da vida. *A more than special thanks to the new friends I could meet during my stay in Canterbury, UK. You all are the most visible proof that each difficulty during the journey in England worth it at all! Good friends are the assurance to overcome barriers and to see goodness in everything around us! You were my strength at the moments I felt alone. Good friends are the secret of happiness and the*

real success in life! Thank you very much indeed!!! Em termos institucionais, gostaria de agradecer à Universidade Estadual de Campinas (Unicamp), em especial ao Instituto de Computação (IC), pela oportunidade e infraestrutura necessários para a condução do trabalho. Também gostaria de agradecer à CAPES e à Universidade de Kent, em especial à Escola de Computação, que viabilizaram o estágio de doutorado sanduíche, que além de ter deixado muita saudade e experiência de vida, contribuiu de forma muito significativa para o resultado final do trabalho. À Fundação de Amparo à Pesquisa do Estado de São Paulo (FAPESP), pelo apoio financeiro tão necessário para a condução da pesquisa e manutenção de uma vida salutar em uma cidade com o custo financeiro de Campinas. Aos grandes amigos do Instituto de Computação (IC) da Universidade Federal de Alagoas (UFAL), antigos colegas de sala e professores, hoje amigos e colegas de trabalho. Vocês representam a plenitude desse sonho, que hoje posso chamar de realidade. Para fechar com chave de ouro, agradeço também aos revisores anônimos dos artigos submetidos e aos professores da banca examinadora da minha defesa, além de todos que colaboraram diretamente com a qualidade final do trabalho e dos textos escritos. O meu muito obrigado a todos aqueles que trabalham nos bastidores, cujo nome eu omiti ou desconheço. Finalmente, um obrigado especialíssimo a todos aqueles que esqueci injustamente de citar aqui. Minhas sinceras desculpas e o meu agradecimento.

"The intuitive mind is a sacred gift and the rationale mind is a faithful servant. We have created a society that honours the servant and has forgotten the gift."

(Albert Einstein)

"O mundo é um livro e quem fica sentado em casa lê somente uma página."

(Santo Agostinho)

"Vencer, crescer... são metas importantes na vida; chegar ao topo, buscar seu lugar ao sol é acima de tudo um sonho a ser conquistado; mas mais importante que tudo isso é chegar através de suas próprias escolhas, fazendo do seu caminho um traço da sua imagem."

(Neusvaldo Júnior)

Sumário

Lista de Tabelas

Lista de Figuras

Capítulo 1

Introdução

1.1 Contextualização

A adoção de componentes de software [100], que normalmente se restringia à construção de sistemas tradicionais, tem se expandido para outras áreas de atuação nas quais o custo de um defeito pode ser inaceitável. Sistemas de software que podem causar risco para vidas humanas ou grandes perdas financeiras podem ser tolerantes a falhas, de modo a possuir a capacidade de oferecer suas respectivas funcionalidades, ainda que na presença de falhas.

Tolerância a falhas[1] é a habilidade de um sistema continuar funcionando normalmente, ainda que parcialmente, apesar da presença de falhas [69]. Uma vez que tolerância a falhas possui necessita do escopo global do sistema, ela se relaciona fortemente com a arquitetura de software, que representa explicitamente a estrutura do sistema e é um dos primeiros artefatos que permitem a análise de atributos de qualidade do software, tal como confiança no funcionamento[2], incluindo confiabilidade[3], disponibilidade[4], segurança de acesso[5] e segurança no funcionamento[6] [7]. Porém, a incorporação de tolerância a falhas em sistemas de software normalmente aumenta consideravelmente a sua complexidade, dificultando a sua análise. Uma maneira de lidar com a complexidade inerente a sistemas tolerantes a falhas é através da adoção de abstrações arquiteturais. Essas abstrações são capazes de esconder parte da complexidade do sistema, além de oferecer meios explícitos para analisar como os erros são detectados, propagados e tratados, além de poder considerar a eliminação da falha [6].

[1] Do inglês *fault tolerance*
[2] Do inglês *dependability*
[3] Do inglês *reliability*
[4] Do inglês *availability*
[5] Do inglês *security*
[6] Do inglês *safety*

A implementação de tolerância a falhas se baseia na existência de redundância, que pode ser incorporada tanto implicitamente, quanto explicitamente na arquitetura de software. Um exemplo de redundância implícita é o uso do mecanismo de tratamento de exceções para possibilitar recuperação de erros. Tratamento de exceções complementa outras técnicas de recuperação de erros, tais como transações tômicas [58], com o objetivo de apoiar a construção de programas que sejam mais confiáveis, concisos e fáceis de evoluir [82]. A utilização de tratamento de exceções para desenvolver sistemas de software de grande porte [29, 88], aliado ao fato de que esse mecanismo é implementado por várias linguagens de programação modernas e orientadas a objetos, tais como Java, Ada, C# e C++, além de alguns modelos de componentes, tais como CCM, EJB, Ice e .NET, confirmam a sua importância para as práticas atuais de desenvolvimento de software. Além do mais, em aplicações onde é inviável desfazer[7] ações, tais como sistemas de controle de tempo real, tratamento de exceções pode ser a única escolha possível. Por outro lado, se não for dada uma atenção especial ao estruturar o sistema, as especificações normais e excepcionais[8] podem ser misturadas, o que aumenta ainda mais a complexidade do sistema. Redundância explícita é um aspecto inerente a sistemas fortemente estruturados[9] [87], isto é, sistemas nos quais a estruturação da redundância é considerada parte do próprio sistema, reduzindo consequentemente o impacto de falhas. Exemplos de redundância explícita são programação em N-versões[10] e blocos de recuperação[11], que são duas técnicas de tolerância a falhas [68].

1.2 Definição do Problema

Tolerância a falhas no nível da arquitetura de software tem recebido uma atenção considerável da comunidade científica, principalmente no contexto de tratamento de falhas[12]. Particularmente, aspectos relacionados a reconfiguração arquitetural, que inclui troca, adição e remoção de elementos arquiteturais ou mudança na topologia da da configuração arquitetural [12]. O mesmo não pode ser dito sobre propagação de erros e tratamento de erros no nível arquitetural. Porém, um trabalho recente considera abstrações arquiteturais como um meio de estruturar sistemas de software tolerantes a falhas baseado em tratamento de exceções [41, 42, 44]. Para considerar o comportamento excepcional[13] desde o início do processo de desenvolvimento de software, outros aspectos devem ser considerados

[7] Do inglês *rollback*
[8] Do inglês *abnormal*
[9] Do inglês *strongly-structured systems*
[10] Do inglês *N-version programming*
[11] Do inglês *recovery blocks*
[12] Do inglês *fault handling*
[13] Do inglês *abnormal behaviour*

durante a fase de projeto arquitetural do software, tais como a existência de divergências arquiteturais[14] envolvendo diferentes tipos de exceções, a existência de exceções inúteis ou obsoletas, a associação de tratadores apropriados para as exceções em pontos estratégicos da arquitetura do sistema.

De maneira complementar, com o intuito de identificar e remover falhas relacionadas ao comportamento excepcional do sistema, técnicas de verificação e teste devem ser utilizadas durante o projeto arquitetural e a implementação do software. Poucas contribuições têm explorado a verificação do comportamento excepcional no nível da arquitetura de software [28]. Por exemplo, o trabalho de Castor *et al.* propõe uma solução para especificar e verificar fluxos de controle de exceções[15] na arquitetura de software utilizando a linguagem de especificação Alloy. Porém, essa abordagem não considera o comportamento dos tratadores de exceções como parte do processo de verificação. Além disso, essa solução não escala quando o processo de verificação necessita considerar muitos tipos de exceções [28]. Em relação às atividades de teste, Castor *et al.* não consideram essa possibilidade. Entretanto, uma vez que a arquitetura de software foi verificada, ela deveria ser utilizada para gerar casos de teste baseados em modelo[16] com o objetivo de aferir a consistência entre a implementação do sistema e a configuração arquitetural previamente verificada. A geração de dados de teste (valores), tal como a geração dos oráculos de teste não serão discutidos no contexto dessa tese.

No contexto de aplicações críticas, confiança no funcionamento é considerado um atributo de qualidade essencial; porém, a implementação de técnicas de tolerância a falhas aumenta consideravelmente o custo do sistema. Com o intuito de reduzir os custos de se desenvolver sistemas de software complexos, muito esforço tem sido despendido para alcançar maiores níveis de reúso no decorrer do processo de desenvolvimento. Uma das principais abordagens discutidas na literatura para promover o reúso de artefatos de software é conhecida como linhas de produto de software[17] (SPL). Uma SPL é uma abordagem que sistematiza o reíso de artefatos de software através da exploração de comunalidades e variabilidades entre produtos de software similares [2]. Um dos principais artefatos no contexto de uma SPL é a arquitetura da linha de produto[18] (PLA),que representa explicitamente as comunalidades e variabilidades dos elementos arquiteturais e suas configurações. As comunalidades são reutilizadas em diferentes produtos, enquanto as variabilidades são resolvidas a partir das decisões de projeto relacionadas às escolhas na PLA.

Considerando a complexidade inerente a sistemas de software tolerantes a falhas e a falta de atenção relacionada a detecção de erros, propagação e tratamento no nível da

[14]Do inglês *architectural mismatches*
[15]Do inglês *exception control flows*
[16]Do inglês *model-based test cases*
[17]Do inglês *software product lines*
[18]Do inglês *product line architecture*

arquitetura de software, essa tese tem o objetivo de responder seis questões de pesquisa:

1. As técnicas de verificação formal e validação podem ser combinadas para definir um processo rigoroso de desenvolvimento e testes centrado na arquitetura?

2. Os cenários de uma configuração arquitetural (cenários arquiteturais) são boas referências tanto para a verificação quanto para a validação de sistemas tolerantes a falhas?

3. É possível definir variabilidades arquiteturais relacionadas às decisões de projeto relativas a tolerância a falhas de software?

4. Os cenários arquiteturais representam um critério de cobertura de teste apropriado no contexto de sistemas tolerantes a falhas?

5. Abstrações arquiteturais são capazes de ocultar efetivamente a complexidade do sistema relacionada a tolerância a falhas de software?

6. Abstrações arquiteturais melhoram o entendimento[19] da verificação formal e validação?

1.3 A Solução Proposta

Nessa tese, é apresentado um processo rigoroso e centrado na arquitetura para desenvolver arquiteturas de software baseadas em componentes. Esse processo utiliza técnicas de verificação formal e validação de software para reduzir o número de falhas adicionadas na configuração arquitetural e no respectivo código-fonte. As atividades de verificação e validação são executadas baseadas em cenários comportamentais envolvendo os elementos arquiteturais de uma configuração arquitetural específica (cenários arquiteturais). Apesar dessa especificação comportamental de alto nível abstrair o projetista de detalhes de implementação, ela oferece informações valiosas sobre os atributos de qualidade do sistema, tais como confiabilidade e disponibilidade [7]. Abstrações arquiteturais também são consideradas importantes no contexto do processo proposto, uma vez que elas podem ser efetivas no desenvolvimento de sistemas de software tolerantes a falhas através do apoio às fases de especificação, verificação e validação de arquiteturas de software tolerantes a falhas e baseadas em componentes. Diversas abstrações podem ser utilizadas, de acordo com as decisões de projeto, modelos de falhas assumidos e disponibilidade de recursos.

[19] Do inglês *understandability*

O processo rigoroso é apresentado no contexto de duas abstrações arquiteturais: (i) o elemento arquitetural tolerante a falhas idealizado[20] (iFTE), que oferece meios para promover confinamento de erros e tolerância a falhas no nível da arquitetura de software utilizando redundância implícita; e (ii) o elemento arquitetural falha-e-pára[21] (HoFE), que assume uma semântica de falhas permanentes[22], assim oferecendo base para detecção de erros e incorporação de redundância explícita no projeto de sistemas tolerantes a falhas. Dependendo do modelo de falhas e a disponibilidade de recursos, a abstração arquitetural mais apropriada deve ser utilizada. Para obter arquiteturas de software tolerantes a falhas e baseadas em componentes, essas abstrações devem ser instanciadas em componentes e conectores arquiteturais, que em seguida devem ser configurados de acrdo com as restrições de interação definidas pelas suas propriedades estruturais e comportamentais. Essas abstrações arquiteturais são apresentadas no contexto de um processo rigoroso e geral que alia o uso de linguagens forais para possibilitar: (i) a verificação automática de modelos arquiteturais para identificar erros de projeto e remover as respectivas falhas, e (ii) a geração de casos de teste baseados em modelos, que são capazes de identificar erros de implementação relacionados a inconsistências entre o código-fonte e a configuração arquitetural do sistema.

O objetivo do trabalho apresentado nessa tese é mostrar que a adoção de abstrações arquiteturais adequadas e de um simples processo rigoroso baseado em componentes pode facilitar a modelagem e análise da arquitetura de software de sistemas de software tolerantes a falhas. Comparada com alguns trabalhos relacionados apresentados na Seção 1.4, as contribuições principais desta tese são quatro.

Primeiramente é mostrado como abstrações arquiteturais devem ser definidas em termos de propriedades estruturais e comportamentais, de modo a serem utilizadas para guiar a especificação, verificação e validação de sistemas de software. Para isso, foram definidas duas novas abstrações arquiteturais: uma baseada em redundância implícita (iFTE) e outra baseada em redundância explícita (HoFE). Em segundo lugar, a tese apresenta como um processo recursivo rigoroso e baseado em componentes pode ser conduzido para lidar com diferentes tipos de abstrações arquiteturais. Em terceiro lugar, o trabalho apresenta um arcabouço[23] formal que pode ser utilizado para verificar propriedades da arquiteturas de software relacionadas a tolerância a falhas, além de gerar casos de teste baseados em modelo. Finalmente, o modelo formal proposto inicialmente foi estendido com o intuito de representar PLAs com variabilidades associadas a diferentes técnicas de tolerância a falhas, assim como diferentes estratégias de tratamento de erros.

Visão Geral da Abordagem de Verificação. No contexto da verificação for-

[20]Do inglês *idealised fault-tolerant architectural element*
[21]Do inglês *halt-on-failure architectural element*
[22]Do inglês *crash failure*
[23]Do inglês *framework*

mal, são identificadas três contribuições importantes: (i) um arcabouço formal que pode ser instanciado de forma a representar formalmente várias configurações e cenários arquiteturais; (ii) propriedades de interesse que representam características importantes de arquiteturas tolerantes a falhas e podem ser reutilizadas para verificar diferentes configurações arquiteturais; e (iii) um processo de verificação que sistematiza a identificação de erros na arquitetura de software utilizando checagem de modelo[24].

Em relação à notação formal utilizada para representar arquiteturas de software, linguagens de descrição de arquitetura[25] (ADLs) foram consideradas inicialmente, uma vez que são linguagens formais cujo propósito específico é representar formalmente arquiteturas de software. Porém, essas linguagens normalmente têm limitações quanto à representação de aspectos específicos do sistema. Quanto ao tratamento de exceções, que é essencial no contexto da abordabem proposta, as ADLs existentes não oferecem suporte para especificar e verificar o fluxo de controle de exceções envolvendo elementos arquiteturais, e nem permitem a especificação de propriedades gerais relacionadas a cenários de tratamento de exceções. Além do mais, para gerar casos de testes partir da especificação formal é necessário representar as interfaces dos elementos arquiteturais com informações sobre suas respectivas operações. Esse tipo de informação detalhada não é normalmente representada por ADLs. Para superar essas limitações, é necessário adotar uma notação formal que possibilite a representação de tipos de maneira explícita, para distinguir diferentes tipos de exceções. Além disso, para representar o fluxo de controle de exceções, incluindo conversão de tipos e mascaramento, a notação formal também deve possibilitar a especificação de cenários envolvendo elementos arquiteturais. Utilizando B-Method [1], que é uma notação baseada na teoria de conjuntos matemáticos, permite a especificação de diferentes tipos através dos elementos de conjuntos; dessa forma, elementos arquiteturais, interfaces e tipos de exceções podem ser representados explicitamente. Cenários arquiteturais são modelados utilizando CSP [22], que é uma álgebra de processos indicados para representar eventos sequenciais. O verificador de modelos[26] utilizado na solução proposta é o ProB [70], uma vez que permite o uso combinado de máquinas B-Method e especificações CSP de forma complementar. No ProB, CSP é utilizado para restringir os possíveis eventos da máquina B-Method.

Visão Geral da Abordagem de Validação. Apesar da estratégia de verificação formal utilizada melhorar a qualidade da arquitetura de software, ela não garante a ausência de falhas no modelo, ou sequer que o código-fonte do software satisfaz as propriedades verificadas na arquitetura. Com o objetivo de superar essa limitação, a abordagem proposta também considera a geração de casos de teste baseados em modelo,

[24]Do inglês *model checking*
[25]Do inglês *Architecture Description Languages*
[26]Do inglês *model checker*

cujo objetivo é aferir a consistência entre a arquitetura de software já verificada e seu respectivo código-fonte.

São utilizadas três técnicas complementares de geração de caos de teste baseados em modelo: (i) casos de teste de unidade são gerados a partir da especificação formal dos elementos arquiteturais. O objetivo desses casos de teste é aferir a consistência entre cada elemento arquitetural e a abstração arquitetural que ele segue; (ii) casos de teste de integração são gerados a partir da especificação formal da configuração arquitetural. O objetivo desses casos de teste é identificar divergências arquiteturais associadas à interação entre os elementos arquiteturais; e (iii) casos de teste de robustez também são gerados a partir da especificação formal da configuração arquitetural. O objetivo desses casos de teste é aferir o comportamento do sistema de software na presença de falhas (comportamento excepcional), que especialmente importante no contexto de sistemas de software tolerantes a falhas. Uma característica importante da abordagem de geração de casos de teste baseados em modelo, adotada pela abordagem rigorosa proposta é o fato dela oferecer um importante critério de cobertura dos testes baseado nos cenário arquiteturais excepcionais[27] do sistema tolerante a falhas em desenvolvimento.

Visão Geral do Processo Rigoroso. Na abordagem proposta, a arquitetura de software é considerada uma unidade de primeiro nível, que guia o desenvolvimento desde a especificação à implementação e teste do software. A Figura 1.1 apresenta uma visão geral da abordagem proposta para desenvolver arquiteturas de software tolerantes a falhas e baseadas em componentes. A Atividade 1 especifica a arquitetura de software, que pode ser feito graficamente utilizando uma ferramenta CASE. A partir dos casos de uso de cenários excepcionais, dois artefatos devem ser especificados: um diagrama de componentes UML representando a estrutura da arquitetura de software, e um conjunto de diagramas de sequência UML representando os cenários arquiteturais excepcionais de controle de fluxo de exceções e dos tratadores. Para gerar os cenários arquitetrais, os cenários dos casos de uso devem ser refinados de acordo com a configuração arquitetural do sistema em questão. A Atividade 2 especifica formalmente a arquitetura de software (configuração arquitetural e cenários). Essa atividade consiste de uma transforação automática de modelo de UML (arquivos XMI) para B-Method e CSP. Essa transformação consiste em instanciar os modelos[28] formais com as especificações estruturais e comportamentais extraídas dos modelos UML. Esses modelos formais são fornecidos como parte da solução proposta, reduzindo assim o esforço da especificação formal. A Atividade 3 é a verificação formal da arquitetura de software, de forma a identificar falhas de projeto relacionadas aos fluxos de controle de exceções e aos tratadores. A Atividade 4, que consiste na geração de casos de teste e do código-fonte do sistema, não é detalhada nessa tese. O processo

[27] Do inglês *abnormal architectural scenarios*
[28] Do inglês *templates*

geral apresentado na Figura 1.1 é considerado recursivo, uma vez que ele ode ser executado tanto para o sistema inteiro, quanto para a estrutura interna de um elemento arquitetural.

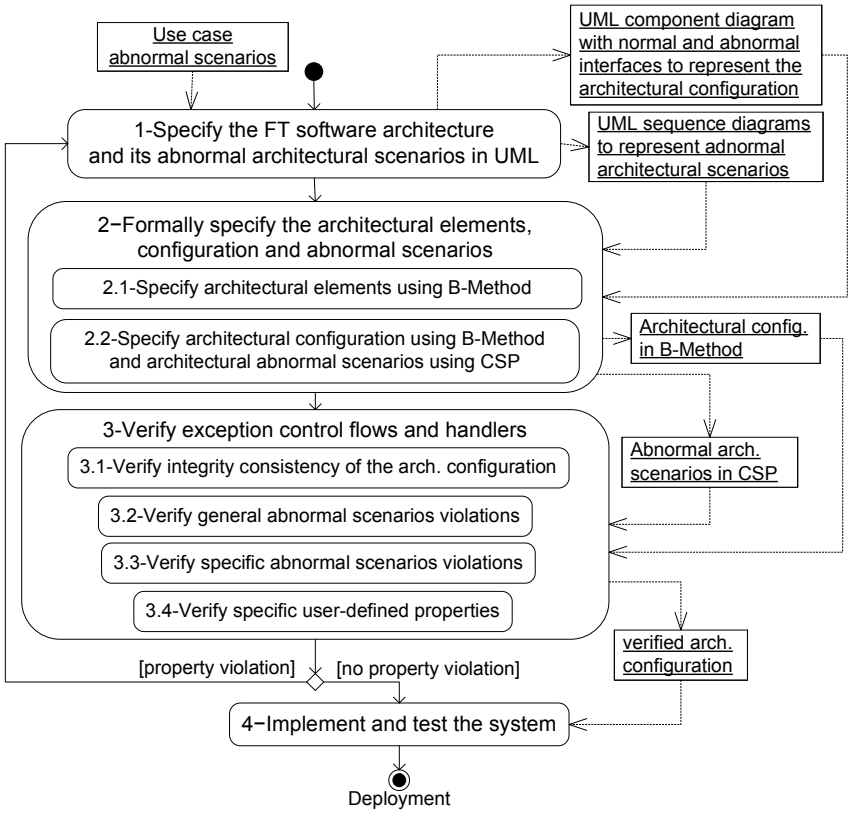


Figura 1.1: Um processo rigoroso para desenvolvimento e teste de sistemas de software tolerantes a falhas

O processo proposto apóia a especificação, verificação e validação de arquiteturas de software tolerantes a falhas e baseadas em componentes, apoiando a prevenção e remoção de falhas em diferentes estágios do processo de desenvolvimento [6]. Como a estrutura interna dos elementos arquiteturais pode ser detalhada recursivamente, a arquitetura de software pode ser especificada, verificada e validada hierarquicamente, o que ajuda a controlar a complexidade adicional associada com provisão de tolerância a falhas. Além disso, como a arquitetura de software é verificada no contexto de cenários comportamentais específicos, ele restringe o modelo a ser verificado e consequentemente, reduz a combinação de estados do modelo formal,aumentando assim a sua escalabilidade.

Apesar dessa tese lidar com a verificação e teste baseados em propriedades arquiteturais, ela não detalha como o comportamento excepcional pode ser especificado sistematicamente. Essa atividade complementar foi considerada em um trabalho anterior [21, 28].

1.4 Trabalhos Relacionados

Nessa seção são revisadas algumas publicações selecionadas relacionadas aos cinco tópicos principais cobertos nessa tese: abstrações arquiteturais, estruturação do comportamento excepcional do sistema, verificação do comportamento excepcional do sistema, teste baseado em modelos da arquitetura de software e arquiteturas de linha de produtos tolerantes a falhas.

1.4.1 Abstrações Arquiteturais

Algumas contribuições propõem notações basedas em teoria dos grafos para representar restrições estruturais de arquiteturas de software [46, 48]. A simplicidade dessas notações é alcançada através do uso de uma abstração arquitetural para representar elementos arquiteturais de alta granularidade. Além disso, Denford et al. [46] utiliza o conceito de refinamento arquitetural, que possibilita a modularização e decomposição interna dos elementos arquiteturais. Assim como nesse último trabalho, a abordagem proposta nessa tese foca na modelagem de arquiteturas de software em diferentes níveis de abstração, considerando a estrutura interna dos elementos arquiteturais de alta granularidade. Porém, as abstrações arquiteturais consideradas no nosso trabalho também focam na representação de restrições comportamentais, de modo que seja possível verificar e validar propriedades relacionadas a tolerância a falhas de software.

1.4.2 Estruturação do Comportamento Excepcional do Sistema

Sobre a definição de abstrações arquiteturais específicas para lidar com tolerância a falhas de software, o componente C2 idealizado[29] (iC2C) [41] é uma técnica de estruturação baseada no componente tolerante a falhas idealizado [69], que foca em sistemas de software compatíveis com o estilo arquitetural C2 [101]. O protocolo interno seguido pelos elementos internos de um iC2C favorece o confinamento de erros e possibilita definir múltiplos contextos de tratamento de exceções no nível da arquitetural de software. Um trabalho seguinte de Castor et al. [30] definiu e implementou um mecanismo de tratamento de exceções no nível arquitetural baseado no conceito do iC2C. A abstração arquitetural iFTE, apresentada na presente tese pode ser vista como uma extensão do iC2C para uma classe mais ampla de arquiteturas de software, uma vez que ele adere ao estilo arquitetural fim-a-fim[30] [33]. Além disso, a solução apresentada nessa tese também oferece suporte apropriado para verificar e validar o sistema baseado nas restrições e propriedades da abstração.

[29] Do inglês *the idealised C2 component*
[30] Do inglês *peer-to-peer*

Simons e Stafford [96] apresentaram um arcabouço, denominado CMEH, para especificar arquiteturas de software confiáveis baseadas em componentes comerciais de prateleira[31] (componentes COTS). CMEH possibilita a contextualização dos tipos de exceções, assim como a associação de tratadores de exceções nos conectores arquiteturais. Esse trabalho é complementar à solução proposta nessa tese, uma vez que foca na fase de implementação, ao invés da verificação e testes. Uma diferença semântica entre o CMEH e a abordagem proposta é o fato do CMEH não considerar os fluxos de controle de exceções na arquitetura de software, mas apenas os tratadores arquiteturais de exceções.

1.4.3 Verificação do Comportamento Excepcional de Sistemas

A respeito da verificação formal, algumas contribuições propõem o uso de análise estática de código-fonte para analisar o fluxo de controle de exceções [31, 92]. Nessas abordagens, a análise do fluxo de exceções consiste em identificar os cainhos de propagação de exceções em um programa, por exemplo, identificar exceções não capturadas em linguagens com tipos polimórficos, tal como Java. Um trabalho recente de Castor et al. [28], apresenta o arcabouço Aereal, que utiliza linguagens e ferramentas existentes para apoiar a descrição e análise de fluxo de controle de exceções em arquiteturas de software. Em relação à verificação de exceções arquiteturais, o arcabouço Aereal é similar ao trabalho proposto nessa tese, porém existem algumas diferenças importantes. O trabalho proposto possibilita a especificação de propriedades baseadas em cenários relacionados à arquitetura de software. Além disso, mesmo considerando apenas o foco comum de verificação, a abordagem proposta se mostrou mais escalável para o propósito de checagem de modelo. Finalmente, além das atividades de verificação, o modelo formal utilizado nesta tese também possibilita a geração de casos de teste para aferir a implementação da configuração arquitetural.

Sobre a representação formal de arquiteturas de software, existem algumas contribuições de linguagens arquiteturais que apóiam refinamentos sucessivos, tais como SADL [78], que permite o refinamento estrutural, e π-ARL [81], que permite tanto o refinamento estrutural, quanto comportamental. Diferentemente dessas notações, a solução proposta não define uma nova linguagem formal. Ao invés disso, são definicos modelos formais em B-Method [1] e CSP [24] para representar a estrutura e o comportamento da arquitetura de software, respectivamente. No contexto do comportamento excepcional de sistemas, isso é particularmente conveniente para representar diferentes tipos de exceções (in B-Method) e cenários de propagação de exceções e de tolerância a falhas (in CSP).

[31] Do inglês *Commercial Of The Shelf components*

1.4.4 Teste Baseado em Modelos Arquiteturais

Abordagens que casos de teste a partir de modelos formais da arquitetura de software já foram propostos anteriormente [9, 80, 89]. Um exemplo desse tipo de abordagem especifica casos de teste de integração utilizando um grafo que representa o comportamento do sistema [89]. A partir da descrição formal da arquitetura de software, a solução deriva um grafo que representa o comportamento do sistema em termos das interações entre os componentes. Em seguida, a solução garante que todas as arestas do grafo sejam cobertas pelos casos de teste. A principal limitação dessa abordagem é o grande número de casos de teste que são gerados. Além disso, além da geração de casos de teste, também é necessário especificar a ordem em que eles devem ser executados para reduzir os custos da integração e a construção de *stubs* de teste. Existem algumas contribuições que sugerem melhorias para as abordagens de geração de casos de teste mencionadas anteriormente [9, 80]. Em uma dessas melhorias, os autores adotam uma estratégia para identificar um conjunto apropriado de grafos reduzidos, focando em propriedades arquiteturais específicas. Essa abordagem é similar à abordagem proposta nessa tese, uma vez que a abordagem proposta utiliza especificações de cenários arquiteturais em CSP para construir grafos com um número reduzido de arestas. Além disso, o trabalho proposto determina a melhor ordem de execução dos casos de teste, através da análise das dependências existentes entre os elementos arquiteturais. Existem outras diferenças que devem ser destacadas, principalmente diferenças relacionadas ao foco dado pela solução proposta nessa tese para tolerância a falhas de software.

1.4.5 Verificação de Arquiteturas de Linhas de Produto Tolerantes a Falhas

Algumas contribuições propõem técnicas para verificar linhas de produto de software. A maioria das soluções existentes focam na verificação do modelo de características[32], com o intuito de aferir a consistência das escolhas realizadas (no nível de requisitos) durante a instanciação de um produto específico [11, 85]. Esses trabalhos são complementares ao trabalho proposto nessa tese, uma vez que é dado foco na verificação de PLAs que lidam com aspectos específicos de tolerância a falhas de software. O trabalho de Auerswald et al. [4] tem o mesmo objetivo que o trabalho proposto aqui no que concerne a especificação de linhas de produto de software para sistemas tolerantes a falhas. No seu trabalho, Auerswald et al. propõem uma PLA com variabilidades relacionadas a padrões para se implementar tolerância a falhas de hardware e software. Em contraste com a PLA simplificada desse trabalho, a PLA apresentada nessa tese é relacionada a decisões de

[32]Do inglês *feature model*

projeto relacionadas especificamente a tolerância a falhas de software.

Finalmente, Lutz e Gannod [73] descrevem experiências com análise arquitetural assistida por ferramentas no contexto de uma linha de produto de software de missão crítica. Os autores utilizam checagem de modelos para determinar o nível de tolerância a falhas baseado em cenários arquiteturais. Entretanto, essa abordagem considera somente propriedades relacionadas a cenários que são comuns a todos os produtos da PLA e não oferece uma maneira de reutilizar o modelo formal para verificar cenários específicos relacionados a técnicas de tolerância a falhas de software.

1.5 Estrutura da Tese

Essa tese foi escrita baseada em uma coleção de artigos científicos publicados em conferências internacionais e nacionais, simpósios e periódicos internacionais. Por essa razão, com exceção dos capítulos de introdução e conclusão (Capítulos 1 e 8), foram preservados os textos originais publicados em inglês. Para prevenir textos redundantes, especialmente sobre motivação, fundamentos teóricos e descrição de estudos de caso, os conteúdos dos artigos originais foram adaptados, com o intuito de serem inseridos sem as suas introduções, fundamentos teóricos e conclusões. No início de cada capítulo foi adicionado um prólogo indicando as referências de onde o respectivo conteúdo foi extraído na íntegra.

O restante dessa tese está organizado como segue. O Capítulo 2 apresenta alguns fundamentos teóricos sobre os conceitos considerados nessa tese e também descreve as três aplicações que foram utilizadas como estudos de caso para avaliar a abordagem proposta. O Capítulo 3 apresenta duas abstrações arquiteturais para estruturar tolerância a falhas de software. O Capítulo 4 detalha a abordagem proposta de verificação baseada em cenários arquiteturais, incluindo os moldes formais utilizados para especificar configurações arquiteturais. O Capítulo 5 descreve a geração de casos de teste baseados em modelos da arquitetura de software, de modo a identificar inconsistências entre a configuração arquitetural e o respectivo código-fonte. O Capítulo 6 apresenta um processo rigoroso que sistematiza o uso de técnicas de verificação e validação para arquitetar sistemas de software tolerantes a falhas. O Capítulo 7 apresenta um estudo de caso onde a solução proposta foi estendida para possibilitar a especificação formal e verificação de variabilidades relacionadas a arquiteturas de linhas de produto tolerantes a falhas. Finalmente, o Capítulo 8 apresenta algumas considerações finais e identifica direções de pesquisas futuras.

Capítulo 2

Fundamentos Teóricos

Este capítulo apresenta alguns conceitos de fundamentos teóricos sobre arquiteturas de software baseadas em componentes, tolerância a falhas na arquitetura de software, a notação formal utilizada na tese, teste baseado em modelos e modelos de implementação de componentes. Além disso, também são descritas três aplicações que são utilizadas como estudos de caso no decorrer da tese com o intuito de avaliar a solução proposta. O conteúdo desse capítulo foi retirado de duas publicações: um artigo publicado no *11th IEEE High Assurance Systems Engineering Symposium* (HASE 2008) [36], que foca na representação formal e verificação de cnfigurações arquiteturais; e em um artigo aceito para publicação no *Journal of Computer Science and Technology* (JCST), Special Issue on Software Engineering for High-Confidence Systems [19], que foca em teste de integração envolvendo elementos arquiteturais. Como o conteúdo do capítulo foi extraído na íntegra desses artigos, foi preservado o idioma original.

2.1 Component-based Software Architectures

The *software architecture* of a program or computing system is the structure or structures of the system, which comprises software elements, the externally visible properties of those elements (architectural components and connectors), and the relationships among them (architectural configuration) [7]. Since the software architecture represents the structure of the software system, it is a set of significant decisions about the organization of a software. Those decisions concerns the selection of the structural elements and their interfaces by which the system is composed, together with their behaviour as specified in the collaborations among those elements [66]. Moreover, since software architecture deals with the design and implementation of the high-level structure of the software, its decisions have a decisive impact into the quality attributes of the whole system, such as availability, reliability, and testability.

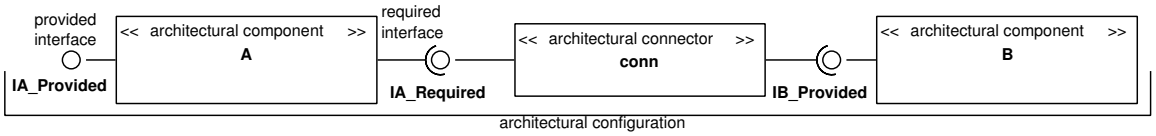


Figura 2.1: Example of an architectural configuration in UML

Figure 2.1 presents an example of an architectural configuration involving two components and a connector. *Architectural components* (e.g., A and B) are primary computational elements or data stores of a system and are defined through the list of operations that they provide (provided interfaces), as well as the operations necessary for executing their functionalities (required interfaces). To exemplify the concept of provided and required interfaces, component A of Figure 2.1 has a provided interface called IA_Provided, and a required interface called IA_Required. *Architectural connectors* (e.g., conn) are mediators of the communication and coordination activities among components. That is, they define the rules governing component interaction and specify any auxiliary implementation mechanism required. The way that components and connectors are connected together is defined by the *architectural configuration*.

Besides representing the structure of a software system, the software architecture can also represent its behaviour through architectural scenarios. A scenario is a sequence of incidents expected during the system operation, which includes environment conditions, expected stimuli and responses [95]. Basically, scenarios focus on how the system behaves to implement its functionalities. Since *architectural scenarios* represent high-level interactions amongst the architectural elements, they allow to verify and assess properties regarding important non-functional requirements, such as, reliability, availability, and scalability [54, 56, 83]. Scenarios can be used to guide the software development during the different phases since they permit multiple views of an interaction at different levels of abstraction [27]. In particular, they are relevant at the architectural level because they enhance the structural representation of the system with an abstract representation of its behaviour.

2.2 Software Fault Tolerance

Fault tolerance is the ability of a system to continue its normal operation despite the presence of faults. Software systems that can cause risks for human lives or great financial losses should be made fault-tolerant since they are considered to be critical. Because fault tolerance has a global system scope, it should be related to both architectural elements (components and connectors) and architectural configurations which implement the rules

by which they interact.

The provision of software fault tolerance relies on the existence of redundancy, which can be incorporated either implicitly or explicitly at the architectural level. Implicit redundancy refers to extra design elements associated to architectural elements, which are activated for handling internal errors. An example of implicit redundancy is the usage of exception handling for supporting error recovery. If special care is not taken when structuring the system, the normal and abnormal specifications can be entangled thus increasing system complexity. Explicit redundancy is an inherent aspect of strongly-structured systems [87] and refers to extra architectural elements, usually activated to replace failed elements. Examples of explicit redundancy are N-version programming and N-self-checking programming, which are two software fault tolerance techniques [68].

2.2.1 Explicit Redundancy Strategy

There are three main techniques for implementing software fault tolerance using design diversity [68]: recovery blocks, N-version programming, and N-self-checking programming. The *recovery blocks* technique implements a sequential execution of multiple versions of a software components such that a different version is activated after an error is detected [86]. Checkpoints are created before a version is executed for providing an operational state for recovering after a version fails. In the *N-version programming* (NVP) and *N-self-checking programming* (NSCP), the existing redundancies are executed in parallel, and then, the provided results are used to detect and tolerate errors. While NVP uses compensation through majority voting of all outputs, NSCP uses either acceptance test or comparison for detecting error [5].

2.2.2 Implicit Redundancy Strategy

Although the techniques of explicit redundancy deal with software fault tolerance, the implementation of these techniques can increase the system cost in a considerable way; as an alternative, techniques of implicit redundancy can be used. In techniques of implicit redundancy, the redundant code is responsible to implement error recovery, which is activated after the detection of an erroneous state. One of the most used ways for implementing implicit redundancy strategy is through the use of the exception handling mechanism of programming languages.

In Java, for example, `try` blocks define exception handling contexts, which represents the part of the system which is covered by error handling. In a complementary way, `catch` blocks define the handling behaviour, which is only activated in the presence of errors. Since the handlers are not considered part of the system's normal execution, they are considered implicit redundancies for implementing software fault tolerance.

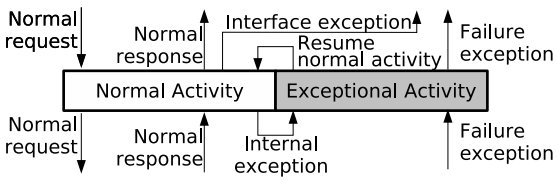


Figura 2.2: Idealised fault-tolerant component (IFTC)

2.3 Idealised Fault-Tolerant Component

Following the terminology adopted by Anderson and Lee [69], a system consists of a set of components that interact under the control of a design. Software components receive operation requests and produce responses, which can be separated into two distinct categories: *normal*, which correspond to those situations where the component has provided its normal operation satisfactorily; and *exceptional* (or abnormal), usually signalled when an error is detected and the component cannot provide the requested operation. Abnormal responses are usually called exceptions [57].

Figure 2.2 presents the idealised fault-tolerant component [69] (IFTC), a structuring concept for building fault-tolerant systems by means of exception handling techniques. An IFTC promotes separation of concerns between the normal behaviour of a system (normal activity) and its abnormal behaviour (exceptional activity), where measures for fault tolerance are implemented. After receiving a service request, an IFTC produces two types of responses: (i) normal responses; and (ii) abnormal responses. Exceptions can be classified into two different categories: *internal exceptions*, which are raised by a component in order to invoke its own error recovery measures and *external exceptions*, signalled if a component determines that, for some reason, it cannot provide its specified operation. External exceptions can be partitioned into *interface exceptions*, which are generated due to an invalid operation request and *failure exceptions*, which are due to a failure in the processing of a valid request. When internal and external exceptions are handled successfully, the system can return to provide its normal operations. In this sense, exceptions and exception handling provide a suitable framework for structuring fault tolerance activities incorporated in a system.

In order to illustrate how an IFTC can be implemented, Figure 2.3 presents the internal structure of the A architectural component of Figure 2.1 according to the IFTC principle. While the normal behaviour is realised by a normal internal component A_int, the abnormal behaviour is realised by two abnormal components: Handlers#1 and Handlers#2. The association between the error detection (into the A_int) and the exception handlers (provided by the Handlers components) is realised by the InternalConnector. The normal interface IA_Provided, which is provided by A_int, specifies the operations imple-

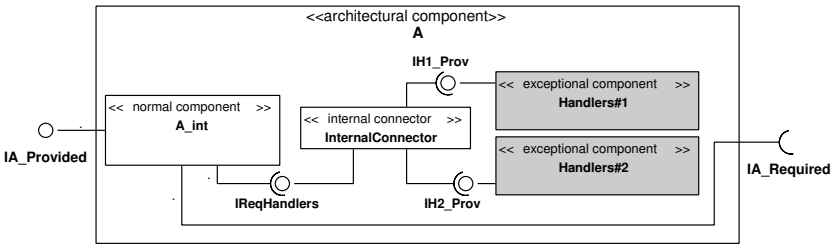


Figura 2.3: Implementation-level structure of the IFTC using UML

mented by component A, while the normal interface IA_Required contains the operations used by A, in order to implement its functionalities. The abnormal interface IReqHandlers contains the operations used by the A_int component to handle exceptions. Finally, interfaces IH1_Prov and IH2_Prov contains the exception handlers implemented respectively by Handlers#1 and Handlers#2 components.

IFTCs may be organised into layers, so that components may handle exceptions raised by components located in other layers. In this approach, the system software architecture is partitioned in layers that comprise different levels of abstraction. Ideally, each layer is responsible for handling only exceptions raised by the layer immediately below it. In the context of this thesis, software architectures that are structured with IFTCs are considered *fault-tolerant software architectures*.

When external exceptions cross the boundary between architectural elements, they are also classified as *architectural exceptions*. Since architectural exceptions affect more than one architectural element, it means that the propagation of these exceptions should follow the interaction protocol dictated by the software architecture.

2.4 Formal Notation and Verification

For overcoming the ADLs' limitations related to exception handling and discussed in Section 1, it is necessary to use a formal language that makes it possible to represent types in an explicit way, in order to distinguish different exceptions. Moreover, for representing the chaining of exception propagation, conversion and masking, the formal notation should also support the specification of scenarios involving the architectural elements.

B-Method is a general-purpose formal language based on set theory for specifying and verifying system models with explicit representation of the state, and a modular representation through the concept of refinement [1]. *Refinement* allows us to build a model gradually by making it more precise. The modularisation of the development facilitates the design and improves the scalability of the verification because it is conducted incrementally. Once the refinement is formally guaranteed, the properties verified at the

abstract level are reused at the refined level, hence do not need to be re-verified.

A limitation of the B-Method, in the context of architectural behaviour, is its inability to easily restrict the correct order for executing operations. Communicating Sequential Process (CSP) [22] is a process algebra that allows an easy representation of execution sequences, and if combined with B-Method, it compensates the aforementioned limitation [70]. As a combined solution, ProB [70] is a model checker that uses B-Method and CSP in a complementary way. In ProB, a CSP specification can be used to restrict the sequence of B-Method operations that are executed.

2.4.1 B-Method Machines

B-Method is a state-based method developed by Abrial [1] for specifying, designing and coding software systems. It is based on set theory with the axiom of choice. Sets are used for data modelling, thus allowing the definition of personalised types according to what is being modelled. B-Method has been used in industry with some success, in a number of applications ranging from the development of control systems to smart cards. As all formal methods, the B method provides a formal language to describe systems, allowing for analysis and verification of certain system properties prior to implementation. An important characteristic of B is that it covers the whole development process, from specification to implementation. In the context of this paper, we have used only the specification and verification of B-Method, not the implementation.

```
1  MACHINE counter
2
3  VARIABLES
4          value
5
6  INVARIANT
7          value : INT
8  INITIALIZATION
9          value := 0
10
11 OPERATIONS
12         inc() =
13         BEGIN
14             value := value + 1
15         END;
16
17         add(number) =
18         PRE
19             number : INT
20         THEN
21             value := value + number
22         END;
23
24         output <-- getValue() =
25         BEGIN
26             output := value
27         END
28 END
```

Figura 2.4: Example of a B-Method machine

B-Method specifications are centred on the notion of abstract machine. Abstract

machines are the units of modularisation of specifications in B-Method, and resemble modules of imperative languages. An abstract machine encapsulates data and behaviour. Data are stored in terms of *variables*, and behaviour in terms of *operations*. Figure 2.4 presents an example of an abstract machine in B-Method. This machine has a single variable (`value`), and three operations: (i) `inc()`, which increments one in the value of the; (ii) `add(number)`, which increments 'number' in the value of the variable, and (iii) `output` ← `getValue()`, which returns the value of the variable. The type of the variable is defined in terms of *invariants*. Besides defining the variable types, invariants might contain other statements in order to indicate which properties must be maintained throughout the lifecycle of the abstract machine.

Tabela 2.1: Some symbols for defining types in B-Method.

Symbol	Meaning	Symbol	Meaning
POW(A)	Powerset of `A`	A −− > B	A total function from `A` to `B`
A +− > B	A partial function from `A` to `B`	A B	`A` union `B`
A * B	Cartesian product of `A` and `B`	seq(A)	A sequence of elements of `A`

Operations are specified by means of pre-conditions and multiple assignments. Pre-conditions can be used to define general rules that should be valid before executing the operation. When there is no precondition (operations `inc()` and `getValue()`), it is assumed a "true" value (logical tautology). Assignments are used to change the value of variables, thus updating the state of the machine. For example, Lines 14 and 21 of Figure 2.4 changes the value of the machine's variable. When executing an operation, it is required that it preserves the invariants specified for the machine. Operations might have parameters and provide a return. While the types of parameters are specified as pre-conditions (Line 19 of Figure 2.4), the type of return is defined as an assignment into the operation (Line 26 of Figure 2.4). In the example of the `add(number)` operation in Figure 2.4 (Line 17 of Figure 2.4) a parameter is passed containing an integer value (INT). Examples of other symbols for defining more complex types are presented in Table 2.1. These definitions can be used into invariants and preconditions, in order to define logical expressions, mathematical relations, power sets, etc.

2.4.2 Communicating Sequential Processes (CSP)

Communicating sequential processes (CSP) is a process algebra that describes patterns of communication by algebraic expressions. The basic element of behaviour and communication in CSP is an *event*. Events may be atomic, or they can have associated data. For example `evt1` and `evt2` are atomic events, while `evt3!5` and `evt4?x` represent events

that output the value 5 and input a value represented by x respectively. A *process* is the unit which associate related events. The simplest process, `STOP`, is the one that engages in no events. Another useful process is `SKIP`, representing successful termination.

When associating events together, processes can define either sequential execution or alternative executions. A process can define sequential events by using the arrow operator ($->$), in such a way that if `PROCESS1 =` $(evt1 -> evt2 -> evt3!7)$, `PROCESS1` defines an exact sequence of events: first `evt1`, followed by `evt2` and then `evt3` outputting the number 7. A process can also define sequences of other processes; for this, it is necessary to define a list of processes separated by the semicolon operator (`;`). For example, process `PROCESS2 = (P1;P2)` behaves like `P1` until `P1` executes a final successful event (`SKIP`),at which point it behaves like `P2`.

Besides specifying sequential execution, CSP also allows the specification of alternative choices. For this, the alternative sequences should be separated by the choice operator (`[]`). For example, if `PROCESS3 = (PROCESS1 []` $(evt5 -> evt6)$, the first event to be executed could be either `evt1` or `evt5`; but after choosing one of them, the respective sequence would be followed.

2.5 Integration Testing

Once the software architecture is defined and the components are provided, it is necessary to determine whether the architectural elements are able to interoperate. This is the purpose of integration testing, which aims to reveal faults at the interfaces of architectural elements. Why is integration testing needed, given that the architecture was rigorously specified and verified? Although a rigorous development is necessary for assuring quality during development, it is not enough to guarantee the quality of the integrated system. Moreover, a rigorous specification may be misinterpreted by developers, unless code is automatically generated. But even then, some third party components, written in different languages and for different contexts, and not integrated in accordance with the architectural specification model may be the cause of incompatibilities. Instead of integrating all the system at once, which is not recommended since it incurs in high debugging costs, architectural elements should be incrementally integrated, while considering these two issues: (i) the order in which architectural elements should be integrated; and (ii) the test cases that should be applied for each incremental step.

2.6 Robustness Testing

The software can be stated as robust if it can produce dependable services even in presence of an aggressive external environment. Robustness testing is a necessary step to complement functional testing, and its role is to guarantee that the software behaviour is acceptable in the presence of internal or external failures, or stressful environmental conditions. In several robustness testing approaches, mostly based on conformance testing [50, 59, 61], the model used for representing the nominal behaviour is extended to allow the representation of behaviour in the presence of faults. A major limitation associated with such approaches is the fact that the fault model is reduced to that that can be represented in the model, and also, that can be applied by a tester. To cope with this limitation, fault injection techniques are a good complement.

Fault injection, in which faults are deliberated introduced into a system, is useful to: (i) test fault-tolerance mechanisms; (ii) evaluate measures, such as, mean time to failure and performance degradation due to error handling; and (iii) characterise the behaviour of a system in the presence of faults. There are several approaches for applying fault injection, and a good summary of these can be found in [60].

In this paper, a software-implemented fault injection technique, based on interface fault injection, will be employed for evaluating software robustness. This technique is the mostly used for robustness testing because of its flexibility and easiness to build and change. The basis of fault injection is a fault injector, which is the tool responsible for injecting faults during runtime. Although there are several tools for automating robustness testing [64, 65], in this work we use Jaca, which is a tool aimed at injecting interface faults in Java applications [74]. Since Jaca uses reflective programming, we do not need the source code of the application for injecting faults. Faults are injected at the public interface of the application by altering attributes values, methods parameters, and return values. The tool also provides mechanisms to store the results obtained from the experiments, which can then be used for evaluating robustness.

2.7 The COSMOS* Component Implementation Model

The COSMOS* model [55] is a generic and platform-independent implementation model, which uses object-oriented structures, such as interfaces, classes and packages, to implement component-based software architectures. The main objectives of the COSMOS* model are: (i) to provide traceability between the software architecture and the source code of the application; and thus (ii) to facilitate the evolution of its implementation.

Figure 2.5 illustrates the internal structure of the A_int component of Figure 2.3 following the COSMOS* model. COSMOS* defines five sub-models, which address different

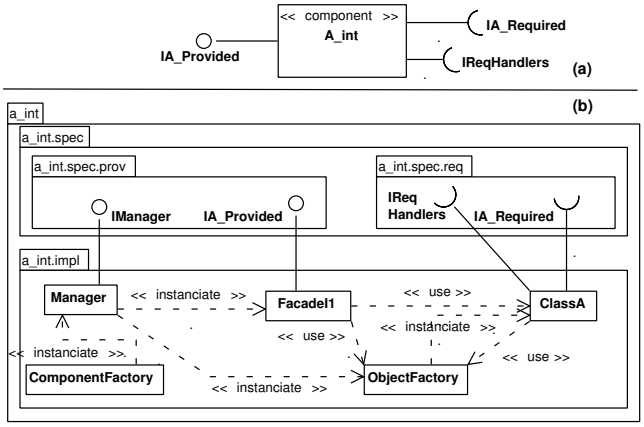


Figura 2.5: Structure of the COSMOS* implementation model in UML

aspects of component-based software systems: (i) the *specification model*, presented in Figure 2.5 (a), specifies the components using UML; (ii) the *implementation model* (Figure 2.5 (b)) explicitly separates the specification of provided and required interfaces (`spec` package) and also defines how the services provided by the component are implemented (`impl` package); (iii) the *connector model* (not illustrated) specifies the connections between components using connectors, thus enabling two or more components to be connected in a configuration; (iv) *composite component model* (not illustrated) specifies the implementation of high-granularity components, which are composed by smaller COSMOS* components; and (v) *system model* (not illustrated) defines a software component which can be deployed and executed, including reference to external dependencies, such as libraries and frameworks. Each of these models is implemented as a set of well-defined patterns which can be automatically translated to source code [103].

2.8 Description of the Applications used as Case Studies

This section presents the description of the three applications used as case studies for evaluating the work proposed in this thesis: (i) a system for controlling the extraction of minerals into a mine (Section 2.8.1); (ii) a banking system for controlling credit limits (Section 2.8.2); and (iii) a product line architecture for mobile applications (Section 2.8.3).

2.8.1 Mining Control System

In this section, it is presented a case study of a mining control system [97], which was been conducted by the author. The system requirements where specified in terms of use cases abnormal scenarios, according to the MDCE methodology [91]. Abnormal scenarios are characterised by the presence of exceptions, e.g., raising, propagation, and handling of exceptions.

The extraction of minerals from a mine produces water and releases methane gas to the air. The mining control system is used to drain mine water from a sump to the surface and to extract air from the mine when the methane level becomes high. A schematic representation of the mining system is given in Figure 11. The mining control system consists of three control stations: one that monitors the level of water in the sump, one that monitors the level of methane in the mine and another that monitors the mineral extraction. When the water reaches a high level, the pump is turned on and the sump is drained until the water reaches a low level. A water flow sensor is able to detect the flow of water in the pipe. However, the pump is situated underground and for safety reasons it must not be started or continue to run when the amount of methane in the atmosphere exceeds a safety limit. For controlling the level of methane, there is an air extractor control station that monitors the level of methane inside the mine and when the level is high an air extractor is switched on to remove air from the mine. The whole system is also controlled from the surface via an operator console that should handle any emergencies raised by the automatic system.

In order to simplify the presentation of the case study, we consider that the sensors AirFlow, WaterFlow, MethaneLevel and WaterLevel never fail, while the actuators Pump and AirExtractor are prone to failure. The abnormal behaviour of the mining control system is related to failures that can affect one of the three major activities of the system; namely, the extraction of minerals, the extraction of air and the drainage of water. These activities are summarized as follows.

- If the MethaneLevel sensor detects a high level of methane, the mineral extraction collaboration will be interrupted, and the air extraction collaboration will be initiated in order to reduce the methane level.

- If the WaterLevel sensor detects a high level of water in the sump, the mineral extraction collaboration will be interrupted, and the water extraction collaboration will be initiated to drain the sump in order to reduce the level of water.

- If both a high level of methane and a high level of water are detected concurrently, the mineral extraction collaboration will be interrupted, the air extraction collaboration will be initiated and the pump will be switched off in order to avoid explosions.

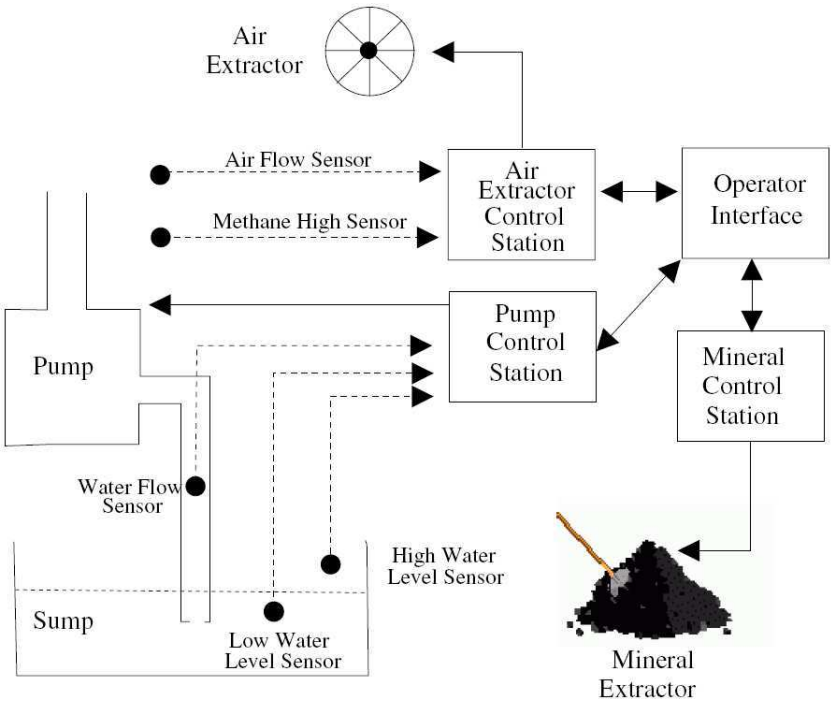


Figura 2.6: Schematic diagram of the mining control system

> If in such a case the Pump component does not switch off, the system has to raise an emergency alarm in order to evacuate the mine.

Since the extraction of minerals is closely related to the extraction of air and water, the ExtractMineral use case is extended by the use cases ExtractAir and PumpWater, and incorporates their behaviour.

The normal behaviour of the ExtractMineral use case is associated with the extraction of minerals from the mine. When the operator switches the mining control system on by means of the OperatorInterface (Figure 2.6), the mineral extraction is activated. Mineral extraction begins and proceeds until either the operator switches the mining control system off, the level of methane becomes high during mineral extraction, or the level of water becomes high during the mineral extraction. The post-condition for the normal behaviour of this use case establishes that the mineral extractor should be switched off.

The abnormal behaviour of this use case is partitioned into four scenarios. The first scenario occurs when the methane level is high and the AirExtractor is on, but no air flow is detected. The handler for this exception sounds an alarm in order to let operators know about the dangerous situation, which may cause an explosion. The second scenario happens when the level of methane is acceptable and the AirExtractor is off, but air flow is detected by the AirExtractorController. The handler for this exception informs the

OperatorInterface that the AirExtractor actuator has failed (it cannot be switched off) and that the operator should be notified. Two analogous scenarios occurs related to the level of water into the mine.

2.8.2 Banking System

This section describes a real banking system for managing loans, which was developed by employees of a Brazilian company. A critical requirement of this system regards the provision of accurate information about automatic loans and the status of the respective contracts. If erroneous information about any loan is provided, the bank may incur into high financial losses. For improving the reliability and availability of the transactions associated with the management of loans, we have defined a fault-tolerant software architecture with both implicit and explicit redundancy.

The part of the banking system that has been developed and tested is responsible for registering and controlling the delivery of chequebooks, account contracts and credit limits, and consists of seven basic functionalities:

1. Request Chequebooks. The customer requests a chequebook (in person, by phone, through the Internet, etc.).

2. Deliver Chequebooks. The system manages the delivery of previously requested chequebooks.

3. Cancel Cheques. In cases of loss, theft, or another specific reason, the customer can cancel chequebooks.

4. Process Cheques. The processing of a cheque occurs during a deposit in cheque. These cheques are restrained and processed for future payment.

5. Cancel Accounting Contract. At any time, the customer can lose the credit of his account. In these cases, the contract of the customer must be cancelled.

6. Change Loan Credit. Depending on necessity and credit conditions, the customer can receive an additional credit limit.

7. Change Business Rules. Financial rules can be changed within the bank. Such a change affects all the system functionalities that depend on the modified rule. Economic rules are established by the Central Bank of Brazil.

Operations #1-3 can be initiated either by an operator or by a customer. Operations #4-7 can only be initiated by an operator of the system. Operations Cancel Accounting

Contract, Cancel Cheques, and Change Business Rules have very strict availability requirements and should be on-line 24/7. The first two operations must be available in order to avoid illicit use of someone's credit, for example, in the case of theft of a chequebook. Operation Change Business Rules should not be unavailable for long because changes in economic rules would take a long time to be effective. This could result in the bank being fined for not adhering to rules established by the Brazilian Central Bank, for instance. The value of the fine is proportional to the time the functionality was unavailable.

The system requirements where specified in terms of use cases abnormal scenarios, according to MDCE methodology [91]. Abnormal scenarios are characterised by the presence of exceptions, e.g., raising, propagation, and handling of exceptions. We have identified 13 use cases, containing a total of 91 use case abnormal scenarios. In 74 abnormal scenarios, the final output was a normal response, what characterises error-masking scenarios. These scenarios refer to the occurrence of failure or unavailability of basic services that are considered critical by the business specialists. The masking has occurred because these critical services were considered redundant at the software architecture: database access, account and branch management. The other 17 abnormal scenarios are considered failure scenarios, since they represent failures that the software architecture is not able to mask.

Figure 2.7 presents the software architecture designed for the application, which was influenced by the existing infrastructure of the company where the system was developed. This architecture has four horizontal layers and two vertical layers. The userInterface layer implements the interaction between the system and its user. The system layer implements the functionalities specified by the use cases. The business layer basic functionalities inherent of the financial domain. The database layer provide access to the data bases. The architecture has two vertical layers, which can be accessed by any horizontal layer, had been defined additionally. The components of the util layer provides general functionalities, such as an e-mail validation and numeric format conversion, while the framework layer is considered an infrastructure layer that provides communication to external systems.

Regarding the patterns of exception control flow, we have used ATAM [7] to analyse the abnormal scenarios which are considered critical to the banking system. We have noticed that most of the errors are raised either at the database layer, or at the business layer. So, if any of such errors need an architectural handling, such as architectural reconfiguration, it has to occur at the connector between the system and the business layers (not presented in Figure 2.7). This connector is considered critical, since it is a candidate to implement architectural exception handlers. In order to guarantee that architectural exception handlers will be always activated, we have defined two patterns of exception propagation. Exceptions raised at the database and business layers have to be propagated until the connector between the system and business layers.

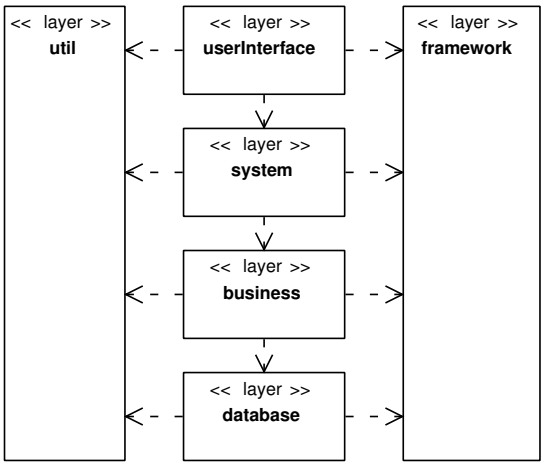


Figura 2.7: Software Architecture of the Financial System in UML

2.8.3 A Product Line Architecture of the Mobile Domain

In the following, in order to better exemplify and evaluate the contribution of this thesis, we present a case study of a real software product line called MobileMedia [51]. MobileMedia is a Software Product Line (SPL) of a mobile application that manipulates photo, music, and video on mobile devices, such as mobile phones. The application uses various technologies based on the Java ME platform, such as SMS, WMA and MMAPI.

We believe that this application is representative of how exception handling is typically used to deal with errors in real software development efforts for two reasons. First, MobileMedia encompasses a large number of exception handlers that implement diverse exception handling strategies ranging from trivial to sophisticated. Second, they present heterogeneous crosscutting relationships involving the normal code, the handler code, the clean-up actions, and other crosscutting concerns.

Figure 2.8 presents a simplified view of the feature model of MobileMedia, following the representation proposed by Ferber et al. [49]. The core features of the MobileMedia are: Create/Delete Media, Persistence (SDcard, RecordStore), Media (photo, music or video), Label Media, and View/Play Media, as well as the types of storage device. The multiple features are the types of media supported: Photo, Music, and/or Video. Finally, the optional features are: send photo via SMS (SMS for short), Copy Media, and set favourite media (Favourites). The core features of MobileMedia are applicable to all the mobile phone devices that are J2ME enabled. The optional and alternative features are configurable on selected mobile phones depending on the API support they provided. MobileMedia was developed for a family of four brands of devices, namely Nokia, Motorola, Siemens, and RIM.

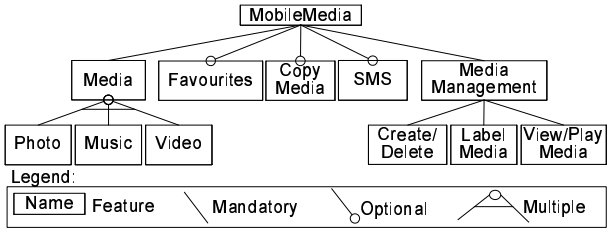


Figura 2.8: Simplified feature model of the MobileMedia SPL

The software architecture of the MobileMedia SPL is mainly determined by the use of the Model-View-Controller (MVC) architectural pattern [23]. For designing the component-based software architecture we have followed the UML Components process [32] with some adaptations. First, the layered architectural style, suggested by the UML Components process, was changed by the MVC architectural style. Second, instead of using the use case model as input to identify components, we have used the feature model of the SPL.

Figure 2.9 presents the component-based Product Line Architecture (PLA), which is composed by three layers: an interface layer, an operation layer, and a data layer. The PresentationDialog component represents the union of the Presentation and Dialog layers proposed by the UML Components process, and it is responsible for manipulating menus and user actions. The operation layer is composed of four components: VideoOperations, which implements functionalities for playing and managing video files, PhotoOperations, which implements functionalities for viewing and managing image files, AudioOperations, which implements functionalities for playing and managing audio files and MessagingOperations, which implements functionalities for sending and receiving messages. Finally, the data layer contains the services related to the management and access of storage devices (e.g. SD cards and the internal file system). This layer has a single component, called MediaManager, which is responsible for managing the media and albums data. Since the features of Audio, Photo, Video, and Messaging are optional features, the existence of different required interfaces into the PresentationDialog component allows the specification of architectural variation points. That is, during the architectural configuration, the dependencies of the PresentationDialog component can be either totally or just partially satisfied.

Since architectural elements should be self-contained entities, we have defined data types, which are classes with public attributes and no operations, containing only the information needed by other components. Instances of data types are used to exchange information between architectural elements, thus providing information hiding of the implementation classes. Examples of data types are: AlbumDataDT and MediaDataDT.

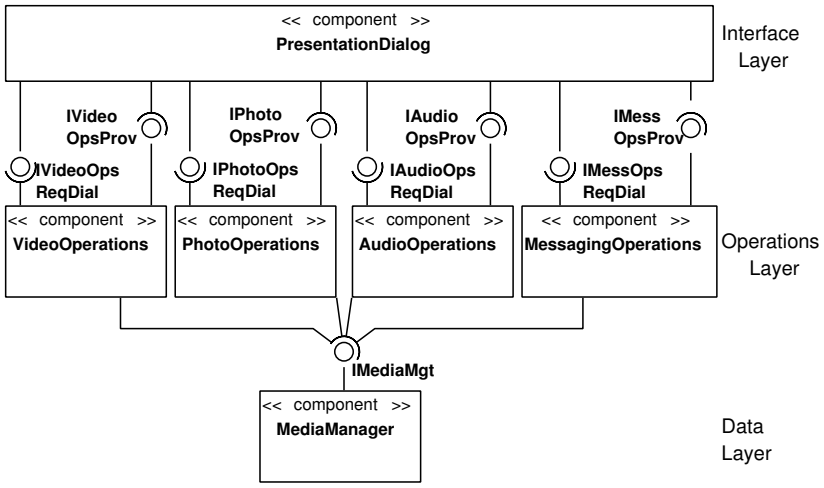


Figura 2.9: Component-based PLA for MobileMedia

Exception Control Flows at the PLA

For better illustrating the abnormal behaviour of MobileMedia, Figure 2.10 depicts an example of exception control flow. The figure presents three architectural components: MediaManager, PhotoOperations and PresentationDialog. The ellipses inside the components represent methods. A black arrow from `a()` to `b()` indicates that method `a()` calls method `b`, while a dashed arrow from `b()` to `a()` indicates that method `b()` signals an exception to `a()`. We also explicitly indicate the types of exceptions that the components raises and signals.

The explicit exception channel presented in Figure 2.10 could be defined by the tuple {{InvalidArrayFormat}, {mm.getImage()}, {}, {po.searchImage(), pd.viewImage()}, {}}. Only exception *InvalidArrayFormat* is raised in this channel, which has method `mm.getImage()` as its sole raising site and has no intermediate site. Methods `po.searchImage()` and `pd.viewImage()` are handling sites, since pluggable handlers *h1* and *h2* are bound to them, respectively. The former catches exception *InvalidArrayFormat* and maps it to exception *ImageNotFound*, whereas the later catches exception *ImageNotFound* and stops its propagation. It is important to stress that method `po.searchImage()` is not an intermediate site because it is associated to a pluggable handler. This channel does not define an exception interface, since it includes handlers for all of its exceptions. Basically, this model provides the means to specify, in a local manner, non-local information regarding exception control flows.

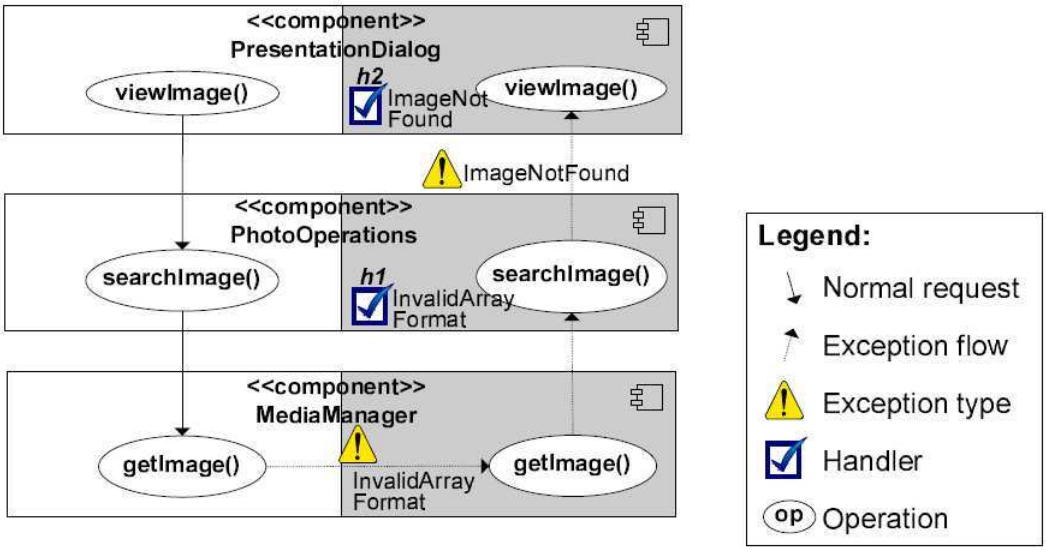


Figura 2.10: Control flow among components

Capítulo 3

Abstrações Arquiteturais para Estruturar Tolerância a Falhas de Software

Esse capítulo apresenta duas abstrações arquiteturais para projetar arquiteturas de software tolerantes a falhas. O conteúdo desse capítulo foi retirado de um artigo publicado na *2nd European Conference on Software Architecture* (ECSA 2008) [18]. Como o conteúdo do capítulo foi extraído na íntegra desse artigo, foi preservado o idioma original.

3.1 Resumo do Capítulo

A incorporação de tolerância a falhas em sistemas de software, normalmente aumenta a sua complexidade, o que consequentemente torna a sua análise uma tarefa difícil. Esse capítulo apresenta duas abstrações arquiteturais que podem ser efetivas no desenvolvimento de sistemas de software tolerantes a falhas: uma baseada em redundância arquitetural explícita, o elemento arquitetural falha-e-para[1] (HoFE); e outra abstração baseada em redundância implícita utilizando tratamento de exceções, o elemento arquitetural tolerante a falhas idealizado[2] (iFTE). Essas abstrações arquiteturais, assim como sseus detalhamento internos, podem ser instanciados em componentes e conectores concretos para projetar arquiteturas de software tolerantes a falhas. A aplicabilidade das abstrações propostas é avaliada no decorrer da tese, no contexto de estudos de caso reais e acadêmicos.

[1]Do inglês *halt-on-failure architectural element*
[2]Do inglês *idealised fault-tolerant architectural element*

3.2 Architectural Abstractions

As presented in Section 2.1, the software architecture represents the structure of the software system in a high-level way, thus being an important artefact to control the system complexity and to represent important design decisions about the organization of a software. In this context, *architectural abstractions* define specific roles to be handled by the architectural elements. Examples of important roles in the context of software fault tolerance are: error detection, error confinement, error propagation and error handling. Since the internal structure of the architectural abstractions are not available in a high-level view of the software architecture, it can be used as a basis for a top-down strategy of architectural design, thus improving the system understandability and better controlling its inherent complexity.

This section presents two examples of architectural abstractions which are used to structure software architectures of fault-tolerant systems. The Idealised Fault-Tolerant Architectural Element (iFTE) [44], presented in Section 3.2.1, implements error handling based on the exception handling mechanism. The initial version of the iFTE was extended in order to improve its internal structure and formal verification [16, 42]. The Halt-On-Failure Architectural Element (HoFE) [18], presented in Section 3.2.2, is based on crash failure.

3.2.1 Idealised Fault-Tolerant Architectural Element

The idealised fault-tolerant architectural element (iFTE) is an architectural abstraction for structuring fault-tolerant systems. This abstraction enforces the principles associated with the concept of the idealised fault-tolerant component presented in Section 2.3, and incorporates mechanisms for detecting errors, as well as handling and propagating exceptions in a structured way. The iFTE abstraction can be used for both components and connectors. The only difference is the role that each element develops in the software architecture. While components are considered the places of computation, connectors are the places of communication, coordinating the interaction between components.

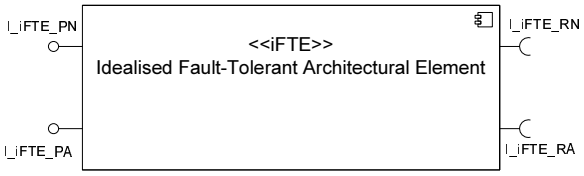


Figura 3.1: iFTE Abstraction

In order to provide a clear separation of concerns between the normal and abnormal behaviour, the iFTE defines four types of interfaces, which are presented in Figure 3.1: (i) the

Provided Normal interface (I_iFTE_PN) defines an access point for the (fault-tolerant) operations provided by the iFTE, without requesting external operations; (ii) the Provided Abnormal interface (I_iFTE_PA) defines an access point where iFTE signals its external exceptions; (iii) the Required Normal interface (I_iFTE_RN) specifies operations required by the iFTE for implementing its normal behaviour or handling exceptions; and (iv) the Required Abnormal interface (I_iFTE_RA) specifies the external exceptions that the iFTE is able to handle. In other words, while I_iFTE_PN and I_iFTE_RN are responsible for the normal behaviour, I_iFTE_PA and I_iFTE_RA are responsible for the abnormal behaviour.

The operations associated with the provided interface have a set of exceptions, which are known as provided exceptions. The operations of the required interfaces have a set of exceptions that can be caught by the iFTE, which are known as required exceptions. The exceptions signalled by an iFTE can either be internally raised, or propagated as a consequence of an exception caught from a required operation. A raised exception can either be an interface exception when an operation is wrongly requested, or a failure exception as a consequence of an internal iFTE error. Finally, for each provided operation, we may associate a set of required operations that can be invoked as part of its execution.

Relations Between Interfaces

As presented in Figure 3.2, the internal behaviour of the iFTE can be described in terms of ten relations among the four types of interfaces. These internal relations explicitly represent the existing relations between the interfaces of the iFTE. Although those relations are not scenarios describing the component's behaviour, they can be combined in order to compose them. The two first letters of the relation's name means its domain (origin) and the following two letters means its image (destination): pn for a provided normal interface, pa for a provided abnormal interface, rn for a required normal interface, and ra for a required abnormal interface. The ten internal relations of the iFTE are: (i) pnpn, which represents the normal response of a provided operation without requesting external operations; (ii) pnpai, which represents the signalling of interface exceptions; (iii) pnpaf, which represents the signalling of failure exceptions without requesting external operations; (iv) pnrn, which represents the request of external operations; (v) rnpn, which represents the normal response of a provided operation, after receiving a normal response of a required operation; (vi) rnpa, which represents the signalling of an exception after receiving a normal response of a required operation; (vii) rapn, which represents a normal response of a provided operation after receiving an exception from a required operation; (viii) rapa, which represents the signalling of an exception after receiving an exception from a required operation; (ix) rnrn, which represents a requesting of a second required operation after receiving a normal response of a previous request; and (x) rarn, which represents a requesting of a required operation after receiving an exception from a previous

request.

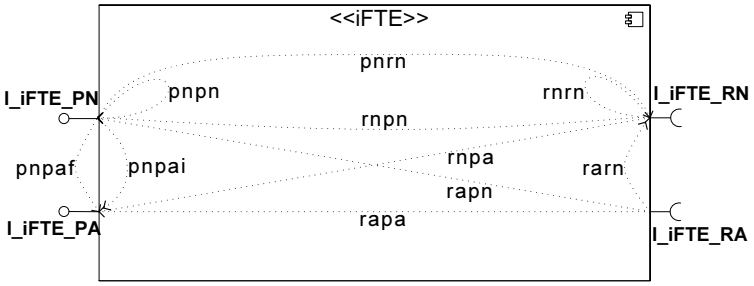


Figura 3.2: Internal Relations of the iFTE

iFTE External Behaviour

The external behaviour of the iFTE is defined through scenarios related to its external interfaces. In the context of this thesis, a scenario is defined as a sequence of events triggered by the request of an operation of the component's provided interface (I_iFTE_PN), including operation responses, other operation requests and signalling of exceptions. The scenarios are derived from the interaction rules existing between the interfaces of the iFTE. These rules involve requests of external services, the reception of the respective returns (normal or abnormal), raising of new exceptions, propagation of received exceptions, and masking of exceptions for tolerating software faults.

Since the internal behaviour of the iFTE is defined through the aforementioned ten relations, the identification of scenarios consists of combining these relations with requests and responses of operations, and with the signalling and capturing of exceptions. Based on these combinations, nine basic scenarios of the iFTE are identified, which describe its external behaviour. These nine scenarios are considered basic since they can be combined for generating other more complex normal and abnormal scenarios:

1. *internal request/response scenario*, which involves only the pnpn relation, and consists of a successful execution of a provided operation;

2. *interface exception scenario*, which involves only the pnpai relation, and consists of a wrong request of a provided operation, which causes the signalling of an interface exception;

3. *internal failure exception scenario*, which involves only the pnpaf relation, and consists of an unsuccessful execution of a provided operation, as a consequence of an internal error;

4. *external request/response scenario*, which involves the pnrn and rnpn relations, and consists of a successful execution of a provide operation that uses external operations;

5. *failure exception scenario*, which involves the pnrn and rnpa relations, and consists of an unsuccessful execution of provided operations, which raises an exception after requesting external operations;

6. *masking scenario*, which involves the pnrn and rapn relations, and consists of a successful execution of a provided operation after masking an exception caught from a required operation. In the context of fault tolerance, this scenario is extremely important, since it explicitly represents that a handler has been executed and successfully recovered the state of the architectural element, thus tolerating the fault of its server;

7. *exception propagation scenario*, which involves the pnrn and rapa relations, and consists of an unsuccessful execution of a provided operation, after catching an exception from a required operation, which could not be successfully masked;

8. *iterative request response scenario*, which involves the pnrn, rnrn, and rnpn relations, and consists of a successful execution of a provided operation after requesting more than one required operations; and

9. *iterative exception propagation scenario*, which involves the pnrn, rarn, and rnpa relations, and consists of an unsuccessful execution of a provided operation, after catching an exception and requesting a further required operation, the exception is not successfully masked.

Internal View of the iFTE.

The internal view of the iFTE (Figure 3.3) is composed of five architectural elements: (i) the Normal component implements the normal behaviour of the iFTE; (ii) the Abnormal component handles the exceptions raised by the Normal component, and those propagated from the environment of the iFTE; (iii) the Provided component acts like a bridge between the services provided by the iFTE and its environment, including the signal of exceptions; (iv) the Required component also acts like a bridge, but between the required services of the iFTE and its environment; and (v) the Coordinator connector coordinates the interaction between the four internal components. It is important to stress that the iFTE also supports the resolution of architectural mismatches. This high-level adaptation is carried out by the Provided and Required components present in Figure 3.3.

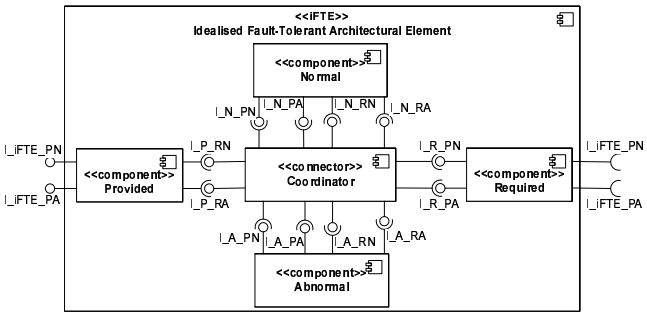


Figura 3.3: Internal View of the iFTE

For preserving the explicit separation of concerns of its external view, each internal element of the iFTE has a specific role regarding either the normal behaviour (Normal component), the abnormal behaviour (Abnormal component), or the resolution of architectural mismatches (Required and Provided components). Moreover, the definition of separate interfaces for the normal and abnormal behaviour facilitates the reuse of the normal part of the iFTE, even when the exceptions and exception handlers have to be adapted to a different architectural context. The Abnormal component is the only one that handles exceptions; the other elements are only capable of identifying erroneous conditions of its own state, raise the corresponding exceptions, and propagating it to the Abnormal component through the Coordinator. The internal architectural elements of the iFTE interact through internal interfaces, and these interfaces also enforce the separation between normal and abnormal behaviour.

Since the Normal component is responsible for providing the functionalities of the iFTE, it can be either implemented by scratch or by reusing an existing component. When reusing an existing component, it is necessary to adapt the interfaces of an existing component to the external interfaces of the Normal component. As presented in Figure 3.4, this can be done in terms of two adapters: one responsible for converting all the provided interfaces of the existing component (the NormalProvided adapter), and the other responsible for converting all the required interfaces of the existing component (the NormalRequired adapter).

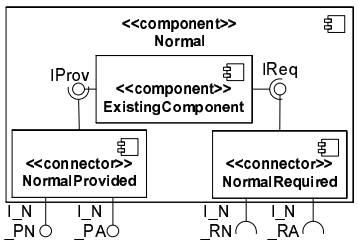


Figura 3.4: Adaptation of an existing Normal component.

iFTE Internal Behaviour.

Analysing the internal details of the iFTE, it is possible to distinguish some interactions between the internal elements, which characterise new scenarios when compared with the external view [15]. As a whole, we have identified four new scenarios involving the masking of internal exceptions, and the other scenarios of the external view.

3.2.2 Halt-On-Failure Architectural Element

The halt-on-failure architectural element (HoFE) is an architectural abstraction for the provision of error confinement and fault tolerance, and which enforces the principles associated with the *crash failures* fault model [93]. When an HoFE fails, it fails silently without producing any error signal. The HoFE abstraction defines two types of interfaces, which are presented in Figure 3.5: (i) I_HoFE_Prov defines a set of operations provided by the HoFE; and (ii) I_HoFE_Req specifies operations required by the HoFE for implementing its behaviour. It is assumed that an HoFE is able to detect failures on other architectural elements from which requests operations, e.g., by associating *time-outs* with the I_HoFE_Req interfaces.

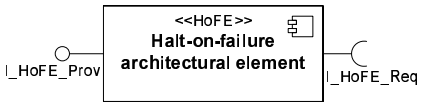


Figura 3.5: HoFE Abstraction

HoFE External Behaviour

The external behaviour of the HoFE architectural abstraction is defined through five basic scenarios. Based on these scenarios, it is possible to describe more complex scenarios of fault-tolerant software architectures that are based on the HoFE architectural abstraction:

1. *internal normal execution*, when an HoFE provides the requested services without requesting external services;

2. *internal erroneous execution*, when an HoFE fails before requesting external services;

3. *external normal execution*, when an HoFE provides the requested services after receiving the requested external services;

4. *external erroneous execution 1*, when an HoFE fails after receiving the requested external services; and

5. *external erroneous execution 2*, when an HoFE fails after failing to receive the requested external services.

It is important to stress that since the role of the HoFE abstraction is to detect errors by using explicit redundancy, the abstraction does not define any scenario of error masking.

Internal View of the HoFE

The internal view of the HoFE can be implemented using different strategies for detecting errors, usually involving explicit redundancy. Figure 3.6 presents a possible implementation using two redundant components. In this approach, the error detection is conducted by the Decider, which evaluates the results of the two executing versions of components. If there is no consensus between them, the result is considered unreliable. Since the error detection depends on both components, this implementation of the HoFE does not provide internal fault tolerance and is not able to recover from internal errors. However, if redundant HoFEs are used, fault tolerance can be implemented at the architectural level.

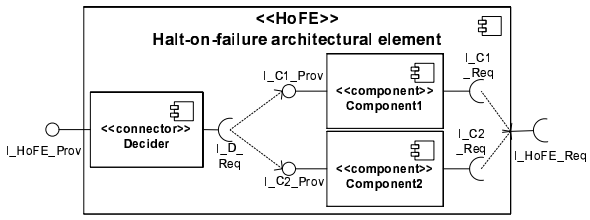


Figura 3.6: Implementing a HoFE Using Redundant Components

3.3 Summary

This chapter presented two architectural abstractions for structuring fault-tolerant software systems. First, it was presented an abstraction which implements error handling based on the exception handling mechanism, named Idealised Fault-Tolerant Architectural Element (iFTE). The external view of the iFTE allows the architect to abstract away from lower-level details related to the representation of abnormal behaviour. Moreover, its internal structure explicitly separate the normal and abnormal behaviour, thus improving the understandability and maintainability of the software architecture. The second architectural abstraction, named Halt-On-Failure Architectural Element (HoFE), is based on crash failure and defines behavioural scenarios based on the fail-stop protocol. Thus, if the architectural element fail, it is not able to provide any return in further

execution scenarios. The internal structure of the HoFE abstraction enforces the existence of two redundant components and an adjudicator, which is responsible to judge the correctness of the returns provided by the two internal components.

Capítulo 4

Verificação de Fluxos de Controle de Exceções e Tratadores Baseados em Cenários Arquiteturais

Esse capítulo apresenta a abordagem proposta para verificação de arquiteturas de software tolerantes a falhas e baseadas em componentes. A solução é composta de três artefatos complementares: (i) um arcabouço formal que pode ser instanciado com o propósito de representar formalmente as configurações arquiteturais com os respectivos cenários comportamentais; (ii) propriedades de interesse que devem ser utilizadas para verificar as configurações arquiteturais; e (iii) um processo de verificação que utiliza checagem de modelo[1] para identificar erros no modelo formal da arquitetura. O conteúdo desse capítulo foi retirado de dois artigos publicados, sendo um deles no *3rd Latin American Symposium on Dependable Computing* (LADC 2007) [15], cujo foco é a representação formal e verificação dos elementos arquiteturais, e outro artigo publicado no *11th IEEE High Assurance Systems Engineering Symposium* (HASE 2008) [36], cujo foco é na representação formal e verificação de configurações arquiteturais. Como o conteúdo do capítulo foi extraído na íntegra desses artigos, foi preservado o idioma original.

4.1 Resumo do Capítulo

Esse capítulo detalha as atividades de verificação parte da abordagem rigorosa e arquitetural proposta. Nessa abordagem, a verificação formal se baseia nos conceitos de abstrações arquiteturais e cenários, de forma a reduzir a complexidade da arquitetura de software e restrições de comportamento derivadas dos cenários arquiteturais especificados. Essas

[1] Do inglês *model checking*

restrições têm o objetivo de reduzir a explosão combinatória de estados durante a execução da verificação, o que aumenta a escalabilidade da solução proposta. Os cenários arquiteturais, que são especificados e verificados formalmente, descrevem tanto o fluxo de controle de exceções, quanto o comportamento dos tratadores de exceções envolvendo os elementos arquiteturais (componentes e conectores). O processo de verificação é conduzido utilizando a ferramenta ProB, que combina o uso de teoria de conjuntos matemáticos (B-Method) e álgebra de processos (CSP). Detalhes sobre os formalismos B-Method e CSP estão disponíveis na Seção 2.4 do Capítulo 2. A aplicabilidade da solução proposta foi avaliada através de um estudo de caso de uma aplicação bancária, que foi apresentada na Seção 2.8.2 e é utilizada para exemplificar e avaliar as atividades de especificação e verificação formal.

4.2 Formal Specification: Architectural Elements, Configuration and Scenarios

4.2.1 Specifying Architectural Elements

To facilitate the formal specification of the architectural elements, we have defined frameworks of the two architectural abstractions presented in Section 3.2 using B-Method and CSP. These frameworks can be reused and instantiated for specifying the specific architectural elements of a particular architectural configuration. In the context of the iFTE, there are two kinds of frameworks: the *iFTE abstract framework* [16, 36], which specifies the external behaviour of the iFTE, and the *iFTE detailed framework* [15], which includes its internal structure. The formal specification of both the iFTE and HoFE models are instances of the architectural configuration framework presented in Section 4.2.2. In this section we will present the iFTE and HoFE abstract frameworks.

Specify iFTE Abstract Elements

Figure 4.1 presents part of the B-Method machine of the OperationsAccount1 component of Figure 6.7. This machine explicitly represents the iFTE abstraction in terms of three basic features: its *interfaces*, through the `oa1_Interfaces` set (Line 4); its provided and required *operations*, through the `oa1_Operations` set (Line 5); and its provided and required *exceptions*, through the `oa1_Exceptions` set (Line 6). Besides the basic features of the iFTE abstraction, the B-Method machine also represents the control information used for the verification. The B-Method machine uses the elements of the `oa1_Events` set (Line 8) to represent all the events that can be associated with the interfaces of an iFTE. Essentially, these are the events that are used to define the behavioural scenarios of the

architectural elements.

```
1  MACHINE oa1
2  /* ================================= */
3  SETS
4          oa1_Interfaces = {i_oa1_pn, i_oa1_rn, i_oa1_pa, i_oa1_ra};
5          oa1_Operations = {oa1_getCreditLimit, oa1_confirmLoan, ba_getAccountBalance, ...};
6          oa1_Exceptions = {ServiceUnavailable, DBFailure, FatalError, ...};
7
8          events = {request, normalResponse, abnormalResponse}
9  /* ================================= */
10 VARIABLES
11 .../*declaration of variables*/
12 /* ================================= */
13 INVARIANT
14         /*interface category*/
15         oa1_pnInterfaces:POW(oa1_Interfaces) &
16         oa1_paInterfaces:POW(oa1_Interfaces) &
17         oa1_rnInterfaces:POW(oa1_Interfaces) &
18         oa1_raInterfaces:POW(oa1_Interfaces) &
19
20         /*interfaces - operations*/
21         oa1_pnOperations:oa1_pnInterfaces +-> POW(oa1_Operations) &
22         oa1_rnOperations:oa1_rnInterfaces +-> POW(oa1_Operations) &
23         /*interfaces - exceptions*/
24         oa1_paIntExceptions:oa1_paInterfaces +-> POW(oa1_Exceptions) &
25         oa1_raIntExceptions:oa1_raInterfaces +-> POW(oa1_Exceptions) &
26
27         /*exceptions - operations*/
28         oa1_pnOpExceptions:oa1_Operations +-> POW(oa1_Exceptions) &
29         oa1_rnOpExceptions:oa1_Operations +-> POW(oa1_Exceptions) &
30 ...
31 /* ================================= */
32 OPERATIONS
33 exception <-- oa1(interface, operation, eventType) =
34 PRE
35     interface : oa1_Interfaces &
36     operation : oa1_Operations &
37     eventType : events &
38     ...
39 THEN
40     ...
41 END
```

Figura 4.1: B-Method machine of the OperationsAccount1 iFTE component

After representing the basic features of the iFTE through sets, it is necessary to categorise and relate them together using respectively, subsets and relations. The subsets are defined using the reserved word `POW`, which indicates the type of a power set. Lines 15 to 18 present the subsets that categorise the interfaces of the `oa1_Interfaces` set. Each interface can only participate in one of these subsets, which is guaranteed through an integrity property of the model that will be presented in Section 4.3.2. After categorising the interfaces, they have to be associated to either operations or exceptions, depending if they are normal or abnormal interfaces, respectively. These associations are made through relations, which are defined in Lines 21, 22, 24 and 25. These relations also capture whether the operations and exceptions are provided or required. The first relation (Line 21) is from `oa1_pnInterfaces` to a power set of `oa1_Operations`. The provided and required exceptions are associated to the abnormal interfaces of the iFTE through the relations presented in Lines 24 and 25. Moreover, the exceptions are also associated to the operations of the iFTE through the relations presented in Lines 28 and 29. Besides the basic relations already presented, there are other 16 variables, 10 for representing the

internal relations of the iFTE, defined in Section 3.2.1 (Figure 3.2), and other six to store information used during the verification process (e.g., the traceability of the events, and the exceptions that where caught by the iFTE in the current state of the model checking). Finally, the B-Method machine defines an operation for representing the occurrence of an event (Lines 33 to 40). An event is characterised by the values of three arguments: (i) the interface through which the event has occurred (`interface`); (ii) the operation that the event refers to (`operation`); and (iii) the type of the event (`eventType`), which is one of the possible types declared in the `oa1_Events` set (Line 8). When the `eventType` is `abnormalResponse`, the type of the exception related to the event is represented in `‘‘exception’’`. When the event does not provide an abnormal response (e.g., a request or a normal response), the value of `‘‘exception’’` is an empty set (ϕ).

Specify HoFE Abstract Elements

Figure 4.2 presents part of the B-Method machine of the HoFE abstract model for the OperationsAccount component of Figure 6.8. A B-Method machine explicitly represents an HoFE in terms of its *interfaces*, through the `oa_Interfaces` set (Line 4) and its provided and required *operations*, through the `oa_Operations` set (Line 5). The B-Method machine represents three types of events (request, response and halt) through the `oa_Events` set (Line 7). Essentially, these events are used to define the behavioural scenarios of an HoFE.

```
1  MACHINE oa1
2  /* ================================= */
3  SETS
4          oa1_Interfaces = {i_oa1_Prov, i_oa1_Req};
5          oa1_Operations = {oa1_getCreditLimit, oa1_confirmLoan, ba_getAccountBalance, ...}
6
7          events = {request, response, halt}
8  /* ================================= */
9  VARIABLES
10 .../*declaration of variables*/
11 /* ================================= */
12 INVARIANT
13         /*interface category*/
14         oa1_provInterfaces : POW(oa1_Interfaces) &
15         oa1_reqInterfaces : POW(oa1_Interfaces) &
16
17         /*operations*/
18         oa1_opsProvInt : oa1_provInterfaces +-> POW(oa1_Operations) &
19         oa1_opsReqInt : oa1_reqInterfaces +-> POW(oa1_Operations) &
20
21         /*operations that can halt*/
22         oa1_operError : POW(oa1_Operations) &
23 ...
24 /* ================================= */
25 OPERATIONS
26 oa1(interface, operation, event) =
27    PRE
28    interface : oa1_Interfaces &
29    operation : oa1_Operations &
30    event : events &
31    ...
32 THEN
33    ...
34 END
```

Figura 4.2: B-Method machine of the OperationsAccount HoFE component

After representing the basic features of the HoFE abstract model through sets, it is necessary to categorise and relate them by means of B-Method variables. Lines 14 and 15 present the variables that categorise the interfaces of the `oa_Interfaces` set. The formal model guarantees, through properties, that each interface can be either provided or required, not both at the same time. Operations have to be associated with either a provided or a required interface. These associations are made through relations from `oa_Interfaces` to a power set of `oa_Operations`, (Lines 18 and 19). The operations that can halt should also be informed into the formal model (Line 22) Other nine variables are defined by the machine: four for representing the HoFE internal relations involving requests and responses between provided and required interfaces, and five to store information used during verification (e.g., the traceability of the events, and the operations which halted after a request), which are not shown in Figure 4.2. Finally, the B-Method machine defines an operation for representing the occurrence of an event (Lines 26 to 34), which is characterised by an interface and an operation (`interface` and `operation` arguments respectively). The type of the event is represented through the argument `event`.

4.2.2 Specifying Architectural Configuration and Abnormal Architectural Scenarios

The architectural configuration should be formally specified in order to contextualise the architectural elements used in a software architecture. The specification of the architectural configuration framework is composed of two complementary artefacts: (i) architectural configuration specification (in B-Method), which represents the architectural elements and the dependencies between required and provided interfaces; and (ii) architectural scenarios specification (in CSP), which represent the abnormal behaviour of the system.

Figure 4.3 presents part of the architectural configuration framework, after its instantiation to the banking system's software architecture (Section 2.8.2). We illustrate the instantiation of the architectural configuration framework in the context of the iFTE-based architecture presented in Section 6.3.2 (Figure 6.7), but this procedure is similar for any architectural abstraction. Differently from the specification of the architectural elements, the architectural configuration contains various architectural elements (iFTEs and non-iFTEs), which are represented by the `bank_ArchElements` set (Line 4). Besides, the full specification of each architectural element is incorporated into the software architecture, i.e., the B-Method machines of the iFTEs and non-iFTEs are imported by the architectural specification (Line 8). For example, the `us` word present in Line 8 means that a B-Method machine called `us.mch`, which represents the UserSession architectural component, is included by the `bank` machine.

```
1  MACHINE bank
2  /* ================================= */
3  SETS
4      bank_ArchElements = {us,alarmS,accS,oa1,oa2,ba,...};
5  ...
6  /* ================================= */
7  EXTENDS
8      us, alarmS, accS, oa1, oa2, ba, ...
9  /* ================================= */
10 VARIABLES
11 .../*declaration of variables*/
12 /* ================================= */
13 INVARIANT
14     bank_ifteArchElements:POW(bank_ArchElements) &
15     bank_nonIfteArchElements:POW(bank_ArchElements) &
16
17     bank_archConfiguration:bank_reqInterfaces +-> POW(bank_provInterfaces) &
18     bank_provExcep_reqExcep:bank_provExceptions +-> POW(bank_reqExceptions) &
19 ...
20 /* ================================= */
21 OPERATIONS
22 exception <-- bank(from, to, interface, operation, eventType) =
23 PRE
24     from: bank_ArchitecturalElements & to: bank_ArchitecturalElements &
25 ...
26 THEN
27     ...
28 END
```

Figura 4.3: Partial architectural configuration of the banking system using B-Method

Using two complementary subsets, the architectural elements are then classified in iFTEs (`bank_ifteArchElements`, Line 14) and non-iFTEs (`bank_nonIfteArchElements`, Line 15). An integrity property was defined for guarantee that $bank\_ifteArchElements \cup bank\_nonIfteArchElements = arch\_ArchElements$. The properties verified are presented in Section 4.3.2. Further, the architectural elements are interconnected in terms of an architectural configuration, which is defined through a relation between required and provided interfaces (`bank_archConfiguration`, Line 17). In addition, to contextualise the provided and required exceptions during their propagations, another relation is defined (Line 18). Finally, there is a B-Method operation for representing events involving a pair of architectural elements (Lines 20 to 28). Besides the three parameters that are also present in the iFTE abstract model (`interface`, `operation`, and `eventType`), Line 22 shows that the architectural configuration framework also identifies the architectural element that has originated the event (`from`), as well as its respective destination (`to`).

In order to define the possible execution scenarios involving architectural elements, it is defined a behavioural specification in CSP. This complementary specification states the possible sequence of events that can occur at the architectural configuration. Figure 4.4 presents the behavioural specification of the software architecture. In order to combine the two parts of the formal model, each B-Method operation corresponds to a channel in CSP followed by the proper parameters. For example, in Line 4, `bank.us.accS.i_us_rn.accS_getCreditLimit.request` corresponds to the execution of the ‘‘`bank`’’ B-Method operation, with the following values of arguments: “`from` = `us`”, “`to` = `accS`”, “`interface`

= i_us_rn", "operation = accS_getCreditLimit", and "eventType = request". Besides, using an external choice operator ([]), the CSP specification defines all possible alternatives of execution.

```
1  MAIN = US;;
2
3  -- US
4  US = (bank.us.accS.i_us_rn.accS_getCreditLimit.request -> bank.us.accS.i_accS_pn.accS_getCreditLimit.
       request -> ACCS_IFTE(...)) [] MAIN;;
5
6
7  -- ACCS
8  ACCS_IFTE(...) = accS.i_accS_pn.accS_getCreditLimit.request -> ACCS_IFTE_INT(...);;
9
10 --ACCS_INTERNALLY
11 ACCS_IFTE_INT(...) =
12     (accS.i_accS_pa.accS_getCreditLimit.abnormalResponse -> bank.accS.us.i_accS_pa.accS_getCreditLimit.
          abnormalResponse -> bank.accS.us.i_us_ra.accS_getCreditLimit.abnormalResponse -> MAIN) []
13     (accS.i_accS_rn1.oa1_getCreditLimit.request -> bank.accS.oa1.i_accS_rn1.oa1_getCreditLimit.request
          -> bank.accS.oa1.i_oa1_pn.oa1_getCreditLimit.request -> OA1_IFTE(...));;
14
15
16 -- OA1
17 ...
```

Figura 4.4: Partial specification of the architectural scenarios in CSP

Since it defines the sequence of events, the behavioural specification is also responsible for the synchronisation between the architectural events and the internal events of the iFTEs. For example, Line 4 states that the internal events of the AccountSession component (ACCS_IFTE) are activated only after an architectural request to ACCS_IFTE's provided interface (bank.us.accS.i_accS_pn.accS_getCreditLimit.request). After receiving a request through its provided interface (Lines 8), the connector either fails and raises an exception to the caller (Line 12), or requests an external service to OperationsAccount1 component OA1_IFTE (Line 13). Note that the sequence of events involves both internal events and architectural ones, requiring synchronisation whenever one of them occurs.

4.3 Verifying Architectural Elements and Configuration

The verification of software architectures requires to take into account properties regarding the individual architectural elements and their configurations. The properties associated with the architectural elements should allow to check whether they are consistent against the properties identified for the iFTE abstraction, including the scenarios of raising, catching, and signalling exceptions. On the other hand, properties associated with the software architecture should allow to check whether the architectural configuration follows the composition rules dictated by the architectural elements, including exception control flow and the correct context for handling exceptions. Section 4.3.1 presents the steps followed during the formal verification, including an explanation of how the model

checker detects inconsistencies in the model. Then, Section 4.3.2 presents the properties of interest, which are verified for the iFTEs and the software architecture.

4.3.1 Verification Process

We have defined a single process to be followed when verifying properties associated with the system abnormal behaviour. This process is composed of four sub-activities to be executed sequentially, which should be employed in the verification of both architectural elements and architectural configuration:

1. **Sub-activity 1**: (*verify integrity consistency*) comprises the syntactical analysis of the formal model, as well as the integrity of its state;

2. **Sub-activity 2**: (*verify general abnormal scenarios violations*) verifies whether there exists violations of the specified scenarios, by checking the existence of impossible scenarios;

3. **Sub-activity 3**: (*verify specific abnormal scenario violation*) checks for the existence (or not) of desired patterns of exception control flows, and handlers; and

4. **Sub-activity 4**: (*verify user-defined properties*) analyses application specific properties.

The properties of interest associated with Sub-activities 1 and 2 are specified as *assertions*, i.e., formal rules that the model should be compliant with. If one of these rules is violated, an error message and a counterexample are presented by the model checker. Differently, the properties associated with Sub-activity 3 are specified as *definitions*, i.e., formal state patterns that the model checker tries to find (not their violations). If there is a state that satisfies those patterns, a warning message and an example are presented by the model checker. Finally, since the properties of Sub-activity 4 are defined by the user, it is possible to use both assertions and definitions.

For executing the verification, the ProB model checker calculates all the possible states of the formal model at runtime. For each state, it checks if it violates any assertion, and if it satisfies any definition. To reduce the statespace of the model checking, we have adopted a scenario-based approach, where architectural scenarios in CSP are used to restrict the way that the B-Method operations change the state of the machine. To support the verification of behavioural properties, the B-Method machines define the `sequenceHistory` variable, which stores the sequence of events that were executed to achieve the current state. As presented in Section 4.2, these events consists on requests and responses of operations.

4.3.2 Properties of Interest

Each sub-activity presented in Section 4.3.1 has a particular goal of verification. Properties of integrity consistency (Sub-activity 1) are grouped in two categories: (i) *normal integrity properties*; and (ii) *abnormal integrity properties*. Tables 4.1 and 4.2 present, respectively, the properties of integrity consistency verified for the iFTEs and for the architectural configuration.

Tabela 4.1: Integrity consistency properties of the iFTEs.

CATEGORY 1. *Normal integrity properties.* Verifies the consistency of the iFTE internal relations with regards to its normal behaviour	
#	Property Name & Description
1	*Internal relations integrity property.* An operation cannot be considered at the same time as internal and external
2	*Provided and required operations integrity property.* The sets of provided and required operations should be disjoint
3	*Interfaces' classification property.* All the interfaces have to be classified in exactly one of the four types defined by the iFTE abstract model
4	*Interface and operation consistency.* The associations between an interface and its operations have to be consistent with their types (provided or required)
5	*Normal and abnormal interfaces integrity property.* Interfaces classified as normal can neither signal nor catch exceptions, and the interfaces classified as abnormal can neither receive operation requests nor provide normal returns
6	*Cardinality of relations property.* Verifies if the cardinalities of the relations remain correct for every architectural element
CATEGORY 2. *Abnormal integrity properties.* Verifies the consistency of the iFTE internal relations with regard to its abnormal behaviour	
#	Property Name & Description
1	*Provided and required exceptions integrity property.* The sets of provided and required exceptions should be disjoint
2	*Interface and raised exceptions integrity property.* The sets of interface and failure exceptions should be disjoint
3	*Prevention of unused provided exceptions property.* Every provided exception should be signalled by at least one provided operation
4	*Masking correctness property.* Only maskable exceptions can be masked by the iFTE, any other exception that would be masked is considered a design failure
5	*Abnormal interfaces and operations integrity property.* The set of exceptions associated to an abnormal interface has to be equivalent to the set of exceptions associated to its operations

Regarding the properties of scenarios violations, it is necessary to verify for each architectural element, if its scenarios are consistent with the nine basic scenarios defined by the iFTE abstraction. For this, we have defined nine assertions, one for each scenario presented in Section 3.2.1. To improve the readability of the text, these properties are not presented in the chapter. In a complementary way, the scenarios of the software architecture are also verified, considering the interaction involving architectural elements.

Tabela 4.2: Integrity consistency properties of the architectural configuration.

CATEGORY 1. *Normal integrity properties.* Verifies the consistency of the interconnections between the architectural elements that are associated with the system normal behaviour	
#	`Property Name & Description`
1	*Legitimacy of the architectural dependencies property.* A required interface cannot depends on a provided interface of the same architectural element
2	*Normal dependency correctness property.* For every required operation, it is necessary to have a correspondent provided operation to be activated
3	*Operation usage property.* Every operation declared in the architectural configuration should be associated to an interface
4	*Interface usage property.* Every interface declared in the architectural configuration should be associated to an architectural element
5	*iFTE decomposition property.* For each iFTE, an iFTE abstract model should be defined
6	*Operation mapping correctness.* The operations associated to iFTEs in the architectural configuration should have a correspondent operation into the respective iFTE abstract model
7	*Architectural elements' classification property.* Each architectural element has to be classified either as iFTE or as non-iFTE, not both at the same time
CATEGORY 2. *Abnormal integrity properties.* Verifies the consistency of the relations among architectural elements that are related with the system abnormal behaviour	
#	`Property Name & Description`
1	*Exception usage property.* Every exception should be associated to at least one operation
2	*Abnormal dependency correctness property.* Should exists a mapping between the required and provided exceptions of two connected interfaces
3	*Exception mapping correctness property.* Every architectural exception associated to iFTEs should have a correspondent exception into the respective iFTE abstract model

For this, we have defined five properties, which are presented in Table 4.3.

Regarding the properties of detailed behavioural analysis, since they are application-specific, there is no predefined property. To exemplify such properties we shall use the following (critical) scenario of exception control flow: *"when there is a failure in the* Database1 *component, a* `DBFailure` *exception should be raised. In this case, the only element capable of handling this exception is the* BusinessAccount *connector, since neither the* DB1Session *connector, nor the* AccountManagement1 *component are capable of reconfiguring the system for tolerating the fault"*. To guarantee that `DBFailure` exception will always be caught by BusinessAccount, we have defined properties of detailed behavioural analysis to identify any scenario that either DB1Session, or AccountManagement1 swallow `DBFailure` exception. If these properties are violated, it means that the formal model is inconsistent with the abnormal behaviour that is expected, and should be fixed. The same occurs when there is a failure in the Database2 component.

The assertions and definitions that constitute the properties are specified in B-Method.

Tabela 4.3: Abnormal scenarios violations properties of the architectural configuration.

#	Property Name & Description
1	*Deadlock freedom property.* The architectural configuration has to be free of deadlocks
2	*Communication uniformity property.* The architectural elements have to communicate each other in a call/return way (response after request)
3	*Dependence integrity property.* An architectural element a can only request services of another architectural element b if a depends on b
4	*Request/response property.* Every response event should refer to the same source and destination elements, as well as the same operation that was immediately requested
5	*Exception control flow property.* When an exception is propagated from an architectural element to another, the provided exception of the first element should be mapped to an equivalent required exception of the second element

For example, in Table 4.2, Property 7 of normal integrity was specified through the following assertion: $bank\_ifteArchElements \cup bank\_nonIfteArchElements = arch\_ArchElements$. So, in case it is violated, the model checker presents a counterexample that shows the violation.

Regarding the execution of the model checking, using an Intel Pentium 4 computer with 1GB of RAM, the verification process took approximately 55 minutes for the architectural configuration, and more 15 minutes to verify all the architectural elements alone. During the verification of the software architecture, the model checker has detected deadlocks, and the violation of properties. An example of the former was the deadlock caused by omitting the declaration that the AccountManagement1 component is able to propagate the exception DBFailure. Since AccountManagement1 is not able to fully mask this exception, a deadlock occurs whenever it catches this exception from the DB1Session connector. Concerning the compatibility of exception control flow properties, one of the identified violations was caused by the absence of two handlers (required exceptions) in the UserSession component. In both cases, the counterexamples helped us to identify the cause of the violations, and after fixing the formal model, all properties were satisfied. Thus, the rigorous process has helped to identify and correct design faults of the architecture in earlier stages of the software development. Although most of these problems were simple to correct, if they were left to be corrected in the later phases of the development, it would have been much harder.

4.4 Evaluation

Modelling of the banking system has been facilitated through the use of the iFTE architectural abstraction as building block of its architecture, instead of using general UML software components. The reuse of the formal frameworks for specifying architectural elements and configurations has been effective, since only 13% of their formal specification,

approximately, is affected by changes. So, 87% of the frameworks presented in Section 4.2 were reused straightforward during specification of the banking system. For example, the information updated for the architectural elements were the names of interfaces, operations and exceptions (Lines 4 to 6 of Figure 4.1) and initialization of variables (Lines 23 to 31 of Figure 4.1). The pre-conditions and behaviour of the B-Method operation (not presented in Figure 4.1) where preserved. Moreover, most of the formal properties were also reused and the modified properties where instantiated from a template, in order to represent specific interfaces, operations and exceptions. It was also clear that the explicit separation between the structural (B-Method) and behavioural (CSP) specifications has facilitated the specification of architectural elements based on the iFTE. Since the architectural elements have their own B-Method machines, this enabled to represent a software architecture containing elements based on different abstractions, e.g., the explicit inclusion of B-Method machines for iFTEs and non-iFTEs as shown in Figure 4.3. On the other hand, the use of CSP for specifying scenarios has allowed to represent complex abnormal architectural scenarios, e.g., the error propagation from the DataBase1 component to the BusinessAccount connector, followed by an exception handler that forces a second try using redundant resources of the software architecture.

Regarding the formal verification of the software architecture, first of all, we notice that the properties of interest initially identified for the architectural abstraction could be re-used on different architectural elements, thus facilitating the verification process. Moreover, since each architectural element can have an internal architecture, the software architecture can be incrementally specified and verified, which helps to control the complexity and improves the scalability of the verification approach. Another advantage comes from the combination of B-Method and CSP, which facilitates the verification of both structural and behavioural properties, e.g., properties regarding the architectural reconfiguration of the BusinessAccount connector when tolerating faults.

Finally, for evaluating the scalability of the proposed solution, we have compared our results with the results obtained when verifying the same system using the Aereal framework [28]. As it is documented in literature [28], after generating the Alloy specification of the banking system with Aereal, the model checker could not reach the end of the verification because it ran out of memory in a computer with 1GB of RAM memory. We believe that there are two reasons for the scalability improvement. First, the usage of the iFTE architectural abstraction has reduced the number of architectural elements, since the components for handling exceptions are encapsulated into the iFTEs. Second, the state space of verification is limited to the architectural scenarios specified for the application.

4.5 Summary

This chapter presented the verification approach, which is part of the rigorous architectural solution proposed in this thesis. First, it was presented a formal framework that can be instantiated in order to formally represent architectural configurations and scenarios. This framework is based on architectural abstractions (see Section 3.2), which are used to specify specific architectural elements (components and connectors) of a particular architectural configuration. Second, properties of interest were also defined; these properties represent important characteristics of fault-tolerant architectures and can be used to verify architectural configurations. Finally, a verification process have been proposed aiming to systematise the identification of errors in the software architecture by using model checking.

Although the verification approach presented here improves the quality of the software architecture, it does not guarantee the absence of faults into the model, neither that the software's source code satisfies the properties verified in the model. A possible solution to overcome the verification's limitations is the generation of model-based test cases, whose objective is to assess the consistency between the system's source code and its previously verified software architecture. The generation of model-based test cases are discussed in Chapter 5.

Capítulo 5

Validação de Fluxos de Controle de Exceções e Tratadores Baseados em Cenários Arquiteturais

Esse capítulo detalha as atividades relacionadas a teste baseado em modelos utilizada para gerar casos de teste de unidade, de integração e de robustez, a partir dos modelos formais da arquitetura de software. O conteúdo desse capítulo foi retirado de três publicações: (i) um artigo publicado no *3rd Latin American Symposium on Dependable Computing* (LADC 2007) [15], que focou na geração de casos de teste de unidade para os elementos arquiteturais, a partir dos cenários e propriedades das respectivas abstrações arquiteturais; (ii) um artigo aceito no *Journal of Computer Science and Technology (JCST), Special Issue on Software Engineering for High-Confidence Systems* [19], que foca em testes de integração envolvendo a configuração entre os elementos arquiteturais; e (iii) um artigo submetido (aguardando resultado) ao *4th Latin American Symposium on Dependable Computing* (LADC 2009) [14], que foca na geração de casos de teste de robustez baseada em abstrações arquiteturais para tolerância a falhas. Como o conteúdo do capítulo foi extraído na íntegra desses artigos, foi preservado o idioma original.

5.1 Resumo do Capítulo

Esse capítulo detalha as atividades de validação executadas pelo processo rigoroso e centrado na arquitetura proposto nessa tese. Em termos gerais, é apresentado aqui como os casos de teste podem ser gerados a partir dos modelos formais da arquitetura de software, que inclusive pode ser automatizado em sua maior parte. É importante destacar que as atividades de teste baseado em modelos adotadas não são o principal foco de contribuição desta tese. Ao invés de propor soluções originais, abordagens existentes foram adaptadas

com o intuito de integrá-las aos modelos formais apresentados no Capítulo 4. Três tipos de casos de teste são gerados: (i) casos de teste de unidade, cujo objetivo é aferir a consistência dos elementos arquiteturais em relação aos cenários especificados pelas abstrações arquiteturais; (ii) casos de teste de integração, cujo objetivo é prevenir divergências arquiteturais através da aferição da consistência da interação entre elementos arquiteturais em relação ao protocolo de interação definido pelas abstrações arquiteturais e configuração arquitetural da aplicação; e (iii) casos de teste de robustez, cujo objetivo é aferir a consistência do código-fonte em relação ao comportamento esperado da arquitetura de software quando na presença de falhas. Por essa razão, o teste de robustez também é conhecido como teste por injeção de falhas.

Para possibilitar a especificação de casos de teste em diferentes níveis de abstração, sem a necessidade de ter acesso ao código-fonte, foi adotada uma estratégia caixa cinza baseada tanto nos cenários, quanto na estrutura da arquitetura de software e iFTEs. Os casos de teste gerados podem ser utilizados para validar o software em relação aos seus requisitos funcionais e não-funcionais. Em relação aos requisitos funcionais, a abordagem possibilita a especificação de cenários específicos de aplicação para representar o comportamento esperado do sistema. Além disso, a abordagem proposta também utiliza a informação estrutural oferecida pela arquitetura de software para avaliar o sistema em relação aos seus requisitos não-funcionais. A validação é apoiada pela abordagem proposta de quatro formas complementares. Primeiro, casos de teste de unidade podem ser gerados para a aplicação inteira, se considerar o sistema como sendo um componente de software. Segundo, casos de teste de integração podem ser utilizados para validar o software em relação à existência de divergências arquiteturais, que podem prejudicar requisitos não-funcionais relacionados à confiança no funcionamento do software. Terceiro, casos de teste de robustez (e as respectivas falhas a serem injetadas) podem ser utilizados para validar o software em relação ao seu comportamento esperado da arquitetura de software, quando na presença de falhas. Esse tipo de validação é especialmente importante no contexto de requisitos não-funcionais relacionados a sistemas tolerantes a falhas. Finalmente, a abordagem baseada em cenários arquiteturais, adotada neste trabalho, possibilita a geração de casos de teste relacionados a comportamentos específicos relativos tanto às funcionalidades quanto ao tratamento de erros, como por exemplo cenários de reconfiguração arquitetural.

Todos os artefatos de teste podem ser reutilizados cada vez que o componente for testado: durante o seu desenvolvimento ou a cada reutilização ou mudança de contexto de uso.

A aplicabilidade da abordagem de validação proposta foi avaliada no contexto de dois estudos de caso apresentados na Seção 2.8. Enquanto a geração de casos de teste de unidade e de integração foi aferida no contexto de um sistema de controle de mineração

baseada na abstração iFTE(Seção 2.8.1), a geração de testes de robustez foi avaliada em um sistema bancário baseado na abstração HoFE (Seção 2.8.2).

5.2 Generate Unit Test Cases

The unit testing can be specified for the architectural elements in a black-box way, thus allowing test cases to be specified and reused even without the components' source code.

For generating test cases for a provided operation of an iFTE, it is necessary to generate a *sequence graph*, which represents the execution of the internal and required operations that the provided operation requires, as well as the respective normal and abnormal returns. The sequence graph consists on a graphical representation of the formal models of the iFTE (Section 3.2.1), which is constructed for each one of its provided operations. The nodes of the sequence graph are identified from the B-Method machine, while the edges are identified from the CSP specification. The nodes of the sequence graph are disposed in different partitions according to the respective interface of the iFTE that is referred. Thus, four partitions are defined for the abstract model, and sixteen for the detailed one.

The test cases generated for testing iFTEs can either take into consideration the internal elements individually (its detailed structure), or generate test cases for a higher abstraction, abstracting away its internal structure and considering only the external interfaces of the iFTE, which is I_iFTE_PN, I_iFTE_PA, I_iFTE_RN, and I_iFTE_RA (Figure 3.1).

For illustrating the artefacts generated, Figure 5.1 presents the execution sequence graph for the controlPumping() operation of the PumpController connector. This graph represents only the external exceptions (provided and required) of the iFTE for generating test cases for a black-box testing. Analysing this graph, 22 paths are identified by using a depth-first search algorithm from the initial node to a final node, producing 22 test cases. Although in this case study the test cases have been manually generated, case tools could use the 'interaction graphs' as input for automatically generating them. In our research group, a prototype tool is being developed for this purpose [84].

The `controlPumping()` operation is represented as an initial node of the `I_PC_PN` partition. In the same way, each required operation that can be executed by `controlPumping()` is represented as a node of the `I_PC_RN` partition. The subset of required operations of `controlPumping()` is determined by the internal relations of the iFTE, defined in the B-Method machine (see Section 4.2.2). The exceptions that can be caught by the PumpController element are represented as nodes of the `I_PC_RA` partition, while the signalled exceptions are disposed in the `I_PC_PA` partition. Finally, it is necessary to define a node to represent the normal return of the element. This node is disposed in the `I_PC_PA` par-

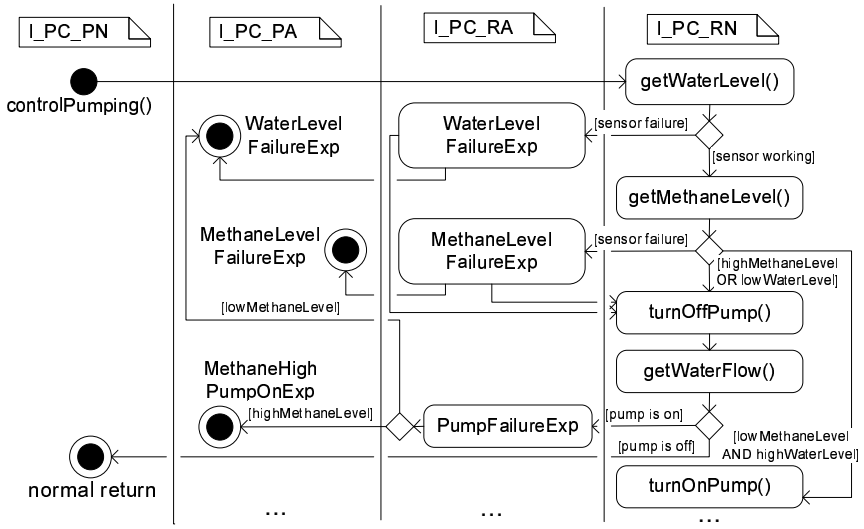


Figura 5.1: Part of execution sequence graph for the controlPumping operation.

tition. The outputs of the architectural element (normal and abnormal) are considered final nodes, representing the end of an execution scenario.

For exemplifying the stub creation, one of the test cases simulates the MethaneHighPumpOnException throwing. In this case, when the pump is turned on, the stub that simulates the getMethaneLevel() required operation was prepared to return the highMethaneLevel. After that, for safety reasons the controller tries to turn off the pump. The stub of the turnOffPump() operation was prepared to throw the PumpFailureException exception when the operation were called. Because it indicates an emergency exception, the controller informs this warning to the MineralExtractorController connector through the MethaneHighPumpOnException. After executing this scenario, the test oracle has to proceed the contract verification, which checks exception class type and context.

The sequence of operation requests and responses is derived from the CSP specification, which represents the architectural scenarios of the application. For example, analysing the following CSP specification: "`pc.I_PC_RN.getWaterLevel.request -> pc.I_PC_RA.getWaterLevel.response?WaterLevelFailureException`" it means that the process `pc.I_PC_RN.getWaterLevel.request` comes before process `pc.I_PC_RA.getWaterLevel.response?WaterLevelFailureException`, thus the provided operation of Conn connector is always executed before any of its required operations.

After constructing the graph, the test cases can be generated in a straightforward way: each path from the start to a final node is considered a test-case. Besides the identification of the test cases (paths of the graph), the graph is also useful for deriving stub synchronisation commands, because it illustrates the sequence on which the required operations are called, as well as the respective expected returns [21]. During the test execution,

stubs should replace required elements, simulating their behaviour in a controlled way, and making it possible to observe behaviour of the component under test in normal and abnormal situations.

Besides the generation of the test cases, it is necessary to determine the correct ordering for executing the test cases of the component's provided operations. Analysing the semantic of the provided operations, it is possible that a mandatory logical sequence of execution exists. For example, because the Pump component is initially turned off, its turnOnPump() operation should be tested before the turnOffPump() one. For determining the testing order, we have to generate the *execution flow graph*, which is a dependency graph that illustrates the sequential dependencies among the provided operations of an architectural element. Beyond its usefulness for determine the best order for proceeding the test, the execution flow graph also derives test drivers for executing test cases.

5.3 Generate Integration Test Cases

Regarding integration testing, the approach adopted by our rigorous process is composed of two parts. First, using dependency analysis we define the integration testing order aiming to optimise the effort necessary for constructing stubs. Second, we present how integration test cases are identified from the formal specification of the software architecture, specially the architectural scenarios.

5.3.1 Dependency Analysis

The existence of dependencies between architectural elements, in terms of their provided and required operations, should impose an order on how these components are integrated, thus facilitating fault localisation. The basis for establishing this order is obtained from the dependency analysis between components. In our work, the order for executing test cases is defined in such a way to reduce the effort of stub creation and consequently the cost of testing. A complementary technique for reducing the effort of stub creation was proposed by Zhang and Ryder [104], which uses advanced program analysis technology to reduce the number of unnecessary nodes and dependencies in the dependency graph. One big challenge in integration testing of component-based systems is how to obtain enough information for dependency analysis given that we cannot assume source code availability. The approach presented here uses the specifications of the architecture and of the iFTEs to apply dependency analysis. At this level an abstract view of the system is considered, but when architectural elements are instantiated by concrete components, this analysis can be updated.

For the dependency analysis, we use the chaining approach [99], which relates the

architectural elements by chaining the existing dependencies between them. Links represent dependencies among these elements and connect elements that are directly related. Analysing those dependency chains, it is possible to support different interests. For example, what components are affected by changes in another component. In the context of this work, we use dependency analysis to identify the best order for integrating the architectural elements in order to facilitate the execution of integration test cases.

The features considered in the dependency analysis rely on the notation used to specify the architecture. In the B-Method and CSP specifications presented in Section 4.2.2, the architecture is described in terms of architectural elements, interfaces, operations, exceptions and events. These features are used to construct a matrix representing the direct dependences among the architectural elements. From this matrix it is possible to obtain, for a given architectural element c, three types of chains: affected-by, affects and related. The *affected-by chain* consists of a set of architectural elements that could potentially affect an architectural element c. The *affects chain* is the set of architectural element that can be affected by c. The *related chain* is a combination of the affected-by and affects chains for c. To determine the integration order, we use two metrics, called *influence factor* (IF) and *late integration factor* (LIF) [71]. These metrics were proposed to help in determining the integration order when testing of object oriented systems and in our work it was adapted for testing component-based systems.

5.3.2 Constructing the Dependency Matrix

The dependency matrix (DM) represents the relationships among architectural elements. The columns of a DM represent the dependency in the relationship, and its rows represent the depended-on element. A cell DM[a, b] = 1 indicates that b depends on a. Both structural and behavioural dependencies are indicated in the matrix, and this information can be obtained from the architectural model. For example, in our case, since we are using B-Method and CSP to describe the architecture, we can obtain structural dependencies from B-Method notation (architectural configuration), and the behavioural ones from CSP (scenarios of use).

The rows and columns of the matrix can represent the architectural elements, interfaces, operations and signalling of exceptions. Events from and to the external environment should also be added, such as, a *start* event provided by the user to initiate the system. Then the dependencies between these elements, as described in the B-Method specification, are inserted into the cells. It is also possible to represent the internal relationships among interfaces of an iFTE, as described by their respective B-Method specifications. After inserting the structural relationships, the matrix is completed with behavioural information obtained from the CSP model. Taking the example of the mining

control system, if the MineralExtractorController (MEC) connector requests services to the PumpController (PC), which requests services to the Pump (P). So, MEC depends on PC, and PC depends on P.

Once the DM is constructed, we can determine the affects(c) and the affected-by(c) chains. The affected-by(c) is determined by the column of the matrix that is related to c. The affects(c) is determined by the row of the matrix that is related to c. Although in the context of this work the dependency matrix was manually generated, the CSP specification of the software architecture could be used as input for automatically generating it.

5.3.3 Determining the Integration Order

As already mentioned, the integration order is obtained based on two metrics: the *influence factor* (IF) and the *late integration factor* (LIF) [71]. The first one computes the number of architectural elements that should be integrated after the architectural element of interest c. It is obtained by counting the number of elements that c directly affects. In other words, it is the number of c' $\neq$ c such that DM[c, c'] = 1. The second factor is LIF(c) = Σ IF(c') for all c' $\neq$ c such that DM[c', c] = 1. The LIF is proportional to the total number of elements that an element depends on. After calculating the LIF of all the elements, we proceed to determine the integration order by performing the following steps:

1. Select the elements with the least LIF. It means that in our strategy the fewer elements an element depends on, the sooner it should be integrated. This rationale is motivated by the reduction of effort for creating stubs. We designate the selected elements by a set, selected_i, where i, in this case, is 1, to indicate that they should be integrated in the first step.

2. Recalculate the LIF of the remaining elements. For each c' $\notin$ selected_i and for each c $\in$ selected_i|c $\in$ affected-by(c') then LIF(c') = LIF(c')-IF(c).

3. Return to Step 1 until all the elements were integrated.

If there are various elements with the same LIF, this indicates the existence of a cycle in the dependency, that is, if LIF(c) = LIF(c') in one step of the integration order determination, then we can conclude that c depends on c' and c' depends on c. They are also designated as *strongly connected elements* [99]. In this case, whichever the element is integrated first, a stub is necessary to substitute the other. Some heuristics were proposed by Lima and Travassos to decide which element to select first [71].

For exemplifying our approach, Table 5.1 presents the two first matrices. To improve the readability, the first column only presents an acronym of the architectural element's

Tabela 5.1: Dependency matrix among architectural elements of the Mining Control System.

	Step 1				Step 2				**IF**	Integration Order
	UI	MEC	AEC	PC	UI	MEC	AEC	PC		
UI	0	0	0	0	0	0	0	0	**0**	11^{th} (step 3)
MEC	1	0	0	0	1	0	0	0	**1**	10^{th} (step 3)
AEC	0	1	0	0	0	1	0	0	**1**	8^{th} (step 2)
PC	0	1	0	0	0	1	0	0	**1**	9^{th} (step 2)
ML	0	1	1	1	—	—	—	—	**3**	1^{st} (step 1)
AF	0	0	1	0	—	—	—	—	**1**	2^{nd} (step 1)
AE	0	0	1	0	—	—	—	—	**1**	3^{rd} (step 1)
ME	0	1	0	0	—	—	—	—	**1**	4^{th} (step 1)
P	0	0	0	1	—	—	—	—	**1**	5^{th} (step 1)
WF	0	0	0	1	—	—	—	—	**1**	6^{th} (step 1)
WL	0	1	0	1	—	—	—	—	**2**	7^{th} (step 1)
LIF	**1**	**8**	**5**	**7**	**1**	**2**	**0**	**0**	—	—

name, and the columns that have no dependency were omitted. Following the algorithm already presented, the influence factor (IF), and the late integration factor (LIF) of the first integration step were calculated. After each step of integration, we have defined the better integration order of the architectural elements, which is presented in the last column of Table 5.1. Notice that in the first step of integration, seven architectural elements have no dependencies (LIF = 0, not presented in Table 5.1). Those are the first elements to be integrated. Then in Step 2, two other elements had LIF = 0 (AEC and PC), which were the next components to be integrated. Finally, the order for integrating the UserInterface (UI) and MineralExtractorController (MEC) is defined only in the third step of integration, which is not presented in Table 5.1.

5.3.4 Generating Integration Test Cases

In this study our concern is to show how architectural formal models can be used for generating test cases. For the sake of conciseness, we sketch here how to use existing techniques to obtain integration test cases. In the particular point of view of our work, the objective of these test cases is to exercise the interactions between the implementation of architectural elements, through the operations at their interfaces, to identify mismatches related to the flow of exceptions between architectural elements.

First, we need to identify the invocation sequence of operations associated to different interfaces. This can be obtained from the CSP specification, which gives the synchronization sequence of internal and architectural events. From this information we can construct an *interaction graph* from which test cases can be derived for each integration step, as identified in Section 5.3.3.

In this graph, operations and the respective possible returns (normal return and exceptions) are represented as nodes. The internal and architectural events (requests and responses) that refer to these operations are represented as edges. When it is an operation request (`ifte...request` in CSP), the edge points to the operation that is being requested; when it is an operation response (`ifte...normalResponse` or `ifte...abnormalResponse`), the edge points to the returned value. The interaction graph also has a start node, which requests the first operation that starts the interaction, and many final nodes, which represent the possible responses of the operation under test. After the interaction graph has been constructed, the test cases can be identified: each path (from the starting node to a final node) is considered a test case for integration testing. But since we focus on the system abnormal behaviour, only the paths that have at least one exception node are considered.

5.4 Generate Robustness Test Cases

From the architectural specification, three model-based test artefacts are created: (i) *types of faults*, which consists on identifying the potential types of faults that can be activated, in order to define the system's fault model; (ii) *points of injection*, which consists on identifying the operations of the architectural interfaces where faults should be injected; and (iii) *the abstract test cases*, which are represented as sequences of operations that should be activated in each test execution.

Since the definition of a representative fault model is one of the most important activities of a fault injection strategy, the proposed test process executes this activity in a top-down way. First, it is essential to clearly identify the type of faults that can be activated in the system under test. This classification must consider both the origin of the faults (internal, external, human, and non-human), and their nature (data value, and event sequence). Further on, these categories are the basis for identifying representative faults for the system under test.

The points in which to inject faults are derived from the B-Method and CSP specifications. For identifying the places in which faults can occur, the proposed approach relies on both structural and behavioural dependencies between architectural elements. In brief, the idea is to select architectural elements based on two types of dependencies: *upstream* dependencies when the failure of an architectural element may cause cascading failures in the rest of the system, and *downstream* dependencies when an architectural element can be affected by failures in the rest of the system. This analysis is facilitated through the construction of a dependency matrix, which represents the dependencies between required and provided interfaces of the architectural elements. Since we use unambiguous notations for representing the software architecture, the dependency matrix could be

automatically generated, using the information of the architectural configuration of the B-Method machine, and the architectural scenarios of the CSP specification.

An abstract test case is directly derived from an architectural scenario specified in CSP, according to the sequence of operations executed in that scenario. For the halt-on-failure property, the specified test case requires the execution of two scenarios: a failure scenario followed by a scenario with no failures. For satisfying the halt-on-failure property, the HoFE should remain halted during the execution of the second part of the test case.

In the following section, these artefacts are refined in order to include details related to the operations' signature, which is obtained from the source code. Due to length restrictions, we do not detail how to identify the points in the architecture where faults should be injected, more details can be found elsewhere [76].

5.4.1 Architectural-Based Robustness Testing

For systematising the identification of potential faults, we have extended an existing classification that categorises the potential faults according to their types [75]. In the following, we present the six categories of faults that have been considered, together with examples in the context of the financial critical system: (i) *sequence faults*, which consists of intercepting method calls to change the sequence of external operations that are requested, e.g., force the BusinessAccount to request first from AccountManagement2 before requesting from AccountManagement1; (ii) *halt-on-failure (HoF) faults*, which consists of provoking the halt condition in an HoFE, e.g., induce the halt of OperationsAccount by injecting faults into the interfaces of its internal components, OperationsAccount1 and OperationsAccount2; (iii) *API calls faults*, which consists of calling the public operations of an architectural element using invalid parameter values, e.g., execute the `getBranch(String branchCode)` operation of the BranchManagement informing "null" as the value of `branchCode`; (iv) *operator faults*, also known as *user faults*, is a special case of API calls faults, and which consists of erroneous usage of the system, e.g., when requesting an operation from UserSession, a user does not supply all the necessary parameters; (v) *hardware failures*, which consists of assessing how the system behaves in case of a host failure, e.g., injection of faults simulating the failure of AccountManagement or BranchManagement hosts; and (vi) *external faults*, which consists of assessing how the system behaves in case of faults being propagated from external components, e.g., injection of faults related to database failures at the interface of DataBase1.

The faults to be injected during the execution of a test case are specified according to the arguments of the operations at each injection point. In this way, for each operation of the tested interfaces, faults are specified using black-box testing techniques, and these are related to the boundary values of argument type, and the business logic [8]. In addition,

we also consider other values from Ballista approach [47], such as, the "null" for Strings and other objects, and zero for numerical values. An example of a boundary value test for the financial critical system refers to the test case where the OperationsAccount halts when there is a divergence between the two internal components of OperationsAccount (difference $\geq 5\%$). The faults to be injected should affect the returning values of the internal components, in order to force a divergence between them. In order to achieve this, six test cases were generated for each of the internal components of OperationsAccount.

In addition to identifying the category of faults and the faults that are to be injected for each test case, the test cases need to be classified since these dictate the order in which these are executed: (i) *testing single failures*, which consists of scenarios where one fault should be injected; and (ii) *testing stressful conditions*, which consists of scenarios where more than one fault should be injected, thus assessing the system behaviour under multiple failures. The testing of stressful conditions should only be applied after all the single failure test cases have been tested. Amongst the test cases identified for injecting faults, there were two of them in which two faults were injected at the same time: failure1 and failure2 test cases. In the failure1 test case, the two replicas of AccountManagement should halt, while in the failure2 test case, the two replicas of BranchManagement should halt. Since they contain more than one fault injected at the same time, these test cases are considered tests for stressful conditions.

Following the proposed scenario-based robustness testing strategy, we have specified a total of 178 faults to be injected in 30 test cases. These faults where injected at the interfaces of the architectural elements using the Jaca case tool. Table 5.2 presents the places where the faults where injected, as well as, depending on the category of the fault, the number of faults injected. For example, for the (*sequence faults*) category, four faults were specified: three injected at the I_BA_Req2 interface, and one injected at the I_BA_Req2 interface. The last row of the table shows the total amount of faults that where injected in each place.

Tabela 5.2: Quantity and types of faults injected in each place

Fault Category	**Places where faults were injected**													
	I_US_Req	*I_AS_Prov*	*I_OA_Prov*	*I_OA_Prov*	*I_OA2_Prov*	*I_BA_Prov*	*I_BA_Req1*	*I_BA_Req2*	*I_AM_Prov*	*I_AM2_Prov*	*I_BM1_Prov*	*I_BM2_Prov*	*I_DBS_Req*	*I_DBS2_Req*
Sequence faults	-	-	-	-	-	-	3	1	-	-	-	-	-	-
HoF faults	-	-	-	6	6	-	-	-	-	-	-	-	-	-
API calls faults	-	11	22	-	-	34	-	-	32	32	2	2	-	-
Operator faults	11	-	-	-	-	-	-	-	-	-	-	-	-	-
Hardware failures	-	-	-	-	-	-	-	-	3	3	1	1	-	-
External faults	-	-	-	-	-	-	-	-	-	-	-	-	4	4
TOTAL	**11**	**11**	**22**	**6**	**6**	**34**	**3**	**1**	**35**	**35**	**3**	**3**	**4**	**4**

The fault injection process has the following activities: select an appropriate test case

to be executed, select the respective interfaces to inject the faults, the faults to be injected, and the points to be monitored. During the automatic fault injection, the Jaca tool observes events and state changes on the monitored points (classes, attributes and operations) and stores them in a logging file. This log is then used to evaluate the test results. In the context of the financial critical system, we have identified two inconsistencies. The first inconsistency was in the context of OperationsAccount. Although the deviation between its two internal components should be less than 5%, when we injected faults considering a deviation of exactly 5%, the OperationsAccount returned a result, contradicting the expected halt-on-failure scenario. The second inconsistency was related to two arguments of the updateAccountCredit(...) operation of the I_BA_Prov interface. When incompatible values were provided for these arguments, the BusinessAccount connector returned result, contradicting again the expected halt-on-failure scenario. All the other test cases were successfully validated for the identified faults.

5.5 Validation Evaluation

The claim being made in this chapter is that the use of an architectural abstraction and architectural scenarios can seamlessly integrate the activities related to the formal specification, verification and validation of fault tolerant software. In the following, in order to complement the material presented so far, we summarise two experiences: first, the employment of the iFTE architectural abstraction while developing a software for the mining control system (Section 2.8.1 and 6.3.1) including the generation of unit and integration test cases. Second, the employment of the HoFE architectural abstraction while developing a software for the banking system (Sections 2.8.2 and 6.3.2) including the generation of robustness test cases.

It is important to justify that even when the software architecture is formally verified (see Chapter 4), the consistency of the final source code with the software architecture is not guaranteed. So, in order to improve the dependability of the running application, test cases have to be generated in order to assess this consistency by analysing the behaviour of the application in architectural scenarios involving critical situations.

5.5.1 Unit and Integration Testing

Overall, the case study of the iFTE-based mining control system has shown how a fault-tolerant architectural abstraction based on exception handling can be used to formally specify software architectures, this providing the means to generate architecture-based unit and integration test cases. Based on the behavioural specification of the architectural abstraction, unit test cases where generated for the architectural elements. Moreover, the

architectural scenarios, which considers the behaviour of architectural elements connected together, were also used to generate integration test cases, which aims to assess the correctness of the source code against the architectural specification.

In a complementary way, the properties specified for the architectural abstraction and the architectural configuration of the application can also be used to generate unit and integration test cases, respectively.

5.5.2 Robustness Testing

Regarding the system validation through robustness testing, we noticed that the adoption of an architectural abstraction allows to refine the fault model and test cases for assessing particular characteristics of the system. For example, when validating the implementation of the halt-on-failure semantics, five specific test cases were generated, together with additional 24 faults. For the specification of the test cases, it was clear that the architectural scenarios were fundamental to identify the strategic places where to inject faults. For instance, faults were injected at the interfaces of the BusinessAccount component of the banking system in order to assess the scenarios related to the architectural reconfiguration. As a measure of the effectiveness of the proposed testing approach, for the 35 test cases executed, the coverage of the source code indicated by the Clover Eclipse plugin [3] was 83% of the total source code and 100% of the exception handlers' source code. Moreover, although the same system has already been tested using unit test cases [21], the proposed approach has identified two new errors in the system implementation. We believe the new errors were activates by scenarios scenarios which are difficult to reproduce without fault injection.

5.6 Summary

This chapter has presented the validation approach, which is part of the rigorous architectural solution proposed in this thesis. An important characteristic of the model-based testing proposed here is that it provides a valuable coverage criteria based on the abnormal architectural scenarios of the fault-tolerant system being developed. To achieve this, three complementary techniques of model-based testing are used. First, unit test cases are generated from the formal specification of the architectural elements (see Section 4.2). The objective of such test cases is to asses the consistency between each architectural element and the architectural abstraction it follows. Second, integration test cases are generated from the formal specification of the architectural configuration. The objective of such test cases is to identify architectural mismatches associated to the interaction between architectural elements. Finally, robustness test cases are also generated from the

formal specification of the architectural configuration. The objective of such test cases is to assess the behaviour of the software system in the presence of faults (abnormal behaviour), which is specially important in the context of fault-tolerant software.

After detailing the verification and validation approaches, it is necessary to systematise the use of such activities in the context of a rigorous development and testing process. Chapter 6 presents how the verification and validation approaches can be combined for developing fault-tolerant software architectures.

Capítulo 6

Um Processo Rigoroso e Centrado na Arquitetura para Sistemas de Software Baseados em Componentes

Esse capítulo apresenta um processo recursivo rigoroso e centrado na arquitetura de software voltado para projetar a arquitetura de software de sistemas tolerantes a falhas utilizando o conceito de abstrações arquiteturais. A recursão do processo proposto o torma mais escalável para lidar com sistemas complexos. O conteúdo desse capítulo foi retirado de um artigo publicado na *2nd European Conference on Software Architecture* (ECSA 2008) [18]. Como o conteúdo do capítulo foi extraído na íntegra desses artigos, foi preservado o idioma original.

6.1 Resumo do Capítulo

A incorporação de tolerância a falhas em sistemas de software normalmente aumenta a sua complexidade complexity, o que consequentemente torna a sua análise mais difícil. Esse capítulo discute como abstrações arquiteturais adequadas e um processo recursivo podem facilitar o projeto e verificação de arquiteturas de software para sistemas tolerantes a falhas. Dependendo do modelo de falhas e da disponibilidade de recursos, diferentes abstrações podem ser utilizadas para representar aspectos relacionados a tolerância a falhas, tais como detecção de erros, propagação de erros, tratamento de erros e tratamento de falhas. Essas abstrações arquiteturais e suas respectivas estruturas internas podem ser instanciados em componentes e conectores concretos para projetar arquiteturas de software tolerantes a falhas. Detalhes sobre a execução das atividades de verificação formal e validação, que são definidas no processo, foram apresentados nos Capítulos 4 e 5, respectivamente. A aplicabilidade da solução proposta foi avaliada no contexto de uma

aplicação crítica de tempo real.

6.2 A Rigorous Development and Testing Process Using Architectural Abstractions

In our approach, architectural abstractions are first-level units, guiding the development from the architectural specification to the implementation of the application. We propose an iterative, recursive and incremental process for developing fault-tolerant software architectures. Figure 6.1 presents an overview of the proposed approach.

Activity 1 specifies the software architecture, which can be done graphically using a CASE tool. From the system requirements, the architect should first choose an architectural abstraction based on the fault model of the application and the available resources for implementing redundancy. Then, two artefacts should be specified: a UML component diagram representing the structure of the software system, and a set of UML sequence diagrams representing the architectural scenarios related to fault tolerance. The scenarios are represented as sequence of events describing how a specific system behaves in presence of errors for the purpose of tolerating faults. These specific scenarios should be consistent with those defined for a particular architectural abstraction.

Activity 2 formally specifies the software architecture through two complementary artefacts: one representing the architectural configuration and another representing the architectural scenarios. This activity consists on an automatic model transformation from UML (XMI files) to B-Method and CSP. Details about the formal framework are presented in Chapter 4.

Activity 3 consists on the execution of the formal verification of the architectural elements in order to identify design faults related to behavioural inconsistencies that might exist when taking as a reference the scenarios associated with a particular architectural abstraction. So, a different set of predefined properties is used according to the adopted architectural abstraction. For example, in the context of the iFTE (Section 3.2.1), the verification focuses on the consistency related to scenarios of exception handling (e.g., exception raising, exception masking). When the architectural element is based on the HoFE (Section 3.2.2), the focus of verification concerns the scenarios of error detection (e.g., same result - *normal*, accepted divergence - *normal*, unaccepted divergence - *abnormal*). A complete list of properties verified for the iFTE abstraction is presented in Section 4.3.2 of Chapter 4.

Activity 4 consists of the generation of unit test cases for assessing the implementation of architectural elements. Although at this stage we do not have detailed information regarding lower-level design, the interfaces of the architectural elements contains

the complete signature of the architectural operations. In other words, in the proposed architecture-based approach, the unit test cases are generated from the verified formal models of the architectural elements. Details about the generation of unit test cases are available in Section 5.2 of Chapter 5.

Activity 5 verifies the architectural configuration, in order to assess the consistency of interactions between architectural elements. Depending of the architectural abstractions, different properties of interest are verified. For example, in the context of the iFTE, the focus of verification is the consistency of the exception control flow and handlers, while the HoFE focus on the verification of scenarios of error detection and halting. A complete list of properties verified for architectural configurations based on iFTEs is presented in Section 4.3.2 of Chapter 4.

Activity 6 uses the verified formal model of the architectural configuration for generating integration and robustness test cases. These high-level test cases aim to assess the existence of architectural mismatches. Beyond the mismatches regarding the compatibility of required and provided operations, in the case of iFTE, it is particularly necessary to assess mismatches of exception types at propagation scenarios. Details about the generation of integration and robustness test cases are available in Sections 5.3 and 5.4 of Chapter 5.

Activity 7 specifies the internal structure of the architectural elements, according to the internal structure of the chosen abstraction (see Section 3.2). This activity consists on the recursive execution of the whole process (rigorousProcess), starting in Activity 1. Recursion is used to provide scalability in terms of an unlimited internal refinement of architectural elements.

After the system (or detailed architectural element) has been properly verified and the test cases already generated, Activity 8 consists on the implementation of the system, which ideally should be automatically executed. In order to provide a mapping between the software architecture and the implementation, we suggest the use of a component implementation model such as COSMOS [38]. Even considering current trends on Model-Driven Development [102], our approach is motivated by the fact that there are no full guarantees that the source code is an accurate implementation of its software architecture.

In Activity 9, the source code should be validated against its specification through the execution of unit and integration test cases, which were previously generated. Finally, if faults are identified during the test execution, they should be fixed in Activity 10, otherwise, the source code is considered deployable. Since each recursion implies in a deliverable source code artefact, the proposed process is also considered incremental for delivering the system in parts. Moreover, the iterative characteristic of the process allows the detection of errors in different stages of the software development, specification (Activities 3 and 5) and implementation (Activity 9).

6.3 Evaluation

Two case studies have been executed in order to assess the benefits of the proposed approach when comparing the modelling of fault-tolerant software architectures using both the Unified Modelling Language (UML) and the two architectural abstractions presented in Section 3.2. First Section 6.3.1 presents the case study of the mining control system, which was presented in Section 2.8.1. Second, Section 6.3.2 presents the architectural design of the banking system presented in Section 2.8.2.

6.3.1 Mining Control System

The mining control system, which was presented in Section 2.8.1, has been adopted as a case study for showing the benefits of the proposed approach when comparing the modelling of fault-tolerant software architectures using both UML and the two architectural abstractions presented in Section 3.2.

Description of the Case Study

The main goal of the case study was to evaluate the feasibility of the proposed approach, as well as the advantages of employing an abstraction-based development process. More specifically, this case study considers the execution of Activity 1 (Specify the FT Software Architecture and its Abnormal Scenarios) of the rigorous process presented in Figure 6.1. We have specified the mining control system using three architectural abstractions: (i) UML software component, which is a non-dependability abstraction; (ii) the iFTE architectural abstraction, which implements reliability through implicit design diversity; and (iii) the HoFE architectural abstraction, which implements reliability through explicit design diversity.

The execution of the case study is composed of four steps. First, we have executed Activity 1 of the process (Figure 6.1) choosing the UML software component as an architectural abstraction (Step 1). Second, we have executed the same activity choosing the iFTE architectural abstraction (Step 2). Third, we have executed the specification once more choosing the HoFE architectural abstraction (Step 3). Finally, we compare the three executions of the process by analysing quantitative and qualitative aspects (Step 4). The quantitative aspects analysed are: (i) effort for verifying the software architecture; and (ii) scalability of verification. The qualitative aspects analysed are: (i) separation of concerns; (ii) control of complexity; (iii) evolvability of the software architecture; and (iv) dependability of the software architecture. The following sections detail the four steps of the case study and then summarises the overall evaluation.

Step 1: Choosing UML Software Components

In Step 1 of the case study, we have chosen the UML software component as an abstraction for specifying the software architecture of the mining control system (Activity 1 of Figure 6.1). The resulting software architecture is presented in Figure 6.2. The software architecture is composed of 20 architectural elements, four of them are sensors: (i) MethaneLevel, which detects the level of methane inside the mine; (ii) AirFlow, which detects the flow of air inside the pipes; (iii) WaterLevel, which detects the level of water inside the mine; and (iv) WaterFlow, which detects the flow of water inside the pipes.

The identified controllers (MineralExtractorController, AirExtractorController, and PumpController), have the role of architectural connectors. Since they have reliability requirements, they are redundant at the software architecture, in order to allow error detection by comparison. Each controller is responsible for dealing with the normal behaviour of the system, and handling any exceptions that are propagated by the components. Depending on the state of the sensors, two of the controllers will always be activated: (i) water low & methane low ⇒ MineralExtractorControllers; (ii) water high & methane low ⇒ PumpControllers; and (iii) methane high ⇒ AirExtractorControllers.

For this architectural configuration, a total of 13 architectural exceptions were identified related to errors in the system architecture. For exemplifying the flow of exceptions, in the following, we consider the case when an error is detected inside the AirExtractorErrorDetector, and an internal exception is raised. If AirExtractorErrorDetector fails to handle this exception locally, it propagates an exception to the AirExtractorControllers. Again this architectural elements attempt to handle the exception once it is caught, but if they fail, the exception is propagated to the MineralExtractorControllers. If the concentration of methane is high and the AirExtractorErrorDetector has failed, there is nothing that the MineralExtractorControllers can do, except to propagate an exception to its collaborating architectural elements. Upon receiving this exception, the MineralExtractor, the PumpControllers and the AirExtractorControllers should shut down their activities, and the UIErrorDetector should raise an alarm for the operator to take the appropriate measures.

As indicated by the development process, the verification of the software architecture follows two steps. First, the architectural elements were verified against the restrictions specified by UML. Each element was verified through 6 properties of interest assessing the call-return nature of its interfaces. Second, for the software architecture, we have specified a total of 43 properties, focusing the attention on the scenarios considered critical to the application. These scenarios occurs when the Pump is turned on, and the MethaneLevel components informs that the level of methane is high. In this case, the only sequence of operations that should be possible is turn the Pump off, and turn the AirFlow component on. If these operations are successfully executed, the system continues working. In any other case, an alarm should be raised into the UIErrorDetector. A total of three properties

have been violated, all related to swallowing of exceptions during architectural propagation. After fixing the model, no property has been violated. The recursive refinement of the architectural elements (Activity 7 of Figure 6.1) have not been executed, since the abstraction used has no internal structure.

Regarding the generation of test cases, we have specified two kinds of tests: (i) unit test cases were generated for each software architectural element (Activity 4 of Figure 6.1); and (ii) integration test cases were generated for assessing the existence of mismatches between the architectural elements (Activity 6 of Figure 6.1). Table 6.1 summarises the number of unit test cases generated for the software elements. Regarding integration testing, test cases were generated for each pair of elements that interact each other, focusing especially in the flow of exceptions between architectural elements. We have specified a total of 55 test cases. The criteria adopted for the integration test was to coverage all the critical scenarios specified for the application (involving error detection and risk of death). To assess the error detection, we have tried to identify both *false positives*, which are limit situations of the normal behaviour that could be wrongly interpreted as an error, and *false negatives*, which are limit situations of the abnormal behaviour that could not be detected by the application.

Tabela 6.1: Number of unit test cases for the software architectural elements.

Arch. Element	#	**Arch. Element**	#
UIErrorDetector	14	UserInterfaces	6 (each)
MineralExtractorControllers	38 (each)	AirExtractorControllers	19 (each)
PumpControllers	22 (each)	AirExtractorErrorDetector	7
AirExtractors	6 (each)	MineralExtractor	6
Pump	6		

Step 2: Choosing the iFTE Abstraction

In Step 2 of the case study, we have chosen the iFTE abstraction for specifying the software architecture of the mining control system (Activity 1 of Figure 6.1). The resulting software architecture is presented in Figure 6.3 using the UML 2.0 notation. The software architecture is composed of 11 architectural elements, four of them are sensors: (i) MethaneLevel, which detects the level of methane inside the mine; (ii) AirFlow, which detects the flow of air inside the pipes; (iii) WaterLevel, which detects the level of water inside the mine; and (iv) WaterFlow, which detects the flow of water inside the pipes.

The identified controllers (MineralExtractorController, AirExtractorController, and PumpController), have the role of architectural connectors. Each controller is responsible for dealing with the normal behaviour of the system, and handling any exceptions that are

propagated by the components. Depending on the state of the sensors, one of the controllers will always be activated: (i) water low & methane low ⇒ MineralExtractorController; (ii) water high & methane low ⇒ PumpController; and (iii) methane high ⇒ AirExtractorController. In case there is a failure that cannot be handled by the system, the AirExtractorController signals an exception to notify the UserInterface element that such a failure has occurred.

For this architectural configuration, a total of 13 architectural exceptions were identified related to errors in the system architecture. For exemplifying the flow of exceptions, in the following, we consider the case when an error is detected inside the AirExtractor, and an internal exception is raised [43]. If AirExtractor fails to handle this exception locally, it propagates an exception to the AirExtractorController. Again this architectural element attempts to handle the exception once it is caught, but if it fails, it propagates the exception to the MineralExtractorController. If the concentration of methane is high and the AirExtractor has failed, there is nothing that MineralExtractorController can do, except to propagate an exception to its collaborating architectural elements. Upon receiving this exception, the MineralExtractor, the PumpController and the AirExtractorController should shut down their activities, and the UserInterface should raise an alarm for the operator to take the appropriate measures.

As indicated by the development process, the verification of the software architecture follows two activities. First, the architectural elements were verified against the restrictions specified by the iFTE. Each element was verified through 22 properties of interest assessing exception signalling and masking. Second, for the software architecture, we have specified a total of 17 properties, focusing the attention on the scenarios considered critical to the application. These scenarios occurs when the Pump is turned on, and the MethaneLevel components informs that the level of methane is high. In this case, the only sequence of operations that should be possible is turn the Pump off, and turn the AirFlow component on. If these operations are successfully executed, the system continues working. In any other case, an alarm should be raised into the UserInterface.

Executing the process recursively (Activity 7 of Figure 6.1), the internal structure of the iFTEs were detailed according to the iFTE abstraction. Figure 6.4 presents the internal details of two of the architectural elements: PumpController and Pump. The Required component of the PumpController and the Provided component of the Pump are responsible for enabling the interaction, adapting the received requests (Provided), and the respective return values (Required). Note that the reuse of the internal structure of the iFTE represents a high-granularity reuse and a consequent reduction of cost.

Regarding the generation of test cases, we have specified two kinds of tests: (i) unit test cases were generated for each software architectural element (Activity 4 of Figure 6.1); and (ii) integration test cases were generated for assessing the existence of mismatches

between the architectural elements (Activity 6 of Figure 6.1). Table 6.2 summarises the number of unit test cases generated for the iFTEs. Regarding integration testing, test cases were generated for each pair of elements that interact each other, focusing especially in the flow of exceptions between architectural elements. We have specified a total of 48 test cases. The criteria adopted for the integration test was to coverage all the critical scenarios specified for the application (involving error detection and risk of death). To assess the error detection, we have tried to identify both *false positives*, which are limit situations of the normal behaviour that could be wrongly interpreted as an error, and *false negatives*, which are limit situations of the abnormal behaviour that could not be detected by the application. Details about the execution of test cases (Activity 9 of Figure 6.1) are available elsewhere [15, 16].

Tabela 6.2: Number of unit test cases for the iFTEs.

Arch. Element	#	**Arch. Element**	#
UserInterface	6	MineralExtractorController	38
AirExtractorController	19	PumpController	22
AirExtractor	6	MineralExtractor	6
Pump	6		

Step 3: Choosing the HoFE Abstraction

In Step 3 of the case study, we have chosen the HoFE abstraction during the specification of the software architecture of the mining control system (Activity 1 of Figure 6.1). The resulting software architecture is presented in Figure 6.5. The use of the HoFE abstraction has considerably reduced the number of architectural elements, since the mechanism of error detection is hidden inside the architectural elements, and noticed only at the second level of recursion. The HoFE-based architecture is composed of 11 architectural elements, four of them are sensors: (i) MethaneLevel, which detects the level of methane inside the mine; (ii) AirFlow, which detects the flow of air inside the pipes; (iii) WaterLevel, which detects the level of water inside the mine; and (iv) WaterFlow, which detects the flow of water inside the pipes. It is assumed that in this system all the architectural elements are HoFEs, except for the four sensors (AirFlow, MethaneHigh, WaterLow, WaterHigh) and the MineralExtractor component, which are assumed to be free of faults.

The three identified controllers (MineralExtractorController, AirExtractorController, and PumpController), have the role of architectural connectors (≪`HoFConnectors`≫). Each controller is responsible for dealing with the normal behaviour of the system, but they will stop once an internal error is detected. For example, if there is a fault that prevents

the air to be extracted, the AirExtractor detects it and halts. After that, the AirExtractorController, which cannot operate without the AirExtractor working, halts to inform the MineralExtractorController that there is a problem. Finally, when the MineralExtractorController halts, the UserInterface detects it and raises a warning alarm.

Regarding the verification of the HoFE architectural elements (Activity 3 of Figure 6.1), each element was verified through 10 properties of interest predefined by the abstraction. For verifying the architectural configuration (Activity 5 of Figure 6.1), we have specified a total of 21 properties, focusing the attention on the scenarios considered critical to the application. There are two kinds of critical scenarios: (i) scenarios involving error detection, which occur when the internal redundancies of the HoFE diverge on their results; and (ii) scenarios involving risk of death, which occur when the Pump is turned on, and the MethaneLevel components informs that the level of methane is high. During the verification, no property have been violated. To test the verification mechanism, we have forced a failure in the model, allowing the AirExtractor component to continue working after a previous halt. In this case, two properties have been violated: *violation of the halt-on-failure principle*, related to the HoFE abstraction, and *invalid architectural scenario*, related to the architectural configuration.

Executing the process recursively (Activity 7 of Figure 6.1), the internal structure of some architectural elements were detailed according to the HoFE abstraction. Figure 6.6 illustrate the recursive decomposition of two of the architectural elements: PumpController and Pump. For verifying the internal details of the HoFEs, other 15 properties have been used.

Regarding the generation of test cases, we have specified two kinds of tests: (i) unit test cases were generated for each HoFE (Activity 4 of Figure 6.1); and (ii) integration test cases were generated for assessing the existence of mismatches between the architectural elements (Activity 6 of Figure 6.1). Table 6.3 summarises the number of unit test cases generated for the HoFEs. Regarding integration testing, test cases were generated for each pair of elements that interact each other, focusing especially in the error detection, and the respective impact of it in the execution flow. We have specified a total of 38 test cases. We have adopted the same coverage criteria used for the UML abstraction.

Tabela 6.3: Number of unit test cases for the HoFEs.

Arch. Element	#	**Arch. Element**	#
UserInterface	3	MineralExtractorController	15
AirExtractorController	7	PumpController	9
AirExtractor	3	Pump	3

Step 4: Case Study Evaluation

Regarding the system verification, we notice that the violation of properties in Step 1 implied that the model has been fixed and the verification has been executed twice. Moreover, the bigger number of architectural elements implied in the specification of more scenarios related to error detection, thus increasing the number of properties related to the critical behaviour of the application in (105%, approximately). Although the verification has been successfully executed in both architectures, the number of state-space necessary in each case indicates a better scalability when using a higher-level abstraction. The definition of more architectural elements increased the state-space in approximately 243%.

Regarding the qualitative analysis of the case study, we have noticed that the use of specific architectural abstractions has provided a gain of separation of concerns, control of complexity, and evolvability of the software architecture. Moreover, the use of architectural abstractions made explicit some important design decisions regarding software fault-tolerance. In the iFTE-based architecture, presented in Figure 6.3, it is explicit that iFTComponents and iFTConnectors are responsible for detecting errors and tolerating faults through implicit design diversity using exception handlers. In the HoFE-based architecture, presented in Figure 6.5, the HoFComponents and HoFConnectors are responsible for detecting errors through explicit design diversity using redundant components. The control of complexity was improved in three complementary ways: (i) reduced number of architectural elements; (ii) reduced number of dependencies between architectural elements; and (iii) simplicity of architectural scenarios, which in the case of the HoFE, abstract away the behaviour of error detection. The internal decomposition of the abstractions and their respective behavioural refinement, has enabled a stepwise development of the system, which simplifies the overall system model, and improves its understandability.

Finally, regarding the evolvability of the software architecture, the use of abstractions facilitates the addition of new non-functional requirements. For example, if we would like to evolve the HoFE-based architecture to also tolerate faults (not only to detect errors), the architecture presented in Figure 6.5 would be almost the same, changing only the architectural abstraction to a Fault-Tolerant-HoFE, which encapsulates two HoFEs and a switcher. In the case of the UML architecture (Figure 6.2), it would be necessary to completely refactor the architecture, in order to add redundancy (extra architectural elements) and the switching behaviour.

6.3.2 Banking System

This section presents the architectural design of the banking system, which was presented in Section 2.8.2. The specification of the software architecture corresponds to the execution of Activity 1 of Figure 6.1. The abstraction-based specification of the banking system

architecture also enforces the feasibility of the proposed approach.

The following sections present, first the iFTE-based software architecture of the banking system and then its HoFE-based architecture. It is important to stress that although we have designed two software architectures, one with each architectural abstraction, it would also be possible to combine the use of different abstractions into a single architecture.

iFTE-Based Architecture

The fault-tolerant software architecture specified for the banking system (Section 2.8.2) is presented in Figure 6.7. This architecture is based on the iFTE abstraction, and contains ten components and four connectors: (i) UserSession is responsible for controlling the user sessions; (ii) AlarmSession is responsible for raising an alarm to warn the user about a problem; (iii) AccountSession is responsible for mediating the interactions between UserSession and OperationsAccount, and between AlarmSession and OperationsAccount; (iv) two redundant instances of OperationsAccount are responsible for processing transactions during the life-cycle of loan contracts; (v) BusinessAccount is responsible for coordinating the collaboration between the two instances of OperationsAccount, and the components responsible for the storage system; (vi) two redundant instances of AccountManagement and BranchManagement are responsible, respectively, for controlling the access to the client's account information, and the details of the branch providing the loan; (vii) DB1Session and DB2Session are responsible for mediating the interactions between the AccountManagement and BranchManagement and their respective databases; and (viii) Database1 and Database2 are a pair of redundant databases, responsible for storing the information of the current accounts and loan contracts of the bank.

The objective of this system is to guarantee that all the loans' transactions are properly recorded, and that all the information provided about the loan contracts is accurate. In the software architecture shown in Figure 6.7, service availability is obtained by replicating AccountManagement and BranchManagement. The replicas of these two components are distributed in different physical locations, and they access redundant data bases. It is the responsibility of the BusinessAccount to select the proper replica in case of failure. The provision of reliable services is obtained through AccountSession, which analyses the output of the two instances of OperationsAccount, and detects divergences between them. If the divergence between OperationsAccount1 and OperationsAccount2 is beyond a prespecified factor of 5%, the AccountSession raises an exception and sends a warning through the AlarmSession component.

HoFE-Based Architecture

For improving the reliability and availability of the transactions associated with the management of loans, we have defined a fault-tolerant software architecture based on the HoFE architectural abstraction, which is shown in Figure 6.8.

The software architecture of the system is composed of eight components and four connectors: (i) UserSession is responsible for controlling the user sessions; (ii) AccountSession is responsible for mediating the interactions between UserSession and OperationsAccount; (iii) OperationsAccount is responsible for processing transactions during the life-cycle of loan contracts; (iv) BusinessAccount is responsible for coordinating the collaboration between the OperationsAccount component and the storage systems; (v) two redundant instances of AccountManagement and BranchManagement, which are responsible, respectively, for controlling the access to the client's account information, and the details of the branch providing the loan; (vi) DB1Session and DB2Session are responsible for mediating the interactions between the AccountManagement and BranchManagement components and their respective databases; and (vii) Database1 and Database2 are a pair of redundant databases, responsible for storing the information of the current accounts and loan contracts of the bank. The objective of this software architecture is to guarantee that all the loans' transactions are properly recorded.

In the software architecture shown in Figure 6.8, service availability is obtained by replicating AccountManagement and BranchManagement. The replicas of these two components are distributed in different physical locations, and accessing different data bases. It is the responsibility of the BusinessAccount to select the proper replica in case of failure. The provision of reliable services is obtained through OperationsAccount, which implements the HoFE architectural abstraction. The implementation relies on two components using different software versions and a Decider. If the divergence between OperationsAccount1 and OperationsAccount2 is beyond a pre-specified factor of 5%, the Decider fails silently. Once AccountSession detects this failure, it sends a warning to the UserSession.

6.3.3 Overall Evaluation

The claim of general approach presented in this chapter is that the use of dependable architectural abstractions can be effective when developing fault-tolerant software systems. Firstly, the explicit consideration of key issues related to fault tolerance, such as, error detection, reduces the chance of neglecting essential decisions during the system design. Secondly, system verification has also benefited from using architectural abstractions. The properties of interest associated with a dependable architectural abstraction can be used as a basis for verifying the architectural elements and their composition. Moreover, the process of verifying, first, a software architecture based on the architectural abstrac-

tion, and then on its detailed model, proportionates an incremental verification process that promotes the identification and correction of faults at earlier stages of software development. Finally, regarding the system validation, the adoption of an architectural abstraction allows the test cases for the architectural elements and their configurations to be gradually refined.

6.4 Summary

This chapter has presented a rigorous development and testing process, which uses the verification and validation approaches presented in Chapters 4 and 5 in a systematic way for developing fault-tolerant software architectures. First, formal verification is used to identify, as soon as possible, errors into the architectural configuration. It is important to highlight the benefits of anticipating the identification of errors to the architectural design phase, since problems at the software architecture normally are difficult and expensive to be fixed in further phases of the software development [98]. After verifying the architectural configuration, model-based test cases are generated from the formal model that has been verified. Such test cases aim to assess the consistency between the source code of the final product and the abnormal architectural scenarios of the architectural configuration.

The main characteristics of the rigorous process presented here are: (i) it is rigorous, since it combines formal verification and model-based test cases in a complementary way; (ii) it is top-down, since the development starts from a high-level view of the software architecture, which is detailed in further iterations; and (iii) it is scalable in terms of complex systems, since it uses recursion as a mechanism to gradually refine the internal structure of the architectural elements.

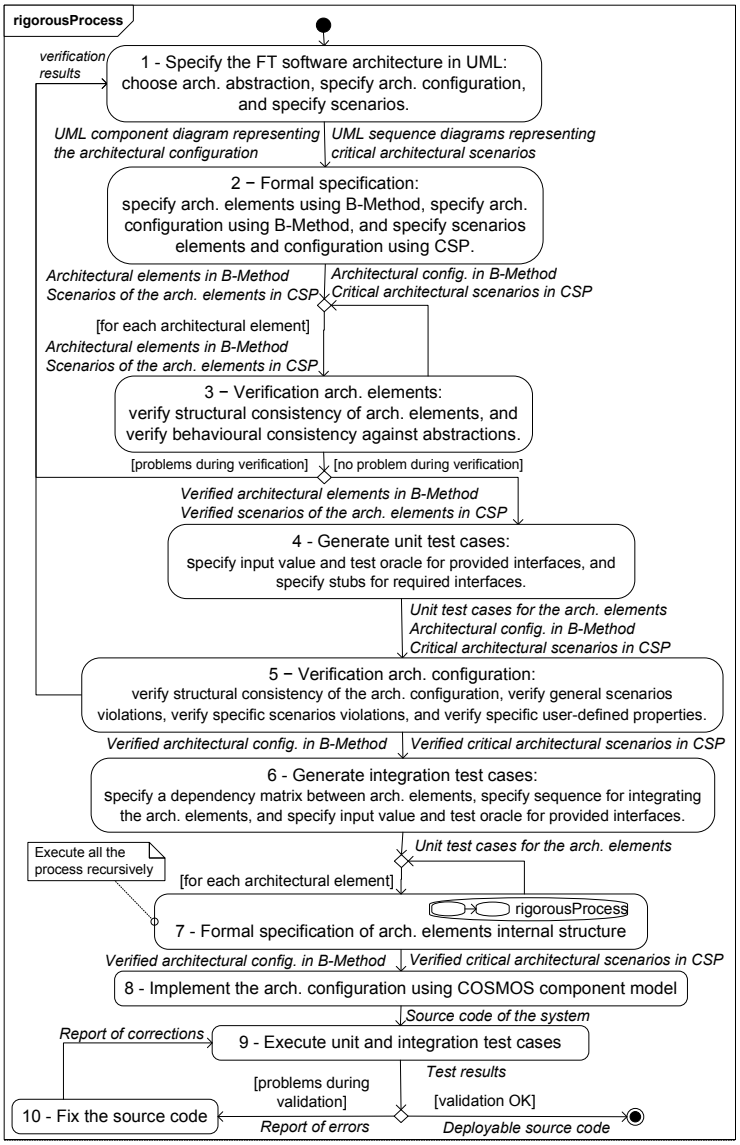


Figura 6.1: A Rigorous Process for Developing and Testing Fault-Tolerant Software Architectures Using Architectural Abstractions

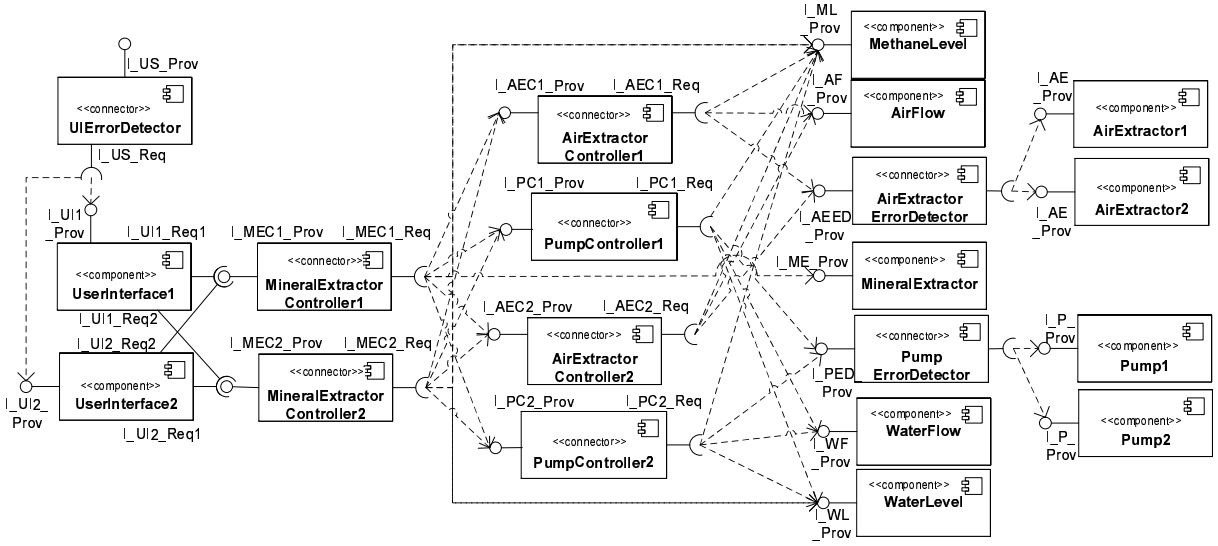


Figura 6.2: Software Architecture of the Mining Control System

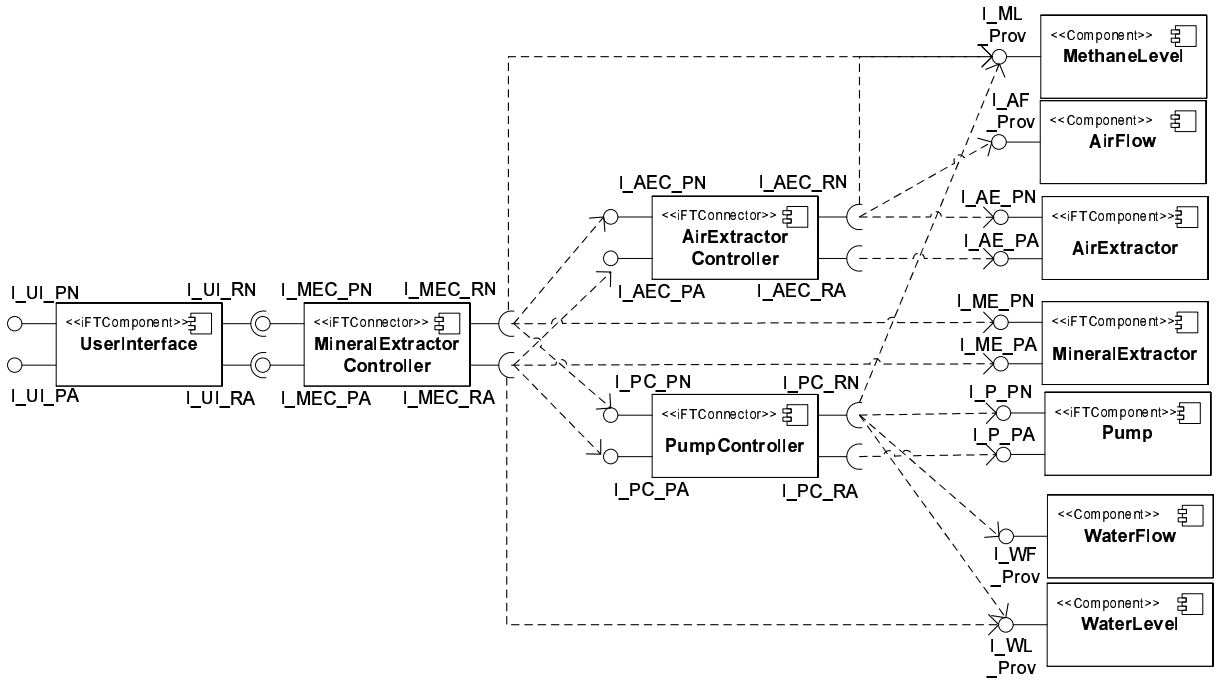


Figura 6.3: iFTE-based Architecture of the Mining Control System

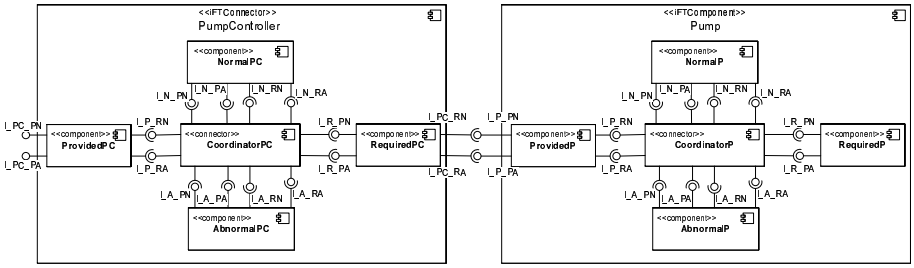


Figura 6.4: Part of the Detailed View of the Mining Control System Architecture (iFTE)

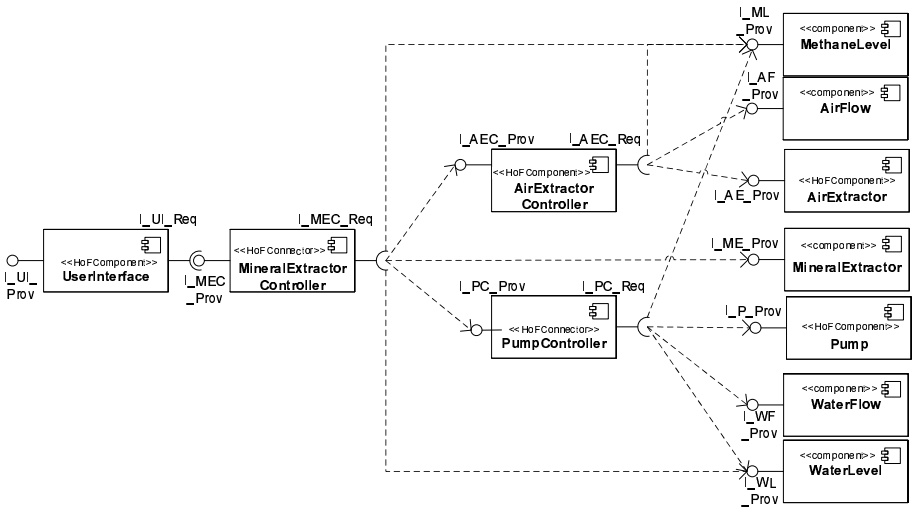


Figura 6.5: HoFE-based Architecture of the Mining Control System

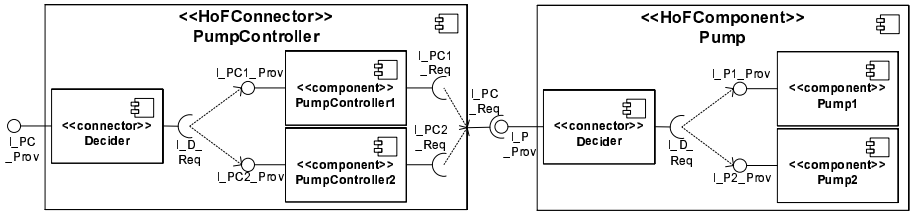


Figura 6.6: Part of the Detailed View of the Mining Control System Architecture (HoFE)

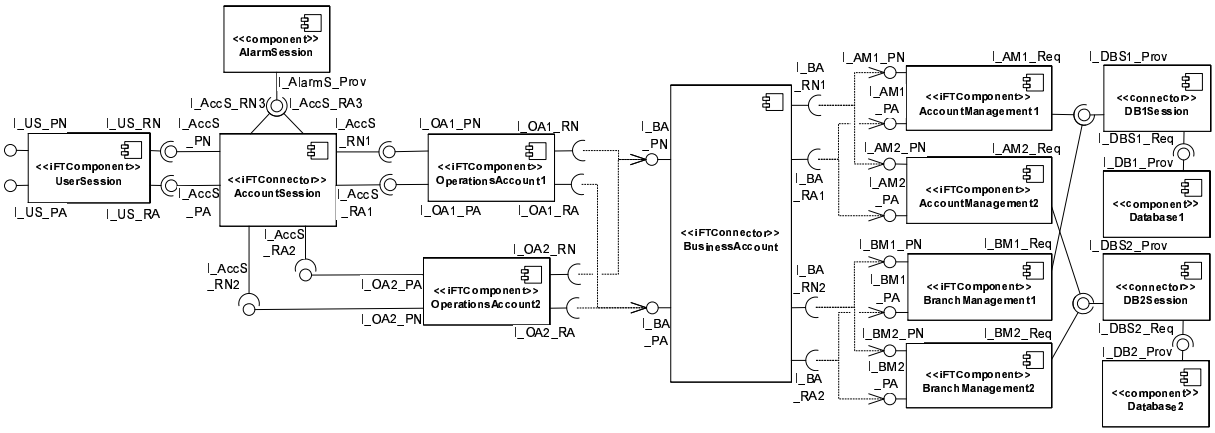


Figura 6.7: Partial software architecture for a reliable and available banking system

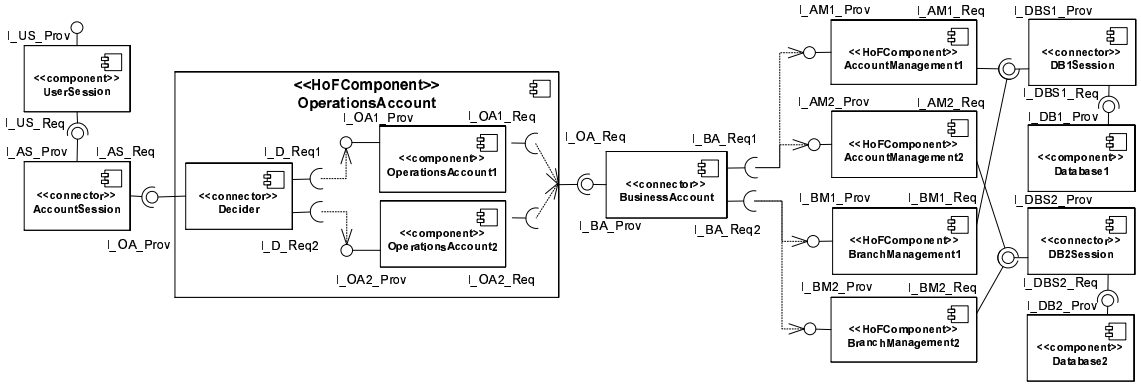


Figura 6.8: Partial software architecture for a reliable and available banking system.

Capítulo 7

Aplicando a Abordagem Rigorosa em Arquiteturas de Linhas de Produtos de Software: Um Estudo de Caso

Esse capítulo apresenta uma adaptação realizada na solução de verificação formal apresentada no Capítulo 4, com o objetivo de possibilitar a representação formal e a verificação de variabilidades arquiteturais relacionadas a tolerância a falhas de software. Esse capítulo foi retirado de dois relatórios técnicos do Instituto de Computação da Unicamp [20, 13], que devem ser formatados em breve e submetidos para conferências. Como o conteúdo do capítulo foi extraído na íntegra desses artigos, foi preservado o idioma original.

7.1 Resumo do Capítulo

Esse capítulo apresenta uma adaptação realizada na solução de verificação formal apresentada no Capítulo 4, com o objetivo de possibilitar a representação formal e a verificação de variabilidades arquiteturais relacionadas a tolerância a falhas de software. Foram considerados dois tipos de variabilidades arquiteturais relacionadas a tolerância a falhas de software: (i) variabilidade relacionada à seleção de diferentes técnicas de redundância explícita para detecção, confinamento e tratamento de erros (Section 7.2); e (ii) variabilidade relacionada à seleção de diferentes comportamentos excepcionais, o que possibilita tanto a seleção de diferentes tratadores de exceções, quanto a seleção de fluxos de controle de exceções (Seção 7.3). A aplicabilidade da abordagem adaptada foi avaliada através de um estudo de caso de um sistema do domínio móvel, que foi apresentado na Seção 2.8.3.

7.2 Variability Related to Fault Tolerance Techniques

This section presents how the proposed verification solution has been extended in order to represent architectural variabilities related to the adoption of different fault tolerance techniques. The feasibility of such extension was evaluated in the context of three fault-tolerance techniques: recovery blocks, n-version programming and n-self-checking programming.

7.2.1 Reference Architectures for Software Fault Tolerance Techniques

As presented in Section 2.2, there are three main techniques for implementing software fault tolerance using design diversity [68]: recovery blocks, N-version programming, and N-self-checking programming. Each technique can be represented by a *reference architecture*, which provides a proven template solution for resolving a specific problem. In the following, we present a reference architectural for each one of the three software fault tolerance technique aforementioned. These reference architectures were designed based on the general structure of each concept, as presented by Laprie [68].

A reference architecture for tolerating a single fault using the *recovery blocks* (RB) technique is shown in Figure 7.1. The Switch is responsible for choosing a proper Alternate to execute the service, in case an error is detected by the AcceptanceTest. The data integrity between two executions is guaranteed by the Checkpoint element.

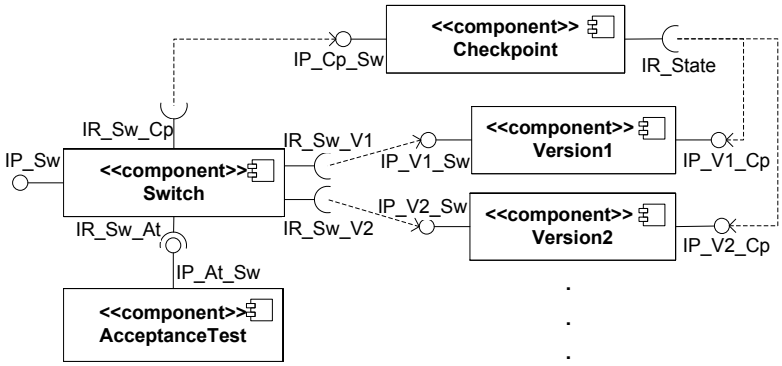


Figura 7.1: Reference Architecture for Recovery Blocks Technique

Figure 7.2 presents a reference architecture for tolerating a single fault using the *N-version programming* (NVP) technique. In this architecture, the Adjudicator connector is responsible to receive the results of all the three versions of components and then judge if there is a reliable result based on majority election.

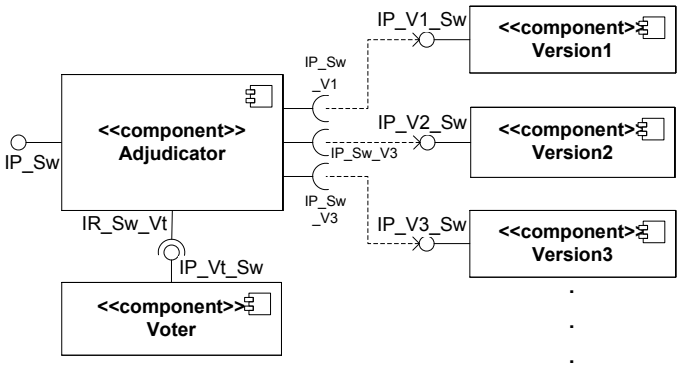


Figura 7.2: Reference Architecture for N-Version Programming Technique

A reference architecture for tolerating a single fault using the *N-self-checking programming* (NSCP) technique with comparison is shown in Figure 7.3. In this case all the Versions are executed in parallel, and the Switch is responsible for switching the result in case an error is detected by the Comparators, which are responsible for comparing two-by-two the results provided by the Versions[1].

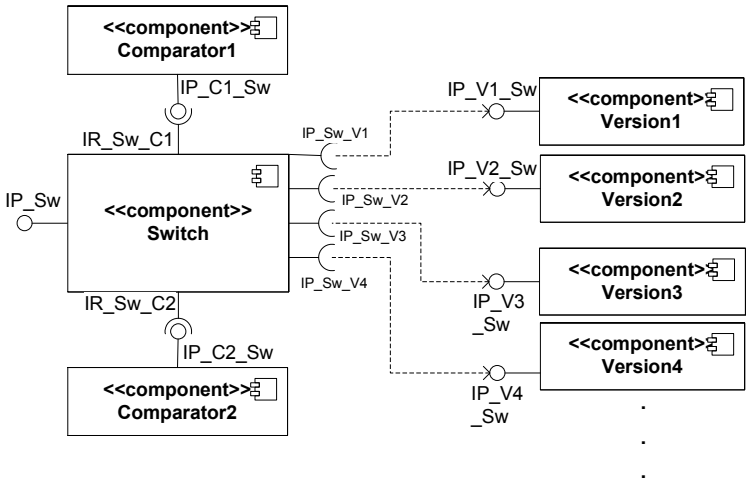


Figura 7.3: Reference Architecture for N-Self-Checking Programming Technique

7.2.2 Feature Model for Software Fault Tolerance Techniques

A feature model for software fault tolerance techniques was derived from the work by Laprie et al. [68]. This feature model, shown in Figure 7.4, follows the representation

[1]Although Laprie et al. [68] justifies the usage of the term 'variant', this paper uses 'version' to avoid conflict with the notion of variant from SPL.

proposed by Ferber et al. [49], and elicits the basic requirements for the design of the PLA. In that diagram, the root feature, SFTArchitecture, is composed of six mandatory features. The Error-processing technique feature captures a set of alternative features related to the different ways that can be employed for detecting errors. The Execution scheme feature represents the two possible ways for executing the system components: either sequential or in parallel. The Number Variants for tolerating f sequential faults represents some characteristics regarding the resources necessary for tolerating "f" faults. The Suspension of service delivery during error processing feature indicates whether the error recovery technique suspends the execution when an error is detected. In case the execution is suspended (the Suspension feature), it is also necessary to define what is the purpose of the suspension: either for re-executing the service, or only for switching to another result. The Judgement on result feature presents how the acceptance test should be performed, either with an absolute criteria (involving the result of only one executor), or a relative criteria (involving the results of more than one executor). Finally, the Consistency of input data feature presents how the consistency of data is achieved, either implicitly through backward error recovery, or explicitly through specialised algorithms of data consistency. According to Figure 7.4, the usage lines between features represent constraints, and each set of alternative features represents a variant (V1 to V7).

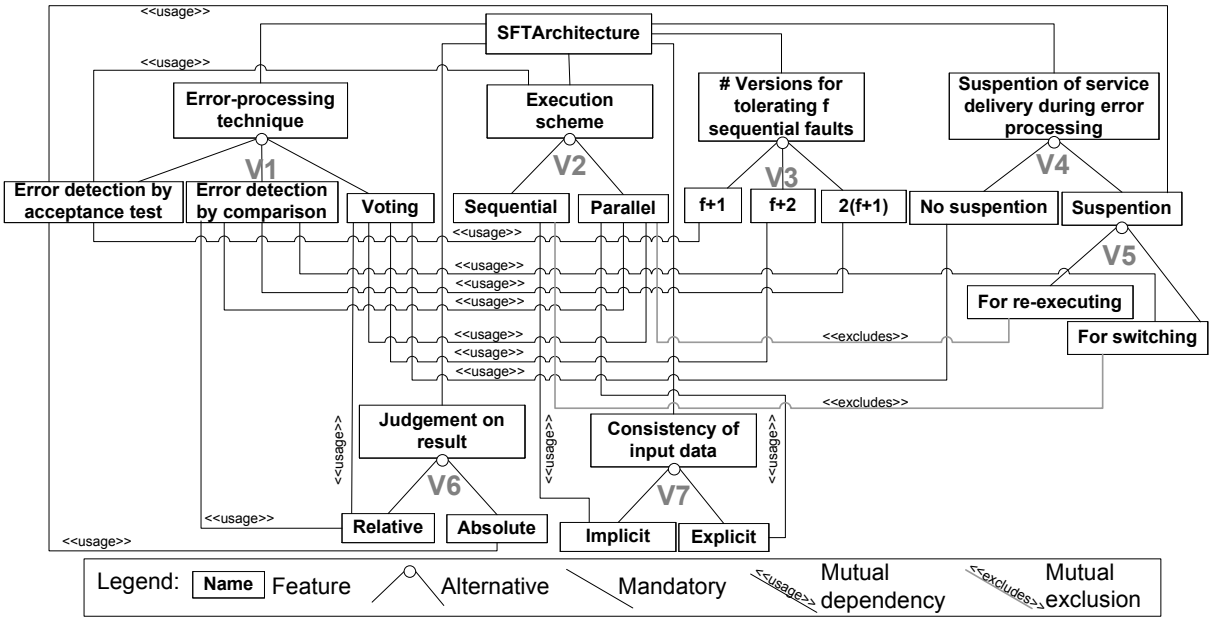


Figura 7.4: Feature Model of Software Fault Tolerance

7.2.3 A PLA for Software Fault Tolerance Techniques

For the design of the product line architecture (PLA) related to the reference architectures of the software fault tolerance techniques three basic steps were followed: (i) design a reference architecture for each of the software fault tolerance techniques; (ii) identify their commonalities and variabilities; and (iii) design the PLA based on these commonalities and variabilities. Figure 7.5 presents a PLA obtained by following the above process, and which realises the feature model of Figure 7.4. The seven variants presented in the feature model (Figure 7.4) were mapped into design decisions represented as variation points. These variation points, which are represented in Figure 7.5 as dashed grey polygons, are used to create the decision model to be employed when instantiating the PLA for generating specific products.

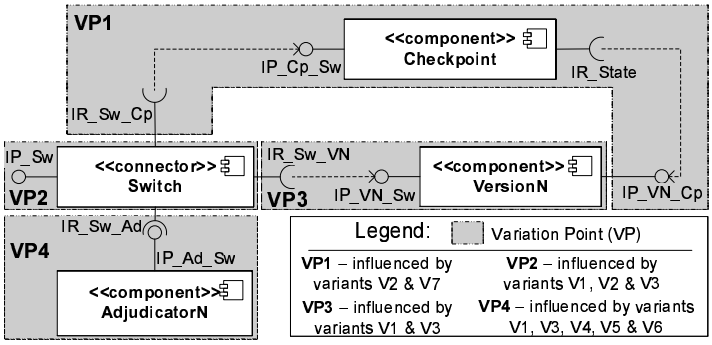


Figura 7.5: PLA for Software Fault Tolerance

Table 7.1 shows the decision model of the PLA presented in Figure 7.5. The goal of this artefact is to list the combination of all variabilities that do not violate the constraints specified in the feature model (Figure 7.4). This information is very useful to identify the impact of the variant choices upon the PLA, and to support the developer in the generation of a specific product without violating any of the system requirements. Each line of Table 7.1 represents the features that can be selected, and the impact the selection has upon the software architecture. Columns V1 to V7 of Table 7.1 present the possible decisions in each variant of the feature model. Column SFT Technique show the technique of software fault tolerance that is used in each scenario. Columns VP1 to VP4 present the impact of the combination of decisions upon the software architecture.

7.2.4 Formal Specification

For the representation of the PLA, shown in Figure 7.5, the B-Method is used for specifying the structure of the PLA, while CSP is employed for specifying its behaviour. Regarding the structural specification, we have defined a hierarchy of B-Method machines,

Tabela 7.1: Decision Model of the PLA for Software Fault Tolerance

Variants of the Feature Model								Variation Points of the PLA			
V1	**V2**	**V3**	**V4**	**V5**	**V6**	**V7**	**SFT Technique**	**VP1**	**VP2**	**VP3**	**VP4**
AccpTst	Seq	f+1	Susp	Re-Exec	Abs	Impl	Recovery Blocks	Used	*switch* between different versions & rollback controller	f+1 Versions	one Adjudicator: *acceptance test*
AccpTst	Par	f+1	Susp	Switch	Abs	Expl	N-self-checking Programming with acceptance test	Not used	*synchronisation* & *switch* between different versions	f+1 Versions	f+1 Adjudicators: *acceptance test*
Comp	Par	2(f+1)	Susp	Switch	Rel	Expl	N-self-checking Programming with comparison	Not used	*synchronisation* & *switch* between different versions	2(f+1) Versions	f+1 Adjudicators: *comparison*
Voting	Par	f+2	No-Susp	—	Rel	Expl	N-version Programming	Not used	*synchronisation*	f+2 Versions	one Adjudicator: *voting*

shown in Figure 7.6, which explicitly separates the commonalities and variabilities of the PLA. Following ProB notation [70], the rectangles represent B-Method machines, and the arrows represent relationships between them. The CSP specifications are represented by dashed ellipses. The featureType and plaTypes machines contain sets that store the data types necessary for representing the feature model and the PLA, respectively. The featureModel machine `uses` the data of featureTypes to represent the feature model presented in Figure 7.4 in terms of its variants, and the relations among features. The pla machine `uses` the data of plaTypes, and the information of the feature model (featureModel) to structure the PLA presented in Figure 7.5 in terms of its architectural configuration, explicit exception channels, and the respective variation points. The `uses` relationship means that the featureModel and pla machines can refer, respectively, to the shared sets of featureTypes and plaTypes for defining its invariants. Finally, each of the other four machines (recoveryBlocks, nvp, nscpAt and nscpComp) have their behavioural scenarios specified in CSP.

The behavioural specification of the PLA is obtained by restricting the B-Method machines with CSP specifications related to them. In the context of the pla machine, the CSP specification defines the behaviour associated with the reference architectures. For example, in the case of the recoveryBlocks machine, the CSP specification states the following restrictions: (i) only a single alternate can be executed in each time (sequential execution); and (ii) before a retry through a different alternate the system state has to be restored using the checkpoint facility.

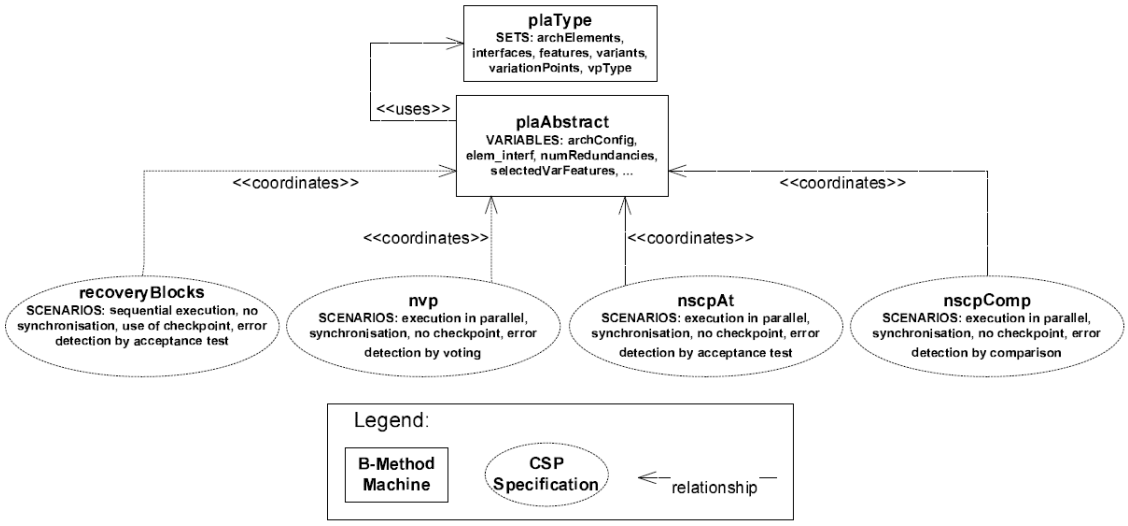


Figura 7.6: Hierarchy of B-Method Machines and CSP Specification

7.2.5 Properties of Interest

We have defined two complementary categories of properties to be verified: (i) decision checking, which checks the consistency between the variants of the feature model and the variation points of the PLA, according to the decision model presented in Table 7.1; and (ii) behavioural checking that verifies the consistency between the behavioural specification of a product against the behaviour of the reference architecture.

The *decision checking* consists of two complementary checks: (i) feature consistency, which verifies the integrity of the selected features when compared with the rules defined in the feature model (see Figure 7.4); and (ii) structural consistency, which verifies the integrity of the software architecture concerning the selected features. An example of a violation of the feature consistency would be the selection of "Voting" as the error processing technique, and "f+1" as the number of redundant versions, which is forbidden according to the feature model (Figure 7.4). An example of a violation of the structural consistency would be the existence of the Checkpoint component in a software architecture when the execution scheme should be parallel.

The objective of *behavioural checking* is to assess the consistency of the behavioural specification of a product according to the behaviour of a reference architecture that implements a specific software fault tolerance technique. The behaviour of each architectural element is assessed according to their specified role in the reference architecture. Examples of roles that are verified for the N-self checking programming technique are: (i) the Versions should be executed in parallel; (ii) the Switch should synchronise the returns from the different Versions, request the error detection to an Adjudicator, and switch the result

in case of error; and (iii) the Adjudicators should detect errors in the values returned by the Versions.

7.2.6 Evaluation 1

One of the claims we have for the proposed approach is that since it explicitly separate type definition, structural variabilities and behavioural variability, the model is easier to evolve in two different ways. First, a common change done in pla (see Figure 7.6) is automatically reflected in all the fault tolerance techniques which are represented on it. Second, it is easy to define new CSP specifications at the behavioural level, which reuse the structural model to deal with other techniques of software fault tolerance.

In order to assess the evolvability of the proposed PLA towards other reference architectures, we have considered two software fault tolerance techniques: (i) *distributed recovery blocks* (DRB), which tolerates both software and hardware faults through the distribution of RBs in different locations [62]; and (ii) *consensus recovery blocks* (CRB), which combines the use of NVP and RB techniques [94]. To illustrate the evolution of PLAs, we will discuss about the adaptation of the PLA presented in Figure 7.5 for also supporting DRB. The adaptation of the PLA for CRB was done in a similar way. For considering the reference architecture of the DRB technique, which is presented in Figure 7.7, first of all it was necessary to update the feature model presented in Figure 7.4. The new feature model permits the selection of two different execution schemes at the same time, as well as, two ways for suspending the execution of service delivery during error processing. For this, variants V1 and V2 had to be changed to multiple selection, instead of alternative selection. In addition, the changes in the feature model had to be reflected in the variation points of the PLA and in the respective decision model presented in Table 7.1. Basically, the decision model had to add another possible decision: the selection of acceptance test as error processing techniques and "f+1" Alternates, which should be instantiated twice, in different physical locations. Finally, since each RB is executed sequentially and the two distributed locations executes in parallel, the execution scheme is considered both sequential and in parallel at the same time. Finally, the PLA also needed to be extended, in order to support many Checkpoints.

To evolve the formal model of the PLA (see Figure 7.6), it was necessary three punctual changes. First, the featureModel machine was updated for reflecting the changes in the feature model. Second, the pla machine was updated in order to modify the variation points and decision model of the software architecture. Third, it was necessary to define a new CSP specification, called drb. This CSP specifies the scenarios according to the expected behaviour of the DRB technique, which is: (i) parallel execution of primary alternate of location 1 (Alternate1.1), and the secondary alternate of location 2 (Alter-

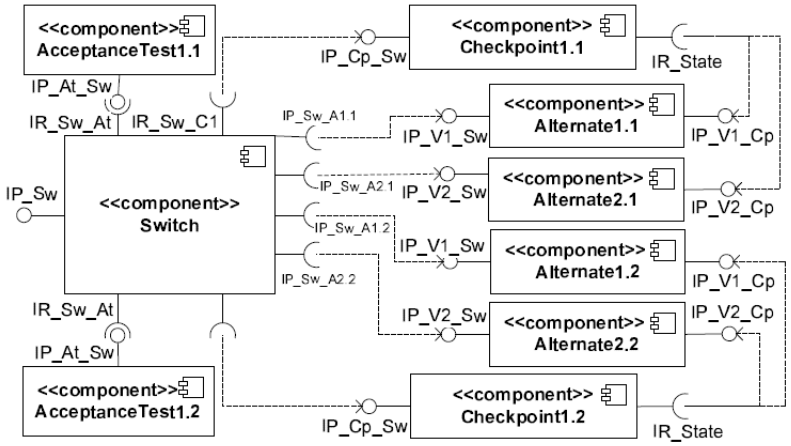


Figura 7.7: Reference Architectures for Distributed Recovery Blocks Technique

nate2.2); (ii) if Alternate1.1 fail, the switch should verify the result of Alternate2.2; (iii) if both fail, it is necessary to execute the roll-back in both physical locations, and (iv) repeat the steps for the Alternate1.2 and Alternate2.1.

7.3 Variability Related to the Abnormal Behaviour

This section presents how the proposed verification solution has been extended in order to represent architectural variabilities related to the exception control flows and handlers. The feasibility of such extension was evaluated in the context of the MobileMedia application, which has been presented in Section 2.8.3.

7.3.1 Exception Handling Model for PLAs

This section presents the exception model adopted by our solution, which was proposed by Cacho *et al.* [25] and extends the Java model by introducing two new concepts: *explicit exception channels* and *pluggable handlers*. We describe these concepts in the following.

Explicit Exception Channels

An *explicit exception channel* (*channel*, for short) is an abstract duct through which exceptions flow from a raising site to a handling site. More precisely, an explicit exception channel (*EEC*) is a 5-tuple consisting of: (i) a set of exception types E, which represents the types of exceptions that flow through the channel; (ii) a set of raising sites RS, which indicates the places where exceptions from E can be raised; (iii) a set of handling sites HS, which indicates the places where exceptions from E are handled. To keep the model

simple, even when the handler raises exceptions, it is not considered a raising site; (iv) a set of intermediate sites *IS*, which are places through which an exception passes from the raising site on its way to the handling site.; and (v) a function *EI*, which defines the list of exceptions that a channel signals to its enclosing context, similarly to Java's throws clause. An example of an *EI* function is the expression ($Ex1 \alpha Ex2$), which defines that exception ($Ex1$) can be mapped to exception ($Ex2$) and propagated to the enclosing exception handling context (EHC).

Pluggable Handlers

A *pluggable handler* is an exception handler that can be associated to arbitrary EHCs, thus separating error handling code from normal code. A single pluggable handler can be associated, for example, to an operation in a component C1, two different operations in another component, C2, and all operations in a third component C3. In this sense, they are an improvement over traditional notions of exception handler. For example, the exception handling model of Guide [67] allows one to bind handlers to blocks, methods, classes, and exceptions, but not all of them at the same time! Another difference is that a pluggable handler exists independently of the EHCs to which it is associated. Therefore, these handlers can be reused both within an application and across different applications.

7.3.2 Formal Representation of the PLA

For the formal representation of the PLA, shown in Figure 2.9 of Section 2.8.3, the B-Method is used for specifying the structure of the PLA, while CSP is employed for specifying its behaviour. It is important to know that the formal specification is automatically generated from XMI files representing the software architecture (see Figure 6.1 of Section 6.2). The adoption of B-Method and CSP was mainly motivated by the explicit separation between the structural and behavioural specifications. We claim that this specification facilitates the comprehension of the formal model, as well as its adjustment for running extra verification. Moreover, the existence of a compliant model checker tool (ProB [70]) was also considered.

Regarding the structural specification, we have defined a hierarchy of B-Method machines, shown in Figure 7.8, that explicitly separates the specification of the feature model and the specification of the PLA. Following ProB notation, the rectangles represent B-Method machines, and the arrows represent relationships between them. The featureTypes and plaTypes machines contain sets that store the data types necessary for representing the feature model and the PLA, respectively. The featureModel machine uses the data of featureTypes to represent the feature model presented in Figure 2.8 in terms of its variants, and the relations among features. The pla machine uses the data of plaTypes,

and the information of the feature model (featureModel) to structure the PLA presented in Figure 2.9 in terms of its architectural configuration, explicit exception channels, and the respective variation points. The `uses` relationship means that the featureModel and pla machines can refer, respectively, to the shared sets of featureTypes and plaTypes for defining its invariants.

It is important to stress that the channels are explicitly represented following the exception model presented in Section 7.3.1. For this, the pla B-Method machine defines relations for representing each element of the channels' five-tuple: raised exceptions, raising sites, handling sites, intermediate sites, and functions of explicit propagation. Moreover, the formal model also represents the exception mappings during the exception control flow, which is important to provide traceability during the model checking process, in order to identify architectural mismatches.

For representing the variabilities associated to the abnormal behaviour, the formal model also associates the exception channels and handlers to decisions of the feature model. For example, in the case of *EEC2* and *EEC3* channels presented in Figure 2.10, the formal model of the MobileMedia case study represents both channels. But when the SMS feature is not selected, EEC2 is considered active, and EEC3 inactive, and the opposite occurs when SMS feature is selected. The same occurs with architectural elements, the respective interfaces, and architectural configurations.

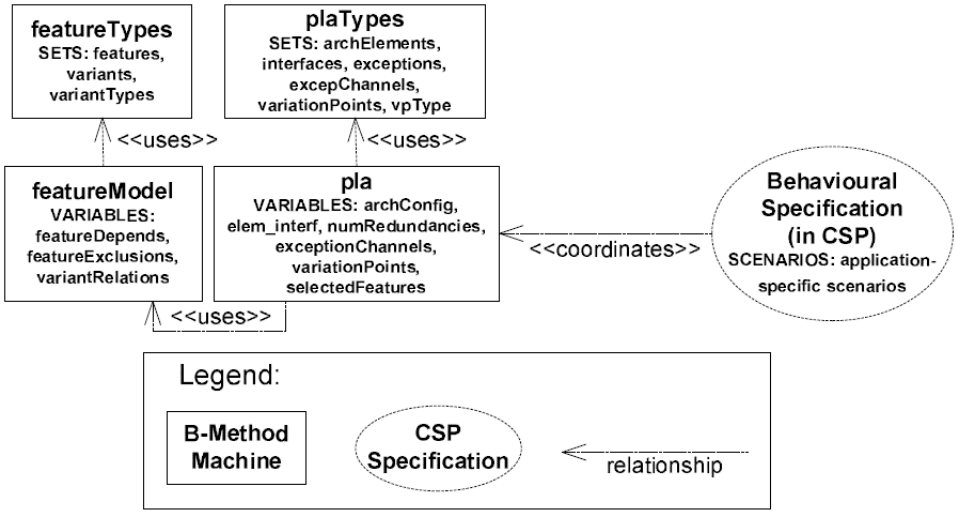


Figura 7.8: Hierarchy of B-Method Machines and CSP Specification

The behavioural specification of the PLA is obtained by restricting the pla B-Method machine with a CSP specification associated to it. This CSP specification defines the interactive behaviour between the architectural elements, including requests, responses and error signalling at the software architecture.

7.3.3 Properties of Interest

We have defined three complementary categories of properties to be verified: (i) decision checking that checks the consistency between the variants of the feature model and the variation points of the PLA, according to the respective decision model; (ii) invalid channels checking that checks the consistency between the exception flows specified in the PLA and the structural restrictions of the software architecture; and (iii) architectural exception mismatches checking that checks the incompatibility between the exceptions propagated from provided interfaces and the respective exceptions expected in required interfaces.

The *decision checking* consists of two complementary checks: (i) feature consistency, which verifies the integrity of the selected features when compared with the rules defined in the feature model (Figure 2.8); and (ii) structural consistency, which verifies the integrity of the software architecture concerning the selected features. An example of a violation of the feature consistency would be the non-selection of the MediaManagement feature, which is mandatory according to the feature model (Figure 2.8). An example of a violation of the structural consistency would be the existence of the SMSController component at the software architecture when the feature SMS is not supposed to be selected. We have defined a total of six consistency properties, which are verified for the featureModel B-Method machine. The properties of structural consistency is specific for each PLA, and is verified for the pla B-Method machine. In the context of the MobileMedia case study, we have specified a total of 24 properties, one for each possible decision of the product line architecture. These decisions concerns the possible selection scenarios of alternative and optional features (see Figure 2.8).

The objective of *invalid channels checking* is to assess if all the exception flows can actually occur in the PLA. This checking is conducted in two complementary perspectives: (i) verification of the architectural configuration, in order to identify architectural elements that, although should be part of a flow, it cannot be as a consequence of a missing dependency in the architectural configuration; and (ii) verification of the exception masking, in order to identify some architectural elements that, although should be part of a flow, it cannot be as a consequence of exception masking before it should happen (impossible propagation). We have specified four properties for verifying invalid channels.

The *architectural exception mismatches checking* focus on the exception propagation between architectural elements, and intends to be sure that all the exceptions propagated from an element to another is proper interpreted. In this way, the propagated exception should either be the same, or an explicit conversion should be specified. Since the behavioural specification states that exceptions should be propagated until the end of the flow, in case of architectural exception mismatches, the flow is obliged to stop in an unpredictable way. The model checker detects these mismatches as deadlocks.

7.3.4 Evaluation 2

Overall, this case study showed that using explicit exception channels and pluggable exception handlers it is possible to develop and evolve product line architectures (PLAs) with variable abnormal behaviour. The formal representation of the PLA provided the means for applying model checking in the verification of key architectural properties, such as, the feasibility of the specified channels, the existence of uncaught exceptions, and the explicit declaration of exception propagation. In the following, we summarise the experience obtained while applying the proposed solution to the development and evolution of the MobileMedia system.

During the verification of the MobileMedia system, the model checker detected two deadlocks, and the violation of two explicit exception channels (EECs). The deadlocks were caused by architectural mismatches in two channels, when no proper conversion was provided between exception types (see Figure 2.10). Concerning the violation of channels, two EECs from the ModelManager component to the PresentationDialog component (see Figure 2.9) were considered impossible because the software architecture did not allow the propagation of the respective exceptions that where propagated among the involved architectural elements. Regarding the existence of exceptions that were not caught by any handler (uncaught exceptions), the model checker detected a single case: in the PresentationDialog layer, an exception handler has not been associated with the exception *PersistenceMechanismError*.

The relevance of the above rigorous analysis of a system based on EECs can be confirmed when considering that the outcome of a previous static analysis of the same MobileMedia system [34] without EECs has identified that approximately 67,5% of the exceptions were not caught by any handler. The possibility of localising and correcting design faults at the earlier stages of the software development provide greater benefit compared to when these are left to be fixed at the later phases of development.

In order to assess the evolvability of the proposed solution towards the addition of new features, we have considered two different releases of the MobileMedia system [34]: (i) *release V4*, which permits the user to view photos and to make a catalogue of favourites; and (ii) *release V6*, which also permits the sending of photos through SMS. First, we have specified, verified, and implemented the PLA of release V4, second, we have evolved that PLA (and the respective formal model) for release V6 by adding new architectural elements and finally, we have updated the decision model of the PLA by incorporating the feature SMS.

For evolving the formal model of the PLA (see Figure 7.8), three punctual changes, both in the pla B-Method machine, were necessary. First, pla was updated for reflecting the new decision model, and incorporating the SMS feature. Second, the pla was updated in order to add new architectural elements, the respective interfaces, and the new EECs.

Finally, it was also necessary to change the content of two relations of pla: (i) association between features and EECs; and (ii) association between features and architectural elements. As presented in Section 7.3.2, these relations enable the activation and deactivation of explicit exception channels and architectural configurations, depending on the features that are being selected for a specific product.

7.4 Summary

This chapter presented a case study that used the verification approach presented in Chapter 4 for verifying fault-tolerant Product Line Architectures (PLAs). First, the verification framework presented in Section 4.2 was extended in order to allow the specification of architectural variabilities related to the choice of different fault tolerance techniques. Second, the same framework was also extended to allow the specification of architectural variabilities related to the choice of different exception handlers. The feasibility of such extensions has been assessed in the context of two case studies.

It was important to extend the verification solution in the context of PLAs, since it is considered an important technology for maximizing the software reuse among application of a common domain.

Capítulo 8

Conclusões e Trabalhos Futuros

Esse capítulo apresenta as considerações finais dessa tese, assim como as direções para trabalhos futuros. As contribuições da abordagem proposta também são apresentadas, assim como as publicações científicas fruto dos resultados obtidos.

8.1 Conclusão

Esta tese apresentou uma abordagem rigorosa para o desenvolvimento e teste de arquiteturas de software tolerantes a falha. O foco da tese foi em uma solução integrada envolvendo especificação formal, verificação e teste do fluxo de controle de exceções e dos tratadores na arquitetura de software. Essa abordagem foi construída utilizando o conceito de abstrações arquiteturais, que tem como objetivo melhorar o controle da complexidade da arquitetura de software, além de melhorar a escalabilidade da sua verificação formal. Duas abstrações arquiteturais foram apresentadas: (i) o elemento arquitetural tolerante a falhas idealizado (iFTE), uma abstração arquitetural baseada no mecanismo de tratamento de exceções, que é utilizado para estruturar no nível de arquitetura de software sistemas com requisitos de confiança no funcionamento; e o (ii) elemento arquitetural falha-e-pára (HoFE), uma abstração arquitetural que implementa o princípio de falha permanente e é utilizado para estruturar sistemas tolerantes a falhas com redundância explícita no nível arquitetural.

Em relação à representação formal, a abordagem proposta combinou o uso de B-Method e CSP para representar conceitos como tipos de exceções, fluxos de controle de exceções e tratadores em termos de cenários. Os cenários e propriedades arquiteturais também são utilizados para gerar casos de teste de unidade, de integração e de robustez, com o objetivo de aferir a consistência entre a arquitetura de software verificada e o código-fonte final da aplicação. Essa aferição é necessária mesmo quando a arquitetura de software é verificada formalmente, uma vez que não há garantias quanto a consistência

entre a configuração arquitetural verificada e o seu respectivo código-fonte.

A abordagem rigorosa pretendida inicialmente foi estendida de forma a possibilitar a representação de arquiteturas de linha de produtos com variabilidades arquiteturais relacionadas a diferentes técnicas de tolerância a falhas. Combinando o uso de B-Method e CSP, essa abordagem representa explicitamente os elementos arquiteturais, suas configurações, variantes e pontos de variação, assim como o comportamento associado às arquiteturas de referência de cada técnica de tolerância a falhas de software. O objetivo dessa extensão foi separar explicitamente comunalidades e variabilidades do modelo formal da arquitetura da linha de produto (PLA). Essa separação promove o reúso tanto do modelo formal, quanto de parte da verificação dos produtos derivados da PLA. Além disso, ao invés de considerar o sistema como um todo, a classificação das comunalidades e variabilidades de uma família de sistemas tolerantes a falhas deve também melhorar a escalabilidade da verificação. Essa extensão também possibilitou a representação explícita de variabilidades associadas aos fluxos de controle de exceções da arquitetura de software, além da especificação de tratadores de exceções variáveis (plugáveis). Os resultados preliminares obtidos na pesquisam fazem acreditar que as novas características do modelo formal podem trazer quatro vantagens ao desenvolvedor de software: (i) facilita a compreensão dos fluxos de controle de exceções de forma localizada, sem a necessidade de examinar outras partes do programa; (ii) melhora a modularidade do modelo, facilitando até o reúso do comportamento excepcional associado às comunalidades da família; (iii) promove uma maior manutenibilidade das especificações normais e excepcionais através da separação dos tratadores e da especificação de variabilidades associadas a eles; e (iv) possibilita a especificação de variabilidade relacionada aos flucos de controle de exceções e os respectivos tratadores arquiteturais da PLA.

Os estudos de caso utilizados para avaliar a abordagem completa mostraram que a abordagem proposta oferece uma solução apropriada para modelar, analisar e testar sistemas de software tolerantes a falhas no nível da arquitetura de software.

Uma limitação da solução proposta é a assunção de que a comunicação entre os elementos arquiteturais segue um protocolo de chamada e requisição[1]. Apesar dessa assunção simplificar a especificação do modelo formal das configurações arquiteturais, ela é restritiva quanto a aspectos ligados a concorrência, que são importantes no contexto de um grande número de aplicações. Uma possível solução para essa limitação seria estender a solução proposta nessa tese de forma a possibilitar a especificação de arquiteturas de software baseadas em eventos, o que poderia ser feito com a definição de uma nova abstração arquitetural. O conceito de abstrações arquiteturais e cenários adotado no trabalho oferece benefícios consideráveis no contexto da área de arquitetura dirigida por modelo[2] (MDA).

[1]Do inglês *call-return*

[2]Do inglês *Model-Driven Architecture*

Primeiramente, uma vez que abstrações e cenários podem ser especificados em diferentes níveis de detalhes, o seu refinamento pode guiar a transformação de um modelo independente de plataforma para um modelo de plataforma específica. Para apoiar MDA, seria necessário desenvolver ferramentas para automatizar a transformação de modelos, incluindo a geração de código-fonte. Em relação aos testes baseados no modelo, a abordagem proposta não abrange a geração dos dados de teste (dados de entrada e oráculo de teste). Para suprir essa carência, seria necessário utilizar uma abordagem complementar, como por exemplo teste de mutação [39].

Em relação às seis questões de pesquisa apresentadas na Seção 1.2, todas elas foram tratadas pela solução arquitetural proposta:

1. As técnicas de verificação formal e validação podem ser combinadas para definir um processo rigoroso de desenvolvimento e testes centrado na arquitetura?

   O processo rigoroso apresentado no Capítulo 6 mostrou como métodos formais podem ser utilizados como base para verificar e validar arquiteturas de software. Além disso, abstrações arquiteturais também foram especificadas de forma a melhorar o controle da complexidade do sistema associada a tolerância a falhas de software.

2. Os cenários de uma configuração arquitetural (cenários arquiteturais) são boas referências tanto para a verificação quanto para a validação de sistemas tolerantes a falhas?

   Apesar dos cenários arquiteturais possuírem descrições comportamentais de alto nível, após analisar os resultados apresentados nos Capítulos 4 e 5, eles foram considerados boas referências para se verificar e validar sistemas de software em relação a requisitos não-funcionais, tal como confiança no funcionamento. Isso ocorre especialmente devido à arquitetura de software tornarem explícitas as interações entre as partes estruturais de um software.

3. É possível definir variabilidades arquiteturais relacionadas às decisões de projeto relativas a tolerância a falhas de software?

   Como apresentado no Capítulo 7, a abordagem proposta foi estendida de forma a representar variabilidades arquiteturais relacionadas a decisões de projetos associadas a tolerância a falhas de software; uma arquitetura e linha de produto foi apresentada na Seção 7.2.3.

4. Os cenários arquiteturais representam um critério de cobertura de teste apropriado no contexto de sistemas tolerantes a falhas?

Após a geração dos casos de teste, que foi apresentada no Capítulo 5, cenários arquiteturais foram considerados boas referências para cobertura dos testes, uma vez que eles possibilitaram a identificação de novos erros, que por sinal não foram identificados utilizando estratégias tradicionais de teste.

5. Abstrações arquiteturais são capazes de ocultar efetivamente a complexidade do sistema relacionada a tolerância a falhas de software?

   Como apresentado no Capítulo 6, a utilização de abstrações arquiteturais reduziu consideravelmente o número de elementos arquiteturais de alto nível, uma vez que cada um desses elementos é capaz de encapsular dentro de si o seu próprio mecanismo de detecção e tratamento de erros. Além disso, abstrações arquiteturais possibilitam uma decomposição uniforme da arquitetura [63] através da especificação da estrutura interna dos elementos arquiteturais em um segundo nível de projeto.

6. Abstrações arquiteturais melhoram o entendimento[3] da verificação formal e validação?

   A redução da complexidade da arquitetura de software também influenciou o modelo formal e o número de casos de teste gerados. Essa interferância ocorreu primeiramente devido ao uso de abstrações arquiteturais, o que reduziu o número de elementos arquiteturais do modelo formal; em segundo lugar, a definição de padrões comportamentais para a arquitetura de software (cenários arquiteturais) ajudou a guiar tanto a especificação de propriedades a serem verificadas, quanto a geração de casos de teste baseados no modelo da arquitetura.

8.2 Contribuições

As contribuições dessa tese cobre algumas sub-áreas da engenharia de software, em particular, áreas relacionadas a confiança no funcionamento do software: verificação formal de arquiteturas de software, teste baseado em modelos, tolerância a falhas de software e tratamento de exceções. A seguir, as contribuições da tese são listadas e descritas brevemente.

8.2.1 Principais Contribuições

As principais contribuições, relacionadas ao novo trabalho desenvolvido, são:

[3] Do inglês *understandability*

1. **Um processo rigoroso para o desenvolvimento de arquiteturas de software tolerantes a falhas.** Foi proposto um processo iterativo, recursivo e incremental para desenvolver arquiteturas de software tolerantes a falhas (Capítulo 6). Nesse processo, abstrações arquiteturais são consideradas unidades de primeira ordem, guiando o desenvolvimento desde a especificação da arquitetura à implementação da aplicação.

2. **Uma abstração arquitetural baseada em tratamento de exceções (redundância implícia).** O elemento arquitetural tolerante a falhas idealizado (iFTE) é uma abstração arquitetural para estruturar sistemas tolerantes a falhas. Essa abstração se baseia nos princípios associados ao conceito de componente tolerante a falhas idealizado, apresentado na Seção 2.3, além de incorporar mecanisms para detecção de erros, assim como a propagação e tratamento de exceções de maneira estruturada.

3. **Uma abstração arquitetural baseada no princípio falha-e-pára (redundância explícita).** O elemento arquitetural falha-e-pára (HoFE) é uma abstração arquitetural para promover a detecção e o confinamento de erros em sistemas tolerantes a falhas. Essa abstração segue os princípios associados ao modelo de falhas permanentes[4] [93]. Quando um HoFE falha, a falha é silenciosa e não produz nenhum sinal de erro. Sendo assim, componentes e conectores HoFE podem ser utilizados com o intuito de abstrair detalhes da lógica de detecção e confinamento de erros, o que apoia diretamente a construção de configurações arquiteturais tolerantes a falhas.

4. **Uma solução formal para verificar arquiteturas de software baseado em cenários arquiteturais.** Primeiramente, modelos formais podem ser utilizados para representar os elementos arquiteturais baseado na respectiva abstração pretendida e na forma como os elementos interagem entre si. Segundo, o modelo formal da configuração arquitetural pode ser verificado formalmente em relação a propriedades associadas a tolerância a falhas de software. A verificação considera cenários arquiteturais de detecção de erros,tratamento de erros e de falha, além da propagação estruturada de erros.

5. **Uma solução formal para verificar arquiteturas de linha de produtos tolerantes a falhas.** Como apresentado no Capítulo 4, a solução formal apresentada anteriormente (Item 4) foi estendida com o intuito de possibilitar a representação de variabilidades arquiteturais associadas às decisões de projeto relacionadas a tolerância a falhas de software.

[4]Do inglês *crash failures*

8.2.2 Contribuições Secundárias

As contribuições secundárias, relacionadas a adaptações realizadas em trabalhos existentes, são:

1. **Uma solução para validar aplicativos de software baseado em cenários arquiteturais.** Algumas soluções de teste existentes foram adaptadas para gerar casos de teste de unidade [90], de integração [26] e de robustez [77] baseados em modelos formais da arquitetura de software. Basicamente, são gerados casos de teste e a melhor sequência de execução a partir de cenários arquiteturais de detecção, propagação e tratamento de erros, além do tratamento de falhas.

2. **Geração automática de código-fonte baseada em abstrações arquiteturais.** Moronte et al. [79] apresentaram uma ferramenta de transformação de modelos que gera código-fonte Java a partir de configurações arquiteturais. O código-fonte gerado é compatível com o modelo de implementação de componentes COSMOS*, utilizado nesse trabalho e apresentado na Seção 2.7. Na presente tese, a ferramenta citada foi estendida com o propósito de gerar código-fonte dos elementos arquiteturais de alto nível, de acordo com a abstração arquitetural que eles se baseiam.

8.2.3 Contribuições Não-Relacionadas

As contribuições não relacionadas se referem a trabalhos relacionados à área de pesquisa da tese, mas que estão fora do seu foco principal. Essas contribuições não foram apresentadas no contexto da tese e são descritas brevemente aqui. As contribuições não relacionadas consiste normalmente de trabalhos conjuntos envolvendo outros membros do grupo de pesquisa em engenharia de software e confiança no funcionamento[5] do IC/UNICAMP.

1. **Um método baseado em reúso para desenvolver sistemas confiáveis baseados em componentes.** Essa contribuição é complementar ao trabalho apresentado nessa tese e consiste em um método de desenvolvimento compatível com o princípio de desenvolvimento com reúso [98]. O método sistematiza as atividades de especificação do comportamento excepcional e utiliza verificação formal durante a fase de projeto arquitetural. Em outras palavras, a solução formal apresentada no Capítulo 4 foi utilizada como ferramenta durante a execução do processo de desenvolvimento.

2. **Projeto e implementação de variabilidades arquiteturais.** Outra contribuição complementar consiste no projeto e implementação de variabilidades arquiteturais baseadas no modelo de implementação de componentes COSMOS*. Um

[5]Do inglês *Software Engineering and Dependability Research Group*

estudo de caso foi conduzido com o intuito de desenvolver uma arquitetura de linha de produto para sistemas de celular com variabilidades relacionadas ao comportamento excepcional.

8.3 Publicações

Esta seção apresenta as publicações associadas às contribuições apresentadas na Seção 8.2.

8.3.1 Publicações Aceitas

As publicações aceitas durante o período de execução dessa tese foram:

1. **Architecting Fault Tolerance using Abstractions [17].** Esse resumo estendido apresenta uma visão geral do processo rigoroso de desenvolvimento e teste apresentado no Capítulo 6. Essa publicação está relacionada à contribuição principal 1.

2. **Development of Fault-Tolerant Software Systems based on Architectural Abstractions [18].** Esse artigo detalha o processo rigoroo de desenvolvimento e teste apresentado no Capítulo 6, oferecendo informações detalhadas sobre as atividades de verificação e validação. Essa publicação está relacionada à contribuição principal 1.

3. **Architecture-Centric Fault Tolerance with Exception Handling [15].** Esse artigo detalha as atividades de verificação formal dos elementos arquiteturais baseados na abstração arquitetural iFTE, mas não contempla a verificação de configurações arquiteturais. Esse artigo também apresenta detalhes sobre a geração de casos de teste de unidade. Essa publicação está relacionada às contribuições principais 2 e 4, além da contribuição secundária 1.

4. **Verification and Validation of a Fault-Tolerant Architectural Abstraction [16].** Esse artigo apresenta uma versão preliminar da solução de verificação formal de abstrações arquiteturais iFTE, incluindo a geração de casos de teste de integração. Essa publicação está relacionada com as contribuições principais 2 e 4, além da contribuição secundária 1.

5. **Verification of Exception Control Flows and Handlers Based on Architectural Scenarios [36].** Esse artigo detalha as atividades de verificação do processo rigoroso que foi apresentado no Capítulo 4. Essa publicação está relacionada com a contribuição principal 4.

6. **Design and Implementation of Architectural Variabilities based on COSMOS* Component Model using Aspects [45] (co-author).** Esse artigo curto apresenta uma extensão feita no modelo COSMOS* com o objetivo de permitir a implementação de variabilidades arquiteturais. A solução para verificar formalmente PLAs não é apresentada nessa publicação. Essa questão deve ser coberta em uma publicação futura. Essa publicação está associada à contribuição não relacionada 2.

7. **Explicit Exception Handling Variability in Component-based Product Line Architectures [10] (co-author).** Esse artigo apresenta uma solução para desenvolver PLAs com variabilidade no comportamento excepcional. A verificação formal de PLAs também não foi coberta nessa publicação. Essa publicação está associada à contribuição não relacionada 2.

8. **Um Método Baseado em Reuso Para o Desenvolvimento de Sistemas Confiáveis Baseados em Componentes [35].** Esse artigo apresenta um processo para maximizar o reúso de componentes de software durante o desenvolvimento de sistemas confiáveis. Essa publicação está associada à contribuição não relacionada 1.

9. **Managing Variabilities in Component-Based Software Product Lines [72] (co-author).** Esse artigo apresenta uma solução para gerenciar variabilidades no contexto de linhas de produtos de software. Essa publicação está associada à contribuição não relacionada" 2.

10. **Architecting Fault Tolerance with Exception Handling: Verification and Validation [19].** Esse artigo de periódico apresenta a solução geral proposta nessa tese, detalhando tanto as atividades de verificação, quanto as atividades de validação. Essapublicaçãoestá relacionada às contribuições principais 2 e 4, além da contribuição secundária 1.

11. **Technical Reports [20, 13, 37, 40, 53, 52].** Além das publicações já citadas, outros seis relatórios técnicos foram publicados no Instituto de Computação da Universidade Estadual de Campinas (UNICAMP). Esses relatórios técnicos apresentam resultados preliminares relacionados às contribuições principais 4 e 5, além das contribuição secundária 1 e das contribuições não relacionadas 1 e 2.

8.3.2 Publicações a Serem Submetidas

Com o objetivo de publicar as demais contribuições da tese, as seguintes publicações devem ser escritas:

1. **A method for developing and testing the abnormal behaviour of component-based software architectures.** Esse artigo está em fase de conclusão e foca na condução sistemática das fases de especificação e teste no contexto do comportamento excepcional de sistemas baseados em componentes. Esse artigo será submetido a um periódico e está associado à contribuição não relacionada 1.

2. **COSMOS*: a COmponent System MOdel for Software Architectures.** Este artigo também está em fase de conclusão e detalha o modelo de implementação de componentes COSMOS*, apresentado na Seção 2.7, além da sua avaliação no contexto de um estudo de caso real. Esse artigo será submetido a um periódigo e está associado à contribuição secundária 2.

3. **Formal Verification of Fault-Tolerant Product Line Architectures.** Esse artigo tem o objetivo de detalhar a solução proposta relativa à representação formal de variabilidades arquiteturais relativas ao comportamento excepcional de sistemas. Essa solução deve ser submetida a uma conferência internacional e está associada à contribuição principal 5.

4. **Robustness Testing Based on Architectural Scenarios and Abstractions.** Esse artigo tem o objetivo de detalhar a atividade de geração automática de casos de teste de robustez, baseado em cenários e abstrações arquiteturais. Além dos casos de teste propriamente ditos, a solução também contempla a identificação das falhas a serem injetadas. Essa solução, que foi apresentada no Capítulo 5, deve ser submetida a uma conferência internacional e está associada à contribuição secundária 1.

5. **Automatic Source Code Generation Based on Architectural Abstractions.** Esse artigo tem o objetivo de detalhar o procedimento de geração automática de código-fonte baseado no modelo de implementação de componentes COSMOS*, de acordo com abstrações arquiteturais tolerantes a falhas. Essa solução deve ser submetida a uma conferência e está associada à contribuição secundária 2.

Referências Bibliográficas

[1] Jean-Raymond Abrial, Matthew K. O. Lee, Dave Neilson, P. N. Scharbach, and Ib Sorensen. The B-Method. In *Proc. of the 4th International Symposium of VDM Europe on Formal Software Development (VDM '91)*, volume 2, pages 398–405, 1991.

[2] Colin Atkinson, Joachim Bayer, Christian Bunse, Erik Kamsties, Oliver Laitenberger, Roland Laqua, Dirk Muthig, Barbara Paech, Jürgen Wüst, and Jörg Zettel. *Component-based Product Line Engineering with UML*. Addison-Wesley, 2002.

[3] Atlassian Clover. Code coverage analysis. http://www.atlassian.com/software/clover/, [December 2007].

[4] Marko Auerswald, Martin Herrmann, Stefan Kowalewski, and Vincent Schulte-Coerne. Reliability-oriented product line engineering of embedded systems. In *Proc of the 4th International Workshop on Software Product-Family Engineering (PFE'01), LNCS 2290*, pages 83–100, London, UK, 2002.

[5] A. Avizienis. The n-version approach to fault-tolerant software. *IEEE Transactions on Software Engineering*, 11(12):1491–1501, 1985.

[6] A. Avizienis, J.-C. Laprie, B. Randell, and C. Landwehr. Basic concepts and taxonomy of dependable and secure computing. *IEEE Transactions on Dependable and Secure Computing*, 1(1):11–33, January-March 2004.

[7] Len Bass, Paul C. Clements, and Rick Kazman. *Software Architecture in Practice*. Addison-Wesley, 2nd edition, 2003.

[8] Boris Beizer. *Black-Box Testing: Techniques for Functional Testing of Software and Systems*. Wiley, 1995.

[9] A. Bertolino, P. Inverardi, H. Muccini, and A. Rosetti. An approach to integration testing based on architectural descriptions. In *Proc. of the Third IEEE International*

Conference on Engineering of Complex Computer Systems (ICECCS '97), pages 77–85, Washington, DC, USA, 1997. IEEE Computer Society.

[10] Ivo Bertoncello, Marcelo de O. Dias, Patrick H. S. Brito, and Cecília M. F. Rubira. Explicit exception handling variability in component-based product line architectures. In *Proc. of the 4th international workshop on Exception handling (WEH'08)*, pages 47–54, 2008.

[11] Colin Blundell, Kathi Fisler, Shriram Krishnamurthi, and Pascal Van Hentenryck. Parameterized interfaces for open system verification of product lines. *Automated Software Engineering (ASE'04)*, 0:258–267, 2004.

[12] J. S. Bradbury. Organizing definitions and formalisms for dynamic software architectures. Technical Report 2004-477, School of Computing, Queen's University, March 2004.

[13] Patrick H. S. Brito, Nelio Cacho, Alessandro Garcia, Cecília M. F. Rubira, and Rogério de Lemos. Design, verification and implementation of exception control flows for product line architectures. Technical Report IC-09-11, Institute of Computing, University of Campinas, March 2009.

[14] Patrick H. S. Brito, Rogério de Lemos, Eliane Martins, Regina Moraes, and Cecília M. F. Rubira. Architectural-based validation of fault-tolerant software. In *Submitted to the 4th Latin American Symposium on Dependable Computing (LADC 2009). The result is not available yet.*, João Pessoa, Brazil, 2009.

[15] Patrick H. S. Brito, Rogério de Lemos, Eliane Martins, and Cecília M. F. Rubira. Architecture-centric fault tolerance with exception handling. In *Proc of the 3rd Latin American Symposium on Dependable Computing (LADC 2007), LNCS*, volume 4746, pages 75–94, Morelia, Mexico, 2007.

[16] Patrick H. S. Brito, Rogério de Lemos, Eliane Martins, and Cecília Mary F. Rubira. Verification and validation of a fault-tolerant architectural abstraction. In *Proc of the Workshop on Architecting Dependable Systems (WADS)*, pages 1–6, Edinburgh, UK, 2007.

[17] Patrick H. S. Brito, Rogério de Lemos, and Cecília M. F. Rubira. Architecting fault tolerance using abstractions (extended abstract). In *Complementary issue of the 38th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN 2008)*, pages 1–2, 2008.

[18] Patrick H. S. Brito, Rogério de Lemos, and Cecília M. F. Rubira. Development of fault-tolerant software systems based on architectural abstractions. In *Second European Conference on Software Architecture, LNCS*, volume 5292, pages 131–147, Paphos, Cyprus, September 2008. Springer.

[19] Patrick H. S. Brito, Rogério de Lemos, Cecília M. F. Rubira, and Eliane Martins. Architecting fault tolerance with exception handling: Verification and validation. *Journal on Computer Science & Technology (JCST)*, (accepted for publication).

[20] Patrick H. S. Brito, Rogério de Lemos, and Cecília M. F. Rubira. Verifying architectural variabilities in software fault tolerance techniques. Technical Report IC-09-10, Institute of Computing, University of Campinas, March 2009.

[21] Patrick H. S. Brito, Camila Ribeiro Rocha, Fernando Castor Filho, Eliane Martins, and Cecília M. F. Rubira. A method for modeling and testing exceptions in component-based software development. In *Proc. of the 2nd Latin American Symposium on Dependable Computing (LADC 2005), LNCS 3747*, pages 61–79, 2005.

[22] S. D. Brookes, C. A. R. Hoare, and A. W. Roscoe. A theory of communicating sequential processes. *J. ACM*, 31(3):560–599, 1984.

[23] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. *Pattern-oriented software architecture: a system of patterns*. John Wiley & Sons, Inc., New York, NY, USA, 1996.

[24] Michael J. Butler and Michael Leuschel. Combining CSP and B for specification and property verification. In *Proc. of Formal Methods*, LNCS 3582, pages 221–236. Springer, 2005.

[25] Nelio Cacho, Fernando Castor Filho, Alessandro Garcia, and Eduardo Figueiredo. Ejflow: Taming exceptional control flows in aspect-oriented programming. In *7th Int. Conf. on Aspect-Oriented Software Development (AOSD'08)*, pages 72–83, 2008.

[26] Josiane Aparecida Cardoso. A method of integration testing for system based on components (in portuguese). Master's thesis, IC, Unicamp, Fevereiro 2006.

[27] John M. Carroll. Making use: scenarios and scenario-based design. In *Proc. of the Conference on Designing Interactive Systems (DIS '00)*, page 4, New York, NY, USA, 2000. ACM Press.

[28] Fernando Castor Filho, Patrick H. S. Brito, and Cecília M. F. Rubira. Specification of exception flow in software architectures. *Journal of Systems and Software*, 79(10):1397–1418, 2006.

[29] Fernando Castor Filho, Nelio Cacho, Eduardo Figueiredo, Raquel Ferreira, Alessandro Garcia, and Ceília Mary F. Rubira. Exceptions and aspects: The devil is in the details. In *Proceedings of the 14th ACM SIGSOFT FSE*, pages 152–162, November 2006.

[30] Fernando Castor Filho, Paulo Asterio de C. Guerra, and Cecília M. F. Rubira. An architectural-level exception-handling system for component-based applications. In *Proc. of the 1st Latin-American Symposium on Dependable Computing*, LNCS 2847, pages 321–340, 2003.

[31] Byeong-Mo Chang, Jang-Wu Jo, Kwangkeun Yi, and Kwang-Moo Choe. Interprocedural exception analysis for java. In *Proc. of the 2001 ACM Symposium on Applied Computing (SAC '01)*, pages 620–625, New York, NY, USA, 2001. ACM Press.

[32] John Cheesman and John Daniels. *UML Components*. Addison-Wesley, 2000.

[33] P. Clements et al. *Documenting Software Architectures: Views and Beyond*. Addison-Wesley, 2003.

[34] Roberta Coelho et al. Assessing the impact of aspects on exception flows: An exploratory study. In *Proc. of the European Conference on Object-Oriented Programming (ECOOP'08)*, pages 207–234, 2008.

[35] Patrick H. da S. Brito, Paulo A. de C. Guerra, and Cecília M. F. Rubira. A reuse-based method for the development of dependable component-based systems (in portuguese). In *Proc. of the Workshop on Fault Tolerance 2007*, Belém, Brazil, 2007.

[36] Patrick Henrique da S. Brito, Rogério de Lemos, and Cecilía M. F. Rubira. Verification of exception control flows and handlers based on architectural scenarios. In *11th IEEE High Assurance Systems Engineering Symposium*, pages 177–186, Nanjing, China, December 2008. IEEE Computer Society.

[37] Patrick Henrique da Silva Brito, Paulo Asterio de Castro Guerra, and Cecília Mary Fischer Rubira. A study of architectural styles for component-based software systems (in portuguese). Technical Report IC-07-10, Institute of Computing, University of Campinas, March 2007.

[38] Moacir C. da Silva Jr, Paulo A. de C. Guerra, and Cecília M. F. Rubira. A java component model for evolving software systems. In *Proc. of the 18th IEEE International Conference on Automated Software Engineering (ASE'03)*, volume 0, page 327, 2003.

[39] Bruno T. de Abreu, Eliane Martins, and Fabiano L. de Sousa. Generalized extremal optimization: an attractive alternative for test data generation. In *Proc. of the 9th annual conference on Genetic and evolutionary computation (GECCO '07)*, pages 1138–1138, New York, NY, USA, 2007.

[40] Ana Elisa de Campos Lobo, Patrick Henrique da Silva Brito, and Cecília Mary Fischer Rubira. A systematic approach for architectural design of component-based product lines. Technical Report IC-07-33, Institute of Computing, University of Campinas, November 2007.

[41] P. A. de Castro Guerra, Cecilia Mary Fischer Rubira, and R. de Lemos. A fault-tolerant software architecture for component-based systems. In *Architecting Dependable Systems. Lecture Notes in Computer Science*, LNCS 2677, pages 129–149. Springer, Berlin, Germany, 2003.

[42] Rogério de Lemos. Architectural fault tolerance using exception handling. In *Architecting Dependable Systems IV, LNCS 4615*, pages 142–162, 2007.

[43] Rogério de Lemos. Architectural fault tolerance using exception handling. In Rogerio de Lemos, Cristina Gacek, and Alexander Romanovsky, editors, *Architecting Dependable Systems IV, LNCS 4615*, pages 142–162, 2007.

[44] Rogerio de Lemos, Paulo Asterio de Castro Guerra, and Cecilia Mary Fischer Rubira. A fault-tolerant architectural approach for dependable systems. *IEEE Software*, 23(2):80–87, 2006.

[45] Marcelo de O. Dias, Leonardo P. Tizzei, Patrick H. S. Brito, and Cecília M. F. Rubira. Design and implementation of architectural variabilities based on cosmos component model using aspects (extended abstract). In *Workshop of the European Group of Aspect-Oriented Software Development (AOSD-Europe 2008)*, pages 1–4, 2008.

[46] Mark Denford, Andrew Solomon, John Leaney, and Tim O'Neill. Architectural abstraction as transformation of poset labelled graphs. *Journal of Universal Computer Science (J.UCS)*, 10(10):1408–1428, 2004.

[47] John DeVale, Philip Koopman, and David Guttendorf. The ballista software robustness testing service. In *Proc. of the 16th International Conference and Exposition on Testing Computer Software (TCS'99)*, Washington DC, 1000.

[48] Hoda Fahmy and Richard C. Holt. Software architecture transformations. In *Proc. of the International Conference on Software Maintenance (ICSM'00)*, pages 88–96, 2000.

[49] Stefan Ferber, Jürgen Haag, and Juha Savolainen. Feature interaction and dependencies: Modeling features for reengineering a legacy product line. In *Proc. of the Second International Software Product Lines Conference (SPLC), LNCS 2379*, pages 37–60, 2002.

[50] Jean-Claude Fernandez, Laurent Mounier, and Cyril Pachon. A model-based approach for robustness testing. In *Proc. of the 17th International Conference on Testing Communicating Systems, LNCS 3502*, pages 333–348, 2005.

[51] Eduardo Figueiredo et al. Evolving software product lines with aspects: an empirical study on design stability. In *ICSE '08: Proc. of the 30th international conference on Software engineering*, pages 261–270, 2008.

[52] Fernando Castor Filho, Patrick Henrique da S. Brito, and Cecília Mary F. Rubira. Reasoning about exception flow at the architectural level. Technical Report IC-06-10, Institute of Computing, University of Campinas, May 2006.

[53] Fernando Castor Filho, Patrick Henrique da S. Brito, and Cecília Mary F. Rubira. Specification of exception flow in software architectures. Technical Report IC-06-09, Institute of Computing, University of Campinas, May 2006.

[54] Xiang Fu, Tevfik Bultan, and Jianwen Su. Formal verification of e-services and workflows. In *Proc. of the International Workshop on Web Services, E-Business, and the Semantic Web (WES'02)*, pages 188–202, London, UK, 2002. Springer-Verlag.

[55] Leonel Aguilar Gayard, Cecília Mary Fischer Rubira, and Paulo Astério de Castro Guerra. COSMOS*: a COmponent System MOdel for Software Architectures. Technical Report IC-08-04, February 2008.

[56] Michael Gillmann, Jeanine Weissenfels, Gerhard Weikum, and Achim Kraiss. Performance and availability assessment for the configuration of distributed workflow management systems. In *Proc. of the 7th International Conference on Extending*

Database Technology (EDBT '00), pages 183–201, London, UK, 2000. Springer-Verlag.

[57] John B. Goodenough. Exceptional handling: Issues and a proposed notation. *CACM*, 18(12), 1975.

[58] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, 1993.

[59] Ahmed Helmy, Deborah Estrin, and Sandeep K. S. Gupta. Fault-oriented test generation for multicast routing protocol design. In *Proc. of the Joint International Conference on Formal Description Techniques for Distributed Systems and Communication Protocols (FORTE XI) and Protocol Specification, Testing and Verification (PSTV XVIII)*, pages 93–109, Deventer, The Netherlands, The Netherlands, 1998. Kluwer, B.V.

[60] Mei-Chen Hsueh, Timothy K. Tsai, and Ravishankar K. Iyer. Fault injection techniques and tools. *IEEE Computer*, 30(4):75–82, 1997.

[61] Fares Saad Khorchef, I. Berrada, Antoine Rollet, and R. Castanet. Automated robustness testing for reactive systems : application to communicating systems. In *6th International Workshop on Innovative Internet Community Systems (I2CS06)*, Neuchatel, Switzerland, June 2006.

[62] K. H. Kim and Howard O. Welch. Distributed execution of recovery blocks: An approach for uniform treatment of hardware and software faults in real-time applications. *IEEE Trans. on Compututers*, 38(5):626–636, 1989.

[63] M. Klein, R. Kazman, L. Bass, J. Carriere, M. Barbacci, and H. Lipson. Attribute-based architecture styles. In *Software Architecture, Proceedings of the First Working IFIP Conference on Software Architecture (WICSA)*, pages 225–243. Kluwer Academic Publishers, 1999.

[64] Philip Koopman, John Sung, Christopher Dingman, Daniel Siewiorek, and Ted Marz. Comparing operating systems using robustness benchmarks. In *Proc. of the 16th Symposium on Reliable Distributed Systems (SRDS '97)*, page 72, Washington, DC, USA, 1997.

[65] N. P. Kropp, P. J. Koopman, and D. P. Siewiorek. Automated robustness testing of off-the-shelf software components. In *Proc. of the 28th International Symposium on Fault-Tolerant Computing (FTCS'28)*, page 230, Washington, DC, USA, 1998. IEEE Computer Society.

[66] Philippe Kruchten, J. Henk Obbink, and Judith A. Stafford. The past, present, and future for software architecture. *IEEE Software*, 23(2):22–30, 2006.

[67] Serge Lacourte. Exceptions in guide, an object-oriented language for distributed applications. In *ECOOP '91: Proceedings of the European Conference on Object-Oriented Programming*, pages 268–287, London, UK, 1991. Springer-Verlag.

[68] Jean-Claude Laprie, Jean Arlat, Christian Béounes, and Karama Kanoun. Definition and analysis of hardware- and software-fault-tolerant architectures. *IEEE Computer*, 23(7):39–51, 1990.

[69] Peter A. Lee and Thomas Anderson. *Fault Tolerance: Principles and Practice*. Dependable computing and fault-tolerant systems. Springer-Verlag, Berlin, 2nd edition, 1990.

[70] M. Leuschel and Michael J. Butler. Prob: A model checker for b. In *Proc. of Interfational Conference on Formal Methods (FME'2003), LNCS 2805*, pages 855–874. Pisa, Italy, 2004.

[71] G. M. P. S. Lima and G. H. Travassos. Integration testing applied to object-oriented software: heuristics for class ordering. Technical Report ES-632/04, COPPE/UFRJ, 2004. (in Portuguese).

[72] Ana Elisa C. Lobo, Patrick H. da S. Brito, and Cecília M. F. Rubira. Managing Variabilities in Component-Based Software Product Lines. In *Proc. of the VI Workshop de Desenvolvimento Baseado em Componentes (WDBC)*, Recife, Brazil, 2006.

[73] Robyn R. Lutz and Gerald C. Gannod. Analysis of a software product line architecture: an experience report. *Journal of Systems and Software (JSS)*, 66(3):253–267, 2003.

[74] Eliane Martins, Cecilia M. F. Rubira, and Nelson G. M. Leme. Jaca: A reflective fault injection tool based on patterns. In *Proc. of the International Performance and Dependability Symposium (IPDS 2002)*, pages 483–487, 2002.

[75] Zoltán Micskei, István Majzik, and Francis Tam. Comparing robustness of ais-based middleware implementations. In *Proc. of the International Service Availability Symposium, LNCS 4526*, May 2007.

[76] Regina Moraes and Eliane Martins. Fault injection approach based on architectural dependencies. In *Architecting Dependable Systems III, LNCS 3549*, pages 300–321, 2005.

[77] Regina L. O. Moraes, João Durães, R. Barbosa, Eliane Martins, and Henrique Madeira. Experimental risk assessment and comparison using software fault injection. In *Proc. of the 37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN'2007)*, pages 512–521, Edinburgh, UK, 2007.

[78] Mark Moriconi and Robert Riemenschneider. Introduction to sadl 1.0 a language for specifying software architecture hierarchies. Technical Report SRI-CSL-97-01, SRI International, March 1997.

[79] Tiago Cesar Moronte. A software infrastructure to support component based software architecture construction (in portuguese). Master's thesis, IC, Unicamp, Abril 2007.

[80] Henry Muccini, Antonia Bertolino, and Paola Inverardi. Using software architecture for code testing. *IEEE Trans. Softw. Eng.*, 30(3):160–171, 2004.

[81] Flavio Oquendo. π-arl: an architecture refinement language for formally modelling the stepwise refinement of software architectures. *SIGSOFT Softw. Eng. Notes*, 29(5):1–20, 2004.

[82] David Lorge Parnas and Harald Würges. Response to undesired events in software systems. In *Proceedings of the 2nd International Conference on Software Engineering*, pages 437–446, San Francisco, USA, October 1976.

[83] Raymond A. Paul, W. T. Tsai, and John S. Mikell. Rapid simulation evaluation from scenario specifications for command and control systems. Technical Report A947464, Department of Defense - Washington DC, June 2004.

[84] Ivan Perez, Eliane Martins, and Julio Viégas. Using uml models for component test (in portuguese). In *Proc. of the 8th Brazilian Workshop on Test and Fault Tolerance (WTF 2007)*, pages 99–112, Belém, Brazil, 2007.

[85] Michael Poppleton. Towards feature-oriented specification and development with event-b. In Peter Sawyer, Barbara Paech, and Patrick Heymans, editors, *13th International Working Conference on Requirements Engineering: Foundation for Software Quality (REFSQ 07), LNCS 4542*, pages 367–381, 2007.

[86] B. Randell. System structure for software fault tolerance. In *Proc. of the International Conference on Reliable software*, pages 437–449, 1975.

[87] Brian Randell. Turing memorial lecture facing up to faults. *Comput. J.*, 43(2):95–106, 2000.

[88] D. Reimer and H. Srinivasan. Analyzing exception usage in large java applications. In *Proceedings of ECOOP'2003 Workshop on Exception Handling in Object-Oriented Systems*, July 2003.

[89] Debra J. Richardson and Alexander L. Wolf. Software testing at the architectural level. In *International workshop on multiple perspectives in software development (Viewpoints '96) on SIGSOFT '96 workshops*, pages 68–71, New York, NY, USA, 1996. ACM.

[90] Camila Ribeiro Rocha and Eliane Martins. A strategy to improve component testability without source code. In *Proceedings of TECOS 2004 (Workshop Testing Component-Based Systems)*, pages 47–62, Erfurt, Germany, 2004.

[91] Cecília Mary F. Rubira, Rogério de Lemos, Gisele Ferreira, and Fernando Castor Filho. Exception handling in the development of dependable component-based systems. *Software – Practice and Experience*, 35(5):195–236, March 2005.

[92] Carl F. Schaefer and Gary N. Bundy. Static analysis of exception handling in ada. *Softw. Pract. Exper.*, 23(10):1157–1174, 1993.

[93] Fred B. Schneider. Byzantine generals in action: implementing fail-stop processors. *ACM Transactions on Computer Systems*, 2(2):145–154, 1984.

[94] R.K. Scott, J.W. Gault, and D.F. McAllister. The consensus recovery block. In *Proc. of Total Systems Reliability Symposium*, pages 74–85, December 1983.

[95] Keng Siau and Terry A. Halpin, editors. *Unified Modeling Language: Systems Analysis, Design and Development Issues*. Idea Group, 2001.

[96] Kevin Simons and Judith Stafford. Cmeh: Container-managed exception handling for increased assembly robustness. In *Proceedings of the 7th International Symposium on Component-Based Software Engineering*, LNCS 3054, pages 122–129, May 2004.

[97] Morris Sloman and Jeff Kramer. *Distributed systems and computer networks*. Prentice Hall International (UK) Ltd., Hertfordshire, UK, UK, 1987.

[98] Ian Sommerville. *Software Engineering*. Addison-Wesley, 6th edition, 1995.

[99] Judith A. Stafford and Alexander L. Wolf. Architecture-level dependence analysis for software systems. *International Journal of Software Engineering and Knowledge Engineering*, 11(4):431–451, 2001.

[100] Clemens Szyperski. *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

[101] R. N. Taylor, N. Medvidovic, K.M. Anderson, Jr. E. J. Whitehead, and J.E. Robbins. A component- and message- based architectural style for GUI software. In *Proc. of the 17th International Conference on Software Engineering*, pages 295–304, April 1995.

[102] Dave Thomas and Brian M. Barry. Model driven development: the case for domain oriented programming. In *Companion of the 18th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications (OOPSLA '03)*, pages 2–7, New York, NY, USA, 2003.

[103] Rodrigo Teruo Tomita, Fernando Castor Filho, Paulo Asterio de C. Guerra, and Cecília Mary F. Rubira. Bellatrix: An environment with arquitectural support for component-based development (in portuguese). In *Proc. of the IV Brazilian Workshop on Component-Based Development*, pages 43–48, 2004.

[104] Weilei Zhang and Barbara G. Ryder. Discovering accurate interclass test dependences. In *PASTE '07: Proceedings of the 7th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, pages 55–62, New York, NY, USA, 2007. ACM.